nirva

Nirva user's guide

Document version: 4.20

Table of Contents

Overview 36
What is Nirva? 36
Application server 36
Integration services 37
Orchestration 38
Who is Nirva for? 39
Projects 39
Products 39
How Nirva works? 40
All-in-One concept 40
Data centric architecture 41
Main components 41
Key features 42
Multiple languages 44
Distributed Architecture 44
Client connector load balancing 45
Web front end load balancing 45
Nirva to Nirva load balancing 46
Nirva to Nirva with failover 46
Message bus 46
Standards 47
Extensible 47
Component reusability 48
Upwards and downwards integration 49
Message bus 49
Session management 50
Security 50
Web Services 51
Scheduler 52
Listeners 53
Presentation layer 53
Html renderers 54
Nidget framework 55
Workflow 56
Event-driven capabilities 57
Database 58
Mass storage 58

Registry ... 59
Configuration ... 59
Monitoring ... 60
Logs ... 60
Multi-application ... 60
Deployment ... 60
Multi platform ... 61
Development tools ... 61
Nirva components ... 63
Server side ... 63
Server (nvs) ... 63
Services ... 63
Applications ... 63
Web services ... 64
Procedures ... 64
External programs ... 64
Client side ... 64
Client library (nvc) ... 64
Command line tool (nvcc) ... 65
Debug tool (nvd) ... 65
Connectors ... 65
Applications ... 65
Configuration ... 66
Features ... 67
Object processing ... 67
Preparing objects ... 67
Connecting session ... 68
Sending objects ... 68
Processing objects ... 68
Getting results ... 68
Disconnecting session ... 68
Objects ... 68
Boolean ... 69
Integer ... 69
String ... 69
String list ... 69
Indexed string list ... 69
Table ... 69
File ... 70
Binary ... 70
Containers ... 70
Sessions ... 70
Applications ... 72
Services ... 73
External ... 73
System ... 73
Local ... 73

- Procedures 74
- Web services 75
- Nidget framework 75
- Renderers 75
- Transactions 75
- Registries 76
- Locking 77
- Semaphores 77
- Variables 77
- Direct access from browser 78
  - XML applications 78
  - Web sites 79
  - Configuration 80
- On line documentation 81
- Logs 81
- Crash condition 82
- Cluster and load balancing 82
- Scheduler 82
- Listeners 83
- Triggers 83
- Mail 83
- Licensing 84
- Security 84
- Unicode 84
- MQ connector 84
- Debug 85

Installation 86
- Download Nirva 86
- Installing NIRVA server 86
- Windows installation 86
- Unix installation 87
- Uninstall 88
- License 88

Getting started 91
- Starting the server 91
- Creating the HELLO application 92
- Testing from browser 97
- Testing from batch 99

Directory structure 101
- Main directory 101
- Applications directory 102
- Services directory 104
- Webservices directory 105

Using Nirva 107
- Starting and stopping Nirva server 107
  - Console mode 107
  - Service mode 107

Restoring default parameters 108
Disabling JAVA VM 108
Disabling MQ connector 108
Encoding 108
Verbose mode 109
Cluster mode 109
Time measurement mode 109
Open doors 109
A Nirva command 110
Using NIRVA with apache 112
Reverse proxy 112
Load balancing 113
Setting HTTPS protocol 115
Creating a private key and a certificate 115
Installing the certificate 116
Configuring NIRVA 116
Testing the HTTPS server 117
Using client certificates 117
Creating certificate requests 117
Signing certificate requests 118
Creating your own self-signed certificate 118
Signing certificate requests yourself 119
Importing client certificates on the user's computer 120
Installing the CA certificate on the Nirva server 121
Testing the client certificate 121
Configuration 122
Starting the configuration 122
Password 123
System 124
Information 125
Parameters 125
General parameters 126
HTTP parameters 126
Debug 128
Sessions 128
Compatibility 129
Security 129
Time 130
XSLT engine 130
Home directories 131
JAVA VM parameters 132
Transaction server parameters 133
Scheduler parameters 134
Default application parameters 135
Mail server parameters 136
Web services parameters 137
Mime types 137

Aliases ... 138
Virtual hosts ... 138
Sessions ... 139
Services ... 141
Service documentation ... 142
Service configuration ... 142
Starting and stopping a service ... 142
Sessions used by a service ... 143
Detail service information ... 143
Working with service logs ... 145
Installing a service package ... 145
Creating a new service ... 146
Mounting a service ... 147
Accessing service registry ... 148
Service tests ... 149
Applications ... 150
Application start page ... 151
Application documentation ... 151
Starting and stopping an application ... 152
Sessions used by an application ... 152
Detail application information ... 152
Installing an application package ... 154
Creating a new application ... 155
Web services ... 156
View detail web service information ... 157
Creating a new web service ... 158
Starting and stopping a web service ... 159
Display web service content ... 159
Edit web service content ... 161
View web service WSDL data ... 166
Installing a web service package ... 167
Queues ... 168
Creating a new queue ... 169
Starting and stopping a queue ... 170
Editing queue parameters ... 170
Removing a queue ... 171
Licenses ... 171
Documentation ... 172
Installation ... 173
Security ... 174
Setting global parameters ... 175
Adding a new user ... 175
Changing user information ... 176
Removing a user ... 177
Enabling or disabling a user ... 177
Setting user roles ... 177
Viewing user permissions ... 178

Displaying roles 178
Adding a new role..... 178
Changing role information 179
Removing a role 179
Setting role inherits..... 179
Setting role permissions 180
Registry..... 181
Locking 181
Semaphores 182
Threads..... 183
Logs 184
Creating a new log 184
Removing a log 185
Changing log parameters 185
Erasing log data 186
Displaying log data 187
Exporting log data 190
Application 190
Information..... 191
Parameters 191
Documentation 192
Configuration 193
Sessions 193
Transactions 194
Removing a transaction 195
Rolling back a transaction 195
Restarting a transaction 196
Validating a transaction 196
Detail of a transaction..... 196
Tasks 197
Creating a new task..... 198
Removing a task..... 198
Running a task 198
Enabling or disabling a task 199
Setting task parameters 199
Listeners 202
Creating a new listener..... 203
Removing a listener..... 205
Staring or stopping a listener 205
Setting listener parameters 206
Viewing listener parameters..... 206
Security..... 206
Setting global parameters 207
Adding a new user..... 208
Changing user information 208
Removing a user 209
Enabling or disabling a user..... 209

Setting user roles 210
Viewing user permissions 210
Displaying roles 210
Adding a new role 211
Changing role information 211
Removing a role 212
Setting role inherits 212
Setting role permissions 213
Registry 213
Debug 214
Locking 214
Semaphores 215
Logs 216
Tests 216
The Nirva command syntax 218
Overview 218
Standard syntax 218
Web browser syntax 219
Syntax 219
Information on HTTP methods 220
XML connector syntax 221
SOAP connector syntax 221
Web service connector syntax 221
Standard variables 223
Parameter reference 224
NV_APPLICATION 227
NV_CLASS 227
NV_CLOSE_SESSION 228
NV_CMD 228
NV_COMMAND 229
NV_CONTAINER 229
NV_CONTAINER_CLEAR 230
NV_DEBUG_ONLY 230
NV_DEBUG_PARAM 230
NV_ERROR_PROC 231
NV_FAST 231
NV_FORM_ENCODING 231
NV_HREND 232
NV_HREND_PAGE 232
NV_IDENT 232
NV_IN_CONTAINER 232
NV_IN_CONTAINER_CLEAR 233
NV_INPUT_TYPE 234
NV_KEEP_ERROR 234
NV_KEEP_VAR 234
NV_LANGUAGE 235
NV_LOCK_SESSION 235

NV_MQ_MSG_HEADER 236
NV_MQ_OBJNAME 236
NV_MQ_OUTPUT 236
NV_NEW_IF_EXPIRED 236
NV_NEW_PASSWORD 237
NV_NEW_PASSWORD_CONFIRM 237
NV_NO_CONVERSION 237
NV_NO_ERROR 237
NV_NO_REDIRECT 238
NV_OUT_CONTAINER 238
NV_OUT_CONTAINER_CLEAR 239
NV_OUTPUT_TYPE 239
NV_PARAM 240
NV_PASSWORD 240
NV_POST_PROC 240
NV_PROC 241
NV_REQUEST 241
NV_REVERSE_CONTAINER 241
NV_SERVICE 242
NV_SESSION_CLOSE 242
NV_SESSION_ID 242
NV_SESSION_NAME 243
NV_SESSION_OPEN 243
NV_STOP_ON_ERROR 244
NV_TIMEOUT 244
NV_TOKEN 244
NV_URL_ENCODING 245
NV_USER 245
NV_VAR 245
NV_VAR_IDENT 246
NV_XML_CONTAINER 246
NV_XML_DEBUG 247
NV_XML_ENCODING 247
NV_XML_HTTP_HEADERS 247
NV_XML_MODEL 248
NV_XML_NSPREFIX 248
NV_XML_NSREF 248
NV_XML_OBJECTS 249
NV_XML_OUTPUT 249
NV_XML_SESSION_INFO 249
NV_XML_SIMPLE 250
NV_XML_STRICT 250
NV_XML_SUBCONTAINERS 251
NV_XML_VARIABLES 251
NV_XML_WITH_DATA 251
NV_XML_XSL 252
NV_XML_XSL_ERROR 252

NV_XML_XSL_IN 253
Error management 254
Applications 256
Overview 256
Managing applications 256
Hello world application 256
Application directory 256
Description file 258
INFO section 259
SETTINGS section 259
PERMISSIONS section 260
ORDER_URLS section 260
REST_URLS section 261
Procedures 263
Overview 263
Calling a procedure 263
Scope 264
Application procedure 264
System procedure 264
Service procedure 264
Web service procedure 264
Programming language 265
Native procedures 265
Perl procedures 265
Java procedures 265
Dotnet procedures 266
Procedure parameters 266
Examples 266
Native procedures 267
Perl procedures 268
Description 268
Reference 269
NV::SetErrorMode 270
NV::SetError 270
NV::SetErrorEx 271
NV::Command 272
NV::GetSessionId 272
NV::GetClass 273
NV::GetService 273
NV::GetCommand 274
NV::GetInContainer 274
NV::GetOutContainer 275
NV::GetLanguage 275
NV::GetError 276
NV::GetNumParameters 276
NV::GetSource 277
NV::ParameterExist 277

NV::GetParameter....278
NV::GetParameterName....278
NV::SetEncoding....279
Java procedures....280
Description....280
Reference....281
SetError....281
SetErrorEx....282
Command....282
GetResult....283
GetSessionId....284
GetService....284
GetClass....285
GetCommand....285
GetInContainer....285
GetOutContainer....286
GetLanguage....286
GetError....287
GetNumParameters....287
GetSource....288
ParameterExist....288
GetParameter....289
GetParameterName....289
Dotnet procedures....290
Description....290
Reference....291
SetError....292
SetErrorEx....292
Command....293
GetResult....293
GetSessionId....294
GetService....294
GetClass....295
GetCommand....295
GetInContainer....296
GetOutContainer....296
GetLanguage....296
GetError....297
GetNumParameters....297
GetSource....298
ParameterExist....298
GetParameter....299
GetParameterName....299
Connectors....301
Dll - C and C++....301
Overview....301
Installation....302

Reference 302
Example 302
ActiveX 303
Overview 303
Installation 303
Reference 304
OpenRequest 304
CloseRequest 307
Command 308
Example 308
Php 309
Overview 309
Installation 309
Reference 310
nvc_openrequest 310
nvc_closerequest 313
nvc_command 314
Example 314
Cold Fusion 315
Overview 315
Installation 315
Reference 318
OpenRequest 319
Command 321
CloseRequest 322
Example 322
Java 323
Overview 323
Installation 323
Reference 324
nvcj 324
Command 327
GetResult 328
Example 328
Dotnet 329
Overview 329
Installation 329
Reference 329
nvcdn 330
Command 333
GetResult 334
Example 334
Perl 335
Overview 335
Installation 335
Reference 336
NVC::OpenRequest 336

NVC::CloseRequest 339
NVC::Command 340
Example 340
Xml 341
Overview 341
Installation 341
Reference 341
Syntax 342
Input XML data 342
Command parameters 342
Output XML data 343
Error management 344
Example 345
Soap 346
Overview 346
Installation 346
Reference 346
Syntax 347
Input SOAP data 347
Command parameters 347
Output SOAP data 349
Error management 350
Example 351
Web service 352
Overview 352
Installation 352
Reference 352
Syntax 352
Input SOAP data 353
Command parameters 353
Output SOAP data 354
Error management 354
Get WSDL data 354
Test 355
IBM WebSphere MQ 355
Overview 355
Installation 356
Linux 356
AIX 357
Solaris 358
Hpux pa-risc 358
Hpux itanium 358
Windows 358
Configuration 358
Reference 359
Message type 359
Filter 359

Message content ... 359
Virtual printer ... 363
Flex ... 363
Overview ... 363
Installation ... 363
Reference ... 364
Object declaration ... 364
Methods ... 365
Error management ... 365
Method Reference ... 366
OpenSession ... 366
CloseSession ... 367
Command ... 368
GetObjectUrl ... 369
GetUrl ... 370
Ajax ... 371
Overview ... 371
Installation ... 372
Reference ... 372
NvAjax.Request ... 373
Command ... 374
XMLCommand ... 375
AsyncXMLCommand ... 376
Rest ... 377
Overview ... 377
Installation ... 378
Reference ... 378
Urls ... 378
HTTP Methods ... 379
Input and output type ... 379
Procedure ... 379
Form processing ... 380
Login ... 380
Session persistance ... 381
Permission ... 381
Encoding ... 381
Error management ... 382
Example ... 382
Tools ... 387
nvcc ... 387
Command line syntax ... 387
Parameters ... 387
Options ... 387
Input file format ... 390
Example ... 391
Test mode ... 392
nvcc command files ... 393

system_package.txt 393
system_install.txt 394
service_package.txt 394
service_install.txt 394
application_package.txt 395
application_install.txt 395
webservice_package.txt 395
webservice_install.txt 396
getmachine.txt 396
license_install.txt 396
testxml.txt 397
testsoap.txt 397
testwebs.txt 397
nvl 398
Overview 398
Installation 398
Using nvl 398
nvd 400
Command line syntax 400
Examples 401
Services 403
What is a Nirva service 403
Requirements 404
Library 404
Multithreading 404
Compilation 404
Entry point 405
Names 405
Description file 405
INFO section 407
SETTINGS section 408
PERMISSIONS section 408
COMMAND_CLASSES section 408
COMMAND_CLASS sections 409
ERROR_CLASSES section 409
ERROR_CLASS sections 409
ERROR_LANGUAGES section 409
Documentation 410
Configuration 410
Installation 411
Service package 411
Installing a service 411
Licensing 412
License policy 412
Installing service licenses 412
Automatic skeleton 413
C++ 415

Without session management 415
With session management 418
Java 423
Without session management 423
With session management 425
Dotnet 429
Without session management 429
With session management 430
Tutorial 434
C++ service 435
Creating the skeleton 435
Compiling the service 436
Mounting the service 437
Starting the service 438
Testing the service 439
Debugging the service 440
Adding service functionality 442
Adding error codes 444
Creating license 446
Checking license 449
Documentation 449
Configuration 450
Packaging the service 451
Installing the service 452
Java service 453
Creating the skeleton 453
Compiling the service 454
Mounting the service 454
Starting the service 455
Testing the service 456
Adding service functionality 457
Adding error codes 459
Creating license 461
Checking license 464
Documentation 464
Configuration 465
Packaging the service 466
Installing the service 467
Dotnet service 468
Creating the skeleton 468
Compiling the service 469
Mounting the service 469
Starting the service 470
Testing the service 471
Adding service functionality 472
Adding error codes 474
Creating license 476

Checking license ... 479
Documentation ... 479
Configuration ... 480
Packaging the service ... 481
Installing the service ... 482
Interface reference ... 483
C++ ... 483
Library entry point ... 483
Library initialization and cleaning ... 484
Session management ... 484
NvServiceCommand class ... 485
SetError ... 486
Command ... 487
GetSessionId ... 488
GetClass ... 488
GetCommand ... 489
GetInContainer ... 489
GetOutContainer ... 490
GetLanguage ... 490
GetMachineName ... 491
GetMachineUser ... 492
GetError ... 492
GetNumParameters ... 493
GetParameterName ... 493
GetSource ... 494
IsCommand ... 494
ParameterExist ... 495
GetParameter ... 495
AddHttpHeader ... 496
SetServiceSession ... 497
GetServiceSession ... 497
Java ... 497
Command ... 498
Class initialization and cleaning ... 498
Session management ... 499
nvcmd class ... 500
SetError ... 500
Command ... 501
GetResult ... 502
GetSessionId ... 502
GetClass ... 502
GetCommand ... 503
GetInContainer ... 503
GetOutContainer ... 504
GetLanguage ... 504
GetMachineName ... 505
GetMachineUser ... 505

GetError 505
GetNumParameters 506
GetParameterName 506
GetSource 507
IsCommand 507
ParameterExist 508
GetParameter 508
AddHttpHeader 509
Dotnet 509
Class declaration 510
Command 510
Class initialization and cleaning 510
Session management 511
nvcmd class 512
SetError 513
Command 513
GetResult 514
GetSessionId 514
GetClass 515
GetCommand 515
GetInContainer 516
GetOutContainer 516
GetLanguage 516
GetMachineName 517
GetMachineUser 517
GetError 518
GetNumParameters 518
GetParameterName 519
GetSource 519
IsCommand 520
ParameterExist 520
GetParameter 521
AddHttpHeader 521
Renderers 522
Building a renderer 523
Create a service 524
Implement the render function 524
C++ 524
Java 525
Dotnet 526
Modify the service description file 528
Nidgets framework 529
Overview 529
Getting started 530
Hello world 530
How it works 531
Dynamic code 534

More examples 535
Detailed concepts 536
Directory structure 536
Configuration file 537
info tag 538
debug tag 538
error tag 538
headers and footers tags 538
includes tag 538
icon tag 539
theme tag 539
nidget_packages tag 539
views tag 539
Views 539
info tag 540
type tag 540
language tag 541
title tag 541
class tag 541
header and footer tags 541
metas tag 541
parameters tag 542
labels tag 542
docs tag 542
forms tag 543
content tag 543
sections tag 543
Nidgets 545
Nidget code 545
Parameters 548
Label files 548
Standard nidgets library 549
Framework functions 549
NvFmAsyncDocCommand 550
NvFmCommand 550
NvFmCommandGetResult 551
NvFmDisplayNirvaError 551
NvFmDocCommand 551
NvFmFileUpload 552
NvFmGetApplication 552
NvFmGetDebug 553
NvFmGetFileObject 553
NvFmGetFileObjectUrl 554
NvFmGetLanguage 554
NvFmGetLastError 555
NvFmGetLoginViewUrl 555
NvFmGetNidgetObject 555

NvFmGetSessionId 556
NvFmGetView 556
NvFmGetViewUrl 557
NvFmHideNidget 557
NvFmHideSection 557
NvFmHtmlEntities 558
NvFmLogout 558
NvFmSetErrorHandler 559
NvFmSetExitHandler 559
NvFmSetInitHandler 559
NvFmShowNidget 560
NvFmShowSection 560
NvFmWriteDebug 560
NvFmWriteDebugLn 561
innerXHTML 561
Web services 562
What is a web service 562
NIRVA implementation of web services 563
Web service example 563
Creating the web service 564
Editing the web service 565
Starting the web service 571
Running the web service 572
With nvcc tool 572
With SoapUI standard tool 575
Deploying the web service 576
Description file 578
INFO section 579
SETTINGS section 579
Client library nvc 580
How to use 580
Request 580
Session ID 581
Network connections 581
Local container 581
Local service 581
Command buffer 581
Sending a command 582
Sending a command to the command buffer 582
Sending a command directly to the server 582
Sending and retrieving objects 582
Variables 582
Error management 583
Global error 583
Command error 583
Example 584
Function reference 587

Overview ... 587
Functions ... 588
NvOpenRequest ... 588
NvCloseRequest ... 591
NvCommand ... 591
NvExitLib ... 593
Local service reference ... 594
Overview ... 594
Output buffer ... 594
Classes ... 594
Error codes ... 595
OBJECT Class ... 595
REQUEST Class ... 596
Commands ... 597
OBJECT class ... 597
CLEAR_ALL ... 597
COPY ... 597
CREATE ... 598
EXIST ... 600
GET_NAME ... 601
GET_NUM ... 601
GET_TYPE ... 602
REMOVE ... 602
SET_NAME ... 602
BOOLEAN_GET_VALUE ... 603
BOOLEAN_SET_VALUE ... 603
INTEGER_GET_VALUE ... 604
INTEGER_SET_VALUE ... 604
STRING_GET_VALUE ... 604
STRING_SET_VALUE ... 605
STRINGLIST_ADD_STRINGLIST ... 605
STRINGLIST_GET_SIZE ... 605
STRINGLIST_GET_VALUE ... 606
STRINGLIST_GET_VALUES ... 606
STRINGLIST_INSERT ... 607
STRINGLIST_REMOVE ... 607
STRINGLIST_SEARCH ... 608
STRINGLIST_SET_VALUE ... 608
STRINGLIST_SORT ... 609
STRINGLIST_SWAP ... 610
INDSTRINGLIST_ADD_INDSTRINGLIST ... 610
INDSTRINGLIST_GET_KEY ... 610
INDSTRINGLIST_GET_SIZE ... 611
INDSTRINGLIST_GET_VALUE ... 611
INDSTRINGLIST_GET_VALUES ... 612
INDSTRINGLIST_KEY_EXIST ... 612
INDSTRINGLIST_REMOVE ... 613

INDSTRINGLIST_SET_VALUE ... 613
TABLE_ADD_COLUMNS ... 614
TABLE_ADD_ROWS ... 615
TABLE_ADD_TABLE ... 615
TABLE_CLEAR ... 616
TABLE_CLEAR_CELL ... 616
TABLE_CLEAR_COLUMN ... 617
TABLE_CLEAR_DATA ... 617
TABLE_CLEAR_ROW ... 618
TABLE_CLEAR_ROWS ... 618
TABLE_CREATE_FROM ... 619
TABLE_EXPORT ... 619
TABLE_FILTER_COLUMNS ... 620
TABLE_GET_CELL ... 621
TABLE_GET_CELL_LINE ... 621
TABLE_GET_CELL_NUM_LINES ... 622
TABLE_GET_COLUMN ... 622
TABLE_GET_COLUMN_DESCRIPTION ... 623
TABLE_GET_COLUMN_INDEX ... 623
TABLE_GET_COLUMN_NAME ... 624
TABLE_GET_COLUMN_NAMES ... 624
TABLE_GET_DESCRIPTION ... 624
TABLE_GET_NUM_COLUMNS ... 625
TABLE_GET_NUM_ROWS ... 625
TABLE_GET_PRIMARY_INFO ... 625
TABLE_GET_ROW ... 626
TABLE_GET_ROWS ... 627
TABLE_GET_ROW_INDEX ... 628
TABLE_IMPORT ... 629
TABLE_INSERT_CELL_LINE ... 630
TABLE_INSERT_COLUMN ... 631
TABLE_INSERT_ROWS ... 631
TABLE_IS_COLUMN_NUMERIC ... 632
TABLE_JOIN ... 633
TABLE_LOAD ... 633
TABLE_MODIFY_COLUMN ... 634
TABLE_REMOVE_CELL_LINE ... 635
TABLE_REMOVE_COLUMN ... 635
TABLE_REMOVE_ROW ... 636
TABLE_REMOVE_ROWS ... 636
TABLE_SAVE ... 637
TABLE_SEARCH ... 637
TABLE_SELECT_FROM ... 638
TABLE_SET_CELL ... 639
TABLE_SET_CELL_LINE ... 640
TABLE_SET_COLUMN ... 641
TABLE_SET_COLUMN_DESCRIPTION ... 641

TABLE_SET_COLUMN_NAME ... 642
TABLE_SET_COLUMN_TYPE ... 642
TABLE_SET_DESCRIPTION ... 643
TABLE_SET_PRIMARY_COLUMN ... 643
TABLE_SET_ROW ... 644
TABLE_SORT ... 644
TABLE_SORT_CELL ... 645
TABLE_SWAP_CELL_LINES ... 646
TABLE_SWAP_COLUMNS ... 646
TABLE_SWAP_ROWS ... 647
FILE_APPEND ... 647
FILE_CLEAR ... 648
FILE_COMPRESS ... 648
FILE_CREATE ... 649
FILE_DECOMPRESS ... 649
FILE_EXIST ... 649
FILE_GET_DIRNAME ... 650
FILE_GET_EXTENSION ... 650
FILE_GET_FILENAME ... 650
FILE_GET_PATHNAME ... 651
FILE_GET_SIZE ... 651
FILE_REMOVE ... 652
FILE_SET_ASCII ... 652
FILE_SET_FILENAME ... 652
FILE_SET_PERSIST ... 653
BINARY_APPEND ... 654
BINARY_CLEAR ... 654
BINARY_GET_SIZE ... 655
BINARY_LOAD ... 655
BINARY_SAVE ... 656
REQUEST class ... 656
ERROR_INFO ... 656
GET_NUM_COMMANDS ... 657
GET_SESSION_ID ... 657
REMOVE_COMMAND ... 658
RESET_COMMAND_BUFFER ... 658
SEND_COMMANDS ... 658
XML_COMMAND ... 659
VARIABLE class ... 660
EXIST ... 660
GET ... 660
REMOVE ... 661
SET ... 661
System service reference ... 662
Overview ... 662
Output buffer ... 662
Classes ... 663

Error codes 664
APPLICATION Class 664
COMMAND Class 664
CONTAINER Class 664
FILEDB Class 665
LICENSE Class 665
LISTENER Class 665
LOCK Class 666
LOG Class 666
MAIL Class 666
MQ Class 666
OBJECT Class 667
PACKAGE Class 668
PRGM Class 668
PROCEDURE Class 669
REGISTRY Class 669
REQUEST Class 669
SCHEDULER Class 670
SECURITY Class 670
SEMAPHORE Class 671
SEQUENCE Class 671
SERVICE Class 671
SESSION Class 672
SOAP Class 672
SYSTEM Class 672
THREAD Class 673
TRANSACTION Class 673
WEBSERVICE Class 673
XML Class 674
Permissions 675
Commands 677
APPLICATION class 677
CONFIG 677
CREATE 678
CREATE_DIR 679
DIR 680
GET_DIR 682
GET_FILE_DIR 682
GET_NAME 683
GET_WORK_DIR 683
GET_WROOT_DIR 683
INFO 684
INFOEX 686
INSTALL 687
PACKAGE 688
REMOVE 689
REMOVE_DIR 689

SESSIONS 691
START 691
STOP 692
COMMAND class 692
GET_PARAMETER 692
GET_HTTP_HEADERS 693
SET_HTTP_HEADER 694
SET_HTTP_RET_CODE 694
SET_OUTPUT_BUFFER 695
REMOVE_PARAMETER 696
SET_PARAMETER 696
CONTAINER class 697
COPY 697
ENCODE 698
EXPORT 699
GET_NUM_SUBCONTAINERS 700
GET_PARENT_NAME 700
GET_SUBCONTAINER_NAME 701
IMPORT 702
LIST 704
MOVE 704
REMOVE 705
VIEW 706
DEBUG class 707
DISPLAY_CONTAINER 707
DISPLAY_MESSAGE 708
DISPLAY_OBJECT 708
DISPLAY_VARIABLES 709
FILEDB class 709
CLOSE 710
OPEN 710
SQL 711
LICENSE class 711
EXPORT 712
GET 713
GET_TYPE 714
INFO 714
INSTALL 715
ISDEMO 716
REMOVE 716
SET 717
LISTENER class 718
CREATE 719
LIST 720
REMOVE 721
SET_PARAM 722
START 723

STOP 724
LOCK class 724
INFO 725
ISLOCKED 726
LOCK 726
UNLOCK 728
LOG class 728
CREATE 729
ERASE 731
GET_LEVEL 732
INFO 733
SEARCH 734
SET_OPTIONS 738
REMOVE 739
WRITE 740
MAIL class 741
SEND 741
MQ class 743
ISAVAILABLE 743
MISC class 744
EXEC 744
GET_CRASH 746
HASH 746
INFOEX 747
NOP 747
SLEEP 748
SEQUENCE 748
UNIQ 749
OBJECT class 750
CLEAR_ALL 750
COPY 750
CREATE 751
ENCODE 754
EXIST 755
GET 755
GET_NAME 758
GET_NUM 759
GET_TYPE 759
MOVE 760
REMOVE 760
SEND 761
SET_NAME 762
BOOLEAN_GET_VALUE 762
BOOLEAN_SET_VALUE 763
INTEGER_GET_VALUE 763
INTEGER_SET_VALUE 764
STRING_GET_VALUE 764

STRING_SET_VALUE 765
STRINGLIST_ADD_STRINGLIST 765
STRINGLIST_GET_SIZE 766
STRINGLIST_GET_VALUE 767
STRINGLIST_GET_VALUES 767
STRINGLIST_INSERT 768
STRINGLIST_REMOVE 769
STRINGLIST_SEARCH 769
STRINGLIST_SET_VALUE 770
STRINGLIST_SORT 771
STRINGLIST_SWAP 771
INDSTRINGLIST_ADD_INDSTRINGLIST 772
INDSTRINGLIST_GET_KEY 772
INDSTRINGLIST_GET_SIZE 773
INDSTRINGLIST_GET_VALUE 773
INDSTRINGLIST_GET_VALUES 774
INDSTRINGLIST_KEY_EXIST 775
INDSTRINGLIST_REMOVE 775
INDSTRINGLIST_SET_VALUE 776
TABLE_ADD_COLUMNS 777
TABLE_ADD_ROWS 777
TABLE_ADD_TABLE 778
TABLE_CLEAR 779
TABLE_CLEAR_CELL 779
TABLE_CLEAR_COLUMN 780
TABLE_CLEAR_DATA 781
TABLE_CLEAR_ROW 781
TABLE_CLEAR_ROWS 782
TABLE_CREATE_FROM 782
TABLE_EXPORT 783
TABLE_FILTER_COLUMNS 784
TABLE_GET_CELL_LINE 785
TABLE_GET_CELL 786
TABLE_GET_CELL_NUM_LINES 786
TABLE_GET_COLUMN 787
TABLE_GET_COLUMN_DESCRIPTION 788
TABLE_GET_COLUMN_INDEX 788
TABLE_GET_COLUMN_NAME 789
TABLE_GET_COLUMN_NAMES 789
TABLE_GET_DESCRIPTION 790
TABLE_GET_NUM_COLUMNS 790
TABLE_GET_NUM_ROWS 790
TABLE_GET_PRIMARY_INFO 791
TABLE_GET_ROW 792
TABLE_GET_ROWS 793
TABLE_GET_ROW_INDEX 794
TABLE_IMPORT 794

TABLE_INSERT_CELL_LINE 796
TABLE_INSERT_ROWS 797
TABLE_INSERT_COLUMN 797
TABLE_IS_COLUMN_NUMERIC 798
TABLE_JOIN 799
TABLE_LOAD 800
TABLE_MODIFY_COLUMN 800
TABLE_REMOVE_CELL_LINE 801
TABLE_REMOVE_COLUMN 802
TABLE_REMOVE_ROW 802
TABLE_REMOVE_ROWS 803
TABLE_SAVE 803
TABLE_SEARCH 804
TABLE_SELECT_FROM 805
TABLE_SET_CELL 806
TABLE_SET_CELL_LINE 807
TABLE_SET_COLUMN 808
TABLE_SET_COLUMN_DESCRIPTION 809
TABLE_SET_COLUMN_NAME 809
TABLE_SET_COLUMN_TYPE 810
TABLE_SET_DESCRIPTION 810
TABLE_SET_PRIMARY_COLUMN 811
TABLE_SET_ROW 812
TABLE_SORT 813
TABLE_SORT_CELL 813
TABLE_SWAP_CELL_LINES 814
TABLE_SWAP_COLUMNS 815
TABLE_SWAP_ROWS 815
FILE_APPEND 816
FILE_CLEAR 817
FILE_COMPRESS 817
FILE_CREATE 818
FILE_DECOMPRESS 818
FILE_EXIST 819
FILE_GET_DIRNAME 819
FILE_GET_EXTENSION 820
FILE_GET_FILENAME 820
FILE_GET_ORIGIN 821
FILE_GET_PATHNAME 821
FILE_GET_SIZE 822
FILE_REMOVE 822
FILE_SET_ASCII 823
FILE_SET_FILENAME 823
FILE_SET_ORIGIN 824
FILE_SET_PERSIST 825
BINARY_APPEND 825
BINARY_CLEAR 826

BINARY_GET_SIZE ... 826
BINARY_LOAD ... 827
BINARY_SAVE ... 828
PACKAGE class ... 828
INFO ... 829
INSTALL ... 829
PACKAGE ... 830
REGISTRY class ... 831
CLEAR_CACHE ... 832
CLOSE_REGISTRY ... 833
CREATE ... 833
EXIST ... 835
EXPORT ... 836
GET ... 837
IMPORT ... 839
OPEN_REGISTRY ... 841
REMOVE ... 842
SET ... 843
REQUEST class ... 845
CLOSE ... 845
OPEN ... 846
SCHEDULER class ... 848
CREATE_TASK ... 849
ENABLE_TASK ... 850
GET_TASK_PARAM ... 851
REMOVE_TASK ... 853
RUN_TASK ... 853
SET_TASK_PARAM ... 854
TASK_LIST ... 858
SECURITY class ... 859
ADD_ROLE ... 859
ADD_SESSION_PERMISSION ... 860
ADD_USER ... 861
CHECK ... 862
ENABLE_USER ... 863
INFO ... 864
LOAD_USER_CONTEXT ... 865
PERMISSION_LIST ... 866
REMOVE_ROLE ... 867
REMOVE_ROLE_INHERITS ... 867
REMOVE_ROLE_PERMISSIONS ... 868
REMOVE_SESSION_PERMISSION ... 869
REMOVE_USER ... 870
REMOVE_USER_ROLES ... 870
ROLE_EXIST ... 871
ROLE_LIST ... 872
SAVE_USER_CONTEXT ... 873

SESSION_PERMISSIONS 874
SET_OTHER_USER_PASSWORD 875
SET_PARAM 875
SET_ROLE_DESCRIPTION 876
SET_ROLE_INHERITS 877
SET_ROLE_PERMISSIONS 878
SET_USER_DESCRIPTION 879
SET_USER_FULLNAME 880
SET_USER_PARAM 881
SET_USER_PASSWORD 882
SET_USER_ROLES 882
USER_EXIST 883
USER_LIST 884
USER_PERMISSIONS 885
SEMAPHORE class 886
INFO 887
LOCK 888
REMOVE 889
UNLOCK 889
SERVICE class 890
CONFIG 890
IDENT 891
INFO 891
INFOEX 893
INSTALL 893
MOUNT 894
PACKAGE 896
PROTECT_CONTAINERS 896
SESSIONS 897
SKELETON 898
START 900
STOP 901
UNMOUNT 902
SESSION class 902
CHECK_CLOSE_REQUEST 903
CLOSE 903
CLOSE_OTHER 904
CREATE 905
GET_NAMED 906
FREE_NAMED 907
GET_NUM 908
GET_PASSWORD 908
GET_SESSION_ID 909
GET_USER 909
LIST 910
SERVICES 912
SET_CREATOR_IP 913

SET_DEFAULT_ERROR_PROC ... 913
SET_DEFAULT_ERROR_XSL ... 914
SET_LANGUAGE ... 915
USE_ID_COOKIE ... 915
SYSTEM class ... 916
GET_ENCODING ... 916
GET_NUM_PROCS ... 917
GET_NUM_EXCEPTIONS ... 917
GET_PLATFORM ... 918
GET_TYPE ... 918
GET_VERSION ... 919
TEST class ... 919
CRASH ... 919
TEST_BROWSER ... 920
THREAD class ... 921
CREATE ... 921
TIME class ... 922
GET ... 922
TRANSACTION class ... 923
GET_CONTAINER ... 925
GET_VARIABLES ... 926
INFO ... 927
INFOEX ... 929
REMOVE ... 930
RESTART ... 930
ROLLBACK ... 933
SET_OPTIONS ... 933
SET_STATUS ... 935
START ... 936
VALIDATE ... 939
VARIABLE class ... 940
ENCODE ... 940
EXIST ... 941
GET ... 941
GET_MULTI ... 942
REMOVE ... 943
SET ... 943
WEBSERVICE class ... 944
EXECUTE ... 944
XML class ... 945
GET_XML ... 945
LOAD_XSL ... 946
SEND ... 947
SET_XML ... 949
SET_XSL ... 951
TRANSFORM ... 952
VALIDATE ... 953

XML 955
Overview 955
Input XML 955
Output XML 956
Error XML 959
Parsers 961
Normal model 961
Input 961
Main structure 961
Reference 962
NIRVA 962
NVCOMMAND 962
NVPARAM 963
NVCONTAINER 963
NVOBJ 964
NVDATA 965
NVDESCRIPTION 966
NVPRIMARY 966
NVCOLDESC 967
NVROW 967
NVCOL 967
NVORIGIN 968
Output 968
Main structure 968
Reference 969
NIRVA 969
NVHTTPHEADER 970
NVSESSIONVAR 970
NVCONTAINER 971
NVOBJ 971
NVDATA 972
NVDESCRIPTION 973
NVPRIMARY 973
NVSIZE 974
NVCOLDESC 974
NVROW 975
NVCOL 975
NVNAME 976
NVEXTENSION 976
NVDIRECTORY 976
NVPATHNAME 977
NVORIGIN 977
Error 978
Main structure 978
Reference 978
NIRVA 978
NVERROR 979

NVERRORCODE 979
NVERRORSERVICE 979
NVERRORCLASS 980
NVERRORDESC 980
NVERRORINFO 981
Simple model 981
Input 981
Main structure 981
Reference 982
Main element 982
Command parameters 983
Container 983
Boolean object 983
Integer object 984
String object 984
StringList object 985
IndStringList object 985
Table object 986
File object 987
Binary object 987
Output 988
Main structure 988
Reference 989
Main element 989
HTTP headers 989
Session variables 990
Container 990
Boolean object 990
Integer object 991
String object 991
StringList object 991
IndStringList object 992
Table object 992
File object 993
Binary object 994
Error 994
Main structure 994
Reference 995
nvfault 995
code 995
service 995
class 996
description 996
info 996
Compact model 996
Output 997
Main structure 997

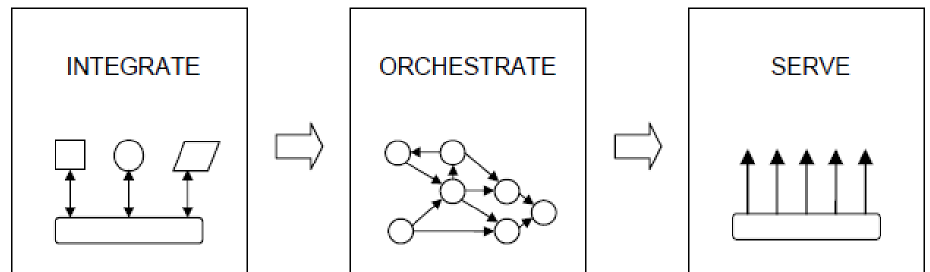
Reference ... 997
Main element ... 997
Session variables ... 998
Container ... 998
Boolean object ... 999
Integer object ... 999
String object ... 999
StringList object ... 1000
IndStringList object ... 1000
Table object ... 1001
File object ... 1001
Binary object ... 1002
Error ... 1002
Main structure ... 1002
Reference ... 1003
nvfault ... 1003
code ... 1003
service ... 1003
class ... 1004
description ... 1004
info ... 1004
Tiny model ... 1004
Output ... 1005
Main structure ... 1005
Reference ... 1005
Main element ... 1005
Session variables ... 1006
Container ... 1006
Boolean object ... 1007
Integer object ... 1007
String object ... 1007
StringList object ... 1008
IndStringList object ... 1008
Table object ... 1009
File object ... 1009
Binary object ... 1010
Error ... 1010
Main structure ... 1010
Reference ... 1011
nvfault ... 1011
code ... 1011
service ... 1011
class ... 1012
description ... 1012
info ... 1012
Security model ... 1013
Overview ... 1013

Permissions ... 1014
Roles ... 1015
Users ... 1015
Checking a permission ... 1015
Security service ... 1015
Overview ... 1015
Implementation ... 1016
LOGIN ... 1016
CHANGE_PASSWORD ... 1017
LOAD_USER_CONTEXT ... 1017
SAVE_USER_CONTEXT ... 1018
Single Sign-On (SSO) ... 1019
Overview ... 1019
Configuring the domain controller ... 1020
Creating a special user account ... 1020
Setting principal names ... 1022
Allowing delegation ... 1023
Configuring the Nirva server ... 1024
Running Nirva as the special user account. ... 1024
Configuring Nirva ... 1025
Configuring Clients ... 1027
Browsers ... 1027
Nirva clients ... 1029
Installation packages ... 1030
Overview ... 1030
The package description file ... 1030
Creating a package ... 1034
Installing a package ... 1035
Test sets ... 1036
Overview ... 1036
The test set file ... 1036
Benchmarks ... 1039
HTTP server ... 1039
Test environment ... 1039
Test description ... 1039
Test results ... 1040
Comments ... 1040
Using nvcc ... 1040
Proprietary rights notice ... 1042
RSA Security ... 1042
Java Runtime Environment ... 1042
OpenSSL ... 1043
The "Artistic License" ... 1044
GNU GENERAL PUBLIC LICENSE ... 1048
Perl ... 1053
LIBXML/LIBXSLT ... 1054

Overview

What is Nirva?

Nirva is a robust application server combined with integration and orchestration services.

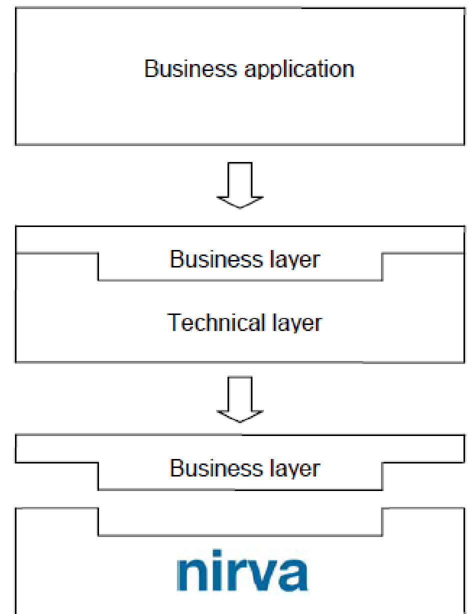


Application server

All enterprise applications or software products have two distinct parts: The business or functional part and an underlying infrastructure bearing the responsibility of managing all the technical aspects: communication with clients and systems, data merging between different sources, scalability, security, integration, session management, Web user interface, component reusability, etc.

The latter is also the main role of an enterprise application server providing a robust and high performance technical platform.

Architects and developers can then concentrate their efforts on building business related functionality while reducing their reliance on the underlying technical aspects.



Nirva provides the necessary infrastructure to host enterprise applications and stand alone products alike.

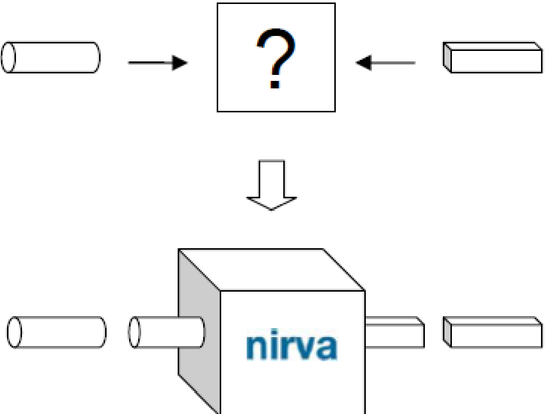
Integration services

The Nirva integration part, although quite capable of being implemented as a strategic component, often remains perceived as primarily tactical as typical similar requirements are frequently met on the market.

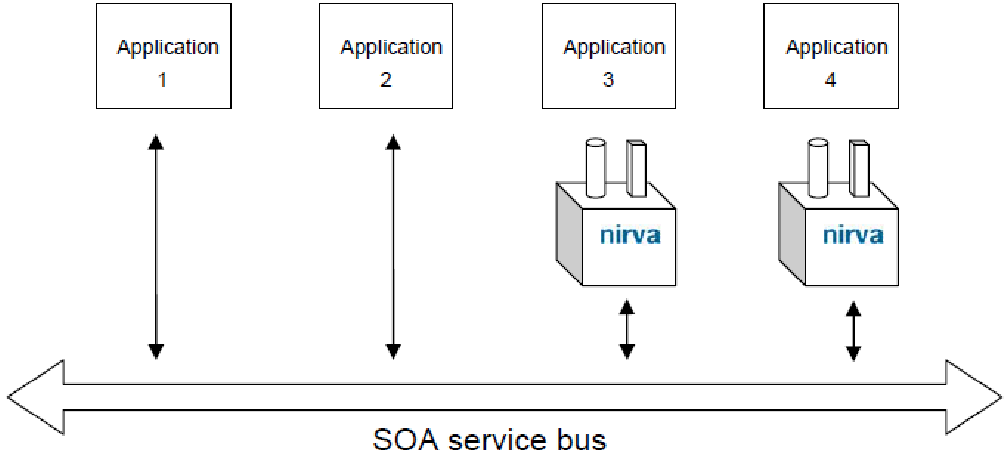
Newer strategic products, based on SOA and Web Services are often too remote from the technical reality to provide a suitable tactical answer for a quick business solution turnaround. In fact such products integrate applications already prepared to communicate via Web Services.

On the other hand, Nirva is able to integrate not only Web Services but also application APIs seen as pieces of code or technology to access the business functionality offered by core systems. For example Nirva can erase boundaries between Java and .Net by allowing these two worlds to coexist in a single process where it can communicate between them.

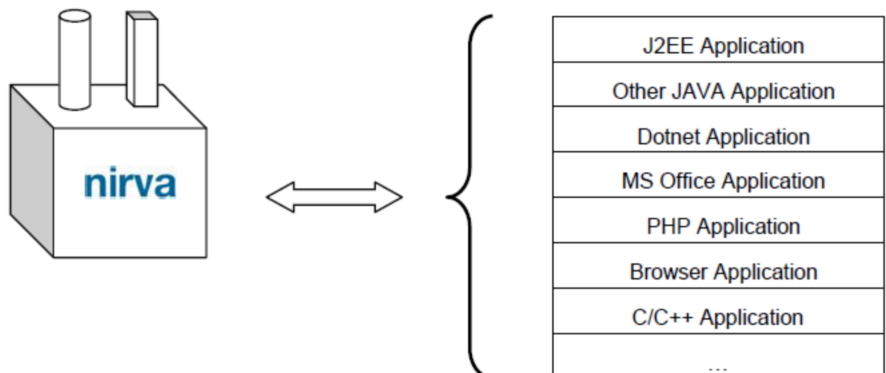
Nirva can be positioned as “the part and the glue that allows a seamless connection between a round peg and a square hole”.



As a tactical integration product, Nirva can be used in conjunction with other SOA products to “Web Service” entire applications or any of their components:



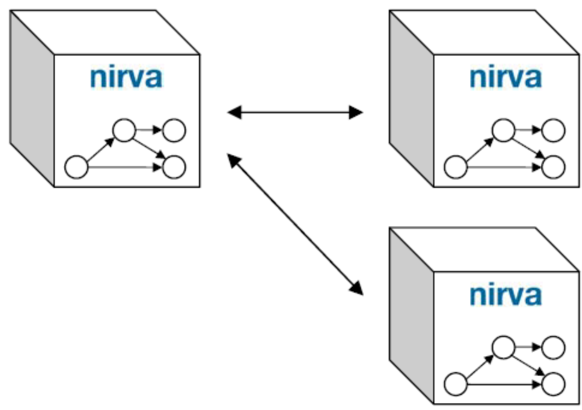
Nirva does not stop at offering Web Services connectivity. Its architecture allows for instance the publishing of a part of a hosted application to third party components through a vast array of technical connectors:



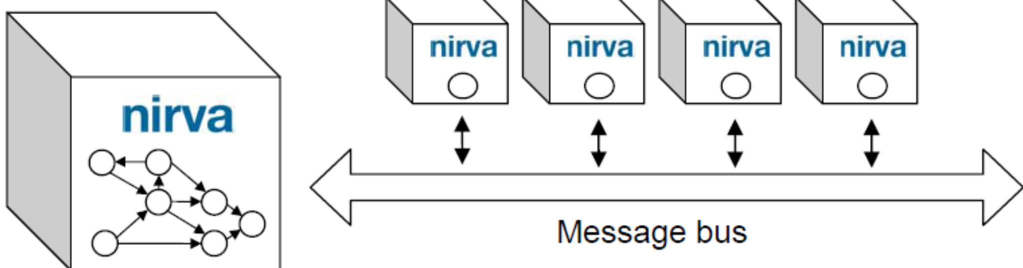
Orchestration

Nirva provides orchestration features that help in building applications with a business perspective in mind.

Orchestration can be coded into simple procedures written in known languages (Java, Perl, .Net) and distributed across several servers.



Orchestration can also be controlled by a dedicated workflow service that routes activities to agents across a message bus. This mode facilitates the scalability of the system and gives a clear business view of the process.



Who is Nirva for?

Nirva is used as:

- An Application Server for enterprise-wide projects.
- An Application Platform for the products.

Projects

Nirva targets tactical projects for large and medium companies, especially for back-end processes. This kind of project is often motivated by cost reduction but is paradoxically supported by oversized products (e.g. WebSphere, WebLogic). Nirva allows significant cost savings, not only for licenses, but also for development, maintenance and operations costs.

For the larger companies, Nirva adequately complements the enterprise strategic tools by allowing them to be perceived as a business solution by hiding the technical aspects of the various components and by federating heterogeneous information and application systems.

For medium size companies, Nirva projects can be used as a more strategic tool by supplying a robust, independent and complete application base layer.

Products

For software suppliers, Nirva is used as an application backbone supporting their own products. Its feature rich and flexible architecture hides a significant part of the technical components, thus allowing product development to focus on core functionality. Nirva offers value from the technical integration layer up to the presentation and front-end layer.

Nirva supports the development of applications opened to the outside world (APIs), secured (users and rights management) and extensible (distributed architecture). Its application domain is unrestricted (bank, insurance, industry, health care, etc.).

Nirva can host and support standard applications (client/server) or «Software as a Service» (SaaS) type with multi tenancy management.

Users are small and medium software suppliers. Nirva can also be used by specialized integrators who wish to standardize and reuse their development.

How Nirva works?

Nirva is a standalone multithread engine containing all the necessary technology to host classic or Web applications.

All-in-One concept

Nirva is a self-sufficient, all-in-one product. All necessary components (with the exception of traditional databases) to manage applications are supplied within the product:

- Web Server
- HTTP/HTTPS Server
- Integrated Java environment
- Integrated Perl interpreter
- Embedded .Net CLR engine (windows only)
- XSLT processor
- Scheduler
- Listeners
- Workflow
- Deployment tools
- Test suite
- Monitoring tools
- Licenses creation and maintenance tool
- Web Services (producer and consumer)
- Application container
- Upward (services) and downward (clients) connectors
- Security services
- SQLite file database

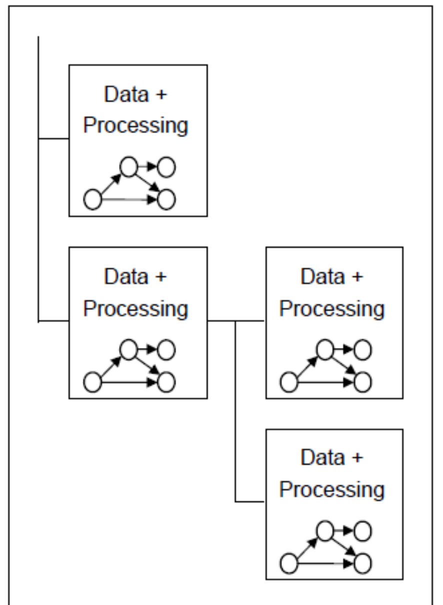
Even if Nirva delivers all these components, some of them can be externalized. For example, the Java virtual machine or the XSLT processor.

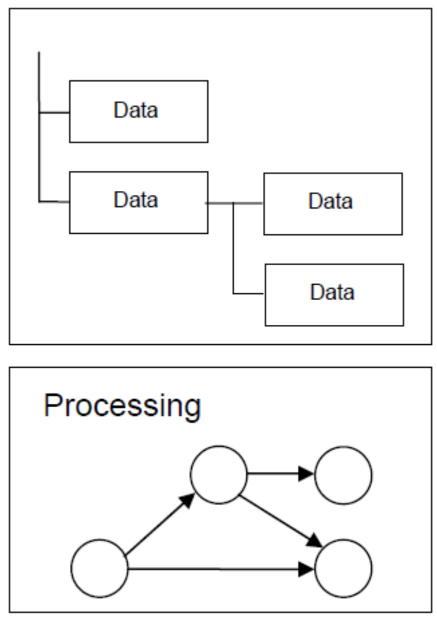
This all-in-one concept caters for significant development and running cost reductions.

Data centric architecture

The architecture is data centric as opposed to language centric, which is often the case with most traditional application servers (Java or .Net). This particular point is at the core of the product originality.
Data are organized in hierarchical containers that can be persistent or not, shared or specific to user sessions. The container data comes from user data, databases or other processing.

Coding is only used to process and organize this data that can then be presented to the user.





Language centric **Data centric**

This model allows for loose coupling between the data and the code that manipulates them. For example, it is possible to use different programming languages for a single application. Nirva uses Java, .Net (C#, Visual Basic), Perl and C++ as processing languages. Each language's strength can be exploited and existing code can be reused.

Main components

Thanks to the simplicity of its architecture (containers, services, procedures, applications, commands and connectors) Nirva focuses on business solutions and reduces development time.

Nirva enables loose coupling between its various components to enhance product evolution.

These components are:

- ***Container***. Stores business data at user context level (session) or at application level (shared data). The hierarchical container stores well defined objects (boolean, integer, string, string list, indexed string list, table, file, and binary data) and can also store sub-containers. Containers can be persistent in registries or in a mass storage service.
- ***Command***. A command is a simple string of characters containing the name of a service, the name of the command and a set of parameters in the form of name=value. A command retrieves data from a container and populates another one. This loose coupling allows unmatched flexibility in the product.

- ***Service***. A service is a set of commands for a particular domain. There is a SYSTEM service fully integrated in Nirva that manages container data access and other system wide features. Users can also create their own services to add functionality to the product. Services can be programmed using C++, Java or .Net. Services can send commands to other services. A Service is a documented, configurable and deployable component with a separate life cycle. A Service can be used by several applications.
- ***Procedure***. A procedure is a portion of code that chains commands according to business logic. They can be written in Perl, Java, .Net or a simple text file. Each command can execute one or several procedures written in different languages.
- ***Application***. An application is composed of a set of procedures, a Web site and the files needed for the presentation layer. All the application components are stored in a dedicated directory.
- ***Connectors***. Connectors are used to access application functionality from external applications or from a Web browser.

Key features

- **All-in-One concept**. Nirva is a self-sufficient, all-in-one product. All necessary components (with the exception of databases) to manage applications are supplied within the product. Learn more…
- **Data centric architecture**. The architecture is data centric as opposed to language centric. Data are organized in hierarchical containers that can be persistent or not, shared or specific to user sessions. Learn more…
- **Simple model**. Thanks to the simplicity of its architecture (containers, services, procedures, applications, commands and connectors) Nirva focuses on business solutions and reduces development time. Learn more…
- **Multi language**. Nirva applications and services can be developed in the following languages: Perl, Java, .Net (C#, Visual Basic), C++. Languages can be mixed. Learn more…
- **Scalable distributed architecture**. Nirva is built as a distributed architecture to support application loads. It provides failover and load balancing features. Learn more…
- **Standards-based**. Nirva is based on well defined standards. Learn more…
- **Extensible**. The product functionality can easily be extended thanks to the service component. The Nirva service is a transverse component with its own life cycle and can be used to add new commands to the system. Learn more…
- **Component reusability**. Components can be reused to share the common part of applications. This boosts application agility and significantly reduces conception and maintenance costs. Learn more…
- **Upwards and downwards integration**. Nirva can integrate components, applications or external technology to build composite applications (upwards integration) but can also integrate these applications in other applications with a set of connectors (downwards integration). Learn more…

- **Message bus**. Nirva contains a message bus to support process management in asynchronous mode. In this mode, a central entity stores messages (activities) in queues where specialized agents can pick them up for processing. Learn more…
- **Session management**. Nirva supports the concept of a session that represents a user's context. Learn more…
- **Security**. Nirva supplies security functions as a standard. The security model is based on permissions, roles and users. Security includes SSO (Single Sign On management) to allow automated authentication of users already connected on a particular domain. Learn more…
- **Web Services**. Nirva is both a producer and consumer of Web Services. Web Service creation is extremely simplified. A Web interface is used to define the structure of the Nirva containers for input and output messages and to define a procedure to execute the relevant Web Service. Nirva automatically generates the WSDL code describing the Web Service. Learn more…
- **Scheduler**. A scheduler is integrated in the core system and supports the planning of various tasks during a day, a week, a month or a year. Learn more…
- **Listeners**. Listeners wait for events. They can be used for example to watch a directory to retrieve files and to store them in the application for processing. Learn more…
- **Presentation layer**. Nirva applications can supply a presentation layer accessible through a standard Web browser. The presentation layer works with HTML renderers (e.g. XSLT or JSP) that transform the content of an output container to HTML flow. Alternatively, Nirva provides a framework based on reusable and customizable JavaScript components named Nidgets. Learn more…
- **Workflow**. The Workflow service supports application development with a business perspective in mind. Business logic is distributed in activities that each user can see and organize according to each project specification. Learn more…
- **Event-driven capabilities**. The Nirva EVENT service allows applications to run in event-driven mode on a publish/subscribe model. Learn more…
- **Database**. Nirva allows database communication using ODBC and JDBC via dedicated services. Learn more…
- **Mass storage**. Nirva supplies a mass storage service that can be used to store or archive large amounts of data coming from containers or files. Learn more…
- **Registry**. A registry system interfaced with a Web editor allows storage of system configuration, application and service data. Learn more…
- **Configuration**. Product configuration is available through a Web interface or programmatically. All system parameters, but also applications and services can be configured this way. Learn more…
- **Monitoring**. Monitoring tools can be used to check system functions and to send automated alerts to an administrator in case of problems. Learn more…
- **Logs**. Nirva supplies logs to track system or application processing. It is possible to add logs to any given application. Nirva controls their size, their retention time and supplies search facilities. Learn more…
- **Multi-applications**. A single Nirva instance can host several applications at the same time. Applications remain isolated from one another, have their own life cycle but can still communicate amongst themselves. Learn more…

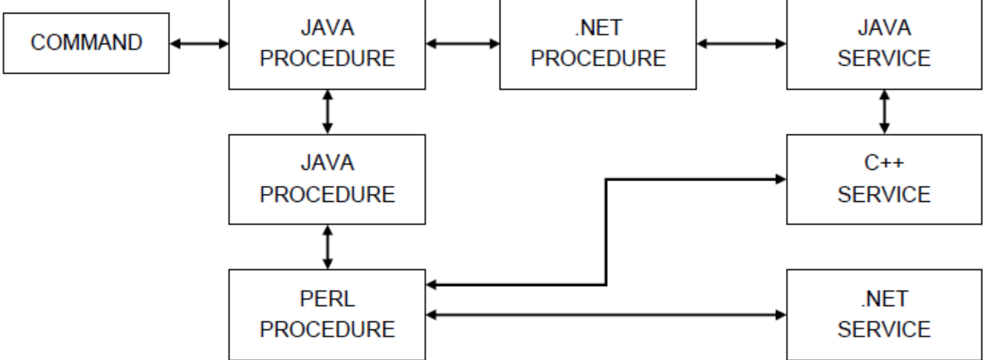
- **Deployment**. Nirva components (application, services, Web Services) can be packaged and deployed in few clicks from a simple Web browser. Learn more…
- **Multi-platform**. Nirva is compatible with the following platforms: Windows, Linux, AIX, Solaris, HP-UX. Learn more…
- **Development tools**. Nirva is programmed using standard development tools (Eclipse, Visual Studio, etc…). Nirva also supplies tools to fine tune, debug and test. Learn more…

Multiple languages

Thanks to its structure, the product supports several programming languages, even within the same application. Supported languages are:

- Java
- .Net (all .Net supported languages)
- Perl
- C++

For simple procedures that do not need specific logic, it is also possible to use simple text files containing commands.



This unmatched flexibility allows for rapid and efficient integration of legacy programs. It also supports the use of specific technology only available in a particular language.

Java and .Net can cohabit in Nirva.

Distributed Architecture

Nirva is built as a distributed architecture to support application loads. An application can be deployed on one or several servers that can be dedicated to particular tasks.

Servers can support load balancing and failover architectures.

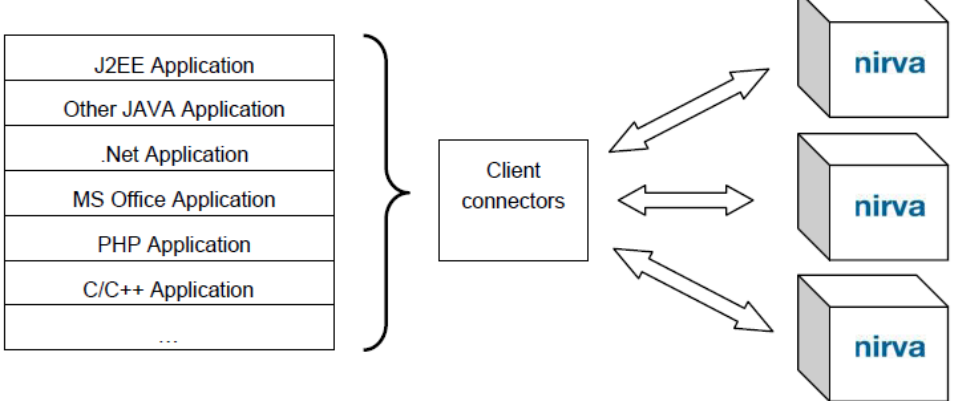
Channels between the various servers allow fast and simple communication between them.

There are several configurations for load balancing and failover:

- Client connector load balancing

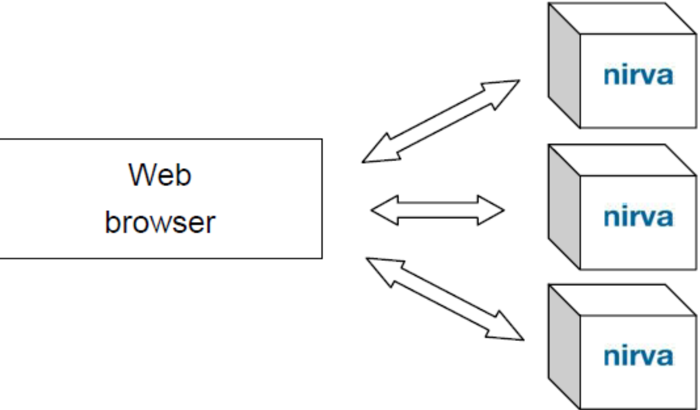
- Web front end load balancing
- Nirva to Nirva load balancing
- Nirva to Nirva with failover
- Message bus

Client connector load balancing

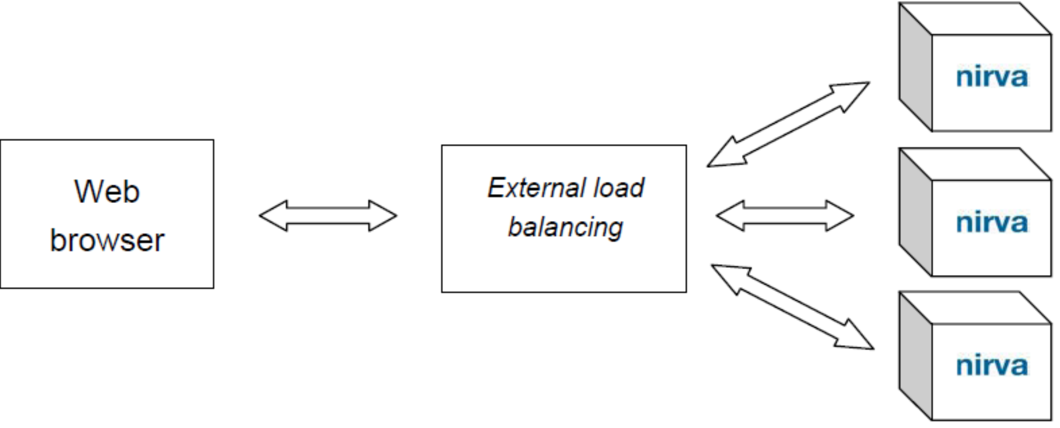


Web front end load balancing

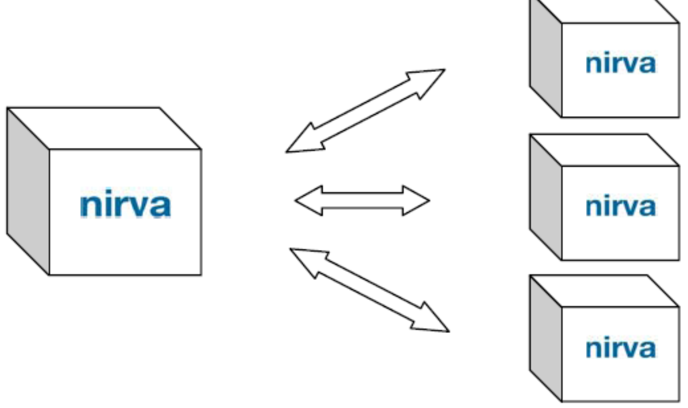
Nirva uses HTTP redirection for Web load balancing:



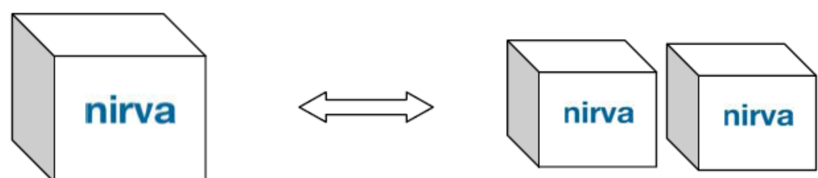
Load balancing can also be provided by dedicated external software (ex apache server) or hardware solutions synchronized on the Nirva session.



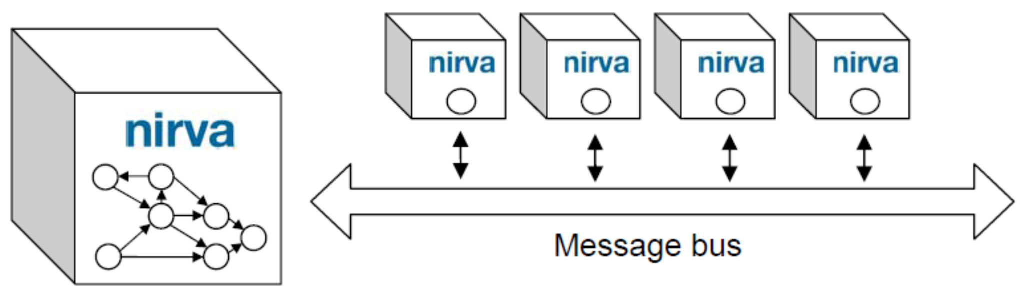
Nirva to Nirva load balancing



Nirva to Nirva with failover



Message bus



Standards

Nirva is based on standards and proven components:

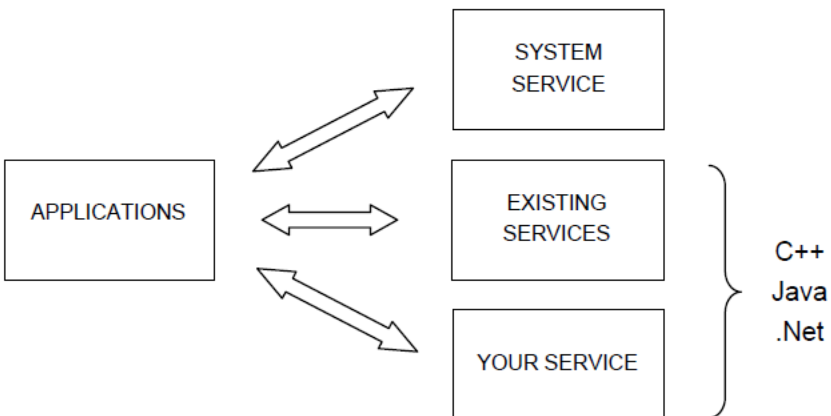
- HTTP/HTTPS
- XML
- XSLT
- SOAP
- WSDL
- Java
- .Net
- Perl

Extensible

The product functionality can easily be extended thanks to the service component.

The Nirva service is a transverse component with its own life cycle and can be used to add new commands to the system. These commands can then be used by applications as any other command of the internal Nirva service. A service can support either additional technology or a business process.

Services can be developed in Java, .Net or C++.



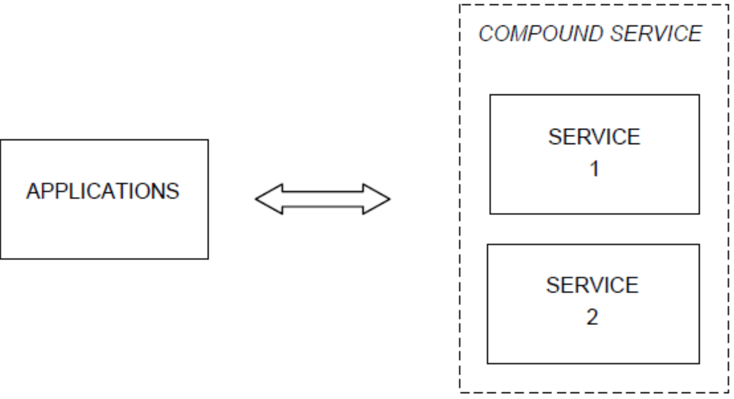
Nirva supplies a license management tool that external partners can use to control their own services' licensing. An external service supplier can easily "sell" its added value and control its deployment.

Here are a few examples of Nirva services:

- DATABASE (ODBC access to databases)
- JDBC (JDBC access to databases)
- WORKFLOW (orchestrating processes)

- STORAGE (mass storage)
- EVENT (event management)
- PDF (PDF file manipulation)
- Etc.

Services can call commands of other services. This feature can be used to create compound services. For example we can build an ARCHIVE service that uses the DATABASE and STORAGE services.

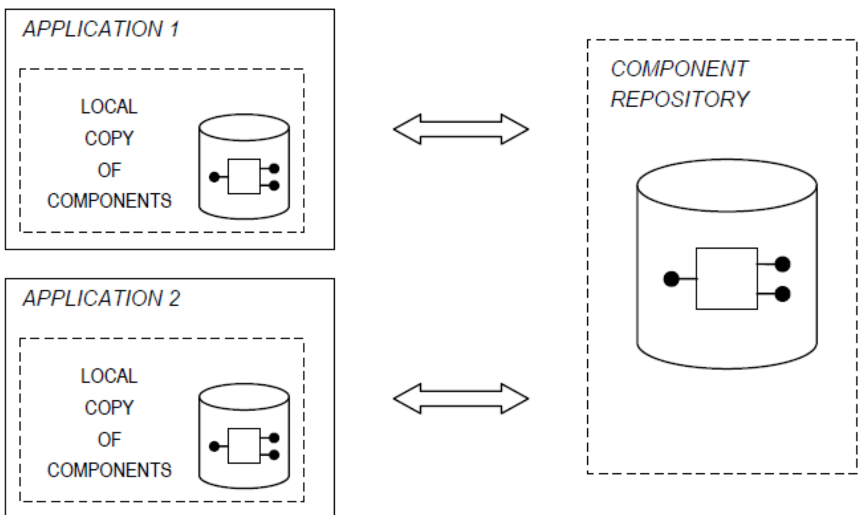


A developer can create a Nirva service in few clicks and start programming it from an automatically generated skeleton code. Nirva services are installed and configured from the main Nirva Web configuration tool.

Component reusability

Components can be reused to share the common parts of applications. This boosts application agility and significantly reduces conception and maintenance costs. Reusability is possible by creating Nirva services (transverse components with their own life cycle) or by using a specialized service (the NASC service or Nirva Application Software Component).

The NASC service manages common components as versioned code used by applications. They can either be processing or presentation components. They are automatically loaded when an application starts.

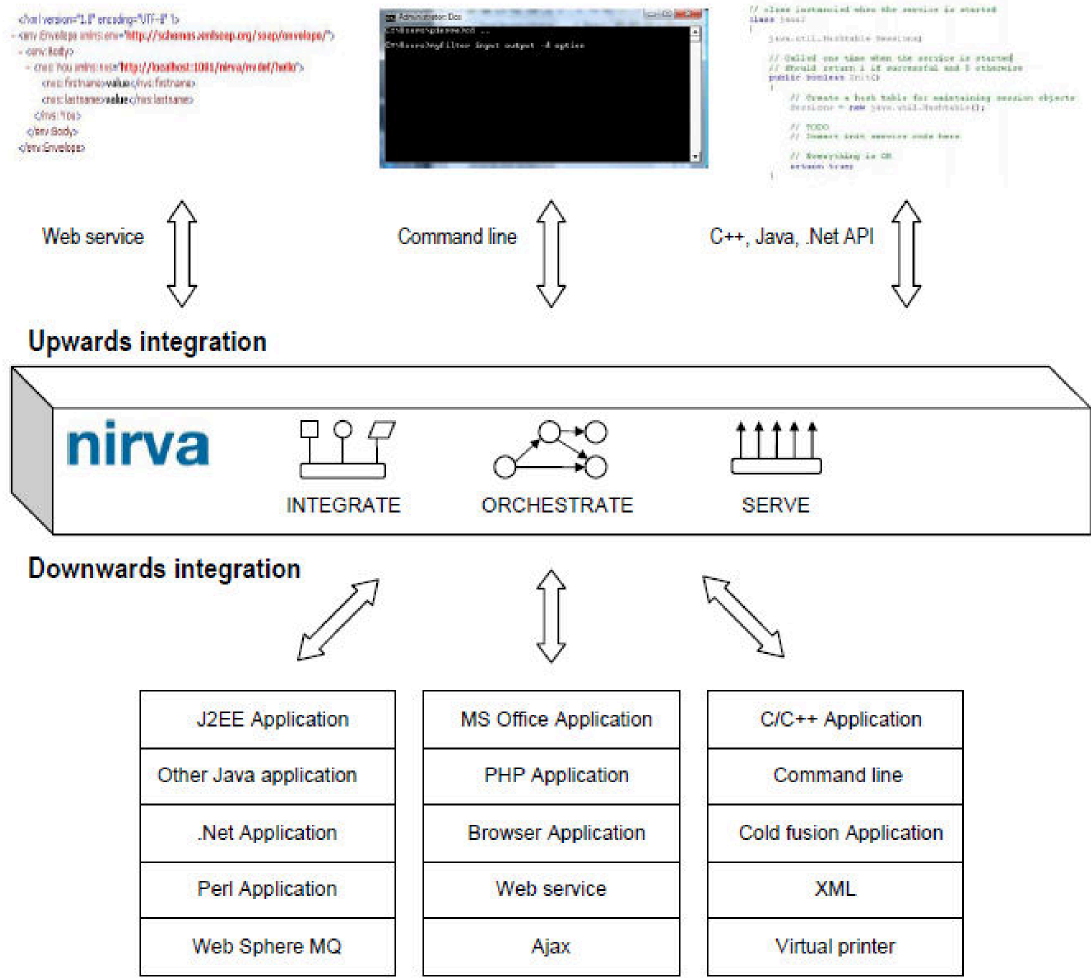


Upwards and downwards integration

Nirva can integrate components, applications or external technology to build composite applications (upwards integration) but can also integrate these applications in other applications with its connectors (downwards integration).

Upwards integration can be achieved with:

- The creation of a service (integration via the API of the target application or technology).
- The connection to a Web Service or any other form of XML-based communication.
- A program launched from a command line.

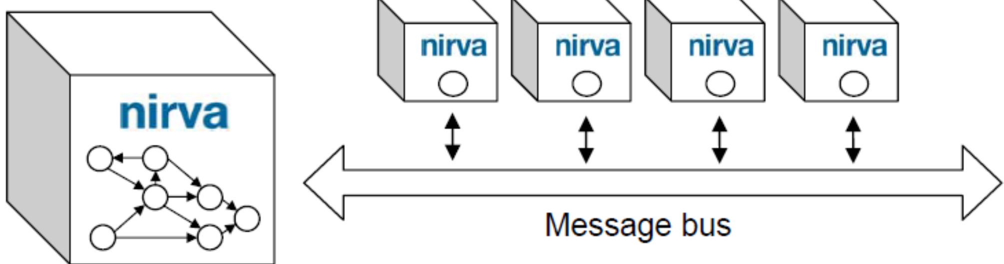


Message bus

Nirva contains a message bus to support process management in asynchronous mode.

In this mode, a central entity stores messages (activities) in queues where specialized agents can pick them up for processing.

Each agent processes the activities according to its capacity. This mode can be used to increase application scalability since agents can be easily added (or extra resources made available for existing agents) to boost performance.

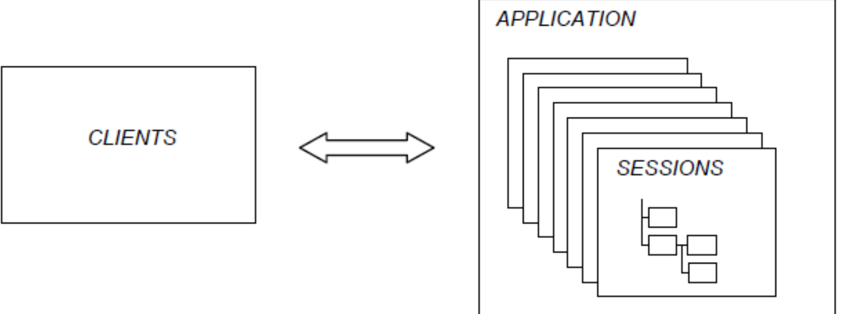


“Message bus” processing mode is supported by the Workflow and Event services.

Session management

Nirva supports the concept of a session that represents a user’s context. An application can only be connected through a session. There are different types of sessions:

- Client: user initiated session from a connector or a Web browser.
- Named: session shared between users. This session type can be used to create database connection pools.
- Scheduler: session initiated by the integrated scheduler.
- Listener: session waiting for an event.
- Thread: session asynchronously triggered in the background.
- Internal: session manually created by a dedicated command from a procedure or service.
- System: session created by the Nirva engine itself.

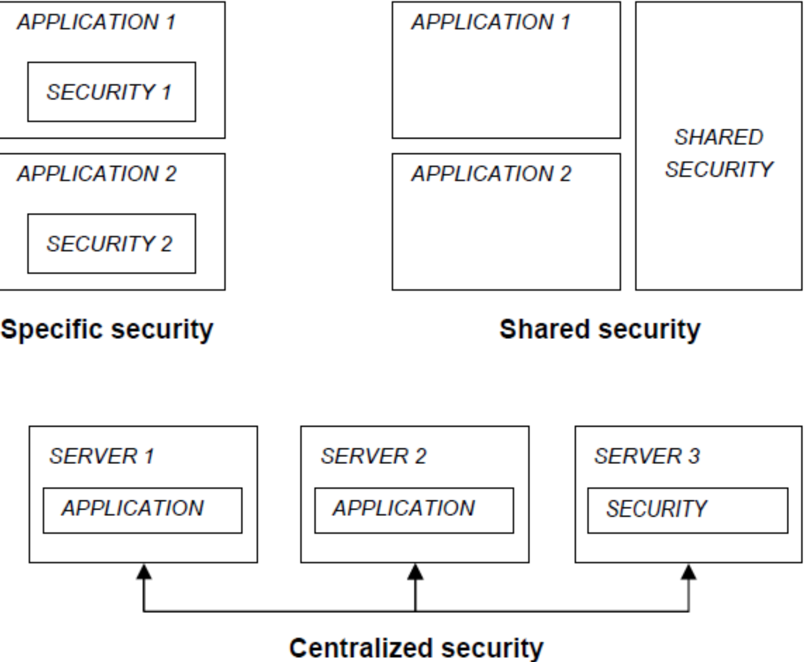


Sessions allow concurrent multiple users (a single server can handle hundreds of them) and the total isolation of data for each user. The user data is located in session containers.

Security

Nirva supplies security functions as a standard. The security model is based on permissions, roles and users. Security includes SSO (Single Sign On management) to allow automated authentication of users already connected on a particular domain.

Security can be specific to an application, shared between several applications or shared between several servers.



The security model can be extended or entirely replaced by an external security system. For that purpose, a security service can be used to replace the standard security in Nirva. Of note is a security service for LDAP.

Web Services

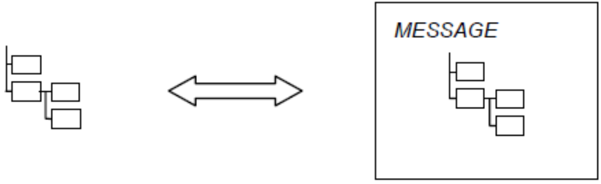
Nirva is both a producer and consumer of Web Services.

Using Web Services is supported with the help of the integrated XSLT processor that transforms Nirva input and output, and with the XML generator that converts container data to XML format.

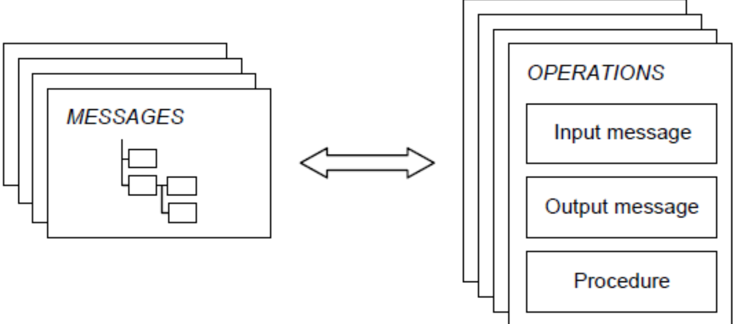
Web Service creation is extremely simple. A Web interface is used to define the structure of the Nirva containers for input and output messages and to define a procedure to execute the relevant Web Service. Nirva automatically generates the WSDL code describing the Web Service.

This occurs in 4 steps:

- Define containers for messages (Web interface).
- Associate messages with operations (Web interface).
- Create and code the procedure for each operation.
- Publish the Web Service (Nirva automatically creates the WSDL).



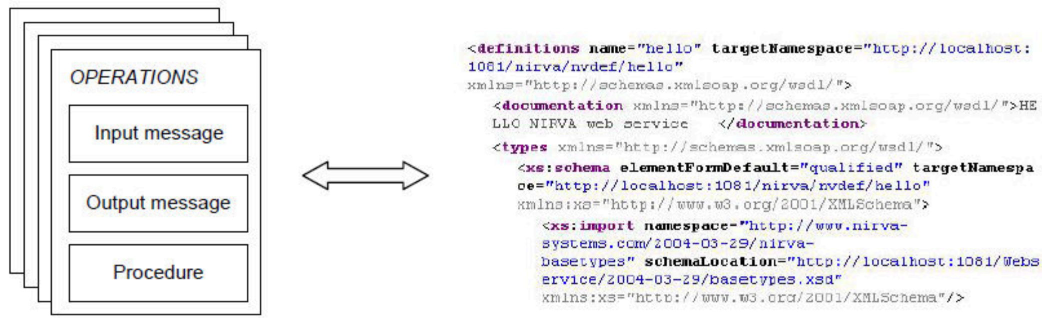
1. Define containers for messages



2. Associate messages with operations

```
//
// HELLO Nirva web service

class Welcome
{
    public static final int main() throws NirvaException
    {
        // Get first and last names
        nvsint MyNirva = new nvsint();
        MyNirva.Command("NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|firstname|");
        String FirstName = MyNirva.GetResult();
        MyNirva.Command("NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|lastname|");
        String LastName = MyNirva.GetResult();

        // Create the welcome message
        MyNirva.Command("NV_CMD=|OBJECT:CREATE| NAME=|welcome| TYPE=|STRING| VALUE=|Wel
            + FirstName + " " + LastName + "|");

        return 0;
    }
}
```

3. Code the procedure for each operation



4. Publish the web service. Nirva automatically creates the WSDL.

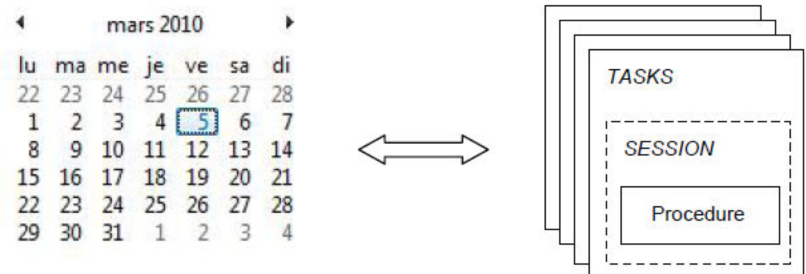
Nirva security specifies which user can access a particular Web Service.

Applications, through Web Services, can communicate with the external world using recognized standards.

Scheduler

A scheduler is integrated in the core system and supports the planning of various tasks during a day, a week, a month or a year.

The tasks can be repeated indefinitely or a fixed number of time during a defined interval.

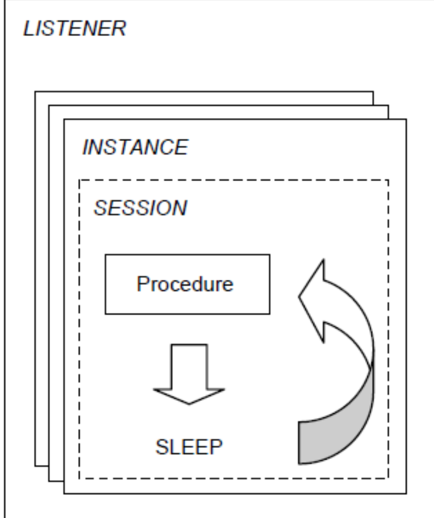


Using the scheduler reduces license costs as no external product is necessary.

Listeners

Listeners wait for events. They can be used for example to watch a directory for retrieving files to store them in the application to be processed.

Several instances of each listener can be executed in separate threads. A Listener runs a procedure in a loop with a defined sleep time between each occurrence.



Listeners are heavily used in distributed applications since they can be used to adjust additional application load. It is possible to define several instances of a particular listener and to distribute them on multiple machines.

They are often used for managing workflow activities in a message bus context.

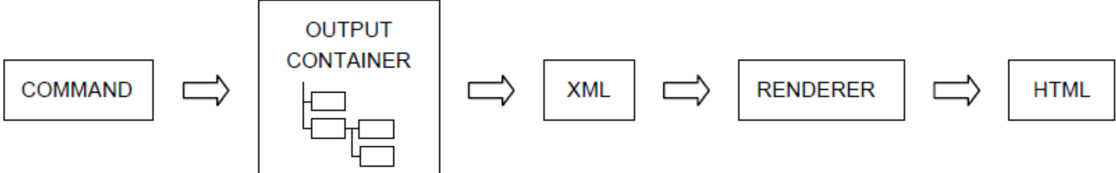
Presentation layer

Nirva provides two ways for building Web presentation layers:

- HTML renderers
- Nidget framework

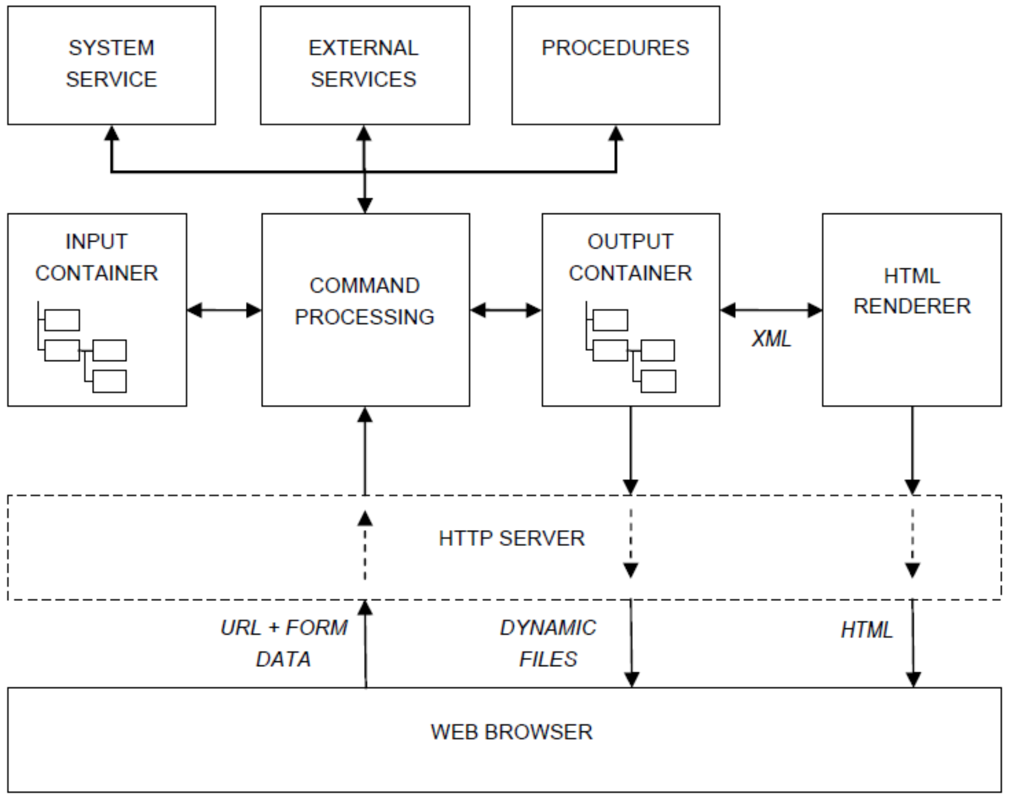
Html renderers

The presentation layer works with HTML renderers that transform the content of an output container to HTML flow.



Nirva provides built-in renderers and delivers an interface for third parties to write their own renderers.

This is the typical request for a dynamic page. Static pages are directly handled by the HTTP server.



The Web browser sends a command to Nirva as a GET or POST HTTP request with optional form data. A typical Nirva URL command is the following:

```
http://myserver:1081/nv_app_myapp/NVS?command&NV_CMD=MYSERVICE:MYCLASS:MYCOMMAND&NV_PROC
=java:myproc&NV_XML_XSL=MYXSL&NV_SESSION_ID=1234567890
```

On receiving this request, Nirva first connects the session identified by the NV_SESSION_ID parameter. If this parameter is not given, Nirva opens a new session for the application “MYAPP” (other parameters authenticating the user must then be given).

The optional URL form data is transformed into session variables or file objects if there is a file upload and are reachable from the procedures and services.

As the NV_PROC parameter gives a name of a Java procedure (java:myproc), Nirva executes this procedure. The NV_PROC parameter contains the name of one or several procedures that are executed before the command itself. Another parameter allows execution of other procedures after the command. A procedure contains itself some Nirva commands and some calls to other procedures. Procedures are the place to build the business logic.

Then Nirva executes the command given in parameter NV_CMD. This is the command named “MYCOMMAND” for the class “MYCLASS” of the service “MYSERVICE”. If the NV_CMD parameter is not given, Nirva executes no command. This is generally the case when using Nirva as a Web application server where the URLs just instruct Nirva to execute procedures.

A command gets information from an input container and delivers its data to an output container. The container names are given as parameters of the command but there are some default values. In this example, both input and output containers point to the same default container. The procedures inherit the name of the input and output containers.

When the command has finished, Nirva transforms the output container into XML and sends this XML stream to the renderer engine that generates the HTML code. In this example, Nirva uses the default renderer which is XSLT. The NV_XML_XSL parameter of the URL gives the name of the XSLT style sheet to use.

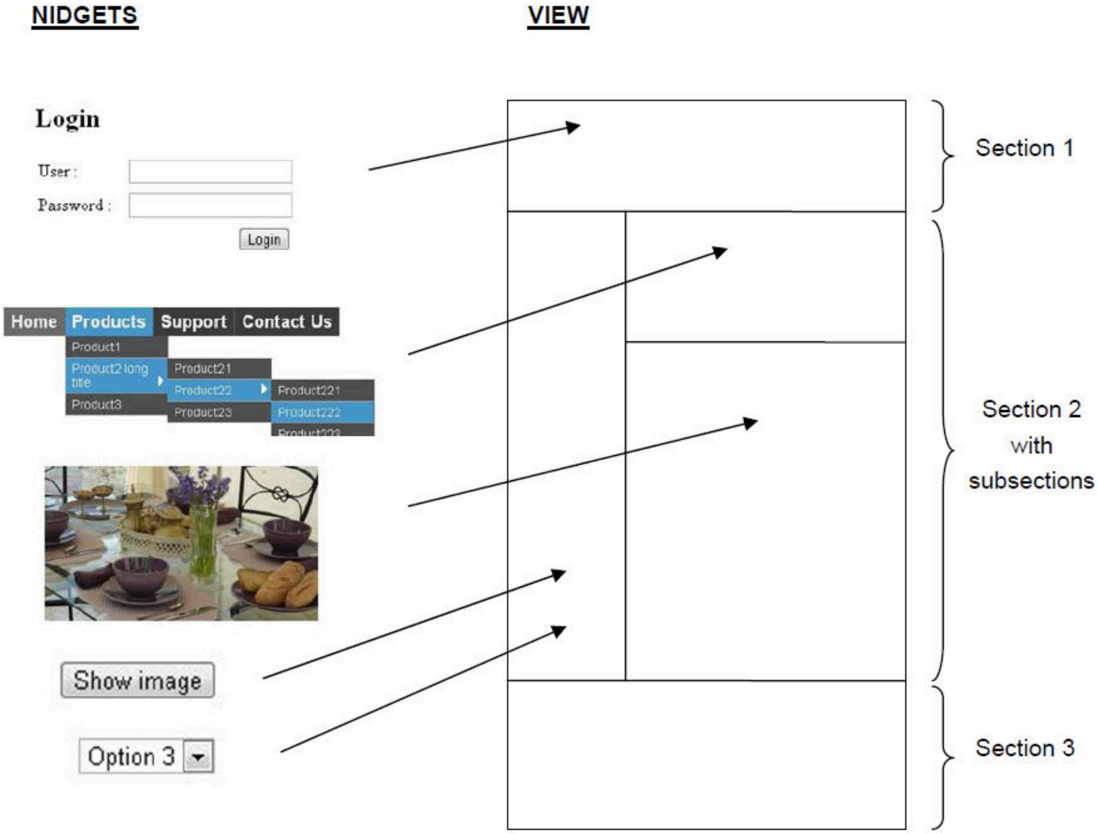
When using the client connectors from external applications, the command cycle is the same except that there is no HTML output. The output container data is directly accessed by dedicated commands.

This technology presents the key advantage of separating the processing part from the presentation part. Development and maintenance are easier.

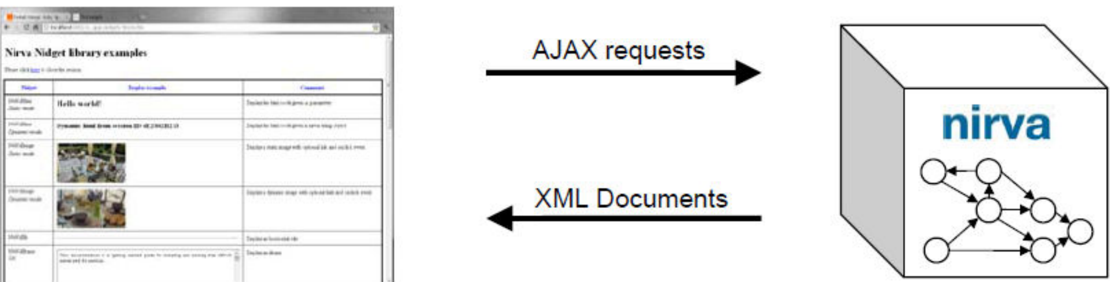
Nidget framework

The Nirva nidget framework is based on JavaScript and Ajax technologies. It is a set of functionality allowing fast creation of Nirva Web applications. “Nidgets” is the contraction of “Nirva” and “Widgets”. A Nidget is a standalone and customizable Web component written in JavaScript. Nidgets are organized in libraries.

The framework defines the concept of view. A view is a Web page divided in sections and subsections like nested tables in an html page. Each cell of the view can be populated with one or several nidgets.



The dynamic data exchange between Nirva and the browser is Ajax based. A view defines one or several XML documents generated by Nirva. Each nidget can be associated to one or several documents. If a document changes, an event is sent to the attached nidgets allowing them to change their display.



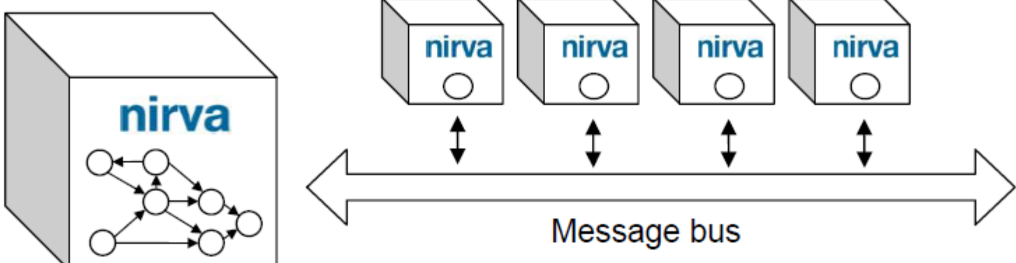
The Html code of the views is kept in a memory cache for being displayed in a very fast way. The code of the view is very short because it only contains the layout and names of the nidgets used in the view. All the real code is inside the nidgets themselves in JavaScript files directly loaded and cached by the browser.

Once compiled and published, the framework code does not change, so the display of Web pages is fast.

Workflow

The Workflow service supports application development with a business perspective in mind. Business logic is distributed in activities that each user can see and organize according to each project's specification.

The service only manages the orchestration of activities. The process itself is handled by specialized agents connected to the message bus. These agents can be Nirva servers or any other application connected to Nirva through one of its available client connectors.



Key features include:

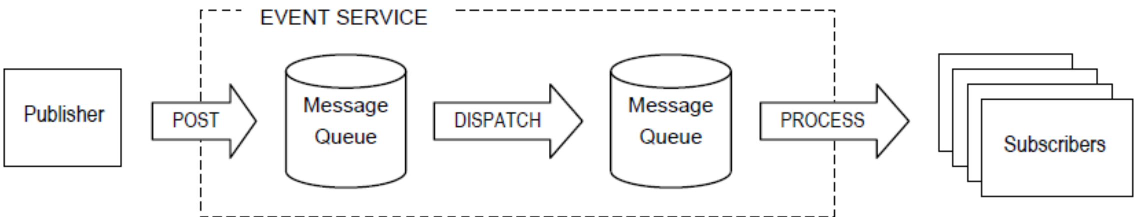
- Creation and deployment of workflows.
- Business logic defined in the activity flow.
- Synchronous and asynchronous activities.
- Process number (workflow cases) unlimited.
- No need for an external database.
- Audit functions supported.
- Business data correlation to identify processes.
- Data storage available to each process.
- Web configuration and management.

Event-driven capabilities

The EVENT service is a Nirva external service which provides Nirva with a message tracking system allowing Event Driven Architectures to be built. It keeps track of messages (events) sent to a channel and the subscribers who are to receive the events. The service allows multiple occurrences of the same messages to be received allowing for escalation mechanisms. It gives the subscribers the possibility of inhibiting further occurrences of a given message.

The EVENT service allows:

- Having multiple channels open to publishing events and event subscriptions
- Processing multiple occurrences of the same messages allowing escalation mechanisms to be configured in case of multiple publications of the same messages (in particular in the case of events being alerts to be processed).
- Management of multiple subscribers per channel and generation of specific actions for each subscriber.
- Inhibiting occurrences of a message on a per-subscriber basis.
- Different possibilities for desynchronizing the posting of an event and its processing.



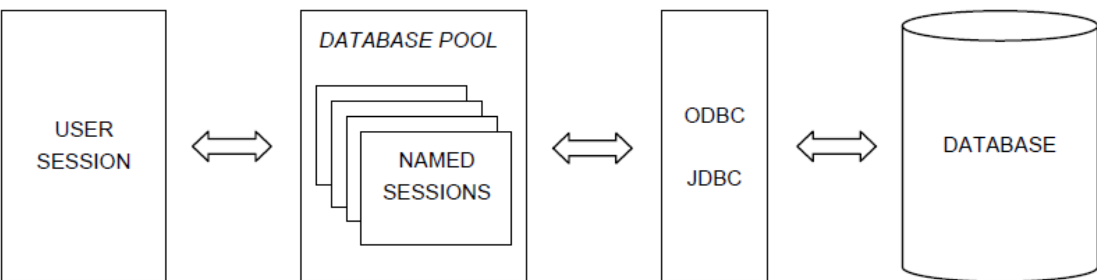
Database

Nirva allows database communication with ODBC and JDBC using dedicated services. Other methods are also possible via alternate Java or .Net components.

Database table data is mapped to Nirva objects allowing easy access.

Nirva provides some features for creating pools of database connections that can be shared by the users.

The SQLite file database engine is directly embedded in the Nirva kernel.



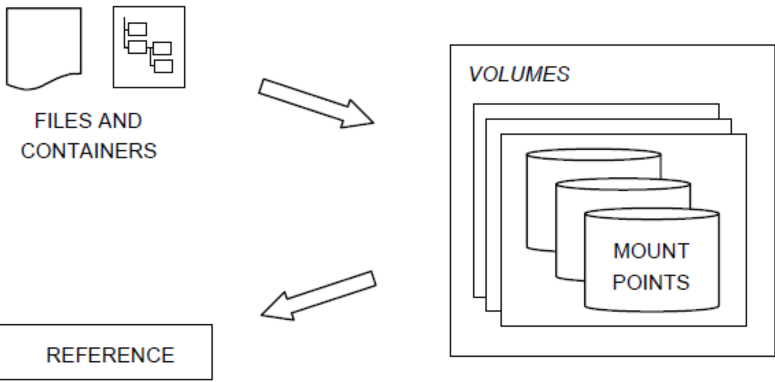
Mass storage

Nirva supplies a mass storage service that can be used to store large amounts of data coming from containers or files.

When storing information, the storage volume returns an identifier that can be recorded with the meta-data in a database and used to retrieve the information.

In the case of storing a container, the service allows retrieval of all or only some objects in the container.

This service can manage logical volumes with one or several physical mount points and supports replication between these mount points.



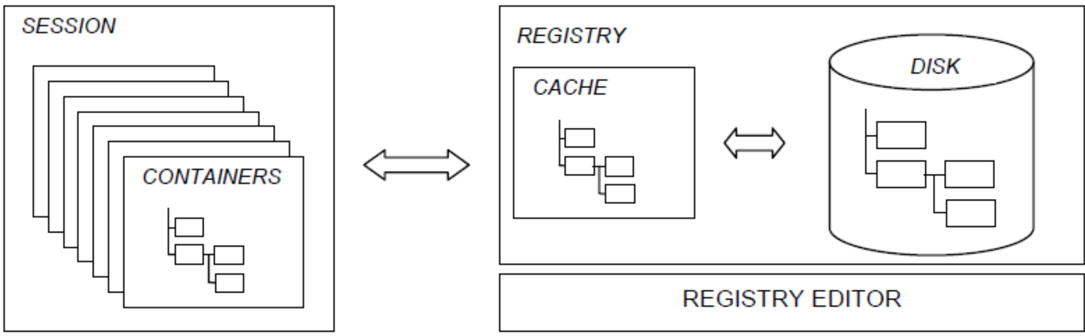
Volumes can be grouped to boost system extensibility and performance.

The mass storage service is usually implemented in failover mode to guarantee business continuity in case of failure.

Registry

A registry system interfaced with a Web editor allows storage of system configuration, application and service data. The registry can also be used to store business data on smaller systems.

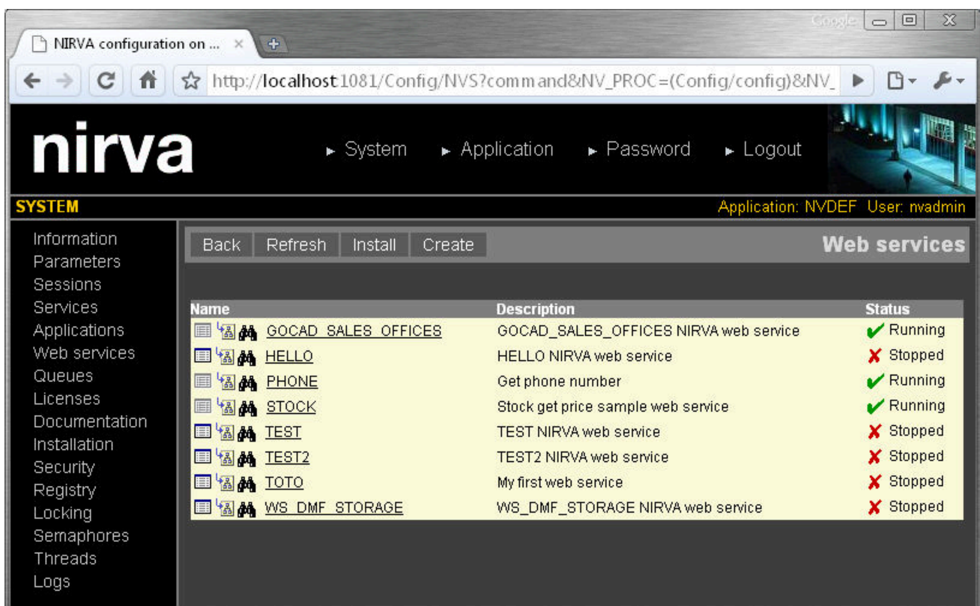
A registry is a persistent container. Nirva provides a registry cache for fast access.



Using the registry saves development time as it becomes unnecessary to create application configuration visual tools.

Configuration

Product configuration is available through a Web interface. All system parameters, but also applications and services can be configured this way.



Configuration can also be automated by using the API from the client connectors. This simplifies processing.

Monitoring

Web based monitoring tools can be used to check system functions.

Logs

Nirva supplies logs to track system, service or application processing. It is possible to add logs to any given application. Nirva controls their size and retention time, and supplies search facilities.

This integrated functionality obviously does not need to be developed for each new application. This significantly reduces development time and overall costs.

Multi-application

A single Nirva instance can host several applications at the same time. Applications remain isolated from one another. Each has its own life cycle but they can still communicate amongst themselves.

This functionality can be used to separate certain parts of a particular project in order to better support their evolution. Alternatively, this can also be used to host several instances of the same application for different users (multi tenancy).

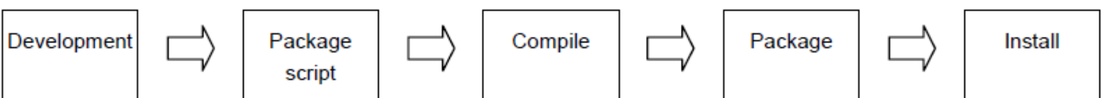
Deployment

Nirva components (application, services, Web Services) can be packaged and deployed in few clicks from a simple Web browser. It is also possible to package only part of components (ex patches).

Nirva provides a simple script file for defining component packages. The default package script is generally enough for packaging components but it can be easily modified to suit requirements.

Once the package script has been created, it can be compiled from a command line, an API or a Web interface to produce a package file.

The package file can then be deployed on the target system.



Multi platform

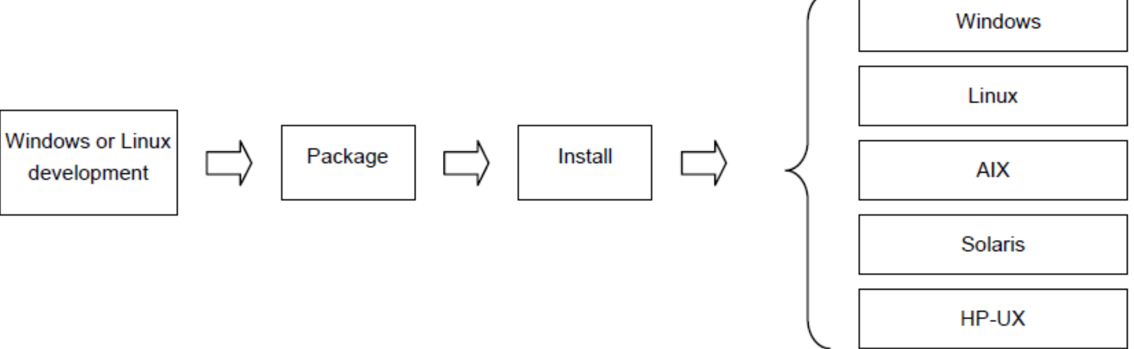
Nirva is compatible with the following platforms:

- Windows
- AIX
- Linux
- Solaris
- HP-UX (RISC and Itanium)

An application can run on any platform (unless it integrates platform specific components, e.g. procedures or services in .Net).

The choice of platforms helps in adapting the product to the customer's strategic choice. This is a key driver for Nirva; it always strives to adapt itself to the existing technology as opposed to imposing its own choices.

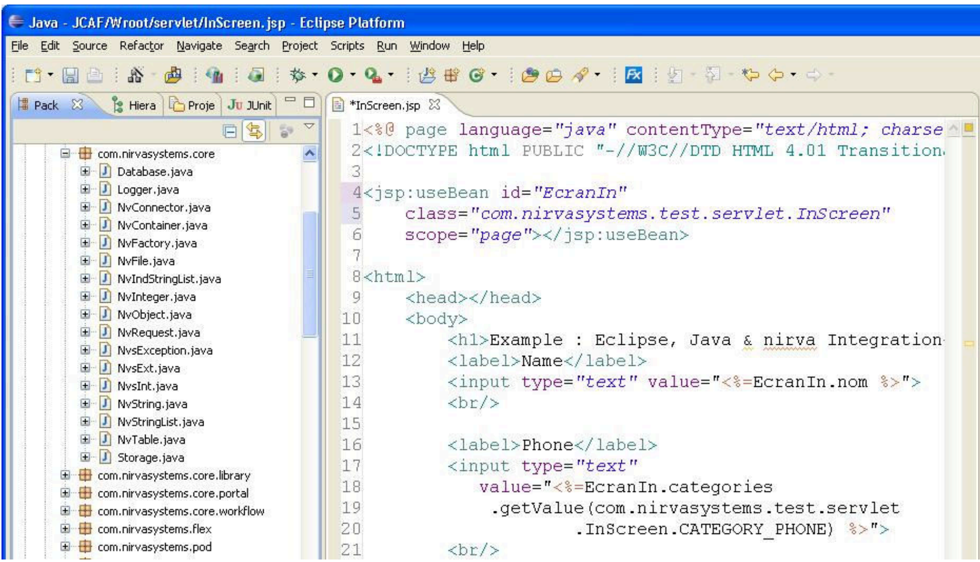
The possibility of running Nirva applications in total independence of the platform guarantees minimal costs. When production imposes expensive environments (e.g. AIX, HP-UX), it remains possible to develop on cheaper platforms.



Development tools

Nirva is relatively independent of the programming language and therefore of the development tools.

Development tools are related to the language used. Generally speaking, Eclipse is used for Java, Perl and C++ development. Visual Studio is most commonly used for .Net and C# Windows development.



Developer’s guide documentation is supplied.

Nirva also supplies some tools to fine-tune, debug and test an application to speed up development and maintenance time.

Nirva components

Server side

Server (nvs)

The nvs server is the heart of Nirva. It receives commands from clients and dispatches them to internal or external services.

It also contains an internal scheduler for running defined tasks at a given time.

Several NIRVA servers can communicate together.

Services

The Nirva services contain the technology that can be accessed by the commands.

Nirva has an internal service that provides basic commands for processing objects, applications, sessions, XML output, registry, external services and many others.

The external services contain third party technology. An external service is simply a dynamic library loaded and maintained by the Nirva server (nvs). The interface to an external service is very simple and well described in this documentation. In this way, third party companies can easily implement their technology on Nirva.

Applications

A NIRVA application is a context of execution of several components. The application is the place where components are arranged to deliver the final functionality to the application server or to the user interface. The application can be considered as the business part while the services are the technology part.

A Nirva application maintains all persistent server contexts. Particularly, the Nirva application manages users and associated security.

A Nirva session always works in the context of an application.

An application contains its own web sites, registry, procedures, cache and eventual XML files.

In this way, the application data and files are completely independent from each other.

Web services

A NIRVA web service is a business component that exposes a set of well defined operations to NIRVA applications and to external applications through standards SOAP and WSDL.

The web service creation is entirely web piloted. Creating a web service with NIRVA requires just few clicks.

NIRVA provides web service deployment and security.

Procedures

The NIRVA procedures allow integrating much functionality at system, application or service level without having to write a complex C++, Dotnet or Java service.

The procedures can be executed before or after each command and can be chained. A procedure can also call other procedures.

A procedure accepts specific parameters.

NIRVA provides 4 kinds of procedures:

- Native procedure is simple text files containing one or several Nirva commands.
- Perl procedures are written in Perl language. NIRVA embed a Perl interpreter so no external Perl installation is necessary. A Perl procedure can callback other commands to NIRVA.
- Java procedures are written in Java language. In fact, a Java procedure is a Java class having a ".class" extension. NIRVA embeds a Java Virtual Machine in its kernel so no external Java installation is necessary. A Java procedure can callback other commands to NIRVA.
- Dotnet procedures are written in any dotnet language (C#, C++, visual basic, etc…). In fact, a Dotnet procedure is a Dotnet class embedded in an assembly file having a ".dll" extension. The Dotnet procedures are available only on windows platform and require the installation of the dotnet environment (3.5 minimum). A Dotnet procedure can callback other commands to NIRVA.

External programs

The Nirva system service implements a command allowing clients to run any kind of server executable program and to directly send parameters to it.

This can be an alternative to external services for implementing server technology in a very fast way.

Client side

Client library (nvc)

The client library called "nvc" provides a simple set of functions for communicating with the Nirva server.

Nvc is a simple dynamic library available on WINDOWS and UNIX platforms that the client application must link with the executable.

The nvc library provides a local service for managing local objects.

Here is the usual way to work with Nirva commands from the client application:

- The client application prepares the objects to process.
- It connects to a Nirva server session or creates a new one.
- It sends the objects to the server into a session container.
- It sends the required commands on the server to process the objects.
- When the server has finished the processing, the client gets back the processed server objects from the resulting session container.
- Then, the client can close the session or do some other server processing.

Command line tool (nvcc)

Nvcc is a simple command line tool allowing sending a set of commands to a Nirva server.

It’s a simple way for integrating Nirva commands into scripts or any third-party applications.

Debug tool (nvd)

The nvd helps developers to discover bugs and reduces the development time.

Connectors

The Nirva client connectors make the real power of the Nirva architecture.

With these connectors, developers can build client applications that communicate to Nirva servers in any kind of language.

In this way, the server technology, including technology built by third-party companies and encapsulated in a Nirva external service, is directly accessible to the most usual development languages for building native and web applications.

Applications

The client application can be built using the Nirva client connectors, the Nirva command line tool (nvcc) or directly in XML on the Nirva server.

Any other application able to send or receive XML or SOAP data can also communicate with NIRVA by the way of its XML, SOAP and web service connectors.

Native

The native client applications can be built on WINDOWS and UNIX platforms with the following Nirva client connectors:

- Dll connector
- MFC connector
- DCOM connector
- Java connector
- Dotnet connector
- Perl connector
- XML connector
- MQ connector
- Web service
- Ajax connector
- Flex connector

Web

For web applications, programmers can use the same connectors than for building native applications but can also directly use Cold Fusion or PHP connectors.

Programmers also have the possibility to build XML/XSLT code that is entirely maintained in a Nirva application context.

At this time, the XML data is produced dynamically by the Nirva server, which also parses this XML data with the application XSLT files in order to produce directly the html files sent to the client browser.

Configuration

The Nirva configuration is entirely accessible from a web browser to authorized users.

Depending on the user application rights, the following items can be configured:

- Server
- Applications
- External services
- Registry
- Logs

Features

Object processing

One of the main purposes of Nirva is to process objects of different kinds. Typically, the client creates and prepares the objects he wants to process, sends them to the server, calls the necessary server commands to process the objects and gets the resulting objects from the server.

Here are the different steps of object processing from a Nirva client:

- Preparing objects
- Connecting session
- Sending objects
- Processing objects
- Retrieving results
- Disconnecting session

Preparing objects

The first step for the client is to prepare the objects he has to process. Let's take an example:

The client has a 3 pdf files that he wants to concatenate. The resulting file must then be sent as an Email attachment to 5 persons and archived into a content repository. The client will finally receive back an archive identifier.

For that, the client will prepare the following objects:

- 3 file objects corresponding to the 3 pdf files.
- 1 string list object containing the Email addresses of the 5 persons.
- 1 string object for the Email subject.
- 1 string object for the Email body text.
- 1 string object for the resulting archive identifier.

For preparing the object, the client uses the Nirva local service of the nvc library using of one of the client connectors.

Connecting session

The second step for the client is to create a server Nirva session or to connect to an existing one.

This operation is very simple. The connection to an existing session is made by giving a session identifier to the commands sent to the server. If this identifier is not provided, the session is automatically created and destroyed after the object processing.

Sending objects

The client uses the Nirva command *SendObject* to send the objects to the server. The objects go to a server session container.

Processing objects

The client sends the necessary commands to the Nirva server for processing the objects.

The server then does the required job and prepares the resulting objects.

In our example, only one resulting object will contain the archive identifier.

Getting results

The client then requests the resulting objects from the server by the use of the Nirva command *GetObject*.

In our example, the client requests the archive identifier string object.

Disconnecting session

The last step for the client is to disconnect from the Nirva server session. This is done automatically when the client has finished its request. The Nirva session can then be also automatically closed if requested.

Objects

Nirva is able to manage different kind of internal objects. Objects are always stored in containers (local or server containers) and can be persistent by the way of the server registries. The objects can be accessed by their name. An object name is case insensitive and cannot contain any of the '/', '\', '.' Or space characters.

Here are the different Nirva objects:

- Boolean
- String
- String list

- Indexed string list
- Table
- File
- Binary

The local and system services provide all necessary commands to manipulate these objects.

Boolean

This is the simplest object. A Boolean object can only take the values TRUE or FALSE.

Integer

An integer is a signed number coded on 32 bits.

String

The string object can contain any kind of ASCII character. There is no special limitation on the string object size except the system memory limitation.

String list

The string list contains a list of strings indexed by a numerical index starting at 1.

Indexed string list

An indexed string list is a list of strings indexed by another string. Each item of the list is composed of a string value and a string key used as index. The string key is case insensitive.

Table

A table is a 2 dimensional array of strings indexed by column and row indexes. The row and column indexes start at 1.

Each cell of the table can itself contain several lines of alphanumeric or numeric data. A numeric data is stored internally as a string and some commands convert this string to a double C type. So the limitation of this numeric data is 52 bits for the mantis, 11 for the exponent and 1 bit for the sign.

Nirva provides many commands for adding, updating, deleting, searching and sorting tables.

A table object can also have a primary key (case sensitive or not) that may be used for fast access to a single row. When there is a primary key, this will be a unique value identifying the row.

File

The Nirva file object maintains a file. 3 kinds of file objects can be used:

The persistent file stays permanently even when the session that owns the file is closed.

The cached file exists for a given time after the file object has been removed (for example 10 minutes after the removing). The Nirva server (nvs) has a cache manager that controls the cached file.

The temporary file is removed automatically when the session is closed.

Binary

The Nirva binary object maintains an array of bytes of any kind.

Containers

The container is one of the most powerful features of Nirva. A container is simply a named entity containing a list of Nirva objects and/or other containers (sub-containers).

Each session maintains its own list of containers, always with a default container. There are also containers at system, service and application levels.

When a command is executed, some parameters tell the server what the input and output containers are so that the service knows where to find the source objects to process and where to create the resulting objects. These input and output containers can point to a container or to a sub-container (by giving the sub-container path). If the container doesn’t exist, it’s automatically created.

There is no limitation on the number of containers managed by a session, the system, a service or an application (except system resource limitations).

A container is maintained in memory but can be saved permanently using the Nirva registry.

The Nirva local service (nvc library) maintains a non-hierarchical container (the local container contains only objects but no sub-containers). This container doesn’t accept cached file objects.

Containers are also used in Nirva commands from web browsers in order to generate XML data automatically.

The structure of a hierarchical container can be compared to a directory structure where each subdirectory corresponds to a sub-container and each file corresponds to an object.

Sessions

The Nirva session keeps the entire session context. This includes the following items:

- A pointer to the connected application (Default application if no specific application has been given).

- A list of hierarchical containers for keeping or exchanging any Nirva object.
- The name of the current connected user
- A list of session variables.

The possibility to keep the session context in a centralized place is a very useful feature for WEB applications. At this time, the web application concentrates only on the user interface and keeps all its data in a Nirva session. The only required parameter to pass to web pages is the Nirva session identifier.

A session has a time out. When the time out occurs or when the session is closed (implicitly or explicitly), Nirva automatically frees any resource owned by the session. NIRVA checks for session time outs every 30 seconds.

A session is identified by a session identifier. If a Nirva command coming from the web or from XML, SOAP, web service and MQ connectors has a valid session identifier as parameter, the command is executed in the context of the given session. Otherwise, a new session is created and kept in memory until it's closed explicitly or by the time out. For web commands, the session ID can be transmitted by a cookie instead of a command parameter.

Nirva distinguishes several session types:

Client session.	This is a session that has been initiated by one of the client connectors including the ones from web browsers.
Named session.	A named session is a shared session that can be accessed and temporary owned by any other kind of session. A named session is a session created explicitly by a dedicated command that is searchable by its name. Several sessions can have the same name allowing the session group management. With session group management, it's possible for example to maintain a stack of sessions having an application scope and that are used on demand by other sessions. For example, one can set 10 sessions that maintain 10 database connections during all a nirva application life.
Scheduled session.	This is session initiated by the Nirva scheduler. Each time an instance of a scheduled task runs, Nirva creates a scheduled session for executing it. This session is automatically removed when the task instance ends.
Transaction session.	Nirva allows to group commands in a transactional context. A NIRVA transaction is simply an application procedure that is executed by a special command named SYSTEM TRANSACTION START. When executing this command, NIRVA saves the initial session context in a persistent place and executes the transaction commands by continuously saving the current context. A transaction can be run in synchronous or asynchronous mode. For running an asynchronous transaction, Nirva creates a dedicated session and removes it after the transaction ends.
Listener session.	This is a session associated with a Nirva listener. A listener session is created explicitly and is removed when the listener ends or when the application that owns it terminates.
Internal session.	This is a session initiated explicitly by the SYSTEM:SESSION:CREATE command.

Thread session.	This is a session initiated explicitly by the SYSTEM:THREAD:CREATE command. A thread session executes a procedure in a separate thread and then closes.
System session.	This is a session initiated by Nirva itself. Nirva creates some temporary sessions for doing some internal management.

Applications

A Nirva session always works in the context of an application. When a client creates a new session, it gives the application name and an eventual user name and password. If the application name is not given, Nirva uses the default application.

Each Nirva application maintains its own environment that is completely independent from other applications. This is the concept of data separation. This concept is very important when a single Nirva server is used to maintain applications for different customers.

Here are the components of a Nirva application:

Containers.	The application has its own containers for maintaining data within an application scope.
Registry.	The application keeps the entire persistent context in a registry. There is one registry for each application. The registry is a persistent storage of a hierarchical container so any kind of Nirva object can be stored in an application registry.
Users.	The application maintains its own user list.
Security.	The Nirva security (access to services and commands) is made at application level.
Documentation.	The documentation of the application is in the application directory.
Procedures.	The application has its specific Nirva procedures that reside in the application procedure directory. Two specific procedures are called respectively when starting and stopping an application allowing startup and cleanup features.
Files.	The persistent, cached or temporary files are also managed at application level.
Web sites.	An application has its own web root directory that can be directly accessed from a web browser.
XML.	The application XML and XSLT files can be stored in the application file directory for building XML applications directly accessible from a web browser.
Locking.	The application maintains a list of locks for controlling access to some global application objects.

Semaphores.	The application maintains a list of semaphores for sharing the access to some limited resources.

Services

All the Nirva accessible technology is contained into Nirva services. The internal technology resides in the system and local services while the external technology is in the external services.

Each service implements some commands accessible from the different Nirva doors. When the user sends a Nirva command, he must give as parameter the name of the service that implements this command.

External

An external service is simply an extension of the nirva code that is loaded and maintained in memory by the Nirva server. The external service is the place where third party companies can implement their own technology, making it accessible to the Nirva client applications. A Nirva service can be written in C++, Dotnet or Java languages.

The interface from Nirva server to an external service is very simple and well described further in this documentation.

Nirva provides a command (SYSTEM SERVICE SKELETON) that automatically creates the source code skeleton for a service.

An external service can communicate directly to the other Nirva external services or to the Nirva system service. This way, Nirva provides a good bridge to establish the communication between different technologies. The external services can also use some centralized technology provided by the Nirva system service like licensing control, logs, etc…

System

The system service is an internal Nirva server service that provides some very useful technology like object management, containers, locking, XML, variables, registry access, etc…

Local

The local service is also an internal Nirva service, but for the client side. It provides the necessary functionality for working with local objects and sending commands to the server.

Procedures

The procedure files are, in their simplest form, text files containing a set of Nirva commands. Each command is on a single line. There are procedures at system, application and service levels. By default, a procedure is considered to be at application level.

When more logic is required into a procedure, this one can be written in Perl, Dotnet or Java language. The Perl interpreter is directly embedded in the NIRVA kernel and there is no need to install Perl. In the same way, NIRVA integrates a Java virtual machine in its kernel with a Java run time environment

Here is an example of native procedure file:

```
; NIRVA procedure 1 for tests
; 03/05/2002


NV_CMD=|OBJECT:CREATE| NAME=|mytable| TYPE=|TABLE| REPLACE=|YES|
NV_CMD=|OBJECT:TABLE_INSERT_COLUMN| NAME=|mytable| COLNAME=|colA|
NV_CMD=|OBJECT:TABLE_INSERT_COLUMN| NAME=|mytable| COLNAME=|col1|
NV_CMD=|OBJECT:TABLE_INSERT_COLUMN| NAME=|mytable| COLNAME=|col2|
NV_CMD=|OBJECT:TABLE_INSERT_COLUMN| NAME=|mytable| COLNAME=|col3|
NV_CMD=|OBJECT:TABLE_INSERT_COLUMN| NAME=|mytable| COLNAME=|col4|
NV_CMD=|OBJECT:TABLE_INSERT_COLUMN| NAME=|mytable| COLNAME=|col5|
NV_CMD=|OBJECT:TABLE_INSERT_ROWS| NAME=|mytable|
NV_CMD=|OBJECT:TABLE_SET_ROW| NAME=|mytable| DATA=|A;1;;3;4;5|
NV_CMD=|OBJECT:TABLE_INSERT_ROWS| NAME=|mytable|
NV_CMD=|OBJECT:TABLE_SET_ROW| NAME=|mytable| DATA=|A2;12;;32;42;52|
NV_CMD=|OBJECT:TABLE_INSERT_ROWS| NAME=|mytable|
NV_CMD=|OBJECT:TABLE_SET_ROW| NAME=|mytable| DATA=|A3;13;;33;43;53|

;NV_CMD=|DEBUG:DISPLAY_MESSAGE| MESSAGE=|Before table_get_row|
NV_CMD=|DEBUG:DISPLAY_OBJECT| NAME=|mytable| ROWS=|1- -1|

;NV_CMD=|OBJECT:TABLE_GET_ROW| NAME=|mytable| MODE=|STRING| NV_VAR=|ZUT|
NV_DEBUG_ONLY=|YES| NV_DEBUG_PARAM=|YES|

;NV_CMD=|DEBUG:DISPLAY_VARIABLES|

NV_CMD=|OBJECT:CREATE| NAME=|toto| TYPE=|STRING| REPLACE=|YES|
NV_CMD=|OBJECT:CREATE| NV_CONTAINER=|nvdef.zut| NAME=|tutu| TYPE=|STRING| REPLACE=|YES|
NV_CMD=|DEBUG:DISPLAY_CONTAINER|
```

A procedure can be called from any Nirva command by giving the procedure name in a dedicated command parameter.

Several procedures can be chained for a single Nirva command.

The procedures can be called before or/and after execution of the command.

A procedure accepts specific parameters.

A procedure contains commands that can call other procedures. This way, a single command can execute a complex hierarchy of other Nirva commands.

Web services

The NIRVA web services are defined at system level. A web service is accessible directly by the web service NIRVA connector from any application able to connect to a standard web service with SOAP and WSDL protocols.

A web service operation can also be launched from a procedure, a service or from any NIRVA connector.

The web service is the place to put business logic to be used into several services or applications.

A web service is a set of operations. For each operation, one defines an input and an output message. These messages are simply NIRVA containers with a fixed structure.

A web service can be packaged and installed by the way of the NIRVA web configuration tool.

A web service is used in the context of a NIRVA application. NIRVA provides security for applications to access web services. The security is based on a permission given to a user for accessing each operation of a web service.

Nidget framework

The Nirva nidget framework is based on JavaScript and Ajax technologies. It is a set of functionality allowing fast creation of Nirva web applications. “Nidgets” is the contraction of “Nirva” and “Widgets”. A Nidget is a standalone and customizable web component written in JavaScript. Nidgets are organized in libraries.

Renderers

When sending a command to Nirva from a web browser, Nirva executes this command, creates XML data from the output container and then uses a renderer to transform this XML data into HTML code. Nirva provides a default embedded renderer that uses XSLT but allows third party programmers to deliver their own renderers (service renderer).

Transactions

Nirva allows to group commands in a transactional context. A NIRVA transaction is simply an application procedure that is executed by a special command named SYSTEM TRANSACTION START.

When executing this command, NIRVA saves the initial session context in a persistent place and executes the transaction commands by continuously saving the current context.

At the end of the transaction commands, NIRVA sets the transaction status to “COMPLETED” or “FAILED”.

A completed transaction can then be validated by a dedicated command and then removed from the transaction list. The validation process is made by calling an external procedure defined with the transaction. The validation allows a user to validate a set of NIRVA commands.

A transaction can also be rolled back or restarted in case of failure.

The starting of a transaction can be scheduled by setting its start time. An embedded transaction server periodically checks for transactions ready to run and then, runs them.

Nirva provides some visual tools for viewing and controlling the application transactions.

Registries

The Nirva registries allow maintaining persistent information at server level. There are two kinds of registries: standard registries and user registries. Standard registries are written in fixed directories in the Nirva disk tree while user registries can be written in a directory given by the user. Standard registries are mainly used for configuration information while user registries are designed to maintain application or service data. The registries can be defined at 3 different levels:

- System registry for system level information. This registry is available to all applications and services.
- Application registry for persistent information specific to each application. Having a separate registry for each application assumes the complete independence between the Nirva applications.
- Service registry is used for external service persistent information.

The Nirva registry is a very powerful registry because it can store in a hierarchical structure any kind of Nirva object. For example, a Nirva table can be stored as a registry entry.

The access to the registry is simple. Only 3 functions are necessary for the registry manipulation:

- The *REGISTRY GET* function copies a branch (or some single entries) of the registry into a session container (or sub-container).
- The *REGISTRY SET* function copies a branch of a session container (or some objects) into the registry.
- The *REGISTRY REMOVE* function removes a branch of the registry (or some single entries).

These 3 functions have also some options to operate on dedicated objects, on a complete branch including sub-containers or not.

The usual way to modify the registry is to call the *GetRegistryKey*, then to modify the necessary entries with the system object commands on the session container and finally to write the result back to the registry by the use of the *SetRegistryKey* function.

The accesses to the registries are serialized automatically by Nirva server, avoiding conflicts. Despite that, it's also possible to use Nirva locking mechanisms (see next chapter) for controlling registry transactional operations when several registry accesses must be controlled in a transactional way.

Nirva maintains a read/write cache for each registry. This cache is automatically cleared every 3 hours for freeing memory resources.

A web registry editor allows accessing registry information in reading and writing modes.

Locking

The Nirva server provides some locking mechanisms for controlling simultaneous accesses to dedicated data or for transactional operations.

A Nirva lock object is a named object having a status locked or unlocked. A lock is owned by a session so only the session who locked a lock object can unlock it.

When a session is closed, all the lock objects it owns are automatically free.

A session who wants to access a lock object can tell the system to wait for the object to be free for a given time, to not wait (return error if the lock is locked) or to wait infinitely.

A string information can be associated to a lock object.

Nirva also provides a function to test the state of a lock object and to return the associated information if there is one.

Nirva maintains lock lists at system and application levels.

Semaphores

The semaphores allow controlling simultaneous accesses to a shared limited resource. It’s a very good tool for tuning very complex applications on a machine with a limited number of resources.

A Nirva semaphore object is a named object having a status locked or unlocked and an initial value. Each time a thread wants to access the resource, the value is decreased. When the value reaches 0, the semaphore object is locked and all threads requesting it are suspended while the value stays at 0. When a thread has got the access to the semaphore, it does its processing and unlocks the semaphore. Unlocking a semaphore means increasing its value by one.

When a session is closed, all the semaphore objects it owns are automatically free.

Nirva maintains semaphore lists at system and application levels.

Variables

A Nirva session manages a list of local variables having a session scope. A variable is a named string object. The system service provides some commands for working with variables.

In fact, the session variables are automatically removed when a command coming from a NIRVA client or a web browser arrives. The session variables are not removed when the command comes from a procedure or a service.

This standard feature can be disabled by using a standard command parameter named NV_KEEP_VAR.

On the client part, the local Nirva library (nvc) also manages a list of variables for a request.

The variables can be used directly as a parameter value on the Nirva commands (see the Nirva command syntax for further information).

A variable name should never start with the '#' character because this character is used as a variable identifier in the Nirva command. A variable name can contain spaces.

Direct access from browser

Since Nirva includes an http server, it can be accessed directly from a web browser. NIRVA also implement the HTTPS protocol.

Each Nirva command can be sent as an URL. By using the procedures and XML capabilities of Nirva, it's possible to directly build client applications in XML.

The whole Nirva configuration tool is made in this way.

XML applications

For each Nirva command sent from a WEB browser, Nirva delivers the output container as an XML data flow. This output container contains the Nirva objects to be displayed. A parameter of the Nirva command also gives the name of an XSLT file that processes the XML data flow to transform it into a displayable HTML file.

The selection of the objects and containers that the command will display in the XML form is made by some dedicated command parameters in the URL. It's possible to select only some of the container objects, to include sub-containers and/or objects data.

Here is an example of Nirva XML output before calling the parser:

http://127.0.0.1:1081/NVS?command&NV_CLASS=misc&NV_COMMAND=Nop&NV_PROC=Proc1 - Microsoft In...

File Edit View Favorites Tools Help

Back Search Favorites History

Address http://127.0.0.1:1081/NVS?command&NV_CLASS=misc&NV_COMMAND=Nop&NV_PROC=Proc1 Go Links


```
<?xml version="1.0" ?>
- <NIRVA session="3DC39CDE00000002" application="NVDEF" user=""
    source="BROWSER" host="127.0.0.1:1081">
    <NVHTTPHEADER key="ACCEPT">*/*</NVHTTPHEADER>
    <NVHTTPHEADER key="ACCEPT-ENCODING">gzip, deflate</NVHTTPHEADER>
    <NVHTTPHEADER key="CONNECTION">Keep-Alive</NVHTTPHEADER>
    <NVHTTPHEADER key="COOKIE">CFID=1;
      CFTOKEN=28133161</NVHTTPHEADER>
    <NVHTTPHEADER key="HOST">127.0.0.1:1081</NVHTTPHEADER>
    <NVHTTPHEADER key="USER-AGENT">Mozilla/4.0 (compatible; MSIE 5.5;
      Windows NT 5.0)</NVHTTPHEADER>
    <NVSESSIONVAR key="ACCEPT">*/*</NVSESSIONVAR>
    <NVSESSIONVAR key="ACCEPT-ENCODING">gzip, deflate</NVSESSIONVAR>
    <NVSESSIONVAR key="CONNECTION">Keep-Alive</NVSESSIONVAR>
    <NVSESSIONVAR key="COOKIE">CFID=1;
      CFTOKEN=28133161</NVSESSIONVAR>
    <NVSESSIONVAR key="HOST">127.0.0.1:1081</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_APPLICATION">NVDEF</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_CLASS">MISC</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_COMMAND">NOP</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_CONTAINER" />
    <NVSESSIONVAR key="NV_IN_CONTAINER" />
    <NVSESSIONVAR key="NV_OUT_CONTAINER" />
    <NVSESSIONVAR key="NV_PROC">Proc1</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_SERVICE">SYSTEM</NVSESSIONVAR>
    <NVSESSIONVAR
      key="NV_SESSION_ID">3DC39CDE00000002</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_USER" />
    <NVSESSIONVAR key="USER-AGENT">Mozilla/4.0 (compatible; MSIE 5.5;
      Windows NT 5.0)</NVSESSIONVAR>
  - <NVCONTAINER name="nvdef">
    - <NVOBJ name="toto" type="FILE">
        <NVNAME>toto.txt</NVNAME>
        <NVEXTENSION>txt</NVEXTENSION>
        <NVDIRECTORY>C:\Nirva\Applications\NVDEF\Work</NVDIRECTORY>

          <NVPATHNAME>C:\Nirva\Applications\NVDEF\Work\toto.txt</NVPATHNAME>
        <NVSIZE>21</NVSIZE>
      </NVOBJ>
    - <NVOBJ name="titi" type="INDSTRINGLIST">
        <NVDATA key="EMPTY" />
        <NVDATA key="KEY1">VALUE1</NVDATA>
        <NVDATA key="KEY2">VALUE2</NVDATA>
```


Done Internet

Web sites

Any standard web site can be published directly on Nirva at system, application or service level.

Here is an example for accessing an application level web site:

For an application web site, the “NV_APP_” string must follow the server address in the URL and must be followed by the application name (here it’s NVDEF) and the wanted page.

Each application has its own web root directory were Nirva will search for requested web pages.

For example, the URL http://127.0.0.1:1081/NV_APP_MYAPP/Test/Test.htm will physically correspond to c:\Nirva\Applications\MYAPP\Wroot\Test\Test.htm if Nirva is installed in c:\Nirva.

For a service web site, the “NV_SRV_” string must follow the server address in the URL and must be followed by the service name and the wanted page.

Each service has its own web root directory were Nirva will search for requested web pages.

For example, the URL http://127.0.0.1:1081/NV_SRV_PLANET/Test/Test.htm will physically correspond to c:\Nirva\Services\PLANET\Wroot\Test\Test.htm if Nirva is installed in c:\Nirva.

For accessing a system level web site the “NV_APP_” or “NV_SRV_” string must be omitted. For example the URL http://127.0.0.1:1081/Test/Test.htm will physically correspond to c:\Nirva\ Wroot\Test\Test.htm if Nirva is installed in c:\Nirva.

If the web site contains a style sheet file named “error.css”, this one will be used for error message reporting.

Configuration

The entire Nirva configuration is made using the XML features of Nirva, so the configuration is entirely web made.

On line documentation

The Nirva documentation is available on line from a web browser.

Logs

Nirva provides some complex log features. Logs exist at system, application or service levels.

The logs can be consulted directly from a web browser and search facilities are provided.

A NIRVA log is a flat file with a proprietary format (textual format with some specific separators) that contains log records. A log record itself is composed of the following information:

- DATE is the record write date (format YYYYMMDD). The Date is automatically set by the system.
- TIME is the record write time (format HHMMSS). The Time is automatically set by the system.
- IDENTIFIER is a free text that identifies a user entity. For example, for the system log, the identifier is the NIRVA session identifier.
- TYPE is a numeric code that defines the type of the record. It can be 0 for an information record, 1 for a warning record and 2 for an error record.
- MESSAGE is the log message itself. It's textual information.
- EXTRA is an optional second message. It can be used to separate some information from the message itself. It's textual information.
- SOURCE is a free text that can be used to give information about the origin of the log record (who has written the record?).
- LEVEL is the log level. It's numeric information that takes values from 1 to 6. It is the responsibility of the log writer to manage the log levels. The NIRVA logs search commands allow specifying the maximum log level to search for.

A log record is limited to 65536 characters.

NIRVA provides some functionality to limit the size of log files and to save them in a dedicated directory structure. This directory structure is date based allowing easily retrieving the log information for a date range.

All the logs can be controlled and searched from the NIRVA configuration tool. Please refer to the configuration chapter for further information.

By default Nirva provides the following logs:

- SYSTEM log. The system log cannot be removed. It displays information at system level... Following the chosen level, it displays the start of each command, the end of each command, the command errors and the command parameters. When logging command parameters, any parameter starting with “PASS”, “PSW” or “PWD” is discarded from the display. This is to avoid logging passwords.

- WEBFILES log. Logs access to web pages. This log can be removed but Nirva will re-create it automatically at start time. At level 1, it logs all files accessed from the web except gif, jpeg or css files. At level 2, it logs all files.
- WEBSERVICE. Logs all web service operations.
- SESSIONS. Logs all open and close of sessions indicating application and user names.
- SCHEDULER. Logs all scheduler tasks.
- THREADS. Logs all Nirva threads.
- PERL. Logs perl interface

Crash condition

Nirva detects if its last stop was a correct stop or a hard stop (power off or hard kill).

It then sets a crash condition flag that can be used by services to do their own cleanup at initialization time.

Cluster and load balancing

Nirva can be run in failover cluster mode allowing a pair of Nirva servers to run in active / passive mode.

If both servers share the same registry, there is no need of synchronization between servers.

For load balancing environments, Nirva always delivers the session ID as an http header helping load balancing hardware or software to maintain the connection based on this session ID. This HTTP header is named "Nv_Session_Id".

For multi-server architectures, Nirva provides communication layers between different nirva servers with the possibility to define virtual hosts pointing to several real hosts in load balancing or failover mode.

Scheduler

Nirva provides an internal scheduler allowing running some defined tasks at given time.

A task contains some parameters for controlling date, time, repetition, status, procedure to run, error procedure, success procedure, etc…

A task is created in session context and runs in a separated thread.

Listeners

Nirva listeners are threaded sessions that have the application life and call a procedure in a loop. After each execution of the procedure, the listener waits for the given sleep time. Several threads can be defined for a single listener allowing controlling the power given to some Nirva processing.

Listeners are good candidate for processing the Nirva workflow queue activities.

Listeners are attached to application so each application maintains its own list of listeners.

Triggers

Nirva provides triggers that operate when certain commands are received. The triggers simply launch Nirva procedures. For example, a trigger can be installed when a user opens a new session in order for an external auditing system to record the event.

The triggers can operate before or after the command they are attached to. If there are some procedures in the command, the pre-triggers operate before the pre- procedures and the post-triggers operate after the post-procedures.

The triggers are configured with the NIRVA configuration tool. NIRVA provides the following default triggers at application level:

- session_open is a pre-trigger called when a new session is opened. Then NIRVA tries to run the session_open procedure that resides in the application procedure directory. If the procedure doesn’t exist, NIRVA doesn’t generate any error message.
- session_close is a post-trigger called when a session is closed. Then NIRVA tries to run the session_close procedure that resides in the application procedure directory. If the procedure doesn’t exist, NIRVA doesn’t generate any error message.
- init is a pre-trigger called when an application starts. It allows initializing some data having an application scope. The init procedure must be in the application procedure directory. If the procedure doesn’t exist, NIRVA doesn’t generate any error message.
- exit is a post-trigger called when an application stops. The exit procedure must be in the application procedure directory. If the procedure doesn’t exist, NIRVA doesn’t generate any error message.

Also some triggers can be set when an error occurs.

Mail

Nirva allows sending e-mails with attachments from a simple command. This is very useful when used with scheduler or scheduled transactions in order to report errors by mail to the administrator.

The mail features requires an external SMTP server.

Licensing

Nirva provides functionality for license control for external services. Third party companies can directly use Nirva license commands to control their own licensing policy.

Security

The security model is a role based access control model with hierarchical roles. Each application has its own security but can also use the security of another application (even on a separate server) or the system security.

Nirva also allows using an external security system defined in a dedicated security service (ex LDAP).

The user authentication can be made using Nirva security (internal or external) and/or Single Sign-On with Kerberos and NTLM.

The security model is described in the "Security model" chapter in this documentation.

Unicode

Nirva supports Unicode. In fact, NIRVA can manage all its internal data into UTF-8 or ISO-8859-1 (Latin1) character sets. The option is chosen when starting the server (-u option to start it as Unicode).

Nirva accepts and delivers all data pertaining to these 2 character sets, converting the data from one character set to the other when necessary or on demand.

MQ connector

Nirva provides a powerful connector to IBM WebSphere MQ messaging transport product. This allows asynchronous communications to Nirva. It's possible to define several listening queues and for each of them the number of threads that listen.

The MQ connector processes DATAGRAM and REQUEST MQ messages. The message themselves can contain, XML, SOAP, files or any binary data. The message structure allows sending Nirva command parameters.

Debug

Nirva provides some debug command and facilities to display the content of containers and other session context in the console and in dedicated XML files. At any time a part of the session context can be sent to XML files and displayed with a dedicated tool of the nirva configuration application.

Installation

Download Nirva

The Nirva software is available for download in http://nirva-systems.com/NAP/Downloads/downloads.html.

Download the version corresponding to the target platform, unzip the file and put the unzipped files on a temporary directory of the target platform.

Installing NIRVA server

The installation of NIRVA is quite simple but is different on WINDOWS and on UNIX.

On any platform, the NIRVA server must be stopped in order to re-install or update.

The user who is doing the installation must have administrator rights.

On Windows, the installation may fail if some programs are using the library nvc.dll. Please stop these programs if it’s the case.

Windows installation

Run the program nvinstall.exe.

Follow the instructions. The installation program creates all necessary files and directories in the given target directory. It also detects if the installation is an update or a new installation. The target directory itself is created by the installation program but only at the lowest level. For example, if the target directory is “c:\Myprgms\Nirva”, the installation program will fail if the directory “c:\Myprgms” does not exist.

The install program then creates or modifies the environment variables NIRVA and PATH and installs the server as a windows service. The NIRVA windows service is named “Nirva server” and can be found after the installation in the list of available windows services. The start-up mode is set to “Manual”. If necessary, change it to automatic in order to automatically start NIRVA in service mode when starting the computer.

If necessary, the installation program requires a computer restart in order to complete the installation.

On some windows OS the Microsoft redistributable package for Visual C++ 2015 should be installed before running Nirva. This package is delivered with the Nirva installation files or it can be found at:

https://www.microsoft.com/en-us/download/details.aspx?id=48145

For Nirva versions until 4.9.001 one must install the Visual C++ 2008 redistribuable package. This package can be found at:

http://www.microsoft.com/en-us/download/details.aspx?id=29 for x86

http://www.microsoft.com/en-us/download/details.aspx?id=15336 for x64

Unix installation

Under AIX, it is recommended to run the command “`slibclean`” at this step. This removes the unused shared libraries from memory.

Under HPUX, Nirva requires the use of the LD_PRELOAD_ONCE feature. This is available for HP-UX 11.11 (v1) on PA-RISC as a patch PHSS_26560. This patch is not needed for more recent versions than v1.

Under Linux, the libstdc++ version 6.0.17 (at least) must be installed. Nirva also requires glibc 2.11.3 minimum.

Run the command “`sh nvinstall`” from your temporary directory.

Follow the instructions. The installation program creates all necessary files and directories in the given target directory. It also detects if the installation is an update or a new installation. The target directory itself is created by the installation program but only at the lowest level. For example, if the target directory is “/usr/Myprgms/Nirva”, the installation program will fail if the directory “/usr/Myprgms” does not exist.

If necessary, the installation program requires you to update or create the environment variables NIRVA and PATH in the profile of the user that will run NIRVA.

The NIRVA environment variable must point to the NIRVA installation directory (for example “/usr/nirva”) and the Nirva\Bin directory must be added to the PATH variable (for example “/usr/nirva/Bin”).

The Nirva installation package also includes some shared libraries:

- libnvc.a for AIX, libnvc.so for LINUX, SOLARIS and HPUX Itanium, libnvc.sl for HPUX PA-RISC.
- libnvperlcore.a for AIX, libnvperlcore.so for LINUX SOLARIS and HPUX Itanium, libnvperlcore.sl for HPUX PA-RISC.

These library files can be found in the Nirva/Bin directory.

These libraries must be in the library path. Two options are available:

- Copy the libraries in a usual place where libraries can be found (/usr/lib for example).
- Modify the LD_LIBRARY_PATH (or LIBPATH under AIX) and SHLIB_PATH (for HPUX only) environment variables in the profile of the user in order to point this variable to the Nirva/Bin directory. This is the preferred solution since the philosophy of NIRVA is to keep everything in its own directories and nothing outside.

Uninstall

Here is the procedure to completely remove a NIRVA installation:

- Stop the NIRVA server.
- Under windows, go into the NIRVA Bin directory and run the command `nvs -r` from the command prompt.
- Remove the entire NIRVA installation directory.
- Remove the environment variable NIRVA and the eventual path to the NIRVA Bin directory from the PATH environment variable.
- Under windows, NIRVA doesn't write anything into the registry because NIRVA has its own registry.

License

Nirva Application Platform is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Nirva Application Platform is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Nirva Application Platform. If not, see http://www.gnu.org/licenses.

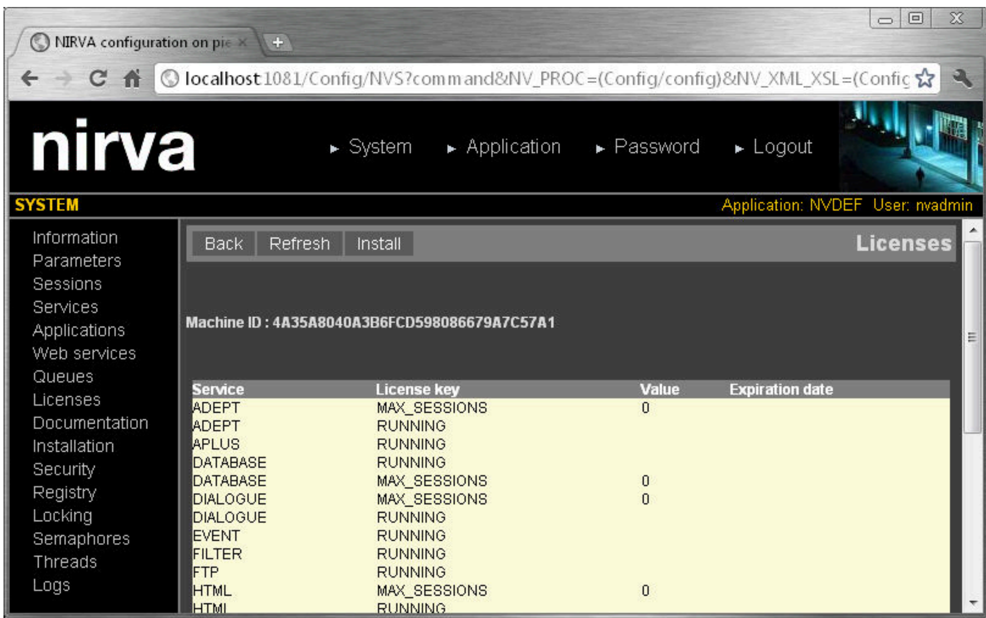
Only the Nirva application Platform is covered by the LPGL license. Applications, services or other external components have their licensing policy under the responsibility of their provider.
Nirva application platform has itself some parts that are not covered by the Nirva LGPL license. These parts are: WebSphere MQ connector, virtual printer connector, license library (nvlc), licensing tool (nvl), contents of the Nirva Perl directory, contents of the Nirva Java directory, libxml, libxslt, openssl. Please consult these

components' documentations for their licensing policy.

NIRVA server is free to use but some of the services may require a license from their providers. If the service you are using requires a license, you must provide the Nirva uniq machine ID to the service provider. The service provider will then send you back a license file that you can install with the Nirva configuration tool.

You can get the unique machine ID by running the configuration tool and go to the system/license menu. If you are using NIRVA for the first time, you can enter "nvadmin" as user name and "nirva" as password:

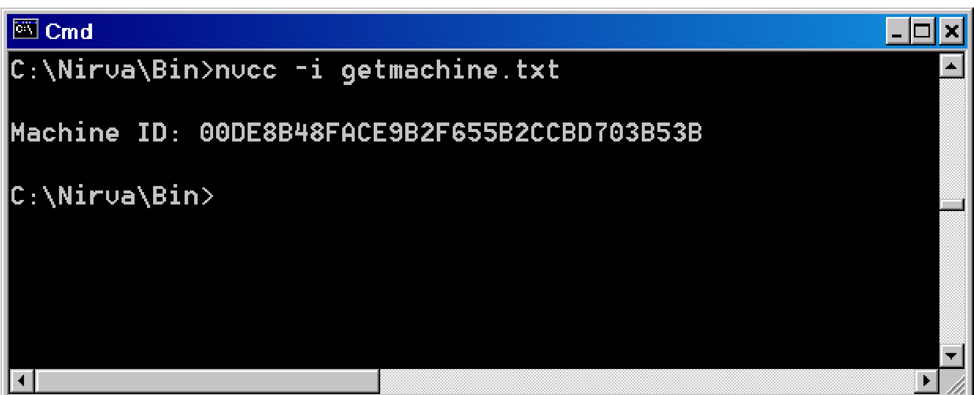
http://localhost:1081/Config/login.htm



You can also run the following command line from your NIRVA Bin directory:

nvcc –i getmachine.txt

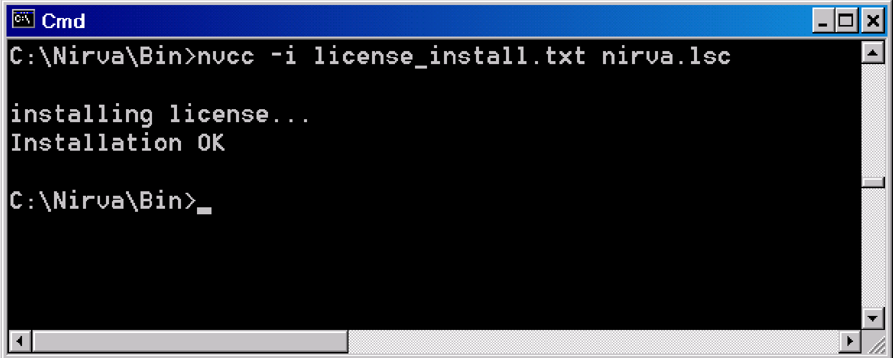
This should display the following screen:



When your NIRVA distributor returns you the license file, you can install it by running the following command from your NIRVA Bin directory:

nvcc –i license_install.txt licensefilename

Where licensefilename is the name of the license file. Here is an example with a file named "nirva.lsc".



You must have enough rights to run this previous command otherwise it returns you a security error message. If it's your first nirva installation and if you didn't change the initial nirva administrator password yet, the command must be the following:

```
nvcc -u nvadmin -w nirva -i license_install.txt licensefilename.
```

You can also install the license file from the NIRVA configuration tool in the system/license menu.

Getting started

This chapter gives a step by step example for building a simple “Hello world” application with NIRVA.

It requires that nirva has been properly installed on the local computer and is not currently running. The administrator user named “nvadmin” is supposed to have the default password “nirva”.

The example can run on Windows or Unix platforms. The description is made with screenshots for a windows platform.

There are 4 steps for the hello world application:

- Starting the server
- Creating the HELLO application
- Testing from web browser
- Testing from batch

Starting the server

After the installation, go into the Bin directory of the nirva installation directory (c:/nirva/Bin if nirva is installed on c:/nirva) and run “`startnv`”. This should start nirva in console mode and display the following screen:

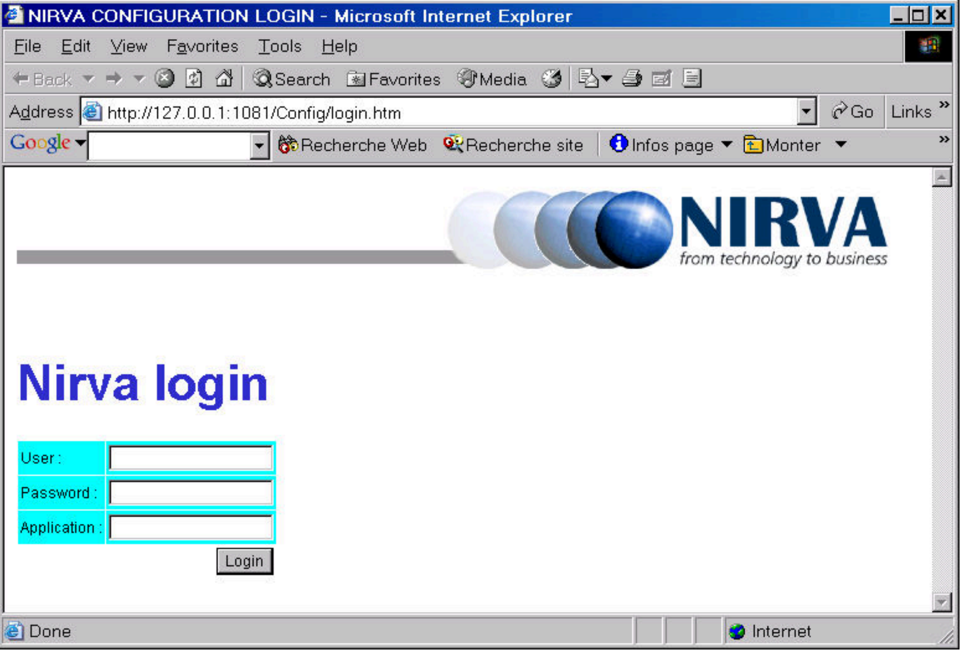
```
C:\WINNT\system32\cmd.exe

C:\Documents and Settings\Administrator\Desktop>c:\Nirva\Bin\nvs -c

NIRVA server started
Get NIRVA environment variable...
Initialize registry...
Starting system log...
Test crash condition...
Checking licenses...
Loading system parameters...
Initialize XML...
Loading java VM...
Loading services...
Loading system security...
Loading applications...
Starting transaction server...
Starting scheduler...
Starting HTTP server...
Starting HTTPS server...
NIRVA server ready
```

Creating the HELLO application

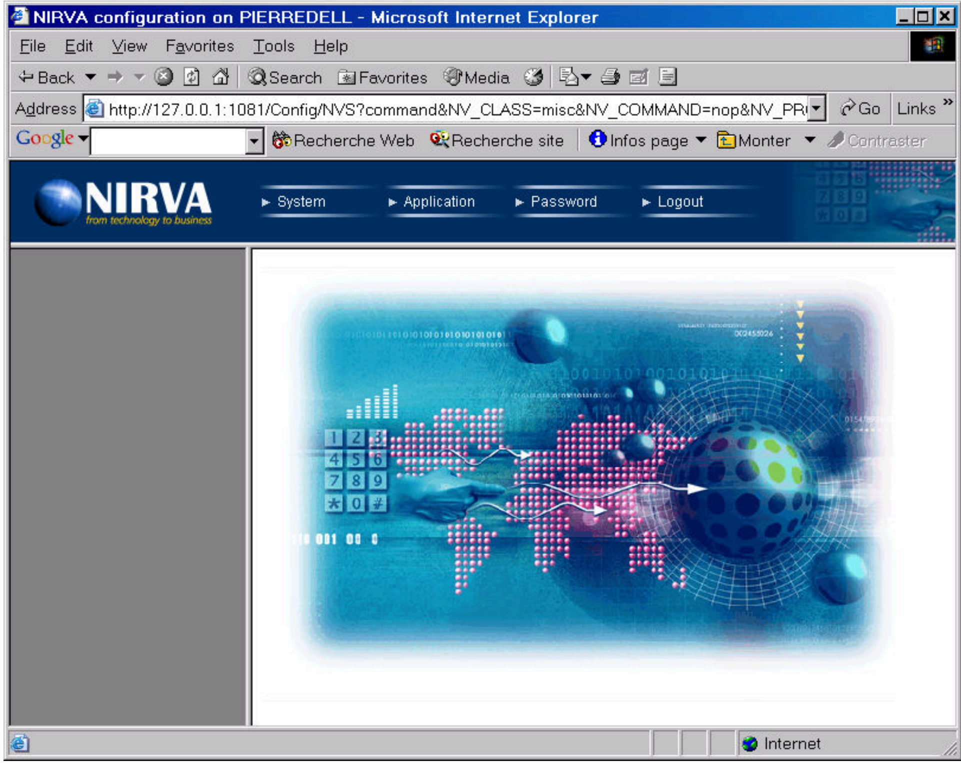
Now run the NIRVA configuration by typing the following URL from your web browser:

http://127.0.0.1:1081/Config/login.htm

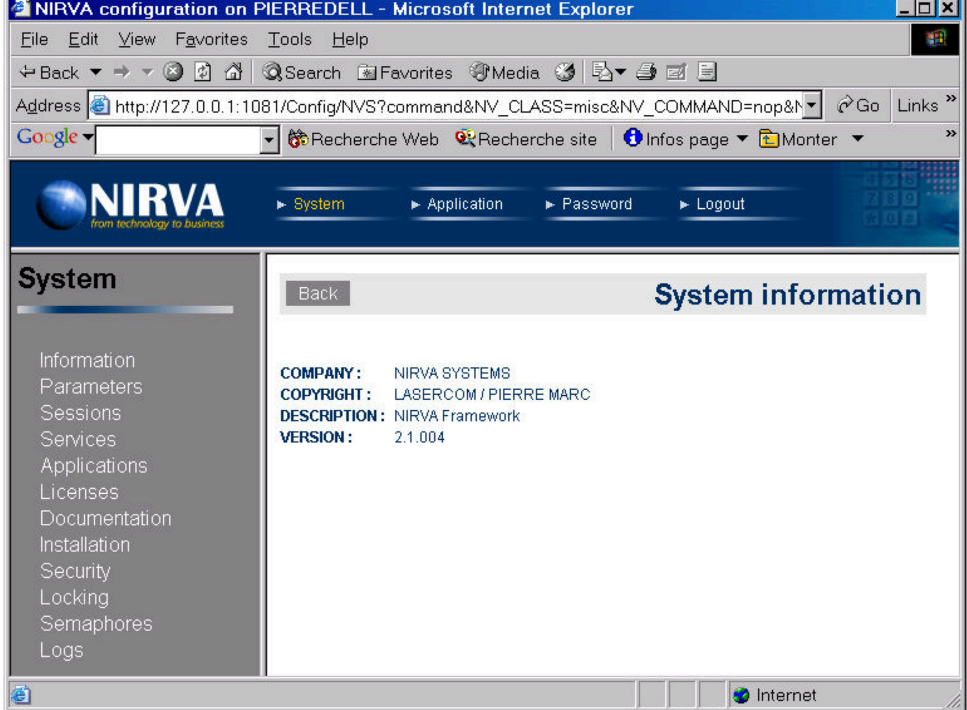
This should display the NIRVA login screen:



Then type “nvadmin” as user name and “nirva” as password (or your nvadmin password if you have modified it). Then the following screen is displayed:



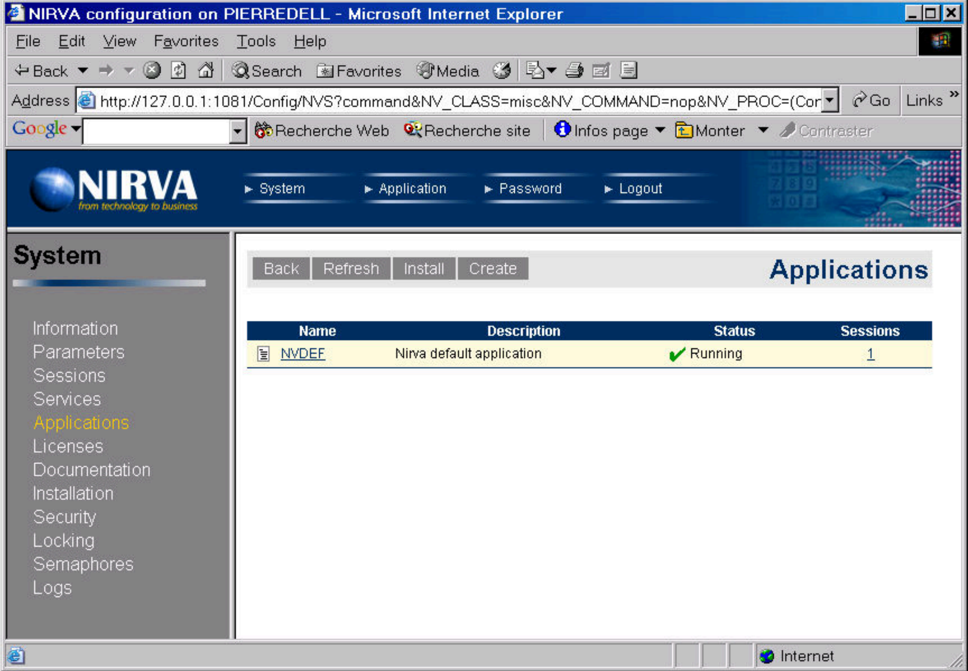
Click on the “System” link in the main menu. The system menu is then displayed:



Choose the “Application” option in the System menu on the left side of the screen.

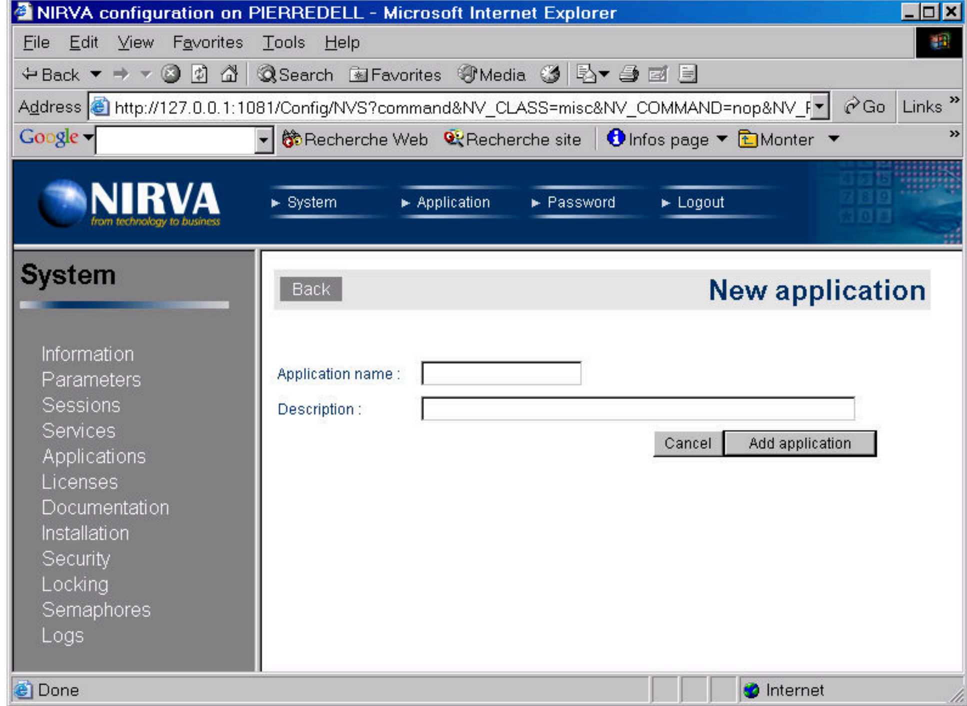
Be careful to choose the “Application” link on the left site of the screen and not the application link found on the main menu. This last is for accessing configuration of an existing application while the “Application” link of the system menu allows creating applications.

This should display the following screen:



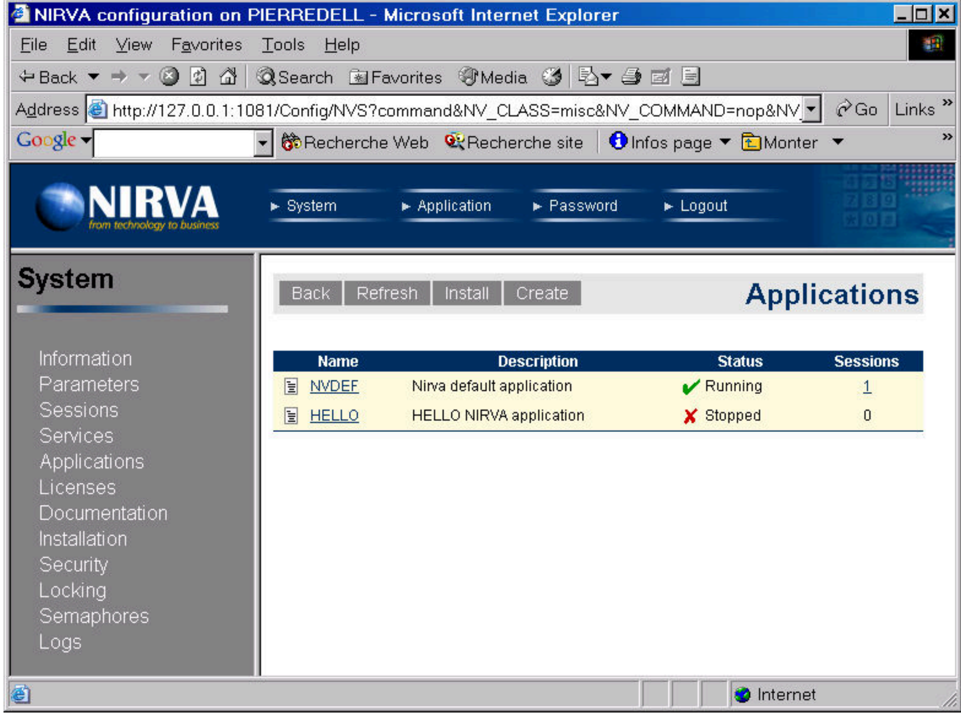
This is the list of current NIRVA applications. There is always one application named NVDEF that is the default NIRVA application.

Now press the “Create” button:



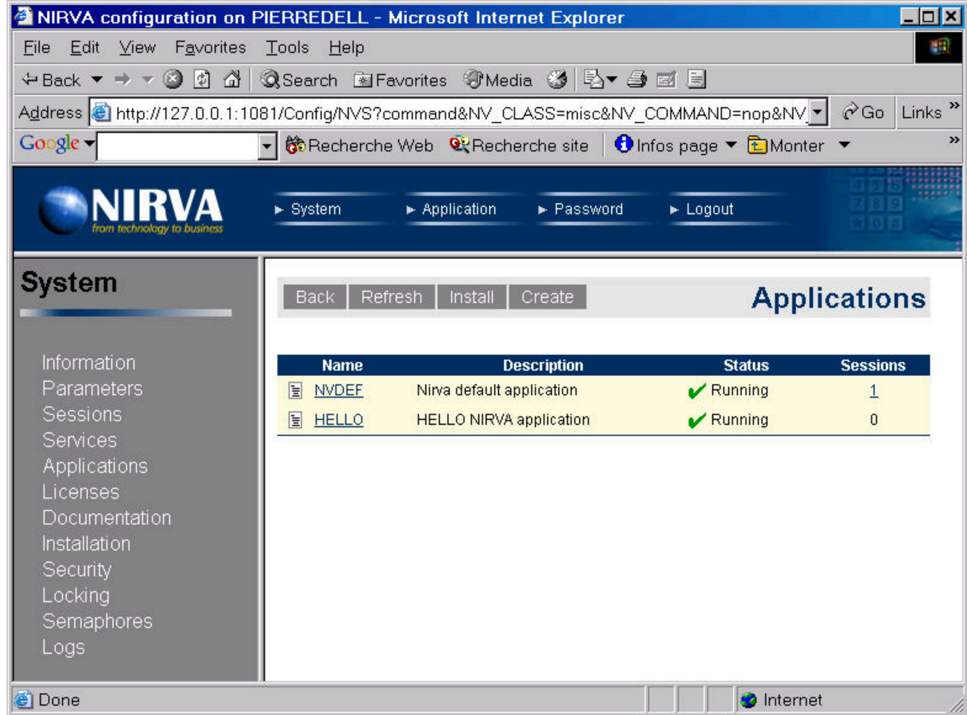
Type “HELLO” as the application name (without the double quotes) and press the “Add application” button.

Then your screen displays the list of applications containing your new HELLO application:



The application is now created but is not running. For starting it, just click on the red cross near the “Stopped” status of the HELLO application.

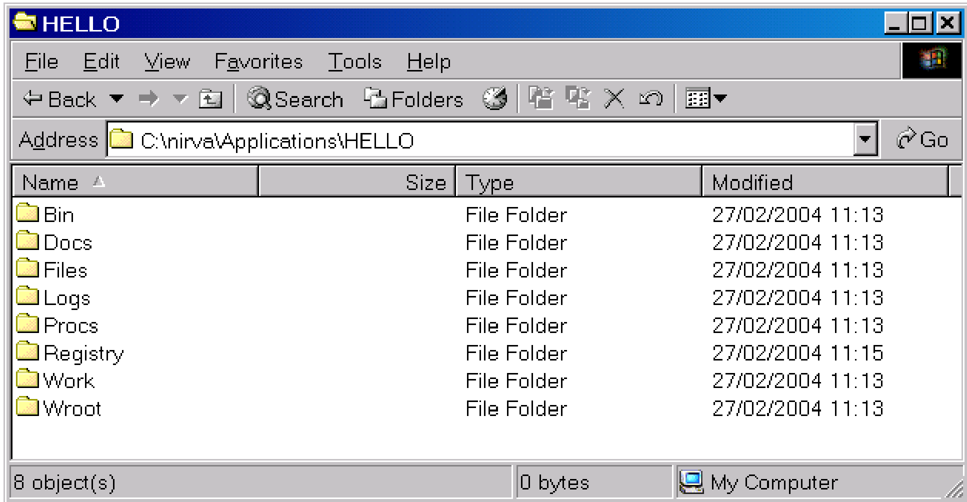
Your application is then ready for use:



You can then leave the NIRVA configuration tool by clicking on the “Logout” link in the main menu.

The HELLO application doesn’t do anything for now so we are going to create a string object into it that contains the “Hello world!” value.

After the application creation, NIRVA has created a specific directory for it in the NIRVA application subdirectory (c:/nirva/Applications/HELLO if NIRVA has been installed on c:/nirva). This HELLO directory itself contains some other directories specific to the HELLO application:

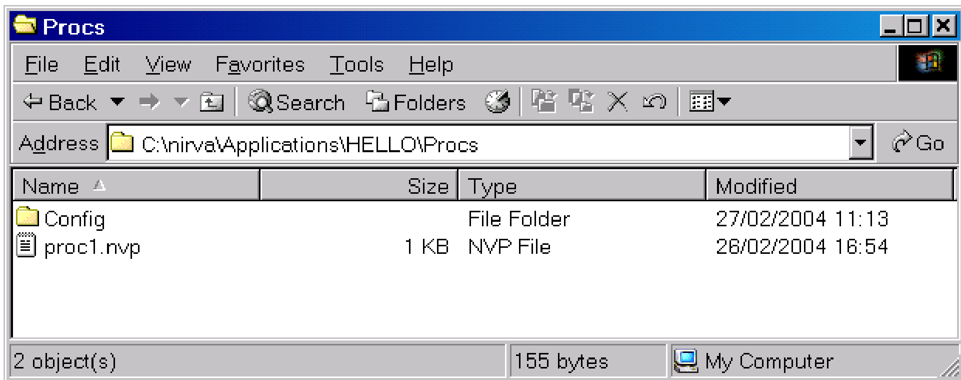


Now go into the “Procs” subdirectory of this HELLO directory and create and edit a file named “proc1.nvp” containing the following NIRVA commands:

```
NV_CMD=|OBJECT: CREATE| NAME=|MYSTRING| TYPE=|STRING|
NV_CMD=|OBJECT:STRING_SET_VALUE| NAME=|MYSTRING| VALUE=|Hello world!|
```

Please respect the syntax.

Your subdirectory Procs should then look like this:



You have then created a simple NIRVA procedure in native language. A NIRVA procedure is the place to create the logic and to arrange NIRVA components for building the business layer.

The NIRVA procedure can be written in native, Perl, Dotnet or Java language.

We are now ready to test our application.

Testing from browser

For testing our new application, just run the following URL from your web browser:

http://127.0.0.1:1081/NVS?command&NV_APPLICATION=Hello&NV_CLASS=misc&NV_COMMAND=nop&NV_PROC=proc1&NV_CLOSE_SESSION=yes

This tells NIRVA to open a new session for the hello application, to execute the proc1 procedure and to close the session.

Any NIRVA command sent from a web browser returns an xml view of what is called “output container” into NIRVA language. The output container is a hierarchical data structure where the application stores any king of data in the context of a session.

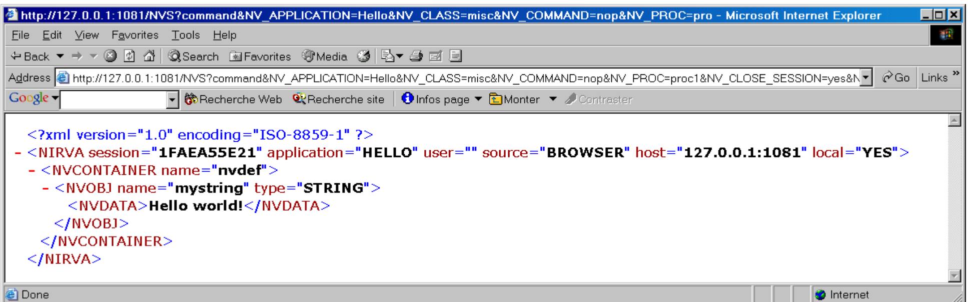
The result is the following screen:

```
http://127.0.0.1:1081/NVS?command&NV_APPLICATION=Hello&NV_CLASS=misc&NV_COMMAND=nop&NV_PROC=pro - Microsoft Internet...
Address 081/NVS?command&NV_APPLICATION=Hello&NV_CLASS=misc&NV_COMMAND=nop&NV_PROC=proc1&NV_CLOSE_SESSION=yes

  <?xml version="1.0" encoding="ISO-8859-1" ?>
- <NIRVA session="E28BC527E9" application="HELLO" user="" source="BROWSER" host="127.0.0.1:1081"
    local="YES">
    <NVHTTPHEADER key="ACCEPT">image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
      application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, application/x-
      shockwave-flash, */*</NVHTTPHEADER>
    <NVHTTPHEADER key="ACCEPT-ENCODING">gzip, deflate</NVHTTPHEADER>
    <NVHTTPHEADER key="ACCEPT-LANGUAGE">fr</NVHTTPHEADER>
    <NVHTTPHEADER key="CONNECTION">Keep-Alive</NVHTTPHEADER>
    <NVHTTPHEADER key="HOST">127.0.0.1:1081</NVHTTPHEADER>
    <NVHTTPHEADER key="USER-AGENT">Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR
      1.0.3705; .NET CLR 1.1.4322)</NVHTTPHEADER>
    <NVSESSIONVAR key="ACCEPT">image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
      application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, application/x-
      shockwave-flash, */*</NVSESSIONVAR>
    <NVSESSIONVAR key="ACCEPT-ENCODING">gzip, deflate</NVSESSIONVAR>
    <NVSESSIONVAR key="ACCEPT-LANGUAGE">fr</NVSESSIONVAR>
    <NVSESSIONVAR key="CONNECTION">Keep-Alive</NVSESSIONVAR>
    <NVSESSIONVAR key="HOST">127.0.0.1:1081</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_APPLICATION">HELLO</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_CLASS">MISC</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_COMMAND">NOP</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_CONTAINER" />
    <NVSESSIONVAR key="NV_HOME_APPWORKDIR">C:\nirva\</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_HOME_LOGDIR">C:\nirva\</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_HOME_REGDIR">C:\nirva\</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_HOSTNAME">PIERREDELL</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_IN_CONTAINER" />
    <NVSESSIONVAR key="NV_NIRVADIR">c:\nirva\</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_OUT_CONTAINER" />
    <NVSESSIONVAR key="NV_PLATFORM">WIN32</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_PROC">proc1</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_SERVICE">SYSTEM</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_SESSION_ID">E28BC527E9</NVSESSIONVAR>
    <NVSESSIONVAR key="NV_USER" />
    <NVSESSIONVAR key="USER-AGENT">Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR
      1.0.3705; .NET CLR 1.1.4322)</NVSESSIONVAR>
  - <NVCONTAINER name="nvdef">
    - <NVOBJ name="mystring" type="STRING">
        <NVDATA>Hello world!</NVDATA>
      </NVOBJ>
    </NVCONTAINER>
  </NIRVA>
```

Since this xml view also contains some un-useful values, we can tell NIRVA to not display them by changing our URL by the following one:

http://127.0.0.1:1081/NVS?command&NV_APPLICATION=Hello&NV_CLASS=misc&NV_COMMAND=nop&NV_PROC=proc1&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=NO&NV_XML_VARIABLES=NO

The resulting screen is then:

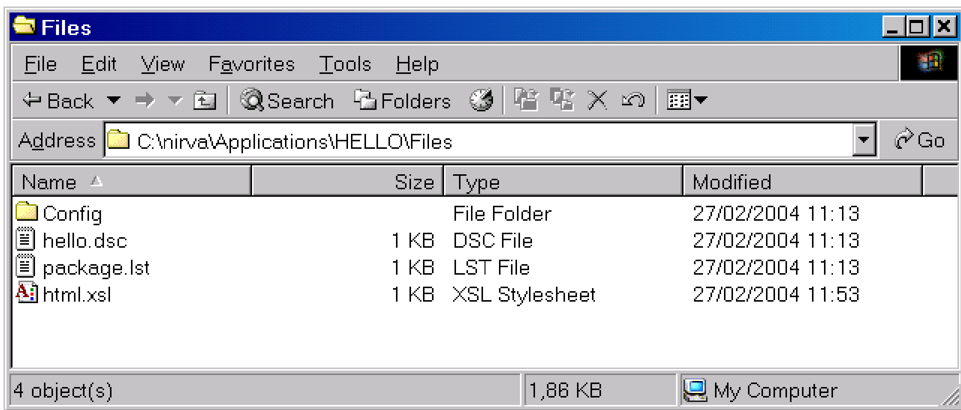


Now instead of displaying the string “Hello world” in XML, we’d like to have it in HTML. For that, we create an XSL file that should transform the XML view into HTML.

Just edit a file named “html.xsl” in the “File” subdirectory of the HELLO application (c:/nirva/Applications/HELLO/Files if NIRVA has been installed on c:/nirva). Set the following content to this file:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
<html>
<body>
  <xsl:value-of select="NIRVA/NVCONTAINER/NVOBJ/NVDATA"/>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

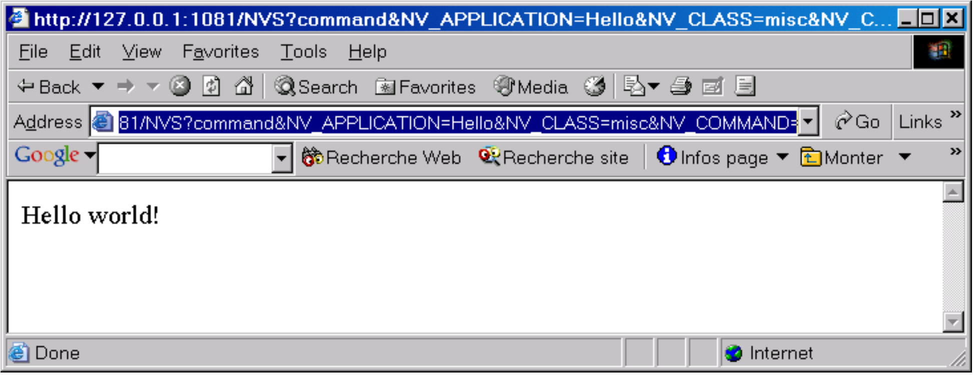
Your HELLO application file directory should then contain the following files:



Finally, change the calling URL from your web browser by the next one:

http://127.0.0.1:1081/NVS?command&NV_APPLICATION=Hello&NV_CLASS=misc&NV_COMMAND=nop&NV_PROC=proc1&NV_CLOSE_SESSION=yes&NV_XML_XSL=html

This then displays your Hello world string in html:



Testing from batch

Your HELLO world application is accessible now from any of the NIRVA connectors including C, C++, ActiveX, Perl, Java, Php, Cold Fusion, Python, XML etc…

NIRVA also provides a batch connector allowing sending NIRVA commands from command line. Let's access our HELLO application in this way.

For that, we create a command file for the batch tool. The file is named "hello.txt" and is created in the NIRVA Bin directory (c:/nirva/Bin if you have installed nirva in c:/nirva).

The hello.txt file must contain the following nirva command:

```
NV_PROC=|proc1|
NV_CMD=|OBJECT:GET| NAME=|MYSTRING|
NV_CMD=|LOCAL:OBJECT:STRING_GET_VALUE| NAME=|MYSTRING|
nvcc::printdata
```

These commands tell NIRVA to execute the proc1 procedure, to get the MYSTRING object back locally, to get its value and finally to display it.

Then open a console, go into the NIRVA Bin directory and type the following command:

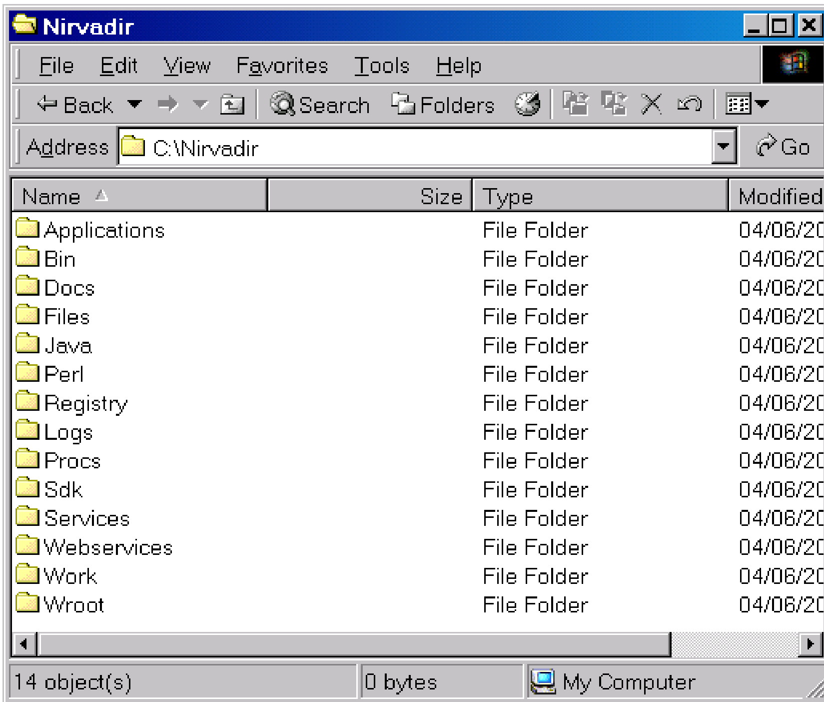
nvcc –p HELLO –i hello.txt

This should display your Hello world string:

```
Cmd
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

c:\>cd nirva

C:\nirva>cd bin

C:\nirva\Bin>nvcc -p hello -i hello.txt
Hello world!
C:\nirva\Bin>_
```

Directory structure

Main directory



The main directory is the installation directory of Nirva. It contains the following subdirectories:

Applications: Nirva applications. Each application is itself contained into a subdirectory of the *Applications* directory. See below for a detailed description of the *Applications* directory.

Bin: Nirva executables.

Docs: Nirva documentations. It includes a subdirectory named “Html” for the html version of the main NIRVA documentation.

Files: Persistent system files. Typically, this directory should contain some XML or XSLT files necessary for the system. It includes a subdirectory named “Config” that contains the system configuration XSLT files.

Java: Java run time environment.

Perl: Perl library modules.

Registry: This is the system registry. This subdirectory content is entirely managed by the Nirva dedicated registry functions and should not be accessed from outside the Nirva server. This directory can be changed from the configuration tool.

Logs: System log files. This directory can be changed from the configuration tool.

Procs: System procedures. It includes a subdirectory named “Config” that contains the system configuration procedures.

Sdk: Software development kits. This directory contains necessary files for developing applications with the NIRVA client connectors.

Services: External services. Each service is itself contained into a subdirectory of the *Services* directory. See later for a detailed description of the *Services* directory.

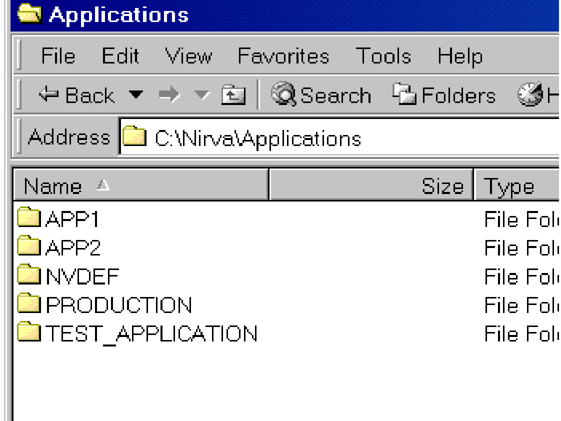
Webservices: Web services. Each web service is itself contained into a subdirectory of the *Webservices* directory. See later for a detailed description of the *Webservices* directory.

Work: This file contains some system level temporary files. NIRVA automatically erases it when starting.

Wroot: Web sites root directory at system level. It includes a subdirectory named “Config” that contains the system configuration web pages.

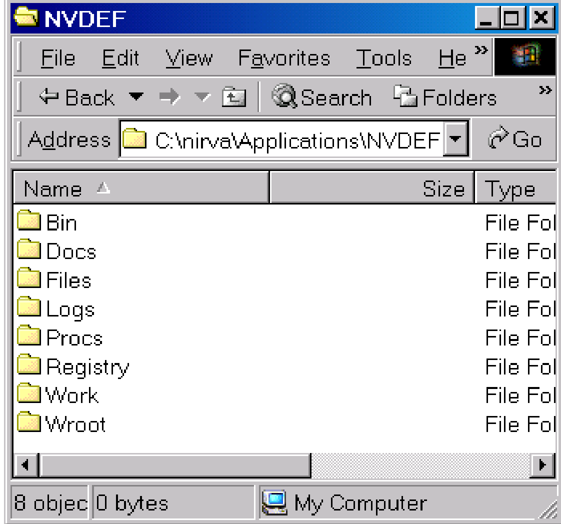
Applications directory

The Application directory is composed of several subdirectories that correspond to the Nirva applications. The names of these subdirectories are directly the application names:



Nirva automatically generates the “NVDEF” application. This is the default application. The default application is used when the user doesn’t require a precise application.

Each application directory contains the following subdirectories:



Bin: Application binary files. By default, Nirva doesn't create any file into this subdirectory but the application builder can use it to store specific application binary files.

Docs: Application documentations. It includes a subdirectory named “Html” for the html version of the application documentation.

Files: Persistent application files. Typically, this directory should contain some XML or XSLT files necessary for the application. It includes a subdirectory named “Config” that contains the application configuration XSLT files.

Logs: Application log files. This directory can be modified from the configuration tool.

Procs: Application procedures. It includes a subdirectory named “Config” that contains the application configuration procedures.

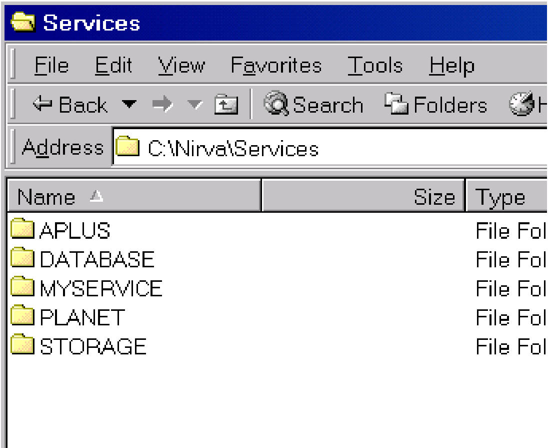
Registry: This is the application registry. This subdirectory content is entirely managed by the Nirva dedicated registry functions and should not be accessed from outside the Nirva server. This directory can be modified from the configuration tool.

Work: Application temporary files. All object files created by the application that are not persistent are stored in this subdirectory. This directory is always cleaned up when NIRVA starts. This directory can be modified from the configuration tool.

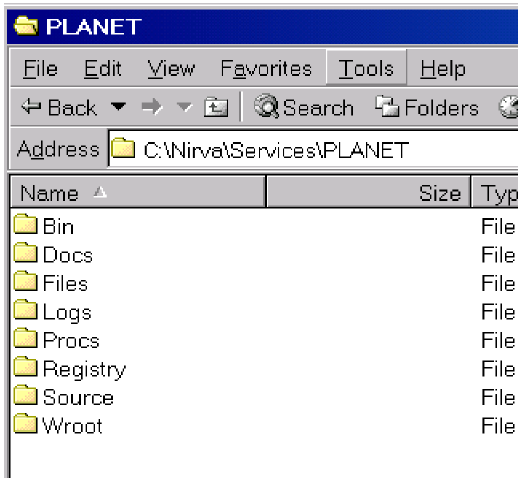
Wroot: Web sites root directory at application level. It includes a subdirectory named “Config” that contains the application configuration web pages.

Services directory

The Service directory is composed of several subdirectories that correspond to the Nirva external services. The names of these subdirectories are directly the service names:



Each service directory contains the following subdirectories:



Bin: Service binary files. Particularly, the service library (or class) itself is generally in this subdirectory.

Docs: Service documentations. It includes a subdirectory named “Html” for the html version of the service documentation.

Files: Persistent service files. Typically, this directory should contain some XML or XSLT files necessary for the service configuration. It includes a subdirectory named “Config” that contains the service configuration XSLT files.

Logs: Service log files. This directory can be modified from the configuration tool.

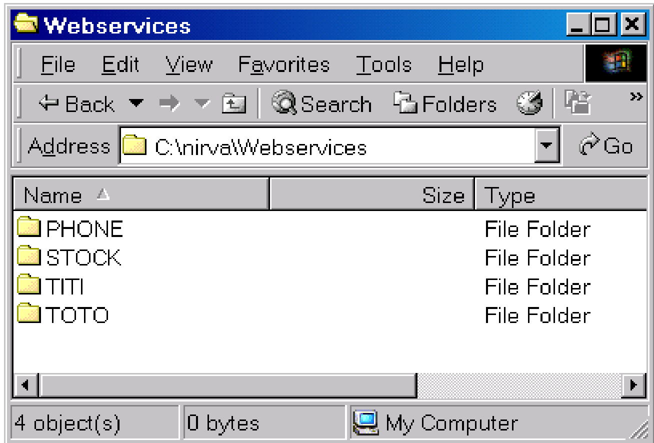
Procs: Service procedures. It includes a subdirectory named “Config” that contains the service configuration procedures.

Registry: This is the service specific registry. This subdirectory content is entirely managed by the Nirva dedicated registry functions and should not be

	accessed from outside the Nirva server. This directory can be modified from the configuration tool.
Source:	Service source files. This is not a mandatory directory. It can be generated automatically with a dedicated Nirva command that creates service source code skeleton.
Wroot:	Web sites root directory at service level. It includes a subdirectory named “Config” that contains the service configuration web pages.

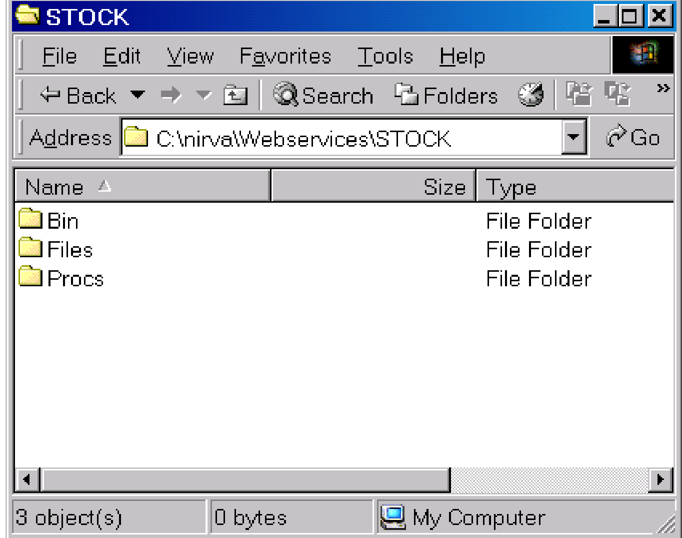
Webservices directory

The Webservice directory is composed of several subdirectories that correspond to the Nirva web services.



The names of these subdirectories are directly the web services names:

Each service directory contains the following subdirectories:



Bin: Web service binary files. By default, Nirva doesn't create any file into this subdirectory but the web service builder can use it to store specific web service binary files.

Files: Persistent web service files. This subdirectory contains the web service installation package file, the web service description file and all the web service definition in a subdirectory named “System”. The web service builder can use the Files directory to store persistent web service files.

Procs: Web service procedures. This directory contains the procedures that process web service operations.

Using Nirva

Starting and stopping Nirva server

After the installation step, the NIRVA server can be started in console or service mode. The service mode is the standard way while the console mode can be used for debugging purposes.

If the environment variable “NIRVA” doesn’t exist or doesn’t point to the Nirva installation directory, NIRVA will fail to start.

Only one instance of NIRVA can be started. So if NIRVA must be started in console mode, it’s necessary to stop the server mode before if this one is running.

Once started, Nirva writes its PID in the file nirva.run in the Bin directory. This file is removed when the program is normaly stopped. If this file is still here while nirva is not running, this may signal a crash condition or a hard kill.

Console mode

For starting the Nirva server in console mode, go into the Nirva Bin directory and type “`nvs -c`” at the command prompt.

To stop a Nirva server started in console mode, just issue a ctrl+c. This will properly stop the server.

Service mode

Under windows, NIRVA can be run as a window service. In order to start or stop NIRVA as a windows service, go to the control panel and choose the “Nirva server” service. You can then start or stop it.

Under UNIX, just type the complete path of the NIRVA server (nvs) without any parameter. For example “\usr\Nirva\Bin\nvs”.

For stopping the NIRVA server from UNIX, go into the Bin directory of Nirva and run the command “`nvcc -i stop_server.txt`”.

Under UNIX, the server can also be stopped by the command "`killall -15 nvs`". This is the only way to stop a nirva server ran in cluster mode and waiting for its peer to be stopped.

Restoring default parameters

NIRVA has some system parameters that can be changed from the configuration tool. These parameters themselves can affect the operation of the configuration tool if they are not correct. In order to restore the NIRVA default system parameters, just stop and restart the server in console mode with the "-d" option: "`nvs -c -d`".

Disabling JAVA VM

NIRVA embeds a JAVA virtual machine for processing JAVA procedures. The JAVA VM is loaded when starting NIRVA.

This JAVA VM can be disabled by using the "-j" command line option: "`nvs -j`". When the java VM has been disabled, NIRVA cannot process JAVA procedures or run JAVA services.

Disabling MQ connector

NIRVA provides a connector for the IBM MQSeries message transport product. This connector can be disabled by using the "-q" command line option: "`nvs -q`".

In order for the MQ connector to run on NIRVA, the IBM MQSeries client must be installed properly on the NIRVA computer and the NIRVA license must authorize the use of the MQ connector.

Encoding

By default, NIRVA supposes that all its internal data is managed in ISO-8859-1 character set allowing main Latin characters to be used.

It's possible to tell NIRVA to use the UTF-8 Unicode character set instead. In this way, NIRVA is able to manage any international character.

For running NIRVA with the Unicode UTF-8 character set, just set the "-u" option: "`nvs -u`".

This is a global choice that must be made right after the installation because NIRVA also stores data in the given encoding into registry. So changing the encoding option later may cause some data to be corrupted or unreadable.

In any case some conversion functions allow to restore the data to the correct encoding but this can be a long operation.

Verbose mode

By default, when nirva has been started in console mode, it displays only commands having an error or coming from client connectors or browser.

The verbose mode allows displaying all commands.

For setting the verbose mode, just set the “-v” option: “`nvs -v`”.

Cluster mode

Nirva can be configured to run in failover cluster mode. At this time, the Nirva instance starts but waits for the peer server to be stopped. When this occurs, the instance continues to load normally.

The cluster mode is being made for architecture with failover cluster. The two servers of a cluster pair should be ran in cluster mode and share the same registry. Sharing the same registry avoid synchronization between servers.

For setting the cluster mode, just set the “-l” option followed by the address (and eventually port if different than 1081) of the peer server: “`nvs -l mycluster:80`”.

For checking if the peer server is running, Nirva tries to connect to it by sending an http request. If not successful, Nirva tries again after 10 seconds and finally starts if the peer server didn’t answer.

When Nirva in cluster mode really starts (when the peer has been detected as not running), a procedure named “failover_start” (in the Nirva/Procs/System directory) is executed. This procedure can be customized to send an alert to an administrator for example.

Time measurement mode

When nirva runs in console mode, one can set the –t option to display time of each procedure a command ran by Nirva. This is very useful to improve the scalability of nirva applications.

Open doors

The power of Nirva architecture resides in its many open doors for programmers.

To add server functionality, the user can:

- Create services
- Use procedures in native, perl, dotnet or java language
- Call any external program from a Nirva command and pass parameters to it
- Call a program that communicates to the Nirva session

To create client applications, the programmer can use:

- XML, SOAP and XSLT Nirva features
- Web service Nirva features
- MQ connector
- HTML Nirva features
- PHP
- Cold Fusion
- Java
- C++
- C#
- Visual basic
- MFC
- Any language able to load a dll
- ActiveX
- Perl
- Ajax
- Flex
- Etc…

It's also possible to create a Nirva session from one connector and to access it from another connector.

Since a big part of the application code is made on the server side, the clients just need to concentrate on the user interface.

For web application using scripting languages, the only parameter to send to each page is the Nirva session identifier. It's enough for a script to re-connect to the Nirva session context.

A Nirva command

A NIRVA command is simply a string that contains a succession of pairs "Parameter name = Parameter value".

In this way, NIRVA can be accessed from any kind of connector or programming language because only a string has to be given.

The exact syntax of the NIRVA command is described later in the chapter "The Nirva command syntax". This syntax is a little bit different if the command comes from a NIRVA client connector or from a Web browser.

Here is the minimum NIRVA command that does nothing but displays the content of the session output container in XML format.

The command is:

http://127.0.0.1:1081/NVS?command&NV_CLASS=misc&NV_COMMAND=nop

This should look like this with an internet explorer 6:


```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <NIRVA session="3E43E13200000021" application="NVDEF" user="" source="BROWSER"
   host="127.0.0.1:1081" local="YES">
   <NVHTTPHEADER key="ACCEPT">image/gif, image/x-xbitmap, image/jpeg,
     image/pjpeg, application/vnd.ms-excel, application/msword,
     application/vnd.ms-powerpoint, */*</NVHTTPHEADER>
   <NVHTTPHEADER key="ACCEPT-ENCODING">gzip, deflate</NVHTTPHEADER>
   <NVHTTPHEADER key="ACCEPT-LANGUAGE">fr</NVHTTPHEADER>
   <NVHTTPHEADER key="CONNECTION">Keep-Alive</NVHTTPHEADER>
   <NVHTTPHEADER key="COOKIE">CFID=1; CFTOKEN=28133161</NVHTTPHEADER>
   <NVHTTPHEADER key="HOST">127.0.0.1:1081</NVHTTPHEADER>
   <NVHTTPHEADER key="USER-AGENT">Mozilla/4.0 (compatible; MSIE 6.0; Windows
     NT 5.0)</NVHTTPHEADER>
   <NVSESSIONVAR key="ACCEPT">image/gif, image/x-xbitmap, image/jpeg,
     image/pjpeg, application/vnd.ms-excel, application/msword,
     application/vnd.ms-powerpoint, */*</NVSESSIONVAR>
   <NVSESSIONVAR key="ACCEPT-ENCODING">gzip, deflate</NVSESSIONVAR>
   <NVSESSIONVAR key="ACCEPT-LANGUAGE">fr</NVSESSIONVAR>
   <NVSESSIONVAR key="CONNECTION">Keep-Alive</NVSESSIONVAR>
   <NVSESSIONVAR key="COOKIE">CFID=1; CFTOKEN=28133161</NVSESSIONVAR>
   <NVSESSIONVAR key="HOST">127.0.0.1:1081</NVSESSIONVAR>
   <NVSESSIONVAR key="NV_APPLICATION">NVDEF</NVSESSIONVAR>
   <NVSESSIONVAR key="NV_CLASS">MISC</NVSESSIONVAR>
   <NVSESSIONVAR key="NV_COMMAND">NOP</NVSESSIONVAR>
   <NVSESSIONVAR key="NV_CONTAINER" />
   <NVSESSIONVAR key="NV_IN_CONTAINER" />
   <NVSESSIONVAR key="NV_NIRVADIR">c:\nirva\</NVSESSIONVAR>
   <NVSESSIONVAR key="NV_OUT_CONTAINER" />
   <NVSESSIONVAR key="NV_SERVICE">SYSTEM</NVSESSIONVAR>
   <NVSESSIONVAR key="NV_SESSION_ID">3E43E13200000021</NVSESSIONVAR>
   <NVSESSIONVAR key="NV_USER" />
   <NVSESSIONVAR key="USER-AGENT">Mozilla/4.0 (compatible; MSIE 6.0; Windows
     NT 5.0)</NVSESSIONVAR>
   <NVCONTAINER name="nvdef" />
</NIRVA>
```

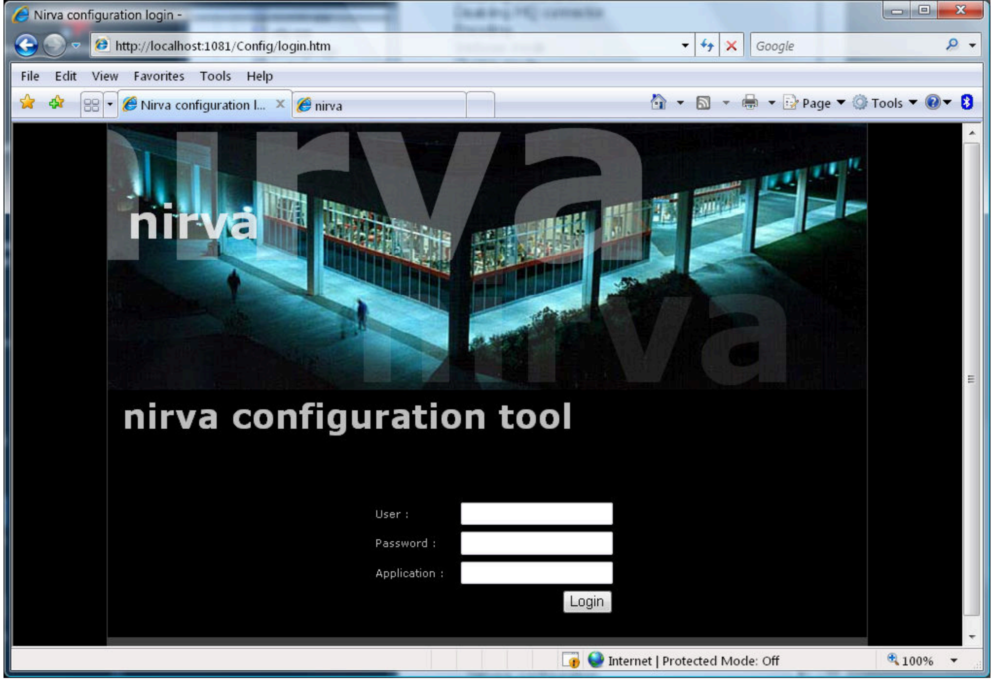

This simple command has created a new session that is not closed automatically. In order to send the same command but closing the session, the command should be the following:

http://127.0.0.1:1081/NVS?command&NV_CLASS=misc&NV_COMMAND=nop&NV_CLOSE_SESSION=yes

A good way to study the power of NIRVA is to try the configuration tool that is built entirely using the XML features of NIRVA. It can be called by typing the following URL from the machine on which NIRVA is installed:

http://127.0.0.1:1081/Config/login.htm

This will display the configuration tool login page (If you are using NIRVA for the first time, enter “nvadmin” as user name and “nirva” as password and nothing for the application name).



Using NIRVA with apache

As an HTTP server, NIRVA may be used as a main web server for an application.

It can also be used as a secondary HTTP server behind a main apache server responding to usual port 80. Apache can then be configured as a reverse proxy or as a load balancer.

Reverse proxy

You must configure apache to redirect all URLs to the NIRVA server. For that, please follow this procedure:

Edit your apache `httpd.conf` file.

Uncomment the following lines if they are commented:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

Add the following virtual host entry:

```
<VirtualHost *>
   ServerName 127.0.0.1
   ProxyPass /nirva/ http://127.0.0.1:1081/
   ProxyPassReverse /nirva/ http://127.0.0.1:1081/
</VirtualHost>
```

This will create a virtual host as a reverse proxy. All URLs paths starting with `/nirva/` will be redirected to `http:/127.0.0.1:1081/`.

For example, if your apache host name is myhost, the URL `http://myhost/nirva/test.htm` will be redirected by apache to `http:/127.0.0.1:1081/test.htm`.

Of course, you can change the TCP/IP address and port to suit your NIRVA installation.

Load balancing

This feature is available only from Nirva server version 4.7.008 that uses machine IDs to identify a Nirva instance.

Apache server (minimum version 2.2.24) must have been installed.

We give here an example of configuration with 3 nirva instances respectively named SERVER1, SERVER2 and SERVER3 and running respectively on machines at address 10.0.0.1, 10.0.0.2 and 10.0.0.3. Nirva instance names are always in uppercase. It can contain only alphanumeric characters.

First go into the nirva configuration in the System/Parameters menu section “Sessions”. On each machine running Nirva, set the m achine ID to SERVER1 for machine at address 10.0.0.1, SERVER2 for machine at address 10.0.0.2 and SERVER3 for machine at address 10.0.0.3.

On apache, create a file named `nv-loadbalancing.conf` with the following content:

```
#
# Load balancing configuration for Nirva Servers
#
# Required modules: mod_proxy, mod_proxy_balancer, mod_proxy_http
#
# The following example shows how to configure load balancing of
# Nirva Servers begind Apache HTTP Server.
#
# The Apache Server is in this case configured in Reverse Proxy.
# (Do not mix up with Forward Proxy).
#
# The configuration relyes on URL for the stickyness.
# The NV_SESSION_ID parameter of the Nirva Server contains the
# route information, which is the machine Id of the server. So
# you have to configure the route parameter of the BalancerMember
# Apache directive according to the server.
#
# This is a sample configuration doing a round robin with the
# same weight on all the servers. For advanced load balancing
# configuration, please refer to Apache HTTP server documentation.
#

# Proxy configuration
ProxyRequests Off # This prevents the server from becoming an open proxy which is very
bad.
ProxyPass / balancer://nvcluster/

# Load balancer configuration
<Proxy balancer://nvcluster>
```

```
  BalancerMember http://10.0.0.1:1081 route=ROUTE1 # ROUTE1 is the machine Id of the
Nirva server installed on machine 10.0.0.1 port 1081
  BalancerMember http:// 10.0.0.2:1081 route=ROUTE2 # ROUTE2 is the machine Id of the
Nirva server installed on machine 10.0.0.2 port 1081
  BalancerMember http:// 10.0.0.3:1081 route=ROUTE3 # ROUTE3 is the machine Id of the
Nirva server installed on machine 10.0.0.3 port 1081
  ProxySet stickysession=NV_SESSION_ID|NV_SESSION_ID
</Proxy>
```

This file must then be copied into the same directory than the apache configuration file `httpd.conf`.

Edit your apache `httpd.conf` file.

Uncomment the following lines if they are commented:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

Under linux (ubuntu), you can start the modules with the following commands:

```
a2enmod proxy
a2enmod proxy_balancer
a2enmod proxy_http
```

Add the following line:

```
# Nirva Servers load balancing
Include conf/nv-loadbalancing.conf
```

Then restart your apache server. If your apache server is responding to port 80, you can test by running the url http://localhost/Config/login.htm in your browser and then connect to nirva. You will see that you access successively to the 3 Nirva servers. The load balancing is synchronized on the Nirva session ID.

The load balancing works with browser clients and with client connectors using the librairy nvc (c++, java, dotnet, virtual printer, perl, php connectors). This last must be at least at version 4.7.008 (there is no version on the library itself so just get it from the corresponding nirva version in the Sdk/Connectors/dll directory of the nirva installation). If you have older nvc clients you can make them work with the apache load balancer but you need to modify your `nv-loadbalancing.conf` file by adding the following content:

```
# Tricks to make load balancing working with old nvc clients.
# Nv_Session_Id is a HTTP header in that case.
SetEnvIfNoCase Nv_Session_Id "^([A-Z0-9]+\.[A-Za-z0-9_]+)$" NV_HTTP_HEADER_NVSESSIONID=$1
RequestHeader merge Cookie "NV_SESSION_ID=%{NV_HTTP_HEADER_NVSESSIONID}e" env=NV_HTTP_HEADER_NVSESSIONID
```

Setting HTTPS protocol

NIRVA implements the HTTPS protocol for secure communications between the client (and browser) and the server.

The HTTPS server runs in parallel from the NIRVA HTTP server. If the NIRVA HTTP server listens on the TCP/IP port 1081, the HTTPS server listens on port 1082 (HTTP port plus one).

By default, the HTTPS server is not started by NIRVA.

In order to use the HTTPS server, you must follow the following steps:

- Creating a private key and a certificate.
- Installing the private key and the certificate on NIRVA.
- Configuring NIRVA to use the HTTPS server.
- Testing the NIRVA HTTPS server.

Creating a private key and a certificate

This step is not NIRVA dependent. There are several ways and tools for creating private keys and certificates. We encourage the reader to look at these tools (A well known one is OpenSSL).

The general procedure is the following one:

- You create a private key.
- You create a certificate request coded with the private key. When creating a certificate request, you are prompted for a certificate name (CN=…). Be careful to use your domain name for it (For example “acme.com” if acme is your company name). This should avoid undesirable messages when you will connect to a NIRVA server with HTTPS on your company site.
- You send the certificate request to a certificate authority (Thawte or Verisign for example).
- The certificate authority sends back to you the certificate after having verified that you are allowed to create it.

Nirva accepts only PEM certificate format. If your certificate is not in this format, you must convert it.
For example if your certificate is in PFX format (PKCS12) you can convert it using openssl tool in this way:

```
openssl pkcs12 -in certificate.pfx -out certificate.pem
```

Where certificate.pfx is the certificate file name in pfx format.

The conversion tool is not delivered with Nirva.

Once you have received your certificate, you are ready to install it on NIRVA.

Installing the certificate

For security reasons, installing the certificate is possible only when the NIRVA server is stopped. So please stop it.

The certificate installation is quiet simple. You just have to run the following command from the command prompt:

```
nvs -k CertificateFile PrivateKeyFile
```

Where *CertificateFile* is the name of the file that contains your certificate and *PrivateKeyFile* is the name of the file that contains your private key.

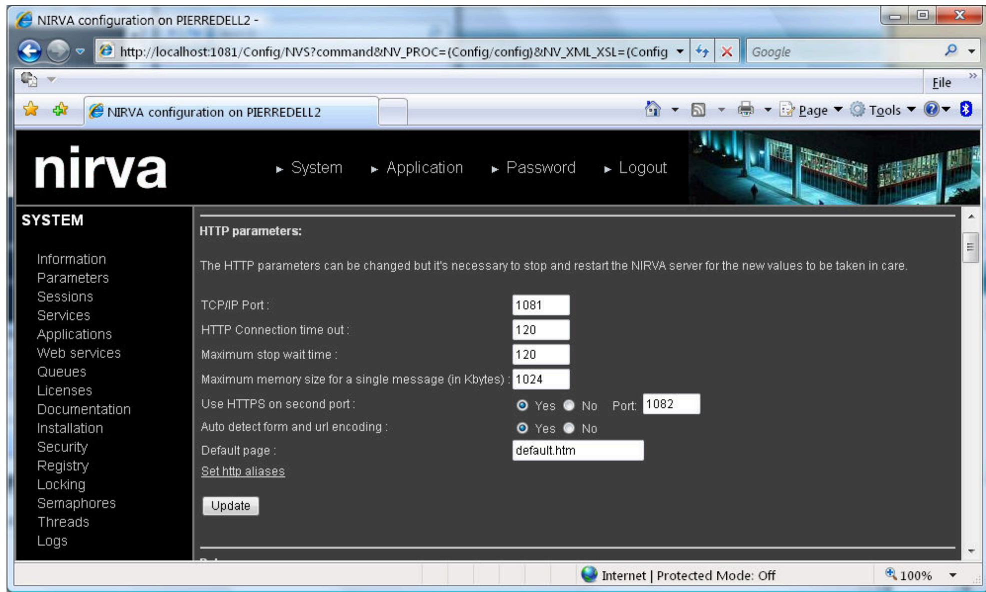
Then NIRVA itself installs the certificate after having checked it.

You can now restart the NIRVA server.

Configuring NIRVA

By default the NIRVA HTTPS server is not enabled. You must enable it by running the nirva configuration tool (see the configuration chapter for learning how to run the configuration tool).

Then go to the system/parameters menu and display the HTTP parameters:



You can then toggle the use of the HTTPS server by choosing the corresponding option.

If the normal HTTP server is listening on a given port (1081 is the default), the HTTPS server will listen on the next port number (so 1082 if 1081 is used as the HTTP port).

You must then stop and restart the server for the new parameters to be taken in care.

Testing the HTTPS server

For this step, just try to access the NIRVA HTTPS server from a web browser by calling for example the following URL (from your local server):

https://127.0.0.1:1082/Config/login.htm

This should open the NIRVA configuration tool login page in https mode. Following your certificate and your browser configuration, the browser may display or request you some information before displaying the page.

Using client certificates

The Nirva HTTPS server can be configured to restrict access to certified users. At this time, Nirva request the client to send its certificate and verify it against one or several certificating authorities (CA).

The option “Require client certificate with HTTPS” must be set in the System/Parameters and the path to the file containing the trusted CA certificates in pem format must be given in the HTTPS parameters. This file may contain several certificates.

Here is an example for creating a self signed client certificate with openssl and use it with nirva:

Creating certificate requests

When using OpenSSL, there are two steps to creating a certificate signing request (CSR): creating the private RSA key, and creating the certificate request containing the user's name and other information.

First, create a key. User-entered input is shown **in bold**:

```
C:\tmp>openssl genrsa -des3 -out clientcert.key 1024
Loading 'screen' into random state - done
Generating RSA private key, 1024 bit long modulus
..............++++++
................++++++
e is 65537 (0x10001)
Enter pass phrase for clientcert.key: (the password is not echoed)
Verifying - Enter pass phrase for clientcert.key:
C:\tmp>
```

This example creates a 1024-bit key and stores it in clientcert.key. 1024 bits is a good level of security, but for even better security (but slower performance) you may choose a 2048-bit key.

Next, create the CSR (certificate signing request):

```
C:\tmp>openssl req -config \moveitdmz\util\openssl.conf -new -key clientcert.key -out
clientcert.csr
Enter pass phrase for clientcert.key: (enter the password given above)
You are about to be asked to enter information that will be incorporated
```

```
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:US
State or Province Name (full name) [Some-State]:Wisconsin
Locality Name (eg, city) []:Madison
Organization Name (eg, company) [ACME Inc.]:Universal Exporters
Organizational Unit Name (eg, section) []:Accounting
Common Name (eg, fully qualified host name) []:Fred
Email Address []:fred@univ-exporters.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
C:\tmp>
```

This example creates a certificate request for fred. The optional challenge password and company name are typically left blank. The file clientcert.csr is ready to be sent to the Certificating Authority who will sign the certificate.

Signing certificate requests

Once a certificate request has been created, it should be sent to a Certificating Authority for signing. A Certificating Authority can be:

- A commercial certificate firm such as Thawte or Comodo, or
- A unit in your organization, such as the security group in your IT department, or
- Yourself (typically only in smaller companies)

If you want to make yourself a Certificating Authority so you will be able to sign CSRs yourself, you need a separate certificate. This type of certificate is issued to an administrator and is NOT needed by individual users. You can obtain such a certificate from various sources, including all three types listed above. If you work for a small organization, or are just testing, you may wish to create your own "self-signed" certificate. Self-signed certificates provide the same level of encryption as commercially-purchased types, but require a bit more effort before the server will "trust" them. Self-signed certificates are free and can have as long a lifetime as you want.

Creating your own self-signed certificate

To create a self-signed certificate so you can sign CSRs yourself:

```
C:\tmp>openssl req -config \moveitdmz\util\openssl.conf -x509 -days 365 -newkey rsa:1024
-keyout MyCAcert.key -out MyCAcert.cer
Loading 'screen' into random state - done
Generating a 1024 bit RSA private key
```

```
.....++++++
....++++++
writing new private key to 'MyCAcert.key'
Enter PEM pass phrase: (enter a new password that will be known only to the
administrator)
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:US
State or Province Name (full name) [Some-State]:Wisconsin
Locality Name (eg, city) []:Madison
Organization Name (eg, company) [ACME Inc.]:Universal Exporters
Organizational Unit Name (eg, section) []:IT Dept
Common Name (eg, fully qualified host name) []:UE IT Security
Email Address []:ueitsec@univ-exporters.com
C:\tmp>
```

This creates a 1024-bit certificate that expires in 365 days. In this example, the administrator creating the certificate is in the same organization as the client certificate applicant above, but in a different department. The key is written to MyCAcert.key and the public certificate to MyCAcert.cer. Be sure to keep the MyCAcert.key file and its password secure.

Signing certificate requests yourself

Once you have a signing key (either created yourself or obtained otherwise), you can sign CSRs:

```
C:\tmp>openssl x509 -req -in clientcert.csr -days 1000 -CA MyCAcert.cer -CAkey
MyCAcert.key -CAcreateserial -out clientcert.cer
Loading 'screen' into random state - done
Signature ok
subject=/C=US/ST=Wisconsin/L=Madison/O=Universal
Exporters/OU=Accounting/CN=Fred/emailAddress=fred@univ-exporters.com
Getting CA Private Key
Enter pass phrase for MyCAcert.key: (enter the password of the CA cert)
C:\tmp>
```

This reads the user's certificate signing request and signs it, creating a client certificate in the file clientcert.cer. In this example, the certificate will be valid for 1000 days.

At this point, clientcert.cer is the public component of the client certificate, and clientcert.key is the private component. Some client software, most notably Microsoft Windows, requires that these files be converted to a different format before they can be used by the client. If you have access to the user's clientcert.key file (for example, if you performed the equivalent of Example 1 yourself), you can convert these two files into the single-file .pfx format required by Windows by using a command like:

```
C:\tmp>openssl pkcs12 -export -in clientcert.cer -inkey clientcert.key -out
clientcert.pfx
Loading 'screen' into random state - done
Enter pass phrase for clientcert.key:  (enter the password created via "openssl genrsa"
at the top)
Enter Export Password:  (enter a new password.  It can be the same as the openssl genrsa
password)
Verifying - Enter Export Password:
C:\tmp>
```

The file clientcert.pfx now contains both the private and public components of the key. If the user created his or her own CSR and did not give you the .key file, the user will have to perform this "openssl pkcs12" command.

Importing client certificates on the user's computer

On the user's computer, the client certificate will have to be imported into the computer's certificate store. If the operating system is Microsoft Windows, the user should copy clientcert.pfx onto the computer and perform these steps:

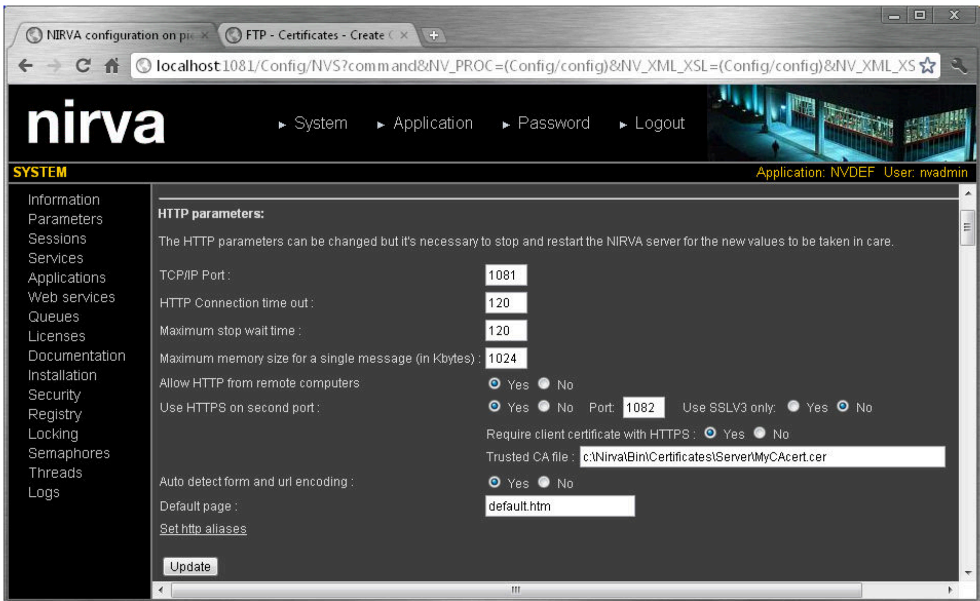
- Double-click on the .pfx filename in Windows Explorer to run the Certificate Import Wizard
- On the "Welcome" page, choose Next
- On the "File to Import" page, choose Next (the filename will already be filled in)
- On the "Password" page:
  - Enter the export password you assigned above
  - Choose "Mark the private key as exportable"
  - Choose Next
- On the "Certificate Store" page, choose Next (Automatically will already be checked)
- On the "Completing" page, choose Finish
- At the "The import was successful" dialog, choose OK

The client certificate is now ready to be used. To double-check that the certificate has been installed, you may wish to examine the list of client certificates:

- Run Internet Explorer
- Choose Tools | Internet Options...
- Choose the Content tab
- Choose the Certificates... button
- On the Personal tab, you should see the newly-imported certificate
- For information on it, double-click the name of the certificate.

Installing the CA certificate on the Nirva server

If you created a self-signed CA certificate (MyCAcert.cer in this example) or if you use a known one, you must tell were these certificates are in order for Nirva to be able to verify if receives client certificates are authorized or not. For that, you must have your CA certificate in pem format in a file and specifies the path to this file in Nirva system/parameters section (HTTP parameters) in the configuration tool. The file may contain several CA certificates concatenated together.



Testing the client certificate

You can test your client certificate with the following command:

```
nvcc -a localhost:1082(SSL) -k clientcertwp.cer -r perl:ptest
```

where clientcertwp.cer is the file containing the concatenation of clientcert.cer and clientcert.key previously created (you can use a simple text editor to create this file).

Configuration

The NIRVA configuration tool is available from a WEB browser. It allows configuring NIRVA itself but also its applications or external services.

Starting the configuration

The configuration tool is available by typing the following URL:

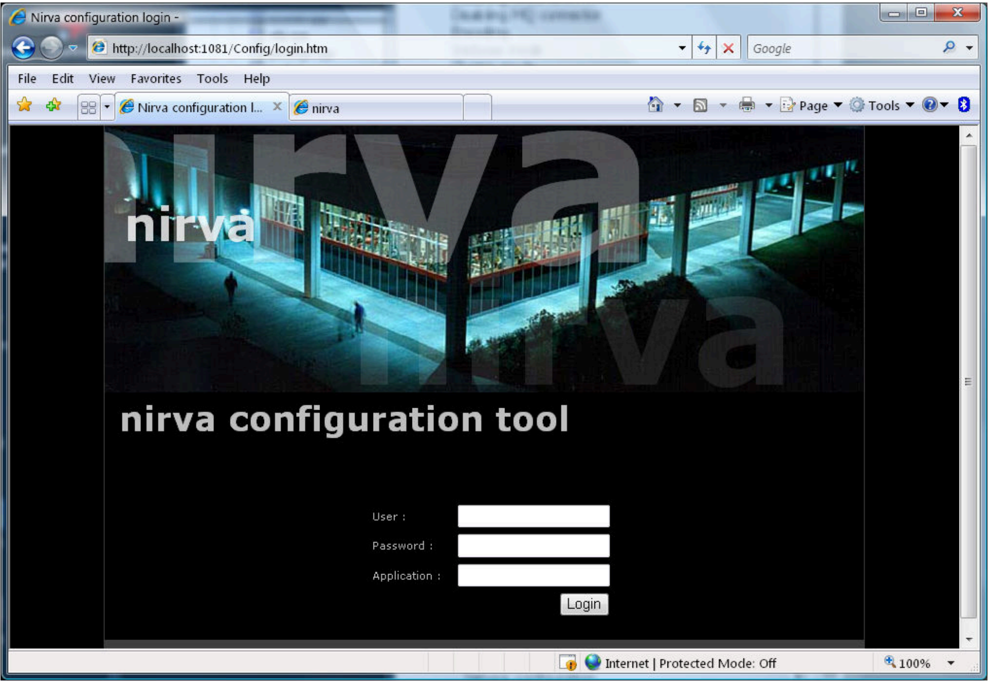
```
http://NirvaServer:1081/Config/login.htm
```

Where NirvaServer is the name or address of the NIRVA server and 1081 is the NIRVA HTTP port.

For example, in order to access the configuration tool when NIRVA is installed locally, just type the URL:

http://127.0.0.1:1081/Config/login.htm

This displays the NIRVA configuration login screen:



When entering the configuration tool, you create a new NIRVA session so you must logon to NIRVA. Since a NIRVA session always runs in an application context, you must also give the name of the application you want to connect.

Choosing the application is important because the tool allows also configuring the connected application.

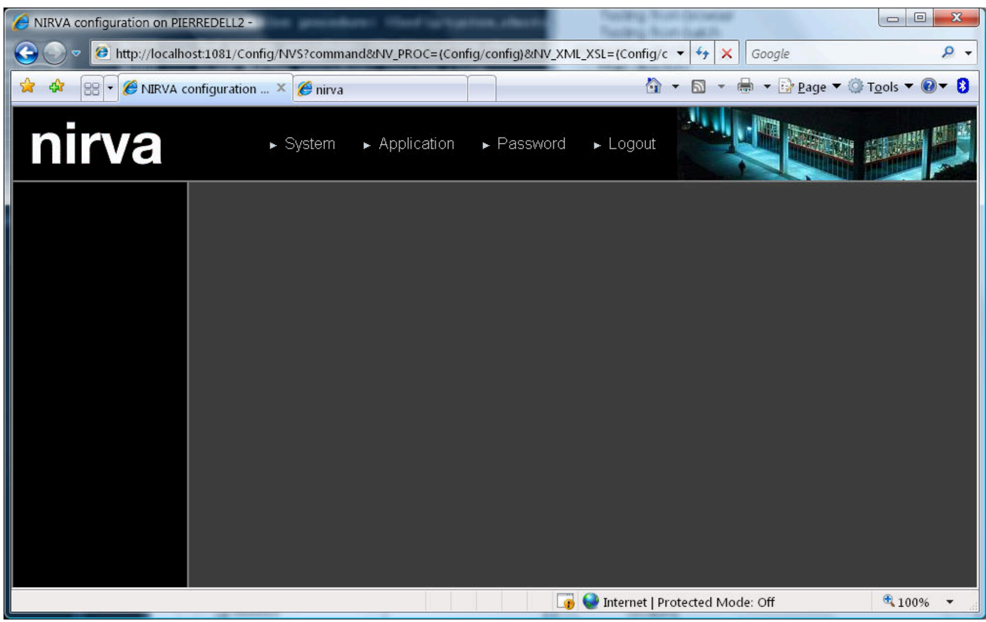
If no user is entered, NIRVA uses the default application user (nvdef).

If no application name is entered, NIRVA connects to the default application (NVDEF).

If you are using NIRVA for the first time, you can enter “nvadmin” as user name and “nirva” as password.

Following the configuration of the connected user Nirva may require it to change its password at login time.

After pressing the “Login” button, NIRVA displays the main configuration menu:



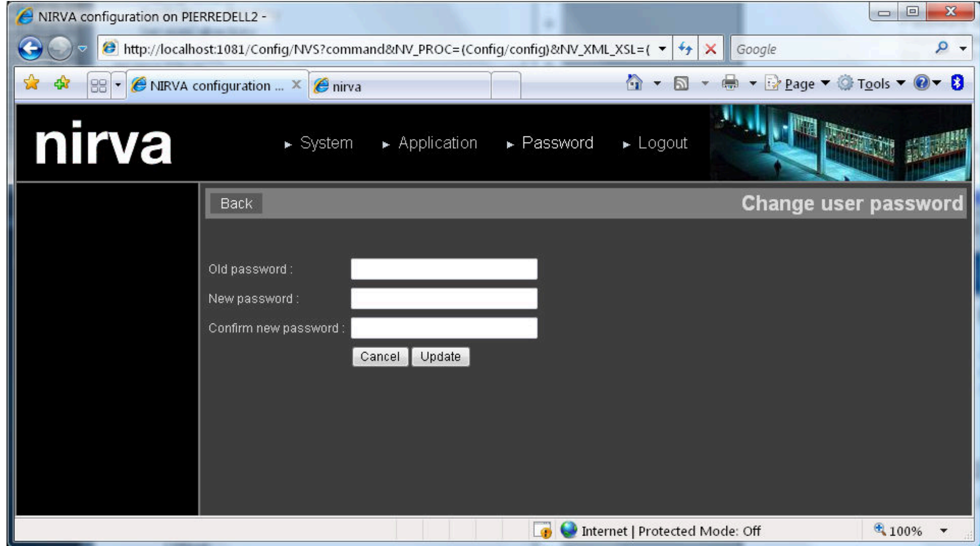
The main configuration menu may be different following the permissions you have. If you want to get all permissions, please use the “nvadmin” user.

This menu provides 4 options:

- System allows configuring NIRVA itself.
- Application allows configuring the connected application (the one chosen in the login screen).
- Password allows the user to change its password.
- Logout allows to properly terminating the configuration session. If the Logout button is not used, the session will be properly closed when the time out occurs.

Password

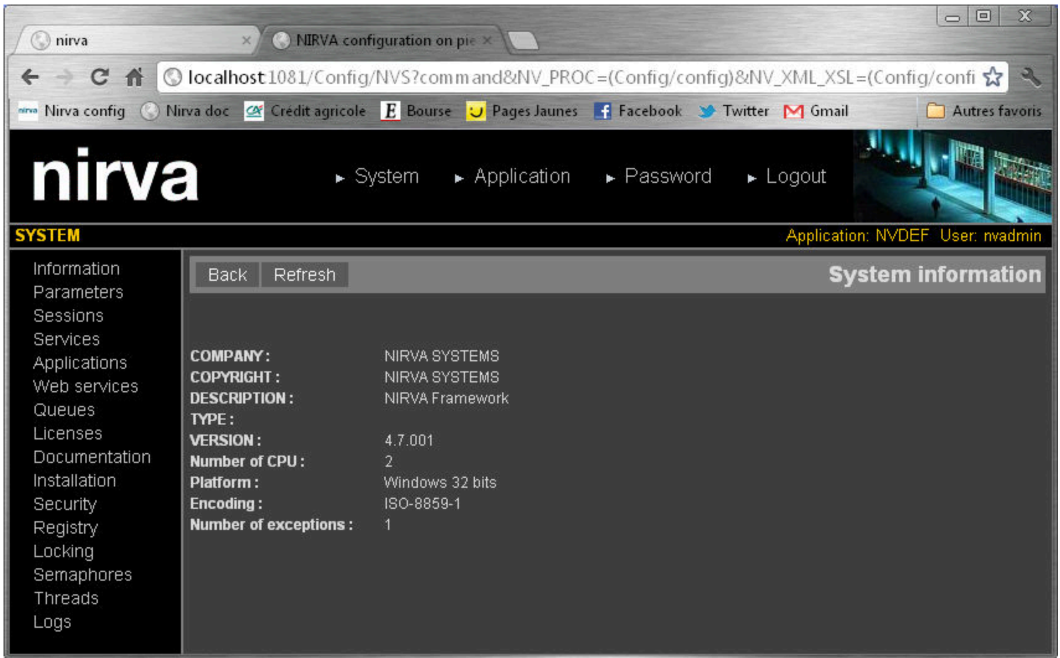
When choosing the main “Password” link, NIRVA displays the following screen allowing the user to change his password:



The user must give his old password and his new one (with confirmation).

System

When choosing the main “System” link, NIRVA displays the following screen:



The left part of the screen provides the “System” configuration menu. When clicking an item of a menu, NIRVA displays information about this item in the main frame.

When entering the system configuration menu, NIRVA automatically activate the “Information” item.

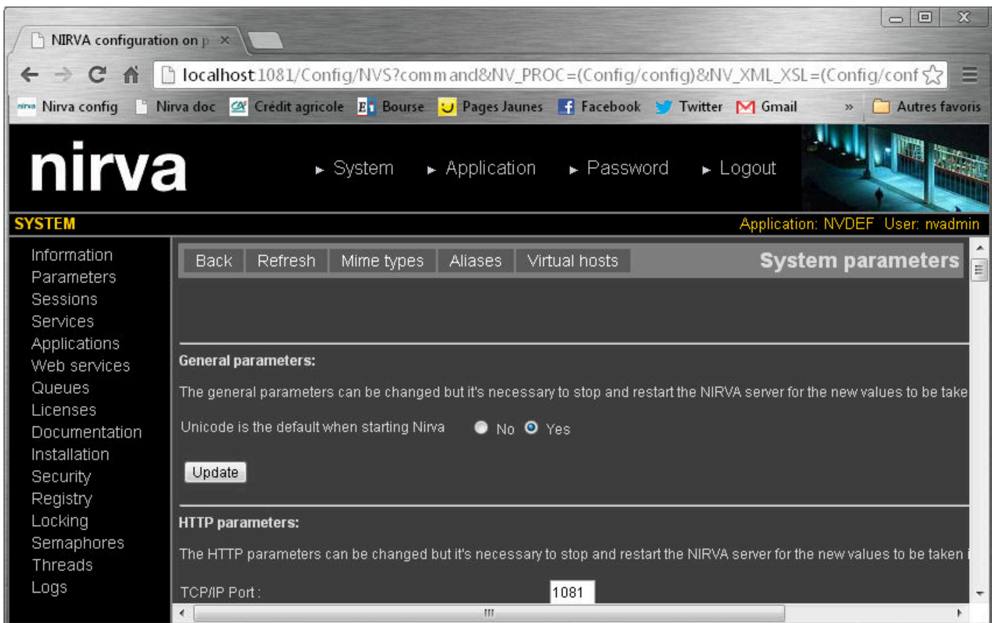
Information

This displays some system information.

If the “Number of exceptions” is not zero, the process may be unstable and it is advised to restart it. This condition may be also checked using the scheduler and the SYSTEM:GET_NUM_EXCEPTIONS command in order to alert an administrator automatically.

Parameters

This option allows changing some general system parameters. It displays the following screen:

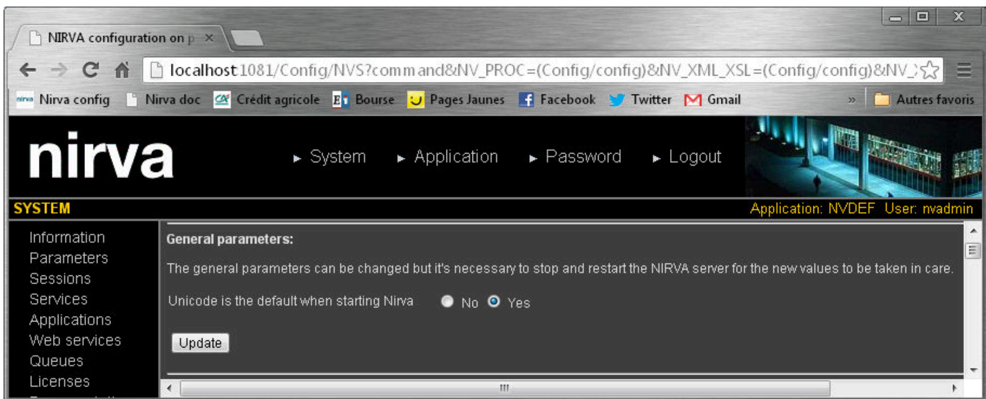


For this screen, it’s possible to change the following parameters:

- General parameters
- Built-in HTTP server parameters
- Debug mode
- Session parameters
- Compatibility parameters
- Time parameters
- XML parameters
- Home directories
- Java VM parameters

- Transaction server parameters
- Scheduler parameters
- Default application parameters
- Mail server parameters
- Web services parameters
- Mime types

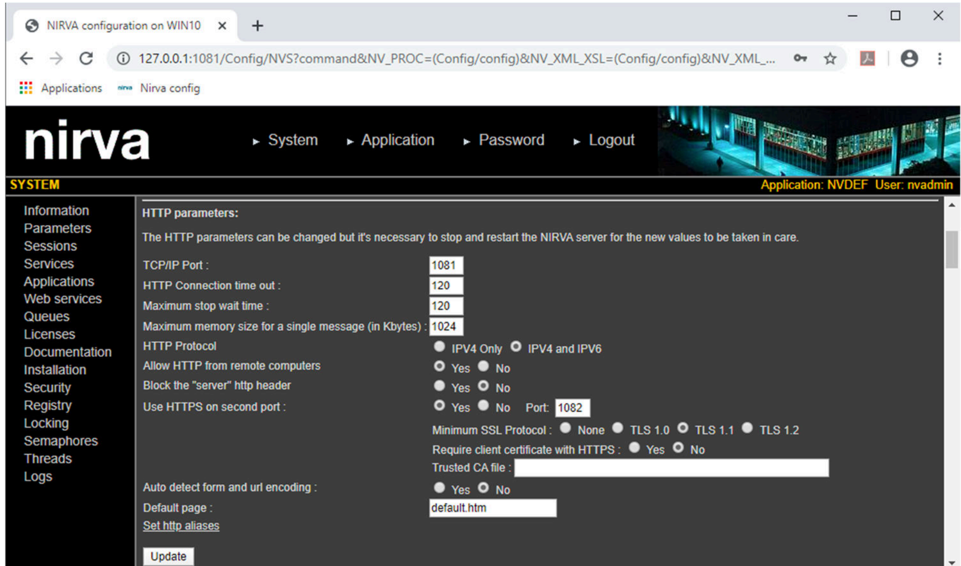
General parameters



The General parameters can be changed but it's necessary to stop and restart the NIRVA server for the new values to be taken in care.

The “Unicode is the default when starting Nirva” allows changing the default encoding behaviour when starting Nirva. If this parameter is set to “Yes”, Nirva will always start in unicode, even if the –u command line parameter is not used.

HTTP parameters



The HTTP parameters can be changed but it's necessary to stop and restart the NIRVA server for the new values to be taken in care.

The TCP/IP port is the socket port where the built in HTTP server is listening for new connections. The default value is 1081.

The HTTP connection time out is the maximum time (in seconds) allowed for a connection to stay inactive. Since the management of sessions and TCP/IP connections are completely independent, it’s not necessary to keep a big value for the HTTP time out. The default value of 120 seconds seems good. When a client tries to send a command via a connection that is in time out state, it creates a new connection to access the session. The minimum value for HTTP connection time out is 10 seconds. The default value is 120 seconds.

The maximum stop wait time parameter is used when stopping the HTTP server (so when stopping NIRVA). The HTTP server must then wait for a certain time for all current commands to be finished. If after this time, some connections are still in use, NIRVA hardly close them. The minimum value for the stop wait time is 5 seconds. The default value is 120 seconds.

The maximum memory size for a single message parameter is the maximum amount of memory (in kilobytes) that a message can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 1 Mbyte (1024 kilobytes) that should be a good compromise between performance and memory management. The minimum value is 10 kilobytes.

HTTP Protocol tells if Nirva must listen on IPV4 only or IPV4 and IPV6 protocols.

The next option allows restricting the use of HTTP to local machine only. So the remote computers will be able to connect this Nirva server only via HTTPS (if enabled).

The next option allows enabling the use of the HTTPS server. The default HTTPS port is 1082. The HTTPS protocol is automatically choosen to the maximum level both accepted by client and server. The minimum authorised level by the Nirva server can be restricted to a specific protocol. HTTPS protocol may require client certificate. At this time the file containing trusted certificating authorities certificates (in pem format) must be given. The file may contain several certificates. Client certificates are used to enable access via HTTP to certified users. They are not used for authentication process.

The “auto detect form and url encoding” flag allows Nirva to automatically detect the form and url encoding because the web browsers doesn’t send this information.

“Default page” is the default server page when the url points to a directory.

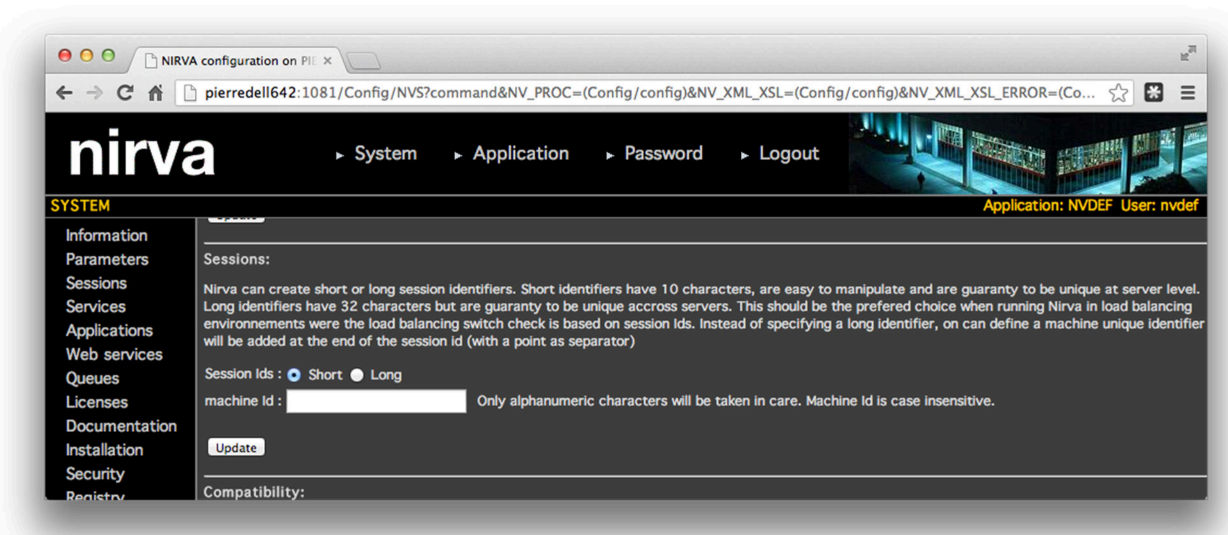
The “Set http aliases” allows accessing to the screen for setting http aliases. Please see the Aliases chapter latter in this documentation.

Debug

When NIRVA is in debug mode, all the generated HTML and XML files are saved in the application file directory into a subdirectory named "Debug". This allows controlling the XML and HTML output of NIRVA.

Also in debug mode, NIRVA launches the deadlock checking. A deadlock is an accidentally blocking of a synchronization object in a multithread program. When in debug mode, any access to a synchronization object is checked and if any potential deadlock is found, NIRVA writes it to a file named “nvdead.lock” in the Bin directory. This file contains information useful for NIRVA support team to identify the origin of the deadlock.

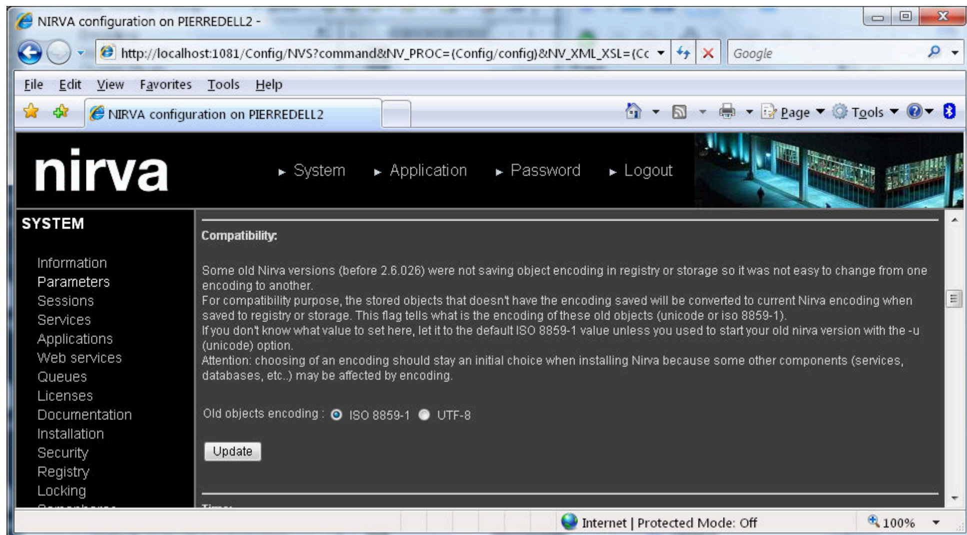
Sessions



Nirva can create short or long session identifiers. Short identifiers have 10 characters, are easy to manipulate and are guaranty to be unique at server level. Long identifiers have 32 characters but are guaranty to be unique across servers. This should be the preferred choice when running Nirva in load balancing environnement were the load balancing switch check is based on session Ids. Instead of specifying a long identifier, on can define a machine unique identifier that will be added at the end of the

session id (with a point as separator). This feature allows some load balancing software (ie apache) to synchronize the session to the correct server.

Compatibility



This section sets the compatibility parameters.

Some old Nirva versions (before 2.6.026) were not saving object encoding in registry or storage so it was not easy to change from one encoding to another.

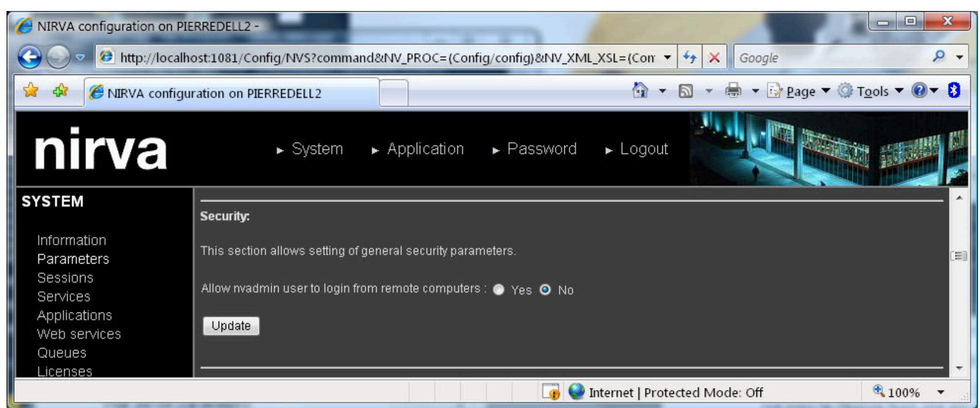
For compatibility purpose, the stored objects that don't have the encoding saved will be converted to current Nirva encoding when saved to registry or storage. This flag tells what is the encoding of these old objects (unicode or iso 8859-1).

If you don't know what value to set here, let it to the default ISO 8859-1 value unless you used to start your old nirva version with the -u (unicode) option.

Attention: choosing of an encoding should stay an initial choice when installing Nirva because some other components (services, databases, etc...) may be affected by encoding.

It's necessary to restart Nirva for the new value of this parameter to be taken in care.

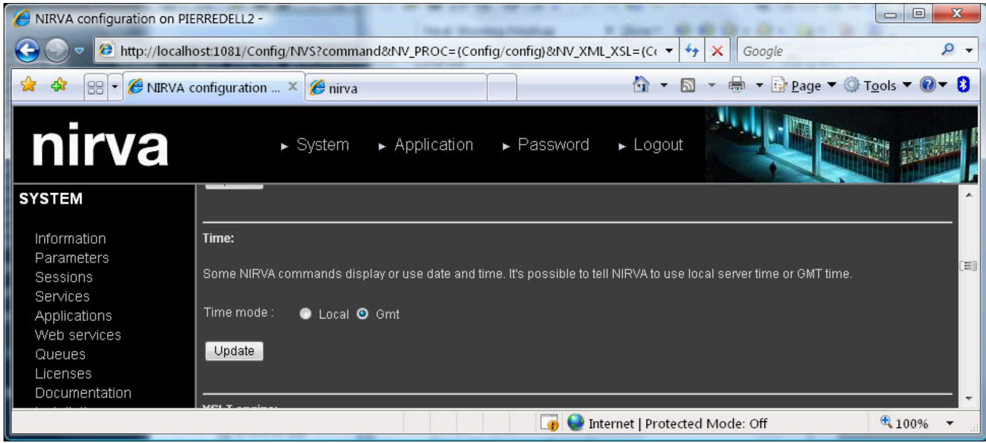
Security



This section sets the global security parameters.

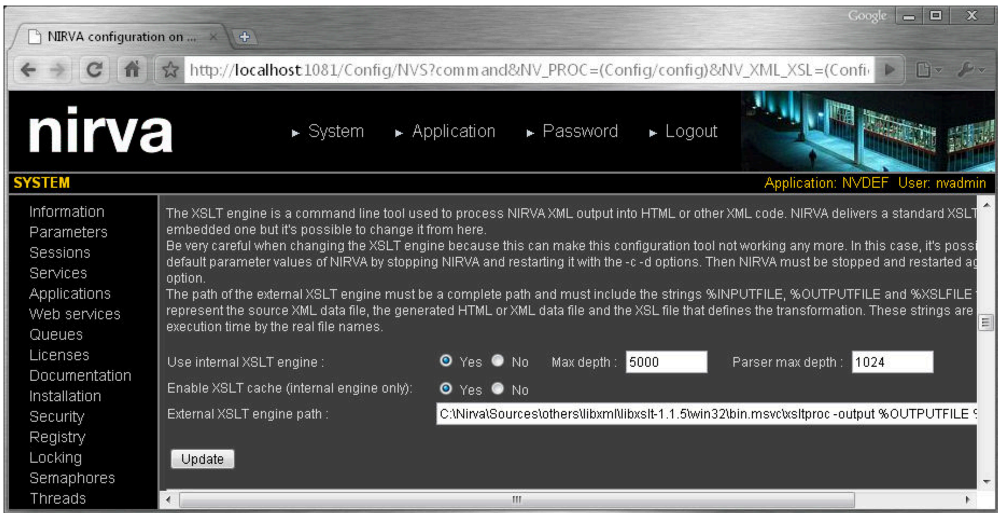
By default Nirva allows nvadmin user to connect from remote computers. This can be disabled by setting “Allow nvadmin user to login from remote computers” to “No”.

Time



Some NIRVA commands display or use date and time. It's possible to tell NIRVA to use local server time or GMT time.

XSLT engine



The XSLT engine is a tool used to process NIRVA XML output into HTML or other XML code.

NIRVA delivers a standard XSLT engine and an embedded one but it's possible to change it.

Be very careful when changing the XSLT engine because this can make the configuration tool not working any more. In this case, it's possible to return to the default parameter values of NIRVA by stopping NIRVA and restarting it with the -c -

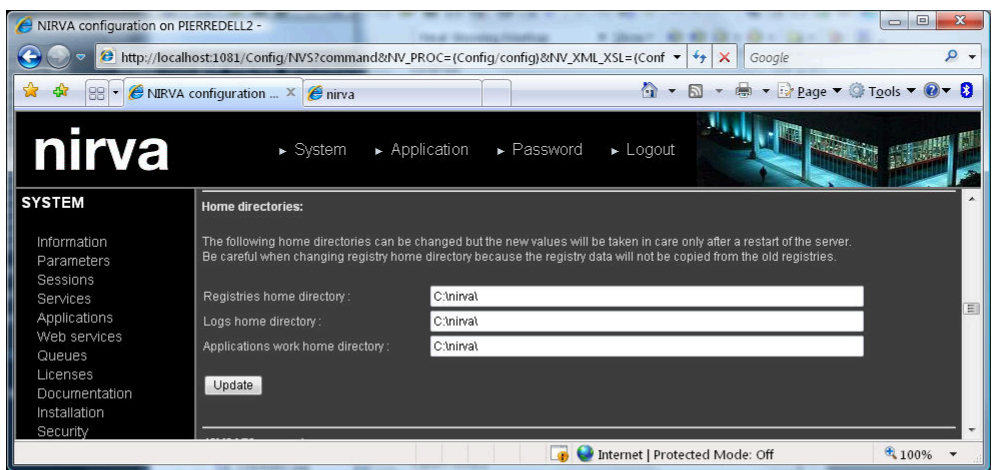
d options. Then NIRVA must be stopped and restarted again without the -d option.

The path of the external XSLT engine must be a complete path and must include the strings %INPUTFILE, %OUTPUTFILE and %XSLFILE that respectively represent the source XML data file, the generated HTML or XML data file and the XSL file that defines the transformation. These strings are replaced at execution time by the real file names.

When using the internal XSLT engine, the depth of the xml and xslt parsers can be changed. This must be done only if some errors are reported by the xslt engine telling (in the nirva console) to increase these values.

Nirva provides a cache of parsed XSLT files in order to increase performance. This cache can be disabled if necessary.

Home directories



Some of the basic NIRVA directories can be changed.

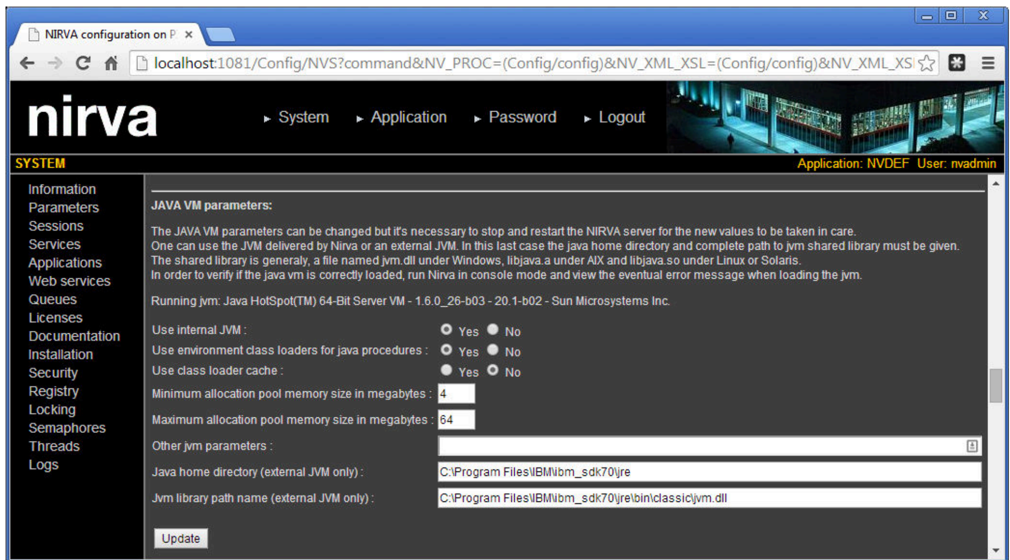
The directories defined here are home directories and NIRVA will create the necessary subdirectories in them. For example, if the “Applications work home” directory is defined as “c:\nvwork\”, NIRVA will use (and create) the directory “c:\nvwork\Applications\NVDEF\Work” for storing the NVDEF application temporary files.

The given home directories must exist.

The changes will be taken in care only after the next restart of the NIRVA server.

Be very careful when changing the registries home directory because the registry data is not copied from the old home registries directory. It’s the same for the logs home directory.

JAVA VM parameters



These are the parameters of the embedded or external JAVA virtual machine used for running JAVA procedures and JAVA services.

Nirva is delivered with a complete java environment but can use also an external jvm.

In this last case the java home directory and complete path to jvm shared library must be given.

The shared library is generally, a file named jvm.dll under Windows, libjava.a under AIX, libjava.so under Linux or Solaris, libjvm.so under Hpux itanium and libjvm.sl under Hpux risc. Under unix platforms some environment variables telling where the OS must search for libraries must be set to the correct directories (LD_LIBRARY_PATH for and solaris, LD_LIBRARY_PATH and SHLIB_PATH for hpux and LIBPATH for aix). See the jvm documentation for further details.

In order to verify if the java vm is correctly loaded, run Nirva in console mode and view the eventual error message when loading the jvm.

Nirva implements a class cache for java procedures. It can be enabled or not. It is advised to set the class cache when using a sun jvm because some sun jvm versions has a bug in memory management that is hidden when the cache is set. This also saves time when loading procedures. The cache can be disabled during development step in order to easily change the java code without having to restart an application.

Environment class loader: As an option, an environment class loader can be defined for java procedures at system, application, service or web service level. This class loader is then used as a parent class loader for each procedure loaded in the corresponding environment. This improves performance (avoids loading some jars at each call of a procedure) and allows sharing global jar files for the entire environment. The global jars must be in a subdirectory named JavaLib in the Procs directory.

The parameters also allow modifying the minimum and maximum heap sizes of the JAVA VM heap. The minimum value cannot be less than 4 (megabytes).

Other jvm parameters are command line parameters transmitted to the jvm when loading it. Please consult the jvm documentation for syntax of jvm parameters. Some of these parameters are reserved and cannot be set, they are “-Djava.class.path=…”, “-Djava.home=…”, “-Xrs, “-Xms”, “-Xmx”.

Since the JAVA VM is loaded when starting NIRVA, the new parameters will be taken in care only when restarting the NIRVA server.

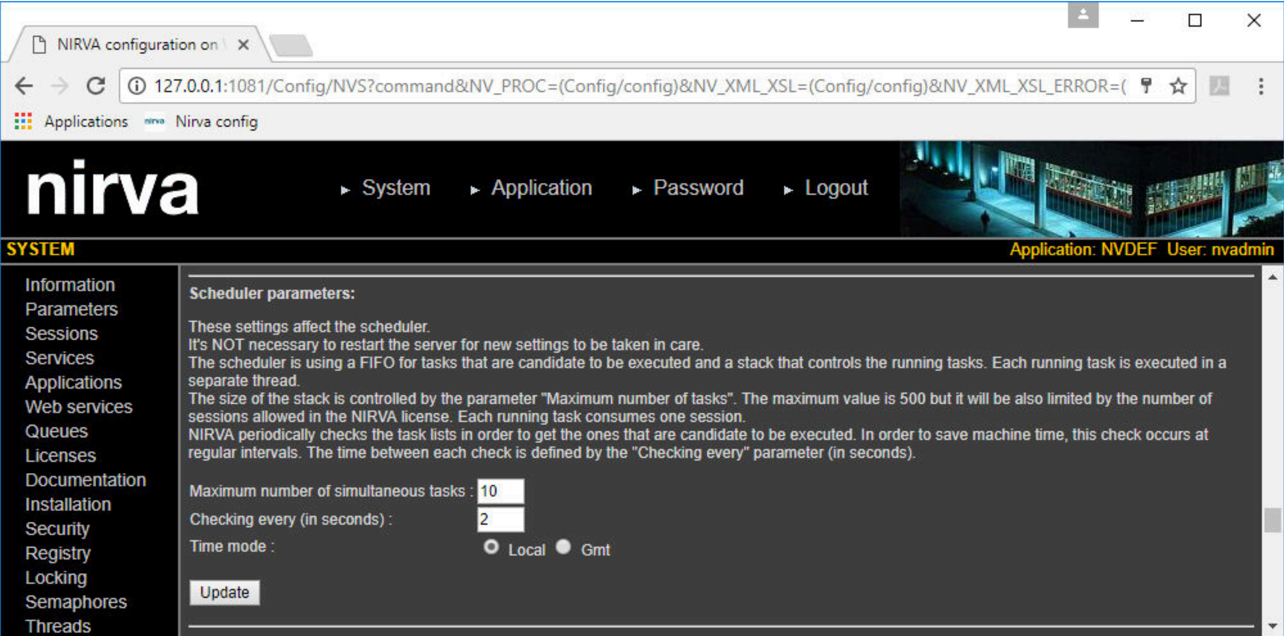
Transaction server parameters

These settings affect the transaction server.

It's NOT necessary to restart the server for new settings to be taken in care.

The transaction server is using a FIFO for transactions that are candidate to be executed and a stack that controls the running transactions. Each running transaction is executed in a separate thread. The size of the stack is controlled by the parameter "Maximum number of transactions". The maximum value is 500. Each running transaction consumes one session. NIRVA periodically checks the transaction lists in order to get the ones that are candidate to be executed. In order to save machine time, this check occurs at regular intervals. The time between each check is defined by the "Checking every" parameter (in seconds).

Scheduler parameters

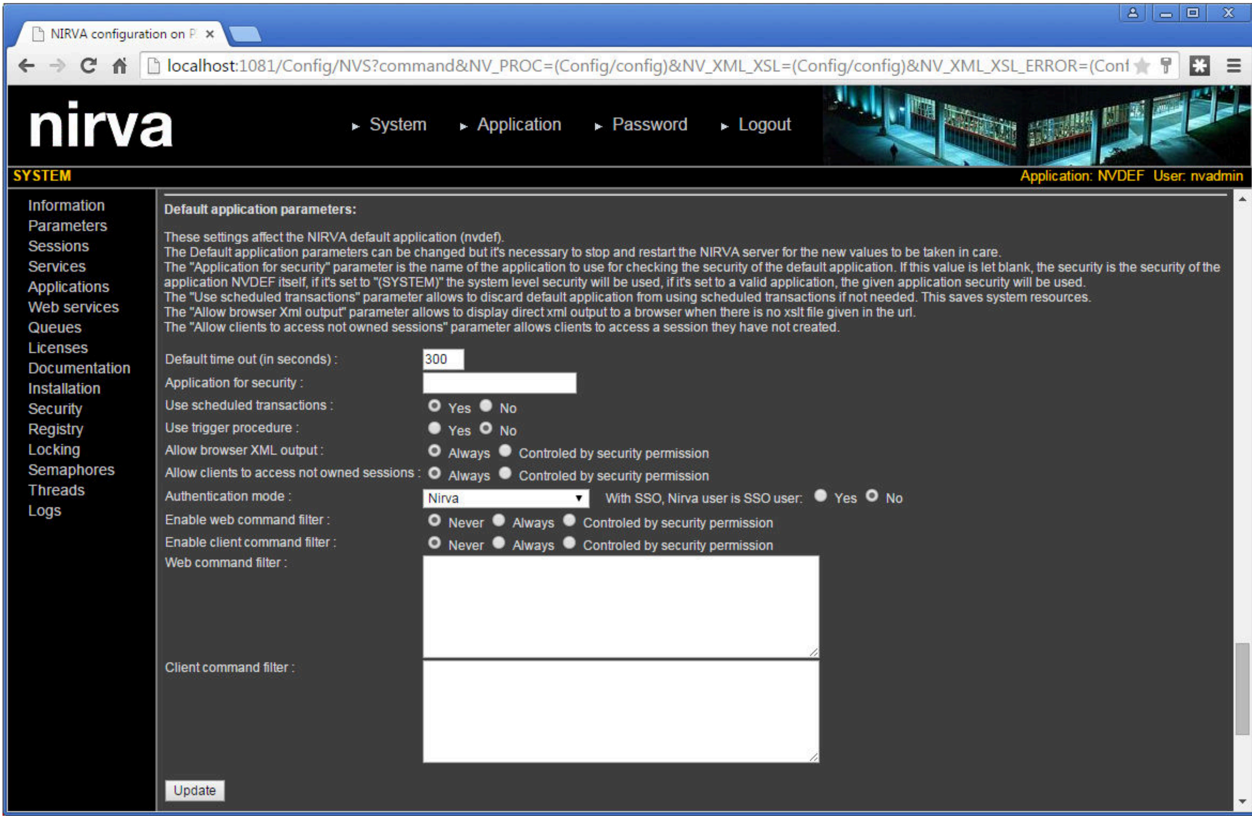


These settings affect the scheduler.

It's NOT necessary to restart the server for new settings to be taken in care.

The scheduler is using a FIFO for tasks that are candidate to be executed and a stack that controls the running tasks. Each running task is executed in a separate thread. The size of the stack is controlled by the parameter "Maximum number of tasks ". The maximum value is 500. Each running task consumes one session. NIRVA periodically checks the task lists in order to get the ones that are candidate to be executed. In order to save machine time, this check occurs at regular intervals. The time between each check is defined by the "Checking every" parameter (in seconds). Time mode allows choosing between gmt or local time for the scheduler.

Default application parameters



These settings affect the NIRVA default application (nvdef).

The Default application parameters can be changed but it's necessary to stop and restart the NIRVA server for the new values to be taken in care.

The “Application for security” parameter is the name of the application to use for checking the security of the default application. If this value is let blank, the security is the security of the application NVDEF itself, if it’s set to “(SYSTEM)” the system level security will be used, if it’s set to a valid application, the given application security will be used. When set to another application, one can define another server where the security application resides. For example `TEST[secu_server:1083]` tells that the NVDEF application uses the security of the TEST application installed on the server secu_server with TCP/IP port 1083. The default port is 1081. One can also use SSL for communicating with the distant security, at this time, the same example will be: `TEST[secu_server:1083(SSL)]`. The server address can be a virtual host defined in system parameters. At this time, the server name must be enclosed in brackets. Example: `TEST[(myvhost)]` without SSL and `TEST[(myvhost)(SSL)]` with SSL.

The "Use scheduled transactions" parameter allows discarding default application from using scheduled transactions if not needed. This saves system resources.

The "Use trigger procedure" parameter allows discarding the default application from using a trigger procedure if one has been defined in the application.dsc file.

The "Allow browser Xml output" parameter allows displaying direct xml output to a browser when there is no xslt file given in the url.

The "Allow clients to access not owned sessions" parameter allows authorizing Nirva clients to send commands to sessions they didn't create themselves. Nirva is using the sender TCP/IP address to check that.

The authentication mode allows choosing 3 options:

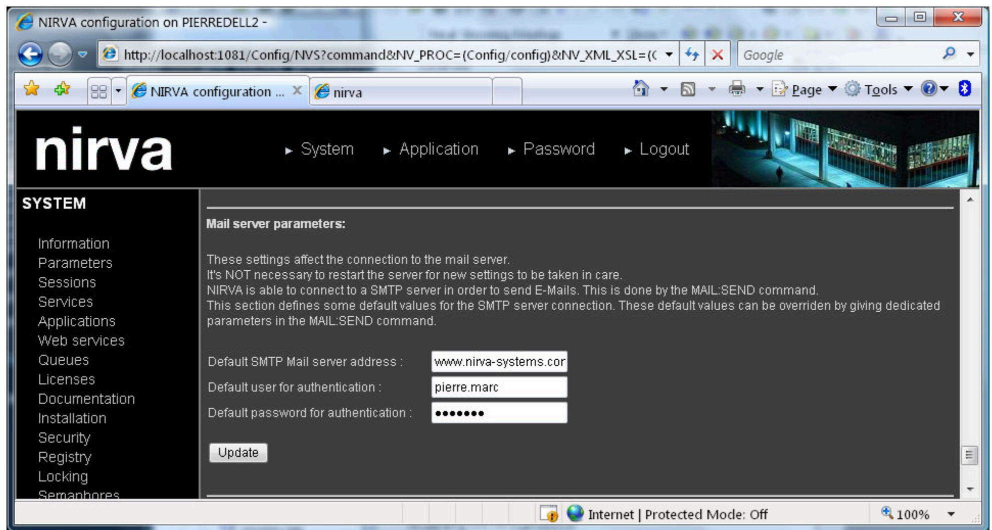
- Nirva: The users are authenticated by Nirva security of the application
- Single Sign-On (SSO): The users are authenticated by Kerberos using SSO. The SSO mode works only if both server and clients are on a domain and if Kerberos is available.
- Nirva or Single Sign-On. The users are authenticated by Nirva or Kerberos using SSO following a flag on the client side. This flag is available for all Nirva connectors using the Nirva client nvc library. Web clients cannot be authenticated by SSO in this mode.

When SSO is used, a parameter allows defining if the Nirva user of the session must be set to the SSO user or not. In any case, the SSO user and domain are available as session variables NV_SSO_USER and NV_SSO_DOMAIN.

Web and Client command filters allows authorising only some of the Nirva commands respectively from web and client connectors. A filter can be always activated or activated only when the user doesn't have a specific permission to bypass it (permissions are BYPASS_COMMAND_WEB_FILTER and BYPASS_COMMAND_CLIENT_FILTER). A filter is composed of a succession of filter lines starting with the word "allow" or "deny". It is then followed by a string of the form SERVICE:CLASS:COMMAND that defines what commands are authorised or allowed. By default there is nothing allowed. The wildchar "*" character can be used for SERVICE,CLASS or COMMAND. For example "deny *" denies all commands, "allow SYSTEM:MISC:*" allows all commands of the SYSTEM:MISC class. When a filter is defined for an application using web service, the command "SYSTEM:WEBSERVICE:EXECUTE" must be allowed in order to use the web services.

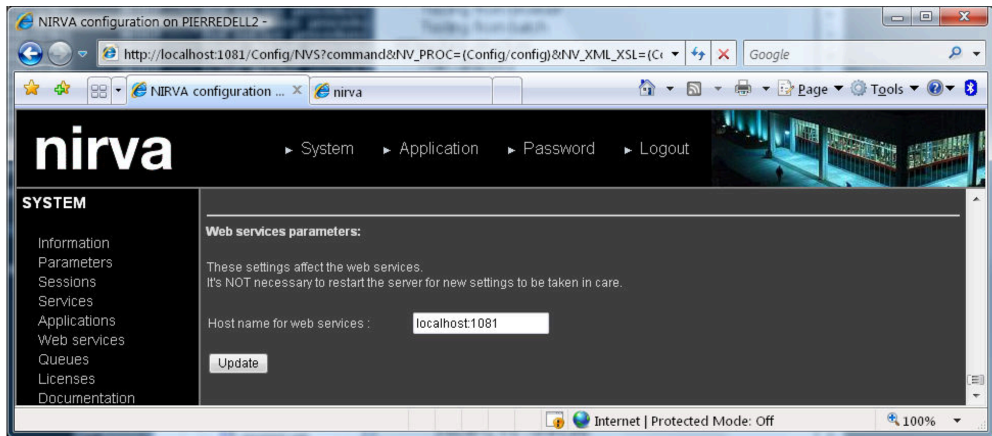
The web and client filters are not used for admin user or when the user connects localy (localhost or 127.0.0.1).

Mail server parameters



These settings affect the connection to the mail server. It's NOT necessary to restart the server for new settings to be taken in care. NIRVA is able to connect to a SMTP server in order to send E-Mails. This is done by the MAIL:SEND command. This section defines some default values for the SMTP server connection. These default values can be overridden by giving dedicated parameters in the MAIL:SEND command.

Web services parameters



These settings affect the web services. It's NOT necessary to restart the server for new settings to be taken in care.

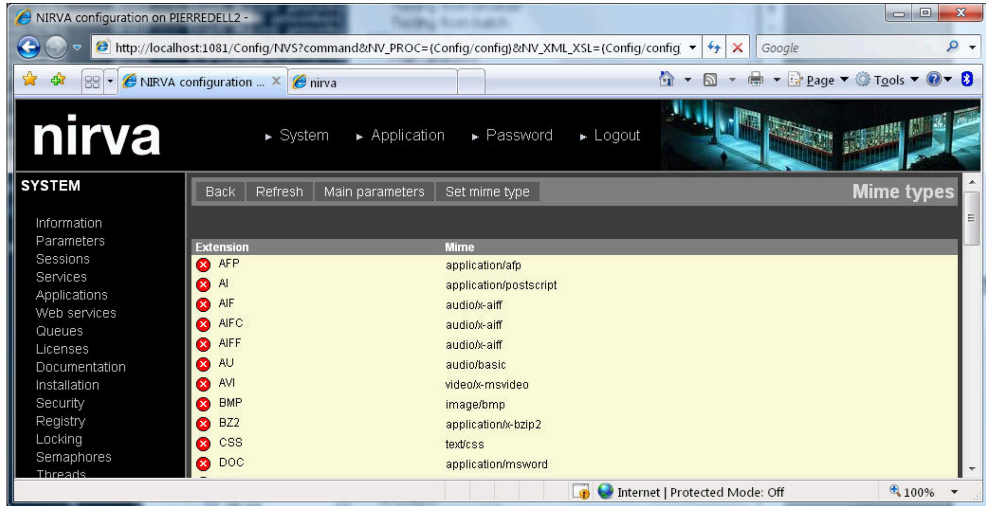
The only required parameter for web services is the nirva server host address that answers to the web service SOAP requests. The default is the server machine name followed by the server TCP/IP port if it's different of 80. If the web services must be accessed in HTTPS instead HTTP one must add “https://” before the host name, for example: “https://localhost:1082”. Several hosts can be defined separated by a semicolon. At this time Nirva will write a port for each host in the wsdl of the web service.

If one of the host name is set to “nv_dynamic_host” then the nirva server will use the value of the host given by the command that requests the wsdl (dynamic host).

The complete url of the web service, as seen by external people, is constructed using the host and the path defined at the web service level.

Mime types

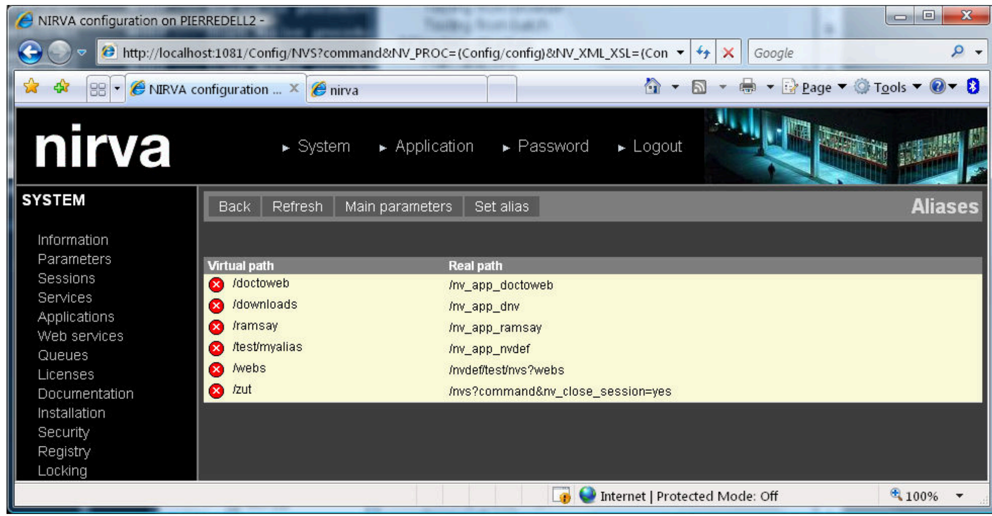
The mime type setting can be accessed via the “Mime types” button at the top of the screen:



Nirva come with some pre-defined mime types that can be changed from this menu.

Aliases

The alias button at the top of the Parameters screen allows defining HTTP aliases. This displays the following screen:



The virtual path is the path has it must be seen on the Url and the real path is the real nirva path

For example, if an application "MYAPP" has been created, one can create the alias "/myapp" pointing to real path "/nv_app_myapp". So the Urls "http://127.0.0.1/nv_app_myapp/test/login.htm" and "http://127.0.0.1/myapp/test/login.htm" will be the same.

Virtual hosts

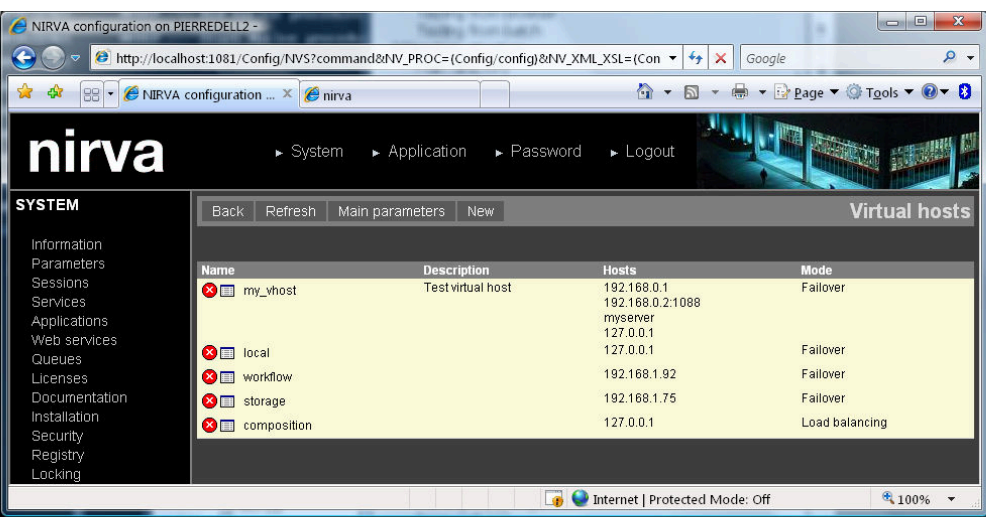
The "virtual hosts" button at the top of the Parameters screen allows defining virtual hosts. Virtual hosts can be used in communication channels between Nirva server (see the REQUEST class in the System service reference chapter). A virtual host is a named object pointing to several real servers. They can be configured in failover or load balancing mode.

In failover mode, Nirva will try to connect to the current active host first and then, if not successful, to the next available host of on the list.

In load balancing mode, Nirva will connect the host following the current active one.

The last connected host becomes the active one.

This “virtual hosts” button displays the following screen:



The new button allows creating a new virtual host.

A virtual host points to one or several physical addresses. A single address is composed by the physical address itself that can be the name of the server eventually followed by a ‘:’ character and the port number. If the port number is omitted, Nirva uses the default port 1081.

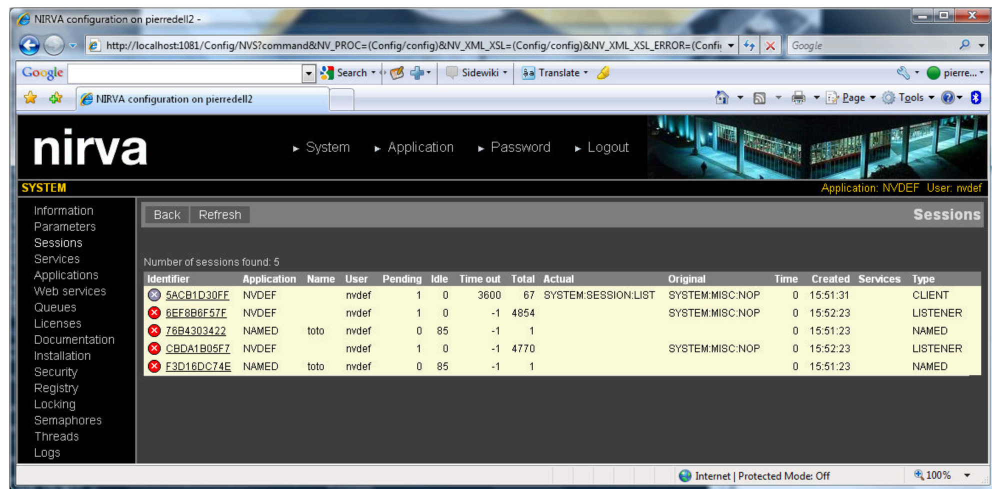
A proxy server can also be defined in a single address. At this time the format is the following:

proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address

Where *protocol* is the protocol for the proxy server. It can be “http” or “https”; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

Sessions

This option displays the complete list of current opened sessions on NIRVA.



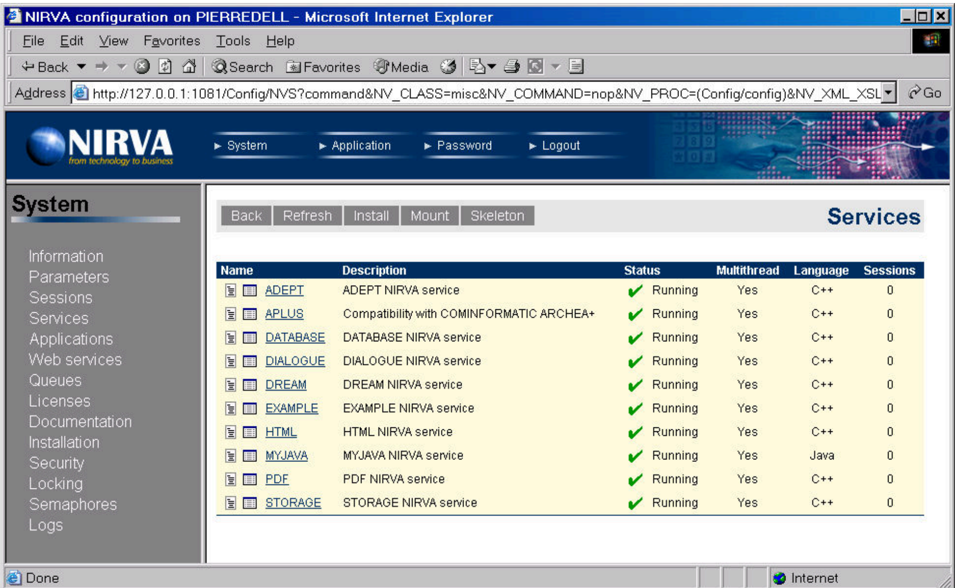
For each session, the list displays the following information:

- "Identifier" is the session unique identifier. It's possible to remove a session by clicking on the icon in the list. All sessions can be closed except the calling one. Before closing a session, NIRVA displays a confirmation message.
- "Application" is the name of the application connected by the session.
- "Name" is the optional session name (for a named session only).
- "User" is the application user name if there is one.
- "Pending" is the number of connections currently using the session (active command). This number will be generally 0 or 1.
- "Idle" is the number of seconds from the end of the last command.
- "Time out" is the session time out in seconds. A value of –1 means the session has an infinite time out and will close when the corresponding application is stopped.
- "Total" is the number of commands already processed by the session since it has been created.
- "Actual" is the command currently processed by the session. The current command is given on the form SERVICE:CLASS:COMMAND.
- "Original" is the original command which has generated the current command. The original command is always a command sent from a client (browser or service or internal) while the current command may come from a procedure or a service.
- "Time" is the number of seconds from the starting of the current command if there is one in process.
- "Created" is the date and time of the session creation. If the session has been created the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- "Services" is the list of services used by the session. It's possible to go directly to the specific service information by clicking on the service name.
- "Type" is the session type. It can be CLIENT, TRANSACTION, SCHEDULER, NAMED, SYSTEM, INTERNAL, LISTENER or THREAD.

When nirva is in debug mode one can display detailed session information by clicking on the session ID. This detail information also contains the current session stack in order to help debugging the code:

Services

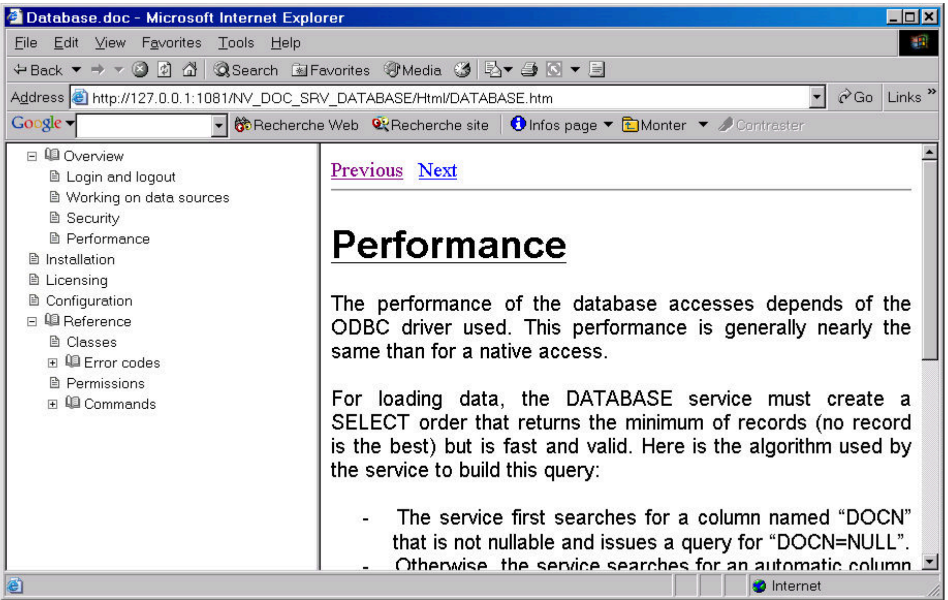
The services option displays the list of current mounted external services on NIRVA.



For each service, the list displays its name, description, status (running or stopped), the multithread flag, the service language and the number of sessions currently using the service.

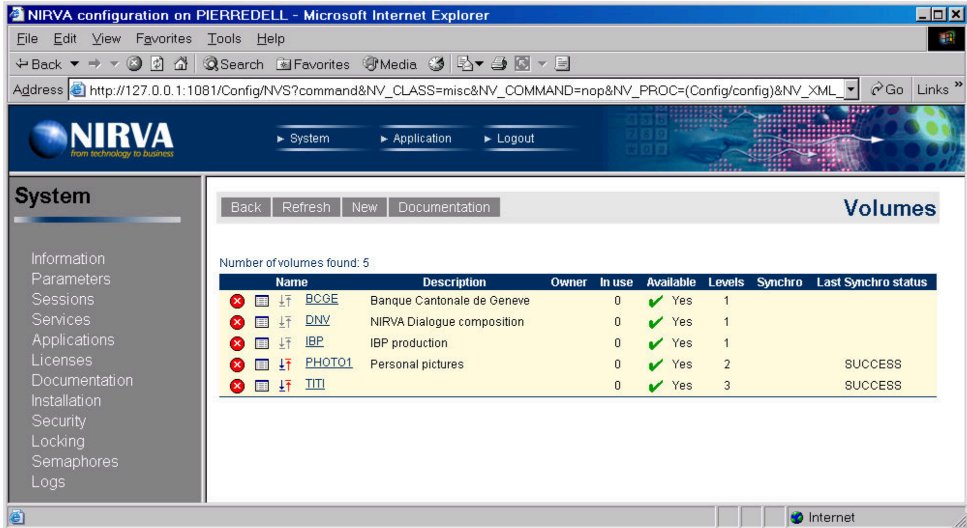
Service documentation

In order to display the service documentation, just click on the icon at the left of the service name. The documentation is then displayed in another browser window:



Service configuration

In order to display the service configuration screen, just click on the icon at the left of the service name:



Each service is responsible of its own documentation and configuration. The NIRVA SYSTEM:SERVICE:SKELETON command creates default documentation and configuration screens.

Starting and stopping a service

In order to start or stop a service, just click on the ✔ or ✘ icons at the left of the service status.

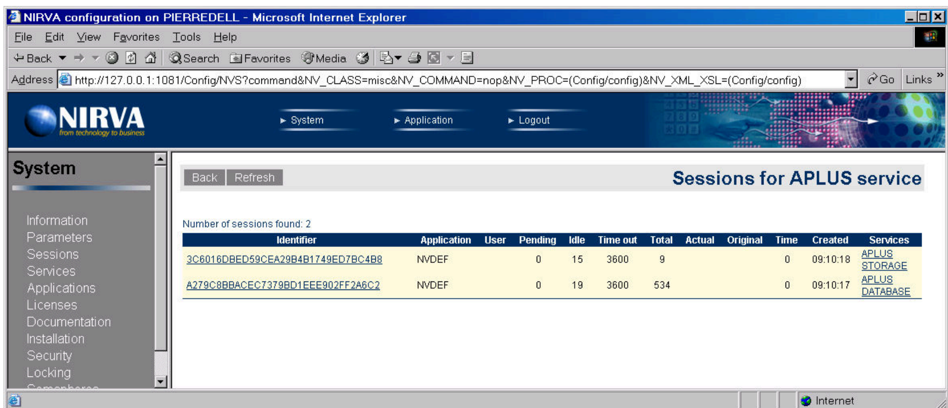
When stopping a service, NIRVA displays a confirmation message first.

Stopping a service consists of unloading its code from memory if it was loaded (the service is in fact a shared library). When a service is stopped, nobody can use it.

If a service is required by an application (listed in the Setting/required entry of the application dsc file), it cannot be stopped while the application is running.

Sessions used by a service

By clicking on the number of sessions (right column of the service list), NIRVA displays the detail list of sessions using this service:



The options available in this screen are the same than the ones described in the "Sessions" option of the configuration tool.

Detail service information

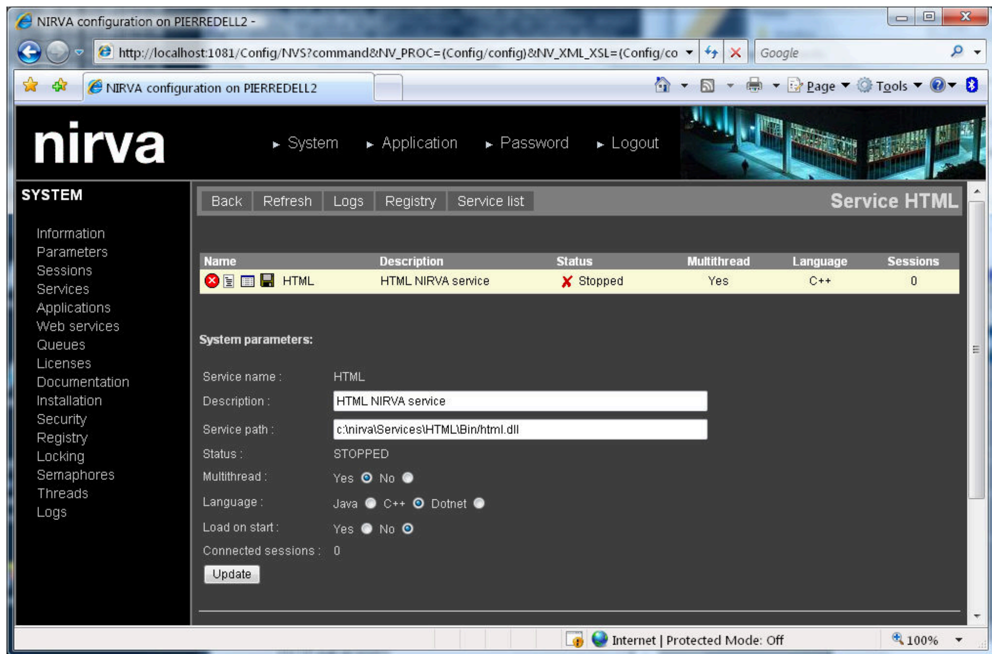
By clicking on the service name itself, NIRVA provides some detail information about the service:

There is only one item in the list for the required service.

The button allows packaging the service for it to be installed on another machine. The packaging is done with the default package.lst definition file. If any other installation package has to be done, please use the "`nvcc -i service_package.txt`" command line.

The "Service information" displays the information that can be found in the info section of the service description file. This file resides on the service files directory.

The "System parameters" are global service parameters managed at system level. It's possible to change them when the service is stopped:



The following service parameters can be changed:

Description

Service description. If the description is blank, NIRVA tries to get it from the SERVICE description file.

Service path

Path name of the library or class implementing the service. This can be a relative (to the service directory) or absolute path. If the service path is not given, NIRVA tries to locate it in the service Bin directory.

For a C++ service, this parameter is the path of the service library.

For a Java service, this parameter is the path to the Java class implementing the service. This can be in a jar file.

For a Dotnet service, this parameter is the path to the assembly file (a dll file) containing the service class implementing the service.

Multithread

If this parameter is set to "No", the library is considered as not multithread and NIRVA will serialize all the commands sent to it.

Language

This tells Nirva in which language the service is written. It can be Java, Dotnet or C++.

Load on start

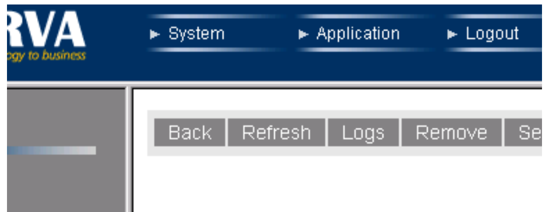
If this parameter is set to “Yes”, the library will be loaded when starting the Nirva server. Otherwise, the library is loaded only when the first command is sent to it. Generally, a library should not be loaded at start time if not necessary in order to save memory space.

Also, when the service is stopped, it can be un-mounted from the NIRVA service list by clicking on the icon. Un-mounting a service means removing the link between Nirva and the library.

When a service is mounted, Nirva creates some subdirectories in the Nirva service directory. These subdirectories are not removed when un-mounting a service.

Working with service logs

The service logs are available when displaying the detail service information. In order to access service logs, just press the “Logs” link at the top of the service detail information screen:

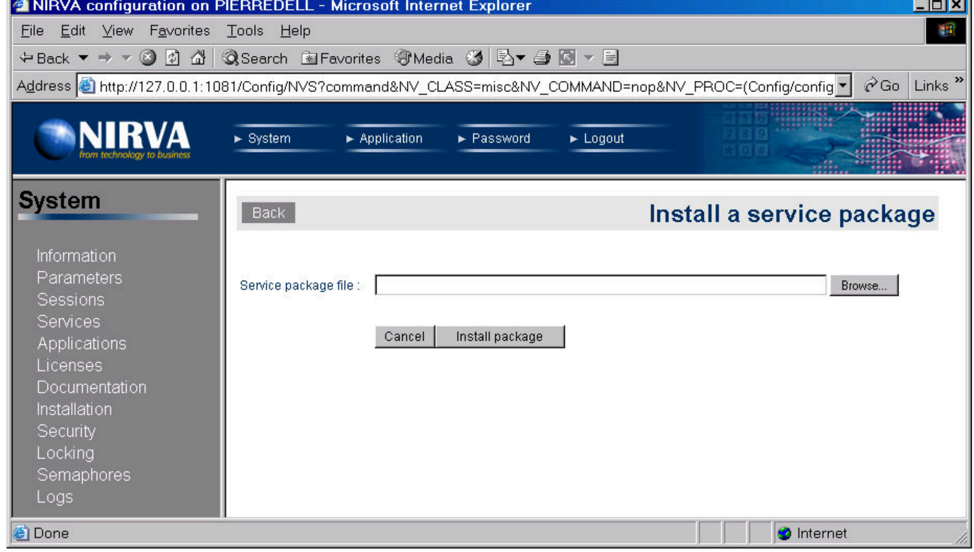


The log management at service level is the same than the log management at system level but it allows to access service logs only.

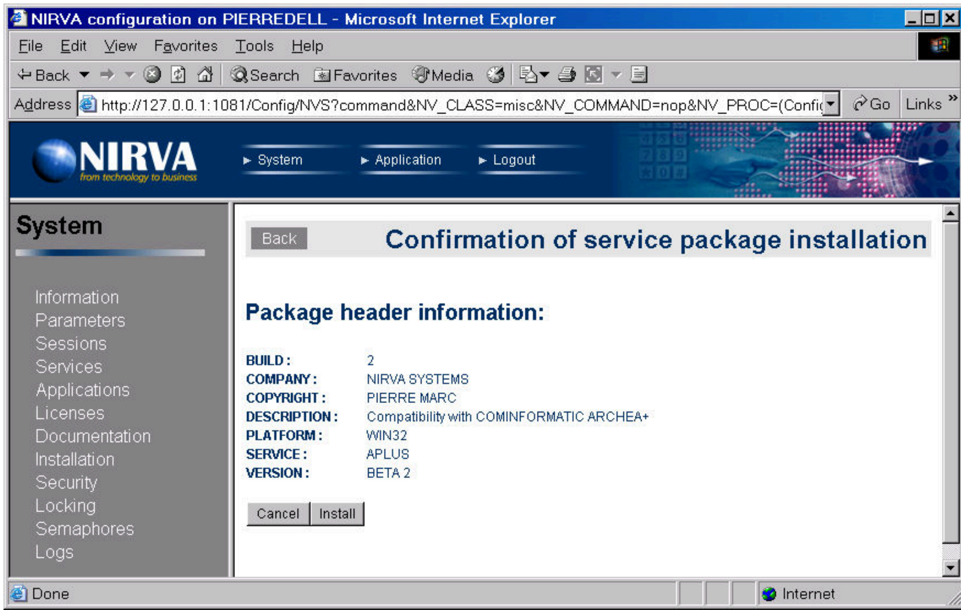
Please refer to the system configuration logs for further information about log features.

Installing a service package

A Nirva service is delivered as a NIRVA package file. The package file can contain the whole service or simply some patches. In order to install a service package, just click on the “Install” menu at the top of the “Service list” screen. This displays the following screen:



After pressing the “Install package” button, NIRVA displays a confirmation screen with some information extracted from the package file:

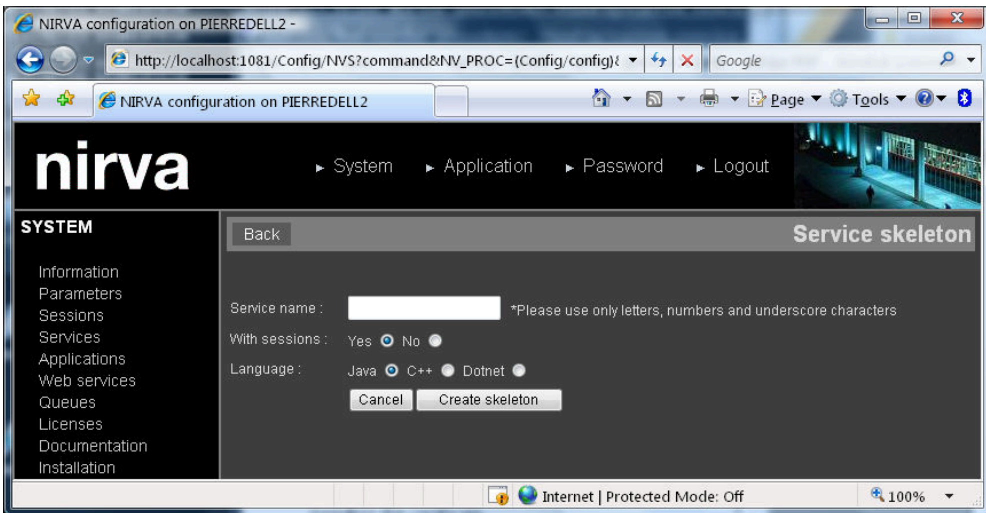


If the service to install is currently running, the command fails. A service package can only be installed if the service is stopped (or not yet installed).

Creating a new service

A Nirva service is a shared library that implements some commands. NIRVA provides a command named “SKELETON” allowing creating all the service source code and associated files. Please see the service tutorial or the description of the SYSTEM:SERVICE:SKELETON command for further information.

In order to create a service skeleton, just click on the “Skeleton” menu at the top of the “Service list” screen. This displays the following screen:



The service name is name of the new service. A service name should contain only alphanumeric characters and the underscore character.

The “With sessions” option tells if the generated source code must implement service session management or not.

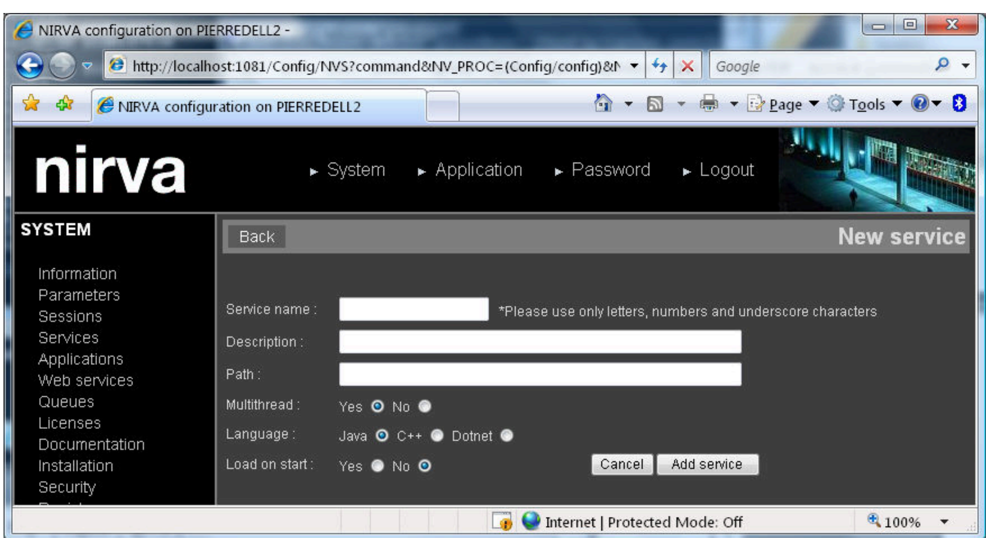
The “language” option tells the language (C++, Dotnet or Java) to be used for developing the service.

After the creation of skeleton, NIRVA displays back the service list but a new directory has been created in the NIRVA service directory. The user just has to go to the new service source directory and to compile it. The service can then be mounted from the service list screen.

Mounting a service

Mounting a service means telling to Nirva what is the name of the service and its path. The service is then added to the NIRVA service list. In order to mount a service, just click on the “Mount” button at the top of the service list screen:

NIRVA then displays the following screen:



Service name	Name of the external service to mount. The service name is case insensitive. It should not contain space or any special character.
Description	Service description. If the description is blank, NIRVA tries to get it from the SERVICE description file.
Path	Complete path name of the library or class implementing the service. If the service path is not given, NIRVA tries to locate it in the service Bin directory. For a C++ service, this parameter is the path of the service library. For a Java service, this parameter is the path to the Java class implementing the service. This can be in a jar file.

For a Dotnet service, this parameter is the path to the assembly file (a dll file) containing the service class implementing the service.

Multithread — If this parameter is set to “No”, the library is considered as not multithread and NIRVA will serialize all the commands sent to it.

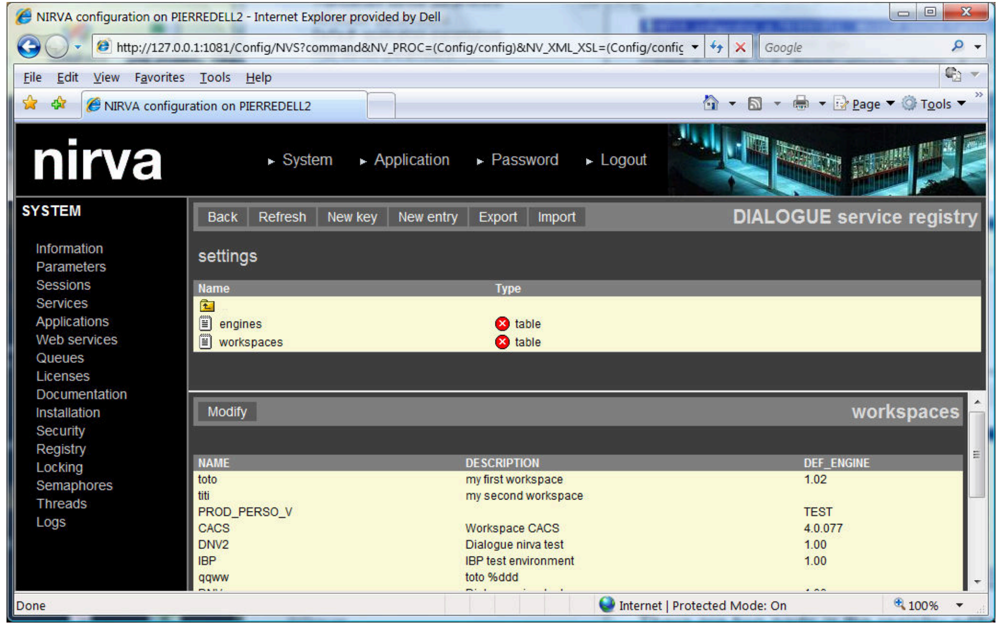
Language — This tells Nirva in which language the service is written. It can be Java, Dotnet or C++.

Load on start — If this parameter is set to “Yes”, the library will be loaded when starting the Nirva server. Otherwise, the library is loaded only when the first command is sent to it. Generally, a library should not be loaded at start time if not necessary in order to save memory space.

When installing a new service package, the service is automatically mounted.

Accessing service registry

The “Registry” button in detail service information screen allows viewing and editing the service level registry:

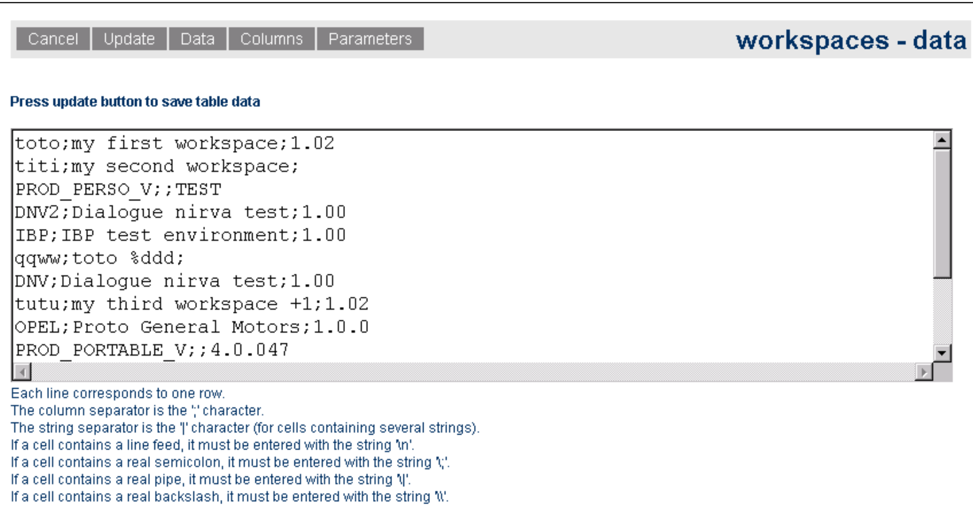


There are two parts in the registry editor.

The upper part displays the current registry key. One can add, modify or remove entries.

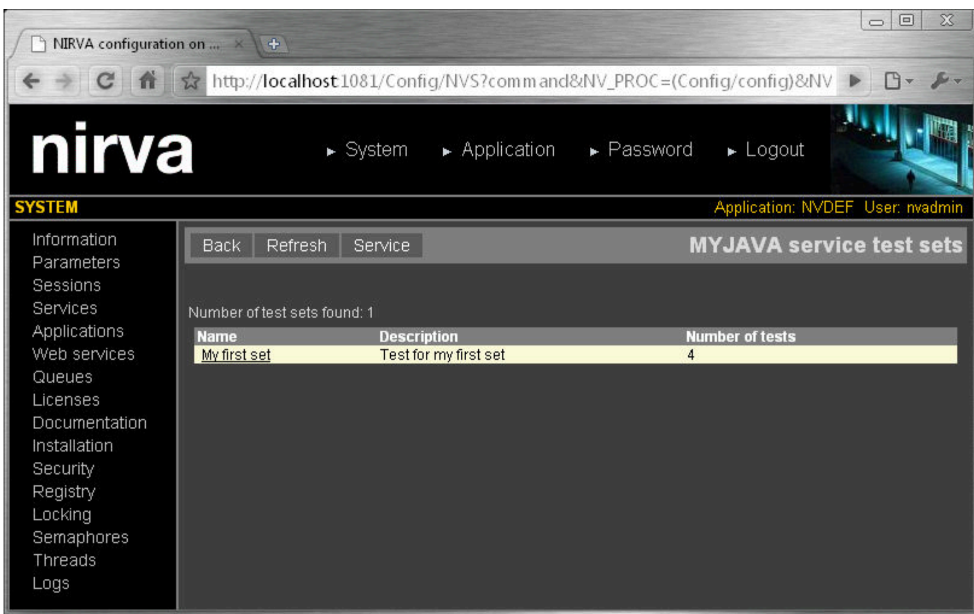
The lower part displays detail information about a given entry and allows editing it.

Specific entry instructions are displayed at the bottom of the screen when editing an entry. Here is an example for the table editing:

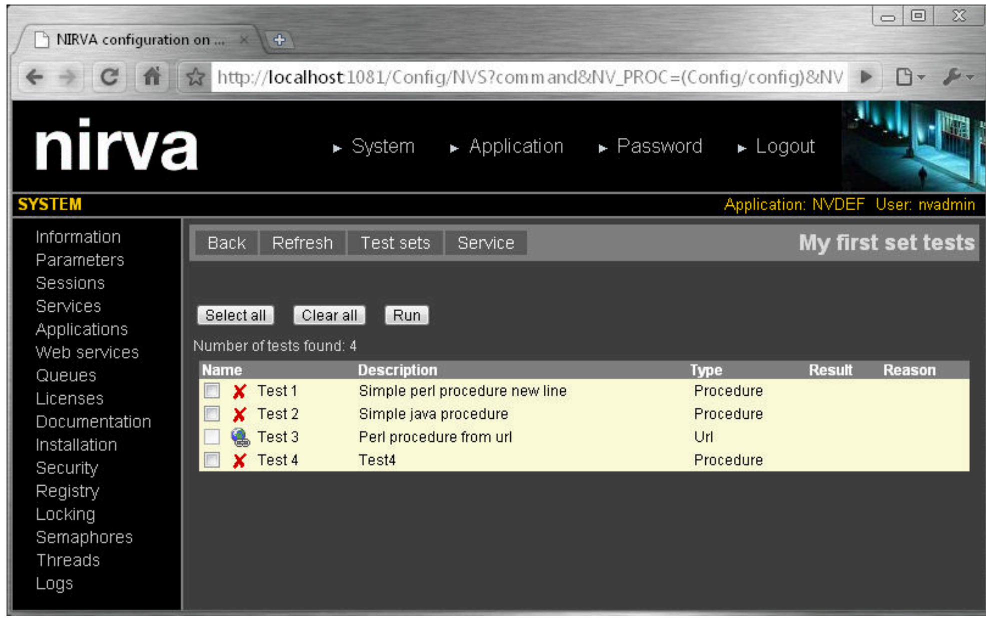


Service tests

The “Tests” button in detail service information screen displays the list of defined test sets for the service (see the chapter “Test sets” in this documentation). It allows running the tests.



In order to run the tests defined for a test set, just click on its name:



"Select all" allows selecting all the tests of the set having the type "Procedure".

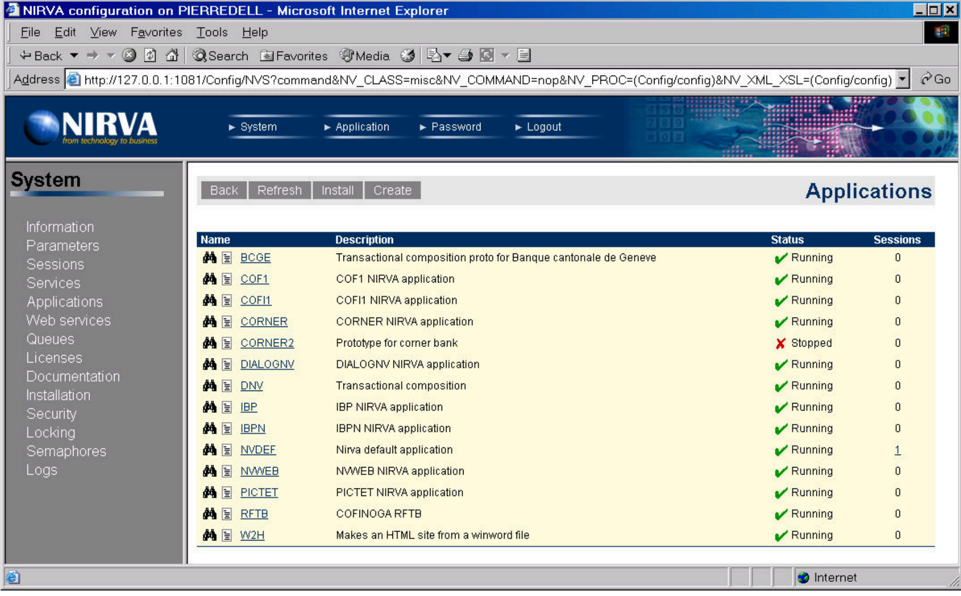
"Clear all" resets the result and the selection of all the tests.

"Run" starts the selected tests of type "Procedure". When running, the tests can be stopped by pressing the "Cancel" button.

In order to start an "Url" test type, just click on the the test icon.

Applications

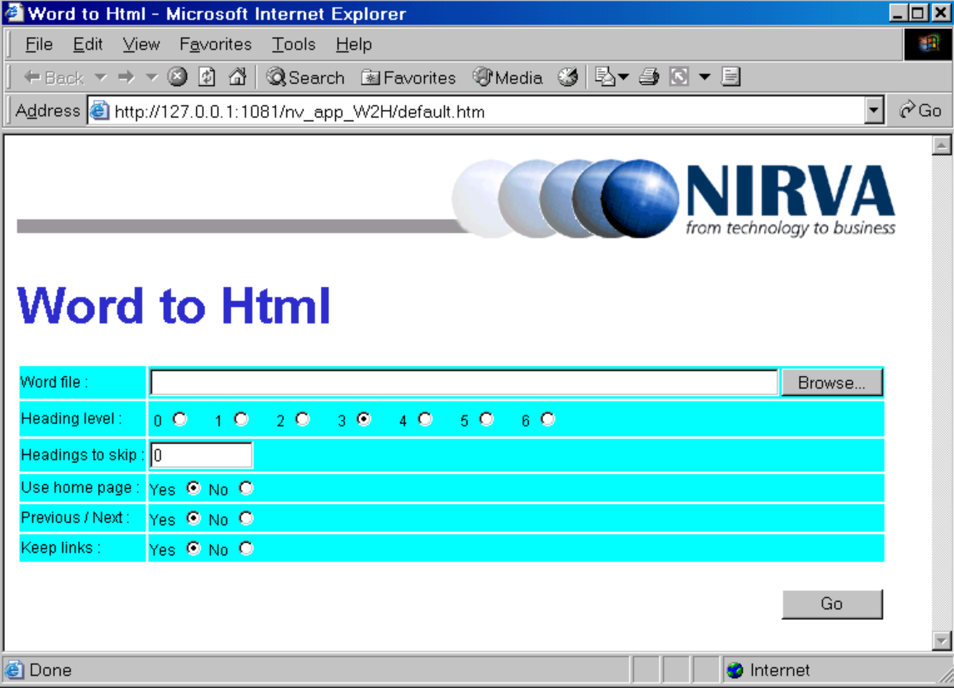
The applications option displays the list of current applications on NIRVA. The "System applications" option is different than the global "Application" option. The "System applications" option concerns the list of application while the global "Application" option concerns only the connected application.



For each application, the list displays its name, description, status (running or stopped) and the number of sessions currently using the application.

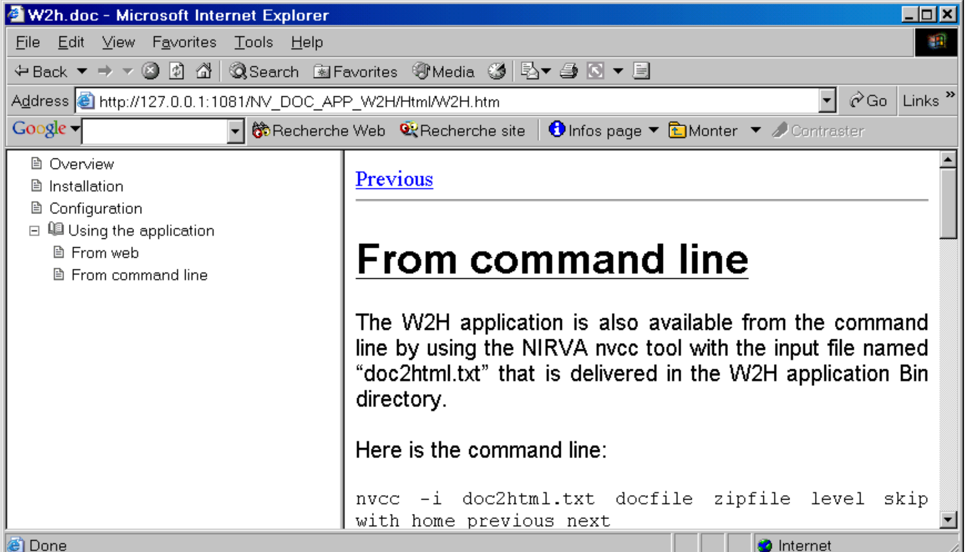
Application start page

In order to display the application start page, just click on the icon at the left of the application name. The application start page is then displayed in another browser window. This is a convenient way to start applications.



Application documentation

In order to display the application documentation, just click on the icon at the left of the application name. The documentation is then displayed in another browser window:



Starting and stopping an application

In order to start or stop an application, just click on the or icons at the left of the application status.

When stopping an application, NIRVA displays a confirmation message first.

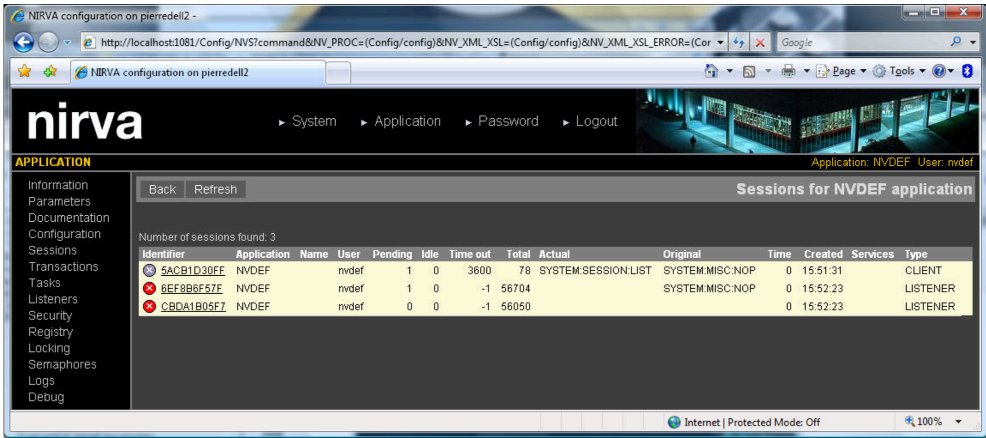
Stopping an application means disabling it and closing all sessions connected to it.

This command may be long because it closes all the sessions using the application and waits for them to be completely terminated.

It's not possible to stop its own application. At this time, the command fails. It's also not possible to stop the NIRVA default application (NVDEF application).

Sessions used by an application

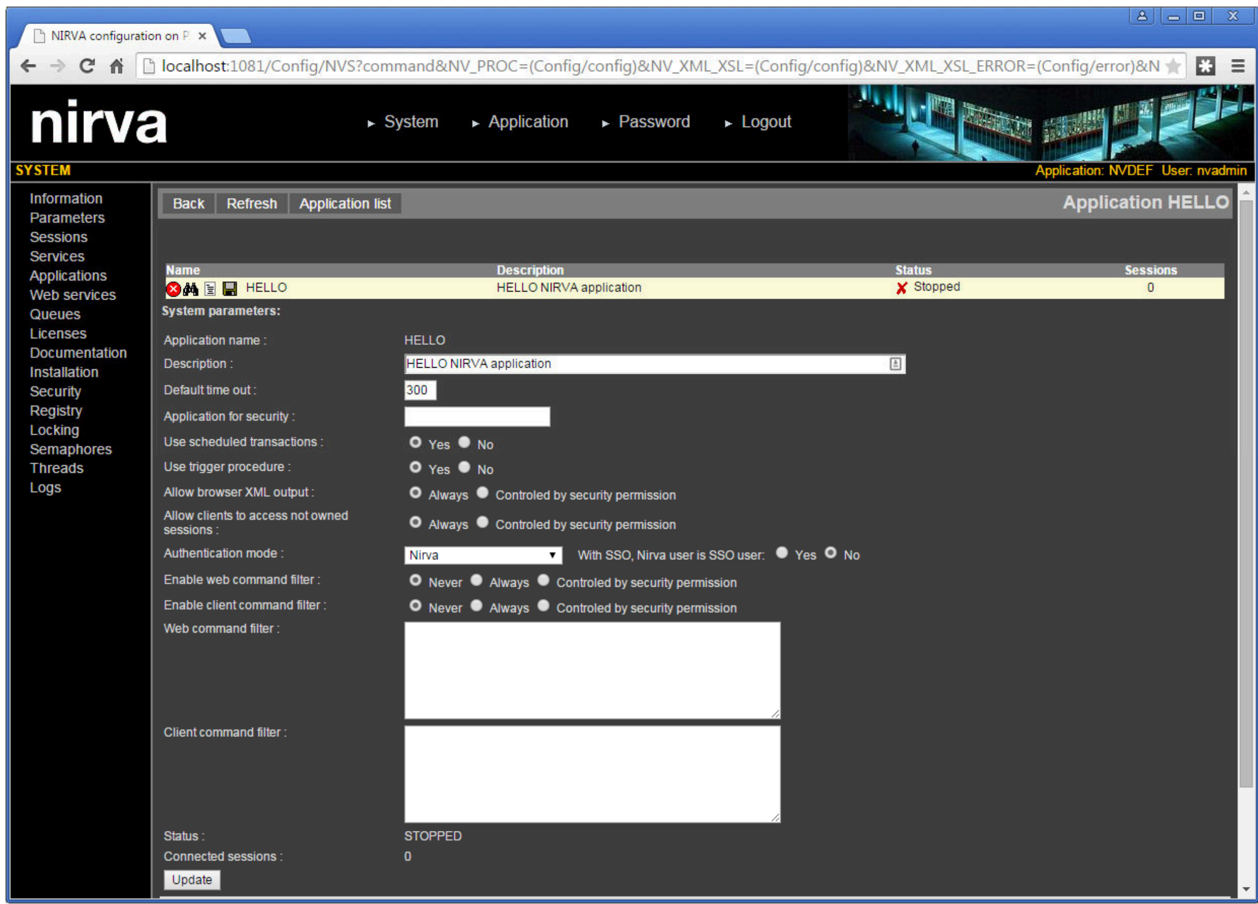
By clicking on the number of sessions (right column of the application list), NIRVA displays the detail list of sessions using this application:



The options available in this screen are the same than the ones described in the “System/Sessions” option of the configuration tool.

Detail application information

By clicking on the application name itself, NIRVA provides some detail information about the application:



The button allows packaging the application for it to be installed on another machine. The packaging is done with the default package.lst definition file. If any other installation package has to be done, please use the "`nvcc –i application_package.txt`" command line.

The "Application information" displays the information that can be found in the info section of the application description file. This file resides on the application files directory.

The "System parameters" are global application parameters managed at system level. It's possible to change them when the application is stopped.

The "Application for security" parameter is the name of the application to use for checking the security of the default application. If this value is let blank, the security is the security of the application itself, if it's set to "(SYSTEM)" the system level security will be used, if it's set to a valid application, the given application security will be used. When set to another application, one can define another server where the security application resides. For example `TEST[secu_server:1083]` tells that the NVDEF application uses the security of the TEST application installed on the server secu_server with TCP/IP port 1083. The default port is 1081. One can also use SSL for communicating with the distant security, at this time, the same example will be: `TEST[secu_server:1083(SSL)]`. The server address can be a virtual host defined in system parameters. At this time, the server name must be enclosed in brackets. Example: `TEST[(myvhost)]` without SSL and `TEST[(myvhost)(SSL)]` with SSL. Finally the security can be defined to be implemented in a specific security service (ex LDAP). At this time, the name of the service enclosed in brackets must be used. For example "(MYSECU)" tells Nirva to use the security defined on the service "MYSECU" instead of the nirva security. See the chapter about security model for further information about nirva security and security services.

The "Use scheduled transactions" parameter allows discarding the application from using scheduled transactions if not needed. This saves system resources.

The "Use trigger procedure" parameter allows discarding the application from using a trigger procedure if one has been defined in the application.dsc file.

The "Allow browser Xml output" parameter allows displaying direct xml output to a browser when there is no xslt file given in the url.

The "Allow clients to access not owned sessions" parameter allows authorizing Nirva clients to send commands to sessions they didn't create themselves. Nirva is using the sender TCP/IP address to check that.

The authentication mode allows choosing 3 options:

- Nirva: The users are authenticated by Nirva security of the application
- Single Sign-On (SSO): The users are authenticated by Kerberos using SSO. The SSO mode works only if both server and clients are on a domain and if Kerberos is available.
- Nirva or Single Sign-On. The users are authenticated by Nirva or Kerberos using SSO following a flag on the client side. This flag is available for all Nirva connectors using the Nirva client nvc library. Web clients cannot be authenticated by SSO in this mode.

When SSO is used, a parameter allows defining if the Nirva user of the session must be set to the SSO user or not. In any case, the SSO user and domain are available as session variables NV_SSO_USER and NV_SSO_DOMAIN.

Web and Client command filters allows authorising only some of the Nirva commands respectively from web and client connectors. A filter can be always activated or activated only when the user doesn't have a specific permission to bypass it (permissions are BYPASS_COMMAND_WEB_FILTER and BYPASS_COMMAND_CLIENT_FILTER). A filter is composed of a succession of filter lines starting with the word “allow” or “deny”. It is then followed by a string of the form SERVICE:CLASS:COMMAND that defines what commands are authorised or allowed. By default there is nothing allowed. The wildchar “*” character can be used for SERVICE,CLASS or COMMAND. For example “deny *” denies all commands, “allow SYSTEM:MISC:*” allows all commands of the SYSTEM:MISC class. When a filter is defined for an application using web service, the command “SYSTEM:WEBSERVICE:EXECUTE” must be allowed in order to use the web services.

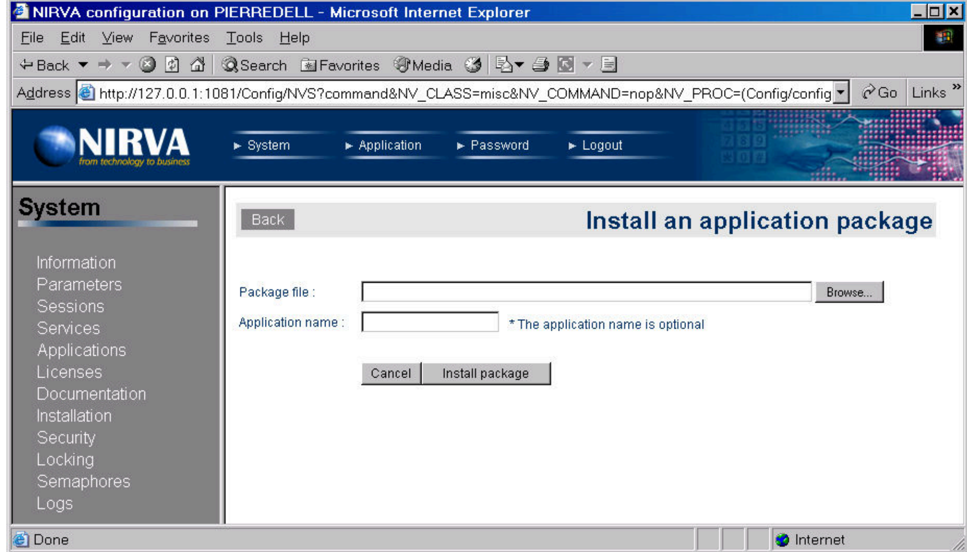
The web and client filters are not used for admin user or when the user connects localy (localhost or 127.0.0.1).

When the application is stopped, it can be removed from the NIRVA application list by clicking on the icon. Removing an application means removing the link between Nirva and the application.

When an application is created or installed, Nirva creates some subdirectories in the Nirva application directory. These subdirectories are not removed when removing an application.

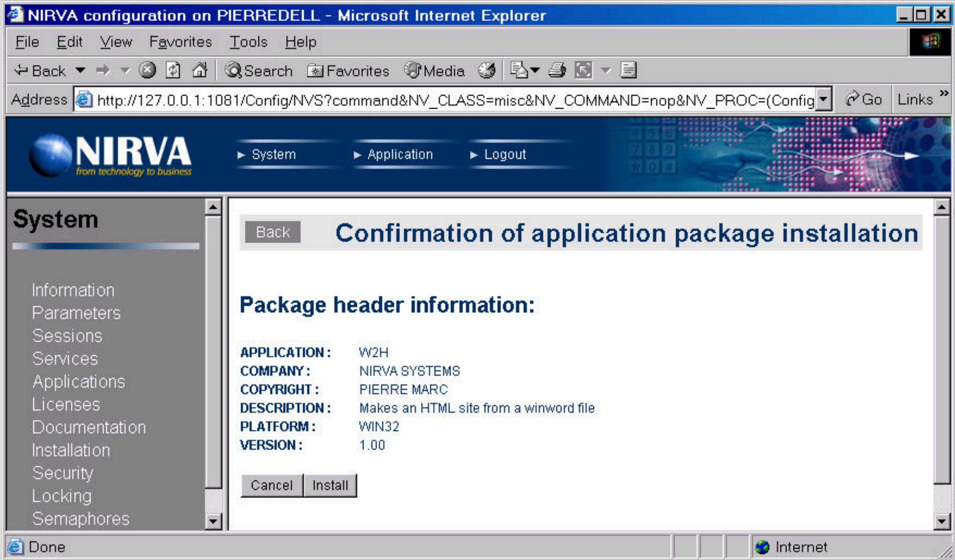
Installing an application package

A Nirva application is delivered as a NIRVA package file. The package file can contain the whole application or simply some patches. In order to install an application package, just click on the “Install” menu at the top of the “Application list” screen. This displays the following screen:



The application name is optional. If it's not provided, NIRVA tries to get it from the package file (this is generally the case but the application name can be overridden allowing to install an application under a different name).

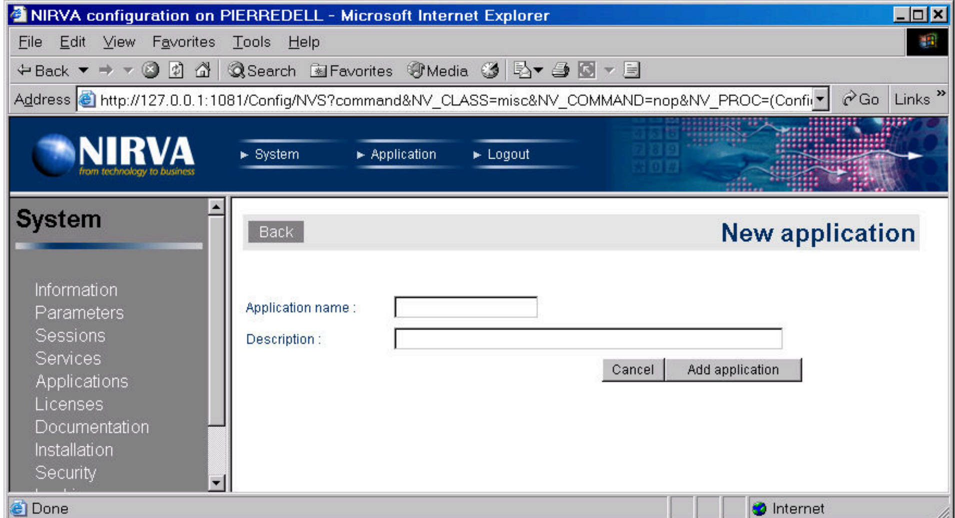
After pressing the "Install package" button, NIRVA displays a confirmation screen with some information extracted from the package file:



If the application to install is currently running, the command fails. An application package can only be installed if the application is stopped (or not yet installed).

Creating a new application

In order to create a new application, just click on the "Create" menu at the top of the "Application list" screen. This displays the following screen:

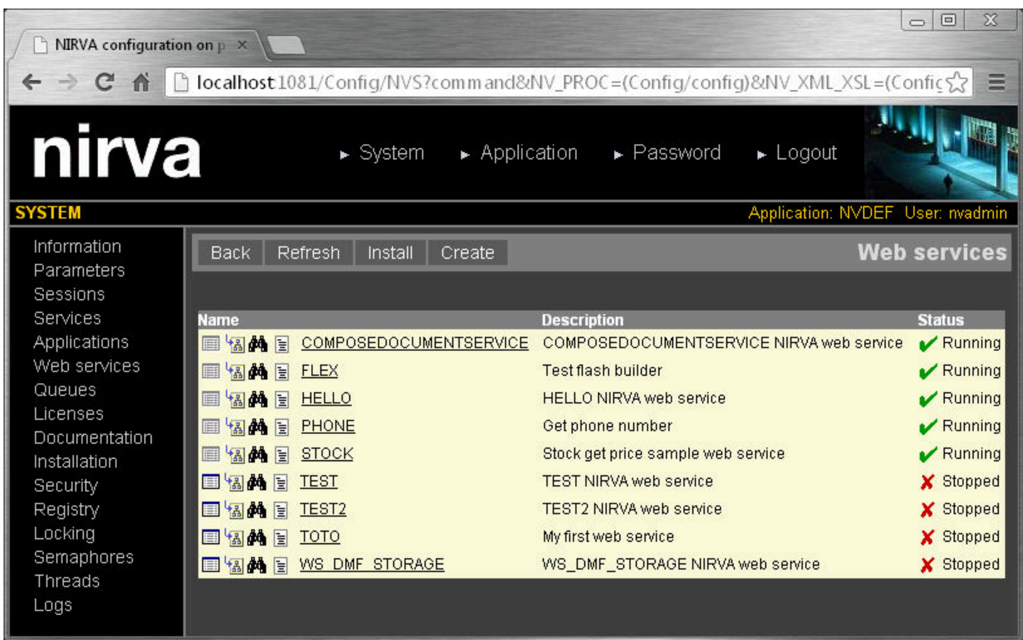


The application name is name of the new application. An application name should contain only alphanumeric characters and the underscore character. It's mandatory.

After the creation of the application, NIRVA displays back the application list containing the new application. Also, a new directory has been created in the NIRVA application directory.

Web services

The web service option displays the list of current web services defined on NIRVA:



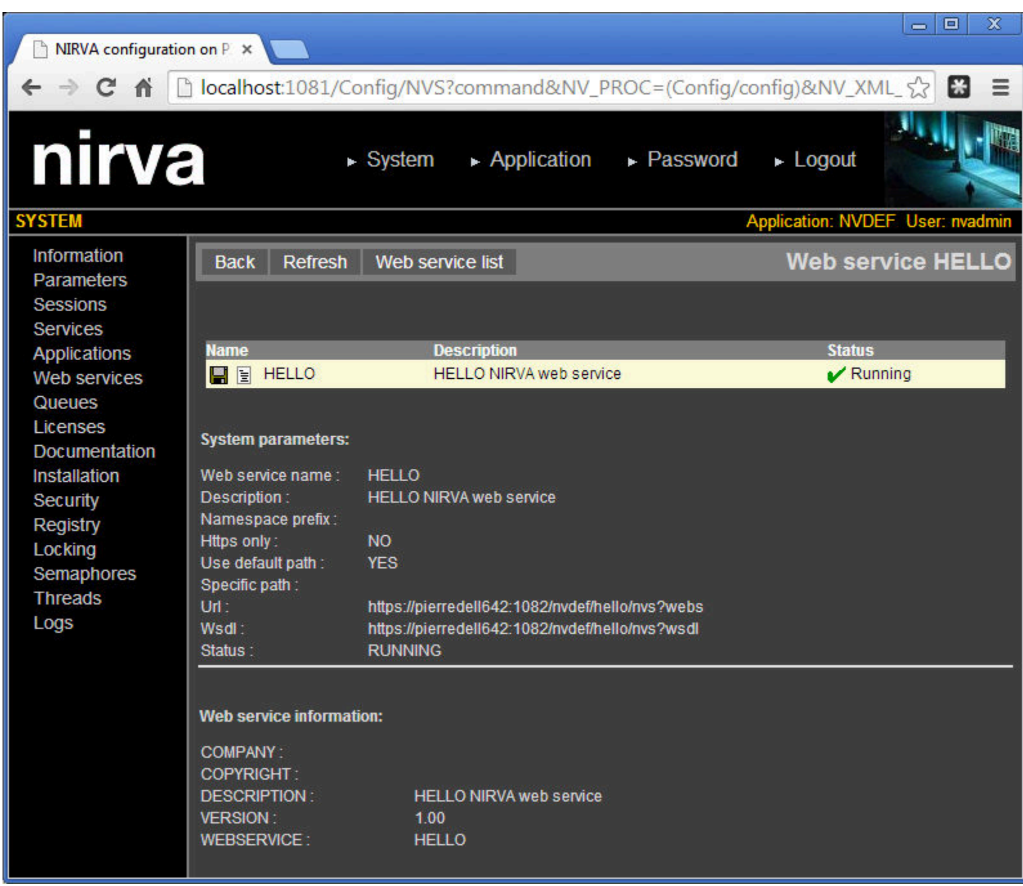
For each web service, the list displays its name, description and status. From this window, one can do the following functions:

- View detail information about a web service
- Display web service documentation
- Create a new web service

- Start and stop a web service
- Display a web service content (operations and messages)
- Edit a web service content (operations and messages)
- View a web service WSDL definition
- Install a web service
- Remove a web service

View detail web service information

By clicking on the web service name itself, NIRVA provides some detail information about the web service:



The button allows packaging the web service for it to be installed on another machine. The packaging is done with the default package.lst definition file. If any other installation package has to be done, please use the “`nvcc -i webservice_package.txt`” command line.

The “System parameters” are global web service parameters managed at system level. When the web service is stopped, it’s possible to change its description. The namespace prefix is being used in namespace of the WSDL. If let blank, Nirva uses the information “Host name for web services” found in the main system parameters (web service parameters section). Url and Wsdl information respectively gives the URLs to run the web service and to get its WSDL definition. These URLs contain the name of the connected application (nvdef for the default one) after the server name because a web service is always executed in the context of an application so the real URLs to use must contain the name of the application that runs the web service.

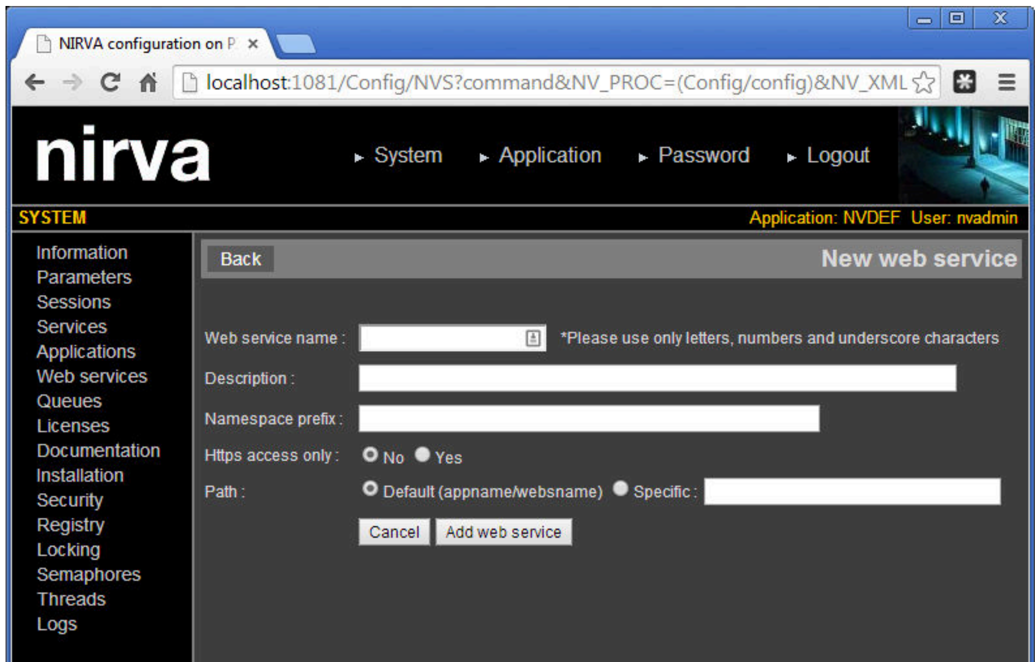
The “Web service information” displays the information that can be found in the info section of the web service description file. This file resides on the web service files directory.

When the web service is stopped, it can be removed from the NIRVA web service list by clicking on the icon. Removing a web service means removing the link between Nirva and the web service.

When a web service is created or installed, Nirva creates some subdirectories in the Nirva Webservices directory. These subdirectories are not removed when removing a web service.

Creating a new web service

In order to create a new web service, just click on the “Create” button at the top of the “Web service list” screen. This displays the following screen:



The web service name is the name of the new web service. A web service name should contain only alphanumeric characters and the underscore character. It’s mandatory.

The namespace prefix is being used in namespace of the WSDL. If let blank, Nirva uses the information “Host name for web services” found in the main system parameters (web service parameters section). When installing a new web service, the namespace prefix information is taken from the web service description file (“SETTINGS” section, “NSPREFIX” entry).

Https access only allows restricting access to the web service to HTTPS protocol. Attempting to access a web service with HTTP protocol will result in an error “invalid web service”.

The “Path” parameter is added to the host name defined in the system parameters in order to construct the url of the web service in the wsdl. One can use the default path (appname/websname) or a specific path. This can be usefull when there is reverse proxy or an allias. In any case the rea l access to the web service must be made using the syntax defined in the Nirva command syntax chapter. Particularly the url really transmitted to nirva (eventually decoded using an allias) must finish with appname/websname where appname is the name of the application and websname is the name of the web service.

The specific Path should not contain the string nvs?webs. if found, Nirva truncates the path before. The same if a "?" character is found.

After the creation of the web service, NIRVA displays back the web service list containing the new web service. Also, a new directory has been created in the NIRVA web service directory.

Starting and stopping a web service

In order to start or stop a web service, just click on the ✔ or ✘ icons at the left of the web service status.

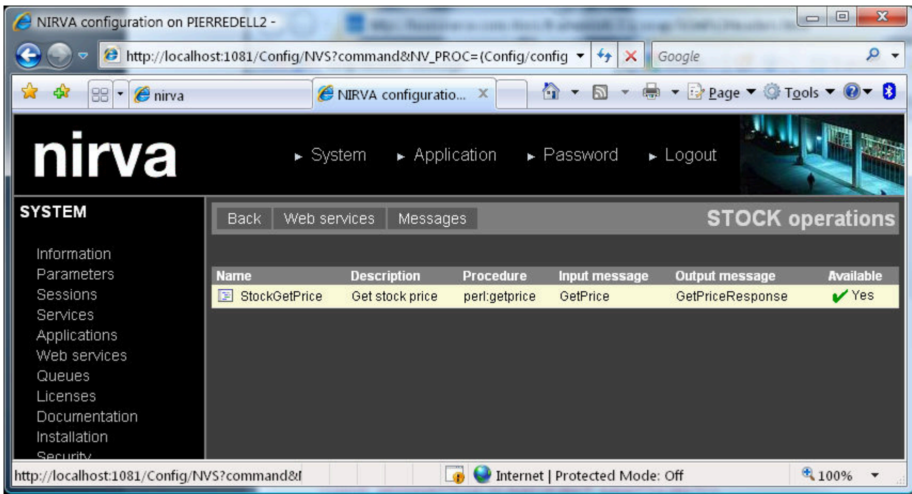
When stopping a web service, NIRVA displays a confirmation message first.

Stopping a web service means disabling it so it becomes unavailable to applications.

Display web service content

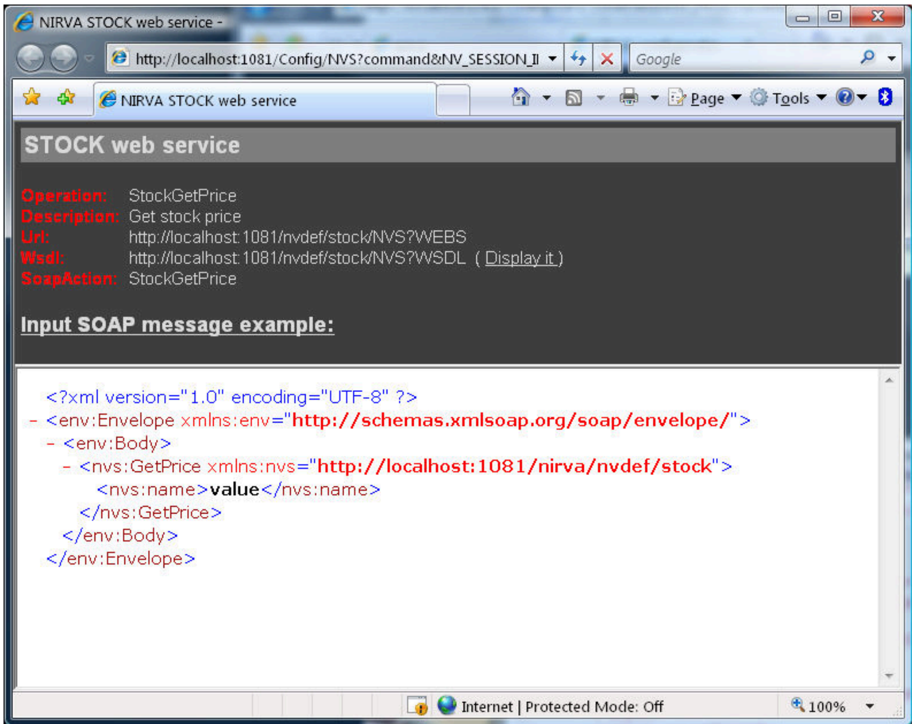
For displaying the web service content (operations and messages), just click on the icon near its name from the web service list screen. This displays the list of defined operations for the selected web service. Then it's possible to switch between operation and message views by clicking on the corresponding button (Messages or Operations buttons).

Operations:



For each operation, the table displays the operation name, the operation description, the name of the NIRVA procedure that processes the operation, the name of associated input and output messages and the availability of the operation. If one of the input or output message is not defined a ⚠ sign is displayed near it.

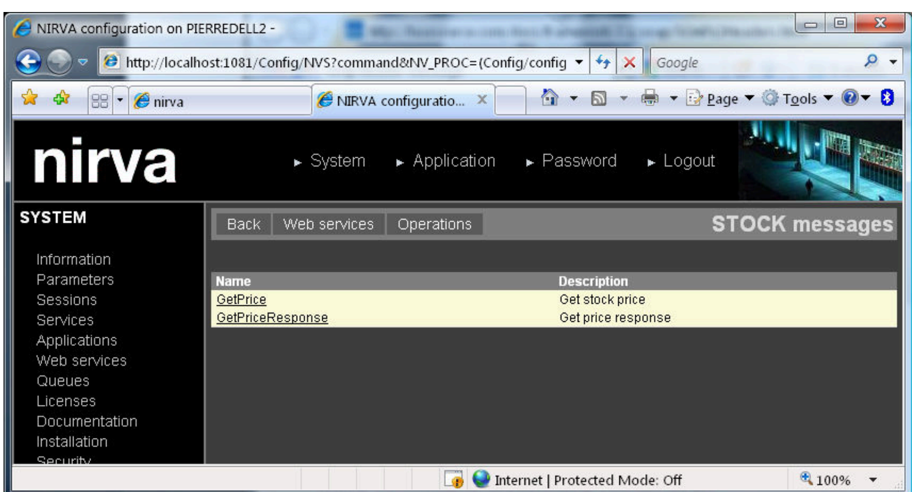
The icon near the operation name allows displaying detail information about the operation:



This operation information is very useful for launching the web service operation. It gives the following information:

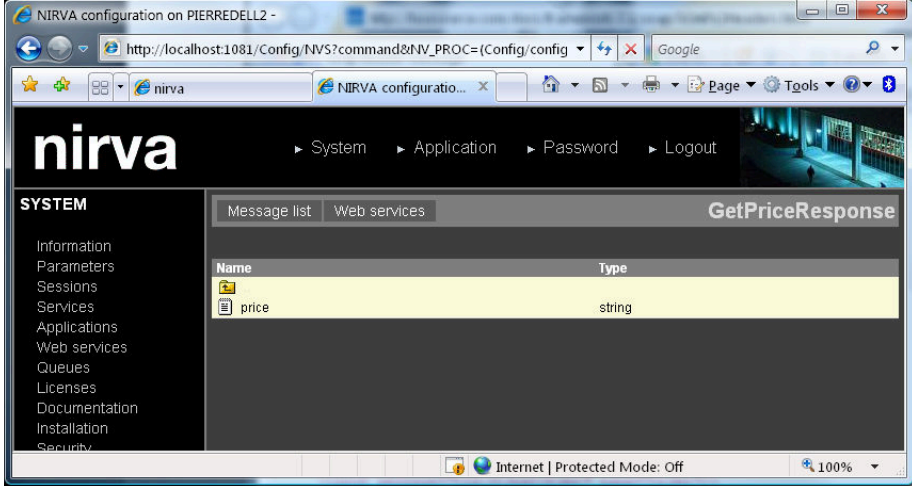
- Web service name
- Operation name
- Operation description
- Url of the web service
- Url of the WSDL definition of the web service
- Value of the SoapAction HTTP header to use in the HTTP SOAP request
- Example of an SOAP input message

Messages:



For each message, the table displays the message name and description.

In order to view the structure of a message, just click on its name. Then, NIRVA displays the hierarchical structure of the message. A message is composed of NIRVA objects and subcontainers containing themselves other objects or subcontainers:



One can walk into the message structure by clicking the containers or parent links.

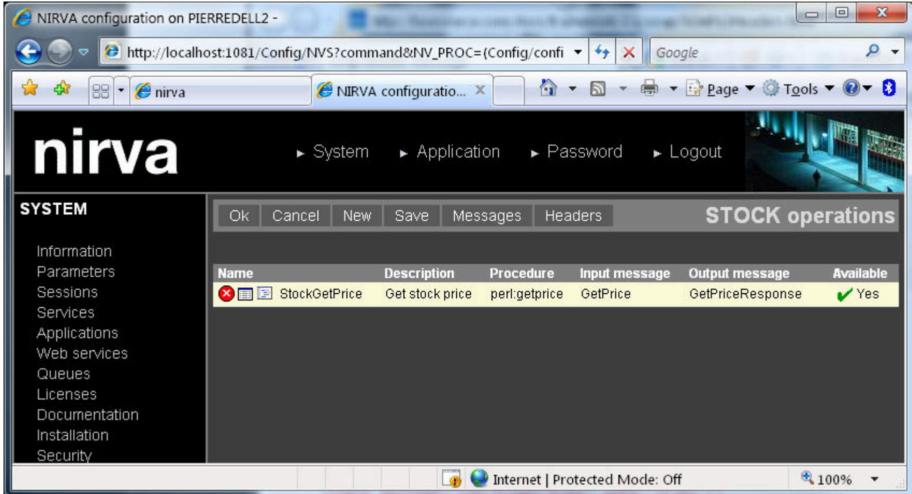
Edit web service content

Editing a web service means defining a set of messages and operations. Each web service operation is associated to an input and output message.

The edition of a web service is possible only when the web service is stopped. Only one user at a time can enter the edit mode for a given web service. When a web service is in edit mode, NIRVA works with a copy of a web service allowing the user to eventually come back to the original web service definition.

For editing the web service content (operations and messages), just click on the icon near its name from the web service list screen. This enters the edit mode and displays the list of defined operations for the selected web service. Then it’s possible to switch between operation and message views by clicking on the corresponding button (Messages or Operations buttons).

Operations:



For each operation, the table displays the operation name, the operation description, the name of the NIRVA procedure that processes the operation, the name of associated input and output messages and the availability of the operation. If one of the input or output message is not defined a sign is displayed near it.

The Ok button saves the changes to the web service, leaves the edit mode and comes back to the web service list.

The Cancel button leaves the edit mode and comes back to the web service list without applying changes to the web service.

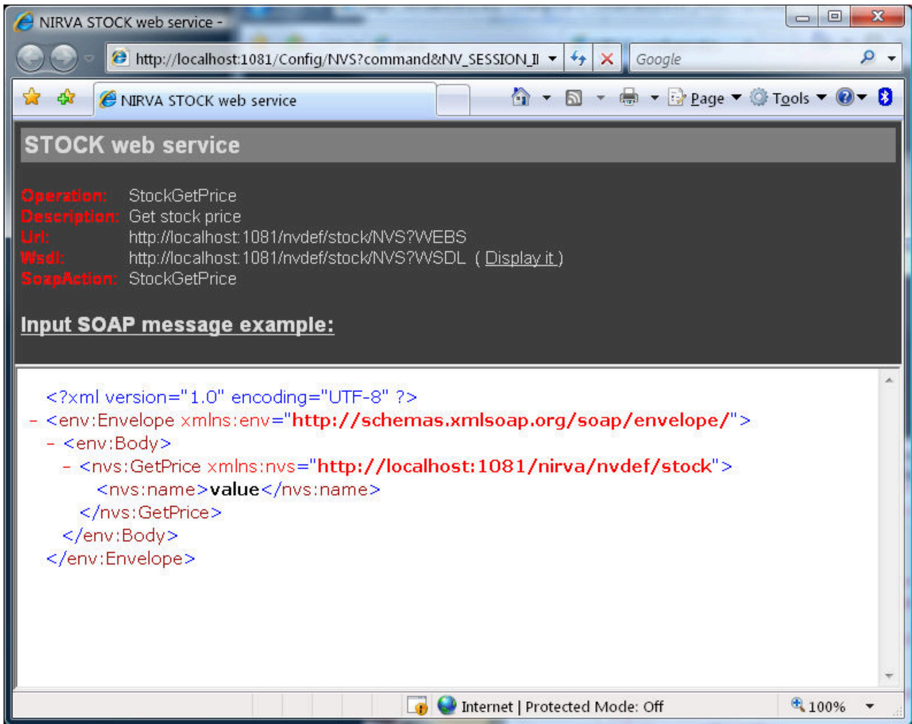
The Save button apply changes to the web service but without leaving the edit mode.

Removing an operation

In order to remove an operation, just click on the icon near its name.

Display operation information

The icon near the operation name allows displaying detail information about the operation:



This operation information is very useful for launching the web service operation. It gives the following information:

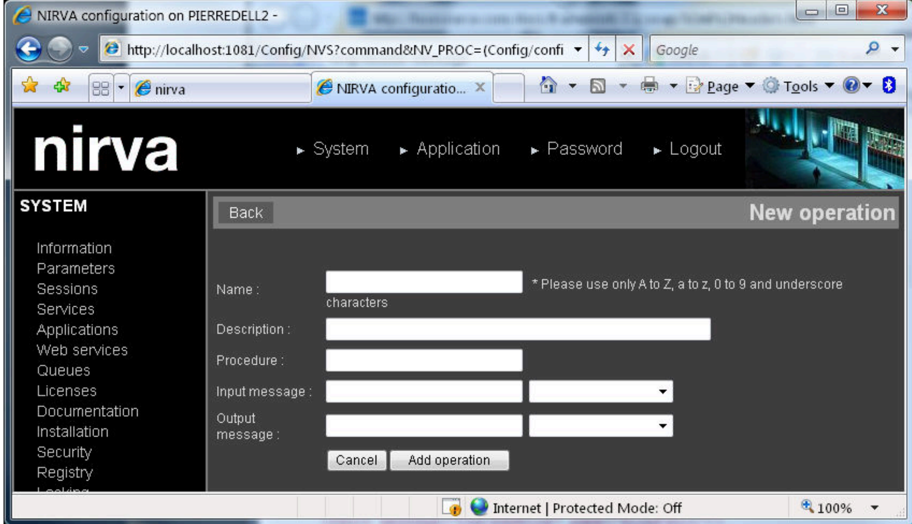
- Web service name
- Operation name
- Operation description
- Url of the web service
- Url of the WSDL definition of the web service
- Value of the SoapAction HTTP header to use in the HTTP SOAP request
- Example of an SOAP input message

Enabling or disabling an operation

In order to enable or disable a web service operation, just click on the ✔ or ✘ icons in the corresponding available le column.

Creating a new operation

In order to create a new operation, just click the “New” button. This displays the following screen:



The only required information is the operation name. The operation name can be only alphanumeric characters and the underscore character. The operation name is case insensitive.

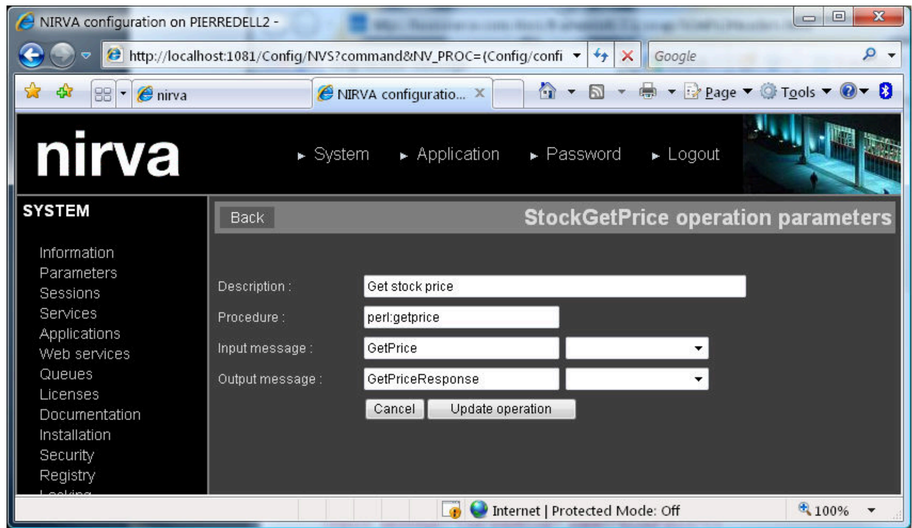
The Procedure information is not mandatory but if not provided, the web service will do nothing. The web service procedures must reside in the procedure directory of the web service.

The input and output messages are also not mandatory at this step but an operation without input or output message has really no meaning.

A same message cannot be used both as input and output even in different opérations. If this is the case, the WSDL generation will fail.

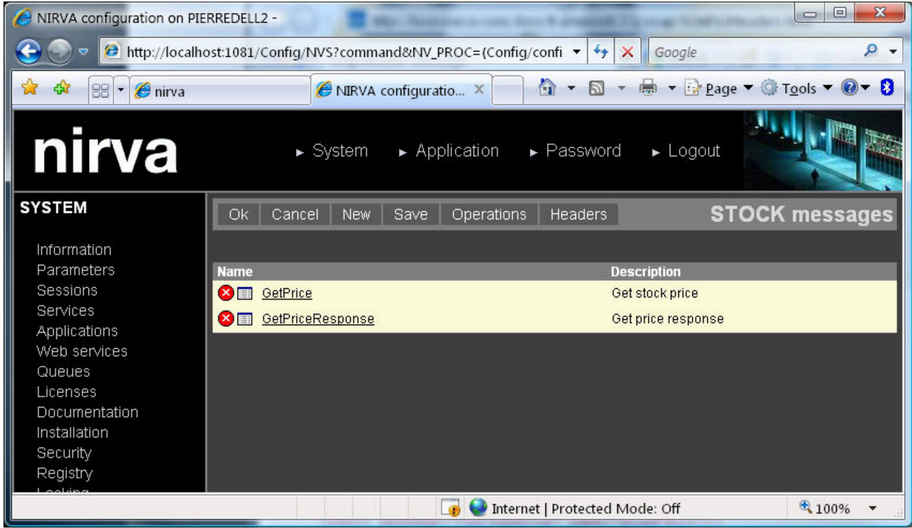
Changing operation parameters

In order to change web service operation parameters, just click on the icon near its name. This displays the following screen:



All the operation parameters can be changed except the operation name.

Messages:



For each message, the table displays the message name and description.

The Ok button saves the changes to the web service, leaves the edit mode and comes back to the web service list.

The Cancel button leaves the edit mode and comes back to the web service list without applying changes to the web service.

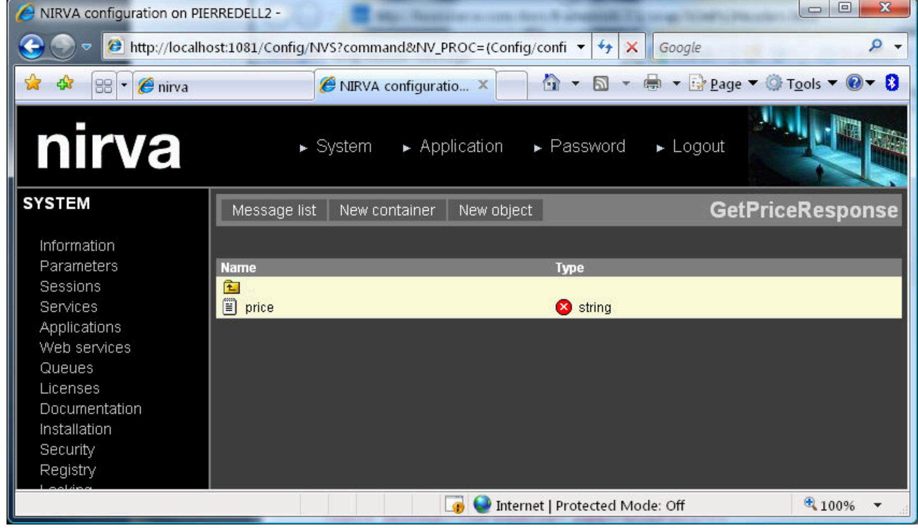
The Save button apply changes to the web service but without leaving the edit mode.

Removing a message

In order to remove a message, just click on the icon near its name.

Editing message content

In order to edit the structure of a message, just click on its name. Then, NIRVA displays the hierarchical structure of the message. A message is composed of NIRVA objects and subcontainers containing themselves other objects or subcontainers:



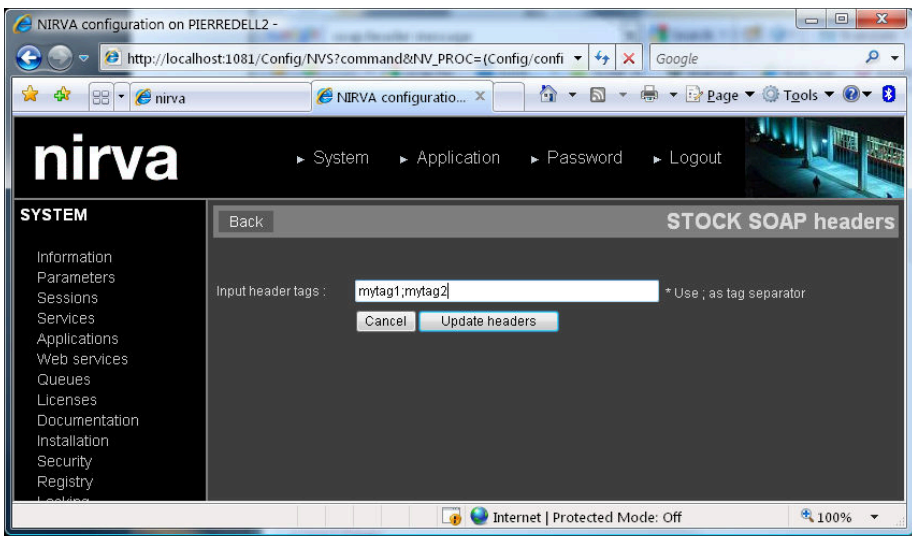
One can walk into the message structure by clicking the containers or parent links. One can also create and remove objects and containers.

In order to remove a container or an object, click on the icon near its type.

For adding a new container, click on the “New container” button.

For adding a new object, click on the “New object” button. Available object types are standard NIRVA object types.

Headers:



This screen allows defining the name of the tags of the optional soap header for input messages. The header is a succession of XML tags (type string) embedded in a nvcommand element.

Example of Soap header:

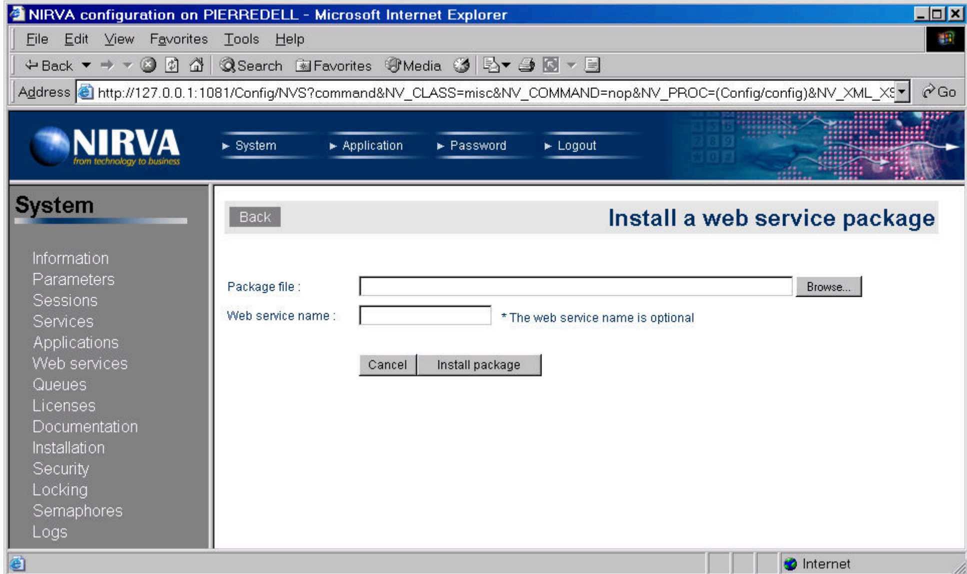
```
<soap:Header>
<nvheader>
  <nvcommand>
    <mytag1>value tag1</mytag1>
    <mytag2>value tag2</mytag2>
  </nvcommand>
</nvheader>
</soap:Header>
```

View web service WSDL data

The web service WSDL data defines the entire web service content for external application to work with it. It respects the WSDL standard and is URL accessible. In order to view the WSDL data, click on the icon near the web service name from the web service list screen:

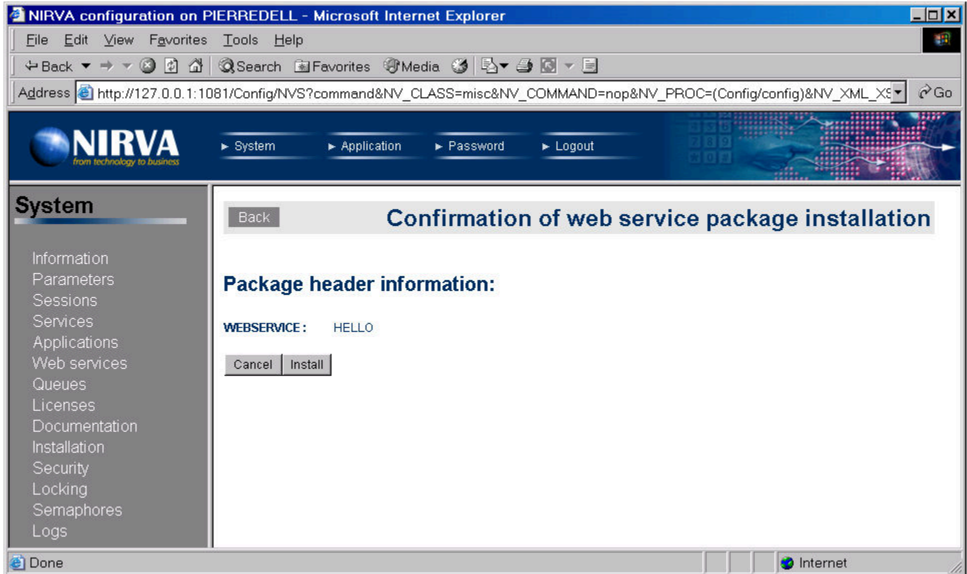
Installing a web service package

A Nirva web service is delivered as a NIRVA package file. The package file can contain the whole web service or simply some patches. In order to install a web service package, just click on the "Install" menu at the top of the "Web service list" screen. This displays the following screen:



The web service name is optional. If it's not provided, NIRVA tries to get it from the package file (this is generally the case but the web service name can be overridden allowing to install a web service under a different name).

After pressing the "Install package" button, NIRVA displays a confirmation screen with some information extracted from the package file:

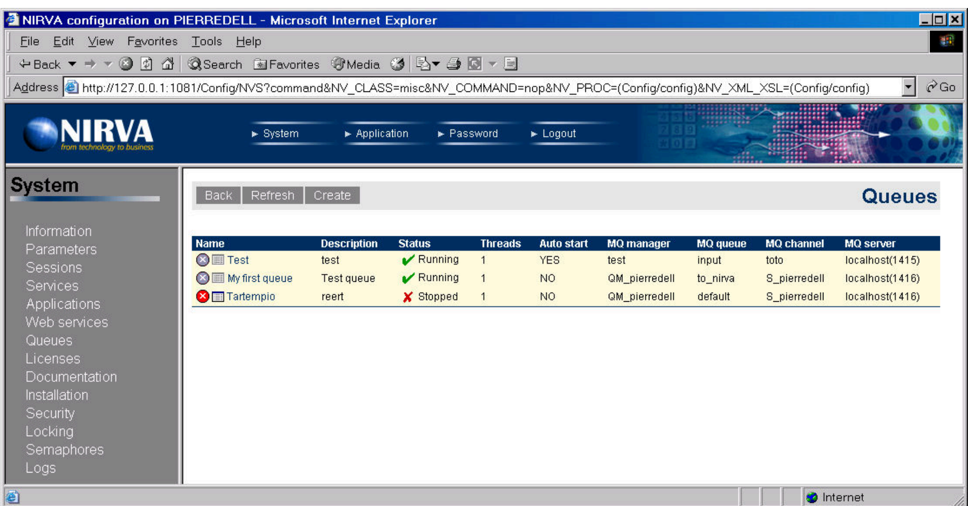


If the web service to install is currently running, the command fails. A web service package can only be installed if the web service is stopped (or not yet installed).

Queues

This menu allows defining the parameters of the IBM WebSphere MQ connector. The use of this connector requires a dedicated license. Without the license or without an IBM WebSphere MQ client installed on the NIRVA machine, it's possible to define the parameters of the connector but the queues cannot be run.

The following screen is displayed when choosing the "Queues" menu:



For each queue, the list displays the following information:

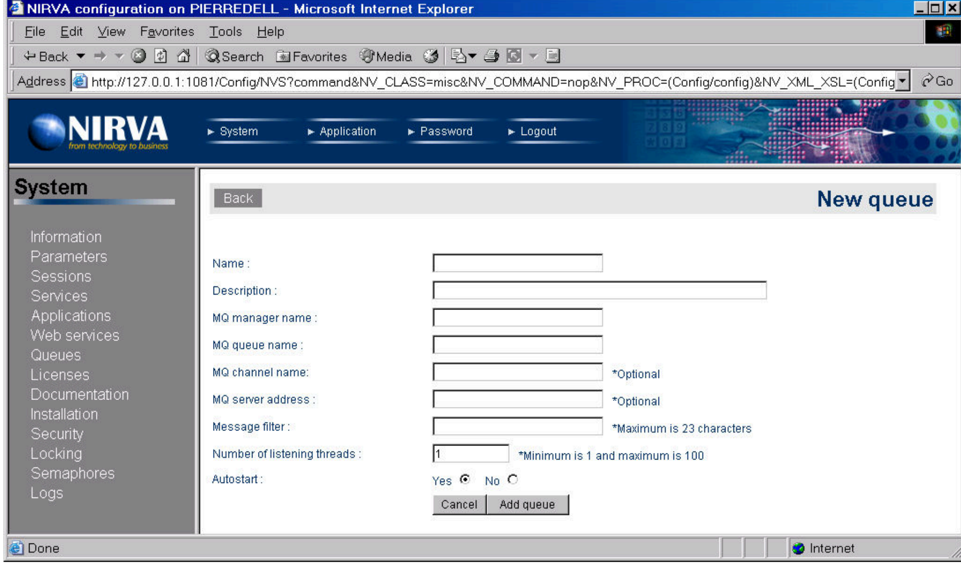
- Name is the queue name that uniquely identifies the queue at NIRVA level.
- Description is the queue description.
- Status tells if the queue is running or not.
- Threads are the number of NIRVA threads that listen on the queue.
- Auto start tells if the queue is automatically started when starting NIRVA or not.
- MQ manager is the WebSphere MQ manager name.
- MQ queue is the WebSphere MQ queue name to listen.
- MQ channel is the WebSphere MQ channel name (defines the communication link to the WebSphere MQ server).
- MQ server is the address of the WebSphere MQ server.

From this window, one can do the following functions:

- Create a new queue
- Start and stop a queue
- Edit queue parameters
- Remove a queue

Creating a new queue

In order to create a new queue just click on the “Create” button at the top of the “Queues” screen. This displays the following screen:



Name is the queue name that uniquely identifies the queue at NIRVA level. It can contain spaces or special characters.

Description is the queue description. It’s not mandatory.

MQ manager name is the name of the queue manager as defined on the IBM WebSphere MQ server. This parameter is mandatory.

MQ queue name is the name of the queue to listen. This must correspond to an existing queue defined on the IBM WebSphere MQ server. This parameter is mandatory.

MQ channel name is the channel name to use for communication to the IBM WebSphere MQ server. This parameter is optional. If not provided, NIRVA will get the content of the MQSERVER environment variable in order to now the address and channel name of the MQ server. Please see the IBM WebSphere MQ client documentation for further information about the MQSERVER variable and channel definition.

MQ server address is the TCP/IP address and port of the IBM WebSphere MQ server channel. This parameter is optional. If the channel name is not given, this parameter has no meaning. If the channel name is given and the server address is blank, NIRVA uses “localhost” as server address and 1414 as port number. The default port number is always 1414. One can change it by giving the port in brackets at the end of the server address. For example: 123.45.67.89(1567) will connect to the server 123.45.67.89 at port 1567. The server address can be the server name.

Message filter is optional. The filter is a string allowing NIRVA to listen only for messages having a correlation ID equal to this string. See the IBM WebSphere MQ documentation for further information about correlation ID. If the message filter is blank, NIRVA gets all messages from the given queue.

Number of listening threads is the number of threads that NIRVA will run for listening the queue. This parameter allows controlling the machine resources dedicated to a given queue. The minimum value is 1 and the maximum value is 100.

Auto start is tells if NIRVA must start listening the queue at start time or not.

The user starting the nirva server must have enough rights to works with the queues defined on nirva.

Starting and stopping a queue

In order to start or stop a web service, just click on the ✔ or ✘ icons at the left of the queue status.

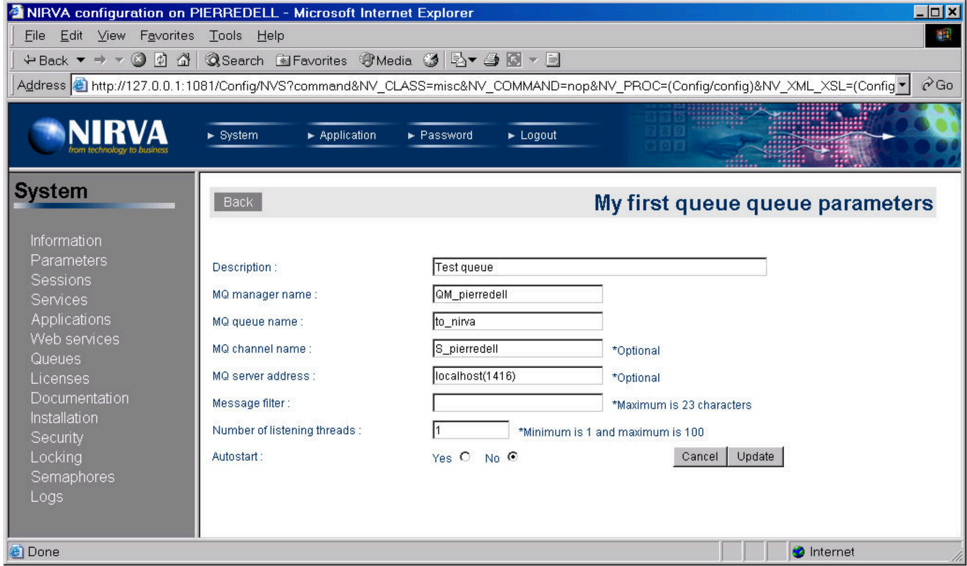
When stopping a queue, NIRVA displays a confirmation message first.

Stopping a queue means stop listening the queue. The queue itself can be stopped only at WebSphere MQ server level.

Editing queue parameters

In order to edit the queue parameters, just click on the icon near its name from the queue list. The queue must be stopped for changing its parameters.

This displays the following screen:



Description is the queue description. It’s not mandatory.

MQ manager name is the name of the queue manager as defined on the IBM WebSphere MQ server. This parameter is mandatory.

MQ queue name is the name of the queue to listen. This must correspond to an existing queue defined on the IBM WebSphere MQ server. This parameter is mandatory.

MQ channel name is the channel name to use for communication to the IBM WebSphere MQ server. This parameter is optional. If not provided, NIRVA will get the content of the MQSERVER environment variable in order to now the address and channel name of the MQ server. Please see the IBM WebSphere MQ client documentation for further information about the MQSERVER variable and channel definition.

MQ server address is the TCP/IP address and port of the IBM WebSphere MQ server channel. This parameter is optional. If the channel name is not given, this parameter has no meaning. If the channel name is given and the server address is blank, NIRVA uses "localhost" as server address and 1414 as port number. The default port number is always 1414. One can change it by giving the port in brackets at the end of the server address. For example: 123.45.67.89(1567) will connect to the server 123.45.67.89 at port 1567. The server address can be the server name.

Message filter is optional. The filter is a string allowing NIRVA to listen only for messages having a correlation ID equal to this string. See the IBM WebSphere MQ documentation for further information about correlation ID. If the message filter is blank, NIRVA gets all messages from the given queue.

Number of listening threads is the number of threads that NIRVA will run for listening the queue. This parameter allows controlling the machine resources dedicated to a given queue. The minimum value is 1 and the maximum value is 100.

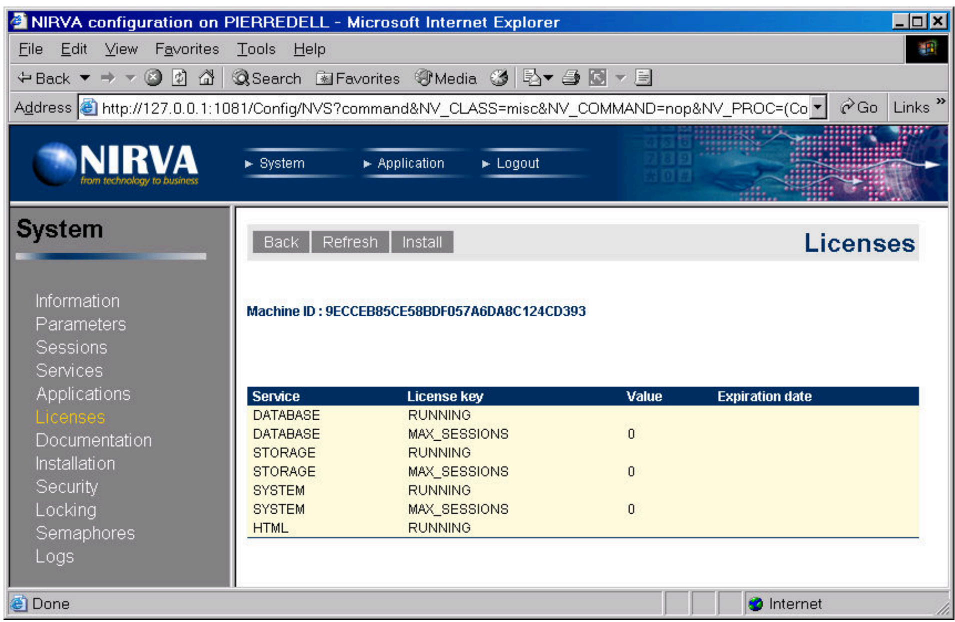
Auto start is tells if NIRVA must start listening the queue at start time or not.

Removing a queue

In order to edit the queue parameters, just click on the ⊗ icon near its name from the queue list. The queue must be stopped for removing it.

Licenses

The "Licenses" menu displays the NIRVA unique machine identifier and the installed license channels.



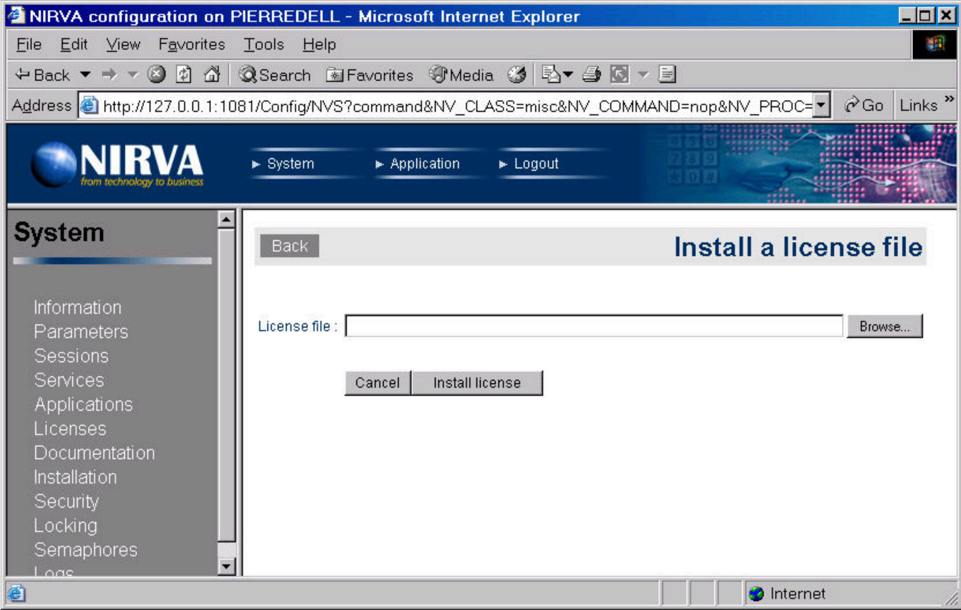
Machine ID is the unique machine identifier that the customer must report to the NIRVA distributor or external service provider in order to activate licenses.

The list of license channels gives the following information:

- "Service" is the service name. It will be "SYSTEM" for a NIRVA system license key.

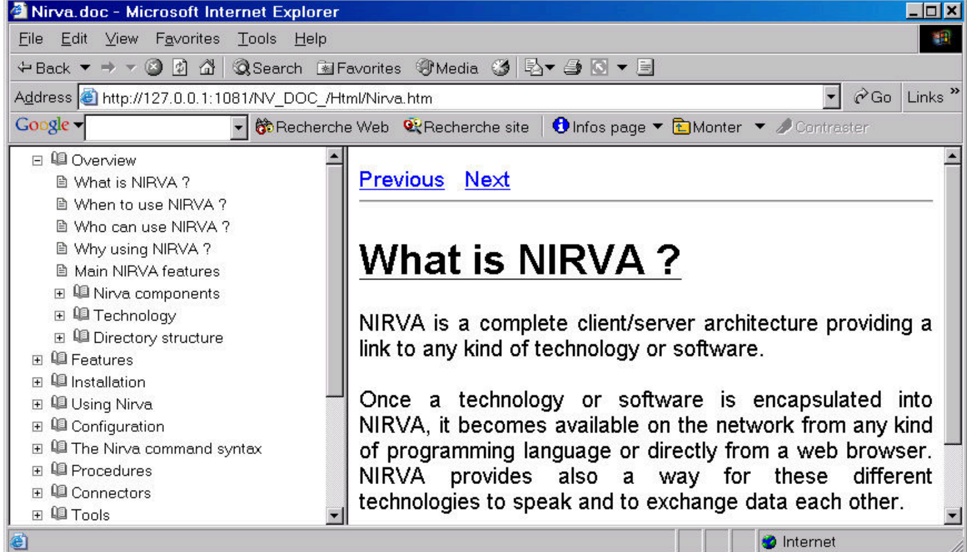
- “License key” is the license key name.
- “Value” is the optional license key value.
- “Expiration” is the optional license expiration date. It has the format YYYYMMDD and is blank if there is no date.

The Install link allows installing a license file returned by a NIRVA distributor or an external service provider:



Documentation

The “Documentation” menu just displays the NIRVA documentation in its HTML form. The documentation is displayed in another browser.



Installation

This menu allows installing any kind of NIRVA package file. It can be an application, service or system package.

NIRVA automatically detects the kind of package following its header information.

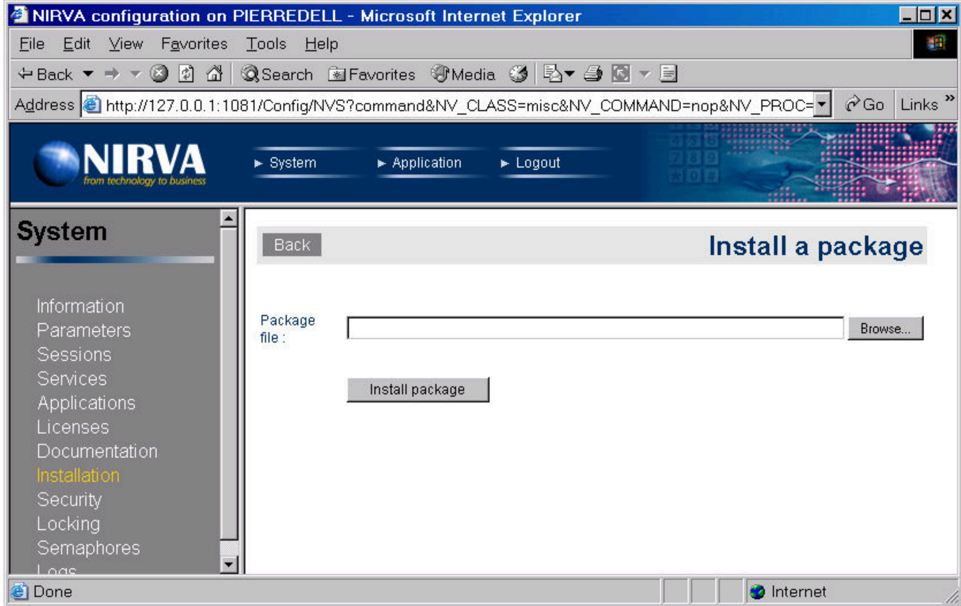
The package file is a binary file that itself contains other files.

This command can only install files in the NIRVA directory (and subdirectories).

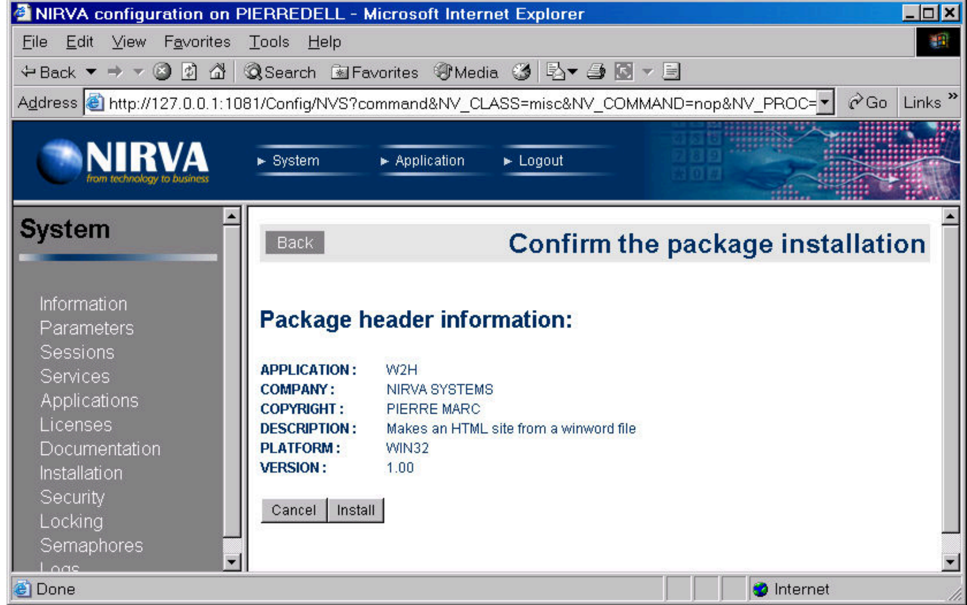
The package file is a binary file that itself contains other files.

If the package file has a header entry "*SERVICE=SrvName*", the command installs the package as a service package for the *SrvName* service.

If the package file has a header entry "*APPLICATION=AppName*", the command installs the package as an application package for the *AppName* application.



After pressing the "Install package" button, NIRVA displays a confirmation screen with some information extracted from the package file:

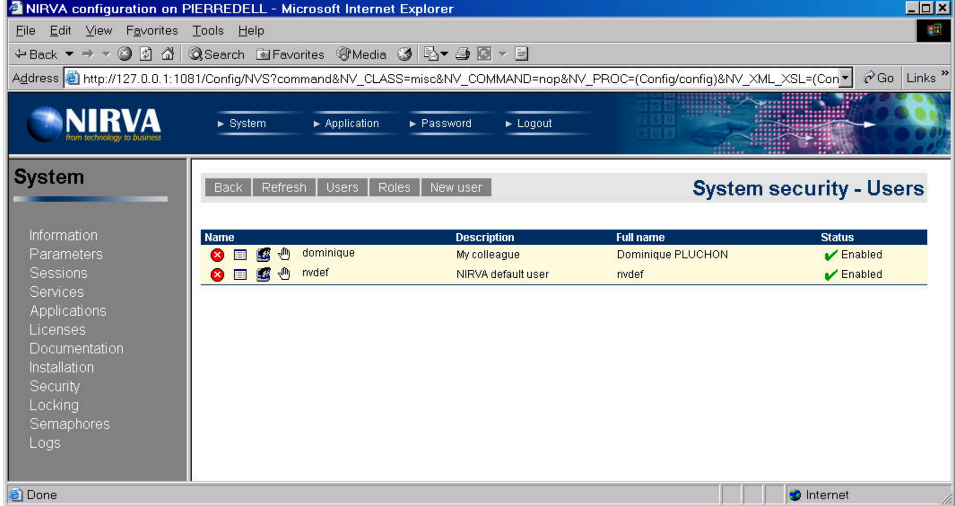


If the package is an application package and the application to install is currently running, the command fails. In the same way, if the package is a service package and the service to install is currently running, the command fails.

Security

With this menu, the user can administrate system security. The NIRVA security layer works at application layer but an application can choose to use the system security or the security of another application.

The security is described in the “Security model” chapter of this documentation. It’s a RBAC model (role based access control) that defines permissions, roles and users. The roles are hierarchical so a role can inherent the permissions of other roles.

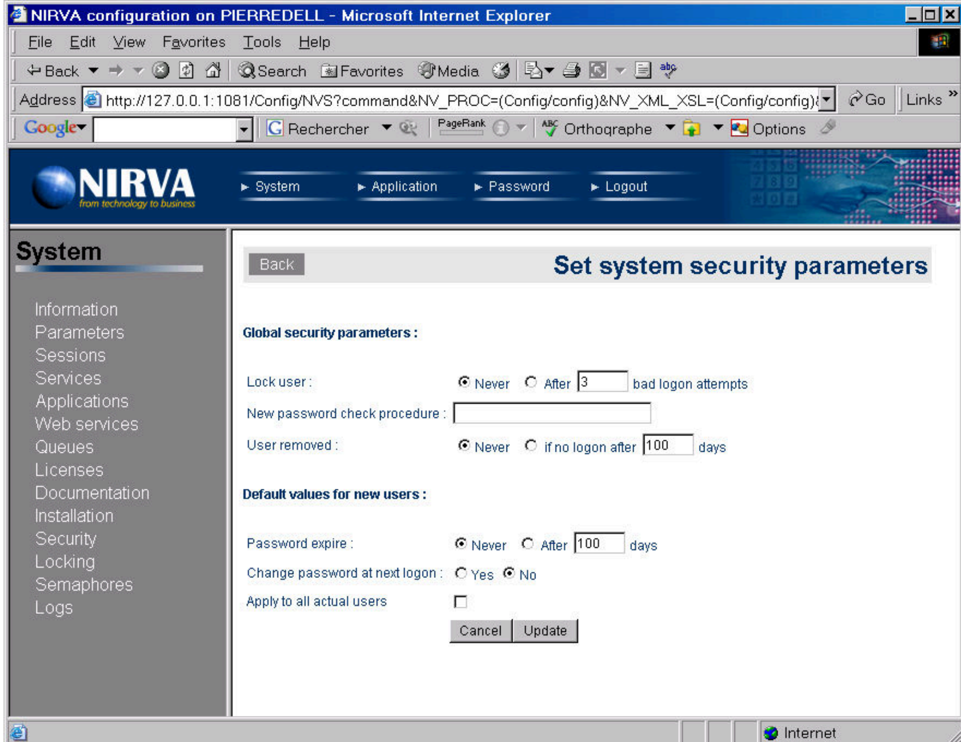


This screen displays the current user list. There is always at least one user named “nvdef” that is the default NIRVA user. The “nvdef” user cannot be removed or disabled.

The administrator user (“nvadmin”) is not displayed in this list.

Setting global parameters

Pressing the “Parameters” button allows changing some global security parameters:



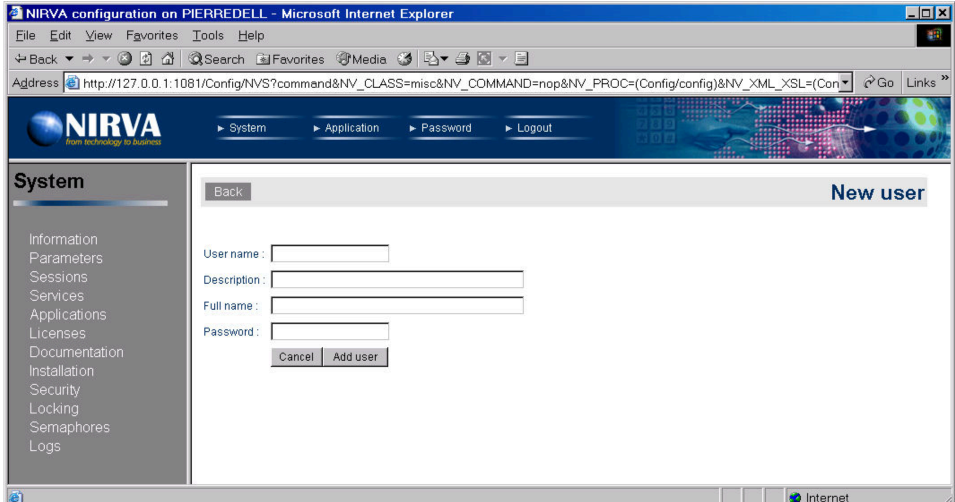
From this screen one can change the following parameters:

- Number of bad logon attempts before the user to be locked.
- The name of a procedure for checking the new password syntax. This procedure, if it exists, will be launched by Nirva when the user is changing its password. It will receive 3 parameters OLD_PASSWORD containing the old password, NEW_PASSWORD containing the new password and USER containing the user name. If the procedure produces an error (SetError function from Perl, Dotnet or Java procedure), Nirva will return a bad password syntax error to the user.
- The number of days after which unused users are automatically removed from the security.
- The default expiration time in days for new users.
- The flag for new user to change their password at first logon.

The last 2 parameters can eventually be applied to all current users.

Adding a new user

For adding a new user, click on the “New user” button:

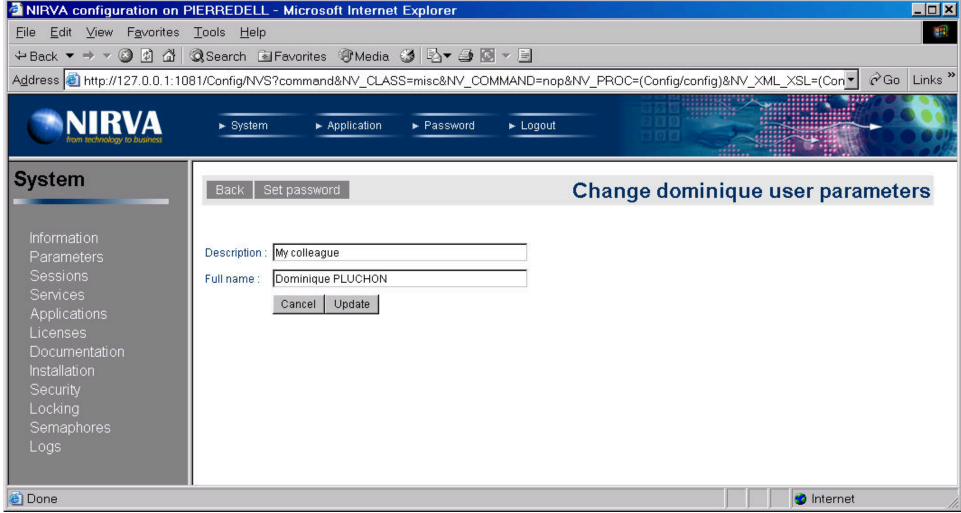


The user name is mandatory. It should not contain any space or special character and is case independent.

The other user parameters are optional.

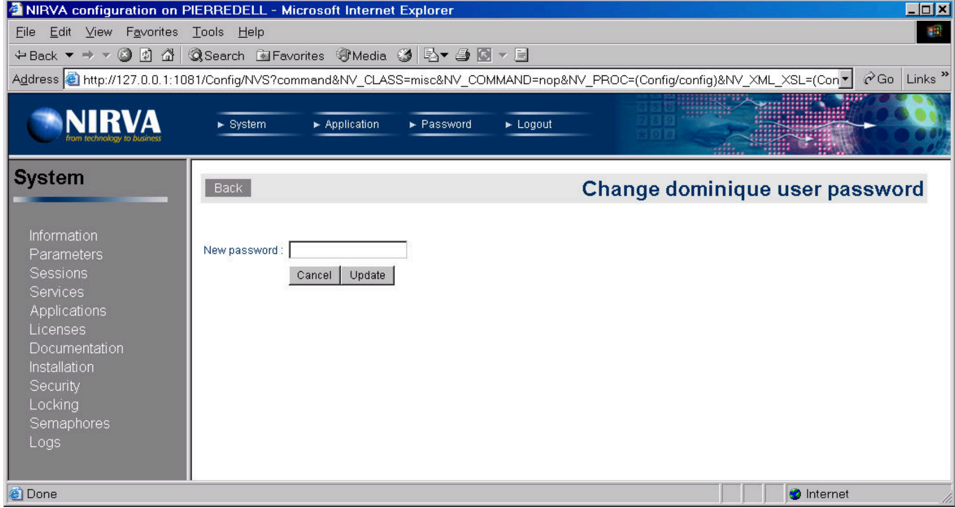
Changing user information

For changing some of the user information, click on the icon near its name:



The description, the user full name and expiration parameters can be changed.

If the current user of the configuration tool has the corresponding security permission, he can also change the user password by pressing the “Set password” button:



Removing a user

In order to remove a user, just click on the icon near its name. There is a confirmation message.

The default NIRVA user (nvdef) cannot be removed.

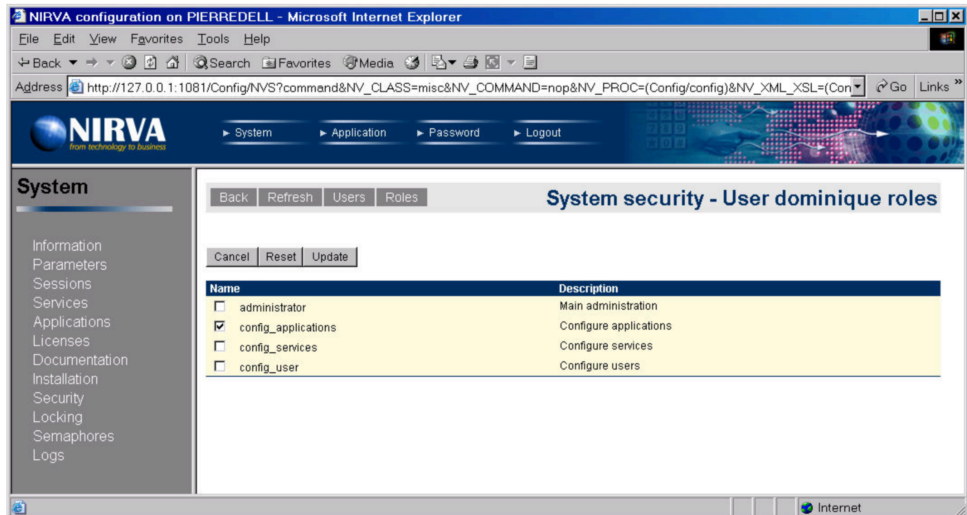
Enabling or disabling a user

A user can be temporary disabled without removing it from the user list. For that, just click on the icon on the status column.

If the user is disabled, you can enable it by clicking click on the icon on the status column.

Setting user roles

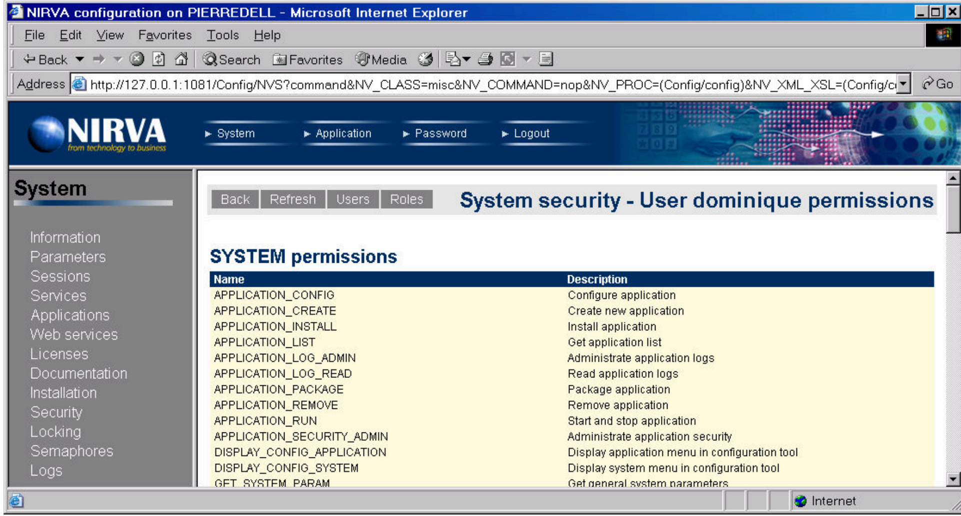
In order to set the user roles, just click on the icon near its name. This displays a list of possible roles with a check box for selecting them. If the user already has a role, the corresponding check box is checked when displaying the list:



For changing the user roles, just select the required roles and press the “Update” button.

Viewing user permissions

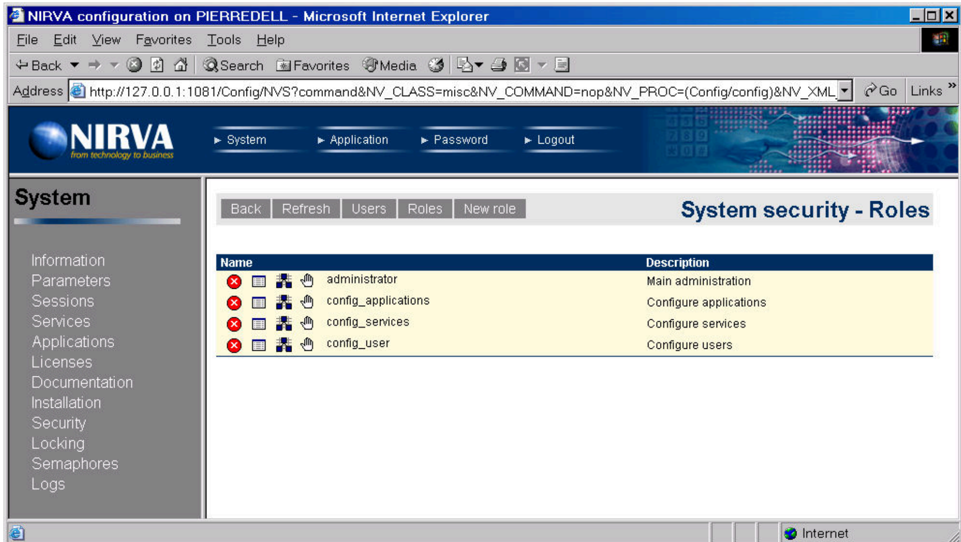
In order to view the detailed user permissions, just click on the icon near its name. This displays the list of the user permissions:



The permissions cannot be changed from this screen.

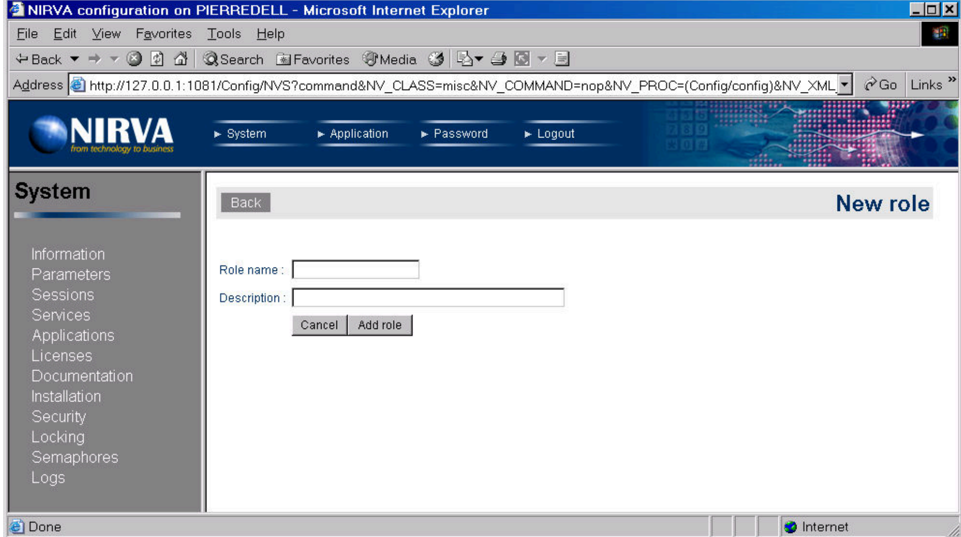
Displaying roles

The role list is available by clicking the “Roles” button. This displays the list of currently defined roles.



Adding a new role

For adding a new role, click on the “New role” button from the role list:

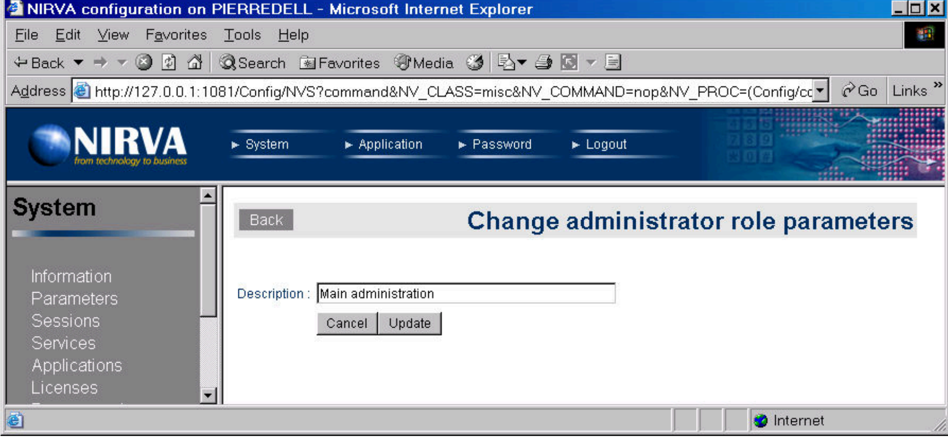


The role name is mandatory. It should not contain any space or special character and is case independent.

The other role parameters are optional.

Changing role information

For changing some of the role information, click on the icon near its name:



Only the description can be changed.

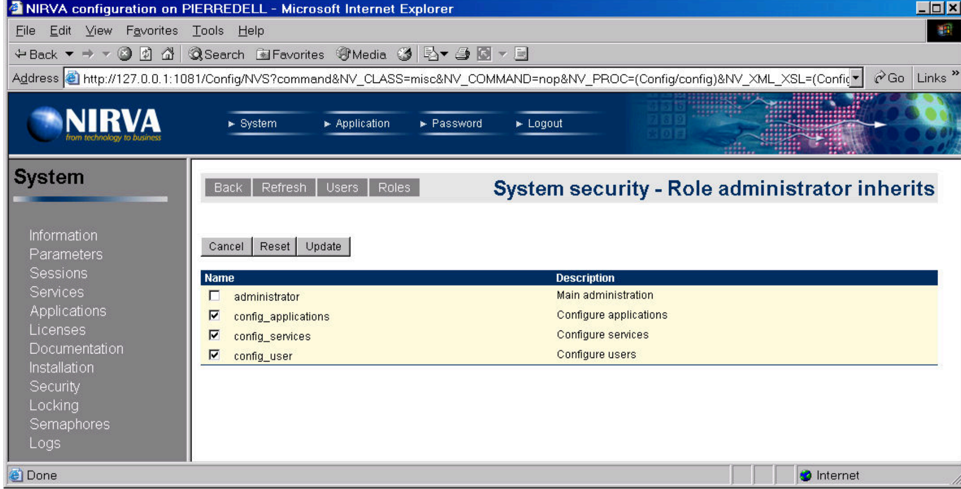
Removing a role

In order to remove a role, just click on the icon near its name (from the role list). There is a confirmation message.

Setting role inherits

A role can inherit the permissions of other roles.

In order to set the role inherits, just click on the icon near its name. This displays a list of possible roles with a check box for selecting them. If the role already inherits a role, the corresponding check box is checked when displaying the list:

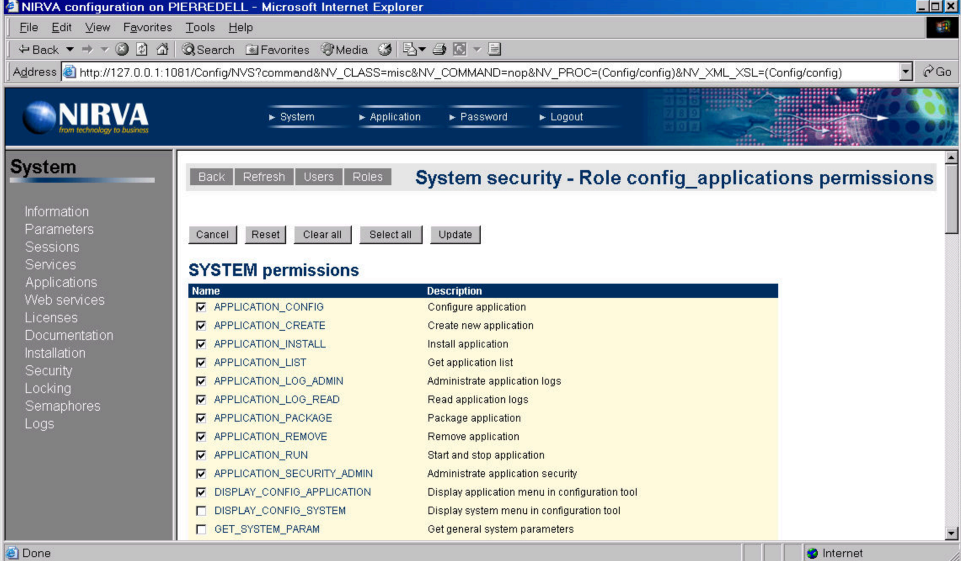


For changing the role inherits, just select the required roles and press the “Update” button.

NIRVA doesn’t display any message if there is a recursive problem in the hierarchy of roles but automatically take cares of this problem by stopping hierarchy when a re-entrant role is detected.

Setting role permissions

In order to set the role permissions, just click on the icon near its name. This displays a list of possible permissions with a check box for selecting them. If the role already has permission, the corresponding check box is checked when displaying the list:

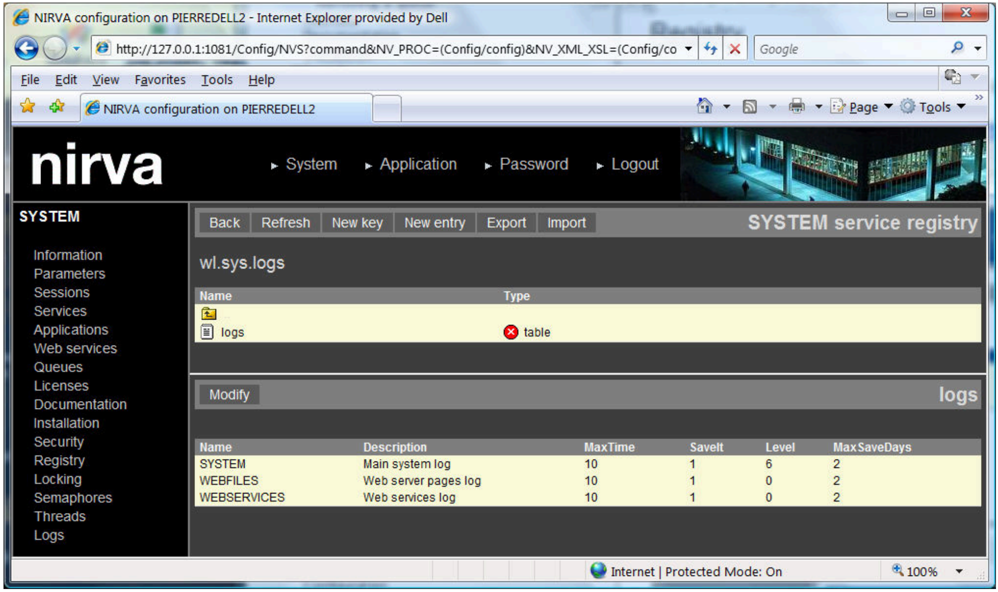


For changing the role permissions, just select the required permissions and press the “Update” button.

The permission list contains all the system, service and web service defined permissions.

Registry

The “Registry” option allows viewing and editing the system level registry:

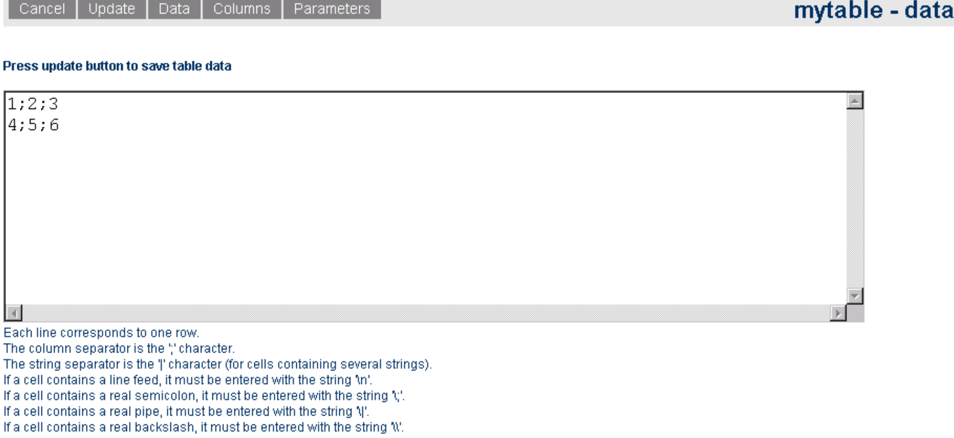


There are two parts in the registry editor.

The upper part displays the current registry key. One can add, modify or remove entries. The registry key or entry cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character.

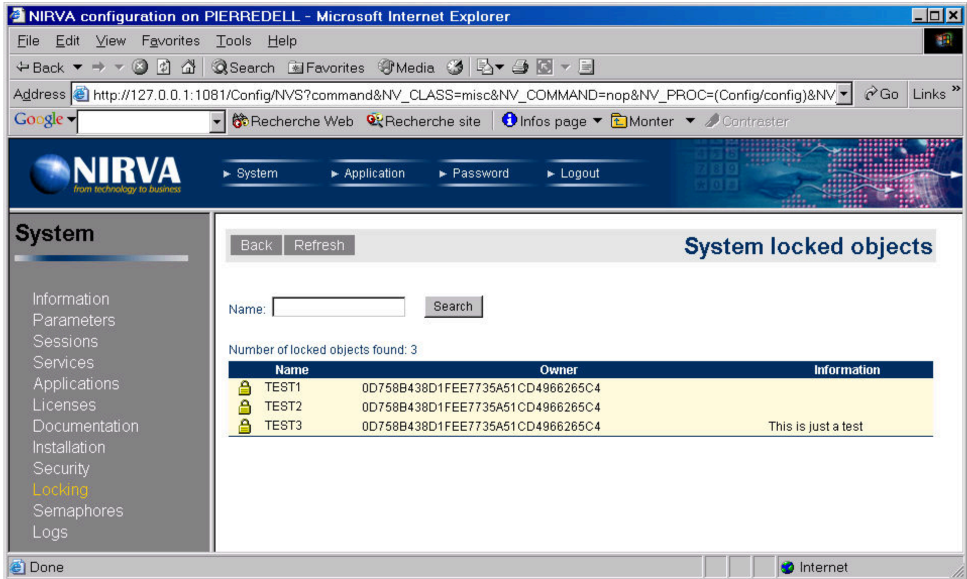
The lower part displays detail information about a given entry and allows editing it.

Specific entry instructions are displayed at the bottom of the screen when editing an entry. Here is an example for the table editing:



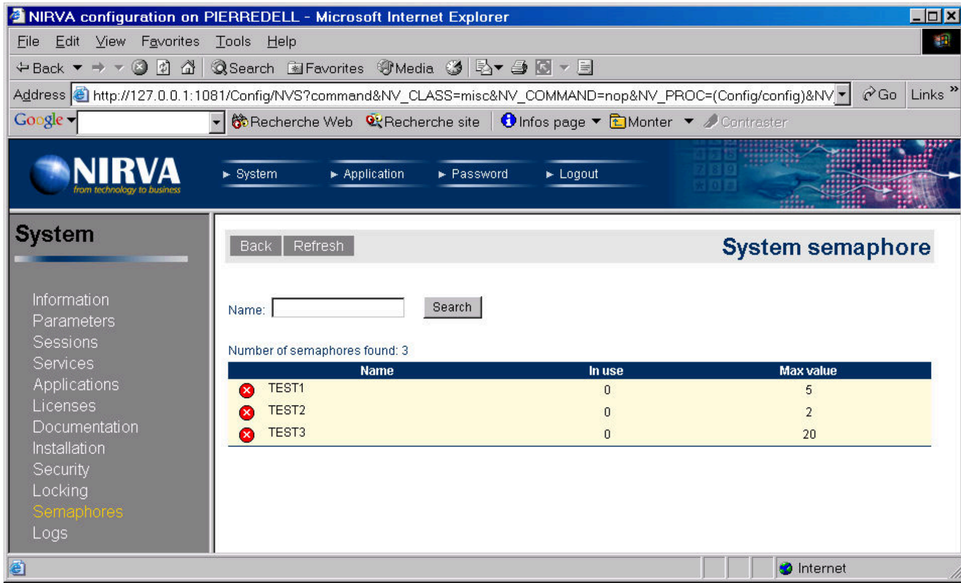
Locking

The “Locking” option displays the list of system locked objects:



Semaphores

The “Semaphores” option displays the list of system semaphores:



By default, the list shows all the semaphore objects but it’s possible to limit it to some semaphore object names.

If the name finishes by the ‘*’ wildcard character, NIRVA searches for all semaphore names starting with the given characters.

For each item of the list, NIRVA provides the following information:

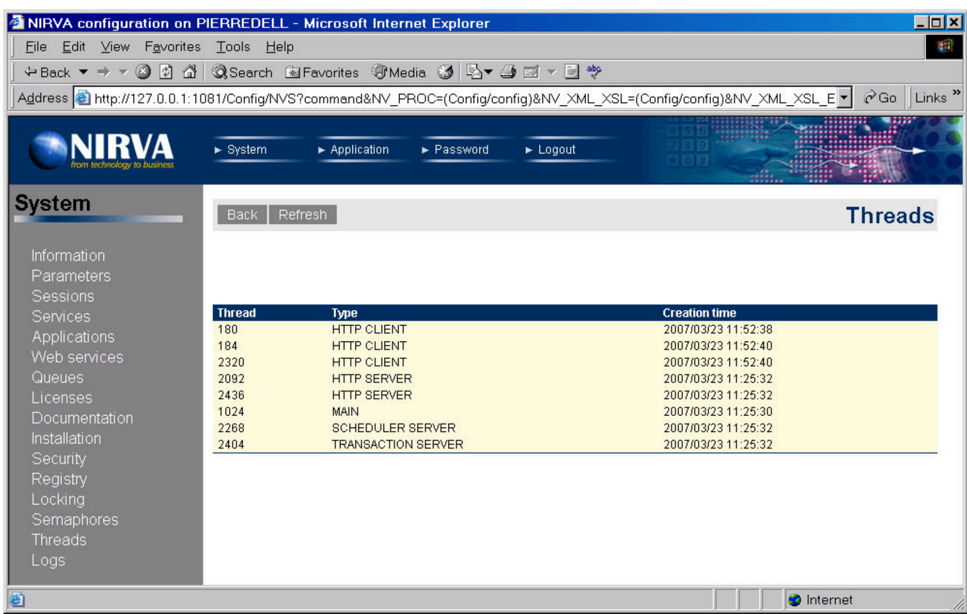
- “Name” is the semaphore name.
- “In use” is the number of threads having requested an access to the semaphore.
- “Max value” is the maximum number of simultaneous accesses to the semaphore.

A semaphore can be removed by clicking on the icon near its name. At this time, NIRVA provides a confirmation message. If the semaphore is in use, the removing operation will fail.

In fact, NIRVA permanently checks for unused semaphores and automatically removes them.

Threads

The “Threads” option displays the list of Nirva threads:



Thread	Type	Creation time
180	HTTP CLIENT	2007/03/23 11:52:38
184	HTTP CLIENT	2007/03/23 11:52:40
2320	HTTP CLIENT	2007/03/23 11:52:40
2092	HTTP SERVER	2007/03/23 11:25:32
2436	HTTP SERVER	2007/03/23 11:25:32
1024	MAIN	2007/03/23 11:25:30
2268	SCHEDULER SERVER	2007/03/23 11:25:32
2404	TRANSACTION SERVER	2007/03/23 11:25:32

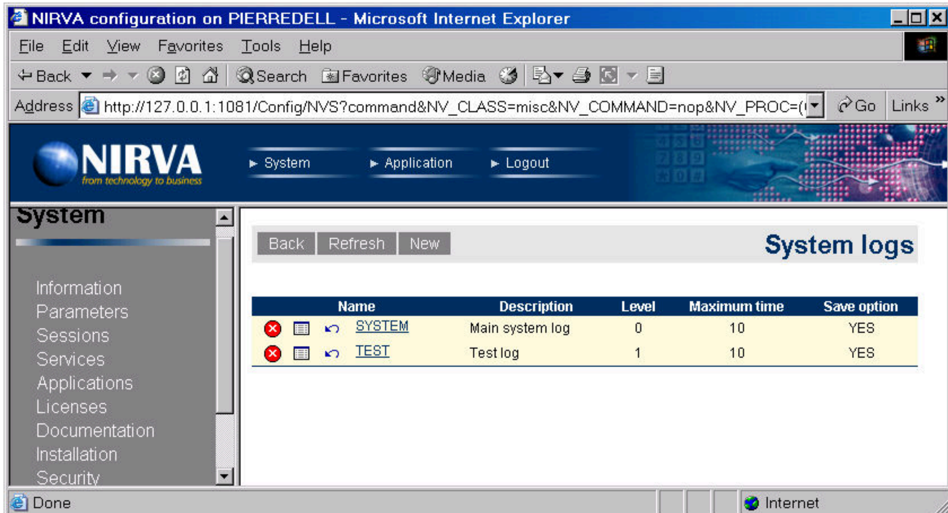
Only the threads directly created by Nirva are shown. All threads created by an external service or procedure, by java, dotnet or perl are not displayed.

The thread type can be one of the following:

HTTP_SERVER	HTTP or HTTPS server. This is the thread that listens for http clients to connect.
HTTP_CLIENT_BUILDER	Thread used to create HTTP client threads. This thread is created by the HTTP server thread to fast accept several clients at the same time. It’s removed after the client threads have been created.
HTTP_CLIENT	Client HTTP thread use to communicate from client via HTTP.
MAIN	Main Nirva thread.
SCHEDULER_SERVER	Thread that checks for scheduled tasks to start.
TASK	Thread that executes an instance of a scheduled task.
TRANSACTION_SERVER	Thread that manages transactions in background.
TRANSACTION	Background transaction.
MQ	Mq series listener thread.
LISTENER	Listener thread.
SESSION	Session thread.

Logs

This menu allows to manipulate logs at system level and to view log data:



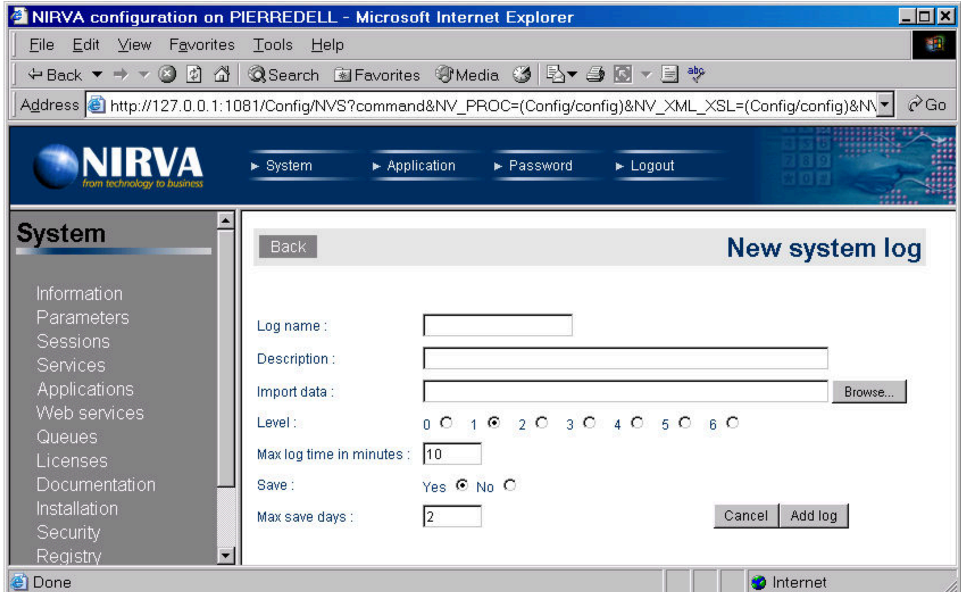
This screen shows the list of all available logs at system level.

For each item of the list, NIRVA provides the following information:

- “Name” is the log name.
- “Description” is the log description.
- “Level” is the current log level (from 0 to 6).
- “Maximum time” is the maximum period for a single log file (in minutes).
- “Save option” is set to “YES” if the logs files are saved.

Creating a new log

In order to create a new log, just click on the “New” button. This displays the following screen:



Log name is the unique name that identifies the log. The log name should not contain space or any special character.

Description is a free text that describes the log.

Import data is the optional file name that contains some data to import to the newly created log. This is useful for support people who can create and consult a log with data received from a customer. The physical import file must be the result of an exported log from the LOG SEARCH command.

Level is the initial log level. This can be a numeric value from 0 to 6. If set to "0", the LOG WRITE command will not do anything. The default value is "1".

Max log time is the maximum time in minutes that a single log file must record. The default is 10 minutes. The minimum value is 1 minute.

Save option. When this parameter is set to "YES", NIRVA saves all the log files to a dedicated directory based on a date structure. The save occurs when the log file has overflowed the maximum record time defined by the MAXTIME parameter. The default is "YES".

Max save days if the maximum number of saved days for log files.

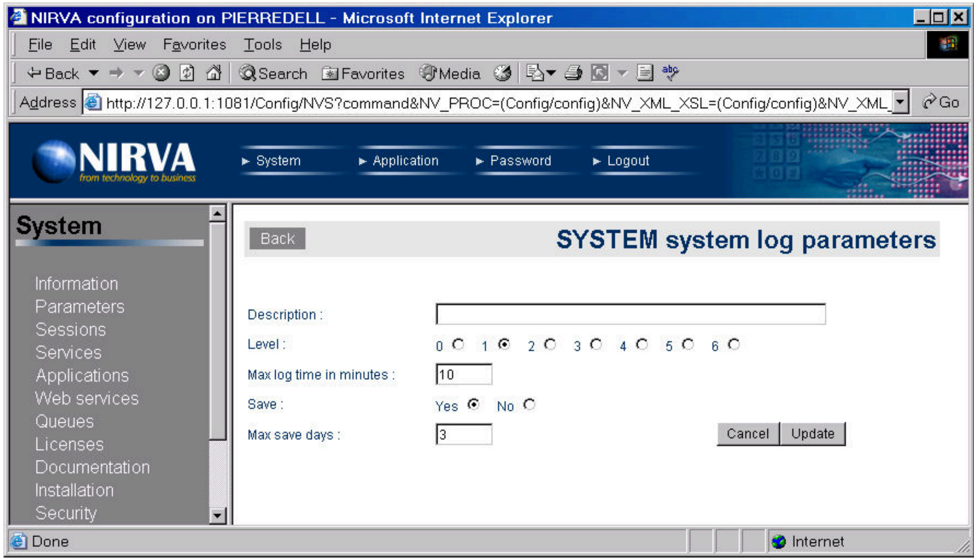
After the creation of the log, NIRVA displays back the log list

Removing a log

In order to remove a log, just click on the icon near its name. NIRVA will ask a confirmation message before removing the log. Any log data for this log will then be removed and the log itself will be removed from the log list.

Changing log parameters

In order to change some of the parameters, just click on the icon near its name. NIRVA will then display the following screen:



Description is a free text that describes the log.

Level is the current log level. This can be a numeric value from 0 to 6. If set to “0”, the LOG WRITE command will not do anything.

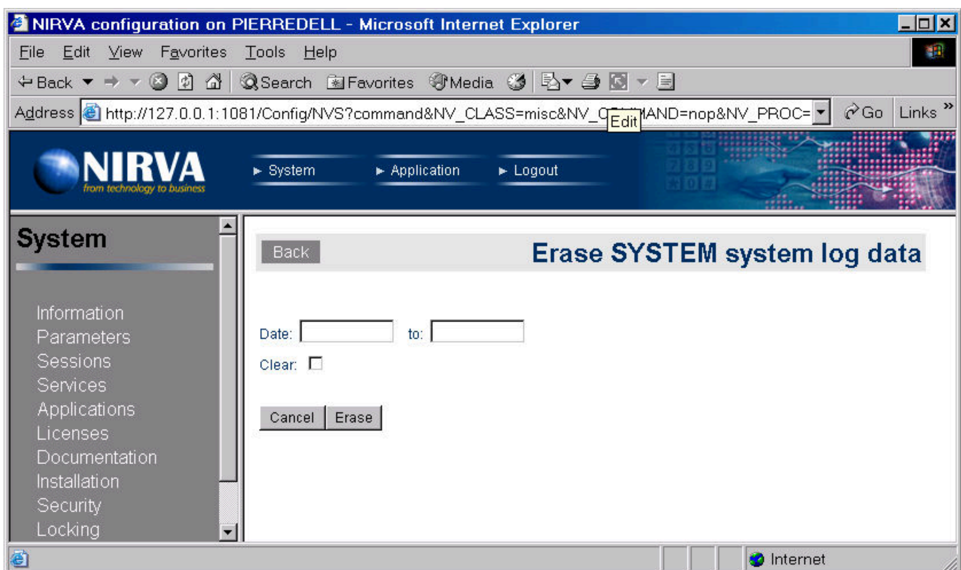
Max log time is the maximum time in minutes that a single log file must record. The minimum value is 1 minute.

Save option. When this parameter is set to “YES”, NIRVA saves all the log files to a dedicated directory based on a date structure. The save occurs when the log file has overflowed the maximum record time defined by the MAXTIME parameter.

Max save days if the maximum number of saved days for log files.

Erasing log data

In order to erase some log data, just click on the icon near its name. NIRVA will then display the following screen:



The “date” parameter allows limiting the selection to a given date or to a date range when used with the “to date” parameter. The general format of the “date” parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year.

Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.

The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.

If only DD/MM is given, NIRVA is using the current year.

If only DD is given, NIRVA is using the current month and year.

NIRVA can also accept the year on 2 digits.

If the “date” is preceded by a minus sign (‘-‘) followed by an integer. NIRVA considers that as a number of days before the current day. For example, “-3” is 3 days before the current day.

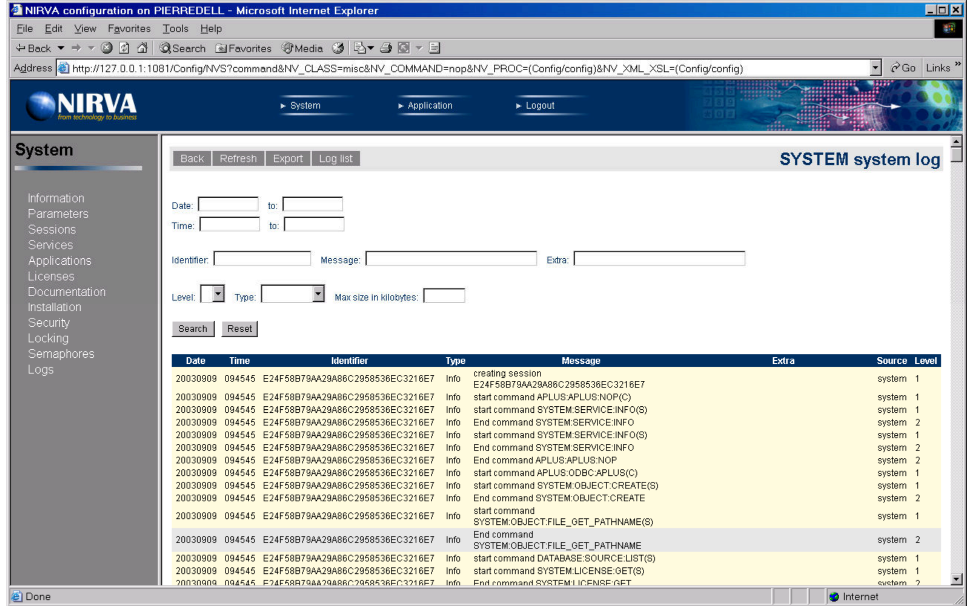
The default value for the “date” field depends of the parameter “to date”. If “to date” is not provided, the command doesn’t remove anything. If “to date” is provided and “date” is empty, NIRVA sets “date” to “01.01.2000”. This allows removing all log data to a given date.

The default value for the “to date” field is the value of the DATE field.

The “Clear” option allows to also removing current log data. The current log data is this one that is not saved. Physically, it’s stored in 2 files named “nvs.log” and “nvs1.log”.

Displaying log data

In order to display some log data, just click on the log name in the log list. NIRVA will then display the following screen:



It’s a list of log records ordered by date. It’s possible to limit the search of log data by entering some search criteria. By default, NIRVA searches for the log data for the last 15 minutes.

This log search can be quiet long in some circumstances. In order to avoid starting too big searches, NIRVA limits the size of an internal log data file (constructed with only the date range criteria) to the first 1024 kilobytes. This limit can be changed by setting the “Max size in kilobytes” box to another value.

The log result displays the following columns:

- Date is the record write date (format YYYYMMDD.
- Time is the record write time (format HHMMSS.mmm).
- Identifier is a free text that identifies a user entity. For example, for the system log, the identifier is the NIRVA session identifier.
- Type is the kind of record. It can be ‘Info” for information record, “Warning” for a warning record and “Error” for an error record.
- Message is the log message itself. It’s textual information.
- Extra is the optional second message.
- Source is a free text that gives the origin of the log record.
- Level is the log level. It’s numeric information that takes values from 1 to 6.
- Thread is the ID of the thread that produced the log entry.

Search criteria

Date

From date. This parameter allows limiting the search to a given date or to a date range when used with the “to date” parameter. The general format of the “date” parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only DD/MM is given, NIRVA is using the current year.
If only DD is given, NIRVA is using the current month and year.
NIRVA can also accept the year on 2 digits.
If the “date” is preceded by a minus sign (‘-‘) followed by an integer. NIRVA considers that as a number of days before the current day. For example, “-3” is 3 days before the current day.
The default value for the “date” field depends of the parameter “time”. If “time” is provided, the default “date” parameter is the current date. If “time” is not provided, the default “date” parameter is date of the day 15 minutes before the current date (so it’s the current day if the command is sent after 00:15).

To date

To date. This parameter allows limiting the search to a date range. If it’s not provided, the search occurs for a fixed date given by the “date” parameter.
If the “to date” parameter is given but the “date” parameter is blank (or not given), NIRVA resets the “to date” parameter to blank. At this time, the search occurs only on the current date.
The general format of the “to date” parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only DD/MM is given, NIRVA is using the current year.
If only DD is given, NIRVA is using the current month and year.
NIRVA can also accept the year on 2 digits.
If the “to date” is preceded by a minus sign (‘-‘) followed by an integer. NIRVA considers that as a number of days before the current day. For example, “-3” is 3 days before the current day.
The default value for the “to date” field is the value of the “date” field.

Time

From time. This parameter allows to limit the search to a given date/time or to a date/time range when used with the “to date” and “to time” parameters. The general format of the “time” parameter is HH:MM:SS where HH is the hour from 0 to 23, MM is the minute from 0 to 59 and SS is the second from 0 to 59.

Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be '/', '\', '.', ' '(space) or ':'. It's also possible to not give the separator.
If only HHMM is given, NIRVA is using 0 for the seconds.
If only HH is given, NIRVA is using 0 for the seconds and for the minutes.
If the "time" is preceded by a minus sign ('-') followed by an integer. NIRVA considers that as a number of hours before the current time. For example, "-3" is 3 hours before the current time. If the integer is followed by a "m" character, NIRVA considers that as a number of minutes before the current current time. For example, "-10m" is 10 minutes before the current time.
The default value for the "time" field depends of the parameter "date". If "date" is provided, the default "time" parameter is 00:00:00. If "date" is not provided, the default "time" parameter is time of the day 15 minutes before the current time.

To time

To time. This parameter allows limiting the search to a given date/time or to a date/time range when used with the "date" and "time" parameters.
The general format of the "to time" parameter is HH:MM:SS where HH is the hour from 0 to 23, MM is the minute from 0 to 59 and SS is the second from 0 to 59.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be '/', '\', '.', ' '(space) or ':'. It's also possible to not give the separator.
If only HHMM is given, NIRVA is using 0 for the seconds.
If only HH is given, NIRVA is using 0 for the seconds and for the minutes.
If the "to time" is preceded by a minus sign ('-') followed by an integer. NIRVA considers that as a number of hours before the current time. For example, "-3" is 3 hours before the current time. If the integer is followed by a "m" character, NIRVA considers that as a number of minutes before the current time. For example, "-10m" is 10 minutes before the current time.
The default value for the "to time" field depends of the parameter "date". If "date" is provided, the default "to time" parameter is 23:59:59. If "date" is not provided, the default "to time" parameter is the current time.

Message

Log message. This parameter allows to limit the search to a given log message string. The wildcard character '*' can be used for searching messages starting with a given value or ending with a given value. If the '*' wildcard character is used before and after the search value, NIRVA will find messages containing the requested value.

Extra

Log extra information. This parameter allows to limit the search to a given log extra message string. The wildcard character '*' can be used for searching messages starting with a given value or ending with a given value. If the '*' wildcard character is used before and after the search value, NIRVA will find extra information containing the requested value.

Type	Log record type. This parameter allows to limit the search to a given log type.
Level	Log level. This parameter allows limiting the search to a maximum log level. This is a numeric value. It can take values "1" to "6".
Identifier	Identifier. This parameter allows to limit the search to a given log identifier string (for example a specific session). The wildcard character '*' can be used for searching messages starting with a given value or ending with a given value. If the '*' wildcard character is used before and after the search value, NIRVA will find identifiers containing the requested value.
Source	Log record source. This parameter allows to limit the search to a given source string. The wildcard character '*' can be used for searching messages starting with a given value or ending with a given value. If the '*' wildcard character is used before and after the search value, NIRVA will find sources containing the requested value.

Exporting log data

In order to export the log data displayed in the log data screen, just click on the "Export" link at the top of the log data page.

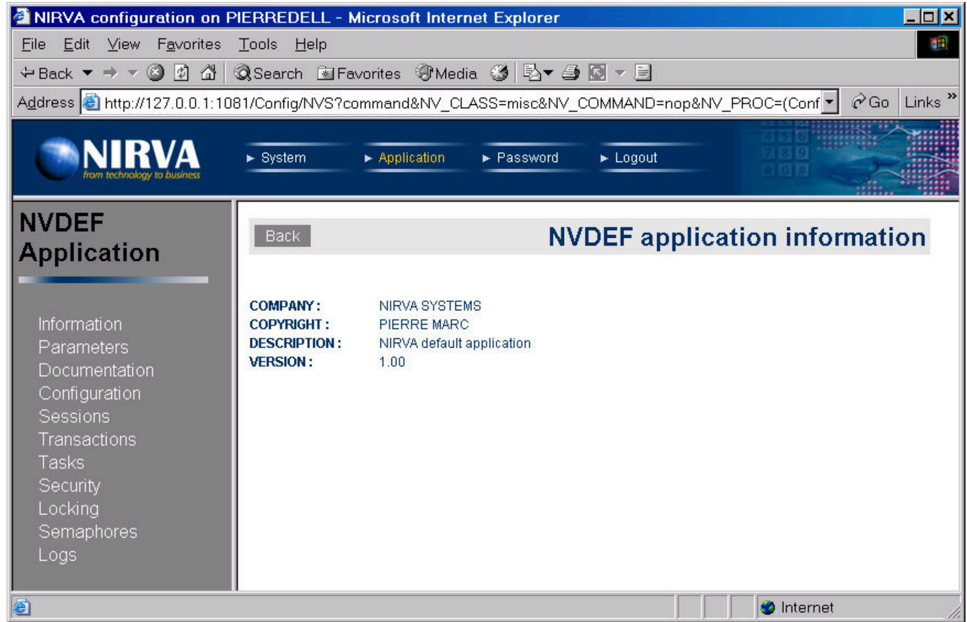
At this time, NIRVA launches the search again and prepares an export file that is to be downloaded locally. This export file can then be import to a newly created log (see the chapter "Creating a new log" for further information).

Since the export button does the search again, the log data can be different than the currently displayed one.

The log export can be quiet long in some circumstances.

Application

When choosing the main "Application" link, NIRVA displays the following screen:



The left part of the screen provides the “Application” configuration menu. When clicking an item of a menu, NIRVA displays information about this item in the main frame.

When entering the application configuration menu, NIRVA automatically activate the “Information” item.

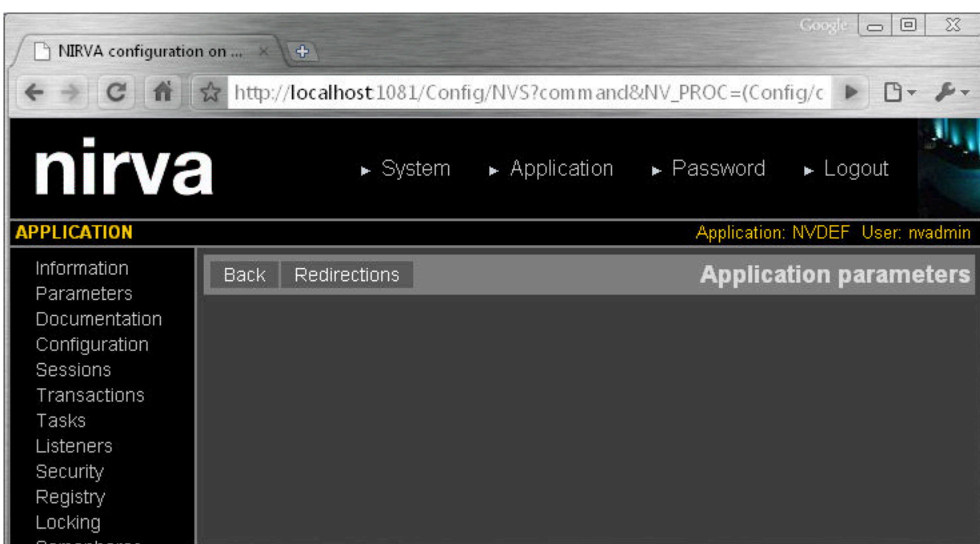
This part of the configuration concerns the connected application (this one chosen at login time). It’s different than the system application menu that manages the list of applications available on NIRVA.

Information

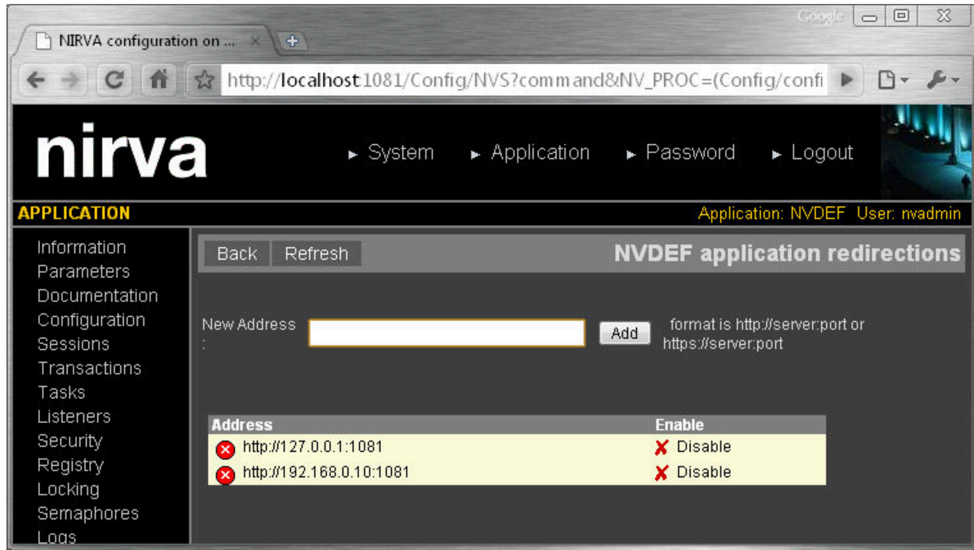
This displays the application information that can be found in the info section of the “application.dsc” file. This file resides on the application files directory.

Parameters

This option allows changing some application parameters. It displays the following screen:



The redirection button controls the front-end load balancing for the application:



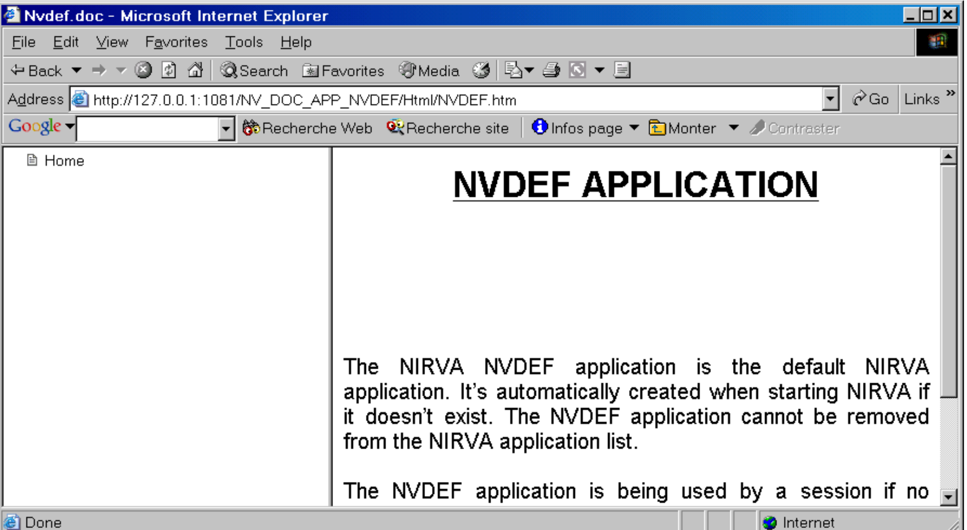
Nirva uses HTTP redirection for managing front-end load balancing. This screen defines a list of servers that will be used for redirection. Redirection works on Web, Xml, Soap and web service client connectors. All machines involved in the front-end load balancing should have the same list of server addresses.

Redirection occurs only when a new session is to be created.

When redirection is configured, setting NV_NO_REDIRECT parameter to “YES”, disable the redirection for the new session.

Documentation

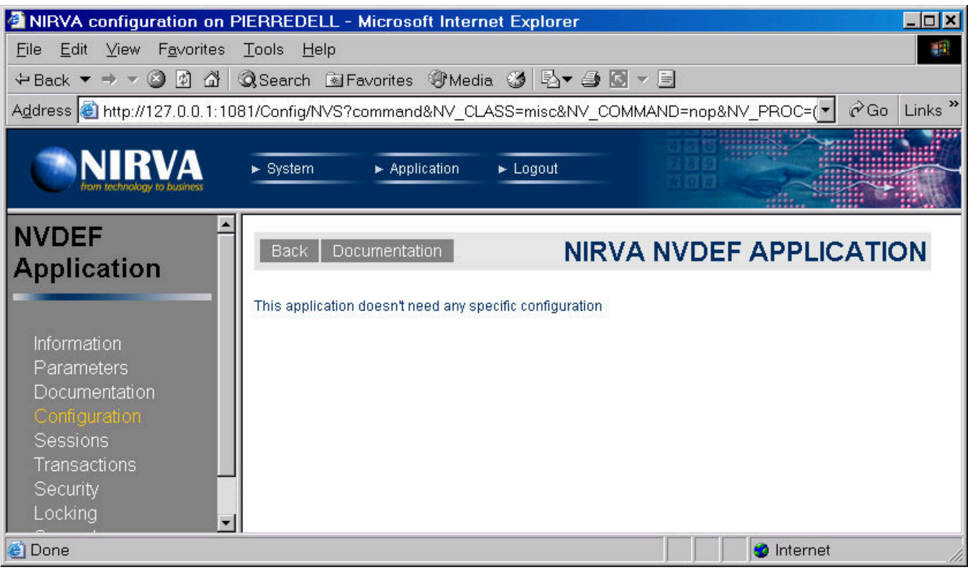
The “Documentation” menu just displays the application documentation in its HTML form. The documentation is displayed in another browser.



Configuration

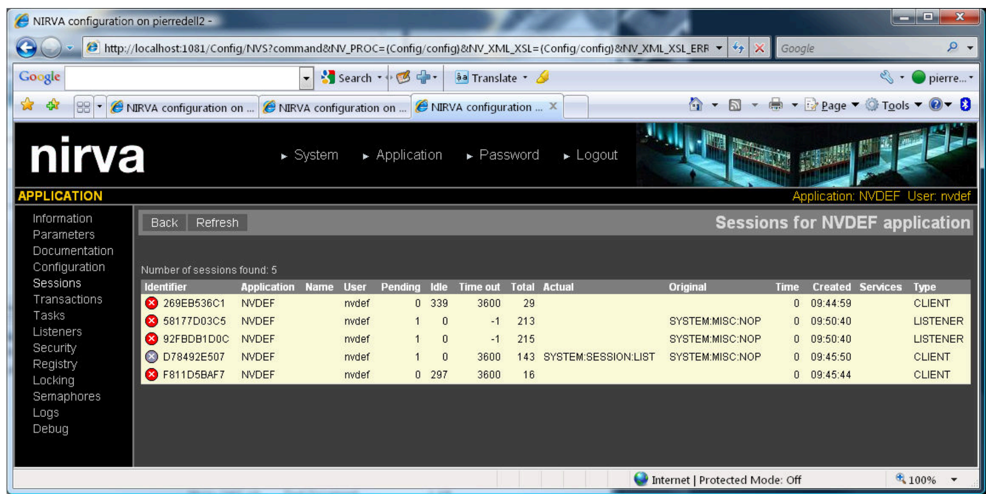
The “Configuration” menu displays the application specific configuration screen. This configuration depends of the application itself and is delivered by the application provider. When creating an application, NIRVA also creates the necessary skeleton for the application configuration.

For example, the default application (NVDEF) displays this configuration screen:



Sessions

By clicking on the “Sessions” menu, NIRVA displays the detail list of sessions connected to the application:



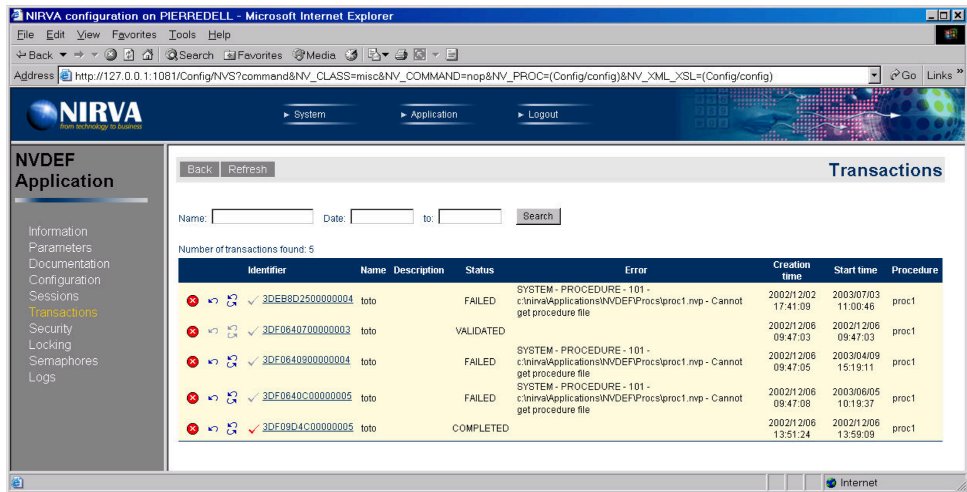
For each session, the list displays the following information:

- “Identifier” is the session unique identifier. It’s possible to remove a session by clicking on the ⊗ icon in the list. All sessions can be closed except the calling one. Before closing a session, NIRVA displays a confirmation message.
- “Application” is the name of the application connected by the session.
- “User” is the application user name if there is one.

- “Pending” is the number of connections currently using the session (active command). This number will be generally 0 or 1.
- “Idle” is the number of seconds from the end of the last command.
- “Time out” is the session time out in seconds.
- “Total” is the number of commands already processed by the session since it has been created.
- “Actual” is the command currently processed by the session. The current command is given on the form SERVICE:CLASS:COMMAND.
- “Original” is the original command which has generated the current command. The original command is always a command sent from a client (browser or service or internal) while the current command may come from a procedure or a service.
- “Time” is the number of seconds from the starting of the current command if there is one in process.
- “Created” is the date and time of the session creation. If the session has been created the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- “Services” is the list of services used by the session. It’s possible to go directly to the specific service information by clicking on the service name.
- “Type” is the session type. It can be CLIENT, TRANSACTION, SCHEDULER, NAMED, SYSTEM, INTERNAL, LISTENER or THREAD.

Transactions

The “Transactions” option displays the list of transactions currently defined for the application:



By default, the list shows all the transactions but it’s possible to limit it to some transaction names or/and to a date range.

The date format can be dd.mm.yyyy, dd/mm/yyyy, dd.mm.yy or dd/mm/yy.

If the name finishes by the ‘*’ wildcard character, NIRVA searches for all transaction names starting with the given characters.

For each item of the list, NIRVA provides the following information:

- "Identifier" is the unique transaction identifier.
- "Name" is the transaction friendly name (can be empty).
- "Description" is the transaction description.
- "Status" is the current transaction status. It can be "NOT_STARTED", "RUNNING", "FAILED", "COMPLETED", "VALIDATED" or "NOT_VALIDATED".
- "Error" gives transaction error information if there is one available.
- "Creation time" is the date and time of the transaction creation. If the transaction has been created the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- "Start time" is the date and time of the transaction last start. If the transaction has been started (or restarted) the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- "Procedure" is the transaction procedure name (it can be Perl script, Dotnet or Java class). It may have parameters.

Please see the Features paragraph and the SYSTEM:TRANSACTION class reference for further information about transactions.

Removing a transaction

In order to remove a transaction, just click on the icon at the left of the transaction identifier. Nirva will then display a confirmation message before removing the transaction.

The removing is possible only if the transaction is not in use. NIRVA waits a certain time for the transaction to be completed and returns an error message if it cannot remove it.

Rolling back a transaction

In order to roll back a transaction, just click on the icon at the left of the transaction identifier. Nirva will then display a confirmation message before rolling back the transaction.

The rollback is done if there is some rollback procedures defined.

It's not possible to rollback a transaction having the status "RUNNING" or "VALIDATED".

After a successful rollback, the transaction goes to the status "NOT_STATED".

The rollback is made by creating a new session specific to the transaction. This session is automatically closed after the end of the rollback. The new session starts with the transaction current saved context.

If the status of the TRANSACTION is "NOT_STARTED", the rollback command does nothing.

If the status of the TRANSACTION is "FAILED", the restart first restores the current context and then runs the rollback procedure for failed transaction (if there is one defined)

If the status of the TRANSACTION is "COMPLETED" or "NOT_VALIDATED", the restart first restores the current context and then runs the rollback procedure for completed transaction (if there is one defined).

The rollback always occurs with the user name used when the transaction has been created.

The rollback fails if the transaction is in use.

Restarting a transaction

In order to restart a transaction, just click on the icon at the left of the transaction identifier. Nirva will then display a confirmation message before restarting the transaction.

It's not possible to restart a transaction having the status “RUNNING” or “VALIDATED”.

The restart is made by creating a new session specific to the transaction. This session is automatically closed after the end of the restart. The new session starts with one of the transaction saved context: initial or current context:

If the status of the TRANSACTION is “NOT_STARTED”, the restart directly occurs with the initial saved context.

If the status of the TRANSACTION is “FAILED”, the restart first restores the current context, runs the rollback procedure for failed transaction (if there is one defined) and restarts the transaction procedure after having restored the initial context.

If the status of the TRANSACTION is “COMPLETED” or “NOT_VALIDATED”, the restart first restores the current context, runs the rollback procedure for completed transaction (if there is one defined) and restarts the transaction procedure after having restored the initial context.

The restart always occurs with the user name used when the transaction has been created.

The restart fails if the transaction is in use.

Validating a transaction

In order to validate a transaction, just click on the icon at the left of the transaction identifier.

The validation is made by creating a new session specific to the transaction. This session is automatically closed after the end of the restart. The new session starts with the current saved context.

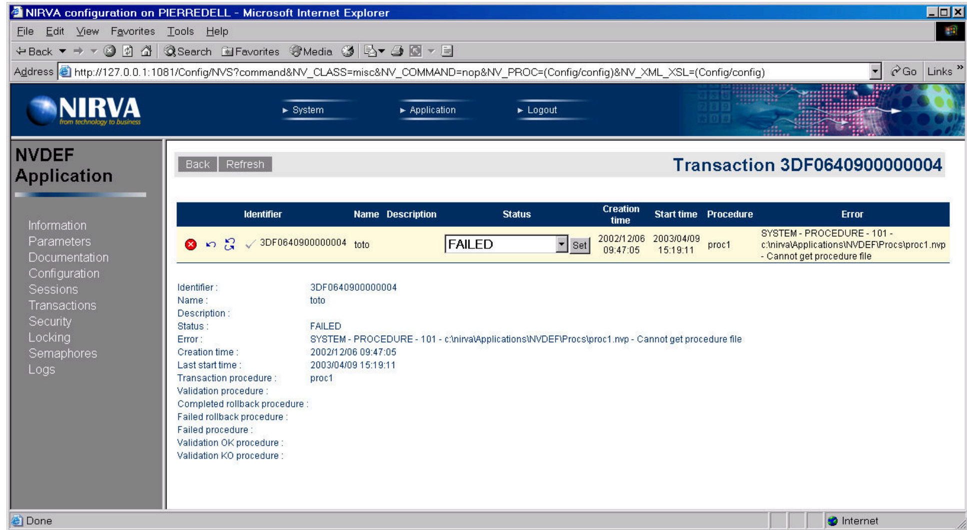
It runs the validation procedure and changes the transaction status to “VALIDATED” or “NOT_VALIDATED” following the result of the validation procedure.

The validation always occurs with the user name used when the transaction has been created.

The validation fails if the transaction is in use.

Detail of a transaction

In order to display some detail of a transaction, just click on the icon at the left of the transaction identifier. This then displays the following screen:

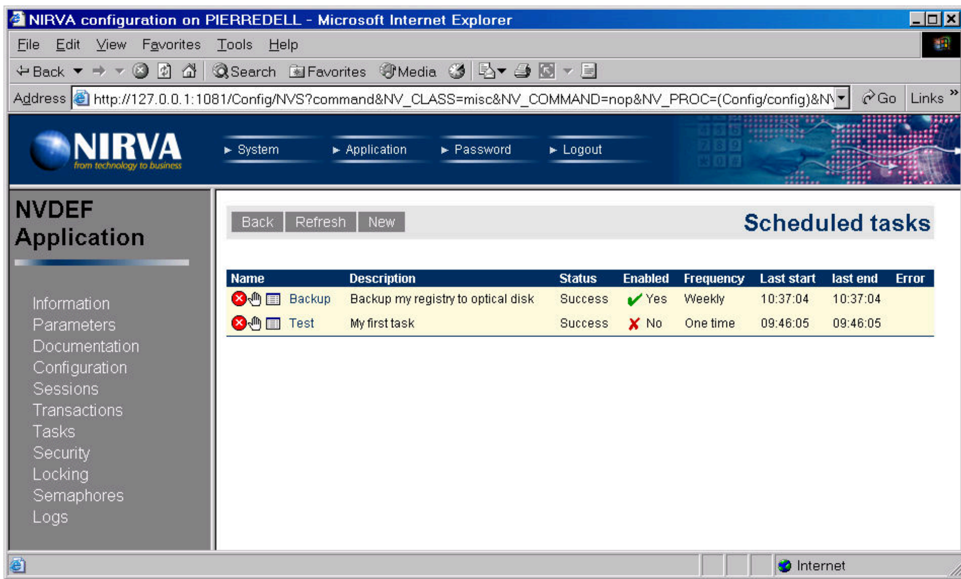


Form this screen, it's possible to remove, rollback, restart or validate the transaction like in the main transaction list but also to change the status of the transaction.

It's not possible to change the status to "RUNNING".

Tasks

The "Tasks" option displays the list of scheduled tasks currently defined for the application:



For each item of the list, NIRVA provides the following information:

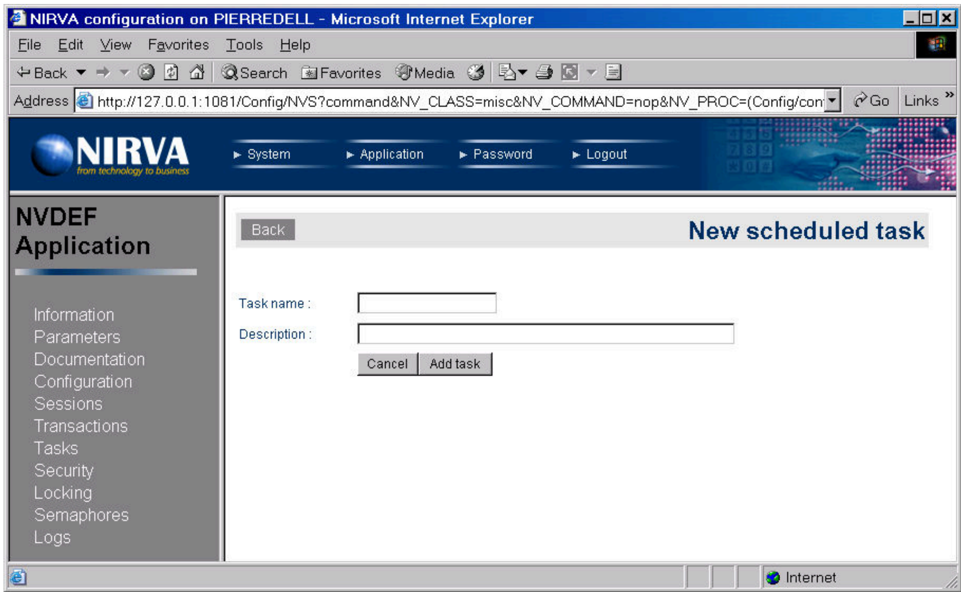
- "Name" is the task name that uniquely identifies the task.
- "Description" is the task description.
- "Status" is the current task status. It can be "Not started", "In queue", "Running", "Failed", "Success" or "Stopped".
- "Error" gives task error information if there is one available.
- "Enable" tells if the task is enabled for scheduling or not.

- "Frequency" gives the task frequency. A task can have the following frequencies: one time, daily, weekly or monthly.
- "Last start" is the last start time and date of the task. If the date is the current date (today), only the time is given. If the task has been ran manually a "(M)" string follows the last start time.
- "Last end" is the last end time and date of the task. If the date is the current date (today), only the time is given.
- "Error" is the eventual error of the last task occurrence.

Please see the Features paragraph and the SYSTEM:SCHEDULER class reference for further information about scheduler.

Creating a new task

For creating a new task, click on the "New" button. This then displays the following screen:



Please enter a task name (required) and an eventual task description.

Removing a task

In order to remove a task, just click on the icon at the left of the task name. Nirva will then display a confirmation message before removing the task.

The removing is possible only if the task is not in use. NIRVA waits a certain time for the task to be completed and returns an error message if it cannot remove it.

Running a task

Normally, a task is being run by the NIRVA scheduler but it's possible to immediately run a task by clicking on the icon at the left of the task name. Nirva will then display a confirmation message before starting the task.

If the task is already running or if no procedure has been defined for the task, NIRVA doesn't run it.

This way for running a task is completely independent of the task scheduler parameters so the task is ran immediately even if it's not enabled.

This allows to test some tasks and to immediately restart a task in case of error.

The task is being run in background in a separate thread so the command doesn't wait for the task to terminate. Please use the refresh button in order to see the task status.

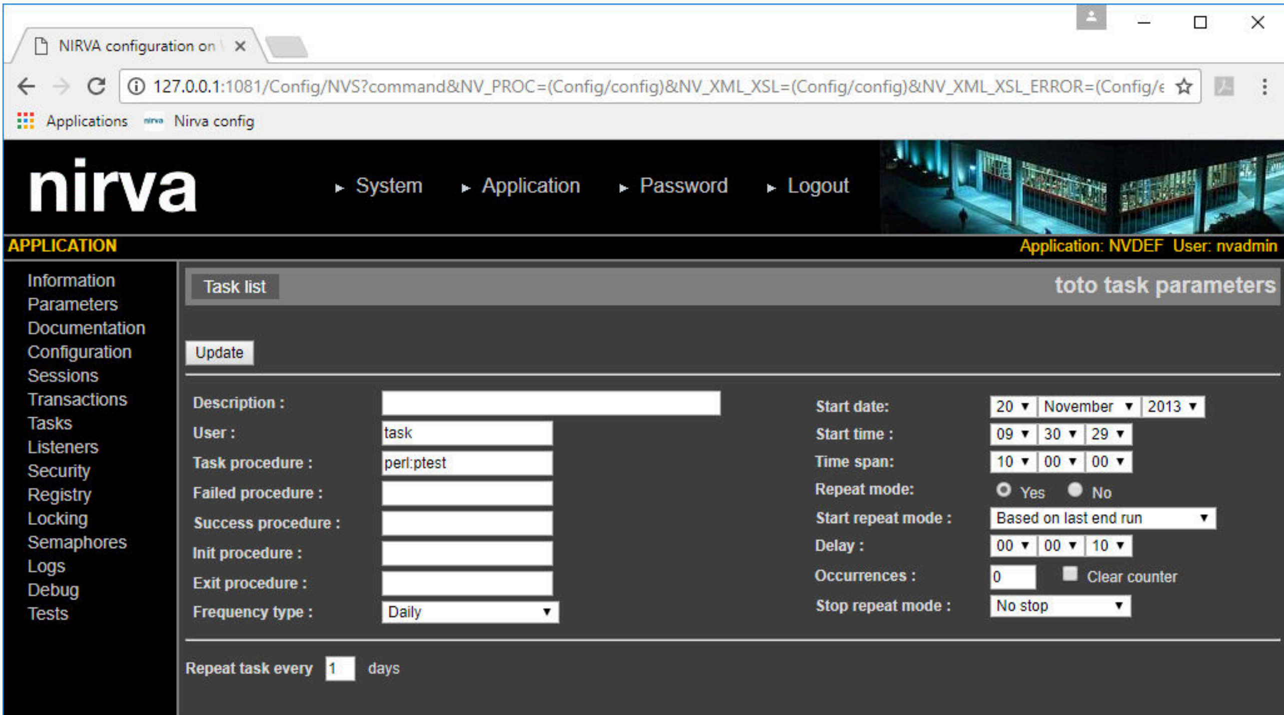
Enabling or disabling a task

A task can be enabled for being used by the scheduler or not.

In order to enable or disable a task, just click on the ✔ or ✘ icons in the "Enabled" column in the task list.

Setting task parameters

For setting the task parameters, just click on the icon at the left of the task name. This display the following screen:



The task parameters screen is divided in two parts. The first part gives general task parameters and the second part allows setting frequency specific parameters.

In order to validate changes, press the "Update" button.

General parameters

Description This is a free string describing the task.

User	User name that will run the task. The task is always run in background by creating a new session. By default, the task runs on the name of the user who has created it but it's possible to change to another user name. This possibility is subject to a special permission in the user security profile. If the User field is let blank, NIRVA uses the current user.
Task procedure	Name of the procedure to be ran by the task. This parameter is required or the task will never run. The task procedure receives the parameter NV_TASK_NAME that contains the task name. Please see the Calling a procedure chapter for description of the procedure syntax.
Failed procedure	Name of a procedure that will be ran in case of failure. This parameter is not mandatory. The failed procedure can be used for error reporting. It receives the error information as following parameters: NV_TASK_NAME, NV_ERROR_CODE, NV_ERROR_SERVICE, NV_ERROR_CLASS, NV_ERROR_DESC and NV_ERROR_INFO.
Success procedure	Name of a procedure that will be ran in case of success. This parameter is not mandatory. The task success procedure receives the parameter NV_TASK_NAME that contains the task name.
Init procedure	Name of a procedure that will be ran before starting the task procedure itself. In fact, when the scheduler starts an instance of a task, it creates a new session and closes it after the end of the task. This parameter is used as the session open procedure name and is executed in the same conditions than for a normal user session (possibility to set specific session permission, for example). This parameter is not mandatory.
Exit procedure	Name of a procedure that will be ran after ending the task procedure itself. In fact, when the scheduler starts an instance of a task, it creates a new session and closes it after the end of the task. This parameter is used as the session close procedure name and is executed in the same conditions than for a normal user session. This parameter is not mandatory.
Frequency type	This defines the frequency of the task. It can be “One time” for a one shot task, “Daily”, “Weekly”, “Monthly based on days” or “Monthly based on weeks”. When changing this parameter, NIRVA updates the general task parameters and displays some specific frequency parameters.
Start date	Minimum starting date of the task. The task will not run before this date.
Start time	Starting time of the task. For each day where the task is able to run, the scheduler defines a starting time so the task will not run before this time.
Time span	Amount of time for the task to start after the start time. The parameters Start time and Time span defines a time window into which the task can run. For example, if Start time has been set 22:30:00 and Time span to 03:00, the task will be able to run from 22:30:00 to 01:30:00 the next day.
Repeat	During the time span, the task can be eventually run several times. For example, it's possible to schedule a task running each day from 23:00:00 to 01:00:00 with infinite occurrences and a delay of 10 seconds between each

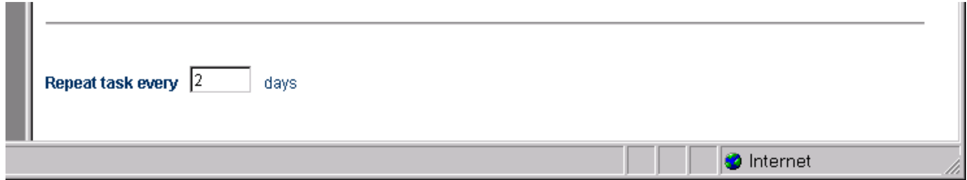
	occurrence. The repeat mode can be disabled (So there is only one occurrence each day).
Start repeat mode	3 modes are available. “Based on last end run”: The next occurrence starts after the defined delay of the last occurrence stop time. “Based on last start run”: the next occurrence starts after the defined delay of the last occurrence start time. If the current occurrence is not terminated at the time of start, the next occurrence starts when the current one has finished. “Based on last start run (fixed): the next occurrence starts after the defined delay of the last occurrence start time. If the current occurrence is not terminated at the time of start, the next occurrence doesn’t start.
Delay	Minimum delay between 2 successive occurrences on the same day. This parameter is meaningful only if the repeat mode has been set.
Occurrences	Maximum number of occurrences on the same day. This parameter is meaningful only if the repeat mode has been set. If Occurrences is set to 0, the number of occurrences is not limited.
Clear counter	The scheduler maintains a counter of the number of occurrences for each day. It’s automatically reset to 0 at the end of the time span. It’s possible to manually clear the counter by checking the “Clear counter” option.
Stop repeat mode	When the repeat mode is set, one can stop the task at first success or failure occurrence.

Frequency specific parameters

One time frequency

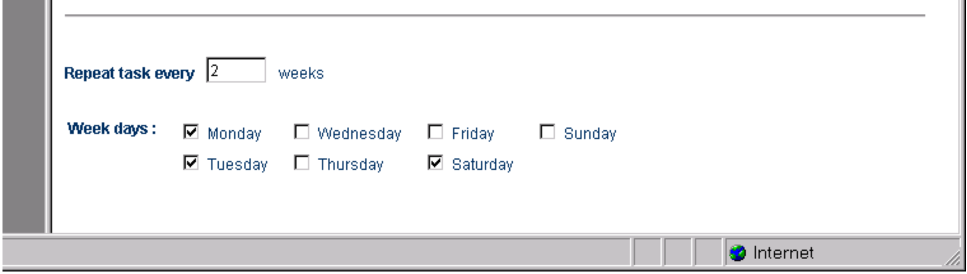
There are no specific parameters for this frequency type.

Daily frequency



Repeat days	Defines a day span for running task. The default is 1 (every day).

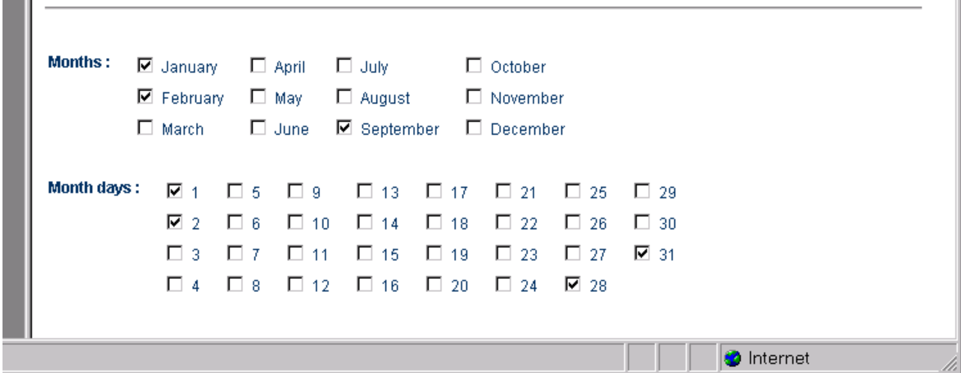
Weekly frequency



Repeat weeks Defines a week span for running task. The default is 1 (every week).

Week days Defines the week days for running the task.

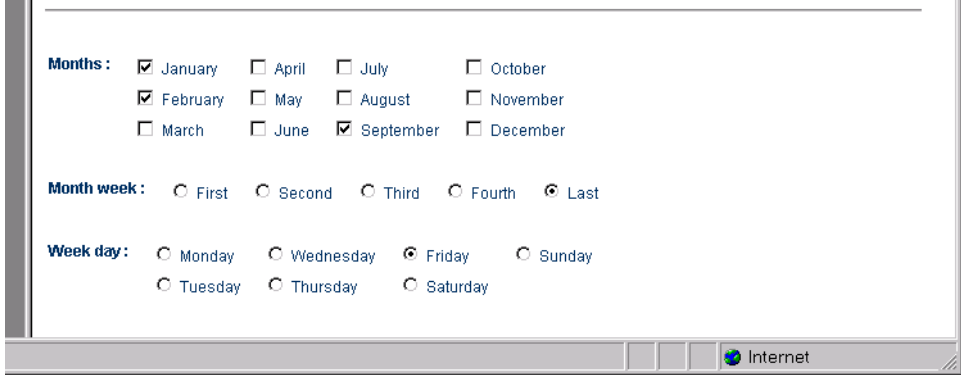
Monthly day based frequency



Months Defines the allowed months for running the task.

Month days Defines the allowed month days for running the task.

Monthly week based frequency



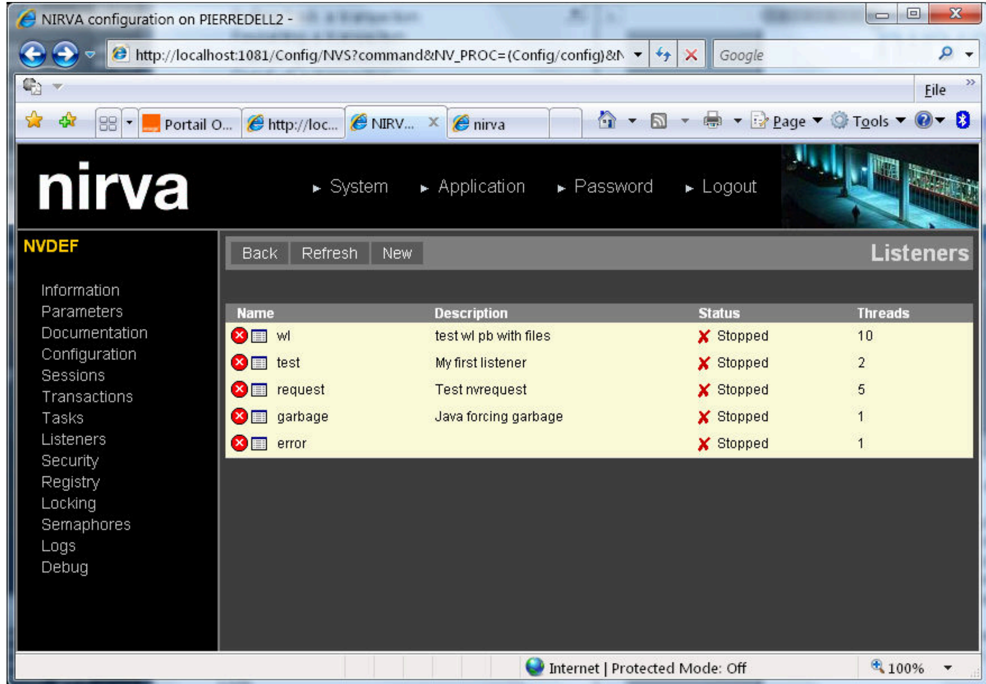
Months Defines the allowed months for running the task.

Month week Defines the single month week on which the task is authorized to run.

Week day Defines the day of the week.

Listeners

The “Listeners” option displays the list of listeners currently defined for the application:



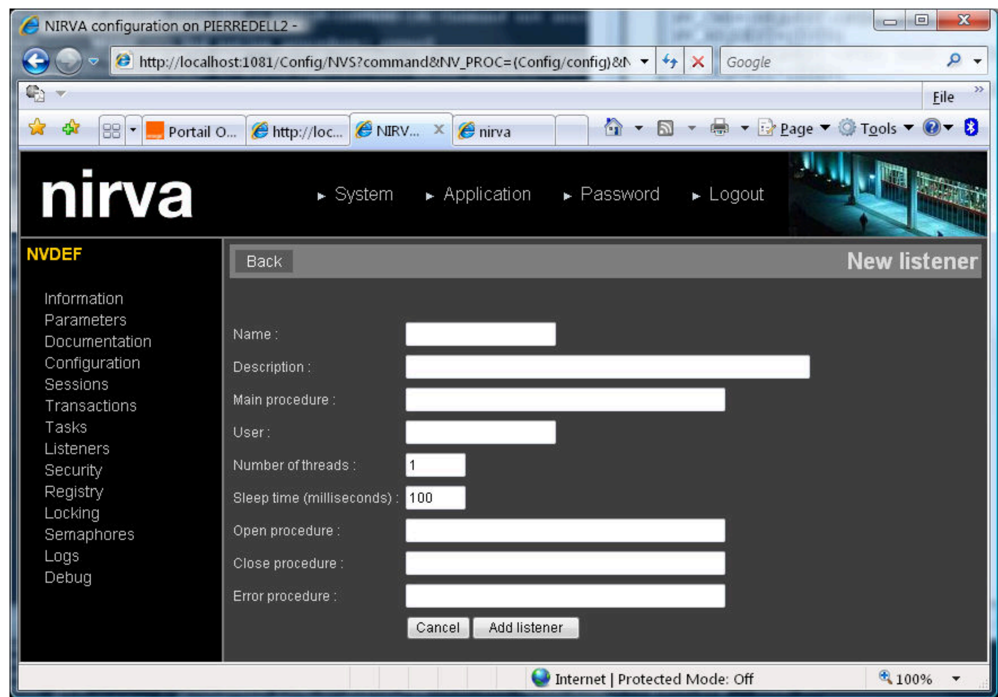
For each item of the list, NIRVA provides the following information:

- “Name” is the listener name that uniquely identifies the listener.
- “Description” is the listener description.
- “Status” is the current listener status. It can be “Stopped” or “Running”. For starting or stopping a listener, just click on the icon near the status.
- “Threads” is the number of threads attached to this listener.

Please see the Features paragraph and the SYSTEM:LISTENER class reference for further information about listeners.

Creating a new listener

For creating a new listener, click on the “New” button. This then displays the following screen:



Here is the parameters description:

Name — Listener name. This name identifies the listener. It can contain space characters. This parameter is mandatory.

Description — Optional description.

Main procedure — Name of the procedure that will be executed by the listener. If this parameter is not provided, the listener will be created with the given number of threads but the execution will do nothing.
The listener procedure will be executed in loop, each occurrence separated by a sleep time given in the SLEEP parameter. If the execution time of the procedure is long, it should include some calls to the SESSION:CHECK_CLOSE_REQUEST command and leave if there is a request to close. The main procedure receives the parameter NV_LISTENER_NAME that contains the listener name.
The listener procedure can be a native, dotnet, java or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name.

User — User name. This is the name of the user of the listener session. This must be a valid user. If this parameter is not provided, Nirva uses the default user (nvdef).

Open procedure — Name of the procedure that will be called when the listener session is opened. If the value is blank, no procedure will be called (default). If a procedure is given and the procedure returns an error, the listener session will not be created.
This can be a native, dotnet, java or perl procedure. See the description of

	the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
Close procedure	Name of the procedure that will be called when the listener session is closed. If the value is blank, no procedure will be called (default). This can be a native, dotnet, java or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
Failed procedure	Name of a procedure that will be ran in case of failure. This parameter is not mandatory. The failed procedure can be used for error reporting. It receives the error information as following parameters: NV_LISTENER_NAME, NV_ERROR_CODE, NV_ERROR_SERVICE, NV_ERROR_CLASS, NV_ERROR_DESC and NV_ERROR_INFO. If defined, the failed procedure is called each time the main procedure fails.
Number of threads	Number of threads to execute for this listener. The minimum value is 1 (default)
Sleep time	Sleep time in milliseconds between each occurrence of the thread. The default is 100 milliseconds. If set to 0, no sleep time will occur.
Open procedure	Name of the procedure that will be called when the listener session is opened. If the value is blank, no procedure will be called (default). If a procedure is given and the procedure returns an error, the listener session will not be created. This can be a native, dotnet, java or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
Close procedure	Name of the procedure that will be called when the listener session is closed. If the value is blank, no procedure will be called (default). This can be a native, dotnet, java or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).

Removing a listener

In order to remove a listener, just click on the icon at the left of the listener name. Nirva will then display a confirmation message before removing the listener.

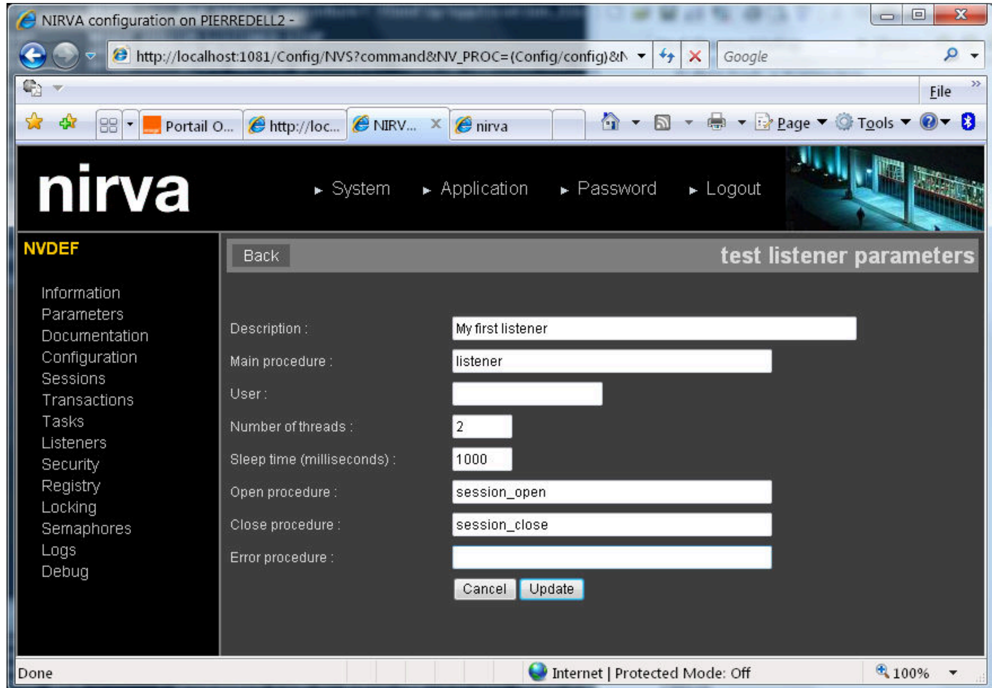
The removing is possible only if the listener is not running.

Staring or stopping a listener

In order to start or stop a listener, just click on the or icons in the “Status” column in the listener list.

Setting listener parameters

For setting the listener parameters, just click on the icon at the left of the task name. This is only possible when the listener is not running. This display the following screen:



The parameters are the same than in the create listener screen.

In order to validate changes, press the “Update” button.

Viewing listener parameters

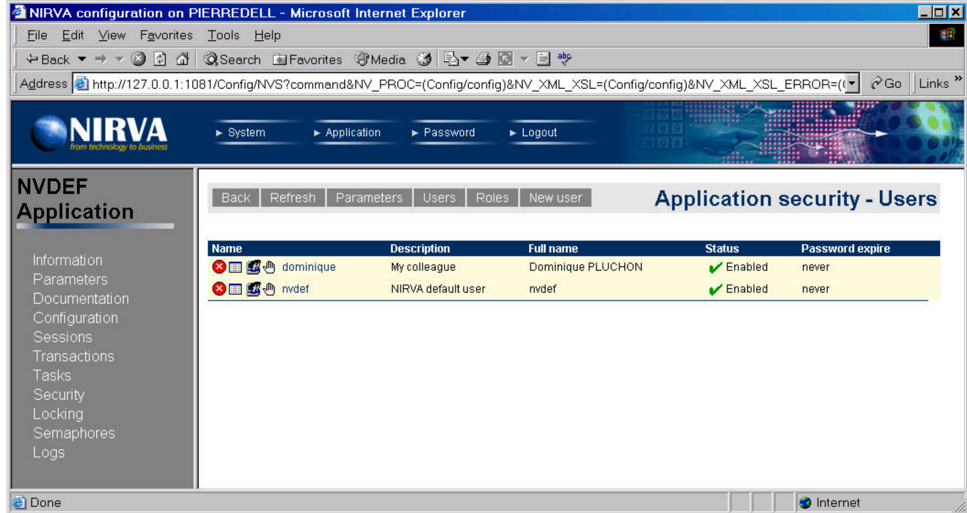
For viewing the current listener parameters, click on its name.

Security

With this menu, the user can administrate application security. The NIRVA security layer works at application layer but an application can choose to use the system security or the security of another application (also in a separate server) so the security managed from the application configuration is the one defined for the application (the application own security or another application security or the system security).

The security is described in the “Security model” chapter of this documentation. It’s a RBAC model (role based access control) that defines permissions, roles and users. The roles are hierarchical so a role can inherent the permissions of other roles.

If the security has been defined on another application and server, Nirva displays an error message. The user must then connect to the server on which the security resides to modify it.

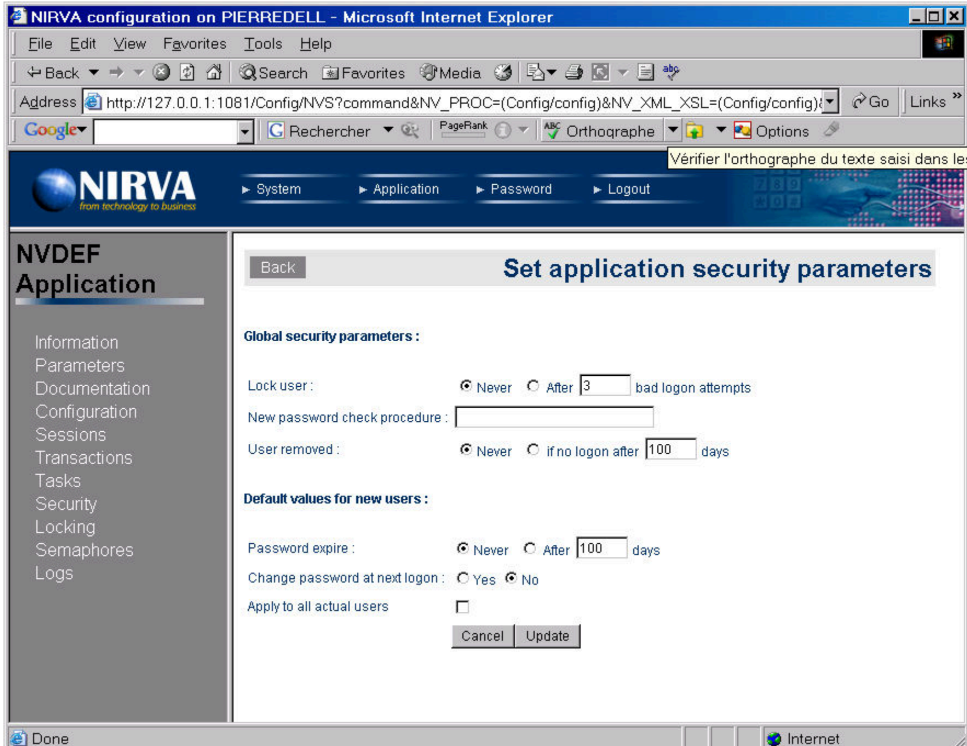


This screen displays the current user list. There is always at least one user named “nvdef” that is the default NIRVA user. The “nvdef” user cannot be removed or disabled.

The administrator user (“nvadmin”) is not displayed in this list.

Setting global parameters

Pressing the “Parameters” button allows changing some global security parameters:



From this screen one can change the following parameters:

- Number of bad logon attempts before the user to be locked.
- The name of a procedure for checking the new password syntax. This procedure, if it exists, will be launched by Nirva when the user is changing its password. It will receive 3 parameters

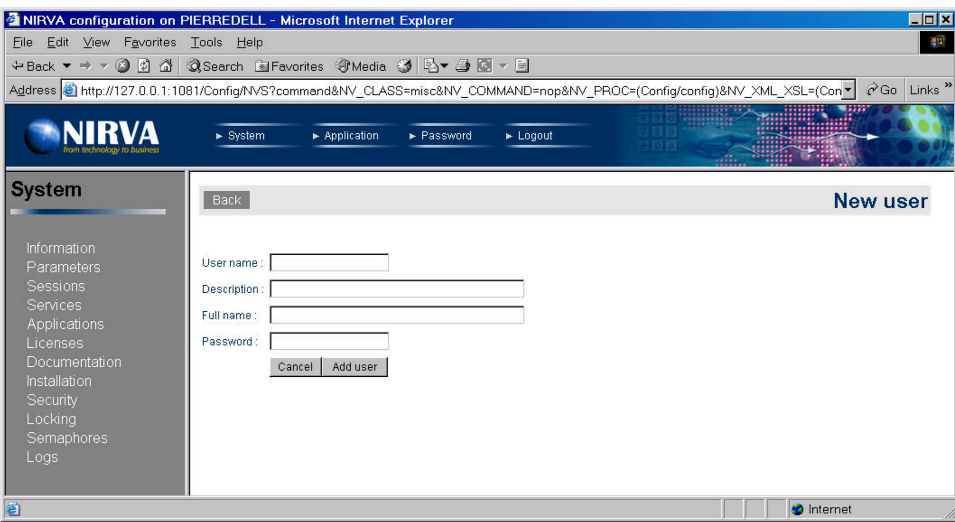
OLD_PASSWORD containing the old password, NEW_PASSWORD containing the new password and USER containing the user name. If the procedure produces an error (SetError function from Perl, Dotnet or Java procedure), Nirva will return a bad password syntax error to the user.

- The number of days after which unused users are automatically removed from the security.
- The default expiration time in days for new users.
- The flag for new user to change their password at first logon.

The last 2 parameters can eventually be applied to all current users.

Adding a new user

For adding a new user, click on the “New user” button:

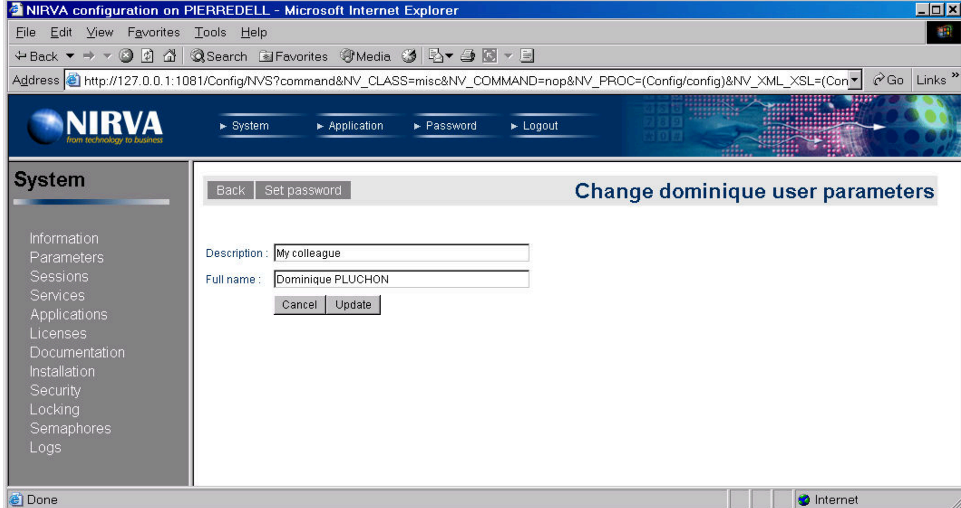


The user name is mandatory. It should not contain any space or special character and is case independent.

The other user parameters are optional.

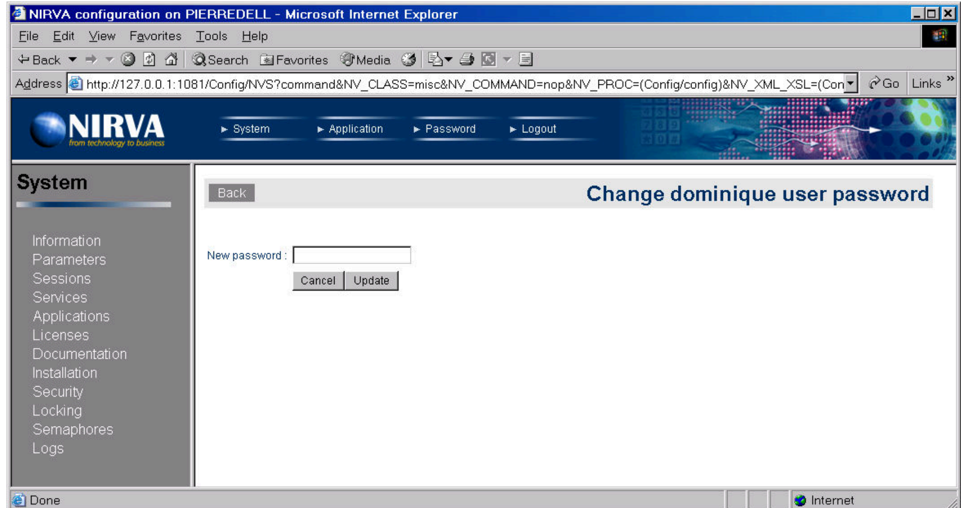
Changing user information

For changing some of the user information, click on the icon near its name:



The description, the user full name and expiration parameters can be changed.

If the current user of the configuration tool has the corresponding security permission, he can also change the user password by pressing the “Set password” button:



Removing a user

In order to remove a user, just click on the icon near its name. There is a confirmation message.

The default NIRVA user (nvdef) cannot be removed.

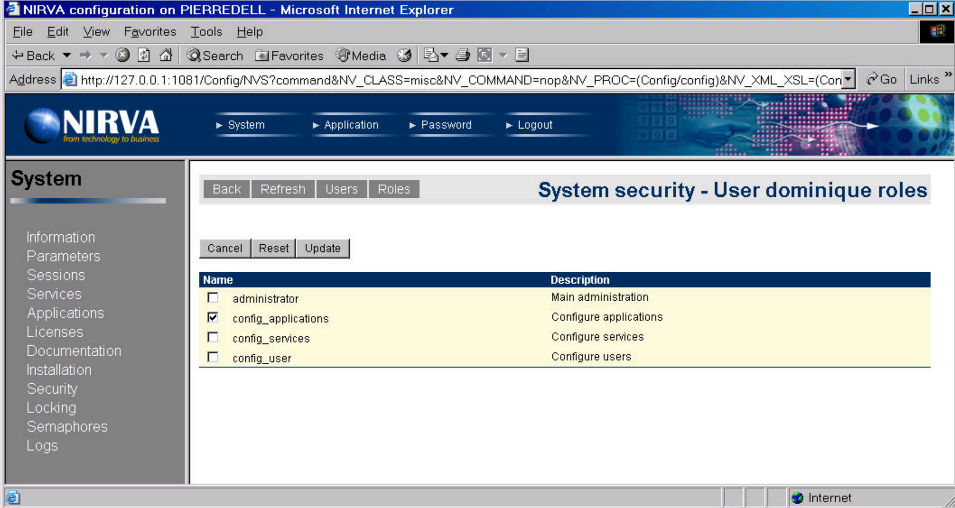
Enabling or disabling a user

A user can be temporary disabled without removing it from the user list. For that, just click on the icon on the status column.

If the user is disabled, you can enable it by clicking click on the icon on the status column.

Setting user roles

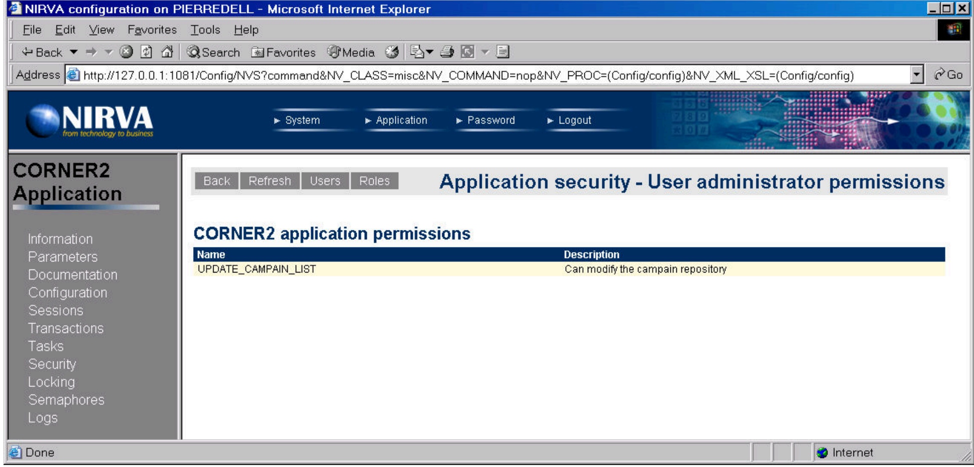
In order to set the user roles, just click on the icon near its name. This displays a list of possible roles with a check box for selecting them. If the user already has a role, the corresponding check box is checked when displaying the list:



For changing the user roles, just select the required roles and press the “Update” button.

Viewing user permissions

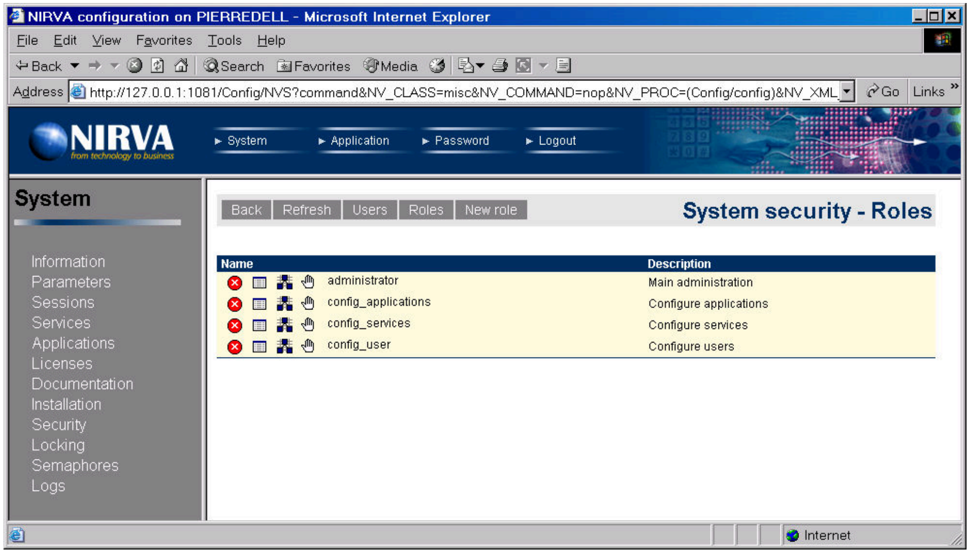
In order to view the detailed user permissions, just click on the icon near its name. This displays the list of the user permissions:



The permissions cannot be changed from this screen.

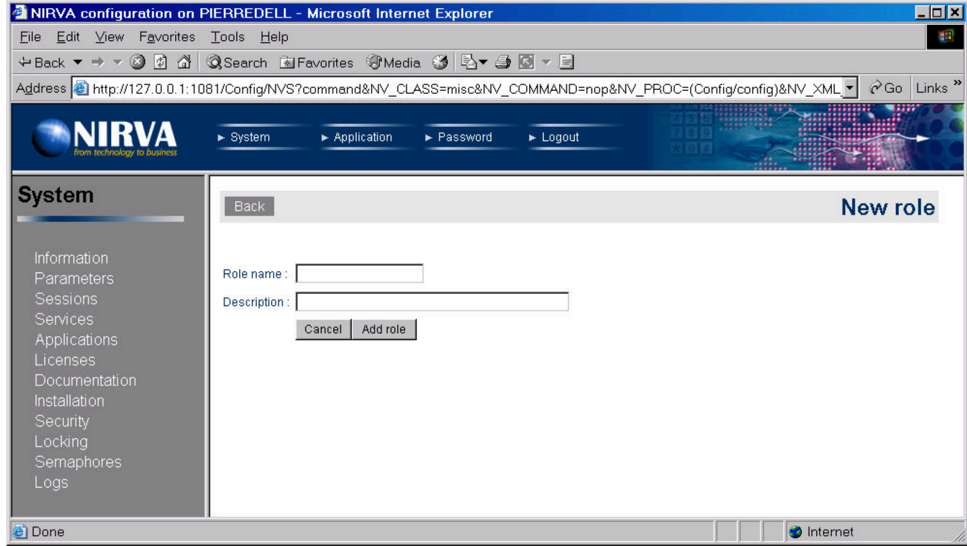
Displaying roles

The role list is available by clicking the “Roles” button. This displays the list of currently defined roles.



Adding a new role

For adding a new role, click on the “New role” button from the role list:

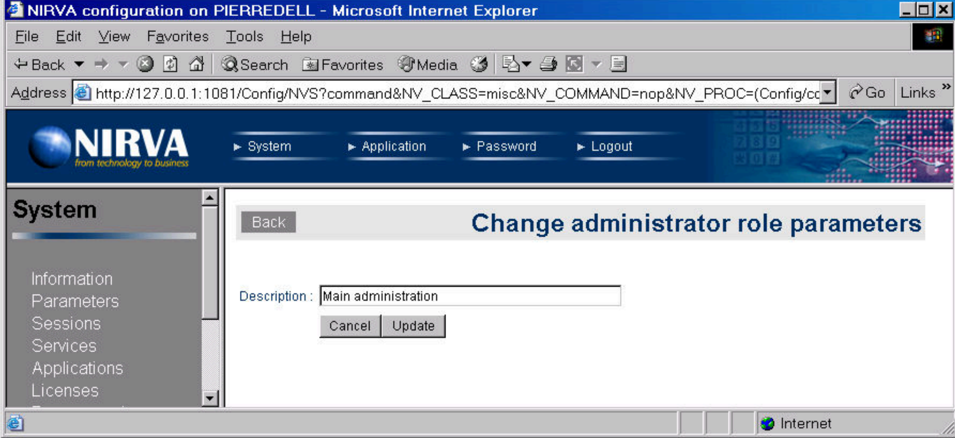


The role name is mandatory. It should not contain any space or special character and is case independent.

The other role parameters are optional.

Changing role information

For changing some of the role information, click on the icon near its name:



Only the description can be changed.

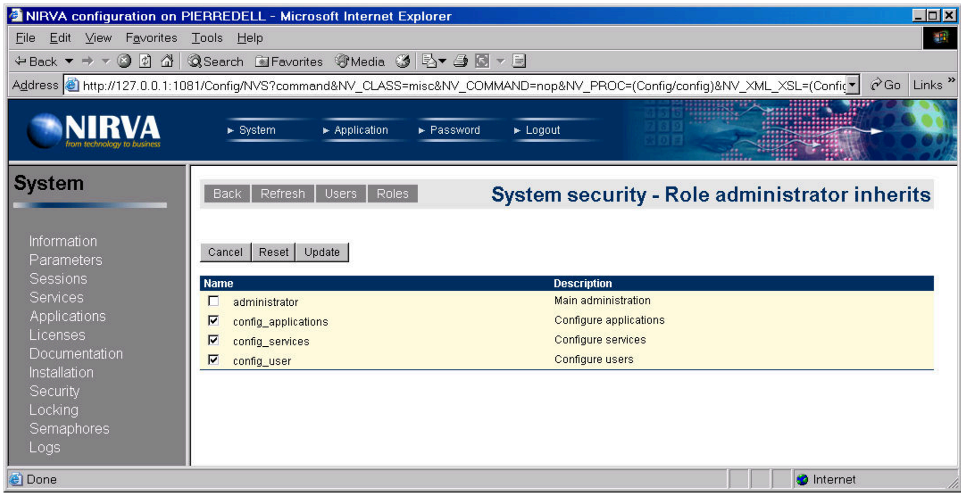
Removing a role

In order to remove a role, just click on the icon near its name (from the role list). There is a confirmation message.

Setting role inherits

A role can inherit the permissions of other roles.

In order to set the role inherits, just click on the icon near its name. This displays a list of possible roles with a check box for selecting them. If the role already inherits a role, the corresponding check box is checked when displaying the list:

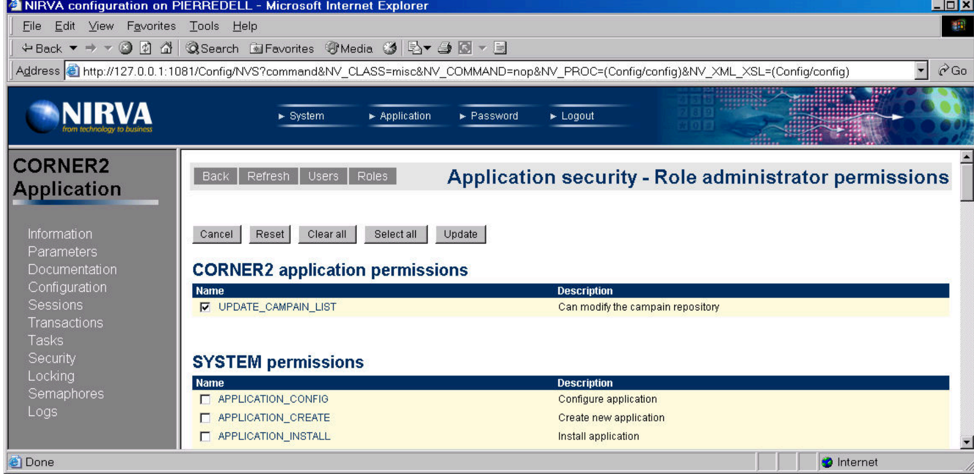


For changing the role inherits, just select the required roles and press the “Update” button.

NIRVA doesn’t display any message if there is a recursive problem in the hierarchy of roles but automatically take cares of this problem by stopping hierarchy when a re-entrant role is detected.

Setting role permissions

In order to set the role permissions, just click on the icon near its name. This displays a list of possible permissions with a check box for selecting them. If the role already has a permission, the corresponding check box is checked when displaying the list:

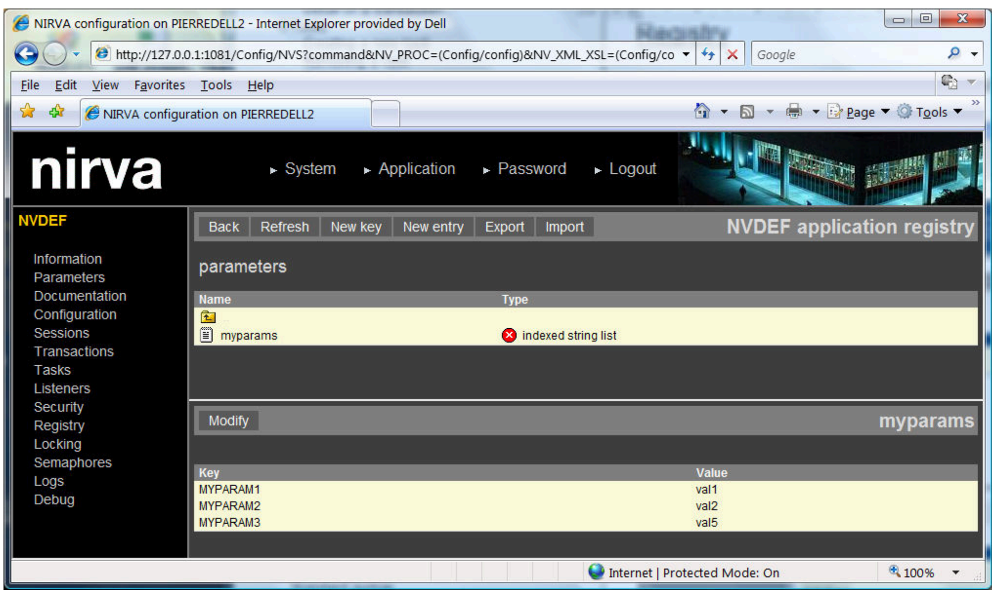


For changing the role permissions, just select the required permissions and press the “Update” button.

The permission list contains all the system, service, web service and application defined permissions.

Registry

The “Registry” option allows viewing and editing the application level registry:



There are two parts in the registry editor.

The upper part displays the current registry key. One can add, modify or remove entries. The registry key or entry cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character.

The lower part displays detail information about a given entry and allows editing it.

Specific entry instructions are displayed at the bottom of the screen when editing an entry. Here is an example for the table editing:

Cancel | Update | Data | Columns | Parameters — **mytable - data**

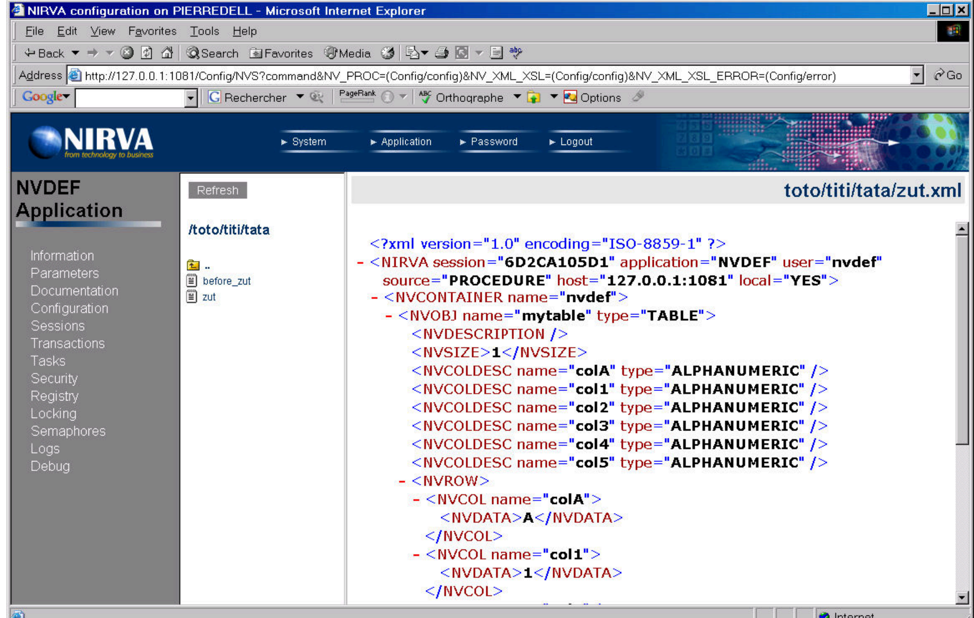
Press update button to save table data

```
1;2;3
4;5;6
```

Each line corresponds to one row.
The column separator is the ';' character.
The string separator is the '|' character (for cells containing several strings).
If a cell contains a line feed, it must be entered with the string '\n'.
If a cell contains a real semicolon, it must be entered with the string '\;'.
If a cell contains a real pipe, it must be entered with the string '\|'.
If a cell contains a real backslash, it must be entered with the string '\\'.

Debug

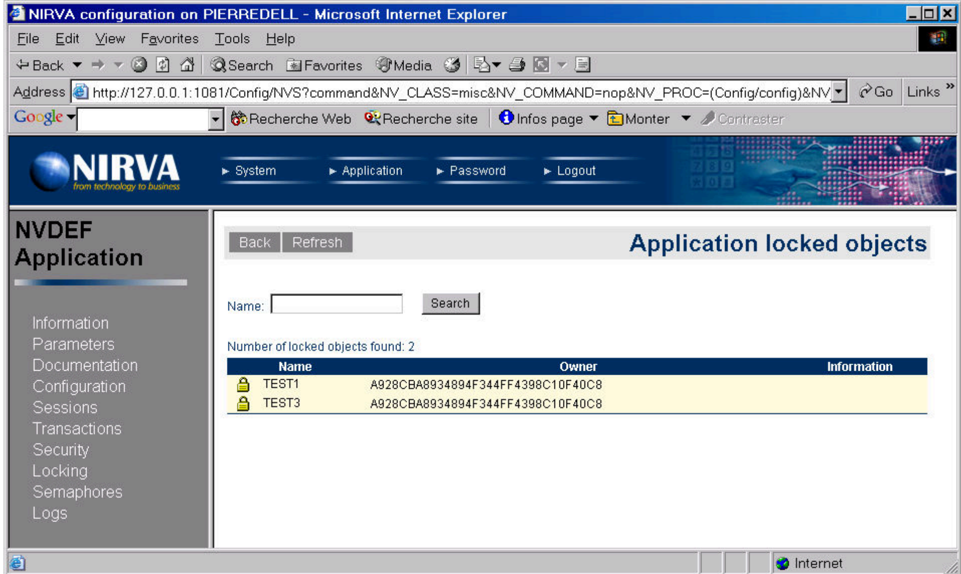
The “Debug” option allows viewing all debug xml files that reside in the application Files/Debug directory and subdirectories:



The files are written into the Debug directory when Nirva is in debug mode and when the NV_DEBUG_XML parameter is given or when the command comes from a URL (web connector).

Locking

The “Locking” option displays the list of application locked objects:



By default, the list shows all the locked objects but it's possible to limit it to some locked object names.

If the name finishes by the '*' wildcard character, NIRVA searches for all locked object names starting with the given characters.

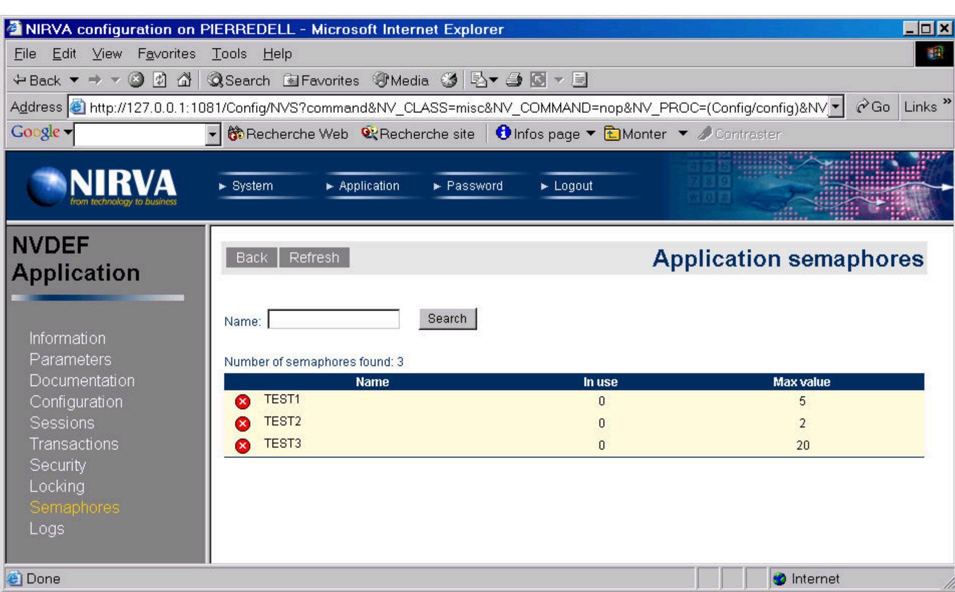
For each item of the list, NIRVA provides the following information:

- "Name" is the locked object name.
- "Owner" is the object owner. In fact it's the session identifier that owns the object.
- "Information" is the locked object optional information.

A locked object can be unlocked by clicking the 🔒 icon near its name. At this time, NIRVA provides a confirmation message.

Semaphores

The "Semaphores" option displays the list of application semaphores:



By default, the list shows all the semaphore objects but it's possible to limit it to some semaphore object names.

If the name finishes by the '*' wildcard character, NIRVA searches for all semaphore names starting with the given characters.

For each item of the list, NIRVA provides the following information:

- "Name" is the semaphore name.
- "In use" is the number of threads having requested an access to the semaphore.
- "Max value" is the maximum number of simultaneous accesses to the semaphore.

A semaphore can be removed by clicking on the ✖ icon near its name. At this time, NIRVA provides a confirmation message. If the semaphore is in use, the removing operation will fail.

In fact, NIRVA permanently checks for unused semaphores and automatically removes them.

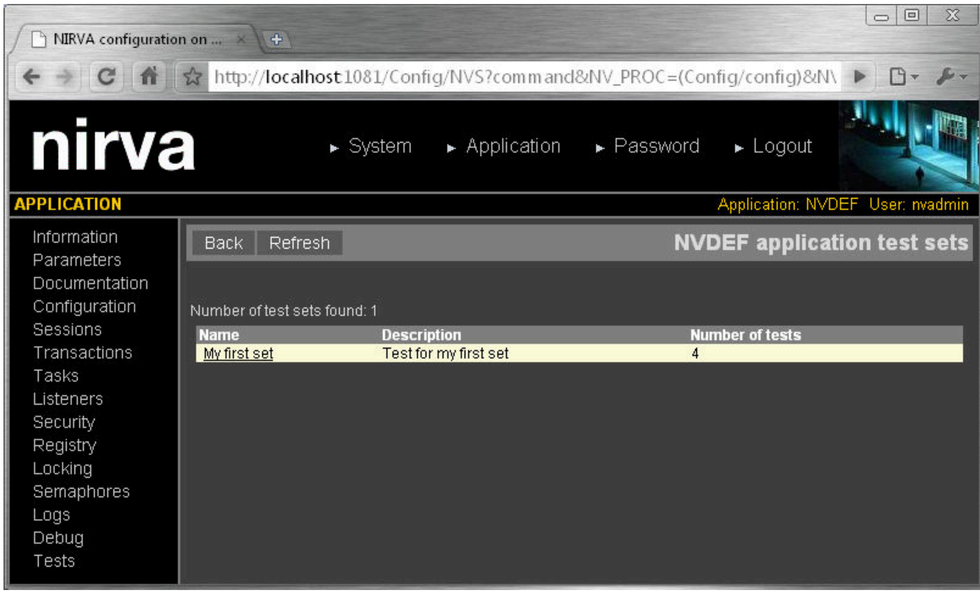
Logs

The log management at application level is the same than the log management at system level but it allows to access application logs only.

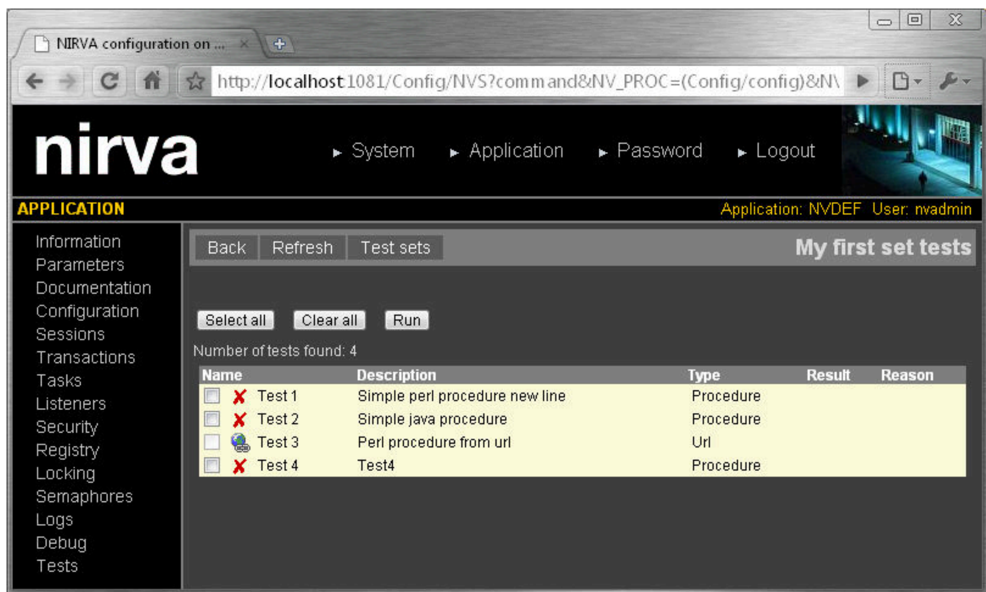
Please refer to the system configuration logs for further information about log features.

Tests

The Tests menu displays the list of defined test sets for the application (see the chapter "Test sets" in this documentation). It allows running the tests.



In order to run the tests defined for a test set, just click on its name:



“Select all” allows selecting all the tests of the set having the type “Procedure”.

“Clear all” resets the result and the selection of all the tests.

“Run” starts the selected tests of type “Procedure”. When running, the tests can be stopped by pressing the “Cancel” button.

In order to start an “Url” test type, just click on the the test icon.

The Nirva command syntax

Overview

The Nirva command is a simple string containing information and parameters about the command.

The Nirva server processes all commands it receives in the context of a Nirva session. The commands may come from:

- The Nirva client connectors (including Nirva nvc library)
- The Web browser or any HTTP client (including the XML, SOAP and web service connectors)
- A Nirva procedure
- A Nirva external service (inter-service communication feature)

The syntax differs a little bit if the command is sent from a web browser or not.

Standard syntax

This Nirva standard command syntax is used in all Nirva client connectors (including Nirva nvc library), in the Nirva procedures or in the Nirva external services.

The command is a succession of pairs `ParameterName=”ParameterValue”`. The pair separator is the blank character. The parameter name is case insensitive. The ‘=’ character must follow the parameter name. The parameter value must be enclosed in double or single quote characters. If the parameter value contains itself a double quote character (or single quote), it must be preceded by another double (or single) quote character. In fact the double quote character can be replaced by any other character. Nirva automatically considers the first character after the equal sign (except blanks) as the character delimiter for the parameter value.

The parameter value can refer to the name of a Nirva variable (server session variable or client request variable). At this time, the parameter value starts with the ‘#’ character. If the parameter value starts with a ‘#’ character but is not to be considered as a variable name, the ‘#’ character must be doubled. In this case and when the command comes from a Nirva client connector, the interpretation of the variable will be made on the server instead of the client side. For example, if the parameter is Param1=”##Value1”, the Nirva client will remove the first ‘#’ character, and will send “#Value1” to the server. The server will then interpret that as the content of the server side variable named “Value1”. Using variables in commands is only available for procedure, service or local client commands. The character recognition for identifying a variable is the ‘#’ character by default but this can be changed at both application level(in application.dsc file for an application

or service.dsc file for a service) and command level (by the way of the NV_VAR_IDENT command parameter. Instead of a single character for identifying a variable, one can use a string (more than one character). At this time, doubling the string has no effect on variable recognition (ex.: if the string for variable recognition is `$%£` and the parameter value is `$%£$%£test`, Nirva will try to find a variable named `$%£test`).

Several parameter values can be specified by using the '+' character as separator. The resulting value will be the concatenation of all the parameter values. For example `Param1="Value1" + "Value2" + "#Value3"`. The resulting parameter value for `Param1` will be the concatenation of `"Value1"`, `"Value2"` and the content of the variable named `"Value3"`.

Here is a Nirva command example:

```
NV_CMD =|LOCAL:OBJECT:CREATE| NAME=|File1| Type=|FILE| FILENAME=|c:\myfile.txt|
```

This simple local service command creates a file object and attaches the file "c:\myfile.txt" to it.

Here is another example with a variable:

```
NV_CMD=|LOCAL:VARIABLE:SET| NAME=|Source file| VALUE=|c:\myfile.txt|
NV_CMD =|LOCAL:OBJECT:CREATE| NAME=|File1| TYPE=|FILE| FILENAME=|#Source file|
```

This local service command does exactly the same than before but using a Nirva variable named "Source file" for storing the file name. In this example, the variable name contains a space character. This is authorized by Nirva.

The parameters can be divided in 2 parts:

- General parameters
- Command specific parameters

The General parameters are documented in this documentation in the "Parameter reference" chapter.

The command specific parameters are documented in Nirva services. This documentation gives the reference of Nirva local and system services. For other services, please refer to the specific service documentations.

Web browser syntax

Syntax

The Web browser syntax is used for commands directly sent from a web browser or from any HTTP client as an URL.

The command must respect the URL syntax. Especially, any un-authorized character must be encoded with its corresponding ASCII code (on 2 digits) preceded by the '%' character. For example, the space character

should be replaced by '%20'. This encoding is generally made directly by the web browser so the user or programmer doesn't have to take care about that.

The URL always starts with the string '`http://`'. This is the name of the used protocol.

Then the URL must give the server name (or address) and it's tcp/ip port. For example: `Myserver:1081` or `127.0.0.1:1081`. The port number to use is defined in the Nirva server configuration.

Then, the string '`/NVS?Command`' must follow. This tells the Nirva server that it's a command for him.

Finally, the command ends with the list of parameters. The parameter syntax is the usual one for URL parameters. Each parameter is composed in this way: `&ParameterName=ParameterValue`. On the contrary of the standard Nirva command syntax, the parameter value is not enclosed in double quotes. The parameter name is case insensitive. There is no space between parameters.

The parameter value can refer to the name of a Nirva variable (server session variable). At this time, the parameter value starts with the '#' character.

Here is an example of a Nirva command sent from a web browser:

```
http://127.0.0.1:1081/NVS?command&NV_CMD=SYSTEM:MISC:NOP&NV_PROC=Proc1
```

This simple command is very often used from browser. It does nothing except calling a Nirva procedure (named 'Proc1' in this example). It directly produces XML output. The SYSTEM:MISC:NOP command is the default command so it can be omitted:

```
http://127.0.0.1:1081/NVS?command&NV_PROC=Proc1
```

Instead of the /NVS?Command string, one can use another url defined in the ORDER_URLS sections of the application.dsc file. This feature also allows predifining some url parameters. See the Application/Description file chapter fo further information about order urls.

Information on HTTP methods

The NIRVA server accepts the HTTP GET and POST methods. With the GET method, NIRVA sends back files that can be viewed in browser (HTML, XML, Images, etc…).

The use of the POST method is reserved for HTML forms. NIRVA recognizes the "`application/x-www-form-urlencoded`" and the "`multipart/form-data`" encoding.

The first one is the standard one. When using it, the couples name/value of the form inputs are automatically added to the command parameters.

For example:

```
<input type="text" name="DESCRIPTION" size="65"/>
```

This HTML form input field will be transmitted to NIRVA as the parameter name "DESCRIPTION".

The second encoding method also adds the couples name/value to the NIRVA command parameters but it allows sending files to the server. For that, the input type must be “file” and the form tag must include the “`enctype="multipart/form-data"`” attribute. When receiving a file in this way, NIRVA creates an object in the application working directory. The name of the object is given by the attribute name of the input form.

For example:

```
<form method="POST" enctype="multipart/form-data" href=”..”>
<input type="file" name="MYFILE" size="65"/>
..
</form>
```

Nirva will then receive the given file and will write it in an object named “myfile” in the input container. The created object is not persistent.

XML connector syntax

The NIRVA XML connector has the same syntax than the web browser syntax except the string ‘`/NVS?Command`’ that is replaced by ‘`/NVS?XMLCommand`’.

However, if the string ‘`/NVS?Command`’ is used and the HTTP order contains a Content-type header for xml, NIRVA automatically considers that the XML connector is used.

The NIRVA command parameters can also be given into the input XML flow instead of the URL.

See the XML connector description for further information.

SOAP connector syntax

The NIRVA SOAP connector has the same syntax than the web browser syntax except the string ‘`/NVS?Command`’ that is replaced by ‘`/NVS?SOAPCommand`’.

However, if the string ‘`/NVS?Command`’ is used and the HTTP order contains a Content-type header for soap (“xml+soap” for example), NIRVA automatically considers that the SOAP connector is used.

The input SOAP message must respect the minimum SOAP standard with a root element named “Envelope” and containing at least the “Body” element.

The NIRVA command parameters can also be given into the input SOAP flow (inside Body) instead of the URL.

See the SOAP connector description for further information.

Web service connector syntax

The Web service syntax is used for commands directly sent from a web service client application.

The Web service command is composed of an URL and an HTTP SoapAction header.

The command must respect the URL syntax. Especially, any un-authorized character must be encoded with its corresponding ASCII code (on 2 digits) preceded by the ‘%’ character. For example, the space character should be replaced by ‘%20’.

The URL has the following format:

```
http://Server:Port/Application/WebService/NVS?WEBSParameters
```

Where *Server* is the nirva server name or address, *Port* is its TCP/IP port, *Application* is the nirva application (can be omitted for default application), *WebService* is the name of the web service to use and *Parameters* are the optional parameters.

The parameter syntax is the usual one for URL parameters. Each parameter is composed in this way: `&ParameterName=ParameterValue`. On the contrary of the standard Nirva command syntax, the parameter value is not enclosed in double quotes. The parameter name is case insensitive. There is no space between parameters.

The parameter value can refer to the name of a Nirva variable (server session variable). At this time, the parameter value starts with the ‘#’ character.

Since the parameters can be given directly in the URL, they can also be given in a dedicated SOAP header or in the SOAP body.

Here is an example of a Nirva URL for a web service:

```
http://127.0.0.1:1081/MyApp/MyWebService/NVS?WEBS
```

The name of the web service operation is generally given in the http SoapAction header but can also be given as the “OPERATION” parameter.

Here is the same example with operation given in the URL:

```
http://127.0.0.1:1081/MyApp/MyWebService/NVS?WEBS&OPERATION=MyOperation
```

In the same way, the name of the web service can be given with the WEBSERVICE parameter instead of the URL path. At this time, the application must also be given as parameter (in the NV_APPLICATION parameter).

Here is the same example with web service name given as parameter:

```
http://127.0.0.1:1081/NVS?WEBS&NV_APPLICATION=MyApp&WEBSERVICE=MyWebServ&OPERATION=MyOp
```

The content of the http body must be a valid SOAP message for the requested web service.

See the Web service connector description for further information.

Standard variables

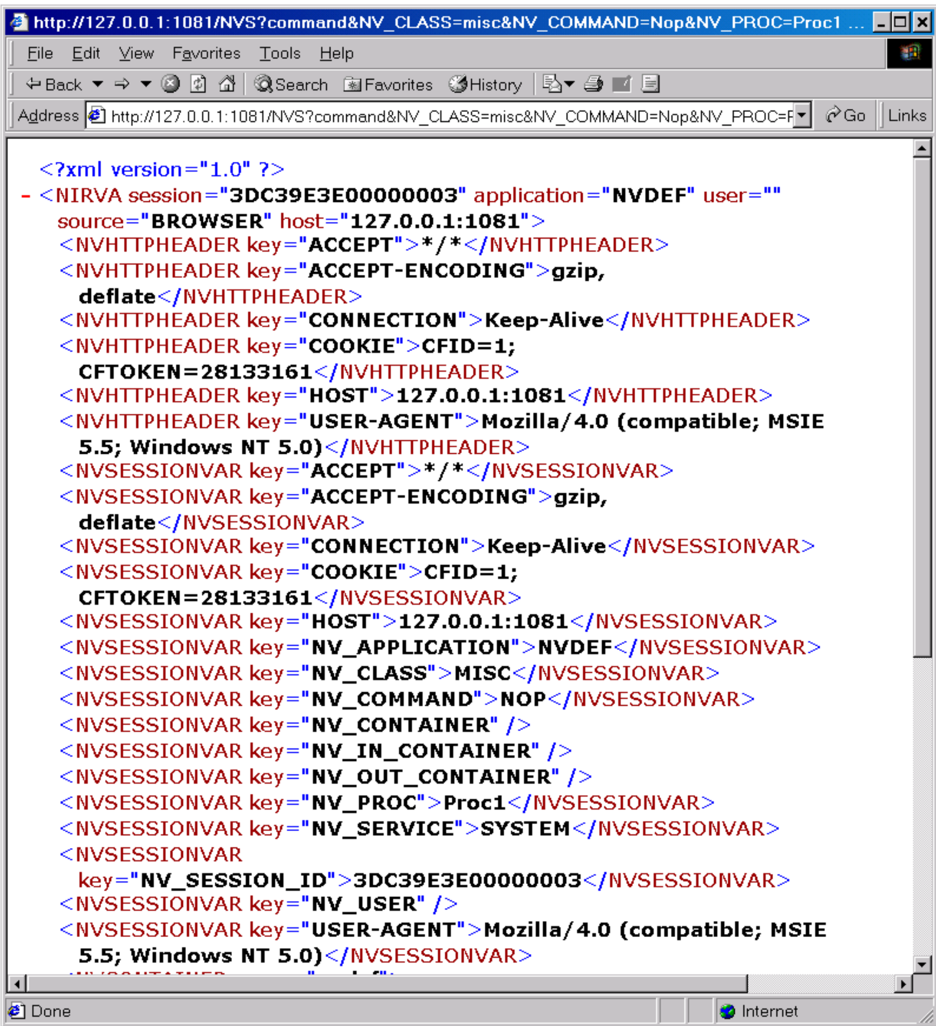
Each time a command is sent to Nirva that comes from a Nirva client or from a web browser, Nirva removes the session variables and creates new ones that can be used in command and procedures. If the command comes from a procedure or from a service, Nirva doesn't create or change these standard variables, giving the procedure or service the chance to work with original command parameters. If the standard parameter NV_KEEP_VAR is set to "YES", Nirva doesn't remove the current session variables.

Here is the list of variables created (or changed) by a command:

- All parameters given in the command except the NV_PASSWORD parameter if it exists.
- All HTTP headers if the command comes from the web.
- NV_NIRVADIR contains the main NIRVA directory (with a '/' or '\' character included at the end).
- NV_HOME_LOGDIR is the base directory where all logs will be written (with a '/' or '\' character included at the end).
- NV_HOME_REGDIR is the base directory containing all the registries (with a '/' or '\' character included at the end).
- NV_HOME_APPWORKDIR is the base directory where all application temporary files are written (with a '/' or '\' character included at the end).
- NV_PLATFORM is the server platform OS. It can take values "WIN32", "WIN64", "AIX", "LINUX", "LINUX64", "HPUX", "HPUXI" or "SOLARIS".
- NV_APPLICATION contains the application name.
- NV_USER is the user name.
- NV_SSO_USER is the SSO (Single sign-on) user name when authentication has been made via SSO (empty otherwise).
- NV_SSO_DOMAIN is the SSO (Single sign-on) domain name when authentication has been made via SSO (empty otherwise).
- NV_SESSION_ID is the current session identifier.
- NV_PARENT_SESSION_ID is the parent session identifier if there is one. There is a parent session when a procedure or a service executes some commands in the context of another session.
- NV_SERVICE is the connected service of original command
- NV_CLASS is the service class of the original command.
- NV_COMMAND is the name of the original command.
- NV_CONTAINER is the name of original input container.
- NV_IN_CONTAINER is the name of original input container.
- NV_OUT_CONTAINER is the name of original output container.
- NV_HOSTNAME is the NIRVA host name.
- NV_UNICODE tells if Nirva has been started in unicode mode or not. Values are YES or NO.
- NV_DEBUG tells if Nirva is in debug mode or not. Values are YES or NO.

- NV_CLIENT_CREATOR_IP is the IP address of the client who has created the session (blank if the session has not been created by a client).
- NV_CLIENT_IP is the IP address of the client who has sent the original command.
- NV_LANGUAGE is the current session language.
- NV_HTTP_HOST is the HTTP host header.

When using the XML features of Nirva, the current session variables are also sent to the XML flow:



Parameter reference

This chapter describes each Nirva command general parameter. The command specific parameters are described in each external service documentation and in this documentation for the Nirva SYSTEM and LOCAL services.

The parameters that are not needed by a specific command will be simply ignored by the Nirva server.

For each parameter, the reference gives the parameter name, the sources for which the parameter have a meaning, the default value if the parameter is omitted and the parameter description.

The available sources are:

- “Client” for all Nirva client connectors including Nirva client library (nvc).
- “Web” for commands from a web browser. This also includes XML, SOAP, Web service and MQ connectors.
- “Procedure” for commands from a Nirva procedure.
- “Service” for commands from service to service

Many of the parameters have a default value allowing sending commands with only few parameters.

Here is a fast summary of the Nirva general parameters:

NV_APPLICATION	Nirva application name to work on.
NV_CLASS	Command Class name.
NV_CLOSE_SESSION	Tells to close the session after the command.
NV_CMD	Nirva command.
NV_COMMAND	Command name.
NV_CONTAINER	Input and output container names.
NV_CONTAINER_CLEAR	Clears both input and output containers before command.
NV_DEBUG_ONLY	Command executed only in debug mode.
NV_DEBUG_PARAM	Display command parameters in debug mode.
NV_ERROR_PROC	Procedure in case of error.
NV_FAST	Fast command parameters decoding.
NV_FORM_ENCODING	Form encoding.
NV_IN_CONTAINER	Input container name.
NV_IN_CONTAINER_CLEAR	Clears input container before command.
NV_KEEP_ERROR	Keep the last error.
NV_KEEP_VAR	Do not remove variable from one command to the other.
NV_LANGUAGE	Language for error description or other messages.
NV_LOCK_SESSION	Controls session locking.
NV_MQ_MSG_HEADER	Send or don’t send the MQ message header.
NV_MQ_OBJNAME	Name of object containing the MQ message.
NV_MQ_OUPTUT	Name of the output queue for sending response.
NV_NEW_IF_EXPIRED	Creates a new session if the requested one is expired.
NV_NEW_PASSWORD	User new password.
NV_NEW_PASSWORD_CONFIRM	User new password confirmation.

NV_NO_CONVERSION	Blocks character set conversion mechanism between Nirva client and server.
NV_NO_ERROR	Do not generate error.
NV_OUT_CONTAINER	Output container name.
NV_OUT_CONTAINER_CLEAR	Clears output container before command.
NV_PARAM	Add command parameters from an object.
NV_PASSWORD	User password.
NV_POST_PROC	Procedures to run after the command.
NV_PROC	Procedures to run before the command.
NV_REQUEST	Allows redirecting the command to a session opened on another NIRVA server.
NV_REVERSE_CONTAINER	Allows reversing input and output container names.
NV_SERVICE	Service name.
NV_SESSION_CLOSE	Name of the procedure to execute when closing the session.
NV_SESSION_ID	Identifier of Nirva session to connect.
NV_SESSION_NAME	Name of session to connect or create.
NV_SESSION_OPEN	Name of the procedure to execute when creating the session.
NV_STOP_ON_ERROR	Stop on error flag.
NV_TIMEOUT	Time out for new Nirva session.
NV_URL_ENCODING	URL encoding.
NV_USER	User name.
NV_VAR	Name of variable for returned value of some SYSTEM and LOCAL Nirva services (output buffer).
NV_XML_CONTAINER	Container to use as XML data flow.
NV_XML_DEBUG	Send containers to debug in debug mode.
NV_XML_ENCODING	Character set to use in the generated XML data.
NV_XML_HTTP_HEADERS	Flag to insert also input HTTP headers into the XML data flow.
NV_XML_NSPREFIX	Defines the namespace prefix for the XML output.
NV_XML_NSREF	Defines the namespace URL for the XML output.
NV_XML_OBJECTS	Container objects to insert into the XML data flow.
NV_XML_OUTPUT	Defines the output type when the command comes from a browser or XML connector.
NV_XML_SESSION_INFO	Adds session information to XMl output.
NV_XML_SIMPLE	XML model to use for output (simple or normal).
NV_XML_STRICT	XML reserved characters strict encoding.

NV_XML_SUBCONTAINERS	Flag to insert also subcontainers into the XML data flow.
NV_XML_VARIABLES	Flag to insert also session variables into the XML data flow.
NV_XML_WITH_DATA	Flag to insert also object data into the XML data flow.
NV_XML_XSL	Name of an xsl file for Nirva to parse XML output data flow into an HTML or other XML page.
NV_XML_XSL_ERROR	Name of an xsl file for Nirva to parse XML error data into an HTML or other XML page.
NV_XML_XSL_IN	Name of an xsl file for Nirva to parse XML input data flow into another XML page.

NV_APPLICATION

Source Web.

Default The default application named “NVDEF”.

Description This is the name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the NV_SESSION_ID is not provided or blank) or when using the XML connector with an input XSL transformation.

In fact, a session is always opened in the context of an application.

This parameter is available only from a web source. From a client source, the application is given when opening a request and is transmitted internally to the server. If the parameter NV_APPLICATION is not given, NIRVA tries to detect it from the URL itself (after the NV_APP_ string). This allows avoiding coding explicitly the application name in html files so the entire application stuff can be independent of the application name. In this way, it’s easier to manage application versions.

NV_CLASS

Source Client, Web (except web service), Procedure, Service.

Default Service name. If this parameter is not provided, NIRVA uses the service name (given by NV_SERVICE) as class name except if the command

comes from the XML or SOAP connectors. If the command comes from the XML or SOAP connectors the default is “MISC” class.

For the web service connector, The NV_CLASS parameter has always the value “WEBSERVICE”.

Description This is the name of the service class. The service commands are divided in classes. In fact, a complete command is identified by the service name, the class name and the command name.

The class name depends of the connected service and is described in the corresponding service reference.

NV_CLOSE_SESSION

Source Client, Web.

Default “NO” except for the XML, SOAP and REST connectors. For XML, SOAP and REST connectors, the default is “NO” if a valid NV_SESSION_ID parameter has been given in the command and “YES” otherwise.

Description If this parameter is set to “YES”, the current session is closed after executing the command. This parameter has no effect if the NV_LOCK_SESSION parameter is set to “NO” because it’s not possible to close a session when in no-lock mode.

This parameter has meaning only when the command is sent from client or browser because it’s not possible to explicitly close a session from a procedure or service.

NV_CMD

Source Client, Web (except web service), Procedure, Service.

Default See description of NV_SERVICE, NV_CLASS and NV_COMMAND parameters for default values.

Description This parameter can be used instead of the NV_SERVICE, NV_CLASS and NV_COMMAND parameters. It has the format SERVICE:CLASS:COMMAND where SERVICE is the service name, CLASS is the class name and COMMAND is the command name. If Service is not given, Nirva considers the service to have the default value. If service is omitted, CLASS can be also omitted to use the default value. If service and class are omitted, CCOMMAND can be also omitted to use the default value.

NV_COMMAND

Source Client, Web (except web service), Procedure, Service.

Default Blank. If the command comes from the XML or SOAP connectors the default is “NOP”.

Description This is the name of the command itself. This is the final string that identifies the command after the service and class names.
The command names are service specific and are described in the service references.
For the web service connector, The NV_CLASS parameter has always the value “EXECUTE”.

NV_CONTAINER

Source Client, Web, Procedure, Service.
This parameter is not used by the LOCAL Nirva client service because this service works only with one local and not hierarchical container.

Default “nvdef”.
If the command comes from a client or web source and this parameter is not provided, Nirva uses the default session container named “nvdef”.
If the command comes from a procedure source and this parameter is not provided, Nirva uses the container name of original web or client command that called the procedure.
If the command comes from a service source and this parameter is not provided, Nirva uses the container name of the service command that sends this new command to another service.
With these default features, the container has only to be defined in the original web or client command.

Description This is the name of the Nirva container that will receive command input and output objects. The container name is case insensitive and cannot contain any of the ‘/’, ‘\’, ‘.’ or space characters.
If the corresponding container doesn’t exist in the session, the server automatically creates it.
Since a container is a hierarchical structure, the access to a subcontainer is made by giving the complete subcontainer path. For example “`Container.sub1`” accesses the subcontainer “`sub1`” of the container “`Container`”.
A container has a session, application, service or system scope. By default, the container has a session scope. In order to access an application container, the container name must be preceded by the character ‘:’ without any other string in front. In order to access a service container, the container name must be preceded by the service name and the character ‘:’ (for

example `Myservice:Myscontainer`). In order to access a system container, the container name must be preceded by the string 'SYSTEM' and the character ':' (for example `SYSTEM:Myscontainer`).
In some situations, a session may have a parent session. This occurs when a session calls a command to be executed by another session. This is the case, for example, when using named sessions. At this time, the called session can access directly the containers of the caller session. For a called session to access its parent containers, the container name must be preceded with the string "[NV_PARENT]:". If the session doesn't have parent, this syntax simply calls the current session container. For example, `[NV_PARENT]:MyContainer` will access a container named MyContainer in the caller session.

NV_CONTAINER_CLEAR

Source Client, Web, Procedure, Service.
This parameter is not used by the LOCAL Nirva client service.

Default "NO".

Description When this parameter is set to "YES", both input and output containers are cleared before the command. This occurs just before the command but after having processed an eventual pre-procedure associated with the command.

NV_DEBUG_ONLY

Source Client, Web, Procedure, Service.

Default "NO".

Description When this parameter is set to "YES", the command will be executed only if Nirva is in debug mode.

NV_DEBUG_PARAM

Source Client, Web, Procedure, Service.

Default "NO".

Description When this parameter is set to "YES", and nirva runs in console and debug mode, Nirva displays the command parameters on the console.

NV_ERROR_PROC

Source	Client, Web, Procedure, Service.
Default	“” (blank).
Description	Name of server procedures to execute when an error occurs. This parameter is not significant for Nirva client LOCAL service. Several procedures can be chained. At this time, they must be separated by a semicolon character (‘;’). They are then executed from left to right order. Please see the Calling a procedure chapter for description of the procedure syntax. The management of error procedures is described in chapter Error management.

NV_FAST

Source	Client, Web, Procedure, Service.
Default	NA.
Description	This parameter allows speeding up the decoding of parameters of the command. It must be used when the command is sure to be well formatted without any unwanted blanks or special characters. This parameter must be the first one of the command. It doesn’t require any value. The systems just checks if it has been given and enters the fast mode in this case. In fast mode, the NV_REQUEST parameter is not decoded and it’s not possible to give the control of the command to another session.

NV_FORM_ENCODING

Source	Web.
Default	“UTF-8”.
Description	This parameter has meaning only when command comes from a web browser and the command is for processing an HTML form. It defines the encoding of the FORM data. The possible values are “ISO-8859-1”, “UTF-8” or “AUTO”. The “AUTO” value allows automatic detection even if the auto detection flag has not been set. If this parameter is not provided the result depends of the auto detection flag. If this last is set, Nirva detects automatically if the inputs are UTF-8 and if not, considers that inputs are ISO-8859-1 encoded. If the auto detection flag is not set, Nirva considers it’s UTF-8.

NV_HREND

Source	Web (not available from the SOAP and web service connectors).
Default	“XSLT”.
Description	Name of the renderer to be used for transforming XML to HTML. A renderer is a special Nirva service that transforms the XML data produced by Nirva into Html.

NV_HREND_PAGE

Source	Web (not available from the SOAP and web service connectors).
Default	“” (blank).
Description	Name of the renderer page. This parameter is used form browser in order to set the name of XML to HTML renderer page processed by the renderer defined in the NV_HREND parameter. If the default renderer is used (XSLT), this parameter can be used instead of the NV_XML_XSL parameter.

NV_IDENT

Source	Web (REST connector only).
Default	“” (blank).
Description	This parameter is used both with NV_TOKEN (public token) in order to identify the user without having to transmit the password. This identification process occurs in 2 steps. The NV_IDENT contains an identification string combining several information as described in the REST connector chapter.

NV_IN_CONTAINER

Source	Client, Web, Procedure, Service. This parameter is not used by the LOCAL Nirva client service because this service works only with one local and not hierarchical container.
Default	“nvdef”. If the command comes from a client or web source and this parameter is not provided, Nirva uses the default session container named “nvdef”. If the command comes from a procedure source and this parameter is not provided, Nirva uses the container name of original web or client command

that called the procedure.
If the command comes from a service source and this parameter is not provided, Nirva uses the container name of the service command that sends this new command to another service.
With these default features, the container has only to be defined in the original web or client command.

Description

This is the name of the Nirva container that will receive command input objects. The container name is case insensitive and cannot contain any of the '/', '\', '.' or space characters.
This parameter has priority to the parameter NV_CONTAINER. It's used when the input and output containers have to be different.
If the corresponding container doesn't exist in the session, the server automatically creates it.
Since a container is a hierarchical structure, the access to a subcontainer is made by giving the complete subcontainer path. For example "`Container.sub1`" accesses the subcontainer "`sub1`" of the container "`Container`".
A container has a session, application, service or system scope. By default, the container has a session scope. In order to access an application container, the container name must be preceded by the character ':' without any other string in front. In order to access a service container, the container name must be preceded by the service name and the character ':' (for example `Myservice:Myscontainer`). In order to access a system container, the container name must be preceded by the string 'SYSTEM' and the character ':' (for example `SYSTEM:Myscontainer`).
In some situations, a session may have a parent session. This occurs when a session calls a command to be executed by another session. This is the case, for example, when using named sessions. At this time, the called session can access directly the containers of the caller session. For a called session to access its parent containers, the container name must be preceded with the string "[NV_PARENT]:". If the session doesn't have parent, this syntax simply calls the current session container. For example, `[NV_PARENT]:MyContainer` will access a container named MyContainer in the caller session.

NV_IN_CONTAINER_CLEAR

Source

Client, Web, Procedure, Service.
This parameter is not used by the LOCAL Nirva client service.

Default

"NO".

Description

When this parameter is set to "YES", the input container is cleared before the command. This occurs just before the command but after having processed an eventual pre-procedure associated with the command.

NV_INPUT_TYPE

Source	Web.
Default	Empty.
Description	This parameter allows storing the content of an http POST,PUT or DELETE request into a file or string object. For that it must be set to "FILE:*object_name*" or "STRING:*object_name*" where *object_name* is the name of the object that receives the content. If only the "FILE" or "STRING" value is given the default object name is respectively "INPUT_FILE" or "INPUT_STRING". This object is created in the input container and cannot be persistent. For a file object, the parameters "EXTENSION", "PREFIX" and "SUFFIX" can be used to control some parts of the file name. The web connector accepts only a FILE type object. The rest connector accepts both FILE and STRING type objects.

NV_KEEP_ERROR

Source	Service, Procedure.
Default	"NO".
Description	This parameter is used only from commands from services or procedures. Each time a service or procedure sends a command, the error management releases the last error. The NV_KEEP_ERROR parameter, when set to "YES" allows keeping the previous error if there is one. If there is no previous error but the current commands produces an error, this error is taken in care. In this way, it's possible to keep only the first error when several errors occur.

NV_KEEP_VAR

Source	Client, Web.
Default	"NO".
Description	When a new session command arrives from a NIRVA client or a web browser, NIRVA removes all the session variables and creates again the standard variables (command parameters, application name, etc…). The NV_KEEP_VAR allows disabling this feature so the old session variables are not removed but just replaced by the new ones having the same name.

NV_LANGUAGE

Source	Client, Web.
Default	“ENGLISH”.
Description	This is the language used for error descriptions. When an error occurs, an error description is given following the required language. This parameter can also be used by any external service or XML application. When the NV_LANGUAGE parameter is given when opening a new session, the language becomes the default session language and it’s not necessary to give the NV_LANGUAGE parameter for other session commands except if the user wants to use another language for his current command.

NV_LOCK_SESSION

Source	Client.
Default	“YES”.
Description	This parameter controls the session locking. A Nirva session can be accessed by different sources at the same time. In fact Nirva creates a thread for each command accessing a session. There is a locking mechanism that avoids processing several commands at the same time for a same session so all the session commands are serialized. It’s possible to disable this locking by setting the NV_LOCK_SESSION to “NO” but this disabling can be dangerous. In fact, Nirva accepts that parameter only if the source of the command is a Nirva client that runs on the same machine and by the same user than the server. This is to authorize a procedure to call an external program (by the SYSTEM MISC EXEC command) that uses the Nirva client library to access the session that called it. In this way, it’s possible to create server executables in any language that work in the context of a Nirva session. This is a good alternative to add external technology or processing to Nirva. This feature can also be used for server scripting with usual scripting languages. If NV_LOCK_SESSION is set to “NO”, and the calling client tries to create a new session (by not giving the session ID), Nirva will automatically set the NV_LOCK_SESSION parameter to “YES” and the client will work with its new session. So the possibility to unlock a session works only when connecting an existing session. In the same way, if a client having disabled the locking tries to remove the connected session, Nirva returns an error message.

NV_MQ_MSG_HEADER

Source	Web (MQ connector only).
Default	“YES”.
Description	This parameter is used only with the MQ connector. Normally, when responding to a MQ REQUEST message, NIRVA answers with a message structure containing some headers (message type and error information). If the NV_MQ_MSG_HEADER parameter is set to “NO”, Nirva only provides the message data.

NV_MQ_OBJNAME

Source	Web (MQ connector only).
Default	“NV_OBJ_DEFAULT_NAME”.
Description	This parameter is used only with the MQ connector when the input message is of FILE or BINARY type. Then, the NV_MQ_OBJNAME gives the name of the file or binary object that will contain the data of the message.

NV_MQ_OUTPUT

Source	Web (MQ connector only).
Default	“”.
Description	This parameter is used only with the MQ connector. It allows changing dynamically the name of the output queue when the input message is of REQUEST type. At this time, Nirva uses the name of the output queue given in the message itself. This queue name can be overridden by the NV_MQ_OUTPUT parameter.

NV_NEW_IF_EXPIRED

Source	Client, Web.
Default	“NO”.
Description	If this parameter is set to “YES” and the requested session has expired or doesn’t exist, the command creates a new session.

NV_NEW_PASSWORD

Source	Web.
Default	No default value. Nirva checks if the parameter has been given or not.
Description	This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.

NV_NEW_PASSWORD_CONFIRM

Source	Web.
Default	No default value. Nirva checks if the parameter has been given or not.
Description	This parameter may be provided when the NV_NEW_PASSWORD parameter has been given. It allows Nirva to check if the new password is correct.

NV_NO_CONVERSION

Source	Client.
Default	“NO”.
Description	This parameter has meaning only when command comes from a NIRVA client. Following the Unicode setting of both client and server, NIRVA translates all input and output data in order to respect the client coding (except file and binary data objects). For example, if NIRVA has been set to no Unicode (ISO-8859-1) and the client to Unicode (UTF-8), then NIRVA translates all incoming data from the client to ISO-8859-1 and all outgoing data to this client to UTF-8. This parameter, when set to “YES” allows punctually blocking this mechanism so no translation occurs.

NV_NO_ERROR

Source	Client, Web, Procedure, Service.
Default	“no”.
Description	This parameter when set to yes tells NIRVA to not generate an error code if the command fails. If the error is critical, the error will be generated whatever

the value of this parameter.
If the NV_STOP_ON_ERROR flag is set, it will stay active even if the NV_NO_ERROR flag is also set.

NV_NO_REDIRECT

Source Web.

Default “no”.

Description This parameter when set to yes tells NIRVA to not use HTTP redirection for the new session when redirection is enabled for the application (see application configuration for information about setting redirection).

NV_OUT_CONTAINER

Source Client, Web, Procedure, Service.
This parameter is not used by the LOCAL Nirva client service because this service works only with one local and not hierarchical container.

Default “nvdef”.
If the command comes from a client or web source and this parameter is not provided, Nirva uses the default session container named “nvdef”.
If the command comes from a procedure source and this parameter is not provided, Nirva uses the container name of original web or client command that called the procedure.
If the command comes from a service source and this parameter is not provided, Nirva uses the container name of the service command that sends this new command to another service.
With these default features, the container has only to be defined in the original web or client command.

Description This is the name of the Nirva container that will receive command output objects. The container name is case insensitive and cannot contain any of the ‘/’, ‘\’, ‘.’ or space characters.
This parameter has priority to the parameter NV_CONTAINER. It’s used when the input and output containers have to be different.
If the corresponding container doesn’t exist in the session, the server automatically creates it.
Since a container is a hierarchical structure, the access to a subcontainer is made by giving the complete subcontainer path. For example “`Container.sub1`” accesses the subcontainer “`sub1`” of the container “`Container`”.
A container has a session, application, service or system scope. By default,

the container has a session scope. In order to access an application container, the container name must be preceded by the character ':' without any other string in front. In order to access a service container, the container name must be preceded by the service name and the character ':' (for example `Myservice:Myscontainer`). In order to access a system container, the container name must be preceded by the string 'SYSTEM' and the character ':' (for example `SYSTEM:Myscontainer`).
In some situations, a session may have a parent session. This occurs when a session calls a command to be executed by another session. This is the case, for example, when using named sessions. At this time, the called session can access directly the containers of the caller session. For a called session to access its parent containers, the container name must be preceded with the string "[NV_PARENT]:". If the session doesn't have parent, this syntax simply calls the current session container. For example, `[NV_PARENT]:MyContainer` will access a container named MyContainer in the caller session.

NV_OUT_CONTAINER_CLEAR

Source Client, Web, Procedure, Service.
This parameter is not used by the LOCAL Nirva client service.

Default "NO".

Description When this parameter is set to "YES", the output container is cleared before the command. This occurs just before the command but after having processed an eventual pre-procedure associated with the command.

NV_OUTPUT_TYPE

Source Rest.

Default Empty.

Description This parameter is used by the REST connector to define the type and name of the object containing the output http data. It must be set to "FILE:*object_name*" or "STRING:*object_name*" where *object_name* is the name of the object containing the output data. If only the "FILE" or "STRING" value is given the default object name is respectively "OUTPUT_FILE" or "OUTPUT_STRING". This object must be in the output container.

NV_PARAM

Source Client, Web, Procedure, Service.

Default “” (blank).

Description This parameter, when given, must point to an indexed string list object of the input container. All the values of the indexed string list object will be added as parameters of the command. If a parameter with the same name already exists, it’s replaced. Some of the Nirva parameters cannot be replaced: NV_CMD, NV_CONTAINER, NV_IN_CONTAINER, NV_OUT_CONTAINER, NV_APPLICATION, NV_COMMAND, NV_CLASS, NV_SERVICE, NV_LANGUAGE, NV_CLOSE_SESSION and all the parameters necessary at login time.
The NV_PARAM parameter is not working with local service.

NV_PASSWORD

Source Web.

Default “” (blank).

Description This is the application user password.
This parameter is significant only when a new session is to be created (so when the NV_SESSION_ID is not provided or blank).
This parameter is available only from a web source. From a client source, the password is given when opening a request and is transmitted internally to the server. The NV_PASSWORD parameter can be also given as a MD5 string of 32 bytes.

NV_POST_PROC

Source Client, Web, Procedure, Service.

Default “” (blank).

Description Name of server procedures to execute after executing the command. This parameter is not significant for Nirva client LOCAL service.
See the description of the NV_PROC parameter for the syntax of the procedure name.

NV_PROC

Source Client, Web, Procedure, Service.

Default “” (blank).

Description Name of server procedures to execute before executing the command. This parameter is not significant for Nirva client LOCAL service.
Several procedures can be chained. At this time, they must be separated by a semicolon character (‘;’). They are then executed from left to right order.
Please see the Calling a procedure chapter for description of the procedure syntax.

NV_REQUEST

Source Client, Web, Procedure, Service.

Default “” (blank).

Description Request Name. This is the name of a connection to another NIRVA server. This parameter allows redirecting the command to another NIRVA server. The connection to the distant NIRVA server must have been previously opened with the NIRVA SYSTEM REQUEST OPEN command.
This command allows establishing the connection parameters. The connection itself is not kept permanently. In fact, NIRVA maintains a pool of TCP/IP connections to other servers and uses one of these connections on demand. The connections have a time out to 200 seconds. If necessary, a reconnection occurs in a transparent way for the user.
If the given NV_REQUEST doesn’t exist, the command fails.

NV_REVERSE_CONTAINER

Source Procedure, Service.
This parameter has meaning only for commands coming from procedures or services.

Default “NO”.

Description When a command comes from a procedure or a service, the default input and output containers (if they are not provided) are the same than the original command. So the original input container becomes the input container of the command and the original output container becomes the output container of the command.
When this parameter is set to “YES”, this reverses the default. So the

original input container becomes the output container of the command and the original output container becomes the input container of the command.

NV_SERVICE

Source Client, Web (except web service), Procedure, Service.

Default “SYSTEM” (system service).

Description This is the name of the service. This can be the name of an internal service (“SYSTEM” for Nirva server internal service or “LOCAL” for Nirva client local service) or the name of an external service that provides third-party technology.
The NV_SERVICE parameter tells Nirva which part of the code will execute the command.
For the web service connector, The NV_SERVICE parameter has always the value “SYSTEM”.

NV_SESSION_CLOSE

Source Client, Web.

Default “session_close”.

Description Name of the procedure that NIRVA calls when closing this session. The default value is “session_close”. In order to not execute any close procedure, this parameter must be set “NV_SESSION_CLOSE_NONE”.
This can be a native, dotnet, java or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name.
This parameter is significant only when a new session is to be created (so when the NV_SESSION_ID is not provided or blank).

NV_SESSION_ID

Source Web, Service, Procedure.

Default “” (blank).

Description This is the Nirva session ID. Its use depends of the source:
From Web source (or web service, XML, SOAP and MQ connectors) if the parameter is given, Nirva tries to connect to an existing session and produces an error if the session doesn’t exist (Except if the NV_NEW_IF_EXPIRED parameter has been set to “yes”).
If the parameter is not given, Nirva creates a new session. From a web

browser, the session ID can be also managed by a cookie. See the SYSTEM:SESSION:USE_ID_COOKIE command for further information.
The newly opened session stays opened until the time out occurs or until it's explicitly closed (by a SYSTEM CLOSE command).
From a client source, this parameter is not used because the session ID is given when opening a request and is transmitted internally to the server.
From a procedure or service, this parameter allows executing the command in the context of another session. This alternate session must exist. It then executes the command and the control is given back to the current session.

NV_SESSION_NAME

Source Service, Procedure, Web

Default “” (blank).

Description This parameter allows executing the command in another session having the given name. Named sessions can be created with the command SYSTEM SESSION CREATE. Several sessions can have the same name.
When the NV_SESSION_NAME parameter is given, Nirva tries to find a session having the same name that is free. If there are some session having the given name but that are in use, Nirva gives the control to the one that has the minimum number of pending requests. If no session having the given name exists, the command fails.
The named sessions can be used for example to maintain a stack of shared database connections accessible by other sessions during the application life.
This parameter can be used from a browser but with restrictions. In fact, one can create and access a named session from a browser only if it starts with the string “test”. This is made for helping application and service developers to create test URLS without having to know the session ID. In order to create or access a named session from web, just don't provide the NV_SESSION_ID parameter and set the NV_SESSION_NAME parameter to a session name starting with the “test” string (for example “test_myservice”).
Be careful to not name a named session with a string starting with “test” from within a procedure or service.

NV_SESSION_OPEN

Source Client, Web.

Default “session_open”.

Description Name of the procedure that NIRVA calls when opening this session. The default value is “session_open”. In order to not execute any open procedure, this parameter must be set “NV_SESSION_OPEN_NONE”.
This can be a native, dotnet, java or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name.
This parameter is significant only when a new session is to be created (so when the NV_SESSION_ID is not provided or blank).

NV_STOP_ON_ERROR

Source Client.

Default “no”.

Description This parameter can be used from a client source in order for the server to eventually close the session in case of error in a command. In fact, the client connectors can queue Nirva commands and send them in a single order. If the Stop on error flag has been set, the server will close the session if one of the queued commands produces an error.

NV_TIMEOUT

Source Web.

Default The default time out value defined in the Nirva server configuration.

Description This parameter gives the time out value in seconds for a session. It’s used only when a new session is to be created (so when the NV_SESSION_ID is not provided or blank).
This parameter is available only from a web source. From a client source, the session time out is given when opening a request and is transmitted internally to the server.
The time out is given in seconds but in fact, NIRVA checks for timeout sessions every 30 seconds.
A time out of 0 closes the session after the command. A time out of –1 sets a time out for the life of the application.

NV_TOKEN

Source Web (REST connector only).

Default “” (blank).

Description	This parameter is used both with NV_IDENT in order to identify the user without having to transmit the password. This identification process occurs in 2 steps. The NV_TOKEN contains a public token previously returnend by the server. See the REST connector chapter for further information.

NV_URL_ENCODING

Source	Web.
Default	"UTF-8".
Description	This parameter has meaning only when command comes from a web browser (or XML, SOAP and Web service connectors). It defines the encoding of the URL. The possible values are "ISO-8859-1", "UTF-8" or "AUTO". The "AUTO" value allows automatic detection even if the auto detection flag has not been set. If this parameter is not provided the result depends of the auto detection flag. If this last is set, Nirva detects automatically if the inputs are UTF-8 and if not, considers that inputs are ISO-8859-1 encoded. If the auto detection flag is not set, Nirva considers it's UTF-8.

NV_USER

Source	Web.
Default	"" (blank).
Description	This is the application user name. This parameter is significant only when a new session is to be created (so when the NV_SESSION_ID is not provided or blank). This parameter is available only from a web source. From a client source, the user is given when opening a request and is transmitted internally to the server.

NV_VAR

Source	Client, Web, Procedure, Service.
Default	"NV_RESULT"
Description	This parameter is used by all the Nirva SYSTEM and LOCAL service commands that return string information in an output buffer. These commands always create a session variable (or request variable for the

LOCAL service) named “NV_RESULT” for storing their result. The variable name can be changed by this parameter.

NV_VAR_IDENT

Source Client, Procedure, Service.

Default “#”

Description This parameter is used by the Nirva SYSTEM and LOCAL service commands to change the character that identifies a variable in a command. The default is the ‘#’ character or the character defined in the application.dsc file if there is one (or service.dsc for commands from service). This parameter, if used, must be given before any other parameter where a variable is to be used. If this parameter is given but is empty, no variable recognition will occur. Instead of a single character, one can use a string.

NV_XML_CONTAINER

Source Client, Web (except web service connector) , Procedure, Service

Default Command output container.

Description This parameter is used for XML output control so it’s normally used from a web browser (or XML or SOAP connectors). It can also be used from inside a procedure or service when Nirva is in debug mode and the NV_XML_DEBUG parameter has been given.
It allows choosing another container than the command output container to use as XML output data.
The given container must exist.
Since a container is a hierarchical structure, the access to a subcontainer is made by giving the complete subcontainer path. For example “`Container.sub1`” accesses the subcontainer “`sub1`” of the container “`Container`”.
A container has a session, application, service or system scope. By default, the container has a session scope. In order to access an application container, the container name must be preceded by the character ‘:’ without any other string in front. In order to access a service container, the container name must be preceded by the service name and the character ‘:’ (for example `Myservice:Myscontainer`). In order to access a system container, the container name must be preceded by the string ‘SYSTEM’ and the character ‘:’ (for example `SYSTEM:Myscontainer`).

NV_XML_DEBUG

Source	Client, Web, Procedure, Service.
Default	None from client, procedure or service. The same name of the xsl file given in the NV_WML_XSL parameter when used from URL (web connector).
Description	This parameter gives the name of a relative xml path file that nirva will use to write the content of the output container (or another one defined by the NV_XML_CONTAINER parameter). This feature works only when Nirva is in debug mode. All debug xml files are written into the Application Files/Debug directory. For example, if NV_XML_DEBUG is set to "mydir1/mydir2/mydebug", Nirva will write the result into a file named "mydebug.xml" in the application Files/Debug/mydir1/mydir2 directory. If subdirectories are given in the path name, Nirva automatically creates them. A debug file viewer is available from the configuration tool.

NV_XML_ENCODING

Source	Web.
Default	For the web connector, the default value is "ISO-8859-1" or "UTF-8" if NIRVA has been launched with the Unicode option. For the XML, SOAP and web service connectors, NIRVA first tries to detect the encoding value from the eventual "charset" option of the HTTP header Content-type. Then it checks the input XML for the "encoding" attribute. If found, this overrides the "charset" value. This new value itself can be overridden by the NV_XML_ENCODING parameter. Finally, if no encoding has been found, NIRVA considers "ISO-8859-1" or "UTF-8" encoding if NIRVA has been launched with the Unicode option.
Description	Name of the character set to use in the generated XML data.

NV_XML_HTTP_HEADERS

Source	Web (except web service connector).
Default	"YES" from browser and "NO" from XML or SOAP connectors.
Description	This parameter is used for XML output control so it's only available from a web browser (or XML connector). It allows to also sending the HTTP headers received in the input HTTP order to the XML flow.

This parameter is not used when the XML model is COMPACT or TINY.
The possible values are “YES” or “NO”.

NV_XML_MODEL

Source Client, Web (except web service connector), Procedure, Service

Default The default value depends of the connector. For Web connector, the default value is “NORMAL”. For SOAP and Web service connectors, the default value is “SIMPLE”. For XML connector, the selection of the model depends of the input. NIRVA tries to detect if input model is simple or normal and adjust the NV_XML_MODEL parameter according to the input model.
For the MQ connector, the default value depends of the input message type. For SOAP, Web service and XML message types, the rule is the same than for the corresponding connectors. For command, binary or file messages, the default value is “SIMPLE”.

Description This parameter tells nirva what XML model to use for output: NORMAL, SIMPLE, COMPACT or TINY. These models are described in the XML chapter in this documentation.
This parameter has no meaning when using the web service connector. At this time, the simple model is automatically selected.
The NV_XML_MODEL parameter must be used instead of the NV_XML_SIMPLE parameter.

NV_XML_NSPREFIX

Source Web (except SOAP and web service connectors).

Default If the command comes from a web browser, the default namespace prefix is empty. If the command comes from the XML connector, the default is the same namespace prefix than the one found in the input XML.

Description This parameter has meaning only if the NV_XML_NSREF parameter is not blank. It defines the prefix of the namespace. If the prefix is empty, the given namespace URL will be the default namespace. If the prefix is given, all the NIRVA XML tags will be namespace qualified.
This parameter has no meaning when used from the SOAP connector.

NV_XML_NSREF

Source Web (except SOAP and web service connectors).

Default If the command comes from a web browser, the default namespace URL is empty (no namespace). If the command comes from the XML connector, the default is the same namespace URL than the one found in the input XML.

Description Name space URL. This is the URL of the namespace to use in the generated output XML. If this parameter is blank, no namespace is used in the output XML. This parameter has no meaning when used from the SOAP connector.

NV_XML_OBJECTS

Source Client, Web (except web service connector), Procedure, Service

Default All objects.

Description This parameter is used for XML output control so it's normally used from a web browser (or XML or SOAP connectors). It can also be used from inside a procedure or service when Nirva is in debug mode and the NV_XML_DEBUG parameter has been given.
It allows specifying only some of the container objects to put in the XML flow.
If used, the parameter should contain the names of valid container objects, separated by semicolon character (';').

NV_XML_OUTPUT

Source Web (except SOAP and web service connectors)

Default If the command comes from a web browser, the default output is HTML. If the command comes from the XML connector, the default is "XML". If the command comes from the SOAP connector, the default is "SOAP" and cannot be changed.

Description Output type for the commands coming from a browser or the XML connector. This can be "HTML", "XML" or "SOAP". This allows NIRVA to properly set the content-type HTTP response header. The output has also influence on the error output.
This parameter has no meaning when used from the SOAP connector because at this time, the output is always SOAP.

NV_XML_SESSION_INFO

Source Client, Web , Procedure, Service.

Default — “YES” from browser and “NO” from XML, SOAP and web service connectors. “YES” from procedure, service or client.

Description — This parameter is used for XML output control so it’s normally used from a web browser (or XML or SOAP connectors). It can also be used from inside a procedure or service when Nirva is in debug mode and the NV_XML_DEBUG parameter has been given.
When it’s set to “YES”, NIRVA delivers additional information in the XML output data about the session.

NV_XML_SIMPLE

Source — Client, Web (except web service connector), Procedure, Service

Default — The default value depends of the connector. For Web connector, the default value is “NO”. For SOAP and Web service connectors, the default value is “YES”. For XML connector, the selection of the model depends of the input. NIRVA tries to detect if input model is simple or not and adjust the NV_XML_SIMPLE parameter according to the input model.
For the MQ connector, the default value depends of the input message type. For SOAP, Web service and XML message types, the rule is the same than for the corresponding connectors. For command, binary or file messages, the default value is “YES”.

Description — This parameter tells nirva what XML model to use for output: normal or simple model. The normal and simple models are described in the XML chapter in this documentation.
This parameter has no meaning when using the web service connector. At this time, the simple model is automatically selected.
The NV_XML_SIMPLE parameter is deprecated but stays for compatibility. One should use the NV_XML_MODEL parameter instead.

NV_XML_STRICT

Source — Client, Web, Procedure, Service.

Default — “NO”.

Description — When transforming container content to XML, Nirva encodes special characters <, >, ‘, “ and & to respectively “<”, “>”, “'”, “"” and “&”. In strict mode (when parameter value is “YES”), if the container data contains a string “<”, this one will be transformed into “<”. If strict mode is not set, the “<” string is unchanged. So if the strict mode is not set, the strings “<”, “>”, “'”, “"” and “&” in the container data are unchanged.

The strict mode also removes all characters less than 0x20 except 0x09, 0x0A and 0x0D.

NV_XML_SUBCONTAINERS

Source Client, Web (except web service connector), Procedure, Service

Default “NO” from browser, “YES” from XML or SOAP connectors. “YES” from procedure, service or client.

Description This parameter is used for XML output control so it’s normally used from a web browser (or XML or SOAP connectors). It can also be used from inside a procedure or service when Nirva is in debug mode and the NV_XML_DEBUG parameter has been given.
It allows also sending the subcontainers of the specified container to the XML flow.
To send subcontainers, the parameter must be set to “YES”.

NV_XML_VARIABLES

Source Client, Web (except web service connector), Procedure, Service

Default “YES” from browser and “NO” from XML or SOAP connectors. “NO” from procedure, service or client (debug only)

Description This parameter is used for XML output control so it’s normally used from a web browser (or XML or SOAP connectors). It can also be used from inside a procedure or service when Nirva is in debug mode and the NV_XML_DEBUG parameter has been given.
It allows also sending the session variables to the XML flow.
The possible values are “YES” or “NO”.

NV_XML_WITH_DATA

Source Client, Web (except web service connector), Procedure, Service.

Default “YES”.

Description This parameter is used for XML output control so it’s normally used from a web browser (or XML or SOAP connectors). It can also be used from inside a procedure or service when Nirva is in debug mode and the NV_XML_DEBUG parameter has been given.
It allows also sending the object data (and not only the object description) of the specified container to the XML flow. It can control also if the NIRVA file

and binary data must be sent.
To not send object data, the parameter must be set to "NO". To send object data but without binary and file data, the parameter must be set to "YES". To send all data including binary and file data, the parameter must be set to "ALL". When NIRVA sends binary or file data, this data is base64 encoded.

NV_XML_XSL

Source Web (not available from the SOAP and web service connectors).

Default "" (blank).

Description This parameter is used for XML output control so it's only available from a web browser (or XML connector). It's not available from the SOAP connector.
If this parameter is provided, it must correspond to an existing server xsl file. The Nirva server will use the XSLT parser defined in the configuration to create an HTML file from the XML data flow.
This is a very powerful feature of Nirva allowing building complex XML applications directly on the Nirva server.
An xsl file can be at application, system or service level.
For an application xsl file, the xsl file name is given as it is. For example: "`file1.xsl`". This must correspond to an existing file of the application file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.
For a system xsl file, the xsl file name must be enclosed in brackets and preceded by the service name. For example: "`(file1.xsl)`". This must correspond to an existing file of the system file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.
For a service xsl file, the xsl file name must be enclosed in brackets. For example: "`MyService(file1.xsl)`". This must correspond to an existing file of the service file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.
If the parameter NV_HREND_PAGE is used, it has priority on the NV_XML_XSL parameter.

NV_XML_XSL_ERROR

Source Web.

Default "" (blank).

Description

This parameter is used for transformation of the XML code generated in case of error. This parameter has priority on the default xsl error file (error.xsl) and on the xsl file eventually defined at session level by the SESSION:SET_DEFAULT_ERROR_XSL command.
If this parameter is provided, it must correspond to an existing server xsl file.
An xsl file can be at application, system or service level.
For an application xsl file, the xsl file name is given as it is. For example: "`file1.xsl`". This must correspond to an existing file of the application file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.
For a system xsl file, the xsl file name must be enclosed in brackets and preceded by the service name. For example: "`(file1.xsl)`". This must correspond to an existing file of the system file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.
For a service xsl file, the xsl file name must be enclosed in brackets. For example: "`MyService(file1.xsl)`". This must correspond to an existing file of the service file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.
This parameter has no meaning when used from the SOAP connector.

NV_XML_XSL_IN

Source

Web (XML connector only).

Default

"" (blank).

Description

This parameter is used for XML input control so it's only available from the XML connector. It's not available from the SOAP connector.
If this parameter is provided, it must correspond to an existing server xsl file.
The Nirva server will use the XSLT parser defined in the configuration to transform the XML input data flow into an XML flow acceptable by NIRVA.
This is a very powerful feature of Nirva allowing building complex XML applications directly on the Nirva server.
An xsl file can be at application, system or service level.
For an application xsl file, the xsl file name is given as it is. For example: "`file1.xsl`". This must correspond to an existing file of the application file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.
For a system xsl file, the xsl file name must be enclosed in brackets and preceded by the service name. For example: "`(file1.xsl)`". This must correspond to an existing file of the system file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.

For a service xsl file, the xsl file name must be enclosed in brackets. For example: "`MyService(file1.xsl)`". This must correspond to an existing file of the service file directory. If there is no extension given, Nirva adds "`.xsl`" after the xsl name. The xsl file name is case insensitive even under UNIX.

Error management

If a NIRVA command fails, it delivers error information. The way the error information is given depends of the command source and the connector used to send this command.

In any case NIRVA delivers the following error information:

CODE	Error code.
SERVICE	Service that produced the error.
CLASS	Error class. The error class can be different than the command class. Each service defines its own error classes.
INFO	Eventual information associated with the error.
DESCRIPTION	Description of the error in the language of the session.

For a nirva application displaying Html code, it's possible to customize the error display. This is done by creating an xsl file named error.xsl in the application File directory.

When such a file is found, Nirva doesn't directly send html error information to the sender but instead creates an XML error output (simple model) and calls the error.xsl file for parsing it.

It is possible to change the default error xsl file by using the SESSION:SET_DEFAULT_XSL_ERROR command or by using the parameter NV_XML_XSL_ERROR.

One can define procedures that will automatically be called when an error occurs. This is done by the way of the global parameter NV_ERROR_PROC and eventually by the command SESSION:SET_DEFAULT_ERROR_PROC.

When an error procedure is defined in a command, it has a scope of the command including all procedures invoked by the command. If in any of the called procedures, a command defines another error procedure, it becomes the current one until another error procedure is defined or until the completion of the command. When this subcommand is completed, the previously defined error procedure is called.

In fact, Nirva maintains a stack of error procedures and only the last of the stack is called in case of error. A new procedure is added when the NV_ERROR_PROC parameter is given and the procedure is removed from the stack when the command that defined the NV_ERROR_PROC parameter terminates. When the stack is empty, nirva uses the default error procedure if one has been defined.

The error procedure receives the error information in the following parameters: NV_ERROR_CODE, NV_ERROR_SERVICE, NV_ERROR_CLASS, NV_ERROR_INFO and NV_ERROR_DESCRIPTION. One

can use the GetParameter function to retrieve it from the procedure (NV::GetParameter from perl procedure and GetParameter from a java or dotnet procedure).

Applications

Overview

A NIRVA application is a context of execution of several components. The application is the place where components are arranged to deliver the final functionality to the application server or to the user interface. The application can be considered as the business part while the services are the technology part.

A Nirva application maintains all persistent server contexts. Particularly, the Nirva application manages users and associated security.

A Nirva session always works in the context of an application.

An application contains its own web sites, registry, procedures, cache and eventual XML files.

In this way, the application data and files are completely independent from each other and a single Nirva server can host several applications.

Managing applications

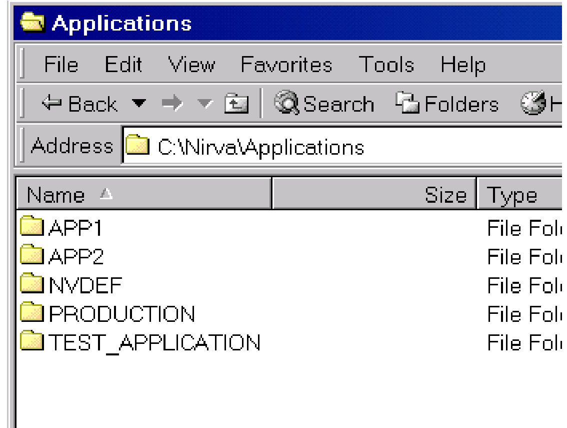
In order to create, start or more generaly manage applications, use the Nirva configuration tool and go to the System/Applications menu. See the application configuration chapter for further information.

Hello world application

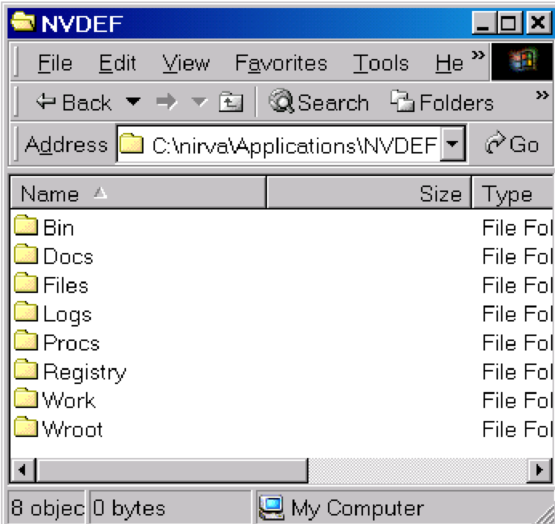
The hello world application example is described in the Getting started chapter.

Application directory

The Application directory is composed of several subdirectories that correspond to the Nirva applications. The names of these subdirectories are directly the application names:



Nirva automatically generates the “NVDEF” application. This is the default application. The default application is used when the user doesn’t require a precise application.



Each application directory contains the following subdirectories:

Bin: Application binary files. By default, Nirva doesn’t create any file into this subdirectory but the application builder can use it to store specific application binary files.

Docs: Application documentations. It includes a subdirectory named “Html” for the html version of the application documentation.

Files: Persistent application files. Typically, this directory should contain some XML or XSLT files necessary for the application. It includes a subdirectory named “Config” that contains the application configuration XSLT files. The Files directory also contains the application.dsc file that describes the application and gives some default parameters and the package.lst file used for packaging and distributing applications (see Installation packages). Finaly the Files directory contains a Test subdirectory for test procedures (see Test sets) and a Framework directory for the application user interface using the nidget framework.

Logs: Application log files. This directory can be modified from the configuration tool.

Procs: Application procedures. It includes a subdirectory named “Config” that contains the application configuration procedures.

Registry: This is the application registry. This subdirectory content is entirely managed by the Nirva dedicated registry functions and should not be accessed from outside the Nirva server. This directory can be modified from the configuration tool.

Work: Application temporary files. All object files created by the application that are not persistent are stored in this subdirectory. This directory is always cleaned up when NIRVA starts. This directory can be modified from the configuration tool.

Wroot: Web sites root directory at application level. It includes a subdirectory named “Config” that contains the application configuration web pages.

Description file

The description file name is “application.dsc”. It can be found in the application Files directory.

The description file is composed of well defined sections. A new section begins with a new line starting with the ‘[‘ character followed by the section name and terminating with the ‘]’ character.

Each section contains a succession of lines with a meaning depending of the section itself.

The description file can include comments. A comment is a line starting with ‘;’, “//” or “\\”.

Any blank line is ignored.

Here are the description file available sections:

INFO General application information.

SETTINGS General application settings.

PERMISSIONS Application security permissions.

ORDER_URLS Predefined URLs for web connector.

Here is an example of a description file for the service “MYAPP”:

```
// myapp.dsc : description file
// NIRVA application
// This file should reside in the NIRVA/Applications/MYAPP/Files directory

// This file contains the MYAPP NIRVA application description
// NIRVA tries to read it when loading the application
```

```
// INFO section
// The INFO section gives some general application information on the form infoname =
info value
// Any new string can be added, removed or modified
[INFO]
APPLICATION = MYAPP
VERSION = 1.00
DESCRIPTION = MYAPP NIRVA application
COMPANY =
COPYRIGHT =

// SETTINGS section
// The SETTINGS section gives some basic application settings
[SETTINGS]
STARTPAGE = default.nvfm
FRAMEWORK = YES
REQUIRED = DATABASE,WORKFLOW
TRIGGER = perl:test

// This section enumerates the MYAPP security permissions on the form permissionname =
permissiondescription
// If the security permissions are not used by the application, this section can be
removed or let empty
[PERMISSIONS]

// This section sets the predefined urls for the application
[ORDER_URLS]
myurl/myoder = NV_PROC=|perl:myorder| PARAM1=|MyParam1| NV_CLOSE_SESSION=|YES|
myurl/myoder2 = NV_PROC=|perl:myorder2|
```

INFO section

The INFO section gives some general application information.

There is a single INFO section in the description file.

The section is composed of several entries of the form *infoname* = *infovalue*. These entries can be retrieved directly by the "SYSTEM:APPLICATION:INFOEX" NIRVA command.

A special entry in the INFO section is checked by NIRVA. This entry is named "ENCODING". If encoding is set to "UTF-8", NIRVA considers that the description file is coded with UTF-8 character set. If encoding is set to "ISO-8859-1", NIRVA considers that the description file is coded with ISO-8859-1 character set (latin1). If there is no entry, Nirva doesn't do any decoding of the values.

SETTINGS section

The SETTINGS section gives some general application information.

There is a single SETTINGS section in the description file.

The section is composed of several entries of the form *paramname* = *paramvalue*.

The *STARTPAGE* entry gives the name of the application start page user interface. This can be a html page residing in the Wroot directory or the name of a view if the application uses the Nidget framework.

The *FRAMEWORK* entry can take values “YES” or “NO”. It defines if the application is using the Nidget framework or not.

The *REQUIRED* entry gives a comma separated list of services required by the application. If a service is required by an application, it ‘s not possible to stop it while the application is running.

The *TRIGGER* entry gives some procedures (semicolun separated list) to be executed each time an external command is sent to Nirva (ex commands from a browser or from a client connector). This can be used for restricting access to some kind of information (for example you can have a trigger procedure that blocks the name of an output container to a given value, avoiding a client to see the content of some application containers). The trigger procedure can be disabled in the application configuration.

The *CHECK_INJECTION* entry enables (YES) or disables (NO) the injection filter. The injection filter verifies that the parameters of the external commands don’t contain themselves some Nirva commands. The default is YES.

The *VARIABLE_IDENT* entry allows to change the default value (# character) for variable identifier in a nirva command. If the parameter is not given, the default value (#) is used. If the parameter is given but is empty, there will be no variable recognition in parameters. One can change the value of the variable identifier at command level using the NV_VAR_IDENT command parameter.

PERMISSIONS section

The PERMISSIONS section enumerates the application security permissions.

There is a single PERMISSIONS section in the description file.

The section is composed of several entries of the form *permissionname = permissiondescription*. These entries are directly used by the security layer of NIRVA. A permission can be checked by a service with dedicated NIRVA commands.

The permission names should use only alphanumeric characters (numbers and letters) and the underscore character.

ORDER_URLS section

The ORDER_URLS section allows setting some predefined URLs for the web connector.

The section is composed of several entries of the form *url = nirvaparameters*.

url is the last part of the real url sent form browser. The real url must of the form:

```
protocol//server:port/nv_order/appname/url
```

where protocol is the protocol (“http:” or “https”), server:port is the server and the tcp/ip port (ex “localhost:1081”), appname is the application name and url is the url defined in the ORDER_URLS section.

For example if you have defined the following url in the MYAPP application.dsc file:

```
myurl/myoder = NV_PROC=|perl:myorder| PARAM1=|MyParam1| NV_CLOSE_SESSION=|YES|
```

Then you can call it by entering the following url in your browser:

```
http://locahost:1081/nv_order/myapp/myurl/myorder
```

If you want you can also add some extra parameters in this url:

```
http://locahost:1081/nv_order/myapp/myurl/myorder&myparam=test
```

The Nirva parameters given in the ORDER_URLS section have priority against the parameters given in the real url.

In order to have more friendly urls, you can use Nirva http alliases by replacing /nv_order/appname with something else.

REST_URLS section

The REST_URLS section allows setting the URLs for the rest connector.

The section is composed of several entries of the form *url* = *nirvaparameters*.

url is the last part of the real url sent form browser. The real url must be of the form:

```
protocol//server:port/nv_rest/appname/url
```

where protocol is the protocol (“http:” or “https”), server:port is the server and the tcp/ip port (ex “localhost:1081”), appname is the application name and url is the url defined in the ORDER_URLS section.

For example if you have defined the following url in the MYAPP application.dsc file:

```
myurl/myrestcommand = NV_PROC=|perl:myrestcommand| PARAM1=|MyParam1|
```

Then you can call it by the following url:

```
http://locahost:1081/nv_rest/myapp/myurl/myrestcommand
```

If you want you can also add some extra parameters in this url:

```
http://locahost:1081/nv_rest/myapp/myurl/myrestcommand&myparam=test
```

The url parameter can end with the wildcard '*' character. At this time any url sent that starts by the defined url will execute the rest order.

The Nirva parameters given in the REST_URLS section have priority against the parameters given in the real url.

In order to have more friendly urls, you can use Nirva http alliases by replacing /nv_rest/appname with something else.

Procedures

Overview

The NIRVA procedures are simple script files allowing doing some processing in the context of the NIRVA session.

There are procedures at system, application, service and web service levels. By default, a procedure is considered to be at application level. The procedure files must reside in a dedicated procedure directory (Proc directory). The system but also each service and application has its own procedure directory.

A procedure contains commands that can call other procedures. In this way, a single command can execute a complex hierarchy of other Nirva commands.

A procedure accepts specific parameters.

NIRVA accepts 4 kinds of procedure: Native, Perl, Dotnet and Java procedures.

The native procedures are simply a succession of NIRVA commands.

Perl procedures are complete Perl script with some dedicated callback functions to work with NIRVA. NIRVA provides an embedded Perl interpreter avoiding to install it externally (except if some Perl extension modules have to be used).

Java procedures are java classes with some dedicated callback functions to work with NIRVA. NIRVA provides an embedded Java virtual machine avoiding installing it externally (except if some java extension classes have to be used).

Dotnet procedures are dotnet classes with some dedicated callback functions to work with NIRVA. The dotnet procedures are available only on windows platform.

Calling a procedure

A procedure can be called from any Nirva command by giving the procedure name in a dedicated command parameter (NV_PROC and NV_POST_PROC).

The NV_PROC parameter gives the procedures to execute before the command and the NV_POST_PROC parameter gives the procedures to execute after the command.

Several procedures can be chained. At this time, they must be separated by a semicolon character (';'). They are then executed from left to right order.

Scope

NIRVA manages procedures at application, system, service or web service levels.

NIRVA knows which level to use by checking the procedure name:

Application procedure

For an application procedure, the procedure name is given as it is. For example: "`proc1.prc`". The procedure name must correspond to an existing file of the application procedure directory. If there is no extension given, Nirva adds "`.nvp`" after the procedure name (or "`.pl`" for a Perl script and "`.dll`" for a Dotnet procedure) except if it's a java class. If the procedure is stored in a subdirectory of the application procedure directory, its name must give the complete relative path, for example "`Test/proc1.nvp`". The procedure name is case insensitive even under UNIX.

System procedure

For a system procedure, the procedure name must be enclosed in brackets. For example: "`(proc1.prc)`". The procedure name must correspond to an existing file of the system procedure directory. If there is no extension given, Nirva adds "`.nvp`" after the procedure name (or "`.pl`" for a Perl script) except if it's a java class. If the procedure is stored in a subdirectory of the system procedure directory, its name must give the complete relative path, for example "`(Test/proc1.nvp)`". The procedure name is case insensitive even under UNIX.

Service procedure

For a service procedure, the procedure name must be enclosed in brackets and preceded by the service name. For example: "`MyService(proc1.prc)`". The procedure name must correspond to an existing file of the service procedure directory. If there is no extension given, Nirva adds "`.nvp`" after the procedure name (or "`.pl`" for a Perl script and "`.dll`" for a Dotnet procedure) except if it's a java class. If the procedure is stored in a subdirectory of the service procedure directory, its name must give the complete relative path, for example "`MyService(Test/proc1.nvp)`". The procedure name is case insensitive even under UNIX.

Web service procedure

For a web service procedure, the procedure name must be enclosed in brackets and preceded by the web service name itself enclosed in {} characters. For example: "`{MyWebService}(proc1.prc)`". The procedure name must correspond to an existing file of the web service procedure directory. If there is no extension given, Nirva adds "`.nvp`" after the procedure name (or "`.pl`" for a Perl script and "`.dll`" for a Dotnet procedure) except if it's a java class. If the procedure is stored in a subdirectory of the web service

procedure directory, its name must give the complete relative path, for example "{MyWebService}(Test/proc1.nvp)". The procedure name is case insensitive even under UNIX.

Programming language

Nirva procedures can be written in native, perl, dotnet or java language.

The native language is not really a language but just a file containing NIRVA commands. This is useful when there is no complex programming to do.

Native procedures

By default, the procedure is a native NIRVA procedure. For calling a native procedure, just set its name in the dedicated parameters (NV_PROC for example).

Please see native procedures chapter for further information.

Perl procedures

In order to execute a Perl procedure, the string "`perl:`" must precede the procedure name and the procedure must be a Perl script.

Please see the Perl procedures chapter for further information.

Java procedures

In order to execute a Java class, the string "`java:`" must precede the procedure name and the procedure must be a java class. It's also possible to specify the name of the java method to execute (the default is "main"). For that, the method name must follow the procedure name separated by a ':' character. For example "`java:test:mymethod`" will call the application java class "`test` " from the procedure directory and the method "`mymethod`" of this class.

When calling a java procedure, NIRVA tries to localize the corresponding class file directly. For example when calling the "`java:test`" procedure, NIRVA tries to find a file named "test.class" in the application procedure directory.

NIRVA can find procedures in subdirectories but also in jar files. For example for the "`java:mydir/test`" procedure, NIRVA first tries to find a subdirectory named "mydir" of the application procedures directory and gives this information to the java virtual machine for it to find the test procedure from this subdirectory. If the subdirectory is not found, NIRVA tries to find a file named mydir.jar in the application procedures directory. If found, it tells the jvm to find the class from this file.

NIRVA can also work with packaged classes. For example, if the class is named "myclass" in the "com.mycompany.mypackage" package itself contained in the mycompany.jar file in the application procedures directory, the name for calling the procedure will be "`java:mycompany/com.mycompany.mypackage.myclass`".

Nirva implements a class cache for java procedures. It can be enabled or not (see the chapter configuration system parameters). It is advised to set the class cache when using a sun jvm because some sun jvm versions has a bug in memory management that is hidden when the cache is set. This also saves time when loading procedures. The cache can be disabled during development step in order to easily change the java code without having to restart an application.

Environment class loader: As an option, an environment class loader can be defined for java procedures at system, application, service or web service level. This class loader is then used as a parent class loader for each procedure loaded in the corresponding environment. This improves performance (avoids loading some jars at each call of a procedure) and allows sharing global jar files for the entire environment. The global jars must be in a subdirectory named JavaLib in the Procs directory.

Dotnet procedures

Dotnet procedures are available only on windows platform.

In order to execute a Dotnet procedure, the string “`dotnet:`” must precede the procedure name and the procedure must be an assembly filename (a corresponding .dll file must exist). It’s also possible to specify the name of the dotnet class to execute (the default is the same than the assembly name). For that, the class name must follow the procedure name separated by a ‘:’ character. For example “`dotnet:test:myclass`” will call the application dotnet class “`myclass`“ contained in the assembly file “`test.dll`“ in the procedure directory. The class name may also have a namespace (ex.: “`dotnet:test:mynamespace.myclass`”). The namespace must then be separated by a dot character from the class name.

By default the class name is the assembly name and there is no namespace. So the procedure “`dotnet:myclass`” will call the `myclass` procedure that is in the `myclass.dll` file.

Procedure parameters

The procedure name can have specific parameters. They are then enclosed into bracket characters. The parameters are pairs name=’value’ where value is enclosed in simple quotes. If the parameter value has to contain a single quote, it must be doubled. Here is an example of procedure name with parameters: `MyService(perl:proc1[par1=’test1’ par2=’test2’])`. The procedure parameters have a procedure scope. A procedure parameter value can refer to a session variable, for that, it must start with the ‘#’ character. When a command calls a procedure, the specific procedure parameters are added to the command that runs the procedure and removed after. One must be careful if a command parameter has the same name than a procedure parameter. At this time, both parameters are the same.

Examples

Here is an example of a web command that calls a native procedure:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=ntest
```

This command does nothing (command SYSTEM:MISC:NOP) but calls a native procedure file named “ntest.nvp” that resides in the procedure application directory.

Here is an example of a web command that calls a Perl procedure:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=perl:ptest
```

This command does nothing (command SYSTEM:MISC:NOP) but calls a Perl procedure file named “ptest.pl” that resides in the procedure application directory.

Here is an example of a web command that calls a Java procedure:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=java:jtest
```

This command does nothing (command SYSTEM:MISC:NOP) but calls the main method of a Java class procedure file named “jtest.class” in the procedure application directory.

Here is an example of a web command that calls a Dotnet procedure:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=dotnet:dtest
```

This command does nothing (command SYSTEM:MISC:NOP) but calls the Main method of a Dotnet procedure class named “dtest” that resides in the file named “dtest.dll” in the procedure application directory.

Native procedures

The NIRVA native procedure is a succession of NIRVA commands that will be executed one after the other.

A line starting with the semicolon character (‘;’) is considered as a comment line.

A line cannot exceed 2048 characters.

A command cannot be split on 2 lines.

The procedure stops at the end or as soon as an error is encountered in one of its commands.

A native procedure can retrieve the command or procedure parameters by using the SYSTEM COMMAND GET_PARAMETER command.

Please see the “Calling a procedure” chapter for information about accessing native procedures.

Here is an example of procedure file:

```
; NIRVA procedure 1 for tests
; 03/05/2002

NV_CMD=|OBJECT:CREATE| NAME=|Obj1| TYPE=|FILE| FILENAME=|file.txt| PERSIST=|-1|
NV_CMD=|OBJECT:FILE_CREATE| NAME=|Obj1|
NV_CMD=|OBJECT:CREATE| NAME=|Obj2| TYPE=|INDSTRINGLIST|
NV_CMD=|OBJECT:INDSTRINGLIST_SET_VALUE| NAME=|Obj2| KEY=|Key1| VALUE=|VALUE1|
NV_CMD=|OBJECT:INDSTRINGLIST_SET_VALUE| NAME=|Obj2| KEY=|Key2| VALUE=|VALUE2|
NV_CMD=|OBJECT:INDSTRINGLIST_SET_VALUE| NAME=|Obj2| KEY=|Key3| VALUE=|VALUE3|
NV_CMD=|OBJECT:INDSTRINGLIST_SET_VALUE| NAME=|Obj2| KEY=|Key4| VALUE=|VALUE4|
NV_CMD=|OBJECT:INDSTRINGLIST_SET_VALUE| NAME=|Obj2| KEY=|Empty| VALUE=||
NV_CMD=|OBJECT:CREATE| NV_CONTAINER=|cont1| NAME=|Obj3| TYPE=|STRINGLIST|
NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NV_CONTAINER=|cont1| NAME=|Obj3| VALUE=|azerty|
NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NV_CONTAINER=|cont1| NAME=|Obj3| VALUE=|azerty1|
NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NV_CONTAINER=|cont1| NAME=|Obj3| VALUE=|azerty2|
NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NV_CONTAINER=|cont1| NAME=|Obj3| VALUE=|azerty3|
NV_CMD=|OBJECT:CREATE| NV_CONTAINER=|cont1| NAME=|obj2| TYPE=|STRINGLIST|
NV_CMD=|OBJECT:CREATE| NV_CONTAINER=|cont1| NAME=|obj3| TYPE=|STRINGLIST|
NV_CMD=|OBJECT:CREATE| NV_CONTAINER=|cont1.sub1| NAME=|obj4| TYPE=|STRINGLIST|
NV_CMD=|OBJECT:CREATE| NV_CONTAINER=|cont1.sub1| NAME=|obj5| TYPE=|STRINGLIST|
```

Here is the way to call it from a web browser:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=ntest
```

The procedure must be named ntest.nvp and must reside in the NVDEF application procedure directory.

Perl procedures

Description

Perl procedures are Perl scripts executed by the NIRVA embedded Perl interpreter. This embedded Perl interpreter has been built based on the 5.10 Perl version.

Any signal management has been removed from the embedded Perl interpreter because all signal management is directly under the control of the NIRVA kernel.

Any output sent to the console will have effect only when NIRVA is running in console mode.

NIRVA provides some callback functions allowing a Perl procedure to call other NIRVA commands in the context of the session. These commands can themselves contain some calls to other Perl, Java, Dotnet or Native NIRVA procedures.

Perl procedures can be encrypted when packaging Nirva components (see the installation packages chapter). This allows hiding the application code to users.

Please see the “Calling a procedure” chapter for information about accessing perl procedures.

Here is an example of Perl NIRVA procedure:

```
# Uncomment next line to give error control to the script
#NV::SetErrorMode("Script");

# Executes a SYSTEM:MISC:NOP NIRVA command
NV::Command("NV_CMD=|MISC:NOP|");

# Get and print the NIRVA session ID
NV::GetSessionId();
print "Session ID = $NV::RESULT\n";
```

This example should be run with NIRVA running in console mode in order to see the screen output.

Here is the way to call it from a web browser:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=perl:ptest
```

The Perl script must be named ptest.pl and must reside in the NVDEF application procedure directory. It's provided at installation time. If you want to see the print output, the NIRVA server must have been started in console mode.

The Perl script can call the perl "`exit()`" function for exiting. At this time, if the exit value is different that 0, NIRVA will generate an error message. By default, the script leaves with the exit value 0.

When the Nirva debug mode is set (system parameters), the perl may display some warning messages in the console (if nirva runs in console mode). When not in debug mode the standard perl I/O is closed (STDIN, STDERR and STDOUT) so any perl print message will not be displayed.

Reference

This chapter gives the reference of all NIRVA callback functions that can be used from a NIRVA Perl script in order to work with NIRVA.

All the functions must be called from the Perl script by adding the string "NV::" before the function. For example, the callback function "`GetSessionId()`" must be called by typing "`NV::GetSessionId()`".

Some of the functions require some parameters. These parameters are described for each function.

Most of the callback functions return an integer that is '1' in case of success and '0' in case of failure. All of them also set the value of a Perl variable named "`NV::RESULT`" that contains the result string of the function. This variable can then be accessed like any Perl variable by adding the '$' character before ("`$NV::RESULT`"). If the function doesn't return a string value, the NV::RESULT variable is reset to an empty string.

NV::SetErrorMode

Syntax

NV::SetErrorMode(ErrorMode)

Description

This function sets the error control mode for the rest of the script file. The error checking can be on the control of NIRVA or on the control of the Perl script.

When it's on the NIRVA control, any command sent from the Perl script to NIRVA (by using the NV::Command Perl function) that produces an error will immediately stop the script and NIRVA affects the error code to the original command that called the script.

When the error checking is on the script control, the script itself must check the return code of the NV::Command function (1 if success and 0 if failure). It's then its responsibility to stop or continue the script. It can then set himself error information by using the `NV::SetError` or `NV::SetErrorEx` function. In case of error, the script can be leaved with the perl `exit` function by giving a value different of 0. For example `exit(-1)` will leave the pearl script telling NIRVA to take care about the error.

Parameters

ErrorMode	Can take the values 'SCRIPT' or 'NIRVA'. If it's set to 'SCRIPT', then the error control is given to the script. The default value is 'NIRVA'.

Return value

Always 1.

Return string (NV::RESULT)

Empty.

NV::SetError

Syntax

NV::SetError(ErrorInfo)

Description

This function generates a SYSTEM:PROCEDURE:105 error code (Error in Perl script) with an error information given as parameter. It doesn't stop the Perl script.

Parameters

ErrorInfo	Error information that NIRVA will use as error information for the original command.

Return value

Always 1.

Return string (NV::RESULT)

Empty.

NV::SetErrorEx

Syntax

NV::SetErrorEx(ErrorService, ErrorClass, ErrorCode, ErrorInfo)

Description

This function generates the given error code. It doesn't stop the Perl script. It can be used for setting specific application or service error codes. The error service, error class and error code must exist on the service description file.

Parameters

ErrorService	Error service.
ErrorClass	Error class.
ErrorCode	Error code. If set to –1, the function retrieves the last error and set it at the current one.
ErrorInfo	Error extended information.

Return value

Always 1.

Return string (NV::RESULT)

Empty.

NV::Command

Syntax

NV::Command(CommandString)

Description

This function sends a command to Nirva. This is the way for the script to communicate to NIRVA but also to the external services.

Parameters

CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command. The double quote characters of the command string must be preceded by a backslash character (\").

Return value

1 if successful and 0 otherwise.

Return string (NV::RESULT)

Depends of the NIRVA command itself. See the NIRVA SYSTEM reference in order to see if the command returns something in the output buffer.

NV::GetSessionId

Syntax

NV::GetSessionId()

Description

This function retrieves the NIRVA session identifier of the session which has started the script.

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

NIRVA Session identifier.

NV::GetClass

Syntax

NV::GetClass()

Description

This function retrieves the NIRVA command class name for the command which has started the script.

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

NIRVA command class name.

NV::GetService

Syntax

NV::GetService()

Description

This function retrieves the NIRVA command service name for the command which has started the script.

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

NIRVA command service name.

NV::GetCommand

Syntax

NV::GetCommand()

Description

This function retrieves the NIRVA command name for the command which has started the script.

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

NIRVA command name.

NV::GetInContainer

Syntax

NV::GetInContainer()

Description

This function retrieves the NIRVA input container name for the command which has started the script.

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

NIRVA input container name.

NV::GetOutContainer

Syntax

NV::GetOutContainer()

Description

This function retrieves the NIRVA output container name for the command which has started the script.

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

NIRVA output container name.

NV::GetLanguage

Syntax

NV::GetLanguage()

Description

This function retrieves the eventual language passed as parameter in the command for the command which has started the script.

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

Language.

NV::GetError

Syntax

NV::GetError(InfoType)

Description

This function retrieves the current command error information. It's useful for the programmer to check an error after a NV::Command function call (when the error is under the control of the script).

Parameters

InfoType	Kind of error information to return. This must be one of the strings "CODE", "INFO", "SERVICE", "CLASS", "DESCRIPTION". For Nirva versions older than 2.5.024, "DESC" must be used instead "DESCRIPTION". After version 2.5.024 both "DESC" and "DESCRIPTION" work. The default is "CODE" that returns the error code.

Return value

Always 1.

Return string (NV::RESULT)

Requested error information.

NV::GetNumParameters

Syntax

NV::GetNumParameters()

Description

This function returns the number of parameters of the command which has started the script. It can be used in conjunction with the NV::GetParameterName function to enumerate the parameters.

Parameters

None.

Return value

Number of command parameters.

Return string (NV::RESULT)

None.

NV::GetSource

Syntax

NV::GetSource()

Description

This function returns the source of the command which has started the script. It can be the following ones:

- CLIENT
- BROWSER
- SERVICE
- PROCEDURE
- SERVER

Parameters

None.

Return value

Always 1.

Return string (NV::RESULT)

Source of the command. It will be one of the strings “CLIENT”, “BROWSER”, “SERVICE”, “PROCEDURE” or “SERVER”.

NV::ParameterExist

Syntax

NV::ParameterExist(ParameterName)

Description

This function checks if the given parameter exists in the command parameters or not.

Parameters

ParameterName Name of the parameter to check. The parameter name is case insensitive.

Return value

1 if the parameter exist and 0 otherwise.

Return string (NV::RESULT)

None.

NV::GetParameter

Syntax

NV::GetParameter(ParameterName)

Description

This function retrieves the value of a command parameter if it exists.

Parameters

ParameterName Name of the parameter to get. The parameter name is case insensitive.

Return value

Always 1.

Return string (NV::RESULT)

Parameter value. If the parameter doesn't exist, the result string is empty but the function doesn't generate any error.

NV::GetParameterName

Syntax

NV::GetParameterName(Index)

Description

This function retrieves the name of a command parameter. It can be used in conjunction with the NV::GetNumParameters function to enumerate the parameters.

Parameters

Index	Index of the parameter. This index starts at 1 and cannot be greater than the number of parameters.

Return value

Always 1.

Return string (NV::RESULT)

Parameter name.

NV::SetEncoding

Syntax

NV::SetEncoding(Charset)

Description

This function tells Nirva what encoding has been used in the perl script. It allows Nirva to encode/decode all data flow between the Nirva kernel and the perl procedure. This should be used if the perl procedure contains other characters than the ASCII ones.

If used, this should be the first command of the perl script.

The encoding/decoding may decrease performances. So this should be used only if necessary.

Parameters

Charset	Encoding of the perl script. This can be set to “UTF-8”, “ISO-8859-1” or “NONE”. The default is none (no encoding).

Return value

Always 1.

Return string (NV::RESULT)

None.

Java procedures

Description

Java procedures are Java methods executed by the NIRVA embedded virtual machine.

Any output sent to the console will have effect only when NIRVA is running in console mode.

When invocating a Java procedure, the command gives the name of the java class and optionally the name of the method of this class to execute. If this last is not provided, NIRVA tries to use “main” or “Main” as method name. See the chapter ‘Calling a procedure” for further information.

The Java class must implement this method. The prototype of the method is the following:

```
public [static] [final] int methodname()
```

Where `methodname` is the method name. The return value must be 0 if successful and different of 0 otherwise.

The Java source code must be compiled and the resulting class file must reside in a procedure directory (system, application or service procedure directory). It can be packaged into a jar file.

Please see the “Calling a procedure” chapter for information about accessing java procedures.

The embedded Java VM is using the CLASSPATH environment variable to locate external classes but also implements an internal class loader allowing to directly loading the NIRVA procedure classes. In this way, the procedure directories don’t have to be declared in the CLASSPATH environment variable.

If a nirva procedure is included into a package, the java class must import the com.nirvasoft.nirva.nvsint class. This class is in the nirva.jar file delivered in the nirva bin directory.

NIRVA provides some callback methods allowing a Java procedure to call other NIRVA commands in the context of the session. These commands can themselves contain some calls to other Java, Perl or Native NIRVA procedures.

The callback functions are all in a special class named “`nvsint`”.

Here is an example of Java NIRVA procedure class:

```
import com.nirvasoft.nirva.*;

class jtest
{
  public static final int main()
  {
          System.out.print("Java test interface\n");
          nvsint MyNirva = new nvsint();
          if(MyNirva == null)
          {
                  System.out.print("Error getting nvinst class\n");
                  return -1;
```

```
            }

            MyNirva.Command("NV_CMD=|MISC:NOP|");
            System.out.print("Session ID = ");
            System.out.print(MyNirva.GetSessionId());
            System.out.print("\n");
            return 0;
    }
  }
```

This example should be run with NIRVA running in console mode in order to see the screen output.

Here is the way to call it from a web browser:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=java:jtest
```

The class must have been compiled with javac and the resulting file named "jtest.class" must reside in the NVDEF application procedure directory. It's provided at installation time. If you want to see the print output, the NIRVA server must have been started in console mode.

Reference

This chapter gives the reference of all NIRVA callback methods that can be used from a NIRVA Java class procedure in order to work with NIRVA.

In fact NIRVA provides a special class named "`nvsint`" that implement all these methods. In order to communicate with NIRVA, the java code must create a "`nvsint`" object.

Some of the methods require some parameters. These parameters are described for each method.

The "`Command`" method is used to send commands to NIRVA. Some of these NIRVA commands return a string object that is written in the `nvsint` object `NvResult` attribute. The `NvResult` attribute is declared like this:

```
  public java.lang.String NvResult;
```

If the command doesn't return a string value, the `NvResult` attribute is reset to an empty string.

SetError

Syntax

void SetError(String ErrorInfo)

Description

This method generates a SYSTEM:PROCEDURE:106 error code (Error in Java method) with an error information given as parameter. It doesn't stop the Java method.

Parameters

ErrorInfo	Error information that NIRVA will use as error information for the original command.

Return value

None.

SetErrorEx

Syntax

void SetErrorEx(String ErrorService, String ErrorClass, int ErrorCode, String ErrorInfo)

Description

This method generates the given error code. It can be used for setting specific application or service error codes. The error service, error class and error code must exist on the service description file.

Parameters

ErrorService	Error service.
ErrorClass	Error class.
ErrorCode	Error code. If set to –1, the function retrieves the last error and set it at the current one.
ErrorInfo	Error extended information.

Return value

None.

Command

Syntax

int Command(String CommandString)

Description

This method sends a command to Nirva. This is the way for the java method to communicate to NIRVA but also to the external services.

The `Command` method returns 1 in case of success and 0 in case of failure. In case of failure, NIRVA directly sets the error code and information for the original command but doesn't stop the java method so the checking of errors is under the responsibility of the java programmer.

The “`nvsint`” class provides some methods to get the last error code.

Some of the NIRVA commands return a string object that is written in the `nvsint` object `NvResult` attribute. The `NvResult` attribute is declared like this:

```
public java.lang.String NvResult;
```

If the command doesn't return a string value, the `NvResult` attribute is reset to an empty string. See the NIRVA SYSTEM reference in order to see if the command returns something in the output buffer.

Parameters

CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command. The double quote characters of the command string must be preceded by a backslash character (\").

Return value

1 if successful and 0 otherwise.

GetResult

Syntax

String GetResult()

Description

This method retrieves the eventual result string of the last command sent to NIRVA with the “`Command`” method.

When the “Command” method is executed, it sets the value of the `NvResult` attribute but also keeps internally the eventual result string.

The programmer can then use directly the NvResult attribute or use the GetResult method in order to retrieve the last command result. The only difference between the two methods is that GetResult is returning the result of the last successful command to Nirva while NvResult returns the result of the last command (empty if the last command has returned an error).

Parameters

None.

Return value

Last command result.

GetSessionId

Syntax

String GetSessionId()

Description

This method retrieves the NIRVA session identifier of the session which has started the java procedure.

Parameters

None.

Return value

NIRVA Session identifier.

GetService

Syntax

String GetService()

Description

This method retrieves the NIRVA command service name for the command which has started the java procedure.

Parameters

None.

Return value

NIRVA command service name.

GetClass

Syntax

String GetClass()

Description

This method retrieves the NIRVA command class name for the command which has started the java procedure.

Parameters

None.

Return value

NIRVA command class name.

GetCommand

Syntax

String GetCommand()

Description

This method retrieves the NIRVA command name for the command which has started the java procedure.

Parameters

None.

Return value

NIRVA command name.

GetInContainer

Syntax

String GetInContainer()

Description

This method retrieves the NIRVA input container name for the command which has started the java procedure.

Parameters

None.

Return value

NIRVA input container name.

GetOutContainer

Syntax

String GetOutContainer()

Description

This method retrieves the NIRVA output container name for the command which has started the java procedure.

Parameters

None.

Return value

NIRVA output container name.

GetLanguage

Syntax

String GetLanguage()

Description

This method retrieves the eventual language passed as parameter in the command for the command which has started the java procedure.

Parameters

None.

Return value

Language.

GetError

Syntax

String GetError(String InfoType)

Description

This method retrieves the current command error information. It's useful for the programmer to check an error after a "`Command`" method call.

Parameters

InfoType	Kind of error information to return. This must be one of the strings "CODE", "INFO", "SERVICE", "CLASS", "DESCRIPTION". For Nirva versions older than 2.5.024, "DESC" must be used instead "DESCRIPTION". After version 2.5.024 both "DESC" and "DESCRIPTION" work. The default is "CODE" that returns the error code.

Return value

Requested error information.

GetNumParameters

Syntax

int GetNumParameters()

Description

This method returns the number of parameters of the command which has started the java procedure. It can be used in conjunction with the "`GetParameterName`" method to enumerate the parameters.

Parameters

None.

Return value

Number of command parameters.

GetSource

Syntax

String GetSource()

Description

This method returns the source of the command which has started the java procedure. It can be the following ones:

- CLIENT
- BROWSER
- SERVICE
- PROCEDURE
- SERVER

Parameters

None.

Return value

Source of the command. It will be one of the strings “CLIENT”, “BROWSER”, “SERVICE”, “PROCEDURE” or “SERVER”.

ParameterExist

Syntax

Int ParameterExist(String ParameterName)

Description

This method checks if the given parameter exists in the command parameters or not.

Parameters

ParameterName	Name of the parameter to check. The parameter name is case insensitive.

Return value

1 if the parameter exist and 0 otherwise.

GetParameter

Syntax

String GetParameter(String ParameterName)

Description

This method retrieves the value of a command parameter if it exists.

Parameters

ParameterName	Name of the parameter to get. The parameter name is case insensitive.

Return value

Parameter value. If the parameter doesn’t exist, the returned string is empty but the function doesn’t generate any error.

GetParameterName

Syntax

String GetParameterName(int Index)

Description

This method retrieves the name of a command parameter. It can be used in conjunction with the “`GetNumParameters`” method to enumerate the parameters.

Parameters

Index	Index of the parameter. This index starts at 1 and cannot be greater than the number of parameters.

Return value

Parameter name.

Dotnet procedures

Description

Dotnet procedures are Dotnet classes executed by the dotnet common language runtime (CLR). The Dotnet procedures are available only on a windows environment. The dotnet environment (3.5 minimum) must have been installed on the target machine.

Any output sent to the console will have effect only when NIRVA is running in console mode.

When invocating a Dotnet procedure, the command gives the name of the dotnet assemble and optionally the name of the class to execute. If this last is not provided, NIRVA uses the assembly name also as a class name (without the dll extension). See the chapter ‘Calling a procedure” for further information.

The Dotnet class must override the Main method of the ProcedureEntryPoint class. The skeleton of a dotnet procedure is the following:

```
using System;
using Nirva;

public class classname : ProcedureEntryPoint
{
    public override int Main(nvsint Command)
    {
        // Procedure code
        return 0;
    }
}
```

Where `classname` is the class name. The return value must be 0 if successful and different of 0 otherwise.

The source code must be compiled and the resulting assembly file must reside in a procedure directory (system, application or service procedure directory.

Please see the “Calling a procedure” chapter for information about accessing dotnet procedures.

Nirva maintains a specific dotnet application domain for Nirva applications, services and web service. These domains are created the first time and application, service or web service is using dotnet and is released when the corresponding component is stopped. Developers should stop the component in order to be able to use a new version of procedure assemblies.

NIRVA provides some callback methods allowing a Dotnet procedure to call other NIRVA commands in the context of the session. These commands can themselves contain some calls to other Java, Dotnet, Perl or Native NIRVA procedures.

The callback functions are all in a special class named “`nvsint`”.

Here is an example of Dotnet procedure class (written in c#):

```
using System;
using Nirva;

public class dtest : ProcedureEntryPoint
{
    public override int Main(nvsint Command)
    {
        Console.WriteLine("Dotnet test interface");
        Console.Write("Session ID = ");
        Console.WriteLine(Command.GetSessionId());
        Command.Command("NV_CMD=|object:create| type=|string| name=|mystring|
                         value=|myvalue| replace=|yes|");
        Command.Command("NV_CMD=|object:string_get_value| name=|mystring|");
        Console.Write("mystring value = ");
        Console.WriteLine(Command.GetResult());

        return 0;
    }
}
```

The Nirva namespace, the ProcedureEntryPoint and nvsint classes are defined in the nirvadn.dll assembly found on nirva bin directory.

This example should be run with NIRVA running in console mode in order to see the screen output.

Here is the way to call it from a web browser:

```
http://127.0.0.1:1081/Config/NVS?command&NV_PROC=dotnet:dtest
```

The class must have been compiled with a c# compiler and the resulting file named “dtest;dll” must reside in the NVDEF application procedure directory. It’s provided at installation time. If you want to see the print output, the NIRVA server must have been started in console mode.

Here is an example of compilation of the procedure using csc compiler:

```
csc /target:library /r:c:/Nirva/Bin/nirvadn.dll dtest.cs
```

Reference

This chapter gives the reference of all NIRVA callback methods that can be used from a NIRVA Dotnet class procedure in order to work with NIRVA.

In fact NIRVA provides a special class named “`nvsint`” that implement all these methods. In order to communicate with NIRVA, the dotnet code of the Main method receives a reference to a “`nvsint`” object. This object contains the context of the command.

Some of the methods require some parameters. These parameters are described for each method.

The “`Command`” method is used to send commands to NIRVA. Some of these NIRVA commands return a String object (output buffer) that is kept by the `nvsint` object. This object data can be retrieved by using the nvsint.GetResult() method.

If the command doesn’t return a string value, the last result is reset to an empty string.

SetError

Syntax

void SetError(String ErrorInfo)

Description

This method generates a SYSTEM:PROCEDURE:107 error code (Error in Dotnet method) with an error information given as parameter. It doesn’t stop the Dotnet method.

Parameters

ErrorInfo	Error information that NIRVA will use as error information for the original command.

Return value

None.

SetErrorEx

Syntax

void SetErrorEx(String ErrorService, String ErrorClass, int ErrorCode, String ErrorInfo)

Description

This method generates the given error code. It can be used for setting specific application or service error codes. The error service, error class and error code must exist on the service description file.

Parameters

ErrorService	Error service.
ErrorClass	Error class.
ErrorCode	Error code. If set to –1, the function retrieves the last error and set it at the current one.

ErrorInfo Error extended information.

Return value

None.

Command

Syntax

int Command(String CommandString)

Description

This method sends a command to Nirva. This is the way for the dotnet method to communicate to NIRVA but also to the external services.

The `Command` method returns 1 in case of success and 0 in case of failure. In case of failure, NIRVA directly sets the error code and information for the original command but doesn't stop the dotnet method so the checking of errors is under the responsibility of the dotnet programmer.

The “`nvsint`” class provides some methods to get the last error code.

Some of the NIRVA commands return a string object (output buffer) that can be retrieved using the `nvsint.GetResult()` method.

If the command doesn't return a string value, the last result is reset to an empty string. See the NIRVA SYSTEM reference in order to see if the command returns something in the output buffer.

Parameters

CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command.

Return value

1 if successful and 0 otherwise.

GetResult

Syntax

String GetResult()

Description

This method retrieves the eventual result string of the last command sent to NIRVA with the "`Command`" method.

When the "Command" method is executed, the nvsint object keeps the eventual result string.

The programmer can then use the GetResult method in order to retrieve the last command result.

Parameters

None.

Return value

Last command result.

GetSessionId

Syntax

String GetSessionId()

Description

This method retrieves the NIRVA session identifier of the session which has started the dotnet procedure.

Parameters

None.

Return value

NIRVA Session identifier.

GetService

Syntax

String GetService()

Description

This method retrieves the NIRVA command service name for the command which has started the dotnet procedure.

Parameters

None.

Return value

NIRVA command service name.

GetClass

Syntax

String GetClass()

Description

This method retrieves the NIRVA command class name for the command which has started the dotnet procedure.

Parameters

None.

Return value

NIRVA command class name.

GetCommand

Syntax

String GetCommand()

Description

This method retrieves the NIRVA command name for the command which has started the dotnet procedure.

Parameters

None.

Return value

NIRVA command name.

GetInContainer

Syntax

String GetInContainer()

Description

This method retrieves the NIRVA input container name for the command which has started the dotnet procedure.

Parameters

None.

Return value

NIRVA input container name.

GetOutContainer

Syntax

String GetOutContainer()

Description

This method retrieves the NIRVA output container name for the command which has started the dotnet procedure.

Parameters

None.

Return value

NIRVA output container name.

GetLanguage

Syntax

String GetLanguage()

Description

This method retrieves the eventual language passed as parameter in the command for the command which has started the dotnet procedure.

Parameters

None.

Return value

Language.

GetError

Syntax

String GetError(String InfoType)

Description

This method retrieves the current command error information. It's useful for the programmer to check an error after a "`Command`" method call.

Parameters

InfoType	Kind of error information to return. This must be one of the strings "CODE", "INFO", "SERVICE", "CLASS", "DESCRIPTION". The default is "CODE" that returns the error code.

Return value

Requested error information.

GetNumParameters

Syntax

int GetNumParameters()

Description

This method returns the number of parameters of the command which has started the dotnet procedure. It can be used in conjunction with the "`GetParameterName`" method to enumerate the parameters.

Parameters

None.

Return value

Number of command parameters.

GetSource

Syntax

String GetSource()

Description

This method returns the source of the command which has started the dotnet procedure. It can be the following ones:

- CLIENT
- BROWSER
- SERVICE
- PROCEDURE
- SERVER

Parameters

None.

Return value

Source of the command. It will be one of the strings “CLIENT”, “BROWSER”, “SERVICE”, “PROCEDURE” or “SERVER”.

ParameterExist

Syntax

bool ParameterExist(String ParameterName)

Description

This method checks if the given parameter exists in the command parameters or not.

Parameters

ParameterName	Name of the parameter to check. The parameter name is case insensitive.

Return value

true if the parameter exists and false otherwise.

GetParameter

Syntax

String GetParameter(String ParameterName)

Description

This method retrieves the value of a command parameter if it exists.

Parameters

ParameterName	Name of the parameter to get. The parameter name is case insensitive.

Return value

Parameter value. If the parameter doesn't exist, the returned string is empty but the function doesn't generate any error.

GetParameterName

Syntax

String GetParameterName(int Index)

Description

This method retrieves the name of a command parameter. It can be used in conjunction with the "`GetNumParameters`" method to enumerate the parameters.

Parameters

Index	Index of the parameter. This index starts at 1 and cannot be greater than the number of parameters.

Return value

Parameter name.

Connectors

This chapter describes the available NIRVA client connectors.

One of the major NIRVA goals is to provide a much opened architecture. In this way, NIRVA delivers several client connectors allowing to access Nirva application and services from any kind of language in a very simple way.

Since all the session management and context is done on the NIRVA server side, it’s possible to access the same NIRVA session from different client connectors. For example, a PHP client can open a session that is then accessible from a JAVA client. This feature also allows using client scripting languages without having to install them as extension modules. For example, in an APACHE / PHP client, the PHP doesn’t need to be installed as an APACHE extension (but it can also). The only necessary NIRVA link between different PHP pages is the NIRVA session ID.

Much of the connectors are based on the NIRVA nvc library that must be accessible by the connector. The nvc library is delivered for WINDOWS, LINUX, AIX, HPUX and SOLARIS platforms in the Nirva/Sdk/Connectors/dll directory. The nvc library must be copied somewhere on the disk where it’s viewable by the system dynamic loading feature (for example in the system32 directory for WINDOWS or in /usr/lib for UNIX). On the local NIRVA server machine, the library is also installed in the Nirva/Bin directory.

The name of the nvc dynamic library is nvc.dll for WINDOWS, libnvc.a for AIX, libnvc.sl for HPUX PA-RISC and libnvc.so for LINUX, SOLARIS and HPUX Itanium.

The nvc library and the NIRVA client connectors can be distributed without any restriction with client applications.

Dll - C and C++

Overview

The NIRVA dll client connector is available on WINDOWS, LINUX, AIX, HPUX and SOLARIS platforms.

This connector is directly the nvc NIRVA client library.

Please refer to the ‘Client library nvc” chapter for a complete description of this connector.

Installation

The dll connector is delivered for WINDOWS, LINUX, AIX, HPUX and SOLARIS platforms in the Nirva/Sdk/Connectors/dll directory. The dll connector consists of the nvc library that must be copied somewhere on the disk where it's viewable by the system dynamic loading feature (for example in the system32 directory for WINDOWS or in /usr/lib for UNIX). On the local NIRVA server machine, the library is also installed in the Nirva/Bin directory.

The name of the nvc dynamic library is nvc.dll for WINDOWS, libnvc.a for AIX, libnvc.sl for HPUX PA-RISC and libnvc.so for LINUX, SOLARIS and HPUX Itanium.

Reference

Please refer to the 'Client library nvc" chapter

Example

Here is a little C program that opens a NIRVA session on the local NIRVA server and sends a NOP command.

```
// nvtest.cpp : Nirva example.

#include <stdio.h>
#include <string.h>

#include "nvc.h"        // Nirva nvc header

// Global variables
char OutputBuffer[2048];


// Entry point
int main(int argc, char* argv[])
{
  NVREQUEST Request;

  // Create the NIRVA request object
  NVREQUEST Request = NvOpenRequest("");
   if(Request == NULL)
   {
      printf("\n\nError opening request\n");
      return -1;
   }

   // Send a NOP command to NIRVA (this also opens the NIRVA session)
   NvCommand(Request, "NV_CMD=|MISC:NOP|", NULL, 0, NULL);
```

```
  // Get the session ID
  NvCommand(Request, "NV_CMD=|LOCAL:REQUEST:GET_SESSION_ID|", OutputBuffer,
            sizeof(OutputBuffer), NULL);
  printf("Session ID is %s", OutputBuffer);


  // Close the NIRVA session
  NvCommand(Request, "NV_CMD=|SESSION:CLOSE|, NULL, 0, NULL);


  return 0;
} // End main
```

For other example, please refer to the ‘Client library nvc” chapter.

ActiveX

Overview

The NIRVA ActiveX client connector consists of a simple ActiveX control. It’s available only on WINDOWS platforms.

The ActiveX control is named “nvcx”.

Installation

The ActiveX connector is delivered in the Nirva/Sdk/Connectors/activex directory.

The file name is “nvcx.ocx”.

The class name is “NVCX.NvcxCtrl.1”.

The CLSID is “{8296D45D-FE97-4CFF-92EA-6DABD0804E64}”.

The file may be copied at any place but it’s necessary to register it in order for WINDOWS to know where to find it.

In order to register the control, just run the command “`RegSvr32` *`controlpath`*” where *`controlpath`* is the complete path of the nvc.ocx file. For example, if the nvc.ocx file is installed in the c:\Winnt directory, just type:

```
“RegSvr32 c:\Winnt\nvcx.ocx”.
```

The ActiveX connector also requires the dll connector to be installed on the target machine. Please see the chapter about the DLL connector for information about the installation.

Reference

This chapter gives the reference of the NIRVA methods that can be used from any program that can call an ActiveX control in order to work with NIRVA.

The calling program must create an instance of the nvcx control.

The “`Command`” method is used to send commands to NIRVA. Some of these NIRVA commands (the ones using output buffer) return a string object that is written in the `NvResult` property. The `NvResult` property is of BSTR type.

If the command doesn’t return a string value, the `NvResult` property is reset to an empty string.

The control provides only 3 methods:

- *OpenRequest* opens the request.
- *CloseRequest* closes the previously opened request.
- *Command* sends commands to Nirva.

OpenRequest

Syntax

short OpenRequest(BSTR ConnectString)

Description

This function opens a new client request. The parameters of the request are given in the ConnectString parameter. The function returns 1 if successful and 0 otherwise.

If a previous request was opened, this one is first closed.

Parameters

ConnectString

Connection string. This string contains the request parameters. This is a succession of pairs `ParameterName=”ParameterValue”`. The pair separator is the blank character. The parameter name is case insensitive. The ‘=’ character must follow the parameter name. The parameter value must be enclosed in double quote characters. If the parameter value contains itself a double quote character, it must be preceded by another double quote character.
Here are the available connection string parameters:

- *Server* is the Nirva server TCP/IP address or machine name. This can be follow by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is installed. For example `Server=”135.12.13.14:2035”` is a valid

Server address. The default value for this parameter is `"127.0.0.1:1081"`.
It’s possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers. This allows a simple load balancing solution without requiring a load balancing box. This is a simple load balancing solution and the connected servers must all run.
Example: `Server="192.168.20.17;192.168.20.19:80"` will connect one of the servers 192.168.20.18 or 192.168.20.18 on the port 80.
A proxy server can also be defined in a single address. At this time the format is the following:
`proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be “http” or “https”; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

- *ConnectionTimeOut* is the time out for establishing the TCP/IP connection to the Nirva server. The default value is 10 seconds.
- *Session* is the Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn’t exist. If the parameter is not given, Nirva creates a new session. The newly opened session stays opened until the time out occurs or until it’s explicitly closed (by a SYSTEM CLOSE command) or after sending a command (or the command buffer) if the auto close mode has been set.
- *AutoClose* is the auto close mode. If this parameter is set to “YES”, the session is automatically closed after sending a command or the content of the command buffer to the server. Otherwise, the session stays open until it’s explicitly closed by a SYSTEM CLOSE command or when the time out occurs.
- *SessionTimeOut* is the time out value in seconds for a session. It’s used only when a new session is to be created (so when the *Session* parameter is not provided or blank). If *SessionTimeOut* is “0”, the default Nirva time out will be used. This is the default.
- *Application* is the name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In fact, a session is always opened in the context of an application. If the

application parameter is not provided, Nirva uses the default application named “NVDEF”.

- *User* is the application user name. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *Password* is the application user password. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *NewPassword* is the application new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
- *NewPasswordConfirm*. This parameter may be provided when the *NewPassword* parameter has been given. It allows Nirva to check if the new password is correct.
- *Open* is the name of the NIRVA procedure to call when opening the new session. The default is “session_open”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”.
- *Close* is the name of the NIRVA procedure to call when closing the new session. The default is “session_close”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”.
- *Ssl* is the SSL mode. If this parameter is set to “YES”, the request will use the HTTPS protocol to communicate to the NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the NIRVA server.
- *Certificate* is the optional client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.
- *MaxMsgSize* is the maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise

between performance and memory management on the client side. The minimum value is 10 kilobytes.

- *TempDir* is the directory name where the connector can write the temporary files. If this parameter is not provided, NIRVA uses the current directory.
- *Unicode* tells if client is Unicode. When this parameter is set to “YES”, all the data sent to the server are supposed to be UTF-8 encoded and all data coming from server are automatically converted to UTF-8 by the server. When not in Unicode mode, the client is supposed to work in ISO-8859-1 character set.
- *Sso* is the SSO (Single sign-on) flag. If this parameter is set to “YES” and if the Nirva application has been configured to authenticate users via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO.
- *SsoPrincipal* is the SSO (Single sign-on) principal name when using Kerberos protocol. This parameter has no meaning when not using SSO or when using SSO with NTLM.
- *SocketLog* is the name of an optional log file at socket (http) level.

Return value

1 if successful and 0 otherwise.

Return string (NvResult)

None.

CloseRequest

Syntax

void CloseRequest()

Description

This function closes a request previously opened with the *OpenRequest* function.

It frees any resource used by the request. This function is automatically called from the control destructor.

Parameters

None.

Return value

None.

Return string (NvResult)

None.

Command

Syntax

short Command(BSTR CommandString)

Description

This function sends a command to Nirva.

Parameters

CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command.

Return value

1 if successful and 0 otherwise.

Return string (NvResult)

Depends of the NIRVA command itself. See the NIRVA SYSTEM or services references in order to see if the command returns something in the output buffer.

Example

Here is a little WINWORD macro written in Visual Basic that opens a NIRVA session on the local NIRVA server and sends a NOP command.

```
Sub Nirva()
'
' Nirva Macro
' Test the connection to NIRVA local server
'

' Create the instance to the activeX control
Dim Nirva As Object
Set Nirva = CreateObject("NVCX.NvcxCtrl.1")
```

```
' Open the nirva request
Nirva.OpenRequest ("")

' Send a nop command to the server
Nirva.Command ("NV_CMD=|MISC:NOP|")

' Request the session ID
Nirva.Command ("NV_CMD=|LOCAL:REQUEST:GET_SESSION_ID|")

' Display it
MsgBox ("Nirva session ID = " + Nirva.NvResult)

' And finally close the session
Nirva.Command ("NV_CMD=|SESSION:CLOSE|")

End Sub
```

Php

Overview

The NIRVA Php client connector is built as a Php extension module. It directly uses the nvc library that must be installed on the target machine.

Installation

The Php connector is delivered in the Nirva/Sdk/Connectors/php directory as a source file. It must be compiled as a php extension module named nvcphp.dll (or .so or other dynamic library extension following the platform). Please consult the php documentation for further information about extension modules.

Each version of php requires a recompilation of the extension module. Nirva systems may provide compiled modules for some versions of php but not for all. Please ask us for that. If not available you must compile it yourself.

Once compiled, the Php connector consists of a single file that is the Php extension library. Under windows, this file is named nvcphp.dll. Under LINUX and SOLARIS the file is named “nvcphp.so” for the single thread version (to be used with the CGI version of PHP) and “nvcphpts.so” for the multithread version that must be used with the Php server versions (for apache for example).

The Php connector also requires the dll connector to be installed on the target machine. Please see the chapter about the DLL connector for information about the installation.

The module must be listed in the extensions section of the php.ini file.

Reference

This chapter gives the reference of the NIRVA functions that can be used from a Php script in order to work with NIRVA.

Some of the functions require some parameters. These parameters are described for each function.

Some of the functions return a boolean value TRUE in case of success and FALSE in case of failure. All of them also set the value of a Php variable named "NVC_RESULT" that contains the result string of the function. This variable can then be accessed like any Php variable by adding the '$' character before ("$NVC_RESULT"). If the function doesn't return a string value, the NVC_RESULT variable is reset to an empty string.

The Nirva client Php connector provides only 3 functions:

- *nvc_openrequest* creates a new request and returns a request handle.
- *nvc_closerequest* closes a previously opened request.
- *nvc_command* sends commands to Nirva in the context of a request.

---

nvc_openrequest

Syntax

bool nvc_openrequest(string ConnectString)

Description

This function opens a new client request. The parameters of the request are given in the ConnectString parameter. The function returns a handle that is used with other functions to communicate with the request.

Parameters

ConnectString	Connection string. This string contains the request parameters. This is a succession of pairs `ParameterName="ParameterValue"`. The pair separator is the blank character. The parameter name is case insensitive. The '=' character must follow the parameter name. The parameter value must be enclosed in double quote characters. If the parameter value contains itself a double quote character, it must be preceded by another double quote character. The double quote characters of the connection string must be preceded by a backslash character (\"). Here are the available connection string parameters: ■ *Server* is the Nirva server TCP/IP address or machine name. This can be follow by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is installed. For example `Server="135.12.13.14:2035"` is a valid

Server address. The default value for this parameter is `"127.0.0.1:1081"`.
It's possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers. This allows a simple load balancing solution without requiring a load balancing box. This is a simple load balancing solution and the connected servers must all run. Example: `Server="192.168.20.17;192.168.20.19:80"` will connect one of the servers 192.168.20.18 or 192.168.20.18 on the port 80.
A proxy server can also be defined in a single address. At this time the format is the following:
`proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be "http" or "https"; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

- *ConnectionTimeOut* is the time out for establishing the TCP/IP connection to the Nirva server. The default value is 10 seconds.
- *Session* is the Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn't exist. If the parameter is not given, Nirva creates a new session. The newly opened session stays opened until the time out occurs or until it's explicitly closed (by a SYSTEM CLOSE command) or after sending a command (or the command buffer) if the auto close mode has been set.
- *AutoClose* is the auto close mode. If this parameter is set to "YES", the session is automatically closed after sending a command or the content of the command buffer to the server. Otherwise, the session stays open until it's explicitly closed by a SYSTEM CLOSE command or when the time out occurs.
- *SessionTimeOut* is the time out value in seconds for a session. It's used only when a new session is to be created (so when the *Session* parameter is not provided or blank). If *SessionTimeOut* is "0", the default Nirva time out will be used. This is the default.
- *Application* is the name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In fact, a session is always opened in the context of an application. If the

application parameter is not provided, Nirva uses the default application named “NVDEF”.

- *User* is the application user name. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *Password* is the application user password. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *NewPassword* is the application new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
- *NewPasswordConfirm*. This parameter may be provided when the *NewPassword* parameter has been given. It allows Nirva to check if the new password is correct.
- *Open* is the name of the NIRVA procedure to call when opening the new session. The default is “session_open”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”.
- *Close* is the name of the NIRVA procedure to call when closing the new session. The default is “session_close”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”.
- *Ssl* is the SSL mode. If this parameter is set to “YES”, the request will use the HTTPS protocol to communicate to the NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the NIRVA server.
- *Certificate* is the optional client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.
- *MaxMsgSize* is the maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise

between performance and memory management on the client side. The minimum value is 10 kilobytes.

- *TempDir* is the directory name where the connector can write the temporary files. If this parameter is not provided, NIRVA uses the current directory
- *Unicode* tells if client is Unicode. When this parameter is set to “YES”, all the data sent to the server are supposed to be UTF-8 encoded and all data coming from server are automatically converted to UTF-8 by the server. When not in Unicode mode, the client is supposed to work in ISO-8859-1 character set.
- *Sso* is the SSO (Single sign-on) flag. If this parameter is set to “YES” and if the Nirva application has been configured to authenticate users via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO.
- *SsoPrincipal* is the SSO (Single sign-on) principal name when using Kerberos protocol. This parameter has no meaning when not using SSO or when using SSO with NTLM.
- *SocketLog* is the name of an optional log file at socket (http) level.

Return value

TRUE if successful and FALSE otherwise.

Return string (NVC_RESULT)

The returned string is a handle to the NIRVA request. It must be used as first parameter to the other functions.

---

nvc_closerequest

Syntax

bool nvc_closerequest(string Request)

Description

This function closes a request previously opened with the *nvc_openrequest* function.

It frees any resource used by the request.

Parameters

Request	Handle of the request as returned by the *nvc_openrequest* function.

Return value

TRUE if successful and FALSE otherwise.

Return string (NVC::RESULT)

None.

---

nvc_command

Syntax

bool nvc_command(string Request, string CommandString)

Description

This function sends a command to Nirva.

Parameters

Request	Handle of the request as returned by the *nvc_openrequest* function.
CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command. The double quote characters of the command string must be preceded by a backslash character (\").

Return value

TRUE if successful and FALSE otherwise.

Return string (NVC_RESULT)

Depends of the NIRVA command itself. See the NIRVA SYSTEM or services references in order to see if the command returns something in the output buffer.

Example

Here is a little Php script that opens a NIRVA session on the local NIRVA server and sends a NOP command.

```
<HTML>
<HEAD>
</HEAD>
<BODY>

My Page PHP
```

```
<?php
print("Test nirva PHP connector");
?>
<br>
<?php
nvc_openrequest("");
$REQUEST = $NVC_RESULT;
?>
<br>
<?php
if(nvc_command($REQUEST, "NV_CMD=|MISC:NOP|") == FALSE)
{
  print("  Error sending command to NIRVA server");
}
else
{
  nvc_command($REQUEST, "NV_CMD=|LOCAL:REQUEST:GET_SESSION_ID|");
  print("  Session ID : $NVC_RESULT\n");
  nvc_command($REQUEST, "NV_CMD=|SESSION:CLOSE|");
}
nvc_closerequest($REQUEST);
?>

</BODY>
</HTML>
```

Cold Fusion

Overview

The NIRVA Cold fusion connector is built as a COLD FUSION java custom tag. It’s directly using the nvc library that must be installed on the target machine. It has been tested with COLD FUSION MX version (version 6).

As for any NIRVA connector, the COLD FUSION connector to NIRVA uses requests to communicate to NIRVA. A request is just a door to access NIRVA and is completely independent from the NIRVA session itself. When creating a new request, the user can decide to connect an existing NIRVA session or to create a new one.

In fact, from COLD FUSION, the best way is to create a request at each page that needs to communicate to NIRVA and to pass the NIRVA session ID from page to page (and not the request ID).

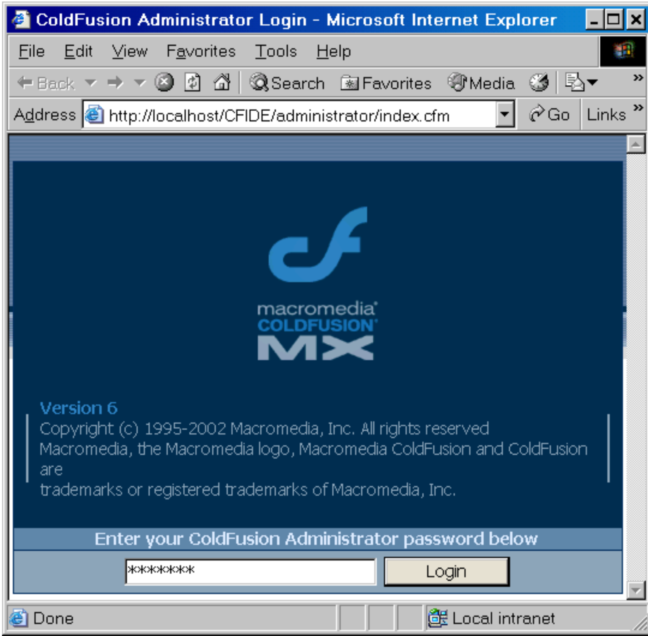
Installation

The COLD FUSION connector is delivered in the Nirva/Sdk/Connectors/cfusion directory. It consists of a single java class embedded in the file “nvcfx.class”. The “nvcfx.class” file must be copied somewhere on the disk where the JAVA run time environment of COLD FUSION can see it.

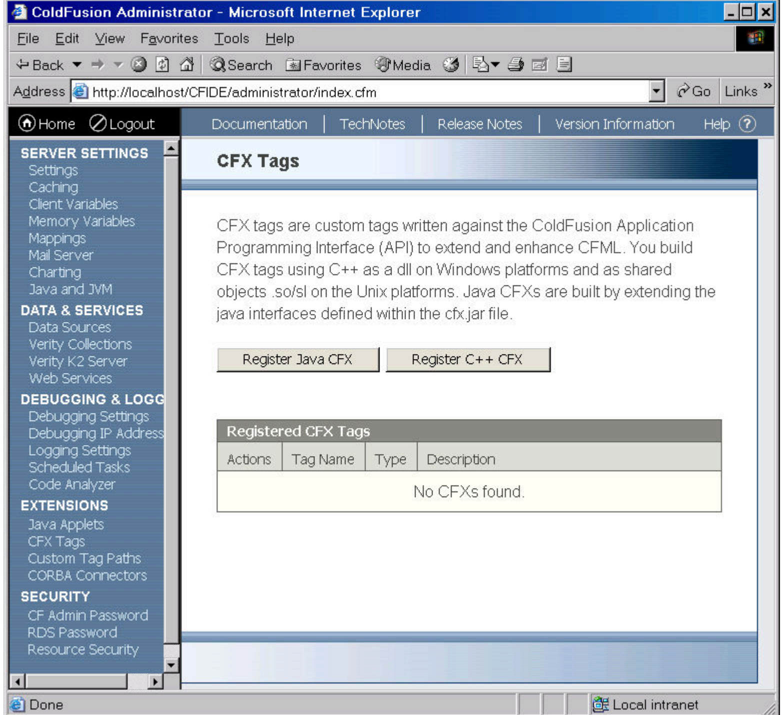
The COLD FUSION connector also requires the dll connector to be installed on the target machine. Please see the chapter about the DLL connector for information about the installation.

It's necessary to make a manual installation of the NIRVA custom tag to COLD FUSION. Here is the procedure:

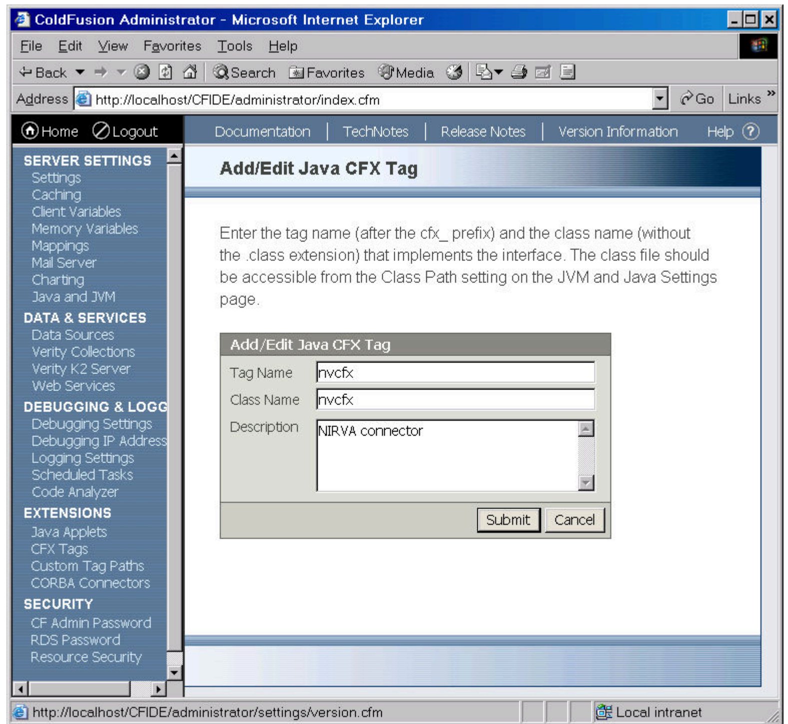
First enter the COLD fusion administration by giving your COLD FUSION administrator password:



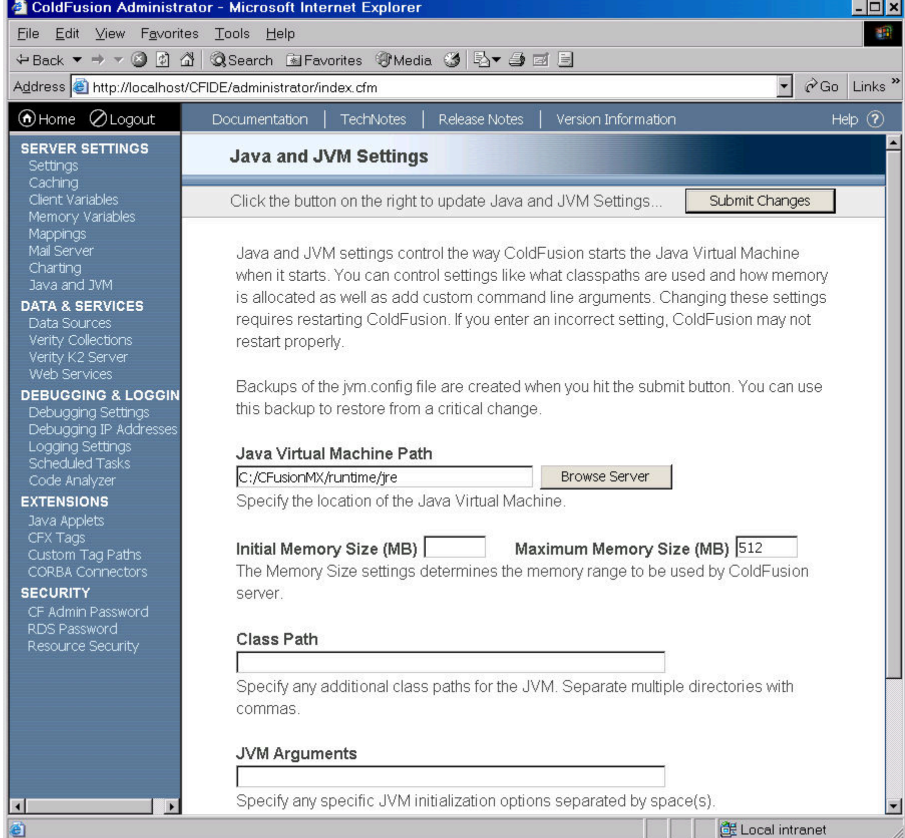
Then click on the "CFX tags" link in the administrator menu. This displays the following window:



Click on the button "Register Java CFX" and enters the following values:



Press the “Submit” button to validate your choice and then go to the “Java and JVM” link in the administrator menu. This displays the following window:



Then in the “Class Path” edit box, enter the path where you have installed the nvcfx.class file (delivered with NIRVA). If you have NIRVA locally installed, this path is the NIRVA bin directory (c:\nirva\Bin if you have installed NIRVA on c:\nirva under WINDOWS).

Press the “Submit changes” button in order to validate your choice. You then may need to stop and restart you COLD FUSION server.

Reference

This chapter gives the reference of the NIRVA calls that can be used from a COLD FUSION page in order to work with NIRVA.

Any call to NIRVA is done by typing the following line in the cfm source file:

```
<CFX_nvcfx NVCFX_ORDER="ConnectorOrder" Parameters>
```

Where *ConnectorOrder* can take the values “OpenRequest”, “CloseRequest” or “Command” and *Parameters* are the usual pairs of NIRVA command parameters.

If the *ConnectorOrder* is for a command (“Command” value), it may be omitted. Here are some examples of valid commands:

```
<CFX_nvcfx NVCFX_ORDER="OpenRequest"> opens a client NIRVA request.
<CFX_nvcfx NVCFX_HREQUEST="#HNIRVA#" NV_CMD=|MISC:NOP|"> sends a nop command to NIRVA.
```

When the connector order is to open a request (“OpenRequest”) NIRVA creates a request handle that can be accessed by the COLD FUSION variable named “NVCFX_RESULT” just after the command. This request handle must necessary be passed as parameter to the CFX_nvcfx calls when NVCFX_ORDER is set to “CloseRequest” or “Command”.

As example, the next lines open a NIRVA request, get its handle and close it:

```
<CFX_nvcfx NVCFX_ORDER="OpenRequest">
<CFSET HNIRVA = #NVCFX_RESULT#>
<CFX_nvcfx NVCFX_ORDER="CloseRequest" NVCFX_HREQUEST="#HNIRVA#">
```

All the NIRVA calls set the value of a COLD FUSION variable named “NVCFX_RESULT” that contains the result string of the call. If the function doesn’t return a string value, the NVCFX_RESULT variable is reset to an empty string.

Here is the description of the 3 possible connector orders (value of the ConnectorOrder parameter in the CFX_nvcfx call):

OpenRequest

Syntax

<CFX_nvcfx NVCFX_ORDER="*OpenRequest*" *Parameters*>

Description

This order opens a new client request. The parameters of the request are given in the parameters. The function returns a handle that is used with other orders to communicate with the request.

Parameters

Parameter	Description
Server	Nirva server TCP/IP address or machine name. This can be follow by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is installed. For example `Server="135.12.13.14:2035"` is a valid Server address. The default value for this parameter is `"127.0.0.1:1081"`. It's possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers. This allows a simple load balancing solution without requiring a load balancing box. This is a simple load balancing solution and the connected servers must all run. A proxy server can also be defined in a single address. At this time the format is the following: `proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be "http" or "https"; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.
ConnectionTimeOut	Time out for establishing the TCP/IP connection to the Nirva server. The default value is 10 seconds.
Session	Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn't exist. If the parameter is not given, Nirva creates a new session. The newly opened session stays opened until the time out occurs or until it's explicitly closed (by a SYSTEM CLOSE command) or after sending a command (or the command buffer) if the auto close mode has been set.
AutoClose	Auto close mode. If this parameter is set to "YES", the session is automatically closed after sending a command or the content of the command buffer to the server. Otherwise, the session stays open until it's

	explicitly closed by a SYSTEM CLOSE command or when the time out occurs.
SessionTimeOut	Time out value in seconds for a session. It's used only when a new session is to be created (so when the *Session* parameter is not provided or blank). If *SessionTimeOut* is "0", the default Nirva time out will be used. This is the default.
Application	Name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In fact, a session is always opened in the context of an application. If the application parameter is not provided, Nirva uses the default application named "NVDEF".
User	Application user name. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
Password	Application user password. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
NewPassword	*NewPassword* is the application new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
NewPasswordConfirm	*NewPasswordConfirm*. This parameter may be provided when the *NewPassword* parameter has been given. It allows Nirva to check if the new password is correct.
Open	Name of the NIRVA procedure to call when opening the new session. The default is "session_open". This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any open procedure, the open procedure name must be set "NV_SESSION_OPEN_NONE".
Close	Name of the NIRVA procedure to call when closing the new session. The default is "session_close". This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any close procedure, the close procedure name must be set "NV_SESSION_CLOSE_NONE".
Ssl	SSL mode. If this parameter is set to "YES", the request will use the HTTPS protocol to communicate to the NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the NIRVA server.
Certificate	is the optional client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.

MaxMsgSize	Maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise between performance and memory management on the client side. The minimum value is 10 kilobytes.
TempDir	Directory name where the connector can write the temporary files. If this parameter is not provided, NIRVA uses the current directory.
Sso	SSO (Single sign-on) flag. If this parameter is set to "YES" and if the Nirva application has been configured to authenticate users via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO.
SsoPrincipal	SSO (Single sign-on) principal name when using Kerberos protocol. This parameter has no meaning when not using SSO or when using SSO with NTLM.
SocketLog	Optional log file at socket (http) level.

Return value (NVCFX_RESULT variable)

The returned value is a handle to the NIRVA request. It must be used as first parameter to the other commands.

Command

Syntax

<CFX_nvcfx NVCFX_ORDER="*Command*" *Parameters*>

Description

This method sends a command to Nirva (LOCAL, SYSTEM or external services).

Some of the NIRVA commands return a string object that is written in the NVCFX_RESULT COLD FUSION variable.

If the command doesn't return a string value, the NVCFX_RESULT attribute is reset to an empty string. See the NIRVA LOCAL or SYSTEM references in order to see if the command returns something in the output buffer.

Parameters

NVCFX_HREQUEST	Handle to the NIRVA request previously returned by the "OpenRequest" order. This parameter is mandatory. If not provided or if its value doesn't' corresponds to a real request, the command will not fail but will do nothing.

NIRVA parameters — All the other parameters are interpreted directly by the NIRVA command. Please see the Nirva command syntax chapter for further information about the Nirva command.

Return value (NVCFX_RESULT variable)

Depends of the NIRVA command itself. See the NIRVA SYSTEM or services references in order to see if the command returns something in the output buffer.

CloseRequest

Syntax

<CFX_nvcfx NVCFX_ORDER="*CloseRequest*" *Parameters*>

Description

This method closes a request previously opened with the *OpenRequest* command.

It frees any resource used by the request.

If a request stays unused from at least one hour, NIRVA automatically closes it.

Parameters

NVCFX_HREQUEST — Handle to the NIRVA request previously returned by the “OpenRequest” order. This parameter is mandatory. If not provided or if its value doesn’t’ corresponds to a real request, the command will not fail but will do nothing.

Return value (NVCFX_RESULT variable)

None.

Example

Here is a little COLD FUSION script that opens a NIRVA session on the local NIRVA server and sends a NOP command.

```
<HTML>
<BODY>
<CFOUTPUT>

<!--- Open a nirva request with the default connection parameters --->
<CFX_nvcfx NVCFX_ORDER="OpenRequest">

<!--- Set the HNIRVA varaible to the NIRVA request handle --->
```

```
<CFSET HNIRVA = #NVCFX_RESULT#>

<!--- Send a nop command to the local NIRVA server (also opens the NIRVA session) --->
<CFX_nvcfx NVCFX_HREQUEST="#HNIRVA#" NV_CMD=|MISC:NOP|">

<!--- Get and print the NIRVA session ID --->
<CFX_nvcfx NVCFX_HREQUEST="#HNIRVA#" NV_CMD=|LOCAL:REQUEST:GET_SESSION_ID|">
<br>Nirva session ID = #NVCFX_RESULT#<br>

<!--- Close the NIRVA session --->
<CFX_nvcfx NVCFX_HREQUEST="#HNIRVA#" NV_CMD=|SESSION:CLOSE| ">

<!--- Close the NIRVA request --->
<CFX_nvcfx NVCFX_ORDER="CloseRequest" NVCFX_HREQUEST="#HNIRVA#">

</CFOUTPUT>
</BODY>
</HTML>
```

Java

Overview

The NIRVA JAVA client connector is built using the java native connector (JNI). It’s directly using the nvc library that must be installed on the target machine.

The JAVA connector is based on a small JAVA class named “nvcj”. This class is encapsulated in a package named “com.nirvasoft.nirva” and is distributed in the file nirva.jar.

Installation

The JAVA connector is delivered in the Nirva/Sdk/Connectors/java directory. The JAVA connector consists of a single java class embedded in the file “nirva.jar”. The “nirva.jar” file must be added to the CLASSPATH in order for the JAVA run time environment to see it.

When installing the NIRVA java connector to be used in a J2EE application server (like IBM WebSphere), the nirva.jar file must be installed at system level. In fact the J2EE application server creates a JVM that controls all installed application. When an application starts, the application server creates another JVM for it. It’s important that the JVM that loads the nirva java connector class is the main application server JVM (system level) otherwise the nirva connector can be inaccessible after a restart of an application. This is a common limitation to all java classes using JNI technology.

The JAVA connector uses the JNI technology to communicate with the NIRVA servers so it also requires the dll connector to be installed on the target machine. Please see the chapter about the DLL connector for information about the installation.

Reference

This chapter gives the reference of the NIRVA methods that can be used from a Java program in order to work with NIRVA.

In fact the NIRVA JAVA connector provides a class named "`com.nirvasoft.nirva.nvcj`" that implements all these methods. In order to communicate with NIRVA, the java code must create an "`nvcj`" object.

The "`Command`" method is used to send commands to NIRVA. Some of these NIRVA commands (the ones using output buffer) return a string object that is written in the `nvcj` object `NvResult` attribute. The `NvResult` attribute is declared like this:

```
public java.lang.String NvResult;
```

If the command doesn't return a string value, the `NvResult` attribute is reset to an empty string.

---

nvcj

Syntax

nvcj()

nvcj(String ConnectString)

Description

This is the constructor of the nvcj class. The constructor requires a connection string. The first constructor (without argument) is using the default connection string.

Parameters

ConnectString	Connection string. This string contains the request parameters. This is a succession of pairs `ParameterName="ParameterValue"`. The pair separator is the blank character. The parameter name is case insensitive. The '=' character must follow the parameter name. The parameter value must be enclosed in double quote characters. If the parameter value contains itself a double quote character, it must be preceded by another double quote character. The double quote characters of the command string must be preceded by a backslash character (\"). Here are the available connection string parameters: ■ *Server* is the Nirva server TCP/IP address or machine name. This can be follow by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is

installed. For example `Server="135.12.13.14:2035"` is a valid Server address. The default value for this parameter is `"127.0.0.1:1081"`.
It's possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers. This allows a simple load balancing solution without requiring a load balancing box. This is a simple load balancing solution and the connected servers must all run.
A proxy server can also be defined in a single address. At this time the format is the following:
`proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be "http" or "https"; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

- *ConnectionTimeOut* is the time out for establishing the TCP/IP connection to the Nirva server. The default value is 10 seconds.
- *Session* is the Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn't exist. If the parameter is not given, Nirva creates a new session. The newly opened session stays opened until the time out occurs or until it's explicitly closed (by a SYSTEM CLOSE command) or after sending a command (or the command buffer) if the auto close mode has been set.
- *AutoClose* is the auto close mode. If this parameter is set to "YES", the session is automatically closed after sending a command or the content of the command buffer to the server. Otherwise, the session stays open until it's explicitly closed by a SYSTEM CLOSE command or when the time out occurs.
- *SessionTimeOut* is the time out value in seconds for a session. It's used only when a new session is to be created (so when the *Session* parameter is not provided or blank). If *SessionTimeOut* is "0", the default Nirva time out will be used. This is the default.
- *Application* is the name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In fact, a session is always opened in the context of an application. If the application parameter is not provided, Nirva uses the default application named "NVDEF".

- *User* is the application user name. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *Password* is the application user password. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *NewPassword* is the application new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
- *NewPasswordConfirm*. This parameter may be provided when the *NewPassword* parameter has been given. It allows Nirva to check if the new password is correct.
- *Open* is the name of the NIRVA procedure to call when opening the new session. The default is “session_open”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”.
- *Close* is the name of the NIRVA procedure to call when closing the new session. The default is “session_close”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”.
- *Ssl* is the SSL mode. If this parameter is set to “YES”, the request will use the HTTPS protocol to communicate to the NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the NIRVA server.
- *Certificate* is the optional client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.
- *MaxMsgSize* is the maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise between performance and memory management on the client side. The minimum value is 10 kilobytes.

- *TempDir* is the directory name where the connector can write the temporary files. If this parameter is not provided, NIRVA uses the current directory
- *Sso* is the SSO (Single sign-on) flag. If this parameter is set to “YES” and if the Nirva application has been configured to authenticate users via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO.
- *SsoPrincipal* is the SSO (Single sign-on) principal name when using Kerberos protocol. This parameter has no meaning when not using SSO or when using SSO with NTLM.
- *SocketLog* is the name of an optional log file at socket (http) level.

Return value

None.

Command

Syntax

int Command(String CommandString)

Description

This method sends a command to Nirva (LOCAL, SYSTEM or external services).

The `Command` method returns 1 in case of success and 0 in case of failure.

Some of the NIRVA commands return a string object that is written in the `nvcj` object `NvResult` attribute. The `NvResult` attribute is declared like this:

```
public java.lang.String NvResult;
```

If the command doesn’t return a string value, the `NvResult` attribute is reset to an empty string. See the NIRVA LOCAL or SYSTEM references in order to see if the command returns something in the output buffer.

Parameters

CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command.

Return value

1 if successful and 0 otherwise. In case of error, the error information is available by using the LOCAL:REQUEST:ERROR_INFO command.

GetResult

Syntax

String GetResult()

Description

This method retrieves the eventual result string of the last command sent to NIRVA with the "`Command`" method.

When the "Command" method is executed, it sets the value of the `NvResult` attribute but also keeps internally the eventual string.

The programmer can then use directly the NvResult attribute or use the GetResult method in order to retrieve the last command result.

Parameters

None.

Return value

Last command result.

Example

Here is a little JAVA program that opens a NIRVA session on the local NIRVA server and sends a NOP command.

```
// Test for connection to NIRVA
import com.nirvasoft.nirva.nvcj;

class nvtest
{
  public static void main(String[] args)
  {
          System.out.println("Test connector to NIRVA from Java\n");

          // Create the NIRVA request object
          // If the connection string has to be different than the default one
          // Please use the nvjc constructor that takes the connection string as argument
```

```
        nvcj MyRequest = new nvcj();

        System.out.println("Opening session\n");

        // Send a NOP command to NIRVA (this also opens the NIRVA session)
        MyRequest.Command("NV_CMD=|MISC:NOP");

        // Get the session ID
        MyRequest.Command("NV_CMD=|LOCAL:REQUEST:GET_SESSION_ID|");
        System.out.println("Session ID is " + MyRequest.NvResult+ "\n");

        // Close the NIRVA session
        MyRequest.Command("NV_CMD=|SESSION:CLOSE|");
  }
}
```

Dotnet

Overview

The Dotnet connector is based on a small Dotnet class named “nvcdn”. This class is encapsulated in the “Nirva” package and is distributed in the assembly file nirvadn.dll.

The Dotnet connector is using the nvc library (nvc.dll) that must be installed on the target machine.

The Dotnet connector is available only on Windows platforms having the dotnet framework installed (at least 3.5 version).

Installation

The Dotnet connector is delivered in the Nirva/Sdk/Connectors/dotnet directory. The Dotnet connector consists of a single dotnet class embedded in the assembly file “nirvadn.dll”.

The Dotnet connector communicates with the NIRVA servers so it also requires the dll connector (nvc.dll) to be installed on the target machine. Please see the chapter about the DLL connector for information about the installation.

Reference

This chapter gives the reference of the NIRVA methods that can be used from a Dotnet program in order to work with NIRVA.

In fact the NIRVA Dotnet connector provides a class named “`nvcdn`” that implement all these methods. In order to communicate with NIRVA, the dotnet code must create an “`nvcdn`” object.

The "`Command`" method is used to send commands to NIRVA. Some of these NIRVA commands (the ones using output buffer) return a string object that is written in the `nvcdn` object `NvResult` attribute. The `NvResult` attribute is declared like this:

```
public String NvResult;
```

If the command doesn't return a string value, the `NvResult` attribute is reset to an empty string.

---

nvcdn

Syntax

nvcdn()

nvcdn(String ConnectString)

Description

This is the constructor of the nvcdn class. The constructor requires a connection string. The first constructor (without argument) is using the default connection string.

Parameters

ConnectString

Connection string. This string contains the request parameters. This is a succession of pairs `ParameterName="ParameterValue"`. The pair separator is the blank character. The parameter name is case insensitive. The '=' character must follow the parameter name. The parameter value must be enclosed in double quote characters. If the parameter value contains itself a double quote character, it must be preceded by another double quote character. The double quote characters of the command string must be preceded by a backslash character (\").

Here are the available connection string parameters:

- *Server* is the Nirva server TCP/IP address or machine name. This can be follow by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is installed. For example `Server="135.12.13.14:2035"` is a valid Server address. The default value for this parameter is `"127.0.0.1:1081"`.
  It's possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers. This allows a simple load balancing solution without requiring a load balancing box. This is a

simple load balancing solution and the connected servers must all run. A proxy server can also be defined in a single address. At this time the format is the following: `proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be “http” or “https”; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

- *ConnectionTimeOut* is the time out for establishing the TCP/IP connection to the Nirva server. The default value is 10 seconds.
- *Session* is the Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn’t exist. If the parameter is not given, Nirva creates a new session. The newly opened session stays opened until the time out occurs or until it’s explicitly closed (by a SYSTEM CLOSE command) or after sending a command (or the command buffer) if the auto close mode has been set.
- *AutoClose* is the auto close mode. If this parameter is set to “YES”, the session is automatically closed after sending a command or the content of the command buffer to the server. Otherwise, the session stays open until it’s explicitly closed by a SYSTEM CLOSE command or when the time out occurs.
- *SessionTimeOut* is the time out value in seconds for a session. It’s used only when a new session is to be created (so when the *Session* parameter is not provided or blank). If *SessionTimeOut* is “0”, the default Nirva time out will be used. This is the default.
- *Application* is the name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In fact, a session is always opened in the context of an application. If the application parameter is not provided, Nirva uses the default application named “NVDEF”.
- *User* is the application user name. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *Password* is the application user password. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).

- *NewPassword* is the application new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
- *NewPasswordConfirm*. This parameter may be provided when the *NewPassword* parameter has been given. It allows Nirva to check if the new password is correct.
- *Open* is the name of the NIRVA procedure to call when opening the new session. The default is “session_open”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”.
- *Close* is the name of the NIRVA procedure to call when closing the new session. The default is “session_close”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”.
- *Ssl* is the SSL mode. If this parameter is set to “YES”, the request will use the HTTPS protocol to communicate to the NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the NIRVA server.
- *Certificate* is the optional client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.
- *MaxMsgSize* is the maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise between performance and memory management on the client side. The minimum value is 10 kilobytes.
- *TempDir* is the directory name where the connector can write the temporary files. If this parameter is not provided, NIRVA uses the current directory
- *Sso* is the SSO (Single sign-on) flag. If this parameter is set to “YES” and if the Nirva application has been configured to authenticate users

via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO.

- *SsoPrincipal* is the SSO (Single sign-on) principal name when using Kerberos protocol. This parameter has no meaning when not using SSO or when using SSO with NTLM.
- *SocketLog* is the name of an optional log file at socket (http) level.

Return value

None.

Command

Syntax

int Command(String CommandString)

Description

This method sends a command to Nirva (LOCAL, SYSTEM or external services).

The `Command` method returns 1 in case of success and 0 in case of failure.

Some of the NIRVA commands return a string object that is written in the `nvcdn` object `NvResult` attribute. The `NvResult` attribute is declared like this:

```
public String NvResult;
```

If the command doesn't return a string value, the `NvResult` attribute is reset to an empty string. See the NIRVA LOCAL or SYSTEM references in order to see if the command returns something in the output buffer.

Parameters

CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command.

Return value

1 if successful and 0 otherwise. In case of error, the error information is available by using the LOCAL:REQUEST:ERROR_INFO command.

GetResult

Syntax

String GetResult()

Description

This method retrieves the eventual result string of the last command sent to NIRVA with the "`Command`" method.

When the "Command" method is executed, it sets the value of the `NvResult` attribute but also keeps internally the eventual string.

The programmer can then use directly the NvResult attribute or use the GetResult method in order to retrieve the last command result.

Parameters

None.

Return value

Last command result.

Example

Here is a little C# program that opens a NIRVA session on the local NIRVA server and sends a NOP command.

```
using System;
using Nirva;

// Test for connection to NIRVA
namespace nvtest
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Test connector to NIRVA from Dotnet");

            // Create the NIRVA request object
            // If the connection string has to be different
            // than the default one please use the nvcdn constructor
            // that takes the connection string as argument
            nvcdn MyNirva = new nvcdn();

            // create a string object on the server
            MyNirva.Command("NV_CMD=|OBJECT:CREATE| TYPE=|STRING| NAME=|MYSTRING|
```

```
                              VALUE=|Test|");
                 // and get its value back
                 MyNirva.Command("NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|MYSTRING|");
                 // display it
                 Console.WriteLine(MyNirva.NvResult);
                 // close the Nirva session
                 MyNirva.Command("NV_CMD=|SESSION:CLOSE|");
             }
         }
     }
```

For compiling this program, put in the same directory the source file (nvtest.cs), the dotnet connector (nirvadn.dll) and the dll connector (nvc.dll).

From a console, go into the source directory and type “`csc /r:nirvadn.dll nvtest.cs`” (you must have the dotnet framework installed and the csc compiler must be in your path). This creates an executable named nvtest.exe.

Perl

Overview

The NIRVA Perl client connector is built as a perl extension module. It directly uses the nvc library that must be installed on the target machine.

Installation

The Perl connector is delivered in the Nirva/Sdk/Connectors/perl directory as a source file. It must be compiled and called by a perl module named nirva.pm. Please consult the perl documentation for information about building perl modules.

The Perl connector consists of 2 files. The first file (“nirva.pm”) is the perl package file while the second file (“nvcperl.dll”, “nvcperl.so” or “nvcperl.a” depending of the platform) is the shared library containing the connector to NIRVA.

These 2 files must be manually copied on one of the Perl include directories. In order to know what are the Perl include directories, just issue a “`perl -V`” command and have a look on the directories listed after the “`@INC:`” line.

The Perl connector also requires the dll connector to be installed on the target machine. Please see the chapter about the DLL connector for information about the installation.

The Perl interpreter must have been compiled with multithread option.

Reference

This chapter gives the reference of the NIRVA functions that can be used from a Perl script in order to work with NIRVA.

All the functions must be called from the Perl script by adding the string “NVC::” before the function. For example, the callback function “`OpenRequest()`” must be called by typing “`NVC:: OpenRequest()`”.

Some of the functions require some parameters. These parameters are described for each function.

Some of the functions return an integer that is ‘1’ in case of success and ‘0’ in case of failure. All of them also set the value of a Perl variable named “`NVC::RESULT`” that contains the result string of the function. This variable can then be accessed like any Perl variable by adding the ‘$’ character before (“`$NVC::RESULT`”). If the function doesn’t return a string value, the NVC:RESULT variable is reset to an empty string.

The NIRVA Perl package is named “nirva” so the following line must be added at the start of the Perl script in order to tell him to use the nirva package:

```
use nirva;
```

The Nirva client Perl connector provides only 3 functions:

- *NVC::OpenRequest* creates a new request and returns a request handle.
- *NVC::CloseRequest* closes a previously opened request.
- *NVC::Command* sends commands to Nirva in the context of a request.

NVC::OpenRequest

Syntax

NVC::OpenRequest(ConnectString)

Description

This function opens a new client request. The parameters of the request are given in the ConnectString parameter. The function returns a handle that is used with other functions to communicate with the request.

Parameters

ConnectString	Connection string. This string contains the request parameters. This is a succession of pairs `ParameterName="ParameterValue"`. The pair separator is the blank character. The parameter name is case insensitive. The ‘=’ character must follow the parameter name. The parameter value must be enclosed in double quote characters. If the parameter value contains itself a double quote character, it must be preceded by another double quote character. The double quote characters of the connection

string must be preceded by a backslash character (\").

Here are the available connection string parameters:

Server is the Nirva server TCP/IP address or machine name. This can be follow by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is installed. For example `Server="135.12.13.14:2035"` is a valid Server address. The default value for this parameter is `"127.0.0.1:1081"`.

It's possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers. This allows a simple load balancing solution without requiring a load balancing box. This is a simple load balancing solution and the connected servers must all run.

A proxy server can also be defined in a single address. At this time the format is the following: `proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be "http" or "https"; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

ConnectionTimeOut is the time out for establishing the TCP/IP connection to the Nirva server. The default value is 10 seconds.

Session is the Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn't exist. If the parameter is not given, Nirva creates a new session. The newly opened session stays opened until the time out occurs or until it's explicitly closed (by a SYSTEM CLOSE command) or after sending a command (or the command buffer) if the auto close mode has been set.

AutoClose is the auto close mode. If this parameter is set to "YES", the session is automatically closed after sending a command or the content of the command buffer to the server. Otherwise, the session stays open until it's explicitly closed by a SYSTEM CLOSE command or when the time out occurs.

SessionTimeOut is the time out value in seconds for a session. It's used only when a new session is to be created (so when the *Session* parameter is not provided or blank). If *SessionTimeOut* is "0", the default Nirva time out will be used. This is the default.

Application is the name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In fact, a session is always opened in the context of

	an application. If the application parameter is not provided, Nirva uses the default application named “NVDEF”.
User	is the application user name. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
Password	is the application user password. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
NewPassword	is the application new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
NewPasswordConfirm	This parameter may be provided when the *NewPassword* parameter has been given. It allows Nirva to check if the new password is correct.
Open	is the name of the NIRVA procedure to call when opening the new session. The default is “session_open”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”.
Close	is the name of the NIRVA procedure to call when closing the new session. The default is “session_close”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”.
Ssl	is the SSL mode. If this parameter is set to “YES”, the request will use the HTTPS protocol to communicate to the NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the NIRVA server.
Certificate	is the optional client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.
MaxMsgSize	is the maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise between performance and memory management on the client side. The minimum value is 10 kilobytes.
TempDir	is the directory name where the connector can write the temporary files. If this parameter is not provided, NIRVA uses the current directory

Unicode	tells if client is Unicode. When this parameter is set to “YES”, all the data sent to the server are supposed to be UTF-8 encoded and all data coming from server are automatically converted to UTF-8 by the server. When not in Unicode mode, the client is supposed to work in ISO-8859-1 character set.
Sso	is the SSO (Single sign-on) flag. If this parameter is set to “YES” and if the Nirva application has been configured to authenticate users via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO.
SsoPrincipal	is the SSO (Single sign-on) principal name when using Kerberos protocol. This parameter has no meaning when not using SSO or when using SSO with NTLM.
SocketLog	Optional log file at socket (http) level.

Return value

1 if successful and 0 otherwise.

Return string (NVC::RESULT)

The returned string is a handle to the NIRVA request. It must be used as first parameter to the other functions.

NVC::CloseRequest

Syntax

NVC::CloseRequest(Request)

Description

This function closes a request previously opened with the *NVC::OpenRequest* function.

It frees any resource used by the request.

Parameters

Request	Handle of the request as returned by the *NVC::OpenRequest* function.

Return value

Always 1.

Return string (NVC::RESULT)

None.

NVC::Command

Syntax

NVC::Command(Request, CommandString)

Description

This function sends a command to Nirva.

Parameters

Request	Handle of the request as returned by the *NVC::OpenRequest* function.
CommandString	NIRVA command string. Please see the Nirva command syntax chapter for further information about the Nirva command. The double quote characters of the command string must be preceded by a backslash character (\").

Return value

1 if successful and 0 otherwise.

Return string (NVC::RESULT)

Depends of the NIRVA command itself. See the NIRVA SYSTEM or services references in order to see if the command returns something in the output buffer.

Example

Here is a little Perl script that opens a NIRVA session on the local NIRVA server and sends a NOP command.

```
# Test for connection to NIRVA

# First tell the script to use the nirva package
use nirva;

# Open the request with the default connection string
NVC::OpenRequest();
$NirvaRequest = $NVC::RESULT;

# Send a NOP command to NIRVA (this also opens the NIRVA session)
NVC::Command($NirvaRequest, "NV_CMD=|MISC:NOP|");
```

```
# Close the NIRVA session
NVC::Command($NirvaRequest, "NV_CMD=|SESSION:CLOSE|");

# Close the request
NVC::CloseRequest($NirvaRequest);
```

Xml

Overview

The XML connector is one of the most powerful NIRVA features. NIRVA can accept and deliver any kind of XML data flow using the built in HTTP server or the MQ connector.

At the client point of view, the only necessary thing is a HTTP or MQ client.

The process is the following:

- The client requests a HTTP order (or MQ message) to NIRVA with an XML body.
- If the XML data doesn't respect the NIRVA XML grammar, NIRVA transforms it by the way of its embedded XSLT processor to an acceptable XML flow.
- Then NIRVA turns the XML data into NIRVA objects of the input container.
- NIRVA executes the command given in the URL (or message header for MQ) or directly inside the XML data.
- Then NIRVA gets the output container and transforms it to XML.
- NIRVA optionally processes the resulting XML via XSLT in order to deliver understandable XML data to the sender.
- Finally, NIRVA sends back the resulting XML data to the sender.

NIRVA accepts four XML models controlled by the NV_XML_MODEL parameter.

Installation

The Xml connector doesn't require any specific installation. In order to use it, one just needs to use a HTTP client.

Reference

The NIRVA Xml connector is accessible from any HTTP client. The HTTP order must respect the following:

- The POST or GET method must be used.

- The Command (URL) must respect the NIRVA XML connector command syntax.
- The body must be well formed XML data.
- The content type must not be multipart.

This chapter gives some general information about the use of the XML connector. For a complete description of the NIRVA XML grammar, please see the XML chapter in this documentation.

Syntax

The NIRVA XML connector command has the same syntax than the web browser syntax except the string '`/NVS?Command`' that is replaced by '`/NVS?XMLCommand`'.

However, if the string '`/NVS?Command`' is used and the HTTP order contains a Content-type header for xml, NIRVA automatically considers that the XML connector is used.

Input XML data

The XML input data may respect or not the NIRVA XML grammar (normal or simple models).

If its not a NIRVA XML, the command must include the "NV_XML_XSL_IN" parameter pointing to a NIRVA application, system or service XSL file in order to transform the input XML to NIRVA acceptable one. As described in the NV_XML_XSL_IN parameter reference, the XSL file may be a system, service or application file. If it's an application xsl file, NIRVA must now the application name. This one can be given in the NV_XML_XSL_IN parameter directly. If NIRVA doesn't find it in the NV_XML_XSL_IN parameter, it tries to get it from the NV_APPLICATION parameter that must then be given in the command. If not found, NIRVA then tries to get it from the given URL after the string "NV_APP". Here are some examples of valid Xml connector URL commands with an XSL transformation:

```
http://localhost:1081/nv_app_test/NVS?XMLCommand&NV_XML_XSL_IN=soaptonv
http://localhost:1081/NVS?XMLCommand&NV_XML_XSL_IN=test:soaptonv
http://localhost:1081/NVS?XMLCommand&NV_XML_XSL_IN=soaptonv&NV_APPLICATION=test
```

All these commands require transforming the input XML data with the xsl file named "soaptonv" of the "test" application.

If the input XML directly respects the NIRVA XML grammar, no transformation is necessary. The most usual URL will then be:

```
http://localhost:1081//NVS?XMLCommand
```

Command parameters

The command parameters may be given directly in the URL or in the XML data. It’s also possible to mix both but the parameters found in the XML data have priority. The only parameter that cannot be given in the XML data is the NV_XML_XSL_IN parameter.

When a parameter is given in the XML data, it must be in the NVCOMMAND element immediately following the NIRVA root element. Please see the XML chapter for a complete description of the NIRVA XML grammar.

Here is an example of an input XML data with command parameters:

```
http://localhost:1081//NVS?XMLCommand

<?xml version="1.0" encoding="ISO-8859-1" ?>
<NIRVA>
  <NVCOMMAND>
    <NVPARAM name="NV_CMD">MISC:NOP</NVPARAM>
    <NVPARAM name="NV_PROC">TEST</NVPARAM>
  </NVCOMMAND>
  <NVOBJ name="mystring" type="STRING">
    <NVDATA>Test string cet été là</NVDATA>
  </NVOBJ>
  <NVCONTAINER  name="mysubcontainer" />
</NIRVA>
```

This simple command populates the default input container with a string object named “mystring” and an empty subcontainer named “mysubcontainer”. It then executes the TEST procedure of the default application.

In this example, the NV_CLASS and NV_COMMAND parameters have been given. They can be omitted in order to use their default values. The default class is “MISC” and the default command is “NOP”.

Here is the same command with parameters given in the URL:

```
http://localhost:1081//NVS?XMLCommand&NV_CMD =MISC:NOP &NV_PROC=TEST

<?xml version="1.0" encoding="ISO-8859-1" ?>
<NIRVA>
  <NVOBJ name="mystring" type="STRING">
    <NVDATA>Test string cet été là</NVDATA>
  </NVOBJ>
  <NVCONTAINER  name="mysubcontainer" />
</NIRVA>
```

Output XML data

The delivered data is an XML view of the output container or a part of it. The resulting XML can also be transformed from NIRVA grammar to another grammar by using the NV_XML_XSL parameter. Here is the list of the parameters having influence on the XML output:

NV_XML_CONTAINER Container to use as XML output data flow.

NV_XML_OBJECTS	Container objects to insert into the XML output data flow.
NV_XML_SUBCONTAINERS	Flag to insert also subcontainers into the XML output data flow.
NV_XML_WITH_DATA	Flag to insert also object data into the XML data output flow.
NV_XML_HTTP_HEADERS	Flag to insert also input HTTP headers into the XML output data flow.
NV_XML_HTTP_VARIABLES	Flag to insert also session variables into the XML output data flow.
NV_XML_XSL	Name of an xsl file for Nirva to parse XML output data flow into another XML page.
NV_XML_ENCODING	Character set to use in the generated XML data.
NV_XML_OUTPUT	Defines the output type when the command comes from a browser or XML connector.
NV_XML_NSREF	Defines the namespace URL for the XML output.
NV_XML_NSPREFIX	Defines the namespace prefix for the XML output.
NV_XML_XSL_ERROR	Name of an xsl file for Nirva to parse XML error data into an HTML or other XML page.
NV_XML_MODEL	Xml model for output (normal, simple, compact or tiny).

Here is an example of output XML data:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<NIRVA xmlns="http://www.nirva-systems.com/2004-03-29/nirva-xml" session="2CCF32BAC0"
application="NVDEF" user="" source="CLIENT" host="127.0.0.1" local="YES">
  <NVCONTAINER name="nvdef">
    <NVOBJ name="mystring" type="STRING">
      <NVDATA>Test string cet été là</NVDATA>
    </NVOBJ>
    <NVCONTAINER  name="mysubcontainer" />
  </NVCONTAINER>
</NIRVA>
```

Error management

When an error occurs, if it’s an error at the HTTP level, the HTTP response gives back an error code different of 200.

If the error is a NIRVA error, the HTTP response is 200 (OK) and the output XML has the following format:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<NIRVA>
  <NVERROR>
    <NVERRORCODE>101</NVERRORCODE>
    <NVERRORSERVICE>SYSTEM</NVERRORSERVICE>
```

```
    <NVERRORCLASS>COMMAND</NVERRORCLASS>
    <NVERRORDESC>Command not available</NVERRORDESC>
    <NVERRORINFO>SYSTEM:MISC:NOPR</NVERRORINFO>
  </NVERROR>
</NIRVA>
```

This error output can be transformed via XSL if a NV_XML_XSL_ERROR parameter has been given in the order.

The encoding is always ISO-8859-1. The error information data maybe namespace qualified following the namespace used in the input XML or given by the NV_XML_NSREF and NV_XML_NSPREFIX parameters.

The exact syntax of the XML error data is given in the XML chapter of this documentation. It depends of the XML model (normal or simple).

Example

The XML connector can be tested by using the nvcc tool with the testxml.txt file.

Let’s write an input file name xmlin.xml containing the following XML data:

```
<?xml version="1.0" ?>
<NIRVA>
  <NVCOMMAND>
    <NVPARAM name="NV_CMD">MISC:NOP</NVPARAM>
  </NVCOMMAND>
  <NVOBJ name="myobj" type="STRING">
    <NVDATA>Test string</NVDATA>
  </NVOBJ>
  <NVCONTAINER  name="mycontainer">
  </NVCONTAINER>
</NIRVA>
```

Then we run the command:

```
nvcc -i testxml.txt xmlin.xml xmlout.xml
```

This should produce the following result in xmlout.xml:

```
<?xml version="1.0" ?>
<NIRVA session="2454FBFBD9" application="NVDEF" user="" source="CLIENT" host="127.0.0.1"
local="YES">
  <NVCONTAINER name="nvdef">
    <NVOBJ name="myobj" type="STRING">
      <NVDATA>Test string</NVDATA>
    </NVOBJ>
```

```
    <NVCONTAINER NAME="mycontainer" />
    </NVCONTAINER>
  </NVCONTAINER>
</NIRVA>
```

In fact, we have the populated the default input container named “nvdef” with a string object named “myobj” and an empty subcontainer named “mycontainer”.

The output gives the XML data of the output container which is in this case the same as the input container (nvdef).

Soap

Overview

The SOAP connector is similar to the NIRVA XML connector except that the XML data is contained into an SOAP message.

The SOAP message must respect the minimum SOAP standard with a root element named “Envelope” containing at least a single element named “Body”. It may be namespace qualified.

The “Body” element of the SOAP message must contain XML data that respects the NIRVA grammar.

NIRVA accepts two XML models: normal and simple one.

With the SOAP connector, the model is automatically selected from the input SOAP body. If the input SAOP body is coded according to the simple model, output will be simple. If the input SAOP body is coded according to the normal model, output will be normal. The NV_XML_SIMPLE parameter allows forcing the selection of the output model.

Installation

The SOAP connector doesn’t require any specific installation. In order to use it, one just needs to use a HTTP client.

Reference

The NIRVA Soap connector is accessible from any HTTP client. The HTTP order must respect the following:

- The POST or GET method must be used.
- The Command (URL) must respect the NIRVA SOAP connector command syntax.
- The body must be well formed SOAP data.
- The content type must not be multipart.

This chapter gives some general information about the use of the SOAP connector. For a complete description of the NIRVA XML grammar, please see the XML chapter in this documentation.

Syntax

The NIRVA SOAP connector command has the same syntax than the web browser syntax except the string '`/NVS?Command`' that is replaced by '`/NVS?SOAPCommand`'.

However, if the string '`/NVS?Command`' is used and the HTTP order contains a Content-type header for soap ("xml+soap" for example), NIRVA automatically considers that the SOAP connector is used.

Input SOAP data

The SOAP input data is contained in the Body part of the SOAP message. It must respect the NIRVA XML grammar.

Command parameters

The command parameters may be given directly in the URL or in the SOAP data. It's also possible to mix both but the parameters found in the SOAP data have priority.

When a parameter is given in the SOAP data, it must be in the NVCOMMAND element immediately following the NIRVA root element (itself in the Body element of the SOAP message). Please see the XML chapter for a complete description of the NIRVA XML grammar.

Here is an example of an input SOAP data with command parameters:

```
http://localhost:1081/NVS?SOAPCommand

<?xml version="1.0" encoding="ISO-8859-1" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<NIRVA>
<NVCOMMAND>
   <NVPARAM name="NV_CMD">MISC:NOP</NVPARAM>
   <NVPARAM name="NV_PROC">TEST</NVPARAM>
</NVCOMMAND>
<NVOBJ name="mystring" type="STRING">
   <NVDATA>Test string cet été là</NVDATA>
</NVOBJ>
<NVCONTAINER  name="mysubcontainer" />
</NIRVA>
</soap:Body>
</soap:Envelope>
```

This simple command populates the default input container with a string object named “mystring” and an empty subcontainer named “mysubcontainer”. It then executes the TEST procedure of the default application.

In this example, the NV_CLASS and NV_COMMAND parameters have been given. They can be omitted in order to use their default values. The default class is “MISC” and the default command is “NOP”.

Here is the same command with parameters given in the URL:

```
http://localhost:1081/NVS?SOAPCommand&NV_CMD=MISC:NOP&NV_PROC=TEST

<?xml version="1.0" encoding="ISO-8859-1" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<NIRVA>
 <NVOBJ name="mystring" type="STRING">
  <NVDATA>Test string cet été là</NVDATA>
 </NVOBJ>
 <NVCONTAINER  name="mysubcontainer" />
</NIRVA>
</soap:Body>
</soap:Envelope>
```

The SOAP connector can use the normal or simple XML input data model. Here is the same example using the simple input data model:

```
http://localhost:1081/NVS?SOAPCommand&NV_CMD=MISC:NOP&NV_PROC=TEST

<?xml version="1.0" encoding="ISO-8859-1" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<nirva>
 <mystring>Test string cet été là</mystring>
 <mysubcontainer  type="container" />
</nirva>
</soap:Body>
</soap:Envelope>
```

The command parameters can also be given in a special SOAP header named “nvheader”. This special SOAP header must have one single child named “nvcommand” that itself contains the command parameters.

```
http://localhost:1081/NVS?SOAPCommand

<?xml version="1.0" encoding="ISO-8859-1" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header>
```

```
<nvheader>
  <nvcommand>
    <nv_cmd>misc:nop</nv_cmd>
    <nv_proc>test</nv_proc>
  </nvcommand>
</nvheader>
</soap:Header>
<soap:Body>
<nirva>
  <mystring>Test string cet été là</mystring>
  <mysubcontainer type="container" />
</nirva>
</soap:Body>
</soap:Envelope>
```

Finally, the command parameters can also be given directly in the SOAP body in the “nvcommand” tag as first second level child of the SOAP Body element:

```
http://localhost:1081/NVS?SOAPCommand

<?xml version="1.0" encoding="ISO-8859-1" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<nirva>
  <nvcommand>
    <nv_cmd>misc:nop</nv_cmd>
    <nv_proc>test</nv_proc>
  </nvcommand>
  <mystring>Test string cet été là</mystring>
  <mysubcontainer type="container" />
</nirva>
</soap:Body>
</soap:Envelope>
```

Output SOAP data

The delivered data is an XML view of the output container or a part of it embedded in a SOAP message.

Here is the list of the parameters having influence on the SOAP output:

NV_XML_CONTAINER	Container to use as XML output data flow.
NV_XML_OBJECTS	Container objects to insert into the XML output data flow.
NV_XML_SUBCONTAINERS	Flag to insert also subcontainers into the XML output data flow.
NV_XML_WITH_DATA	Flag to insert also object data into the XML data output flow.
NV_XML_HTTP_HEADERS	Flag to insert also input HTTP headers into the XML output data flow.
NV_XML_HTTP_VARIABLES	Flag to insert also session variables into the XML output data flow.
NV_XML_ENCODING	Character set to use in the generated XML data.

NV_XML_SIMPLE	Allow choosing the output XML model (normal or simple). The default is simple model.

Error management

When an error occurs, if it's an error at the HTTP level, the HTTP response gives back an error code different of 200.

If the error is a NIRVA error, the HTTP response is 200 (OK) and the output SOAP has the following format:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Body>
  <env:faultcode>Server</env:faultcode>
  <env:faultstring>NIRVA-ERROR:SYSTEM:COMMAND:101:Command not
available:SYSTEM:MISC:NOPU</env:faultstring>
  <env:faultactor>127.0.0.1:1081</env:faultactor>
  <env:detail>
    <nvs:NVERROR xmlns:nvs="http://www.nirva-systems.com/2004-03-29/nirva-xml-fault">
    <nvs:NVERRORCODE>101</nvs:NVERRORCODE>
    <nvs:NVERRORSERVICE>SYSTEM</nvs:NVERRORSERVICE>
    <nvs:NVERRORCLASS>COMMAND</nvs:NVERRORCLASS>
    <nvs:NVERRORDESC>Command not available</nvs:NVERRORDESC>
    <nvs:NVERRORINFO>SYSTEM:MISC:NOPU</nvs:NVERRORINFO>
    </nvs:NVERROR>
  </env:detail>
</env:Body>
</env:Envelope>
```

If the simple model has been chosen (default), the error output will be the following one:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Body>
  <env:faultcode>Server</env:faultcode>
  <env:faultstring>NIRVA-ERROR:SYSTEM:COMMAND:101:Command not
available:SYSTEM:MISC:NOPU</env:faultstring>
  <env:faultactor>127.0.0.1:1081</env:faultactor>
  <env:detail>
    <nvs:nvfault xmlns:nvs="http://www.nirva-systems.com/2004-03-29/nirva-xml-fault">
    <nvs:code>101</nvs:code>
    <nvs:service>SYSTEM</nvs:service>
    <nvs:class>COMMAND</nvs:class>
    <nvs:description>Command not available</nvs:description>
    <nvs:info>SYSTEM:MISC:NOPU</nvs:info>
    </nvs:nvfault>
  </env:detail>
```

```
</env:Body>
</env:Envelope>
```

Example

The SOAP connector can be tested by using the nvcc tool with the testsoap.txt file.

Let's write an input file name soapin.xml containing the following SOAP data:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<nirva>
  <nvcommand>
    <nv_class>misc</nv_class>
    <nv_command>nop</nv_command>
    <nv_proc>test</nv_proc>
  </nvcommand>
  <mystring>Test string cet été là</mystring>
  <mysubcontainer type="container" />
</nirva>
</soap:Body>
</soap:Envelope>
```

Then we run the command:

```
nvcc -i testsoap.txt soapin.xml soapout.xml
```

This should produce the following result in soapout.xml:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Body>
<nvs:nirva xmlns:nvs="http://www.nirva-systems.com/2004-03-29/nirva-xml">
  <nvs:myobj>Test string cet été là</nvs:myobj>
  <nvs:mycontainer type="container" />
</nvs:nirva>
</env:Body>
</env:Envelope>
```

In fact, we have the populated the default input container named “nvdef” with a string object named “myobj” and an empty subcontainer named “mycontainer”.

The output gives the XML data of the output container which is in this case the same as the input container (nvdef).

Web service

Overview

The Web service connector allows to directly access web services defined into NIRVA.

A nirva web service is a business component that defines operations. Each operation has an input and output message. The structure of the input and output messages are also defined on the web service.

The web service access protocol is HTTP but one can also use it by the way of the MQ connector. The web service input and output data is transported into SOAP messages. A nirva web service also has a WSDL data that explicitly define the parameters of the web service (messages, operations, protocols, etc…)

Installation

The Web service connector doesn't require any specific installation. In order to use it, one just needs to use a HTTP client.

Reference

The NIRVA Web service connector is accessible from any HTTP client. The HTTP order must respect the following:

- The POST or GET method must be used.
- The Command (URL) must respect the NIRVA Web service connector command syntax.
- The body must be well formed SOAP data.
- The content type must not be multipart.

Syntax

The Web service command is composed of an URL and an HTTP SoapAction header.

The command must respect the URL syntax. Especially, any un-authorized character must be encoded with its corresponding ASCII code (on 2 digits) preceded by the '%' character. For example, the space character should be replaced by '%20'.

The URL has the following format:

```
http://Server:Port/Application/WebService/NVS?WEBSParameters
```

Where *Server* is the nirva server name or address, *Port* is its TCP/IP port, *Application* is the nirva application (can be omitted for default application), *WebService* is the name of the web service to use and *Parameters* are the optional parameters.

The parameter syntax is the usual one for URL parameters. Each parameter is composed in this way: `&ParameterName=ParameterValue`. On the contrary of the standard Nirva command syntax, the parameter value is not enclosed in double quotes. The parameter name is case insensitive. There is no space between parameters.

The parameter value can refer to the name of a Nirva variable (server session variable). At this time, the parameter value starts with the '#' character.

Since the parameters can be given directly in the URL, they can also be given in a dedicated SOAP header or in the SOAP body.

Here is an example of a Nirva URL for a web service:

```
http://127.0.0.1:1081/MyApp/MyWebService/NVS?WEBS
```

The name of the web service operation is generally given in the http SoapAction header but can also be given as the "OPERATION" parameter.

Here is the same example with operation given in the URL:

```
http://127.0.0.1:1081/MyApp/MyWebService/NVS?WEBS&OPERATION=MyOperation
```

In the same way, the name of the web service can be given with the WEBSERVICE parameter instead of the URL path. At this time, the application must also be given as parameter (in the NV_APPLICATION parameter).

Here is the same example with web service name given as parameter:

```
http://127.0.0.1:1081/NVS?WEBS&NV_APPLICATION=MyApp&WEBSERVICE=MyWebService&OPERATION=My
Operation
```

The content of the http body must be a valid SOAP message for the requested web service.

Input SOAP data

The SOAP input data is contained in the Body part of the SOAP message. It must respect the NIRVA XML grammar (simple model only) and especially the structure defined in the input message of the web service operation.

Command parameters

The command parameters can be given in the same way than for the SOAP connector except that only the simple XML model can be used. The preferred way is inside the SOAP header.

Please see the SOAP connector description for further information.

The NV_SERVICE, NV_CLASS and NV_COMMAND global parameters has no meaning with the web service connector.

Output SOAP data

The delivered data is an XML view of the output container that should respect the structure defined in the output message of the web service operation. This is under the responsibility of the web service builder. If some objects or subcontainers are missing, NIRVA automatically creates them but they are empty.

Error management

When an error occurs, if it's an error at the HTTP level, the HTTP response gives back an error code different of 200.

If the error is a NIRVA error, the HTTP response is 200 (OK) and the output SOAP has the following format:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
<env:Body>
  <env:faultcode>Server</env:faultcode>
  <env:faultstring>NIRVA-ERROR:SYSTEM:WEBSERVICE:127:The operation doesn't
exist:Operation</env:faultstring>
  <env:faultactor>127.0.0.1:1081</env:faultactor>
  <env:detail>
    <nvs:nvfault xmlns:nvs="http://www.nirva-systems.com/2004-03-29/nirva-xml-fault">
      <nvs:code>127</nvs:code>
      <nvs:service>SYSTEM</nvs:service>
      <nvs:class>WEBSERVICE</nvs:class>
      <nvs:description>The operation doesn't exist</nvs:description>
      <nvs:info>Operation</nvs:info>
    </nvs:nvfault>
  </env:detail>
</env:Body>
</env:Envelope>
```

The error message is also defined in the WSDL data of the web service.

Get WSDL data

The WSDL data is given to the user by the way of an URL of the form:

```
http://Server:Port/Application/WebService/NVS?WSDL
```

Where *Server* is the nirva server name or address, *Port* is its TCP/IP port, *Application* is the nirva application (can be omitted for default application) and WebService is the name of the web service to use. Here is an example:

http://127.0.0.1:1081/nvdef/toto/NVS?WSDL - Microsoft Internet Explorer


```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions name="toto"
   targetNamespace="http://pierredell:1081/nirva/nvdef/toto"
   xmlns:tns="http://pierredell:1081/nirva/nvdef/toto"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:nvt="http://www.nirva-systems.com/2004-03-29/nirva-
   basetypes" xmlns="http://schemas.xmlsoap.org/wsdl/">
   <documentation>My first web service</documentation>
 - <types>
   - <xs:schema elementFormDefault="qualified"
       targetNamespace="http://pierredell:1081/nirva/nvdef/toto">
       <xs:import namespace="http://www.nirva-systems.com/2004-
         03-29/nirva-basetypes"
         schemaLocation="http://pierredell:1081/Webservice/2004-03-
         29/basetypes.xsd" />
     - <xs:element name="input">
       - <xs:complexType>
         - <xs:all>
             <xs:element name="test" type="nvt:in_table" />
           - <xs:element name="toto">
             - <xs:complexType>
               - <xs:all>
                   <xs:element name="qsdfg"
                     type="nvt:string" />
                   <xs:element name="xmlfile"
                     type="nvt:in_file" />
                 </xs:all>
               </xs:complexType>
             </xs:element>
```


Test

The Web service connector can be tested by using the nvcc tool with the testwebs.txt file. For each web service, the nirva configuration tool delivers an example of input SOAP message for each operation.

IBM WebSphere MQ

Overview

The MQ connector allows accessing NIRVA by sending and receiving messages through the IBM WebSphere MQ product.

In this way an asynchronous communication to NIRVA is possible.

NIRVA is configured to listen on IBM WebSphere MQ queues for incoming messages. The number of listening queues is infinite. For each queue, one can define from 1 to 100 listening threads. In this way, it's possible to control the resources dedicated to each queue.

NIRVA accept MQ DATAGRAM and REQUEST type messages. A DATAGRAM message is a one way message from client to NIRVA. A REQUEST message is a 2 way messages. When NIRVA receives a REQUEST message, it answers with a REPLY message.

The message content can be one of the following types:

- XML
- SOAP
- Web service
- Binary
- File
- Command

Installation

The MQ connector requires the IBM WebSphere MQ client (or server) to be properly installed on the target machine. The user starting the nirva server must have enough rights to works with the queues defined on nirva. Nirva automatically detects the MQ client and displays an error message in the console (when nirva is started in console mode) and in the log if the MQ client is not found.

The MQ connector has a specific NIRVA license. Without this license, it’s possible to configure the queues but not to listen on them.

On the NIRVA server side, one must check that the MQ connector is available to NIRVA (if planned to use it).

Linux

Go into the Nirva Bin directory (generally /usr/nirva/Bin) and issue the following command:

```
ldd nvmq.so
```

This should display the list of library dependencies for the Nirva MQ interface. If some of these dependencies are not found, please consult IBM documentation for correct installation of the MQ client libraries.

Under Linux, IBM WebSphere MQ client delivers several sets of libraries depending of the installed compiler. By default, MQ client installs libraries for at least the 3.03 version of the gcc compiler. If the libstdc++.so.3 and libgcc_s.so.1 dependencies are not found, this may come from an older version of the compiler. At this time, please replace the MQ libraries found in /opt/mqm/lib by the ones corresponding to your compiler.

Here is an extract of the Linux MQ IBM documentation that should help to solve this problem:

When you are using the GNU C++ compiler you must make sure that any libraries containing C++ functions are built using the same version of the C++ complier as your application. To give you as much flexibility as possible, and to adhere to this condition, WebSphere MQ contains three versions of the WebSphere MQ C++ interface libraries. Each one matches a supported level of the GNU compiler.

The supported versions of the compiler are:

- v 2.95.2
- v 3.0.3 (the default)

Libraries that match these versions are in the /opt/mqm/lib/<version> directory.

Links are created from the default version to the /opt/mqm/lib directory.

To verify the version of the compiler that you are using, enter the command

```
g++ --version
```

If your chosen compiler version does not match the default version selected by WebSphere MQ, you must reconfigure the compiler to use the correct libraries by linking the C++ libraries in the WebSphere MQ lib directory to those that match your compiler. To do this, issue the following commands replacing <version> with the version of your compiler.

```
ln -s -f /opt/mqm/lib/<version>/libimqb23gl.so /opt/mqm/lib/libimqb23gl.so
ln -s -f /opt/mqm/lib/<version>/libimqs23gl.so /opt/mqm/lib/libimqs23gl.so
ln -s -f /opt/mqm/lib/<version>/libimqc23gl.so /opt/mqm/lib/libimqc23gl.so
ln -s -f /opt/mqm/lib/<version>/libimqb23gl_r.so /opt/mqm/lib/libimqb23gl_r.so
ln -s -f /opt/mqm/lib/<version>/libimqs23gl_r.so /opt/mqm/lib/libimqs23gl_r.so
ln -s -f /opt/mqm/lib/<version>/libimqc23gl_r.so /opt/mqm/lib/libimqc23gl_r.so
```

If the version of your compiler doesn’t match, please use the previous provided version. For example if your compiler version is 2.96, please use the 2.95.2 libraries.

AIX

Go into the Nirva Bin directory (generally /usr/nirva/Bin) and issue the following command:

```
Dump -H nvmq.a
```

This should display the list of library dependencies for the Nirva MQ interface. If some of these dependencies are not found, please consult IBM documentation for correct installation of the MQ client libraries.

Solaris

Go into the Nirva Bin directory (generally /usr/nirva/Bin) and issue the following command:

```
ldd nvmq.so
```

This should display the list of library dependencies for the Nirva MQ interface. If some of these dependencies are not found, please consult IBM documentation for correct installation of the MQ client libraries.

Hpux pa-risc

Go into the Nirva Bin directory (generally /usr/nirva/Bin) and issue the following command:

```
ldd nvmq.sl
```

This should display the list of library dependencies for the Nirva MQ interface. If some of these dependencies are not found, please consult IBM documentation for correct installation of the MQ client libraries.

Hpux itanium

Go into the Nirva Bin directory (generally /usr/nirva/Bin) and issue the following command:

```
ldd nvmq.so
```

This should display the list of library dependencies for the Nirva MQ interface. If some of these dependencies are not found, please consult IBM documentation for correct installation of the MQ client libraries.

Windows

There is no special to do for checking the MQ library availability. If MQ has been correctly installed, the libraries should be available.

Configuration

The MQ connector can be configured from the NIRVA configuration tool. Please see the configuration chapter for further information.

Reference

Message type

NIRVA accepts MQ DATAGRAM and REQUEST type messages. A DATAGRAM message is a one way message from a client to NIRVA. A REQUEST message is a 2 way message. When NIRVA receives a REQUEST message, it answers with a REPLY message.

When receiving a DATAGRAM message, NIRVA processes it but doesn't send back any response, even in case of error.

When receiving a REQUEST message, NIRVA answers with a REPLY message. Several input message parameters coming from the message descriptor (MQMD) control the way NIRVA will answer:

- Correlation ID
- Message ID
- Report
- Persistence

If the REQUEST message has a persistent status, the REPLY message will be also persistent.

By default, NIRVA copies the REQUEST message ID to the REPLY correlation ID and creates a new message ID for the REPLY message. This behavior can be changed by using the QRO_NEW_MSG_ID, MQRO_PASS_MSG_ID, MQRO_COPY_MSG_ID_TO_CORREL_ID and MQRO_PASS_CORREL_ID options in the Report field of the REQUEST message descriptor. Please see the IBM WebSphere MQ Application programming reference for further information.

When receiving a request message, gets the name of the output queue (for writing response) from the ReplyToQ message descriptor field. If this parameter is blank, NIRVA doesn't answer and generates an error except if a filter has been defined on the queue. When a filter has been defined, NIRVA takes the input queue name as output queue. See next chapter for message filtering information.

The name of the output queue can also be changed by providing the NV_MQ_OUTPUT parameter in the command part of the message content.

Filter

One can configure NIRVA for listening messages having a specific correlation ID value. When a filter has been defined, NIRVA only accept messages having the same value (case dependant) in the correlation ID field of the message descriptor than the filter string. See the configuration chapter for learning haw to set a queue filter.

Message content

The message content can be one of the following types:

- XML
- SOAP

- Web service
- Binary
- File
- Command

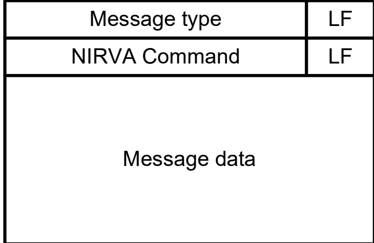
The message itself is always composed of 2 parts: a header and a data part.

The message header is different for input and output messages.

Input message

The input message header is composed by a first string giving the NIRVA message type and a second string giving the optional NIRVA command. Both strings must be followed by a line feed character:

Message with header:



Message without header:

The message type can be one of the following values:

- NVMQ_MSG_XML
- NVMQ_MSG_SOAP
- NVMQ_MSG_WEBS
- NVMQ_MSG_BINARY
- NVMQ_MSG_FILE
- NVMQ_MSG_COMMAND

The NVMQ_MSG_XML message type tells NIRVA that the message data contains XML data. The message is then processed by NIRVA exactly as if it comes from the XML connector.

The NVMQ_MSG_SOAP message type tells NIRVA that the message data contains SOAP data. The message is then processed by NIRVA exactly as if it comes from the SOAP connector.

The NVMQ_MSG_WEBS message type tells NIRVA that the message data contains a request to a NIRVA web service. The message is then processed by NIRVA exactly as if it comes from the Web service connector. Since there is no URL or SoapAction header, the “OPERATION” and “WEBSERVICE” parameters must be given in the NIRVA command.

The NVMQ_MSG_BINARY message type tells NIRVA that the message data must be written into a NIRVA binary object in the input container. The name of the binary object is then controlled by the "NV_MQ_OBJNAME" parameter in the NIRVA command.

The NVMQ_MSG_FILE message type tells NIRVA that the message data must be written into a NIRVA file object in the input container. The name of the file object is then controlled by the "NV_MQ_OBJNAME" parameter in the NIRVA command.

The NVMQ_MSG_COMMAND message type tells NIRVA that there is no message data so NIRVA just have to process the given command.

The second string of the header is the optional NIRVA command. Even if not provided, the nirva command must be followed by a line feed character. The syntax of the nirva command is the standard syntax.

If the message type cannot be determined by NIRVA, NIRVA considers that the entire message is message data with XML format (message without header). In this way, it's possible to send simple XML messages to NIRVA without specifying any header.

Here is an example of an input message:

```
NVMQ_MSG_FILE
NV_CMD=|MISC:NOP| NV_MQ_OBJNAME=|MyFile| NV_PROC=|MyProc|
My file content
```

This simple message tells NIRVA to put the string "My file content" into a NIRVA file object named "MyFile" and then to run the application procedure named "MyProc".

Output message

The output message header is composed by a first string giving the NIRVA message type and 1 or 5 strings giving the error information. All strings are followed by a line feed character:

Message without error:

Message type	LF
0	LF
Message data	

Message with error:

Message type	LF
Error code	LF
Error service	LF
Error class	LF
Error description	LF
Error information	LF
Message data	

Message without header:

The message type can be one of the following values:

- NVMQ_MSG_XML
- NVMQ_MSG_SOAP
- NVMQ_MSG_WEBS
- NVMQ_MSG_BINARY
- NVMQ_MSG_FILE

If the message type is NVMQ_MSG_XML, the data part contains XML data as returned by the XML connector.

If the message type is NVMQ_MSG_SOAP, the data part contains SOAP data as returned by the SOAP connector.

If the message type is NVMQ_ MSG_XML, the data part contains SOAP data as returned by the Web service connector.

If the message type is NVMQ_ MSG_BINARY, the data part is the content of a binary object. This can occur only if the command is SYSTEM:OBJECT:GET for a binary object.

If the message type is NVMQ_ MSG_FILE, the data part is the content of a file object. This can occur only if the command is SYSTEM:OBJECT:GET for a file object.

The next part of the output header is the error code. If this value is equal to “0”, the message data directly follows (after the next line feed character). Otherwise, 4 lines follow that gives further information about the error. These 4 lines respectively correspond to the error service, error class, error description and error information. When there is some error, output data may follow the error information if the message is of kind NVMQ_MSG_XML, NVMQ_MSG_SOAP or NVMQ_MSG_WEBS. This corresponds to the usual error information of the XML, SOAP or Web service connectors.

It’s possible to avoid generating the output header by setting the NV_MSG_HEADER parameter to “NO”.

The format of the output message depends of the format of the input message.

If the input message is NVMQ_MSG_XML, NVMQ_MSG_SOAP or NVMQ_MSG_WEBS, the output message will be on the same type except if the command is SYSTEM:OBJECT:GET. At this time, the output message will be NVMQ_MSG_BINARY or NVMQ_MSG_FILE following the king of object requested.

If the input message is NVMQ_MSG_BINARY or NVMQ_MSG_FILE, the output message will be NVMQ_MSG_XML except if the command is SYSTEM:OBJECT:GET. At this time, the output message will be NVMQ_MSG_BINARY or NVMQ_MSG_FILE following the king of object requested.

Virtual printer

The virtual printer (VP) is a Nirva connector that retrieves files issued from desktop applications to send them on a Nirva server to be processed.

The virtual printer is seen as a printer on the user's workstation.

The virtual printer connector is located on the Sdk/Connectors/virtual_printer subdirectory of Nirva installation directory.

Please contact Nirva Systems for further information and documentation about this connector.

Flex

Overview

Adobe Flex is a software development kit (SDK) released by Adobe Systems for the development and deployment of cross-platform rich Internet applications based on the Adobe Flash platform. Flex applications can be written using Adobe Flash Builder or by using the freely available Flex compiler from Adobe.

Flex applications can communicate to Nirva using a dedicated connector delivered as source code for the Flex projects.

Installation

The FLEX connector is delivered with the Nirva Application Server in the Sdk/Connectors/flex directory. This is a single action script file named "NvFlex.as" that must be inserted in your Flash builder project.

In your flash builder project src directory, create a subdirectory "services/nirva" and copy the file NvFlex.as into it.

The Sdk/Connectors/flex directory contains also a Nirva application named « FLEX » that contains an example of a Nirva/Flex project including source code. You can install this application, run its start page (TestNirva.html) from the Nirva console and have a look on the Source/TestNirva/src directory for the source code.

Reference

The Flex connector is implemented into an action script class named NvFlex. This class provides properties for the Nirva connection and result and methods for sending commands to Nirva. Flex is an event based technology so for any command sent to Nirva, one must give the name of a callback function that will process the result.

Object declaration

To use the FLEX connector in your project, after installation, just create a namespace into your Application tag as in the example below:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
                    xmlns:s="library://ns.adobe.com/flex/spark"
                    xmlns:mx="library://ns.adobe.com/flex/mx"
                    xmlns:nirva="services.nirva.*">
...
```

Then you can instanciate the NvFlex object in your declaration tag:

```
 <fx:Declarations>
        <nirva:NvFlex id="nirva" withsession="true" application="FLEX"/>
 </fx:Declarations>
```

Here is the list of NvFlex object properties that can be used when instanciating the object:

withsession	Keep session context. This is a boolean value that can take values “true” or “false”. The default is “false”. When set to “true” the NvFlex object keeps the session context opened on The Nirva side. One must use the OpenSession and CloseSession methods for opening and closing the Nirva session. When set to “false” any command sent to Nirva opens and immedialtly closes the session after the command.
application	Name of the Nirva application to connect. Default is “NVDEF”.
user	User for connecting the nirva session. Default is “nvdef”.
password	User’s password.

host Host url. Example: “http://www.myhost:1081” or “https://thishost”. If this parameter is not gven, the connector uses the same host than the one where the flex application resides.

timeout Nirva session time out in seconds. If provided, this parameter overrides the default Nirva session time out.

open Name of the NIRVA procedure to call when opening the new session. The default is “session_open”. In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”.

close Name of the NIRVA procedure to call when closing the session. The default is “session_close”. In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”.

Methods

When the NvFlex object has been declared, one can use its methods. Here is the list and short description of the methods:

OpenSession Opens a session when the NvFlex object has been declared with the parameter “withsession” set to “true”.

CloseSession Closes a session opened with the OpenSession method.

Command Sends a command to Nirva.

GetObjectUrl Constructs a URL for retrieving a Nirva file or binary object. Used for example to get dynamic images.

GetUrl Constructs a URL to be used in some other flex objects (example: file upload).

Error management

When sending a command to nirva, one must provide a fault callback function that will be called in case of error. For example:

```
<mx:Button x="115" y="54" label="Run procedure"
click="nirva.Command('NV_PROC=|perl:aproc|', ProcResult, NirvaFaultHandler)"/>
```

The callback receives the fault event:

```
protected function NirvaFaultHandler(event:FaultEvent, mynirva:NvFlex):void
{
  // Just display the error message
  Alert.show(event.fault.faultCode + "\n" + event.fault.faultString + "\n" +
   event.fault.faultDetail, "Nirva error my handler");
}
```

The fault event is a standard flex fault object containing a fault code, a fault string and a fault detail. The fault code is a concatenation of the Nirva error service, error class and error code. The fault string is the Nirva error description and the fault detail is the Nirva error information.

The callback function also receives a reference to the NvFlex object that has generated the error. This object contains a propertiy named “error” that is an array containing the following members:

error[“service”]	Nirva error service.
error[“class”]	Nirva error class.
error[“code”]	Nirva error code.
error[“description”]	Nirva error description.
error[“info”]	Nirva error extra information.

Whe the callback method is not provided, the NvFlex object provides a default one that displays the error message.

Method Reference

OpenSession

Syntax

OpenSession(cbSuccess:Function = null, cbError:Function = null):void

Description

This function is used only when the “withsession” property has been set to true, otherwise it’s a nop operation. It opens a new session on nirva side if not already opened. Parameters of the session are controlled by the object properties (see Object declaration). If successful, the new session id is available in the object “session” property.

This function takes 2 callbacks as parameters that must be declared in this way:

```
function MySuccessCallback(mynirva:NvFlex):void

function MyErrorCallback(event:FaultEvent, mynirva:NvFlex):void
```

The first one is called in case of success and the second one is called in case of failure. These callbacks always receive a reference to the NvFlex so the user can access its results.

Parameters

cbSuccess	Callback function called in case of success.

cbError Callback function called in case of failure.

Return value

None

Example

```
<fx:Declarations>
  <nirva:NvFlex id="nirva" withsession="true" application="FLEX"/>
</fx:Declarations>
…
<mx:Button x="11" y="54" label="Open session"
click="nirva.OpenSession(NirvaSuccessHandler, NirvaFaultHandler)"/>
…
protected function NirvaSuccessHandler(mynirva:NvFlex):void
{
  Alert.show(mynirva.session,"In callback");
}
protected function NirvaFaultHandler(event:FaultEvent, mynirva:NvFlex):void
{
  Alert.show(event.fault.faultCode + "\n" + event.fault.faultString + "\n" +
event.fault.faultDetail, "Nirva error my handler");
}
```

CloseSession

Syntax

CloseSession():void

Description

This function is used only when the “withsession” property has been set to true, otherwise it’s a nop operation. It closes the current session if one has been opened.

Parameters

None

Return value

None

Example

Example:

```
<fx:Declarations>
  <nirva:NvFlex id="nirva" withsession="true" application="FLEX"/>
</fx:Declarations>
…
<mx:Button x="11" y="54" label="Close session" click="nirva.CloseSession()"/>
```

Command

Syntax

Command(cmd:String, cbSuccess:Function = null, cbError:Function = null):void

Description

This function sends a command to Nirva. Some Nirva commands may return a string value (output buffer). In this case, this value is available via the “result” property of the NvFlex object. The “result” property is defined as a String object. The FLEX connector uses the nirva XML connector with the tiny model. The resulting XML is available in the “document” property of the NvFlex object. The “document” property is defined as a XML object. Both “result” and “document” properties are bindable allowing direct access to other FLEX objects.

This function takes 2 callbacks as parameters that must be declared in this way:

```
function MySuccessCallback(mynirva:NvFlex):void

function MyErrorCallback(event:FaultEvent, mynirva:NvFlex):void
```

The first one is called in case of success and the second one is called in case of failure. These callbacks always receive a reference to the NvFlex so the user can access its results.

Parameters

cmd	This parameter is a string which contains the command to be sent to Nirva.
cbSuccess	Callback function called in case of success.
cbError	Callback function called in case of failure.

Return value

None

Example

```
<fx:Declarations>
  <nirva:NvFlex id="nirva" withsession="true" application="FLEX"/>
</fx:Declarations>
…
<mx:Button x="115" y="54" label="Run procedure"
click="nirva.Command('NV_PROC=|perl:aproc|', ProcResult, NirvaFaultHandler)"/>
…
protected function ProcResult(mynirva:NvFlex):void
{
  Alert.show(mynirva.result,"Procedure result");
}
protected function NirvaFaultHandler(event:FaultEvent, mynirva:NvFlex):void
{
  Alert.show(event.fault.faultCode + "\n" + event.fault.faultString + "\n" +
event.fault.faultDetail, "Nirva error my handler");
}
```

GetObjectUrl

Syntax

GetObjectUrl(name:String, params:String, cbError:Function = null):String

Description

This functions constructs an url for the OBJECT:GET Nirva command in order to get an object file. It can be used for retrieving dynamic images for example.

Parameters

name	Nirva file or binary object to get.
params	Other parameters of the OBJECT:GET Nirva command.
cbError	Callback function called in case of failure.

Return value

Nirva url.

Example

```
<fx:Declarations>
  <nirva:NvFlex id="nirva" withsession="true" application="FLEX"/>
</fx:Declarations>
…
```

```
<mx:Image id="myimage" x="397" y="135" width="306" height="171"/>
<mx:Button x="554" y="106" label="Image1"
click="{myimage.source=nirva.GetObjectUrl('MYIMAGE', 'NV_PROC=|perl:get_image|
IMG=|exterieur_1_small.jpg|', NirvaFaultHandler)}"/>
…
protected function NirvaFaultHandler(event:FaultEvent, mynirva:NvFlex):void
{
  Alert.show(event.fault.faultCode + "\n" + event.fault.faultString + "\n" +
event.fault.faultDetail, "Nirva error my handler");
}
```

GetUrl

Syntax

GetUrl(params:String, cbError:Function = null):String

Description

This function constructs an url to be used in FLAX object requesting an url as parameter.

Parameters

params	Nirva command parameters.
cbError	Callback function called in case of failure.

Return value

Nirva url.

Example

```
<fx:Declarations>
  <nirva:NvFlex id="nirva" withsession="true" application="FLEX"/>
</fx:Declarations>
…
private function fileRef_select(evt:Event):void {
  try {
   message.text = "size (bytes): " + numberFormatter.format(fileRef.size);
    fileRef.upload(new URLRequest(nirva.GetUrl("NV_PROC=|perl:upload|")), "MYFILE");
  }
   catch (err:Error) {
   message.text = "ERROR: zero-byte file";
  }
}
```

Ajax

Overview

What is Ajax ?

Asynchronous JavaScript and XML or Ajax for short is new web development technique used for the development of most interactive website. Ajax helps you in making your web application more interactive by retrieving small amount of data from web server and then showing it on your application. You can do all these things without refreshing your page.

Usually in all the web applications, the user enters the data into the form and then clicks on the submit button to submit the request to the server. Server processes the request and returns the view in new page (by reloading the whole page). This process is inefficient, time consuming, and a little frustrating for the user if only a small amount of data exchange is required. For example in a user registration form, this can be frustrating thing for the user, as whole page is reloaded only to check the availability of the user name. Ajax will help in making your application more interactive. With the help of Ajax you can tune your application to check the availability of the user name without refreshing the whole page.

Ajax is a popular technology for Web 2.0 applications.

Technology

Ajax is not a single technology, but it is a combination of many technologies. These technologies are supported by modern web browsers. Following are techniques used in the Ajax applications.

JavaScript. JavaScript is used to make a request to the web server. Once the response is returned by the webserver, more JavaScript can be used to update the current page. DHTML and CSS is used to show the output to the user. JavaScript is used very heavily to provide teh dynamic behavior to the application.

Asynchronous Call to the Server: Most of the Ajax application used the XMLHttpRequest object to send the request to the web server. These calls are Asynchronous and there is no need to wait for the response to come back. User can do the normal work without any problem.

XML. XML may be used to receive the data returned from the web server. JavaScript can be used to process the XML data returned from the web server easily.

How it works ?

When user first visits the page, the Ajax engine is initialized and loaded. From that point of time user interacts with Ajax engine to interact with the web server. The Ajax engine operates asynchronously while sending the request to the server and receiving the response from server. Ajax life cycle within the web browser can be divided into following stages:

- User Visit to the page. User visits the URL by typing URL in browser or clicking a link from some other page.
- Initialization of Ajax engine. When the page is initially loaded, the Ajax engine is also initialized. The Ajax engine can also be set to continuously refresh the page content without refreshing the whole page.

- Event Processing Loop.
- Browser event may instruct the Ajax engine to send request to server and receive the response data
- Server response - Ajax engine receives the response from the server. Then it calls the JavaScript call back functions
- Browser (View) update - JavaScript request call back functions is used to update the browser. DHTML and css is used to update the browser display.

Benefits

Ajax is new very promising technology, which has become extremely popular these days. Here are the benefits of using Ajax:

- Ajax can be used for creating rich, web-based applications that look and works like a desktop application.
- Ajax is easy to learn. Ajax is based on JavaScript and existing technologies like XML, CSS, DHTML. etc. So, its very easy to learn Ajax
- Ajax can be used to develop web applications that can update the page data continuously without refreshing the whole page.

Nirva Ajax connector

Nirva is particularly suited to work with Ajax thanks to its embedded http server and XML connector.

The Nirva Ajax client connector is fully written in JavaScript and relies on the base XML communication classes provided by the major web browsers. Namely Firefox version 2.0 and above as well as Internet Explorer versions 7.0 and above are supported.

The Ajax connector is based on a small JavaScript class named “NvAjax.Request”. This class is available by including the JavaScript file “nvajax.js” found in all Nirva distributions since version 3.1.001.

Installation

The AJAX connector is delivered with the Nirva Application Server in the Sdk/Connectors/ajax directory. This is a single JavaScript file named “nvajax.js”. There is therefore no particular installation requirement. To use the AJAX connector, one must simply copy the file “nvajax.js” in the application Wroot directory and include a reference to them in your html code as in the example below:

```
<script type="text/javascript" src="nvajax.js"></script>
```

Reference

The NvAjax.Request class provides methods to send commands to a Nirva server. There are 3 ways to send a command:

The “Command” method sends a command that returns just the Nirva output buffer. In case of success the command output buffer is available in the “Result” property (string).

The “XMLCommand” is a synchronous method that sends a command and retrieves the resulting XML. The resulting XML document is available in the “Document” property as a DOM document. It can be accessed by current JavaScript DOM functions.

The “AsyncXMLCommand” is an asynchronous method that sends a command and retrieves the resulting XML. The “AsyncXMLCommand” has a parameter that points to a callback function called when the operation terminates. This callback function receives the NvAjax.Request object and its “Document” property that contains the resulting XML.

NvAjax.Request

Syntax

var myNirva = new NvAjax.Request(host,application,session)

Description

This constructor opens a new client request. The parameters of the request are given by the host, application and session parameters. The constructor returns a JavaScript object that is used with other functions to communicate with the request.

Parameters

host	The host parameter gives the address of the Nirva server to connect to. This can be empty for a relative path. Example: “http://localhost:1081”.
application	The application parameter gives the name of the Nirva application to connect to.
session	This parameter gives the Nirva session to connect to. This is usually the same session as the one which was used to generate the HTML page which is using the AJAX connector.

Example

This XSLT example creates a request to the current host.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="utf-8" media-type="text/html" omit-xml-
declaration="yes" indent="yes" />

<xsl:template match="/">
  <html>
  <head>
```

```
  <title>NIRVA application tests</title>
                 <script src="nvajax.js" />
                 <script>
                        <xsl:text>var NV = new NvAjax.Request("","</xsl:text>
                        <xsl:value-of select="/NIRVA/@application"/>
                 <xsl:text>","</xsl:text>
                 <xsl:value-of select="/NIRVA/@session" />
                        <xsl:text>");</xsl:text>
                 </script>
  </head>
  <body>
  <xsl:apply-templates select="NIRVA"/>
  </body>
  </html>
</xsl:template>

<xsl:template match="NIRVA">
...
</xsl:template>
```

Command

Syntax

var result = myNirva.Command(command)

Description

This method allows sending a command to Nirva via the given request object. It returns true in case of success and false if the command failed. Some Nirva commands may return a string value (output buffer). In this case, this value is available via the Result JavaScript property of the request object.

Parameters

command

This parameter is a string which contains the command to be sent to Nirva.

Return code

This command returns a boolean which is “true” when the command succeeded and “false” when the command failed. In the case of an error (return value of false) the information about the error will be found in the Error property of the request object as in the following example:

```
var service = myNirva.Error["service"];
var class = myNirva.Error["class"];
var code = myNirva.Error["code"];
var description = myNirva.Error["description"];
var info = myNirva.Error["info"];
```

See the Nirva error management in the Nirva documentation for further information about Nirva errors.

Example

```
var result = myNirva.command("NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|TEST|"));
if(result)
{
  var StringValue = myNirva.Result;
}
```

XMLCommand

Syntax

var result = myNirva.XMLCommand(command)

Description

This method allows sending a command to Nirva via the given request object. It returns true in case of success and false if the command failed. If the command succeeds the “Document” property of the request object is set with the resulting XML document. The calling script can then analyze the response and do the necessary processing such as updating HTML content.

The XML document always uses the Nirva simple xml schema.

Parameters

command	This parameter is a string which contains the command to be sent to Nirva.

Return code

This command returns a boolean which is “true” when the command succeeded and “false” when the command failed. In the case of an error (return value of false) the resulting Document property is empty and the information about the error will be found in the Error property of the request object as in the following example:

```
var service = myNirva.Error["service"];
var class = myNirva.Error["class"];
var code = myNirva.Error["code"];
var description = myNirva.Error["description"];
var info = myNirva.Error["info"];
```

See the Nirva error management in the Nirva documentation for further information about Nirva errors.

Example

The example below relies on an XSL file on the server side to generate an HTML table after a Nirva command is executed. The javascript code below sends a Nirva command which adds a line to the TEST table which is then transformed to HTML with the Tests/table.xsl file. The javascript code then replaces an existing HTML table in the browser with the one generated by the XSL.

```
if(myNirva.XMLCommand("NV_CMD=|OBJECT:TABLE_ADD_ROWS| NAME=|TEST| DATA=|one;A first
example| NV_XML_XSL=|Tests/table|"))) {
  // Retrieve new html table from the server's response
  var newTable = myNirva.Document.getElementById("test");
  // Replace the old html table with the new one
  var oldTable = document.getElementById("test");
  oldTable.parentNode.replaceChild(oldTable,newTable);
} else {
  alert("Cannot update table");
}
```

AsyncXMLCommand

Syntax

var result = myNirva.XMLCommand(command, callback)

Description

This method allows sending a command to Nirva via the given request object. The order is asynchronous so the method returns immediately but when the processing is finished, the callback function passed as parameter is called in order to process the resulting XML document.

The callback function is a JavaScript object member defined in this way:

```
myclass.prototype.NVCallback(request)
```

The request parameter will receive the NvAjax.Request object so the implementation of the NVCallback method can access the Document property of the NvAjx.Request object. This property contains the XML response from Nirva.

The XML document always uses the Nirva simple xml schema.

Parameters

command	This parameter is a string which contains the command to be sent to Nirva.
callback	This parameter is the class that implements the callback function "NVCallback".

Return code

AsyncXMLCommand doesn’t return any value. If there is a connection error, the function returns immediately, fills the Error property with the error “LOCAL:CONNECTION:100:Communication error” and calls the callback function. Otherwise, the function returns immediately.

Example

A typical example of the AsyncXMLCommand is the implementation of the Nirva application tests from the configuration tool. It runs asynchronously all the tests defined in a Nirva test set and displays the result of these tests on the screen.

Here are the involved files:

- *Nirva/Files/Config/application_testset.xsl*: instantiates and calls the NvAjax.Request object.
- *Nirva/Wroot/Config/testunit.js*: intermediate JavaScript code that processes the display and implements the callback function.

Rest

Overview

Representational state transfer (REST) or RESTful web services are one way of providing interoperability between computer systems on the Internet. REST-compliant web services allow requesting systems to access and manipulate textual representations of web resources using a uniform and predefined set of stateless operations. Other forms of web service exist, which expose their own arbitrary sets of operations such as WSDL and SOAP "Web resources" were first defined on the World Wide Web as documents or files identified by their URLs, but today they have a much more generic and abstract definition encompassing every thing or entity that can be identified, named, addressed or handled, in any way whatsoever, on the web. In a REST web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources. Using HTTP, as is most common, the kind of operations available include those predefined by the HTTP verbs GET, POST, PUT, DELETE and so on. By making use of a stateless protocol and standard operations REST systems aim for fast performance, reliability, and the ability to grow, by using reused components that can be managed and updated without affecting the system as a whole, even while it is running.

Nirva implementation or the REST connector is also called universal connector. It gives full control to the user to the input request data and to the response. The programmer defines the REST urls in the application.dsc file and provides procedures that will process input and output data returned to the user. The procedure will receive all input data including HTTP method, headers, content, etc…

Installation

The Rest connector doesn’t require any specific installation. In order to use it, one just needs to use a HTTP client.

Reference

In order to use the REST connector, the programmer must define the REST urls in the application.dsc with the name of a procedure that will process the data. Then the user, after a login step, can call this url.

Urls

The REST urls are defined in the application.dsc file in theREST_URLS section.

The section is composed of several entries of the form *url* = *nirvaparameters*.

url is the last part of the real url sent form browser. The real url must be of the form:

```
protocol//server:port/nv_rest/appname/url
```

where protocol is the protocol (“http:” or “https”), server:port is the server and the tcp/ip port (ex “localhost:1081”), appname is the application name and url is the url defined in the ORDER_URLS section.

For example if you have defined the following url in the MYAPP application.dsc file:

```
myurl/myrestcommand = NV_PROC=|perl:myrestcommand| PARAM1=|MyParam1|
```

Then you can call it by the following url:

```
http://locahost:1081/nv_rest/myapp/myurl/myrestcommand
```

If you want you can also add some extra parameters in this url:

```
http://locahost:1081/nv_rest/myapp/myurl/myrestcommand&myparam=test
```

The Nirva parameters given in the REST_URLS section have priority against the parameters with the same name given in the real url.

In order to have more friendly urls, you can use Nirva http alliases by replacing /nv_rest/appname with something else.

HTTP Methods

The REST connector accepts HTTP methods GET, PUT, POST and DELETE.

Input and output type

The input and output data are managed in string or file objects. In order to define the type of input or output, you must use respectively the NV_INPUT_TYPE and NV_OUTPUT_TYPE parameters in your REST url definition. This can be defined statically in the application.dsc file or dynamically when you call your URL.

The NV_INPUT_TYPE parameter allows storing the content of the http POST,PUT or DELETE request into a file or string object. For that it must be set to “FILE:object_name” or “STRING:object_name” where object_name is the name of the object that receives the content. If only the “FILE” or “STRING” value is given the default object name is respectively “INPUT_FILE” or “INPUT_STRING”. This object is created in the input container and cannot be persistent. For a file object, the parameters “EXTENSION”, “PREFIX” and “SUFFIX” can be used to control some parts of the file name.

The NV_OUTPUT_TYPE parameter is used to define the type and name of the object containing the output http data. It must be set to “FILE:object_name” or “STRING:object_name” where object_name is the name of the object containing the output data. If only the “FILE” or “STRING” value is given the default object name is respectively “OUTPUT_FILE” or “OUTPUT_STRING”. This object must be in the output container.

Typically, the procedure defined to process the URL takes input http content from the input object (file or string) and delivers output http content in the output object.

Example of a rest order defined in the application.dsc file with input and ouput:

```
testrest = NV_PROC=|perl:testrest| NV_INPUT_TYPE=|string:mystr| NV_OUTPUT_TYPE=|string|
```

The NV_INPUT_TYPE parameter is not taken in care if the input content type is multipart/form-data. In this case the input contains form data and will be managed differently (see later in this chapter).
The input data is not taken in care for the http GET method.

Procedure

The REST urls defined in the application.dsc file must contain a parameter NV_PROC that gives one or more procedures that will process the HTTP order. If no procedure is given, there will be no output. The procedure will receive 2 parameters: NV_HTTP_METHOD that contains the http method in uppercase and NV_HTTP_URL that contains the http url.

The input http headers are available with the command COMMAND:GET_HTTP_HEADER or GET_HTTP_HEADERS.

The input data is available in the object defined by the NV_INPUT_TYPE parameter (file or string).

The procedure have the responsibility to create the output data (if there is one) in the output object defined by the NV_OUTPUT_TYPE parameter.

The procedure can set http headers by using the COMMAND:SET_HTTP_HEADER command. It is not necessary to set the http Content-length header since this one is automatically set by the server.

Finaly the procedure can set the http return code by using the COMMAND:SET_HTTP_RET_CODE command. By default the return code is “200 OK” if no error or “500 Internal server error” otherwise.

The output data is taken in care after the call to the procedures defined in the NV_PROC parameter. If the NV_POST_PROC parameter is used, the post procedures cannot set the output data (except error object data, see error management). They can set http headers and return code.
.

Form processing

The HTML form input can be processed as described for the usual web browser access (see web syntax chapter) but the NV_INPUT_TYPE parameter must then be empty, otherwise all the input content is sent to the input type object without extracting form parameters.

Login

The login for the REST connector can be done in the same way than other connectors by using the NV_USER and NV_PASSWORD parameters.

Now, the REST connector introduces another way for login where the password is never transmitted directly from client to server. It is a good practice to log-in in this way. This login occurs in 2 steps:

First the client sends a request to the server using the following URL with the HTTP HET method:

```
protocol//server:port/nvs?gettoken/&NV_APPLICATION=appname&NV_USER=username
```

where appname is the application name and username is the user name.

Then the server answers with a couple of private/public tokens separated by a “:” character in the body of the response:

```
PrivateToken:PublicToken
```

The user must then compute an identification string on the following way:

```
Ident = sha256_up(sha256_up(md5_up(user)) + sha256_up(md5_up(password)) + PrivateToken)
```

Where sha256_up is the SHA256 hexadecimal hash converted to uppercase and md5_up is the MD5 hexadecimal hash converted to uppercase.

Once this has been done, the user has 10 seconds to send the REST order with login. For that he must add the following parameters to the REST url:

- NV_IDENT – This the Ident string computed by the client as described above
- NV_TOKEN – This is the public token just returned by the server as described above

The rest order must come from the same IP address than the request for the private/public tokens.
The 2 steps login is available only with internal and local application security.

Session persistance

By definition the REST order is stateless so an order creates and closes a nirva session. Now it is possible to let the session opened by explicitly setting the NV_CLOSE_SESSION parameter to “NO”. If the session is let opened the user can send several rest orders to the session using the NV_SESSION_ID parameter. If a valid NV_SESSION_ID parameter is given then the order doesn’t close the session.

Permission

In order to use the REST connector, the user must have the “REST_CONNECTOR” security permission defined in the application security.

Encoding

Nirva accepts both iso-8859-1 and utf-8 encodings. It is advised to use utf8 encoding for the rest connector.

The input default encoding is the nirva encoding (so iso-8859-1 or utf8 as defined in the nirva configuration). The input encoding can be changed by giving the encoding in the content-type http header (ex: “plaint/text; charset=utf-8”).

The default input content type is plain/text.

Output encoding is under control of the programmer. It is best practice to use utf-8.

Error management

In case of error (internal error or if the rest procedure returns an error), the rest connector returns by default the HTTP “500 internal error” error code. Some extra information about the error is available in the body of the server response in text format and utf-8 encoding.

It’s possible to change this default behaviour.

First you can take full control of the output. For that let the procedure return no error and set yourself the http return code (COMMAND:SET_HTTP_RET_CODE command) and the output data (in the output object as described below).

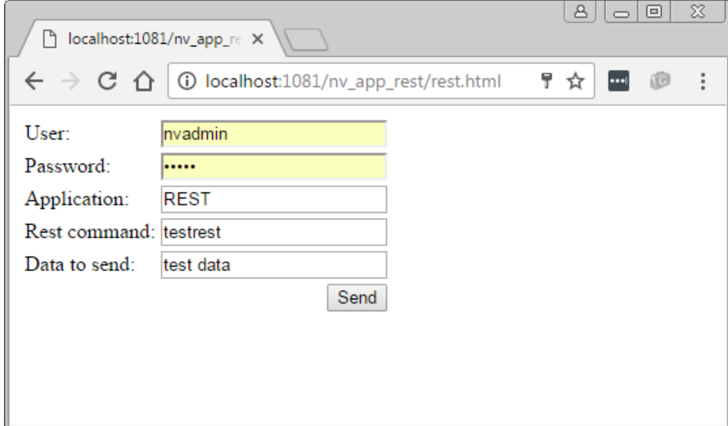
You can also let the procedure return an error and create a string object named “ERROR” in the output container. The server will then use the content of this object as body of the HTTP response.

Example

After installing Nirva, an example of the REST connector is available in the Nirva Sdk/Connectors/rest directory. This is a nirva application named “REST”. Just install it as an application, start it and call locally the following url from your browser:

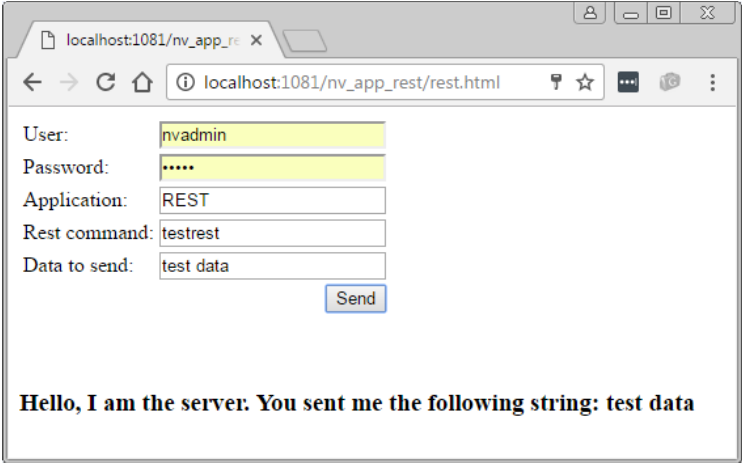
http://localhost:1081/nv_app_rest/rest.html

This will display the following screen:



Enter the user name (use nvadmin in order to have administrator rights), password, some data to send to the server and press “Send” button. Then your data will be displayed in the nirva console (if you have started nirva in colsole mode) and the server will send you back a message:

```
C:\Windows\system32\cmd.exe
09:58:27 2CAB510E0D       NVDEF --- End procedure: (java:System/getjavaversion)
09:58:27 2CAB510E0D       NVDEF --- End procedure: (Config/system_parameters)
09:58:30 2CAB510E0D       NVDEF --- Start procedure: (Config/system_application_l
ist)
09:58:30 2CAB510E0D       NVDEF --- End procedure: (Config/system_application_lis
t)
09:58:34 2CAB510E0D       NVDEF --- Start procedure: (Config/system_application_d
etail)
09:58:34 2CAB510E0D       NVDEF --- End procedure: (Config/system_application_det
ail)
09:58:35 2CAB510E0D       NVDEF --- Start procedure: (Config/system_application_p
ackage)
09:58:37 2CAB510E0D       NVDEF --- End procedure: (Config/system_application_pac
kage)
09:58:37 2CAB510E0D       NVDEF SYSTEM:OBJECT:GET
10:09:12 15E414C810        REST --- Start procedure: session_open
10:09:12 15E414C810        REST --- End procedure: session_open
10:09:12 15E414C810        REST --- Start procedure: perl:testrest
Hello, I'm the testrest perl procedure
I receive the following input string: test data
10:09:12 15E414C810        REST --- End procedure: perl:testrest
10:09:12 15E414C810        REST SYSTEM:MISC:REST
10:09:12 15E414C810        REST --- Start procedure: session_close
10:09:12 15E414C810        REST --- End procedure: session_close
```



If you don't use the nvadmin user be sure that you have set the "REST_CONNECTOR" security permission in your user's role.

How it works?

In the application.dsc file you can see the following line at the end:

```
// THis section defines the rest connector urls for the application
[REST_URLS]
testrest = NV_PROC=|perl:testrest| NV_INPUT_TYPE=|string:mystringobject| NV_OUTPUT_TYPE=|string|
```

This defines a rest url named "testrest" (will be called by http://locahost/nv_rest/rest/testrest)

This url just gives the name of the procedure (perl:testrest) that processes the order and the name and type of input and output objects (the name of the output is "OUTPUT_STRING" by default). Here is the code of the procedure:

```
# Use:
# http://localhost:1081/nv_rest/rest/testrest

NV::SetEncoding("UTF-8");
```

```
NV::SetErrorMode("SCRIPT");
print "Hello, I'm the testrest perl procedure\n";
NV::Command("NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|mystringobject| NV_NO_ERROR=|YES|");
$Input = $NV::RESULT;
print "I receive the following input string: $Input\n";

# Uncomment next lines to display rest url
#NV::GetParameter("NV_HTTP_URL");
#print "Http url: $NV::RESULT\n";

# Uncomment next lines to display rest method
#NV::GetParameter("NV_HTTP_METHOD");
#print "Http method: $NV::RESULT\n";

# Create output string
if(!NV::Command("NV_CMD=|OBJECT:CREATE| TYPE=|STRING| NAME=|output_string|")){exit 1;}
$Output = "Hello, I am the server. You sent me the following string: $Input";
if(!NV::Command("NV_CMD=|OBJECT:STRING_SET_VALUE| NAME=|output_string|
VALUE=|$Output|")){exit 1;}

# Set Content type (default is plain/text and same charset than server)
if(!NV::Command("NV_CMD=|COMMAND:SET_HTTP_HEADER| HEADER=|Content-Type|
VALUE=|plain/text; charset=UTF-8|")){exit 1;}

# uncomment next lines to generate an error
#NV::SetErrorEx("SYSTEM", "SECURITY", 105);
#exit 1;
```

The client itself is written in java script. The code is inside the Wroot/rest.html file:

```
<!DOCTYPE html>
<!-- http://localhost:1081/nv_app_rest/rest.html -->
<html>
<head>
<meta charset="UTF-8">
<title></title>
<meta name="description" content="">
<script src="md5.js" type="text/javascript" charset="utf-8"></script>
<script src="sha256.js" type="text/javascript" charset="utf-8"></script>
<script type="text/javascript" charset="utf-8">

var client = new XMLHttpRequest();
var datatosend = "";
var user = "";
var password = "";
var application = "rest";
var restcommand = "testrest";

function ProcessForm()
{
  user = document.frm1.user.value;
  password = document.frm1.password.value;
  application = document.frm1.application.value;
  datatosend = document.frm1.data.value;
  restcommand = document.frm1.restcommand.value;
  loadDoc("/nvs?gettoken&NV_APPLICATION=" + application + "&NV_USER=" + user,
```

```
on_get_token, "GET");
}

function loadDoc(url, cFunction, Method, data)
{
  var xhttp;
  xhttp=new XMLHttpRequest();
  xhttp.onreadystatechange = function()
  {
          if (this.readyState == 4)
          {
                  if(this.status == 200)
                          cFunction(this);
                  else
                          alert("The request did not succeed!\n\nThe response status was:
" + this.status + " " + this.statusText + "\n" + this.responseText);
          }
  };
  xhttp.open(Method, encodeURI(url), true);
  xhttp.setRequestHeader("Content-Type", "text/plain; charset=utf-8");
  xhttp.send(data);
}

function on_get_token(xhttp) {
  var res = xhttp.responseText.split(":");
  var PrivateToken = res[0];
  var PublicToken = res[1];
  var Ident = SHA256(SHA256(hex_md5(user).toUpperCase()).toUpperCase() +
SHA256(hex_md5(password).toUpperCase()).toUpperCase() +
PrivateToken.toUpperCase()).toUpperCase();
//        document.getElementById('token').innerHTML = "PrivateKey = " + PrivateToken +
"<br/>PublicKey = " + PublicToken + "<br/>Ident = " + Ident;
  loadDoc("/nv_rest/" + application + "/" + restcommand + "&NV_TOKEN=" + PublicToken +
"&NV_IDENT=" + Ident, on_get_data, "POST", datatosend);
}

function on_get_data(xhttp) {
  document.getElementById('response').innerHTML = xhttp.responseText.toHtmlEntities();
}

// Convert a string to HTML entities
String.prototype.toHtmlEntities = function() {
    return this.replace(/./gm, function(s) {
        return "&#" + s.charCodeAt(0) + ";";
    });
}
</script>
</head>

<body>
<form name="frm1" action="javascript:ProcessForm();">
  <table>
          <tr><td>User: </td><td><input type="text" name="user"></td></tr>
          <tr><td>Password: </td><td><input type="password" name="password"></td></tr>
          <tr><td>Application: </td><td><input type="text" name="application"
value="REST"></td></tr>
          <tr><td>Rest command: </td><td><input type="text" name="restcommand"
value="testrest"></td></tr>
```

```
          <tr><td>Data to send: </td><td><input type="text" name="data"></td></tr>
          <tr><td></td><td style="text-align:right"><input type="submit"
value="Send"></td></tr>
  </table>
</form>
<script type="text/javascript" charset="utf-8">
//document.write("<p id='token'></p>");
document.write("<br/><br/><h3 id='response'></h3>");
</script>
</body>
</html>
```

This script identifies the user in 2 steps as described below.

Tools

nvcc

nvcc is a console mode client tool for sending a set of commands to a NIRVA server. The commands must reside in an input file given as parameter.

nvcc is built using the NIRVA client library nvc. Each nvcc instance creates an nvc request. See the nvc chapter for further information about an nvc request.

nvcc is installed on the Nirva/Bin directory.

nvcc can be used as stand alone client without having to install the entire NIRVA environment.

The only necessary files are “`nvcc.exe`” (or “`nvcc`” for UNIX platforms) and the nvc library “`nvc.dll`” (or “`libnvc.a`” for AIX or “`libnvc.sl`” for HPUX PA-RISC or “`libnvc.so`” for LINUX, SOLARIS and HPUX Itanium).

Command line syntax

The nvcc command line usage is the following one:

```
nvcc [options] parameters
```

Parameters

Any command line string that is not an option is considered as a parameter. A parameter is automatically transformed by nvcc into a request variable so a parameter can be accessed directly from any NIRVA command found in the nvcc input file.

The created request variables corresponding to the parameters are named Param1, Param2, Param3, etc... Where Param1 corresponds to the first parameter, Param2 to the second and so on.

Options

Here is the list of nvcc options:

`-i <inputfile>`	Name of the input file containing NIRVA commands. If this parameter is not provided, nvcc uses a default name “input.txt”. The input file contains a succession of text lines corresponding to NIRVA commands. The input file can also contain some comments or some specific nvcc print orders. Please consult the next chapter for further information about the nvcc input file format.
`-r <procedure>`	Name of Nirva procedure. If this parameter is provided, nvcc doesn't use the input file but directly asks the server to execute the given procedure.
`-z <Nirva command>`	Nirva command. If this parameter is provided, nvcc doesn't use the input file or procedure but directly executes the given command.
`-d <display mode>`	Display mode is a string composed of letters c, e, p and n. If ‘n’ is in this string, nvcc runs in a complete silent mode. If ‘c’ is in the string, nvcc displays commands. If ‘e’ is in the string, nvcc displays errors. If ‘p’ is in the string, nvcc displays the specific nvcc print output found in the input file. The default display mode is “ep” which displays errors and specific nvcc output print orders found in the nvcc input file.
`-a <NIRVA server address>`	This is the TCP/IP address of the NIRVA server to connect. This can be followed by `:PortNumber` if the port number of the NIRVA server is different than the default one (1081 for HTTP and 1082 for HTTPS). For example, `-a 153.20.30.121:1256` will connect the NIRVA server found at address 153.20.30.121 and port 1256. The remote machine name can be used in place of the real TCP/IP address. The default server address is 127.0.0.1:1081 (local server). If the server address is followed by the “(SSL)” string (in brackets), nvcc will use the HTTPS protocol to communicate to the NIRVA server. The HTTPS protocol must has been enabled on the server side. When using HTTPS, the default port is 1082. If the server address is followed by the “(SSO)” string (in brackets) and if the Nirva application has been configured to authenticate users via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO. The SSO string may also contain an optional name for the SSO principal (ex “(SSO:myprincipal)”) when using Kerberos protocol. It’s possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers.

This allows a simple load balancing solution without requiring a load balancing box. This is a simple load balancing solution and the connected servers must all run.
Example: `Server="192.168.20.17;192.168.20.19:80"` will connect one of the servers 192.168.20.18 or 192.168.20.18 on the port 80.
A proxy server can also be defined in a single address. At this time the format is the following:
`proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be "http" or "https"; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

`-k <certificate>`

Client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.

`-s <session ID>`

NIRVA session to connect. If not provided, NIRVA creates a new session.

`-p <application name>`

Name of the NIRVA application to connect. If not provided, nvcc connects to the NIRVA default Application (NVDEF).

`-u <user name>`

NIRVA application user name. If not provided, nvcc uses to the NIRVA application default user name.

`-w <user password>`

NIRVA application user password. If a previous login required the user to change its password, the new password can be given after the old password, separated from it by a semicolon character (';').

`-o <open!close>`

Name of the session open and close procedures to use for the new session. This parameter has meaning only if the –s option is not given so only for a new session. The open and close procedure names are separated by a "!" character. If one is omitted or if the -o parameter is not provided NIRVA uses "session_open" and "session_close" as default names. In order to not execute any open procedure, the open procedure name must be set "NV_SESSION_OPEN_NONE". In order to not execute any close procedure, the close procedure name must be set "NV_SESSION_CLOSE_NONE".

`-c`	No close option. If this option is given, nvcc doesn't close the NIRVA session when exiting. The default is to close the session.
`-t`	No stop on error option. If this option is given, nvcc doesn't exit in case of error on a command. The default is to stop in case of error.
`-n`	Unicode mode. Tells if client is Unicode. When this parameter is set to “YES”, all the data sent to the server are supposed to be UTF-8 encoded and all data coming from server are automatically converted to UTF-8 by the server. When not in Unicode mode, the client is supposed to work in ISO-8859-1 character set.
`-b <test string>`	Test mode. This option allows running several times the given input file. The test string has the format “`l=numloop:r=numthread`” where `numthread` is the number of simultaneous threads to be created and `numloop` is the number of times each thread will execute the input file. In test mode, nvcc first connects a NIRVA session to each thread, then executes all the threads simultaneously and finally disconnects each thread from the NIRVA sessions. nvcc then displays the execution time of the real processing.
`-h`	Display a help screen when used without any other parameter or option.
`-m <max message size>`	Maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise between performance and memory management on the client side. The minimum value is 10 kilobytes.

Input file format

The input file contains a succession of line. Any blank line is ignored. The other lines can be:

A NIRVA command as described in the “Nirva command syntax” chapter in this documentation. If a NIRVA command ends with the ‘/’ or ‘\’ characters, nvcc considers that the command continues on the next line. If a NIRVA command has to be split on several lines, it must be cut between two parameters.

A comment. A comment is a line starting with ‘;’, ‘/’, or ‘\’ characters.

An nvcc output print order. This specific nvcc output is printed if the –d option has been set with the ‘p’ display mode (this is the default). 2 kind of nvcc output print orders are available:

`nvcc::printf <string to print to the screen>`

The line must start with `nvcc::printf` and followed by a space character. Any character that follows is printed to the screen when nvcc decodes the line. In order to print a new line character, the string to print must contain “\n”. In the same way, in order to print a tabulation character, the string to print must contain “\t”.

`nvcc::printdata`

The line must start with `nvcc::printdata`. This nvcc print order tells nvcc to display the output buffer of the previous NIRVA command. This is a good way to display some command results.

Example

This example is named filecopy.txt. It just makes a file copy but by passing the file to a NIRVA server as a file object.

```
; NIRVA input file for tests
; 03/07/2002

nvcc::printf file copy using nirva server\n\n
nvcc::printf usage: nvcc -i filecopy.txt sourcefile destinationfile
nvcc::printf \n\n

// Create local object source
NV_CMD=|LOCAL:OBJECT:CREATE| NAME=|Source| TYPE=|FILE| FILENAME=|#param1|
NV_CMD=|LOCAL:VARIABLE:GET| NAME=|param1|
nvcc::printf \nsource file:
nvcc::printdata

// Create local object source
NV_CMD=|LOCAL:OBJECT:CREATE| NAME=|Destination| TYPE=|FILE| FILENAME=|#param2|
NV_CMD=|LOCAL:VARIABLE:GET| NAME=|param2|
nvcc::printf \ndestination file:
nvcc::printdata

// Do the copy by sending the file to the server
NV_CMD=|OBEJCT:SEND| NAME=|Source|
NV_CMD=|OBEJCT:GET| NAME=|Source| LNAME=|Destination|

nvcc::printf \n\nAll OK\n
```

In order to run this example, create a file name test1.txt, put some content into it and type the following command from the command prompt:

```
nvcc -i filecopy.txt test1.txt test2.txt
```

This should copy the file test1 to the file test2.txt. The local NIRVA server must be started.

Here is the displayed result:

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd nirva\bin

C:\Nirva\Bin>nvcc -i filecopy.txt test1.txt test2.txt
file copy using nirva server

usage: nvcc -i filecopy.txt sourcefile destinationfile


source file: test1.txt
destination file: test2.txt

All OK

C:\Nirva\Bin>
```

Test mode

nvcc can be launched in test mode by using the “-b” option. This option allows defining the number of threads and the number of loops.

In test mode, nvcc processes in the following way:

- Create the requested number of threads.
- For each created thread, establishes a connection to the NIRVA server by sending a SYSTEM MISC NOP command.
- Waits for all the threads to be connected to the server.
- Sends an event to all the threads for them to start processing. In this way, all the threads start processing simultaneously. The thread processing consists of executing the input file in loop mode with the number of loops given by the “–b” option.
- Waits for all threads to end their processing.
- Disconnects the threads from the NIRVA server.
- Waits for all threads to be disconnected.
- Reports the time of real processing (without the ramp up and ramp down periods).

The reporting gives the total number of requests, the total time and the number of requests per second. A request is an execution of the script (the input file). The number of requests per second is different than the number of commands per second because a single script may contain an important number of commands.

In this way, the processing time for a given operation can be easily measured.

Here is an example of nvcc test mode reporting:

```
Cmd
C:\nirva\Bin>nvcc -i test.txt -b l=100:r=2


Connecting sessions...
Waiting for all sessions to be connected...
Start processing commands...
Waiting for end processing...
Processing has finished...
Disconnecting sessions...
Waiting for all sessions to be disconnected...

TEST RESULTS

Number of threads = 2
Number of loops = 100
Total number of requests = 200
Processing time = 430 ms
Rate = 465 requests / second

Press enter to continue

C:\nirva\Bin>
```

This example executes 2 threads in parallel, each of them running 100 times the script test.txt.

nvcc command files

Here are a set of nvcc standard command files that are delivered in standard in the NIRVA Bin directory.

This chapter describes these nvcc command files.

The description supposes that the command files are launched from the NIRVA server machine. If they have to be run from another machine, one just needs to use the nvcc "–a" option to set the TCP/IP address. In the same way, it's assumed that the default user of the default NIRVA application has the rights to run the commands of the command files. If not, one must use the nvcc options to connect the correct user of the correct application.

system_package.txt

The system_package.txt command file gives a standard way to package some system files. Packaging system files means to create a package file that can be used later by the customer to install the package.

Usage:

```
nvcc -i system_package.txt <Package file> [<Description file>]
```

Where `Package file` is the name of the local package file created by the command. `Description file` is an optional parameter that tells NIRVA which package description file to use for doing the packaging. The default value is "package.lst".

See the SYSTEM PACKAGE PACKAGE command description for further information about system packaging.

system_install.txt

The system_install.txt command file gives a standard way to install a server package (for example a patch).

Usage:

```
nvcc -i system_install.txt <Package file>
```

Where `Package file` is the NIRVA package to install.

See the SYSTEM PACKAGE INSTALL command description for further information about system package installation.

service_package.txt

The service_package.txt command file gives a standard way to package a service. Packaging a service means to create a package file that can be used later by the customer to install the service.

Packaging a service is reserved to service providers.

Usage:

```
nvcc -i service_package.txt <Service name> <Package file> [<Description file>]
```

Where `Service name` is the name of the service to package and `Package file` is the name of the local package file created by the command. `Description file` is an optional parameter that tells NIRVA which package description file to use for doing the packaging. The default value is “package.lst”.

See the SYSTEM SERVICE PACKAGE command description for further information about service packaging.

service_install.txt

The service_install.txt command file gives a standard way to install a service. It first stops it before doing the installation.

Usage:

```
nvcc -i service_install.txt <Service name> <Package file> [<WaitTime>]
```

Where `Service name` is the name of the service to install and `Package file` is the NIRVA package to install. The package file is delivered by the service provider. `WaitTime` is the maximum number of seconds for the command to wait for the service to be stopped. The default value is 60 seconds.

See the SYSTEM SERVICE INSTALL command description for further information about service package installation.

application_package.txt

The application_package.txt command file gives a standard way to package an application. Packaging an application means to create a package file that can be used later by the customer to install the application.

Packaging an application is reserved to application providers.

Usage:

```
nvcc -i application_package.txt <Application name> <Package file> [<Description file>]
```

Where `Application name` is the name of the application to package and `Package file` is name of the local package file created by the command. `Description file` is an optional parameter that tells NIRVA which package description file to use for doing the packaging. The default value is “package.lst”.

See the SYSTEM APPLICATION PACKAGE command description for further information about application packaging.

application_install.txt

The application _install.txt command file gives a standard way to install an application. It first stops it before doing the installation.

Usage:

```
nvcc -i application_install.txt <Application name> <Package file>
```

Where `Application name` is the name of the application to install and `Package file` is the NIRVA package to install. The package file is delivered by the application provider.

See the APPLICATION SERVICE INSTALL command description for further information about application package installation.

webservice_package.txt

The webservice_package.txt command file gives a standard way to package a web service. Packaging a web service means to create a package file that can be used later by the customer to install the web service.

Packaging a web service is reserved to web service providers.

Usage:

```
nvcc -i webservice_package.txt <Web service name> <Package file> [<Description file>]
```

Where `Web service name` is the name of the web service to package and `Package file` is name of the local package file created by the command. `Description file` is an optional parameter that tells NIRVA which package description file to use for doing the packaging. The default value is “package.lst”.

See the SYSTEM WEBSERVICE PACKAGE command description for further information about web service packaging.

webservice_install.txt

The webservice_install.txt command file gives a standard way to install a web service.

Usage:

```
nvcc -i webservice_install.txt <Web service name> <Package file>
```

Where `Web service name` is the name of the web service to install and `Package file` is the NIRVA package to install. The package file is delivered by the web service provider.

See the WEBSERVICE SERVICE INSTALL command description for further information about web service package installation.

getmachine.txt

The getmachine.txt command file returns the unique machine identifier used to create the license files. It also returns the number of core CPUs detected by Nirva.

Usage:

```
nvcc -i getmachine.txt
```

Where `License file` is the NIRVA license to install.

See the SYSTEM LICENSE INFO command description for further information about the unique machine identifier.

license_install.txt

The license_install.txt command file gives a standard way to install a license file. The license file is given the service provider for NIRVA external services.

Usage:

```
nvcc -i license_install.txt <License file>
```

Where `License file` is the NIRVA license to install.

See the SYSTEM LICENSE INSTALL command description for further information about license installation.

testxml.txt

The textxml.txt command file allows testing the NIRVA XML connector. It takes an input XML file and delivers an output XML.

Usage:

```
nvcc –i testxml.txt <Input XML file> <Output XML file> [<XSL filter>]
```

Where `Input XML file` is the input XML, `Output XML file` is the result XML file and `XSL filter` is the optional XSL server file for changing the input XML to a NIRVA acceptable XML if the input file doesn't respect the NIRVA XML grammar (it corresponds to the NV_XML_XSL_IN command parameter).

See the XML connector description for an example of using testxml.txt.

testsoap.txt

The textsoap.txt command file allows testing the NIRVA SOAP connector. It takes an input SOAP file and delivers an output SOAP.

Usage:

```
nvcc –i testsoap.txt <Input SOAP file> <Output SOAP file>
```

Where `Input SOAP file` is the input SOAP and `Output SOAP file` is the result SOAP file.

See the SOAP connector description for an example of using testsoap.txt.

testwebs.txt

The textwebs.txt command file allows testing the NIRVA WEB service connector. It takes an input SOAP message file and delivers the output SOAP message.

Usage:

```
nvcc –i testwebs.txt <WebService name> <WebService operation> <Input SOAP message file>
<Output SOAP message file>
```

Where `WebService name` is the name of the web service to connect `WebService operation` is the name of the web service operation to launch, `Input SOAP message file` is the input SOAP and `Output SOAP message file` is the result SOAP file.

nvl

Overview

nvl is a WINDOWS tool for managing licenses. It’s dedicated only to NIRVA external service providers or application providers.

It allows to modify or to create NIRVA license files. The service and application providers have only access to their own license keys.

NIRVA maintains the license keys in a file named “nirva.lsc” that resides in the Nirva bin directory. A license key is composed of the following items:

The service name is the name of the service concerned by this license.

The private service ID is a unique identifier known only by the service or application provider. This is the service private identifier that should never be given to anybody.

The license key name is the name of the license key.

The license key value is an optional value associated to the license key. For example, it should be the maximum number of users.

The license key date is an optional expiration date of the license key.

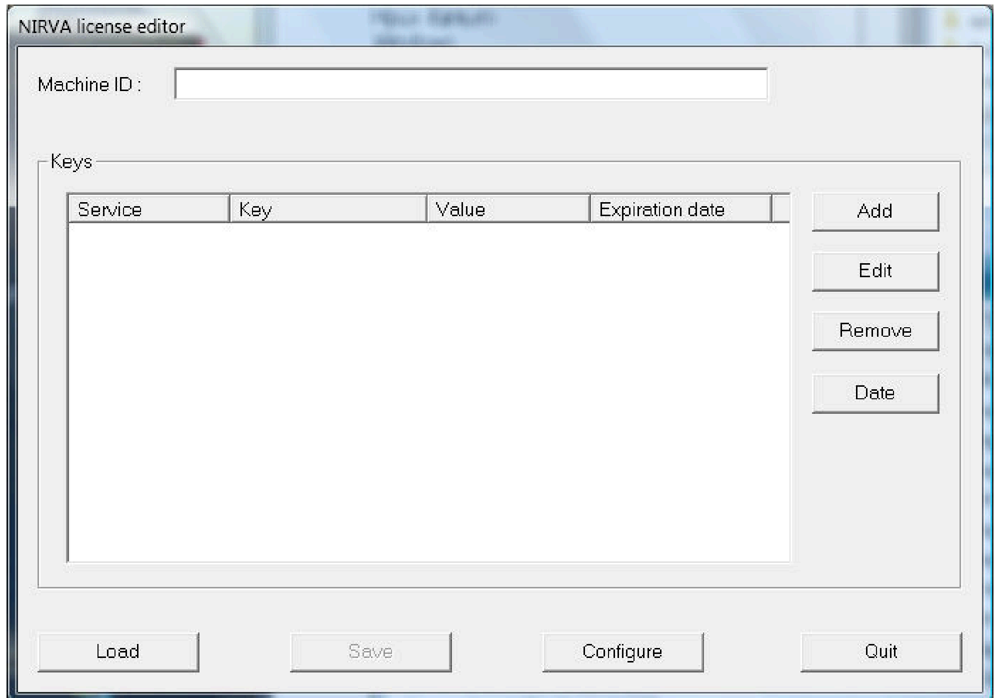
The NIRVA license manager uses a unique machine identifier to assume that any installation has unique license keys.

Installation

The nvl executable (named nvl.exe) is delivered on the NIRVA Bin directory on all NIRVA installations including UNIX ones. But in fact, the executable is a WINDOWS executable. For providers who only install a UNIX version, they must copy the nvl.exe file into a WINDOWS computer. No other file is required.

Using nvl

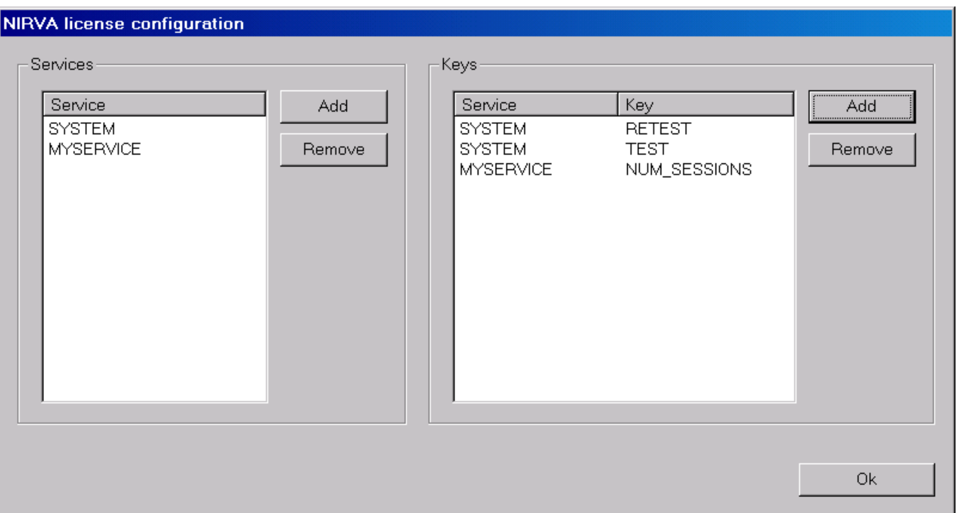
When launching “nvl.exe”, the following window is displayed:



It displays the list of license keys found in a NIRVA license file. The “Machine ID” is the unique machine identifier that must be given by NIRVA customers. This identifier can be get from the NIRVA configuration tool in the system/licenses menu. The machine identifier is 32 characters long.

The Load and Save buttons are used to load or save a license file. When a license file has been created or modified, it can be delivered to the final customer for him to install it with the NIRVA configuration tool or an nvcc command.

The configure button allows to register the list of possible license keys. It displays this list in the following window:



“Services” is the list of possible service names. When registering a new service, the user must enter the service name and service ID. The service name is the one known by NIRVA when installing a service and the service ID is the unique private service identifier generated by NIRVA when creating a new service. The service ID must be kept secret by the service provider. The service ID is 32 characters long.

“Keys” is the list of possible service keys. A service key is composed at minimum of a service name and a key name. The service name must have been registered previously in the “Services” list.

When the configuration has been done, the tool creates 2 files respectively named keys.lsc and services.lsc in the NIRVA Bin directory if NIRVA has been installed or in the current directory otherwise.

The keys.lsc and services.lsc files should not be delivered in any case to final users or they will be able to create some of your licenses.

nvd

nvd is a console client tool that helps debugging applications. It creates or connects an existing or future session and receives debug data from it.

This tool runs only on the local nirva server.

Command line syntax

The nvd command line usage is the following one:

```
nvd [options]
```

Here is the list of nvd options:

Option	Description
`-s <session ID>`	Nirva session to attach. If not provided, a new session is created. Instead of a session ID a value "next:type" can be given. At this time nvd connects to the next session of the given type. Type can be one of: any, client, web, webs, xml, soap, transaction, scheduler, named, listener, thread, internal, mq. "-s next" is the same as "-s next:any”.
`-l <debug level>`	Debug level. This can be 1, 2 or 3. The default is 1.
`-p <Nirva http port>`	This is the TCP/IP port of the NIRVA Http local server. The default is 1081.
`-a <application name>`	Name of the NIRVA application to connect. This parameter has meaning only when a session ID is not given. The default is nvdef. If "-s next" is used and -a parameter is not

	given nvd will connect the next given type session whatever the application.
`-u <user name>`	Application user name. This parameter has meaning only when -s parameter is not used. The default is nvdef.
`-w <user password>`	User password. This parameter has meaning only when -s parameter is not used.
`-o <open!close>`	Name of the session open and close procedures. The open and close procedure names are separated by the ! character. If one is omitted or if the -o parameter is not provided NIRVA uses session_open and session_close as default names. In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”. In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”. This parameter has meaning only when -s parameter is not used.
`-r <procedure>`	Name of Nirva procedure. If this parameter is provided, nvd requests nirva to execute the given procedure. This parameter is not used when the next session is to be attached.
`-z <Nirva command>`	Nirva command. If this parameter is provided, nvd requests nirva to execute the given command. This parameter has priority on the -r parameter (-r is then ignored). This parameter is not used when the next session is to be attached.
`-c`	No close option. If this option is given, nvd doesn't close the NIRVA session when exiting. The default is to close the session. This parameter has meaning only when -s parameter is not used.
`-h`	Display a help screen.

Examples

Create a session, run a procedure and exit the session:

```
C:\>nvd -r perl:ptest -l 3
Waiting for a session to be attached...
Session 37EBFF35D3 has been attached

Current command stack:
Command: SYSTEM:MISC:NOP(V)

15:16:38 start SYSTEM:MISC:NOP(V)
Debug: parameters:
        LEVEL = 3
        NV_POST_PROC =
        NV_PROC = session_open
        PID = 5100
        TYPE =
15:16:38 --- Start Native procedure: session_open
15:16:38 --- End native procedure: session_open
15:16:38 end   SYSTEM:MISC:NOP(V)
15:16:38 start SYSTEM:MISC:NOP(C)
15:16:38 end   SYSTEM:MISC:NOP(C)
Debug: output buffer: 168
15:16:38 start SYSTEM:SESSION:CLOSE(C)
15:16:38 end   SYSTEM:SESSION:CLOSE(C)
Debug: output buffer: 168
15:16:38 --- Start Native procedure: session_close
15:16:38 --- End native procedure: session_close

Console has been closed by nirva
```

Other examples:

```
Connect the next web session with level 3 (open the nirva config to see it):
nvd -s next:web -l 3


Connect the next threaded session for application myapp with level 2:
nvd -s next:thread -l 2 -a myapp


Connect a specific session with level 1:
nvd -s CB47B271A7
```

Services

What is a Nirva service

A NIRVA service is a library, a dotnet assembly (windows only) or a java class loaded by the NIRVA server that implements a set of NIRVA commands.

When sending a command to NIRVA, the command has a parameter named “NV_SERVICE” that tells NIRVA which service must process the command. If NV_SERVICE is “SYSTEM”, NIRVA itself will process the command into its internal service. Otherwise, NIRVA tries to locate and load the requested service and forward the command to it.

Here are the components of a service:

- A library or assembly file or a java class that implements the service functionality.
- A description file that give NIRVA some information about the service.
- A registry that maintains specific persistent service data.
- A documentation.
- A configuration tool.
- Some service specific log files.
- Some NIRVA procedures specific to the service.

NIRVA provides a framework that encapsulates services. In this way, much functionality is directly taken in charge by NIRVA itself. This discards service providers to implement this functionality. The possibility to communicate from service to service also allows service providers to use functionality from other services or from NIRVA itself.

NIRVA takes in charge the following service functionality:

- Automatic creation of service source files.
- Display of the service configuration from the main NIRVA configuration tool or from a NIRVA command.
- Display of the service documentation.
- License management.
- Security for accessing service functionality.

- Session management.
- Log files.
- Calling of service procedures.
- Service starting and stopping.
- Installation.
- Service registry access.
- Display of service information and state.
- Multithreading.
- Error management.

NIRVA has been made to really facilitate the encapsulation of technology into external services. Service providers just need to concentrate on the functionality itself.

Requirements

Library

A C++ service is a dll under windows and a shared library under UNIX. A Java service is a java class instanced by Nirva. A dotnet service is a dotnet class instanced by Nirva.

Multithreading

Since a service doesn't need to be multithread, it must be compiled with the compiler multithread options. If the service is not really multithread, it will be mounted to NIRVA (with the NIRVA SYSTEM SERVICE MOUNT command) without the multithread option so NIRVA will serialize any service commands.

Compilation

The service must be written in C++ (or any other language able to create a shared library) or Java or any dotnet language for a dotnet service.

A C++ service must be always compiled with the platform native compiler. This is very important because some compilers may generate incompatible code with other services or with NIRVA itself. For example, if the gcc compiler is used under SUN SOLARIS, this creates incompatible libraries with SOLARIS libraries compiled with the native CC compiler. Also for IBM AIX, it's not recommended to use the gcc compiler because it may create problems in exception management in multithread environment.

Here are the specific platform native compilers certified with NIRVA:

Microsoft:	Visual C++
AIX:	xlC_r (VisualAge)
SOLARIS:	CC
LINUX:	gcc
HPUX:	aCC

A Java service is compiled with a java compiler. The nirva.jar located in the Nirva/Bin directory must be in the class path.

A Dotnet service is compiled with the compiler of the chosen language (ex csc for c#). The nirvadn.dll assembly located in the Nirva/Bin directory must be added in the reference list.

Entry point

A C++ service must implement a function named “NvsCommand” that returns an integer and accept only one parameter that is a pointer on a C++ class of type NvServiceCommand. The prototype of this class is declared in the file “nvsext.h” that can be found in the Nirva/Sdk/Service directory.

The NvsCommand function is described later in this chapter.

A Java service must deliver a class that contains at least 3 methods: Command, Init and Exit.

A Dotnet service must deliver a class that contains at least 3 methods: Command, Init and Exit.

Names

The service name, the class and command names, the permission names and the error class names defined in the service should use only alphanumeric characters (numbers and letters) and the underscore character.

Description file

The external service description file is a text file named “service.dsc”. The description file must reside in the service “Files” directory.

The external service description file is not mandatory but it gives important information about error messages and security.

The description file is composed of well defined sections. A new section begins with a new line starting with the ‘[‘ character followed by the section name and terminating with the ‘]’ character.

Each section contains a succession of lines with a meaning depending of the section itself.

The description file can include comments. A comment is a line starting with ‘;’, “//” or “\\”.

Any blank line is ignored.

Here are the description file available sections:

INFO	General service information.
SETTINGS	General settings.
PERMISSIONS	Service security permissions.
COMMAND_CLASSES	List of service command classes.
COMMAND_CLASS	List of commands for each command class.
ERROR_CLASSES	List of service error classes.
ERROR_LANGUAGES	List of service error languages.
ERROR_CLASS	List of error codes and description for each error class.

Here is an example of a description file for the service “MYSERVICE”:

```
// service.dsc : description file
// NIRVA service
// This file should reside in the NIRVA/Services/MYSERVICE/Files directory

// This file contains the MYSERVICE NIRVA service description
// NIRVA tries to read it when loading the service
// The service.dsc file should be installed in the service File directory
// This file is not required but is very usefull for NIRVA configuration and for error
// reporting


// INFO section
// The INFO section gives some general service information on the form infoname = info
value
// Any new string can be added, removed or modified
[INFO]
VERSION = 1.00
BUILD = 1
DESCRIPTION = MYSERVICE NIRVA service
COMPANY =
COPYRIGTH =


// SETTINGS section
// The INFO section gives some general service paramters on the form param name = param
value
// Any new string can be added, removed or modified
[SETTINGS]
LANGUAGE = JAVA
PATH =


// PERMISSIONS section
// This section enumerates the MYSERVICE security permissons on the form permissionname
= permissiondescription
// If the security permissions are not used by the service, this section can be removed
or let empty
[PERMISSIONS]
```

```
// COMMAND_CLASSES section
// This section enumerates the MYSERVICE command classes on the form classname = class
description
// If the class names are not used by the service, this section can be removed or let
empty
[COMMAND_CLASSES]
MYSERVICE = default service command class


// COMMAND_CLASS sections
// This section enumerates the MYSERVICE commands of one class on the form commandname =
command description
// Each section itself has the form [COMMAND_CLASS_classname] where classname is the
name of the class
// as defined in the COMMAND_CLASSES section
[COMMAND_CLASS_MYSERVICE]
NOP = No operation


// ERROR_CLASSES section
// This section enumerates the MYSERVICE error classes on the form classname = class
description
// If the class names are not used by the service, this section can be removed or let
empty
[ERROR_CLASSES]
MYSERVICE = default service error class


// ERROR_LANGUAGES section
// This section enumerates the error languages of the error descriptions given in the
ERROR CLASS sections
// This is a succession of language strings. Valid languages are ENGLISH, FRENCH,
GERMAN, ITALIAN and SPANISH
// Any other language can be added.
// The language order must correspond to the description order given in the ERROR_CLASS
sections
[ERROR_LANGUAGES]
ENGLISH
FRENCH
GERMAN
ITALIAN
SPANISH


// ERROR_CLASS sections
// This section enumerates the MYSERVICE errors of one class on the form
// errorcode = error description language 1;error description language 2;etc..
// The error description language order must correspond to the order given in the
ERROR_LANGUAGES section.
// Each section itself has the form [ERROR_CLASS_classname] where classname is the name
of the class
// as defined in the ERROR_CLASSES section
[ERROR_CLASS_MYSERVICE]
0 = No error;pas d'erreur;Keine Störung;Nessun errore;Ningún error
```

INFO section

The INFO section gives some general service information.

There is a single INFO section in the description file.

The section is composed of several entries of the form *infoname* = *infovalue*. These entries can be retrieved directly by the “SYSTEM SERVICE INFOEX” NIRVA command.

A special entry in the INFO section is checked by NIRVA. This entry is named “ENCODING”. If encoding is set to “UTF-8”, NIRVA considers that the description file is coded with UTF-8 character set. If encoding is set to “ISO-8859-1”, NIRVA considers that the description file is coded with ISO-8859-1 character set (latin1). If there is no entry, Nirva doesn’t do any decoding of the values. This entry is also used for C++ services to tell Nirva what is the encoding waited by the service. Nirva will then encode/decode all data flow between Nirva and the service following the way nirva has been launched (utf8 or latin1).

SETTINGS section

The SETTINGS section gives some general service information.

There is a single SETTINGS section in the description file.

The section is composed of several entries of the form *paramname* = *paramvalue*.

Nirva checks the entries LANGUAGE and PATH that respectively gives the service language (C++, DOTNET or JAVA) and the relative path of the library for a C++ service, the relative path of the assembly for a Dotnet service or class for a Java service. The PATH is relative to the service directory.

The *VARIABLE_IDENT* entry allows to change the default value (# character) for variable identifier in commands send from the service code to Nirva. If the parameter is not given, the default value (#) is used. If the parameter is given but is empty, there will be no variable recognition in parameters. One can change the value of the variable identifier at command level using the NV_VAR_IDENT command parameter.

PERMISSIONS section

The PERMISSIONS section enumerates the service security permissions.

There is a single PERMISSIONS section in the description file.

The section is composed of several entries of the form *permissionname* = *permissiondescription*. These entries are directly used by the security layer of NIRVA. A permission can be checked by a service with dedicated NIRVA commands.

The permission names should use only alphanumeric characters (numbers and letters) and the underscore character

COMMAND_CLASSES section

The COMMAND_CLASSES section enumerates the service command classes.

There is a single COMMAND_CLASSES section in the description file.

The section is composed of several entries of the form *classname* = *classdescription*.

In order for a COMMAND_CLASS section to be taken in care (see next paragraph), the class name must be in the COMMAND_CLASSES section.

The class names should use only alphanumeric characters (numbers and letters) and the underscore character

COMMAND_CLASS sections

There are several class sections in the description file. In fact, there is one section for each command class.

A COMMAND_CLASS section name is always followed by the class name. For example COMMAND_CLASS_TEST is the COMMAND_CLASS section for the TEST class.

A COMMAND_CLASS section enumerates the commands of a given class.

The section is composed of several entries of the form *commandname* = *commanddescription*.

The command names should use only alphanumeric characters (numbers and letters) and the underscore character

ERROR_CLASSES section

The ERROR_CLASSES section enumerates the service error classes.

There is a single ERROR_CLASSES section in the description file.

The section is composed of several entries of the form *errorclassname* = *errorclassdescription*.

In order for an ERROR_CLASS section to be taken in care (see next paragraph), the class name must be in the ERROR_CLASSES section.

The error class names should use only alphanumeric characters (numbers and letters) and the underscore character

ERROR_CLASS sections

There are several error class sections in the description file. In fact, there is one section for each error class.

An ERROR _CLASS section name is always followed by the error class name. For example ERROR_CLASS_TEST is the ERROR_CLASS section for the TEST error class.

An ERROR_CLASS section enumerates the errors of a given class.

The section is composed of several entries of the form *errorcode* = *error description language 1;error description language 2;etc…* The error description language order must correspond to the order given in the ERROR_LANGUAGES section (see next paragraph).

ERROR_LANGUAGES section

This section enumerates the error languages of the error descriptions given in the ERROR_CLASS sections.

There is a single ERROR_LANGUAGES section in the description file.

This is a succession of language strings. Valid languages are ENGLISH, FRENCH, GERMAN, ITALIAN and SPANISH. Any other language can be added.

The language order must correspond to the description order given in the ERROR_CLASS sections.

When an error occurs, NIRVA directly uses the error codes given in the service description file to display the error description in the required language. The language to use is given by the general NIRVA command parameter NV_LANGUAGE.

Documentation

NIRVA is able to display the service documentation in html. For that the service must give the documentation start page. This page is named “servicename.htm” where servicename is the service name in lowercase. This page must reside in the “Html” subdirectory of the service documentation directory. (Nirva/Services/servicename/Docs).

In order to see the documentation one just need to get the following URL from a WEB browser:

```
http://127.0.0.1:1081/NV_DOC_SRV_servicename/Html/servicename.htm
```

Where 127.0.0.1:1081 can be replaced by the real NIRVA server address and port and servicename is the name of the service. This causes NIRVA to answer with the service “servicename.htm” page that resides in the service doc directory.

Configuration

NIRVA can also call directly the configuration page of a service. This is an XSL page. For that the service must give the documentation start page. This page is named “config.xsl” and must reside in the Config subdirectory of the service files directory. (Nirva/Services/servicename/Files/Config). The service can also give a procedure file named “config.nvp” that resides in the Config subdirectory of the service procedure directory. (Nirva/Services/servicename/Procs/Config).

In order to see the configuration one just need to get the following URL from a WEB browser:

```
http://127.0.0.1:1081/NV_SRV_servicename/Config/NVS?command&NV_CMD=SERVICE:CONFIG&SERVIC
E=servicename
```

Where 127.0.0.1:1081 can be replaced by the real NIRVA server address and port and servicename is the name of the service. This is a call to the NIRVA SYSTEM SERVICE CONFIG command that causes NIRVA to call the config.nvp procedure and to generate an html page using the config.xsl file. Please see the

description of the SYSTEM SERVICE CONFIG for further information about the service configuration command.

The config web site must reside in the Config directory of the service Wroot directory.

Installation

The service provider should deliver some NIRVA installation package that can be used directly with the SYSTEM SERVICE INSTALL command.

Service package

Nirva provides commands for creating and installing service packages. A service package is created following some information given in a package description file. The service provider creates a service package with the SYSTEM SERVICE PACKAGE command then the package can be installed by using the SYSTEM SERVICE INSTALL command.

For security reasons, these packages can only install service files on the target service directory. If some other files have to be installed in other server directories, this can be done by using a special service installation procedure named “install.nvp”.

Please consult the SYSTEM SERVICE PACKAGE and SYSTEM SERVICE INSTALL command description for further information about the service installation. The package file syntax is described in the chapter “installation packages”.

Installing a service

The service must be stopped for the installation procedure to be successful.

In order to install a service, go into the nirva configuration tool in the System/Services menu and press the install button. Please see the NIRVA configuration chapter for further information.

The service can also be installed from the nvcc command line tool.

For that the service provider may also deliver some nvcc command files to install these packages. If they do not provide them, one can use the standard “`service_install.txt`” command file delivered in standard with NIRVA in the Bin directory. Please consult the “tools” chapter for further information on the “`service_install.txt`” command file.

Here is an example of using the “`service_install.txt`” command file to install a service named “PLANET” with a package file named “planet.pkg” under windows:

```
C:\Nirva\Bin>nvcc -d ep -i service_install.txt PLANET planet.pkg

Service PLANET installation
stopping service...
installing service...

Installation OK

C:\Nirva\Bin>
```

Licensing

License policy

Nirva license policy is to let license control to service provider. When a service provider builds a new service, Nirva generates a pair of public and private keys. The private key is used by the service provider to construct license channels by the way of the nirva license tool (nvl). It then uses the public key for checking the license channels from its source code.

Installing service licenses

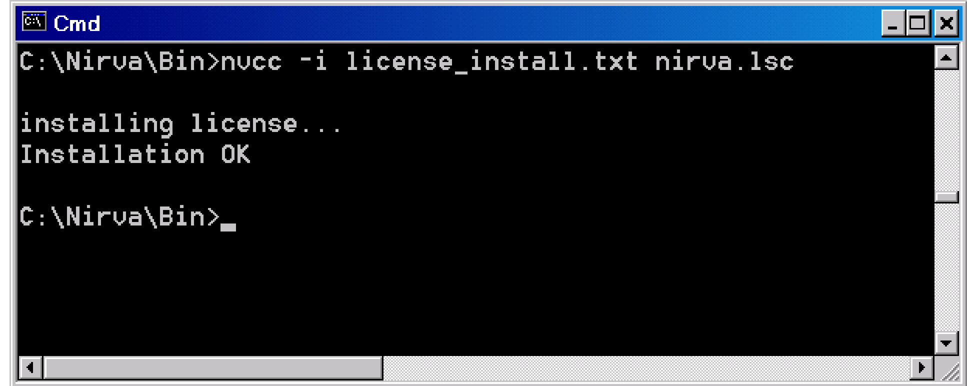
The licensing of a service is similar to the global NIRVA licensing procedure.

The customer must send its unique machine identifier to the service provider. For that, he can go into the system/license menu of the configuration tool or use the standard "getmachine.txt" command file delivered in standard with NIRVA in the Bin directory. Here is an example of using the command file:

```
C:\Nirva\Bin>nvcc -i getmachine.txt

Machine ID: 00DE8B48FACE9B2F655B2CCBD703B53B

C:\Nirva\Bin>
```

The returned machine ID must then be transmitted to the service provider with the required license information requested by the service provider.

Then, the service provider returns a license file that can be installed by the customer using the configuration tool or using the standard "license_install.txt" command file delivered in standard with NIRVA in the Bin directory. Here is an example of using the command file with a license file named nirva.lsc:



This command file installs the given license file on NIRVA license manager.

Some service providers may require some different procedures for the installation of the licenses. At this time, please use their procedure instead of the standard one.

Automatic skeleton

NIRVA provides a powerful command named SYSTEM SERVICE SKELETON that generates all the service components in a very simple way including the source files in C++, Dotnet or Java.

The generated source can include session management or not.

If the service is C++, the SYSTEM SERVICE SKELETON command creates the following files in the service source directory if they don't exist:

servicename.cpp — where *servicename* is replaced by the service name is the service C++ source file that contains the NvsCommand entry point and the eventual session class if the session option has been requested. The file extension is ".C" under AIX instead of ".cpp".

servicename.h — where *servicename* is replaced by the service name is the corresponding header file containing declarations of the Nirva command interface.

nirva.def — is a WINDOWS specific file that exports the NvsCommand function.

makefile.aix — is the AIX make file.

makefile.linux — is the LINUX make file.

makefile.solaris — is the SOLARIS make file.

makefile.hp — is the HPUX PA-RISC make file.

makefile.hpi — is the HPUX Itanium make file.

servicename.dsp — where *servicename* is replaced by the service name is the Microsoft Visual Studio 6 project file.

servicename.dsw — where *servicename* is replaced by the service name is the Microsoft Visual Studio 6 workspace file.

nvthread.h	Header file that defines the class NirvaMutex that can be used by the programmer for managing accesses to shared resources from the multithreaded service. This file also defines a macro named SLEEP that allows sleeping the current thread for the given number of milliseconds.
nvthread.cpp	Implementation file of the NirvaMutex class. The file extension is ".C" under AIX instead of ".cpp".

If the service is Java, the SYSTEM SERVICE SKELETON command creates the following files in the service source directory if they don't exist:

servicename.java	where *servicename* is replaced by the service name is the service java source file that contains the service class and the eventual session class if the session option has been requested.

If the service is Dotnet the SYSTEM SERVICE SKELETON command creates the following files in the service source directory if they don't exist:

servicename.cs	where *servicename* is replaced by the service name is the service c# source file that contains the service class and the eventual session class if the session option has been requested.
servicename.sln	where *servicename* is replaced by the service name is the service solution file for Visual C# 2008.
servicename.csproj	where *servicename* is replaced by the service name is the service project file for Visual C# 2008.
compile.bat	This is a batch file for compiling the service if Visual C# 2008 is not used.
Properties/assemblyinfo.cs	Assembly information file used by Visual C# 2008 to describe the assembly file.

It also creates the following files if they don't exist:

license.txt	This file is also in the Source directory. It contains the service private and public Ids for creating and checking licenses.
service.dsc	This is the service description file that contains some information that describe the service. This file is used by NIRVA when mounting the service. This file is created in the service files directory.
config.xsl	This is the xsl file used to create the service configuration WEB page when using the SYSTEM SERVICE CONFIG command. This page should be modified by the service programmer. This file is created in the subdirectory Config of the service files directory.
config.nvp	This is an empty procedure called by the SYSTEM SERVICE CONFIG command. This procedure should be modified by the service programmer if necessary. This file is created in the subdirectory Config of the service procedure directory.

servicename.htm	where *servicename* is replaced by the service name is the starting html page for service documentation. This page should be modified by the service programmer if necessary. This file is created in the service documentation directory.
package.lst	This is the installation package file. This file contains information for creating a NIRVA installation package for the service.
install.nvp	This is an empty procedure called by the SYSTEM SERVICE INSTALL command. This procedure should be modified by the service programmer if necessary. This file is created in the service procedure directory.

C++

Without session management

Without session management, the SYSTEM SERVICE SKELETON command generates basic C++ and header files.

By default the command generates the code for session management. In order to not include this code, the command must have the parameter WITH_SESSIONS="NO".

servicename.cpp C++ file

Here is the file generated for a service named “MYSERVICE” without session management:

```
// myservice.cpp : implementation file
// Contains the Nirva external service entry point NvsCommand

// Please insert code where there is a TODO string in comments


#include "myservice.h"

int NvInitLib(NvServiceCommand *Command);        // Library init
int NvExitLib(NvServiceCommand *Command);        // Library exit
void OnInternalError(NvServiceCommand *Command); // Called in case of internal error
bool OnCommand(NvServiceCommand *Command);       // Called for each command


// Main Nirva external service entry point
int NvsCommand(void *pCommand)
{
  // The pCommand points in fact on a NvServiceCommand class
  NvServiceCommand *Command = (NvServiceCommand *)pCommand;

  // Check the service init and exit commands
  if(Command->IsCommand("SYSTEM", "NV_INIT_SERVICE", NV_SOURCE_SERVER))
          return NvInitLib(Command);
  if(Command->IsCommand("SYSTEM", "NV_EXIT_SERVICE", NV_SOURCE_SERVER))
          return NvExitLib(Command);
  if(Command->IsCommand("SYSTEM", "NV_INIT_SESSION", NV_SOURCE_SERVER))
```

```
        return true;
  if(Command->IsCommand("SYSTEM", "NV_EXIT_SESSION", NV_SOURCE_SERVER))
        return true;


  // Used for testing the service
  if(Command->IsCommand("MYSERVICE", "NOP"))
        return 1;


  // By default the command is successfull
  bool RetCode = true;

  // Send now the command
  try
  {
        RetCode = OnCommand(Command);
  }
  catch(...)
  {
        // The command generated an exception
        // This is an error
        OnInternalError(Command);
        RetCode = false;
  }

  //      Now we can return the result
  if(!RetCode)
        return 0;

  return 1;
}


// Called for each Nirva command to the service
bool OnCommand(NvServiceCommand *Command)
{
  // TODO
  // Insert command processing here
  return true;
}



// Called on time when the library is loaded in memory
// Should return true if successful and false otherwise
int NvInitLib(NvServiceCommand *Command)
{
  // TODO
  // Insert init libary code here

  // Everything is OK
  return 1;
}

// Called on time when the library is unloaded from memory
// Should return true if successful and false otherwise
int NvExitLib(NvServiceCommand *Command)
{
  // TODO
```

```
  // Insert cleanup libary code here

  // Everything is OK
  return 1;
}


// Called in case of internal error or when an exception occurs
void OnInternalError(NvServiceCommand *Command)
{
  // TODO
  // Insert eventual code here
  // This is a good place for calling the function Command->SetError in order to set an
error code
  // Example: Command->SetError("MYSERVICE", MYSERVICE_INTERNALERROR);
}
```

This source file implements the necessary NvsCommand function that is the entry point of the library. The implementation is complete and should not be changed.

The programmer should modify the following functions for implementing its technology:

OnCommand	This function is called for each command to the service except for the NOP command that is directly managed in the NvsCommand function. This is the main place were the functionality of the service should be implemented.
NvInitLib	This function is called each time the library is loaded in memory by the NIRVA server. It can be used for library initialization code.
NvExitLib	This function is called each time the library is unloaded from memory by the NIRVA server. It can be used for library cleanup code.
OnInternalError	This function is called when an exception occurs in the OnCommand function. It can be used to set a specific error code for this condition.

servicename.h header file

```
// myservice.h : header file
// Nirva service

#ifndef _MYSERVICE_H
#define _MYSERVICE_H

#include "nvsext.h"

// NIRVA entry point function
extern "C"
{
  int NvsCommand(void *pCommand);
}


#endif //_MYSERVICE_H
```

This header itself includes the nvsext.h header file that contains some NIRVA definitions. It especially contains the declaration of the NvServiceCommand class that defines the interface with NIRVA server.

With session management

With session management, the SYSTEM SERVICE SKELETON command generates nearly the same basic C++ and header files but adds a class for session management.

This class is named *SERVICENAME*Session where *SERVICENAME* is the name of the service in uppercase. A new class is created when a new NIRVA session tries to send the first command to the service. The class is deleted when a NIRVA session that worked with the service is deleted. There is a unique correspondance between a NIRVA session and a service session. In fact, a NIRVA session communicates with only one service session by external service.

The programmer doesn't have to take care about the creation and deleting of the session class. This is made automatically by the NvsCommand function of the library.

By default the SYSTEM SERVICE SKELETON command generates the code for session management.

servicename.cpp C++ file

Here is the file generated for a service named "MYSERVICE" with session management:

```
// myservice.cpp : implementation file
// Contains the Nirva external service entry point NvsCommand
// Please insert code where there is a TODO string in comments


#include "myservice.h"

int NvInitLib(NvServiceCommand *Command);        // Library init
int NvExitLib(NvServiceCommand *Command);        // Library exit
void OnInternalError(NvServiceCommand *Command); // Called in case of internal error

// Main Nirva external service entry point
// This function should not be modified
// All implementation must be made in the MYSERVICESession class
int NvsCommand(void *pCommand)
{
  // The pCommand points in fact on a NvServiceCommand class
  NvServiceCommand *Command = (NvServiceCommand *)pCommand;

  // Check the service init and exit commands
  if(Command->IsCommand("SYSTEM", "NV_INIT_SERVICE", NV_SOURCE_SERVER))
          return NvInitLib(Command);
  if(Command->IsCommand("SYSTEM", "NV_EXIT_SERVICE", NV_SOURCE_SERVER))
          return NvExitLib(Command);


  // Used for testing the service
  if(Command->IsCommand("MYSERVICE", "NOP"))
          return 1;
```

```
// Check the session init command
if(Command->IsCommand("SYSTEM", "NV_INIT_SESSION", NV_SOURCE_SERVER))
{
        // Session init
        // We create a new MYSERVICE session object
        MYSERVICESession *Session = new MYSERVICESession;
        if(Session == NULL)
                return 0;


        // Init the object
        if(!Session->OnInit(Command))
        {
                delete Session;
                return 0;
        }

        // And attach it to the Nirva session
        Command->SetServiceSession((void *)Session);

        // We don't need to stay any more
        return 1;
}


// Get the MYSERVICE session object attached to the Nirva session
MYSERVICESession *Session = (MYSERVICESession *)Command->GetServiceSession();


// Check the session exit command
if(Command->IsCommand("SYSTEM", "NV_EXIT_SESSION", NV_SOURCE_SERVER))
{
        // Session exit

        // The session should not be null
        if(Session == NULL)
                return 0;

        // We cleanup and delete the MYSERVICE session object
        // We do that in a try/catch pair in case of a bad Session handle
        try
        {
                Session->OnCleanup(Command);
                delete Session;
        }
        catch(...)
        {
        }


        // We don't need to stay any more
        return 1;
}

// The session should not be null
if(Session == NULL)
{
        OnInternalError(Command);
        return 0;
}
```

```
  // By default the command is successfull
  bool RetCode = true;

  // Send now the command
  try
  {
          RetCode = Session->OnCommand(Command);
  }
  catch(...)
  {
          // The session object is not valid or the command generated an exception
          // This is an error
          OnInternalError(Command);
          RetCode = false;
  }

  //      Now we can return the result
  if(!RetCode)
          return 0;

  return 1;
}


// Called on time when the library is loaded in memory
// Should return true if successful and false otherwise
int NvInitLib(NvServiceCommand *Command)
{
  // TODO
  // Insert init libary code here

  // Everything is OK
  return 1;
}


// Called on time when the library is unloaded from memory
// Should return true if successful and false otherwise
int NvExitLib(NvServiceCommand *Command)
{
  // TODO
  // Insert cleanup libary code here

  // Everything is OK
  return 1;
}


// Called in case of internal error or when an exception occurs
void OnInternalError(NvServiceCommand *Command)
{
  // TODO
  // Insert eventual code here
  // This is a good place for calling the function Command->SetError in order to set an
error code
  // Example: Command->SetError("MYSERVICE", MYSERVICE_INTERNALERROR);
}
```

```
////////////////////////////////////////////////////////////////////////////
// MYSERVICESession
// This is the session class specific to this service

// Constructor
MYSERVICESession::MYSERVICESession()
{
  Initialized = false;

  // TODO
  // Insert eventual construction code here
  // The constructor should only initialize class members to default values
  // The main initialization should be done in the OnInit function
}

// Destructor
MYSERVICESession::~MYSERVICESession()
{
  // TODO
  // Insert eventual destruction code here
  // The main cleanup should be done in the OnCleanup function

  // By security, we cleanup but this should have already been done
  OnCleanup();
}


// Called one time when initializing the session
bool MYSERVICESession::OnInit(NvServiceCommand *Command)
{
  if(Initialized)
        return true;

  // TODO
  // Insert eventual initialization code here

  // All is OK
  Initialized = true;
  return true;
}


// Called one time when closing the session
bool MYSERVICESession::OnCleanup(NvServiceCommand *Command)
{
  if(!Initialized)
        return true;

  // TODO
  // Insert eventual cleanup code here (called by the destructor)

  // All is OK
  Initialized = false;
  return true;
}


// Called for each Nirva command to the service
// This is the command entry point for session
bool MYSERVICESession::OnCommand(NvServiceCommand *Command)
```

```
{
  // TODO
  // Insert command processing here

  // Everything is OK
  return true;
}
```

This source file implements the necessary NvsCommand function that is the entry point of the library. The implementation is complete and should not be changed.

The programmer should modify the following functions for implementing its technology:

NvInitLib	This function is called each time the library is loaded in memory by the NIRVA server. It can be used for library initialization code.
NvExitLib	This function is called each time the library is unloaded from memory by the NIRVA server. It can be used for library cleanup code. This is not necessary for the programmer to cleanup the *SERVICENAME*Session classes allocated in memory because NIRVA first sends the necessary commands to the library for these classes to be removed before calling the NvExitLib function.
OnInternalError	This function is called when an exception occurs in the OnCommand function. It can be used to set a specific error code for this condition.
*SERVICENAME*Session::OnInit	This class function is called just after a new session class has been instanced in memory. It may be use for session initialization. A new session class is created when a new NIRVA session tries to send the first command to the service.
*SERVICENAME*Session::OnCleanup	This class function is called just before a session class to be deleted. It may be use for session cleanup code. The session class is deleted when a NIRVA session that worked with the service is deleted.
*SERVICENAME*Session::OnCommand	This function is called for each command to the service session except for the NOP command that is directly managed in the NvsCommand function. This is the main place were the functionality of the service should be implemented.

servicename.h header file

```
// myservice.h : header file
// Nirva service

#ifndef _MYSERVICE_H
```

```
#define _MYSERVICE_H

#include "nvsext.h"

// NIRVA entry point function
extern "C"
{
  int NvsCommand(void *pCommand);
}


/////////////////////////////////////////////////////////////////////////////
// MYSERVICESession class
// TODO
// Add necessary class members and functions to implement your service functionnality


class MYSERVICESession
{
 public:
  bool Initialized;
  MYSERVICESession();
  virtual ~MYSERVICESession();

  bool OnInit(NvServiceCommand *Command);
  bool OnCleanup(NvServiceCommand *Command = NULL);
  bool OnCommand(NvServiceCommand *Command);
};


#endif //_MYSERVICE_H
```

This header itself includes the nvsext.h header file that contains some NIRVA definitions. It especially contains the declaration of the NvServiceCommand class that defines the interface with NIRVA server.

The servicename.h header file declares the *SERVICENAME*Session class that implements the session management at service level.

Java

Without session management

Without session management, the SYSTEM SERVICE SKELETON command generates a basic java file.

By default the command generates the code for session management. In order to not include this code, the command must have the parameter WITH_SESSIONS="NO".

Here is the file generated for a service named “MYSERVICE” without session management:

```
// myservice.java : implementation file
// Contains the Nirva external service class myservice

// Please insert code where there is a TODO string in comments
```

```
import com.nirvasoft.nirva.nvcmd;

// class instancied when the service is started
class myservice
{

  // Called one time when the service is started
  // Should return true if successful and false otherwise
  public boolean Init()
  {
          // TODO
          // Insert init service code here
          // Everything is OK
          return true;
  }


  // Called one time when the service is stopped
  // Should return true if successful and false otherwise
  public boolean Exit()
  {
          // TODO
          // Insert service cleanup code here

          return true;
  }


  // Called for each Nirva command to the service
  public boolean Command()
  {
          nvcmd NvCommand = new nvcmd();

          // Used for testing the service
          if(NvCommand.IsCommand("MYSERVICE", "NOP", ""))
                  return true;
          try
          {
                  // TODO
                  // Insert command processing here
          }
          catch(Throwable e)
          {
          }
          return true;
  }
}
```

This source file implements the myservice class with the required Init, Exit and Command methods.

The programmer should modify the following methods for implementing its technology:

servicename::Command	This method is called for each command to the service. This is the main place were the functionality of the service should be implemented.
servicename::Init	This method is called each time the service is started. It can be used for library initialization code.

servicename::Exit	This method is called each time the service is stopped. It can be used for library cleanup code.

With session management

With session management, the SYSTEM SERVICE SKELETON command generates nearly the same basic java file but adds a class for session management.

This class is named *servicename*session where *servicename* is the name of the service in lowercase. A new class is created when a new NIRVA session tries to send the first command to the service. The class is deleted when a NIRVA session that worked with the service is deleted. There is a unique correspondance between a NIRVA session and a service session. In fact, a NIRVA session communicates with only one service session by external service.

The programmer doesn’t have to take care about the creation and deleting of the session class. This is made automatically by the Command function of the service class.

By default the SYSTEM SERVICE SKELETON command generates the code for session management.

Here is the file generated for a service named “MYSERVICE” with session management:

```
// myservice.java : implementation file
// Contains the Nirva external service class myservice

// Please insert code where there is a TODO string in comments

import com.nirvasoft.nirva.nvcmd;

// class instancied when the service is started
class myservice
{
  java.util.Hashtable<String,myservicesession> Sessions;

  // Called one time when the service is started
  // Should return true if successful and false otherwise
  public boolean Init()


  {
         // Create a hash table for maintaining session objects
         Sessions = new java.util.Hashtable<String,myservicesession>();


         // TODO
         // Insert init service code here

         // Everything is OK
         return true;
  }


  // Called one time when the service is stopped
  // Should return true if successful and false otherwise
  public boolean Exit()
  {
```

```
        // TODO
        // Insert service cleanup code here

        // Free the session hash table
        Sessions.clear();
        Sessions = null;
        return true;
}


public boolean Command()
{
        nvcmd NvCommand = new nvcmd();

        // Used for testing the service
        if(NvCommand.IsCommand("MYSERVICE", "NOP", ""))
                return true;

        // Check the session init command
        String SessionId = NvCommand.GetSessionId();
        if(NvCommand.IsCommand("SYSTEM", "NV_INIT_SESSION", "SERVER"))
        {
                // Create a new session object
                myservicesession Session = new myservicesession(SessionId);
                // Init the object
                boolean Result = true;
                try
                {
                        Result = Session.OnInit(NvCommand);
                }
                catch(Throwable e)
                {
                        Result = false;
                }
                if(!Result)
                        return false;

                // Put the session object into the hash table
                Sessions.put(SessionId, Session);
                return true;   // end processing init session
        }

        // Get the session object from the hash table
        myservicesession Session = Sessions.get(SessionId);
        if(Session == null)
                return false;

        // Check the session exit command
        if(NvCommand.IsCommand("SYSTEM", "NV_EXIT_SESSION", "SERVER"))
        {
                // Cleanup the session object
                boolean Result = true;
                try
                {
                        Result = Session.OnCleanup(NvCommand);
                }
                catch(Throwable e)
                {
```

```
                        Result = false;
                }
                if(!Result)
                        return false;
                // And remove it from the hash table
                Sessions.remove(SessionId);
                return true;    // end processing exit session
        }

        // Other command processing
        boolean Result = true;
        try
        {
                Result = Session.OnCommand(NvCommand);
        }
        catch(Throwable e)
        {
                Result = false;
        }
        if(!Result)
                return false;

        return true;
  }
}

class myservicesession
{
  // Session Id
  String sessionId;

  // Constructor with session if, do not modify
  public myservicesession(String sessionId) {
        this.sessionId = sessionId;
  }

  // Used for sessions hash table, do not modify
  public int hashCode() {
        return sessionId.hashCode()
  }

   // Called one time when initializing the session
   // return true if successful and false otherwise
  public boolean OnInit(nvcmd Command)
  {
        // TODO
        // Insert eventual initialization code here

        // All is OK
        return true;
  }

  // Called one time when closing the session
  // return true if successful and false otherwise
  public boolean OnCleanup(nvcmd Command)
  {
        // TODO
        // Insert eventual cleanup code here
```

```
        // All is OK
        return true;
    }

    // Called for each Nirva command to the service
    // This is the command entry point for session
    // return true if successful and false otherwise
    public boolean OnCommand(nvcmd Command)
    {
        // TODO
        // Insert command processing here
        // Everything is OK
        return true;
    }
}
```

This source file implements the myservice class with the required Init, Exit and Command methods.

The programmer should modify the following methods for implementing its technology:

servicename::Init	This method is called each time the service is started. It can be used for library initialization code.
servicename::Exit	This method is called each time the service is stopped. It can be used for library cleanup code.
*servicename*session::OnInit	This method is called just after a new session class has been instanced in memory. It may be used for session initialization. A new session class is created when a new NIRVA session tries to send the first command to the service.
*servicename*session::OnCleanup	This class method is called just before a session class to be deleted. It may be used for session cleanup code. The session class is deleted when a NIRVA session that worked with the service is deleted.
*servicename*session::OnCommand	This function is called for each command to the service session except for the NOP command that is directly managed in the myservice::Command method. This is the main place were the functionality of the service should be implemented.

Dotnet

Without session management

Without session management, the SYSTEM SERVICE SKELETON command generates a basic c# file.

By default the command generates the code for session management. In order to not include this code, the command must have the parameter WITH_SESSIONS="NO".

Here is the file generated for a service named “MYSERVICE” without session management:

```
// myservice.cs : implementation file
// Contains the Nirva external service class myservice

// Please insert code where there is a TODO string in comments

using System;
using Nirva;

// class instancied when the service is started
public class myservice : ServiceEntryPoint
{

  // Called one time when the service is started
  // Should return true if successful and false otherwise
  public override bool Init(nvcmd NvCommand)
  {

          // TODO
          // Insert init service code here

          // Everything is OK
          return true;
  }


  // Called one time when the service is stopped
  // Should return true if successful and false otherwise
  public override bool Exit(nvcmd NvCommand)
  {
          // TODO
          // Insert service cleanup code here
          return true;
  }


  // Called for each Nirva command to the service
  public override bool Command(nvcmd NvCommand)
  {
          // Used for testing the service
          if(NvCommand.IsCommand("MYSERVICE", "NOP", ""))
                  return true;

          bool Result = true;
          try
```

```
            {
                    // TODO
                    // Insert command processing here
            }
            catch(Exception e)
            {
                    Result = false;
                    // Uncomment next line if no console output required
                    Console.WriteLine(e.Message);
            }

            if (!Result)
                    return false;
            return true;
    }
  }
```

This source file implements the myservice class with the required Init, Exit and Command methods.

The programmer should modify the following methods for implementing its technology:

servicename.Command	This method is called for each command to the service. This is the main place were the functionality of the service should be implemented.
Servicename.Init	This method is called each time the service is started. It can be used for library initialization code.
Servicename.Exit	This method is called each time the service is stopped. It can be used for library cleanup code.

With session management

With session management, the SYSTEM SERVICE SKELETON command generates nearly the same basic c# file but adds a class for session management.

This class is named *servicename*session where *servicename* is the name of the service in lowercase. A new class is created when a new NIRVA session tries to send the first command to the service. The class is deleted when a NIRVA session that worked with the service is deleted. There is a unique correspondance between a NIRVA session and a service session. In fact, a NIRVA session communicates with only one service session by external service.

The programmer doesn't have to take care about the creation and deleting of the session class. This is made automatically by the Command function of the service class.

By default the SYSTEM SERVICE SKELETON command generates the code for session management.

Here is the file generated for a service named “MYSERVICE” with session management:

```
// myservice.cs : implementation file
// Contains the Nirva external service class myservice

// Please insert code where there is a TODO string in comments
```

```
using System;
using Nirva;
using System.Collections;
using System.Threading;

// class instancied when the service is started
public class myservice : ServiceEntryPoint
{
  Hashtable Sessions;
  Object ProtectSessions;

  // Called one time when the service is started
  // Should return true if successful and false otherwise
  public override bool Init(nvcmd NvCommand)
  {
        // Create a hash table for maintaining session objects
        // Create also the associated thread synchronisation object
        ProtectSessions = new Object();
        Sessions = new Hashtable();

        // TODO
        // Insert init service code here

        // Everything is OK
        return true;
  }

  // Called one time when the service is stopped
  // Should return true if successful and false otherwise
  public override bool Exit(nvcmd NvCommand)
  {
        // TODO
        // Insert service cleanup code here

        // Free the session hash table
        Monitor.Enter(ProtectSessions);
        Sessions.Clear();
        Sessions = null;
        Monitor.Exit(ProtectSessions);
        return true;
  }

  public override bool Command(nvcmd NvCommand)
  {
        // Used for testing the service
        if(NvCommand.IsCommand("MYSERVICE", "NOP", ""))
                return true;

        bool Result = true;

        // Check the session init command
        String SessionId = NvCommand.GetSessionId();
        myservicesession Session = null;
        if (NvCommand.IsCommand("SYSTEM", "NV_INIT_SESSION", "SERVER"))
        {
                // Create a new session object
                Session = new myservicesession(SessionId);
                // Init the object
                try
```

```
			{
				Result = Session.OnInit(NvCommand);
			}
			catch (Exception e)
			{
				Result = false;
				// Uncomment next line if no console output required
				Console.WriteLine(e.Message);
			}
			if(!Result)
				return false;

			// Put the session object into the hash table
			Monitor.Enter(ProtectSessions);
			Sessions.Add(SessionId, Session);
			Monitor.Exit(ProtectSessions);
			return true;	// end processing init session
		}

		// Get the session object from the hash table
		Monitor.Enter(ProtectSessions);
		if (Sessions.Contains(SessionId))
			Session = (myservicesession)Sessions[SessionId];
		Monitor.Exit(ProtectSessions);
		if (Session == null)
			return false;

		// Check the session exit command
		if(NvCommand.IsCommand("SYSTEM", "NV_EXIT_SESSION", "SERVER"))
		{
			// Cleanup the session object
			try
			{
				Result = Session.OnCleanup(NvCommand);
			}
			catch (Exception e)
			{
				Result = false;
				// Uncomment next line if no console output required
				Console.WriteLine(e.Message);
			}

			// And remove it from the hash table
			Monitor.Enter(ProtectSessions);
			if (!Sessions.Contains(SessionId))
				Sessions.Remove(SessionId);
			Monitor.Exit(ProtectSessions);
			if(!Result)
				return false;
			return true;	// end processing exit session
		}

		// Other command processing
		try
		{
			Result = Session.OnCommand(NvCommand);
		}
		catch (Exception e)
```

```
            {
                    Result = false;

                    // Uncomment next line if no console output required
                    Console.WriteLine(e.Message);
            }

            if (!Result)
                    return false;
            return true;
    }
}

class myservicesession
{
  // Session Id
  String sessionId;

  // Constructor with session if, do not modify
  public myservicesession(String sessionId)
  {
          this.sessionId = sessionId;
  }

  // Called one time when initializing the session
  // return true if successful and false otherwise

  public bool OnInit(nvcmd Command)
  {
          // TODO
          // Insert eventual initialization code here

          // All is OK
          return true;
  }


  // Called one time when closing the session
  // return true if successful and false otherwise
  public bool OnCleanup(nvcmd Command)
  {
          // TODO
          // Insert eventual cleanup code here
          // All is OK
          return true;
  }

  // Called for each Nirva command to the service
  // This is the command entry point for session
  // return true if successful and false otherwise
  public bool OnCommand(nvcmd Command)
  {
          // TODO
          // Insert command processing here

          // Everything is OK
          return true;
```

```
    }
  }
```

This source file implements the myservice class with the required Init, Exit and Command methods.

The programmer should modify the following methods for implementing its technology:

Servicename.Init	This method is called each time the service is started. It can be used for library initialization code.
Servicename.Exit	This method is called each time the service is stopped. It can be used for library cleanup code.
*Servicename*session.OnInit	This method is called just after a new session class has been instanced in memory. It may be used for session initialization. A new session class is created when a new NIRVA session tries to send the first command to the service.
*Servicename*session.OnCleanup	This class method is called just before a session class to be deleted. It may be used for session cleanup code. The session class is deleted when a NIRVA session that worked with the service is deleted.
*Servicename*session.OnCommand	This function is called for each command to the service session except for the NOP command that is directly managed in the myservice::Command method. This is the main place were the functionality of the service should be implemented.

Tutorial

This example is only a school case. It will create a service named PLANET that implements a single command named GET that creates a NIRVA string list object containing the planet names.

This example is described under a windows platform. On the UNIX platforms, the commands are nearly similar except for the compilation and debugging steps of C++ services.

Before trying this example, the NIRVA server must run and the user must have enough rights to run the necessary commands. The example runs with the NIRVA default application (NVDEF) and user (nvdef). Please see the configuration chapter in order to give all the necessary permissions to the nvdef user in the NVDEF application. The best way is to give him all permissions. For commodity, please do not assign any password to the default user.

The example assumes that the client is on the same machine than the NIRVA server. The NIRVA application is the default application.

In order to view the orders sent by the NIRVA server, this one can be started in console mode (nvs –c command).

The creation of a running service without any functionality takes 5 steps:

- Creation of the service skeleton.

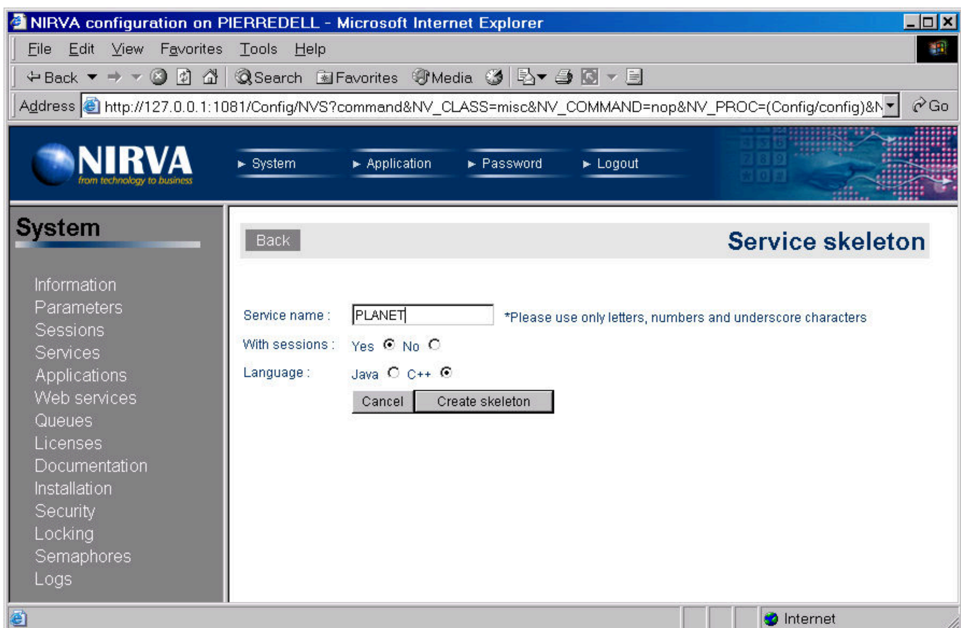
- Compilation of the service
- Mounting of the service on NIRVA
- Starting the service
- Testing the service

C++ service

For compilation, the example assumes that the MICROSOFT VISUAL STUDIO with the C++ compiler is installed.

Creating the skeleton

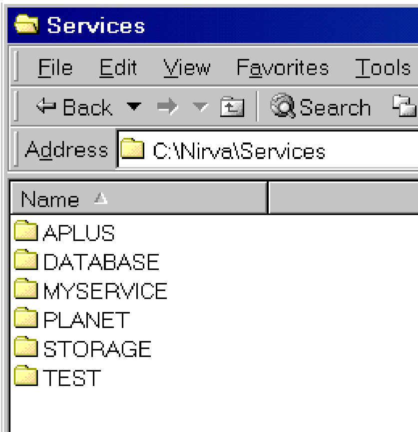
The first step is to create the service skeleton. For that one just need to run the Nirva configuration tool, go to the system/service menu and press the “Skeleton” button. Then edit the form as follow:



Don’t forget to set the service language as C++ (default is Java).

Then press the “Create skeleton” button. If there is no error message, the skeleton has been created on the server side.

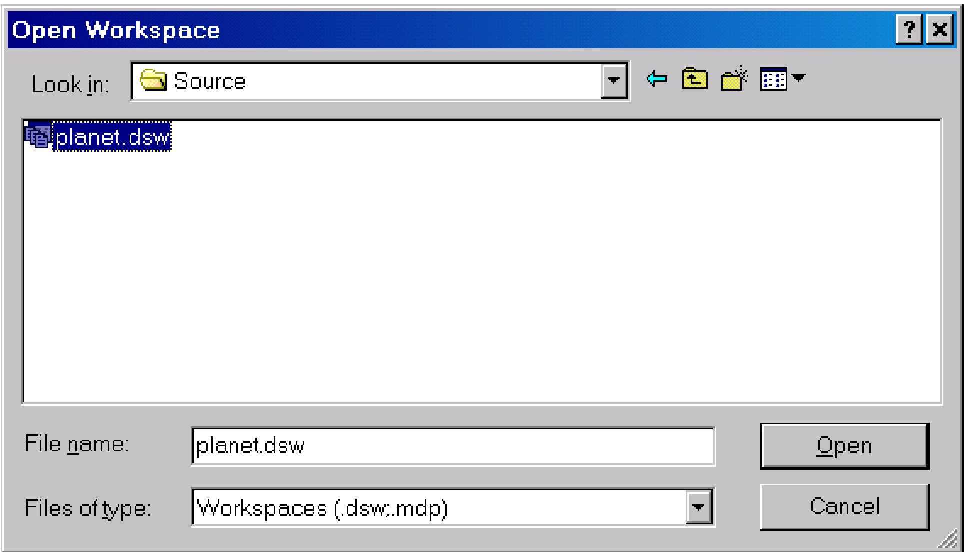
This can be verified by checking the NIRVA services directory. A new directory named PLANET should have been created:



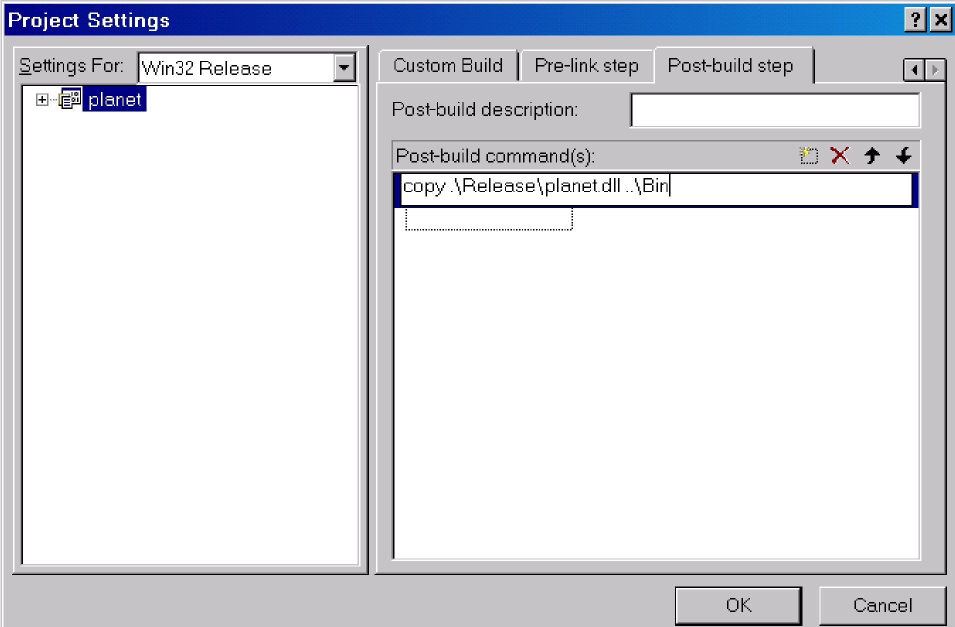
Now the first step is finished

Compiling the service

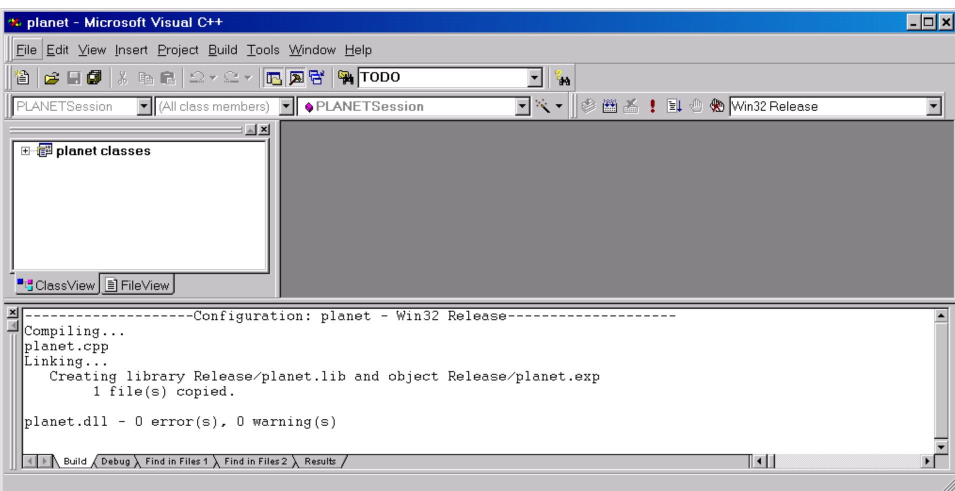
In order to compile, just open the MICROSOFT VISUAL STUDIO and choose the File/Open workspace menu. Then open the workspace planet.dsw found on Nirva/Services/PLANET/Source:



In order to automatically copy the compiled file in the service Bin directory, adds the following command in the Project/Settings menu release version:



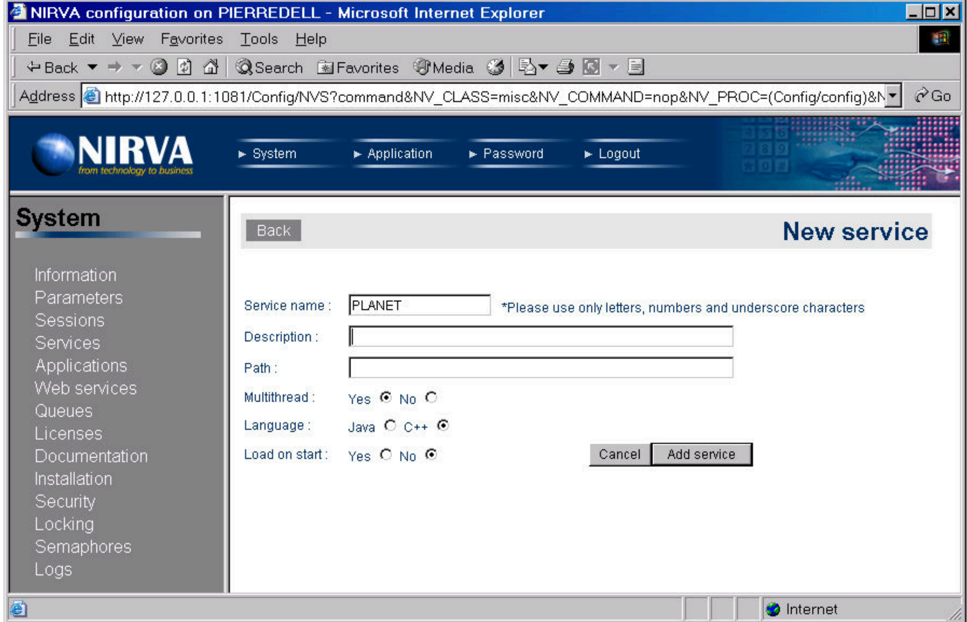
Then compile the release version:



Now the second step is finished

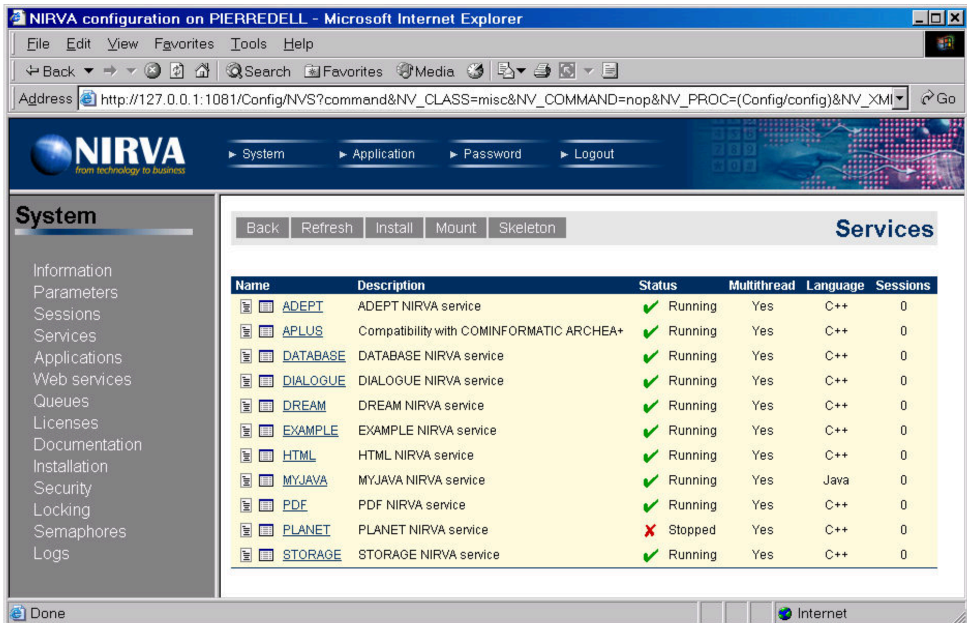
Mounting the service

We must now mount the service in the NIRVA service manager. For that, one can come back to the Nirva configuration tool in the systems/service menu and press the “Mount” button. Then edit the form as follow:



Don’t forget to set the service language as C++ (default is Java).

Then press the “Add service” button. If there is no error message, the service has been mounted and appears in the Nirva service list:

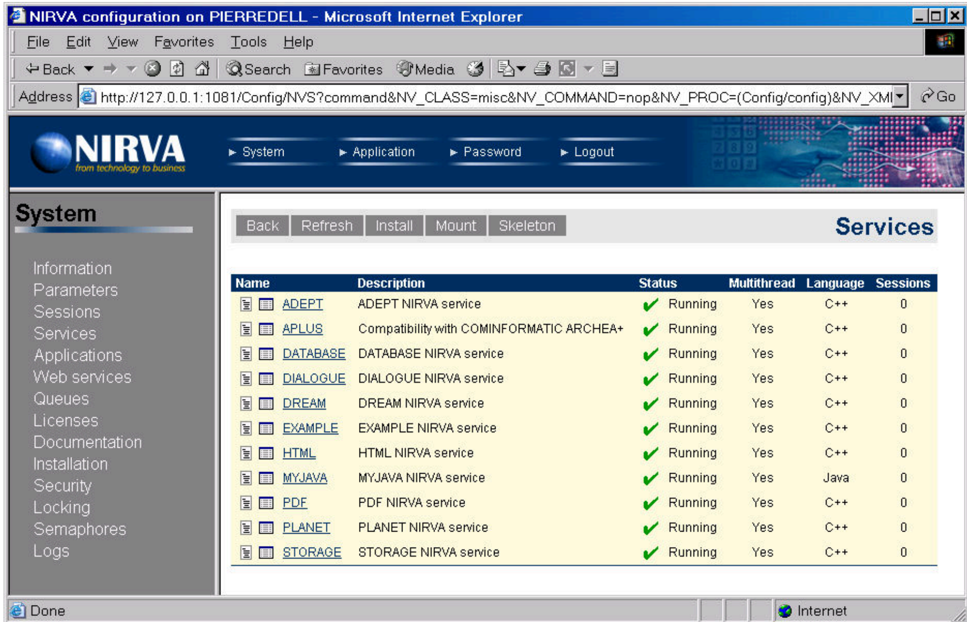


Name	Description	Status	Multithread	Language	Sessions
ADEPT	ADEPT NIRVA service	Running	Yes	C++	0
APLUS	Compatibility with COMINFORMATIC ARCHEA+	Running	Yes	C++	0
DATABASE	DATABASE NIRVA service	Running	Yes	C++	0
DIALOGUE	DIALOGUE NIRVA service	Running	Yes	C++	0
DREAM	DREAM NIRVA service	Running	Yes	C++	0
EXAMPLE	EXAMPLE NIRVA service	Running	Yes	C++	0
HTML	HTML NIRVA service	Running	Yes	C++	0
MYJAVA	MYJAVA NIRVA service	Running	Yes	Java	0
PDF	PDF NIRVA service	Running	Yes	C++	0
PLANET	PLANET NIRVA service	Stopped	Yes	C++	0
STORAGE	STORAGE NIRVA service	Running	Yes	C++	0

Now the third step is finished

Starting the service

For starting the service, just press the ✗ icon at the left of the service status in the service list.



Now the fourth step is finished

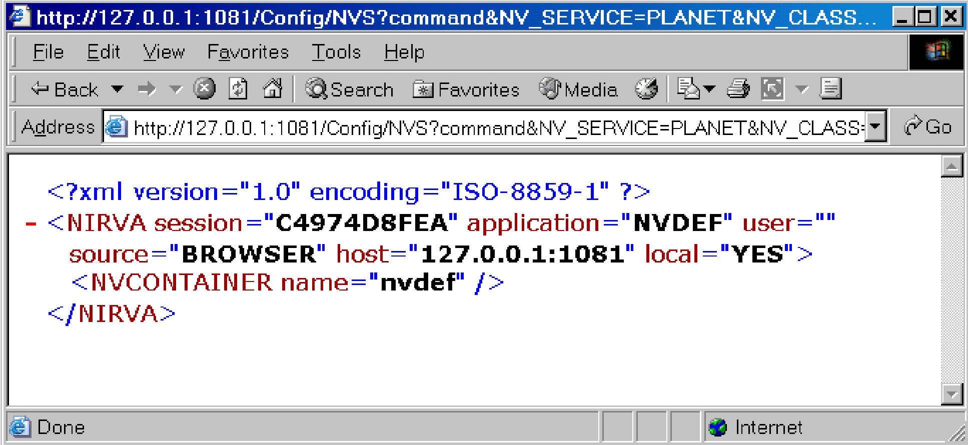
Testing the service

The final step is to test the service. Testing a service means sending a command to it. The service skeleton automatically creates the first service command named “NOP” that does nothing but allows this test. The command class to use is the same than the service name: “PLANET”.

For testing, one just has to send this URL from a browser (be careful, this requires that the default user has no password defined otherwise you’ll get an error message):

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:misc:nop&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no

This should display some XML data on your browser:

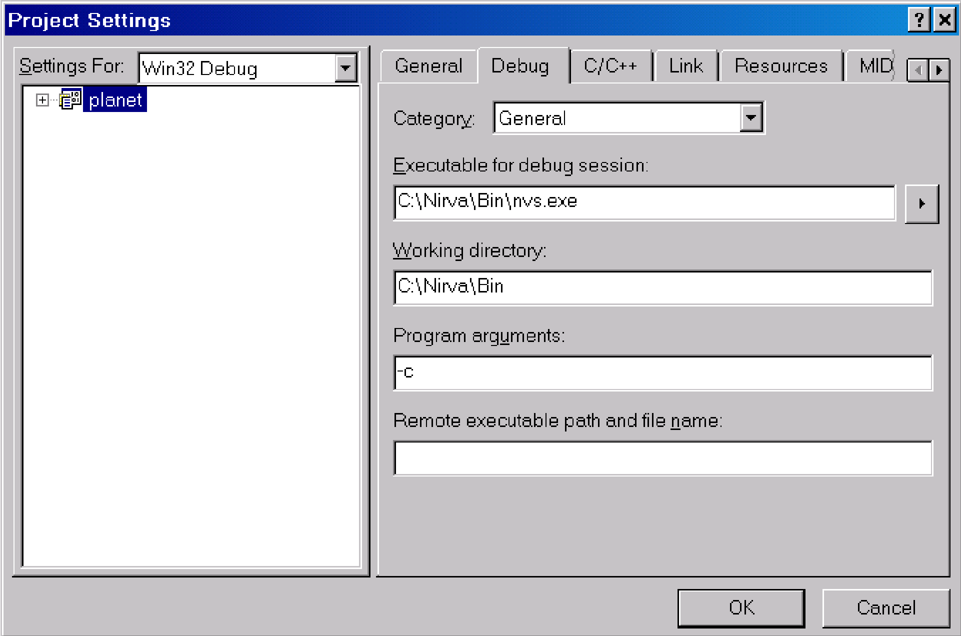


Now the last step is finished and our new PLANET service is functional.

The continuation of this example will show how to work with some more service functionality.

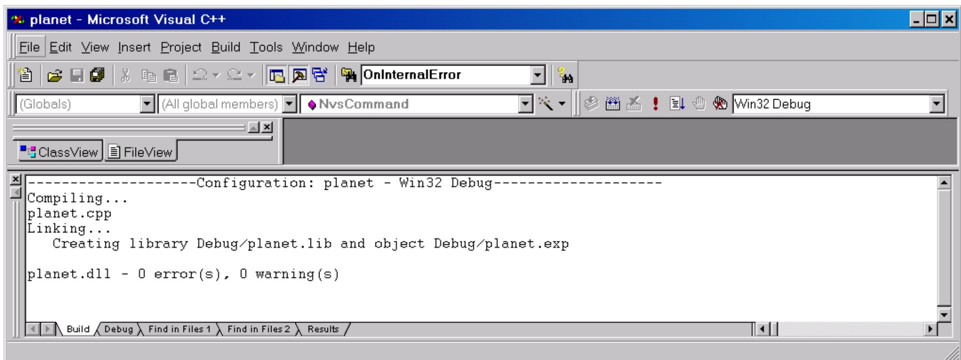
Debugging the service

For debugging the service, the best way is to stop the NIRVA server (nvs) and to set it as executable for debug in the project settings:



This will set the NIRVA server in console mode as debug program.

Now we can compile the PLANET service in debug mode:

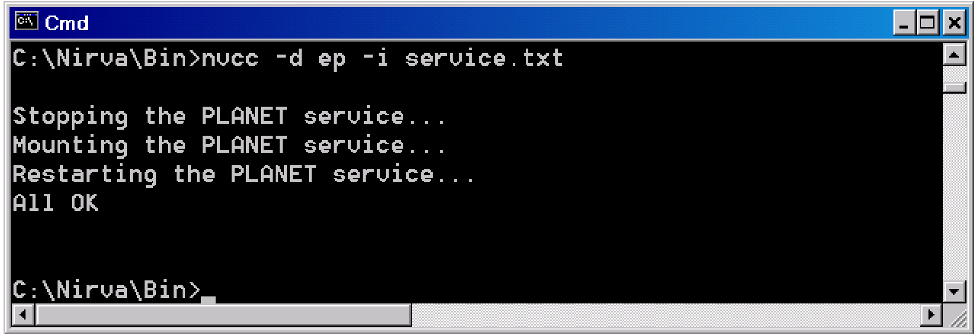


In order to be able to debug, the service debug version must be mounted on NIRVA. For the next steps, we will prepare a command file that will do all necessary stuff to be able to work in debug mode without any worry. For that, let’s edit a file named service.txt in the Nirva/Bin directory:

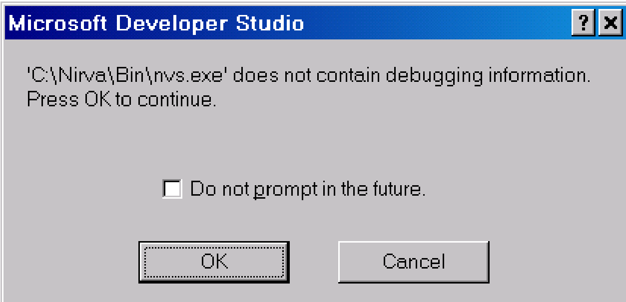
```
; NIRVA planet service debug
; 03/07/2002

; First stop the service if already running
nvcc::printf \nStopping the PLANET service…
NV_CMD=|service:stop| SERVICE=|Planet|

; Service debug mount
nvcc::printf \nMounting the PLANET service…
NV_CMD=|service:mount| SERVICE=|Planet|
```

```
PATH=|C:\Nirva\Services\Planet\Source\Debug\Planet.dll|

; Service start
nvcc::printf \nRestarting the PLANET service…
NV_CMD=|service:start| SERVICE=|Planet|

nvcc::printf \nAll OK\n\n
```

Now open a DOS console, go into the Nirva/Bin directory and type "`nvcc -i service.txt`" command from the client console in order to run this script (the NIRVA server must be started before):



Then stop the NIRVA server and run the service debug (F5 key from VISUAL STUDIO). The first time, VISUAL STUDIO will ask:

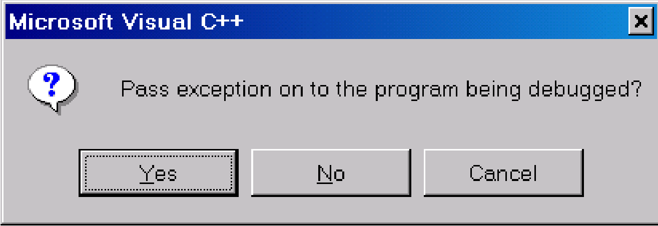


This is normal because the executable is a release version. Please check the button "Do not prompt in the future" if you don't want this message to be displayed any more and press OK to continue.

Then the service runs in debug mode.

Note: when issuing a CTRL+C from the NIRVA (nvs) console in order to stop the server while in debug mode, VISUAL STUDIO will display this message:

This is normal so you should press the OK button and continue the debugging (F5 key). Then VISUAL STUDIO will ask the following:



Please answer yes because NIRVA is catching this exception and terminates properly after at least 2 seconds.

Adding service functionality

Now the service is created and runs but it does really nothing. We'll now implement a command named PLANET GET that creates a NIRVA string list object containing the planet names.

For that, let's modify the PLANETSession::OnCommand function in the following way:

```
// Called for each Nirva command to the service
// This is the command entry point for session
bool PLANETSession::OnCommand(NvServiceCommand *Command)
{
  // TODO
  // Insert command processing here

  if(Command->IsCommand("PLANET", "GET"))
  {
    // Create the PLANETS string list object
    Command->Command("NV_CMD=|OBJECT:CREATE| NAME=|PLANETS| TYPE=|STRINGLIST|");

    // Populate the object with planet names
    Command->Command ("NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NAME=|PLANETS| SEPARATOR=|;|
                VALUE=|MERCURY;VENUS;EARTH;MARS;JUPITER;SATURN;URANUS;NEPTUNE;PLUTO|");
  }

  // Everything is OK
  return true;
}
```

The Command->IsCommand function test a given command while the Command->Command function sends a command to another service (here the NIRVA SYTEM service because the NV_SERVICE parameter is omitted).

After the code modification and compilation, please stop and restart the service from the nirva configuration tool (system/service menu).

In order to view the result, we can send the command PLANET PLANET GET from a web browser. Here is the URL to call for that:

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:PLANET:GETt&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no&NV_XML_SIMPLE=yes

We can see the result from the web browser:

```
http://127.0.0.1:1081/Config/NVS?command&NV_SERVICE=PLANET&NV_CLASS=PLANET...

<?xml version="1.0" encoding="ISO-8859-1" ?>
- <nirva session="275FCF8FF9" application="NVDEF" user=""
    source="BROWSER" host="127.0.0.1:1081" local="YES">
  - <planets type="stringlist">
      <data>MERCURY</data>
      <data>VENUS</data>
      <data>EARTH</data>
      <data>MARS</data>
      <data>JUPITER</data>
      <data>SATURN</data>
      <data>URANUS</data>
      <data>NEPTUNE</data>
      <data>PLUTO</data>
    </planets>
  </nirva>
```

We can see the “planets” object created by the PLANET PLANET GET command.

The result is in XML when the command has been sent from a browser. Let’s do now an html view. For that, we create the following XSL file named “planets.xsl” in the default application file directory (Nirva/Applications/NVDEF/Files):

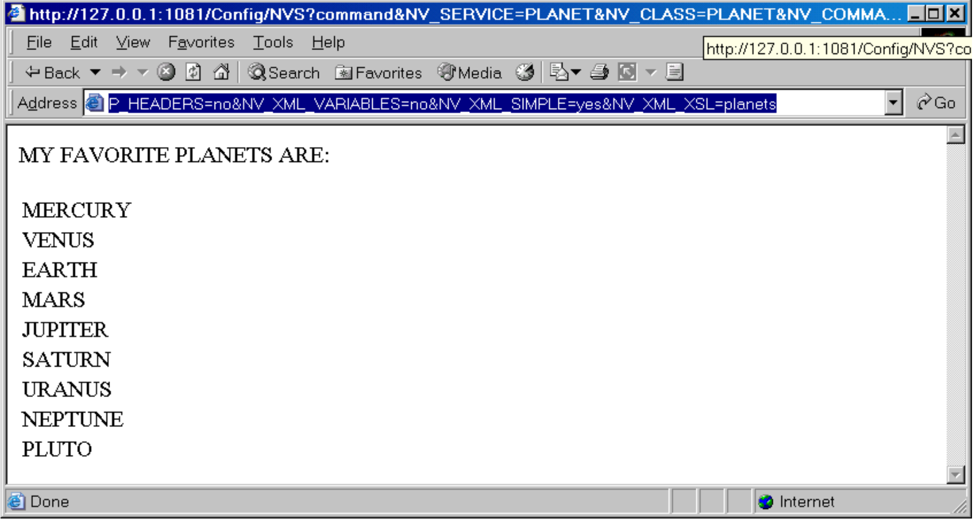
```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <html>
  <body>
  <xsl:apply-templates select="nirva"/>
  </body>
  </html>
</xsl:template>

<xsl:template match="nirva">
  <P>MY FAVORITE PLANETS ARE:</P>
  <table>
  <xsl:apply-templates select="planets/data"/>
  </table>
</xsl:template>

<xsl:template match="planets/data">
  <tr>
          <td><xsl:value-of select="."/></td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

The purpose of this file is not making a beautiful display but just to show how to arrange the NIRVA display using integrated XML features.

Now change the URL in order to display the result as html by calling the planets.xsl file:

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:PLANET:GET&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no&NV_XML_SIMPLE=yes&NV_XML_XSL=planets

Here is the result in the web browser:



Adding error codes

In this lesson, we'll add an error code for the error code class PLANET that will be returned if the command is not known by the service.

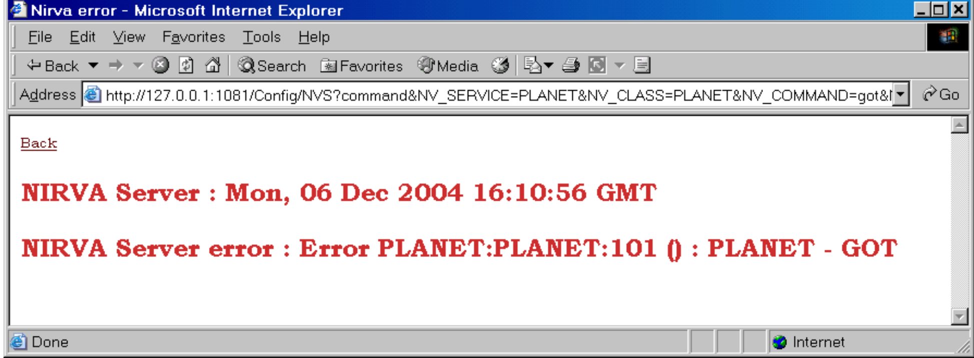
Let's modify again our PLANETSession::OnCommand function in the following way:

```
// Called for each Nirva command to the service
// This is the command entry point for session
bool MYSERVICESession::OnCommand(NvServiceCommand *Command)
{
  // TODO
  // Insert command processing here

  if(Command->IsCommand("PLANET", "GET"))
  {
    // Create the PLANETS string list object
    Command->Command("NV_CMD=|OBJECT:CREATE| NAME=|PLANETS| TYPE=|STRINGLIST|");

    // Populate the object with planet names
    Command->Command ("NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NAME=|PLANETS| SEPARATOR=|;|
                VALUE=|MERCURY;VENUS;EARTH;MARS;JUPITER;SATURN;URANUS;NEPTUNE;PLUTO|");
    // Everything is OK
    return true;
   }

  // Unknown command
  // Prepare an error information and return it to NIRVA
  char Buffer[1024];
  char ErrorInfo[1024];
```

```
  strcpy(ErrorInfo, "");

  // Get command class
  Command->GetClass(Buffer, sizeof(Buffer));
  strcat(ErrorInfo, Buffer);
  strcat(ErrorInfo, " - ");

  // Get command name
  Command->GetCommand(Buffer, sizeof(Buffer));
  strcat(ErrorInfo, Buffer);

  // Set the error message


  Command->SetError("PLANET", 101, ErrorInfo);

  return false;
}
```

After the code modification and compilation, please stop and restart the service form the nirva configuration tool (system/service menu).

Now we try to generate an error by sending a bad URL. We change the command name from “get” to “got”. Here is the result in the browser:

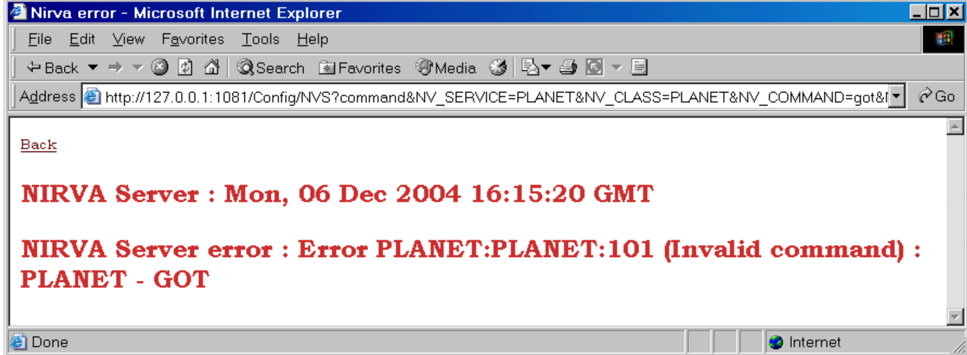


There is one thing more missing. We have the error class (PLANET), the error code (101), the error info (PLANET –GOT) but there is no error description. As we already saw, the error description is maintained in the service description file. Let’s modify the ERROR_CLASS_PLANET section of the service description file (in Nirva/Service/PLANET/Files):

```
[ERROR_CLASS_PLANET]
0   = No error;pas d'erreur;Keine Störung;Nessun errore;Ningún error
101 = Invalid command;commande non reconnue
```

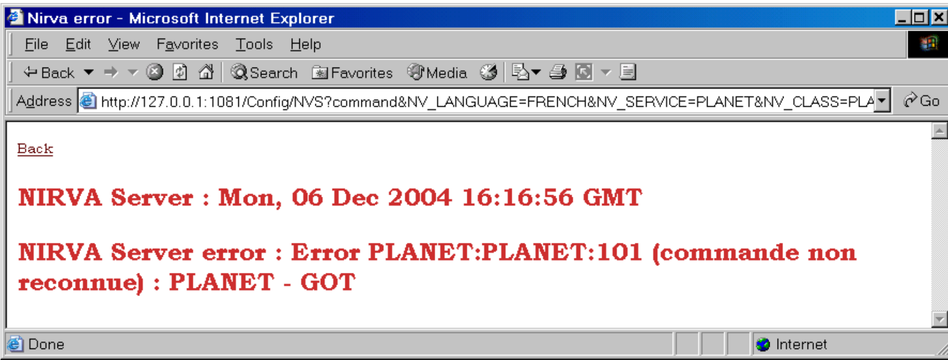
This section defines the error codes for the PLANET error class. Now we must stop and start again the NIRVA server nvs in order for the new content of the description file to be taken in care or we can just stop and restart the PLANET service from the nirva configuration tool (system/service menu).

Than, let’s try again our bad command from the web browser:



We have now the error description.

We can try it in French by adding the NV_LANGUAGE parameter (set to “FRENCH”) to the command:



Creating license

NIRVA provides all the mechanism for service providers to create and control their license.

A license is always linked to a unique machine ID. In order to get the local machine ID, we can use the getmachine.txt command file that resides in the Nirva/Bin directory. Here is the content of this file:

```
; NIRVA get machine identifier

NV_CMD=|license:info|
NV_CMD=|OBJECT:GET| NAME=|MACHINE_ID|
NV_CMD =|Local:OBJECT:STRING_GET_VALUE| NAME=|MACHINE_ID|
nvcc::printf \nMachine ID:
nvcc::printdata
nvcc::printf \n
```

```
Cmd
C:\nirva\Bin>nvcc -i getmachine.txt

Machine ID: 9ECCEB85CE58BDF057A6DA8C124CD393

C:\nirva\Bin>
```

This machine identifier must be kept somewhere.

When the service has been created, Nirva also created a file named “license.txt” in the PLANET/Source directory. This file contains a private and a public service ID. The private service ID is used to generate service license channels and the public service ID is used to check the license channels from inside the service code.

```
Here is an example of license.txt file:
PLANET Nirva service


This file contains the uniq service IDs you must use to create and check your license

Two service IDs are defined: the public service ID and the private service ID.

The public service ID must be used from inside your service in order to check
license channels that you have created. You check licenses by using the
SYSTEM LICENSE GET command.

The private service ID must be used only for creating new license channels by the way
of the Nirva nvl tool. The private service ID is confidential and must not be delivered
to anybody.


private service ID : 1E040EC4AB141DF7AD185B5206E90AA2
public service ID  : 21DE2F4F19C32C0A0A15C89D40D5B663
```

In our example, we will just create a license key for running the service. We’ll call it “RUN”.

Practically, a service provider distributes the license by giving a license file to the customer. The first step is for the customer to give its machine identifier to the service provider. This one then creates a license file and sends it back to the customer. Finally, the customer installs the license file to the NIRVA license manager.

We already saw the first step so we must now create the license file. For that, we use the following command file that we name “planet_license.txt”:

```
; NIRVA license creation for planet service
; 03/07/2002

NV_CMD=|license:set| /
  FILE=|license| /
  MACHINE_ID=|#param1| /
  SERVICE=|Planet| /
  SERVICE_ID=|1E040EC4AB141DF7AD185B5206E90AA2| /
  key=|RUN|
NV_CMD=|object:get| name=|license| Filename=|#param2|
```

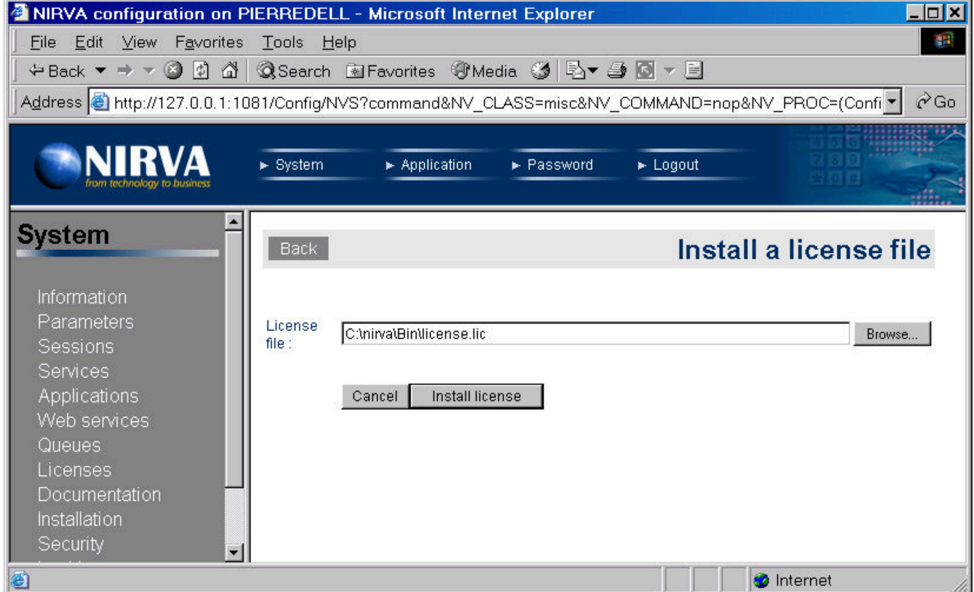
Please replace the SERVICE_ID parameter with your real service ID private key found in the “license.txt” file.

This command file first creates a license file on a server file object named “license” with the correct keys and then gets back this license file to the client in a file given as parameter. We use the following command to execute the license_create.txt file:

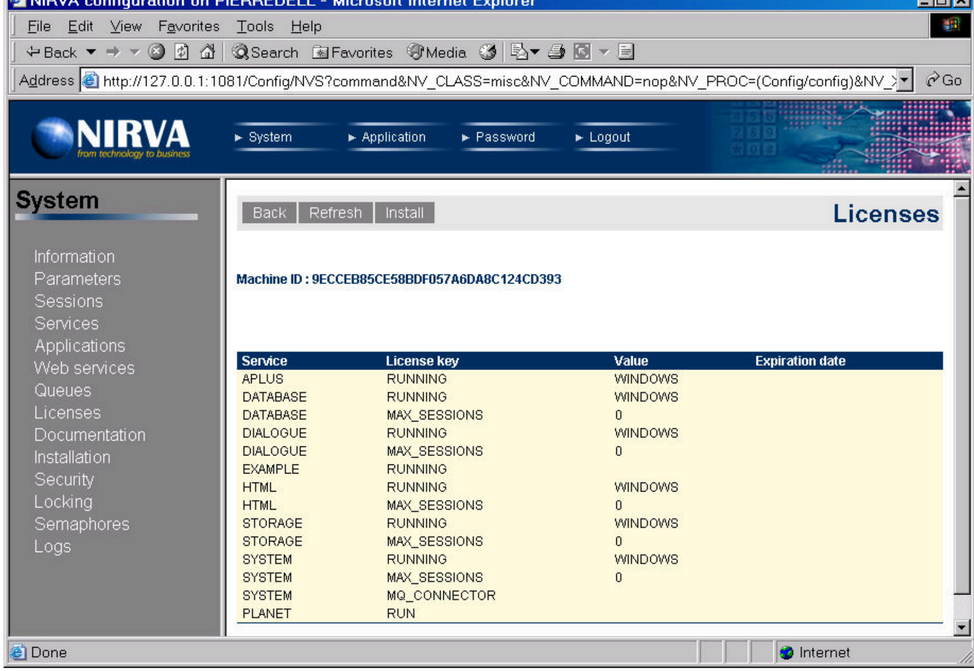
```
nvcc -i planet_license.txt 9ECCEB85CE58BDF057A6DA8C124CD393 license.lic
```

In this command you must replace the parameter “`9ECCEB85CE58BDF057A6DA8C124CD393`” with your real machine ID.

Now we have the license file in license.lic so we must import it to NIRVA. Let’s do that from the configuration tool in the system/license menu by pressing the install button:



You must browse your just created license.lic file in order to install it to Nirva server. Then, the license channel has been added to Nirva:



You can see the PLANET RUN license chanel added to the list.

License can also be created using the nvl.exe (windows tool) delivered in the Nirva/Bin directory. (Please see the chapter Tools/nvl in this documentation for further information).

Checking license

We know now how to generate license files and how to distribute them to the customers but we need to make some license control inside the service code itself.

A good place for doing that is when a new session is created on service side so we can add the license control code in the PLANETSession::OnInit function:

```
// Called one time when initializing the session
bool PLANETSession::OnInit(NvServiceCommand *Command)
{
  if(Initialized)
        return true;

  // TODO
  // Insert eventual initialization code here

  // Checks the RUN license key
  char CommandBuffer[1024];
  strcpy(CommandBuffer, "NV_CMD=|LICENSE:GET| SERVICE=|PLANET|");
  strcat(CommandBuffer, " KEY=|RUN| SERVICE_ID=|21DE2F4F19C32C0A0A15C89D40D5B663|");

  if(!Command->Command(CommandBuffer))
        return false;  // No license found

  // All is OK
  Initialized = true;
  return true;
}
```

Please replace the SERVICE_ID parameter value with you real service ID public key.

This is a simple way to check the license. Now it’s also possible to associate a value to a license key. We could use this feature in our example for controlling the maximum number of simultaneous sessions in the service planet.

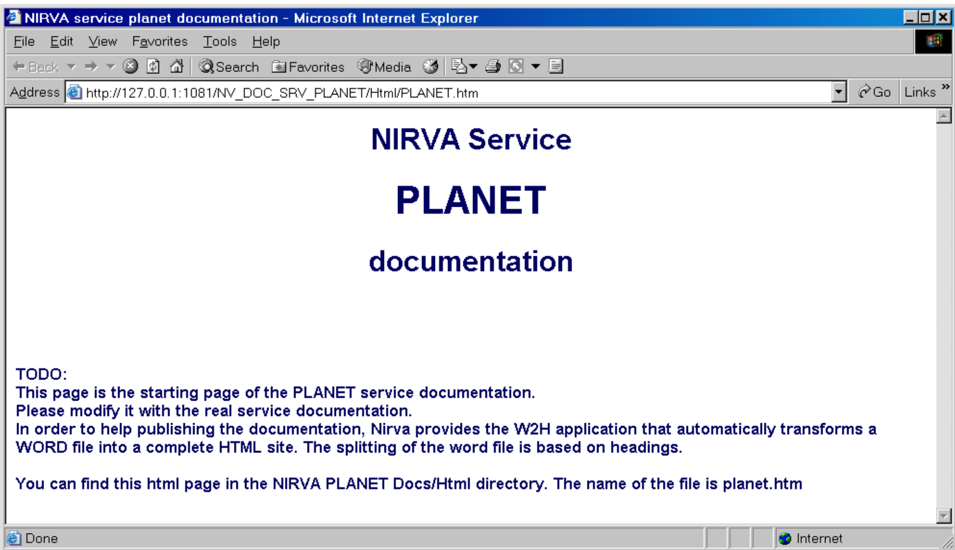
Please consult the SYSTEM service LICENSE class for further information about licenses.

Documentation

The service documentation is directly available by an URL from a web browser. It’s the responsibility of the service provider to write this documentation.

The skeleton creates a file named “servicename.htm” where servicename is replaced by the name of the service (in lowercase) in the service Html documentation directory (in our example in

Nirva/Service/PLANET/Docs/Html). This is the entry point of the service documentation. Here is the way to call it from the web browser:



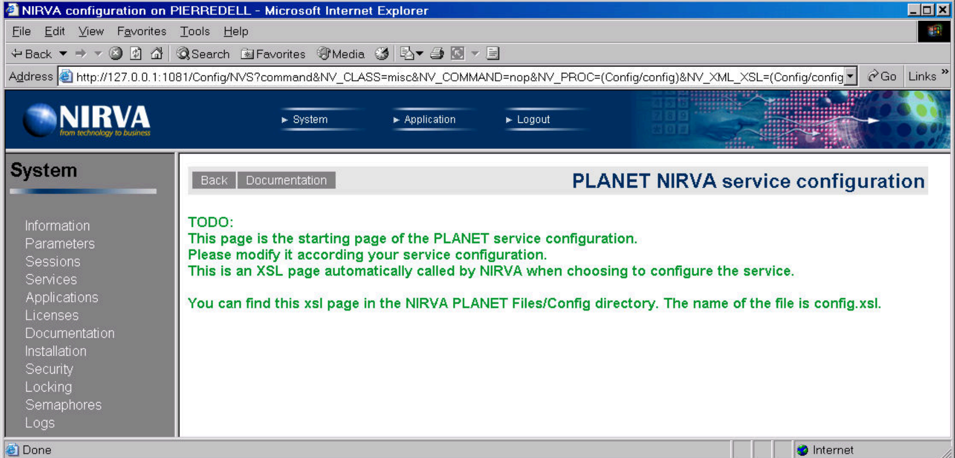
This URL points directly to Nirva/Service/PLANET/Docs/html/planet.htm

Configuration

In the same way than for the documentation, the service configuration is directly available by an URL from a web browser. It's the responsibility of the service provider to write the code of the configuration.

The configuration is made by using the XML and XSL capabilities of NIRVA.

The skeleton creates a file named "config.xsl" in the service config files directory (in our example in Nirva/Service/PLANET/Files/Config). This XSL file is used by NIRVA when calling the SYSTEM SERVICE CONFIG command from a web browser:



This commands first call a procedure named "config.nvp" in the service config procedure directory (here Nirva/Services/PLANET/Config/Procs directory) then prepares the content of the output container in XML data and finally calls the XML parser to transforms the XML data using the "config.xsl" file into viewable html code.

The service provider just has to modify the “config.xsl” and “config.nvp” files in order to set the service configuration.

Packaging the service

The SYSTEM SERVICE SKELETON command has also created a file named “package.lst” that resides in the service files directory (here Nirva/Services/PLANET/Files directory). This file is a package description file that will be used to create an installation package file with the help of the SYSTEM SERVICE PACKAGE command.

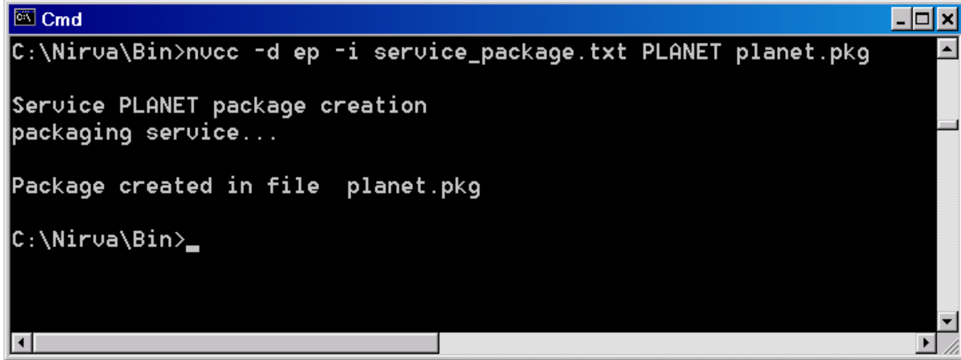
Here is the content of the package.lst file:

```
// package.lst : installation package listing
// for NIRVA SERVICE PLANET


// Header section
// This section is transmitted as it is to the package file
[HEADER]
SERVICE = PLANET

// All files and subdirectories of SERVICE Bin directory
[/Bin]
copydirsub:

// Description file
[/Files]
copyfile:service.dsc

// All files and subdirectories of SERVICE Files/Config directory
[/Files/Config]
copydirsub:

// Removes SERVICE Docs/Html directory
[/Docs/Html]
removedirsub:

// All files and subdirectories of SERVICE Docs directory
[/Docs]
copydirsub:

// All files and subdirectories of SERVICE Procs directory
[/Procs]
copydirsub:

// All files and subdirectories of SERVICE Wroot directory
[/Wroot]
copydirsub:
```

The format of the package file is described in the “installation packages” chapter.

We must have compiled the service in release mode or verify that the planet.dll file has been copied into the service Bin directory (here Nirva/Services/PLANET/Bin directory).

In order to create the package file, we can use the standard command file service_package.txt delivered in the NIRVA Bin directory. Here is the command:

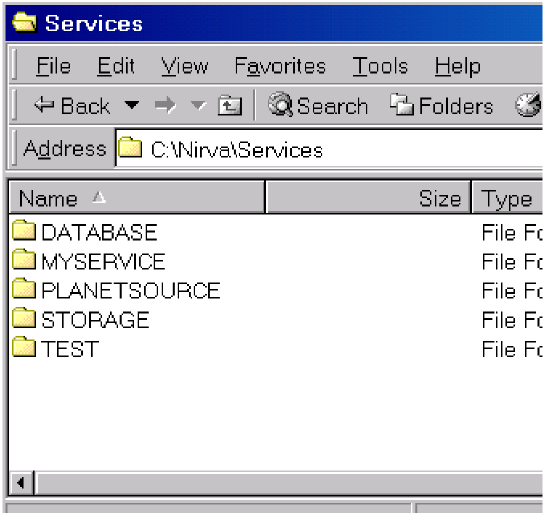


This command has created the local package file named “planet.pkg”. We can now use it to install the service.

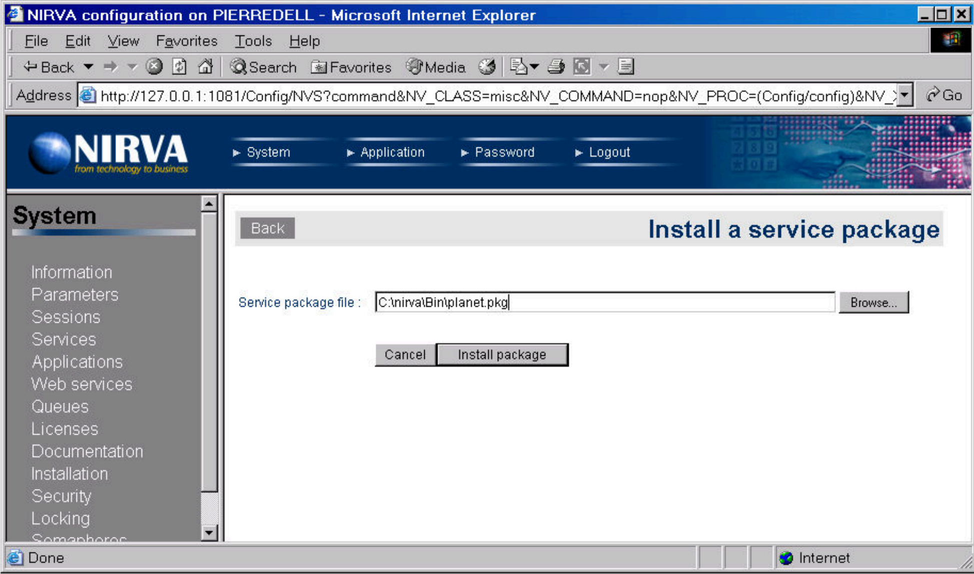
The package can also be created directly from the Nirva configuration tool.

Installing the service

For the service installation, we must consider that we are a customer receiving our service. For that, we will temporary rename the NIRVA PLANET service directory from PLANET to PLANETSOURCE. This is just for demonstrating the installation:



Now we use the Nirva configuration to do the service installation. Open the Nirva configuration tool, go to the systems/Services menu and press install:



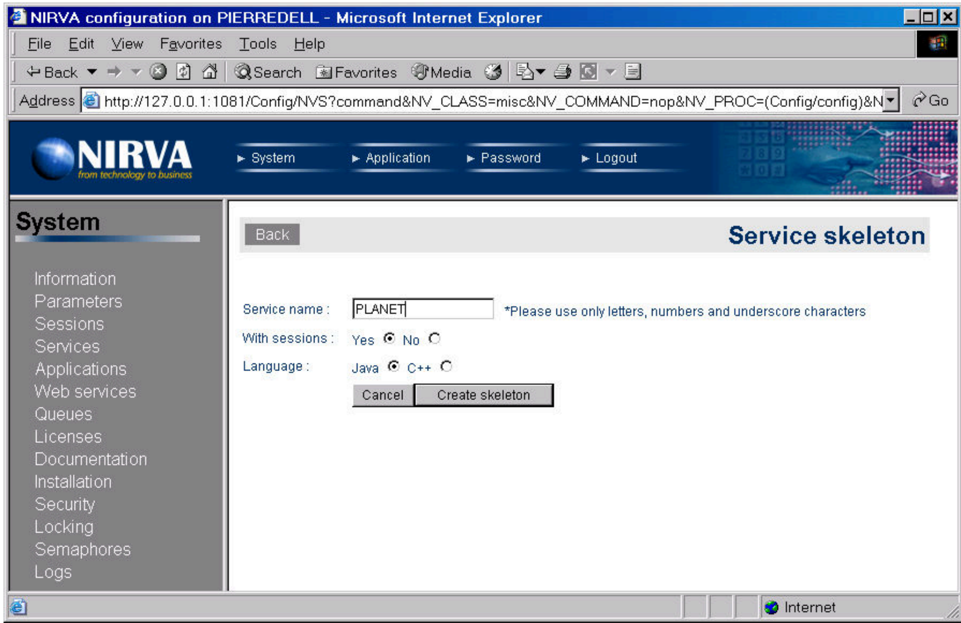
After confirmation, the service should have been installed on the target platform.

Java service

For compilation, the example assumes that a JDK (at least 1.3 version) is installed on your computer.

Creating the skeleton

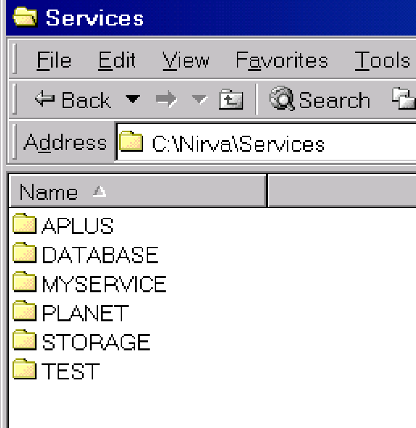
The first step is to create the service skeleton. For that one just need to run the Nirva configuration tool, go to the system/service menu and press the “Skeleton” button. Then edit the form as follow:



Don’t forget to set the service language as Java (default).

Then press the “Create skeleton” button. If there is no error message, the skeleton has been created on the server side.

This can be verified by checking the NIRVA services directory. A new directory named PLANET should have been created:



Now the first step is finished

Compiling the service

In order to compile, just open a DOS console, go into the Nirva/Services/PLANET/Source directory and run

```
javac planet.java
```

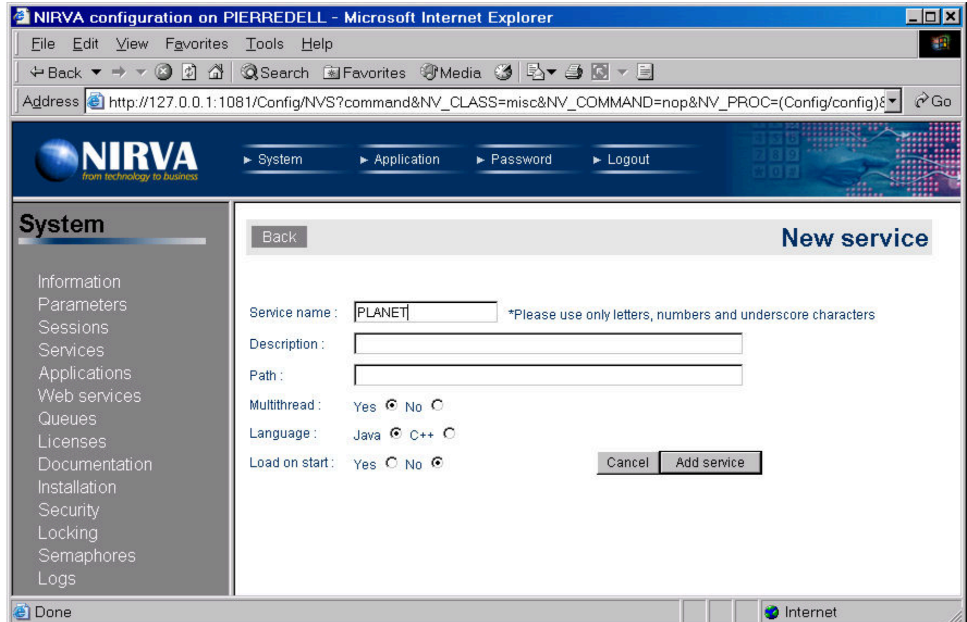
```
Cmd
C:\nirva\Services\PLANET\Source>javac planet.java

C:\nirva\Services\PLANET\Source>_
```

This creates to classes named planet.class and planetsession.class. Please copy these files into the Nirva/Services/PLANET/Bin directory.

Now the second step is finished

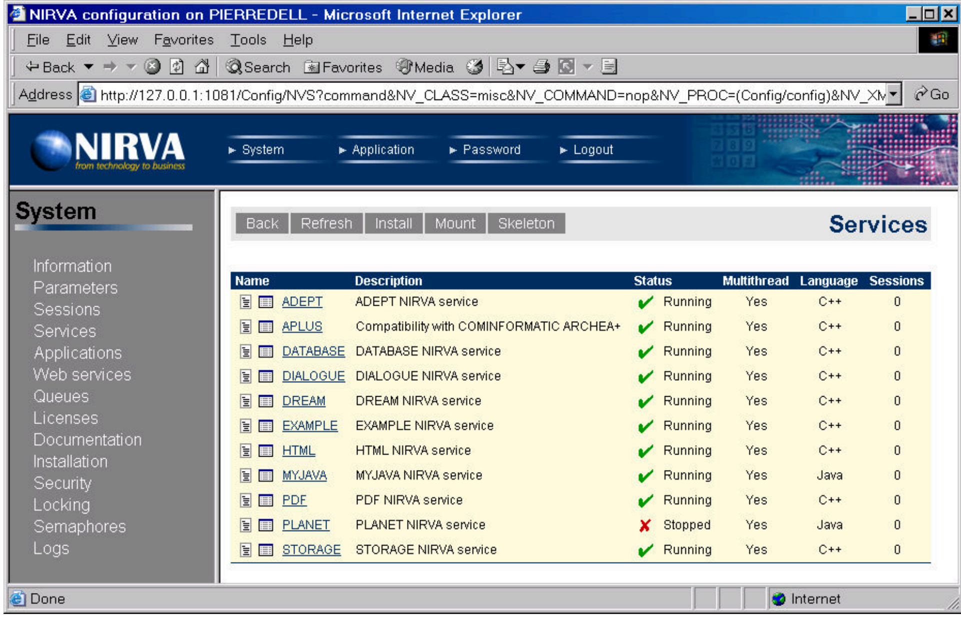
Mounting the service

We must now mount the service in the NIRVA service manager. For that, one can come back to the Nirva configuration tool in the systems/service menu and press the “Mount” button. Then edit the form as follow:



Don’t forget to set the service language as Java (default).

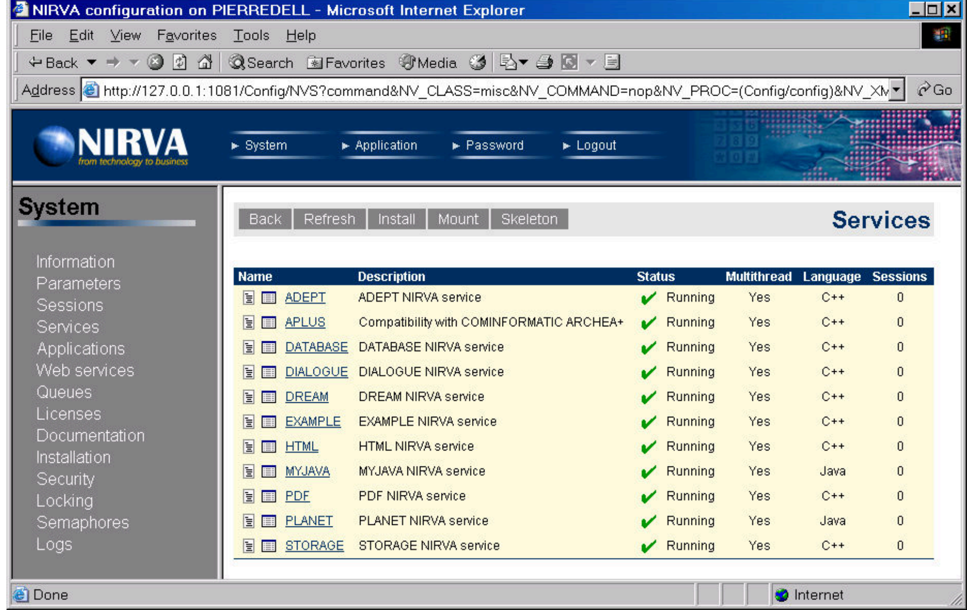
Then press the “Add service” button. If there is no error message, the service has been mounted and appears in the Nirva service list:



Now the third step is finished

Starting the service

For starting the service, just press the ✗ icon at the left of the service status in the service list.



Now the fourth step is finished

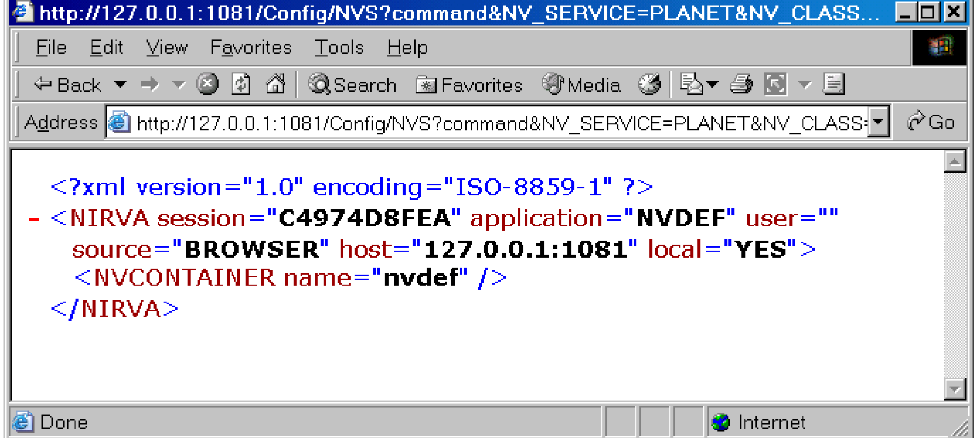
Testing the service

The final step is to test the service. Testing a service means sending a command to it. The service skeleton automatically creates the first service command named “NOP” that does nothing but allows this test. The command class to use is the same than the service name: “PLANET”.

For testing, one just has to send this URL from a browser (be careful, this requires that the default user has no password defined otherwise you’ll get an error message):

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:misc:nop&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no

This should display some XML data on your browser:



Now the last step is finished and our new PLANET service is functional.

The continuation of this example will show how to work with some more service functionality.

Adding service functionality

Now the service is created and runs but it does really nothing. We'll now implement a command named PLANET GET that creates a NIRVA string list object containing the planet names.

For that, let's modify the planetsession::OnCommand method in the following way:

```
// Called for each Nirva command to the service
// This is the command entry point for session
public boolean OnCommand(nvcmd Command)
{
  // TODO
  // Insert command processing here

  if(Command.IsCommand("PLANET", "GET", ""))
  {
    // Create the PLANETS string list object
    String sCommand = "NV_CMD=|OBJECT:CREATE| NAME=|PLANETS| TYPE=|STRINGLIST|";
     Command.Command(sCommand);

    // Populate the object with planet names
    sCommand = "NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NAME=|PLANETS| SEPARATOR=|;|";
    sCommand += " VALUE=|MERCURY;VENUS;EARTH;MARS;JUPITER;SATURN;URANUS;NEPTUNE;PLUTO|";
    Command.Command (sCommand);
  }

  // Everything is OK
  return true;
}
```

The Command.IsCommand method tests a given command while the Command.Command function sends a command to another service (here the NIRVA SYTEM service because the NV_SERVICE parameter is omitted).

After the code modification and compilation, please stop the service from the nirva configuration tool (system/service menu), then copy the planetsession.class file into the service Bin directory and restart the service.

In order to view the result, we can send the command PLANET PLANET GET from a web browser. Here is the URL to call for that:

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:PLANET:GETt&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no&NV_XML_SIMPLE=yes

We can see the result from the web browser:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <nirva session="275FCF8FF9" application="NVDEF" user=""
    source="BROWSER" host="127.0.0.1:1081" local="YES">
  - <planets type="stringlist">
      <data>MERCURY</data>
      <data>VENUS</data>
      <data>EARTH</data>
      <data>MARS</data>
      <data>JUPITER</data>
      <data>SATURN</data>
      <data>URANUS</data>
      <data>NEPTUNE</data>
      <data>PLUTO</data>
    </planets>
  </nirva>
```

We can see the “planets” object created by the PLANET PLANET GET command.

The result is in XML when the command has been sent from a browser. Let’s do now an html view. For that, we create the following XSL file named “planets.xsl” in the default application file directory (Nirva/Applications/NVDEF/Files):

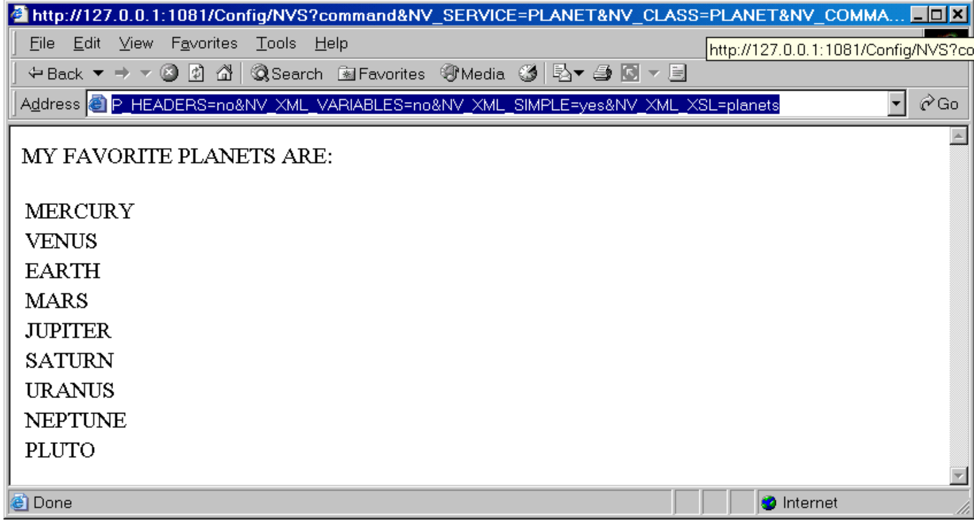
```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <html>
  <body>
  <xsl:apply-templates select="nirva"/>
  </body>
  </html>
</xsl:template>

<xsl:template match="nirva">
  <P>MY FAVORITE PLANETS ARE:</P>
  <table>
  <xsl:apply-templates select="planets/data"/>
  </table>
</xsl:template>

<xsl:template match="planets/data">
  <tr>
          <td><xsl:value-of select="."/></td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

The purpose of this file is not making a beautiful display but just to show how to arrange the NIRVA display using integrated XML features.

Now change the URL in order to display the result as html by calling the planets.xsl file:

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:PLANET:GET&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no&NV_XML_SIMPLE=yes&NV_XML_XSL=planets

Here is the result in the web browser:



Adding error codes

In this lesson, we'll add an error code for the error code class PLANET that will be returned if the command is not known by the service.

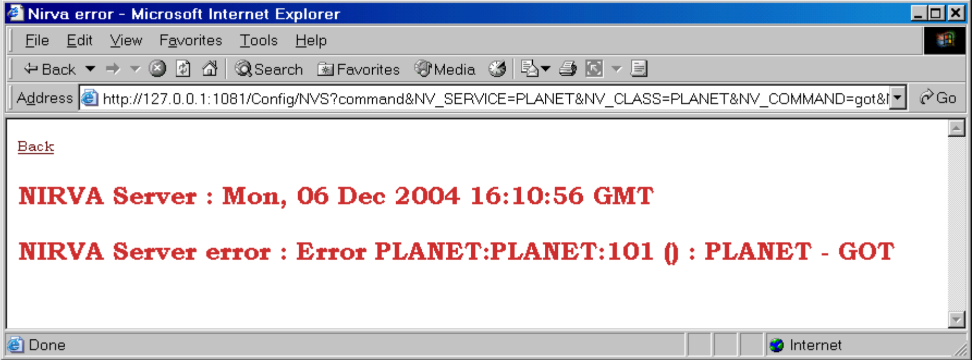
Let's modify again our PLANETSession::OnCommand function in the following way:

```
// Called for each Nirva command to the service
// This is the command entry point for session
public boolean OnCommand(nvcmd Command)
{
  // TODO
  // Insert command processing here

  if(Command.IsCommand("PLANET", "GET", ""))
  {
    // Create the PLANETS string list object
    String sCommand = "NV_CMD=|OBJECT:CREATE| NAME=|PLANETS| TYPE=|STRINGLIST|";
     Command.Command(sCommand);

    // Populate the object with planet names
    sCommand = "NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NAME=|PLANETS| SEPARATOR=|;|";
    sCommand += " VALUE=|MERCURY;VENUS;EARTH;MARS;JUPITER;SATURN;URANUS;NEPTUNE;PLUTO|";
    Command.Command (sCommand);
    // Everything is OK
    return true;
  }

  // Unknown command
  // Prepare an error information and return it to NIRVA
  String ErrorInfo = Command.GetClass() + " - " + Command.GetCommand();
```

```
// Set the error message
  Command.SetError("PLANET", 101, ErrorInfo);

  return false;
}
```

After the code modification and compilation, please stop and restart the service form the nirva configuration tool (system/service menu).

Now we try to generate an error by sending a bad URL. We change the command name from “get” to “got”. Here is the result in the browser:

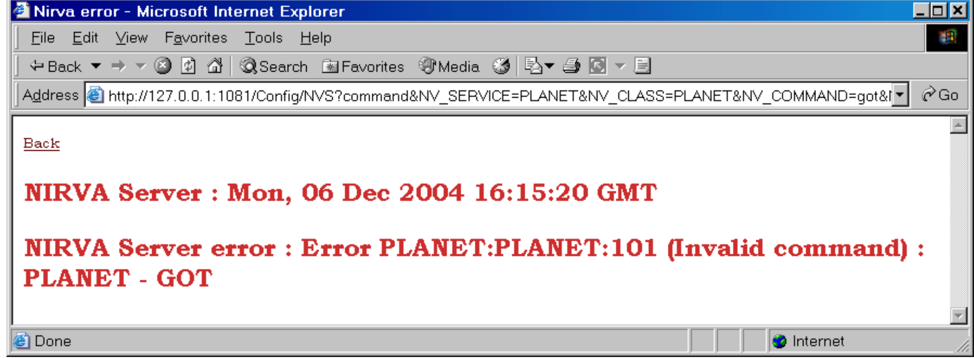


There is one thing more missing. We have the error class (PLANET), the error code (101), the error info (PLANET –GOT) but there is no error description. As we already saw, the error description is maintained in the service description file. Let’s modify the ERROR_CLASS_PLANET section of the service description file (in Nirva/Service/PLANET/Files):

```
[ERROR_CLASS_PLANET]
0   = No error;pas d'erreur;Keine Störung;Nessun errore;Ningún error
101 = Invalid command;commande non reconnue
```

This section defines the error codes for the PLANET error class. Now we must stop and start again the NIRVA server nvs in order for the new content of the description file to be taken in care or we can just stop and restart the PLANET service from the nirva configuration tool (system/service menu).

Than, let’s try again our bad command from the web browser:



We have now the error description.

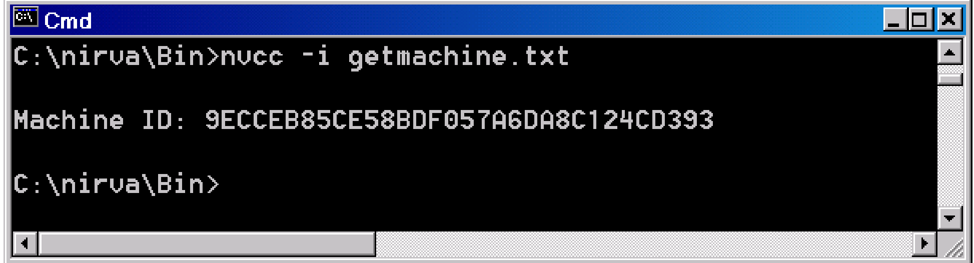
We can try it in French by adding the NV_LANGUAGE parameter (set to “FRENCH”) to the command:

Creating license

NIRVA provides all the mechanism for service providers to create and control their license.

A license is always linked to a unique machine ID. In order to get the local machine ID, we can use the getmachine.txt command file that resides in the Nirva/Bin directory. Here is the content of this file:

```
; NIRVA get machine identifier

NV_CMD=|license:info|
NV_CMD=|OBJECT:GET| NAME=|MACHINE_ID|
NV_CMD =|Local:OBJECT:STRING_GET_VALUE| NAME=|MACHINE_ID|
nvcc::printf \nMachine ID:
nvcc::printdata
nvcc::printf \n
```



This machine identifier must be kept somewhere.

When the service has been created, Nirva also created a file named “license.txt” in the PLANET/Source directory. This file contains a private and a public service ID. The private service ID is used to generate service license channels and the public service ID is used to check the license channels from inside the service code.

```
Here is an example of license.txt file:
PLANET Nirva service
```

```
This file contains the uniq service IDs you must use to create and check your license

Two service IDs are defined: the public service ID and the private service ID.

The public service ID must be used from inside your service in order to check
license channels that you have created. You check licenses by using the
SYSTEM LICENSE GET command.

The private service ID must be used only for creating new license channels by the way
of the Nirva nvl tool. The private service ID is confidential and must not be delivered
to anybody.


private service ID : 1E040EC4AB141DF7AD185B5206E90AA2
public service ID  : 21DE2F4F19C32C0A0A15C89D40D5B663
```

In our example, we will just create a license key for running the service. We'll call it "RUN".

Practically, a service provider distributes the license by giving a license file to the customer. The first step is for the customer to give its machine identifier to the service provider. This one then creates a license file and sends it back to the customer. Finally, the customer installs the license file to the NIRVA license manager.

We already saw the first step so we must now create the license file. For that, we use the following command file that we name "planet_license.txt":

```
; NIRVA license creation for planet service
; 03/07/2002

NV_CMD=|license:set| /
  FILE=|license| /
  MACHINE_ID=|#param1| /
  SERVICE=|Planet| /
  SERVICE_ID=|1E040EC4AB141DF7AD185B5206E90AA2| /
  key=|RUN|
NV_CMD=|object:get| name=|license| Filename=|#param2|
```

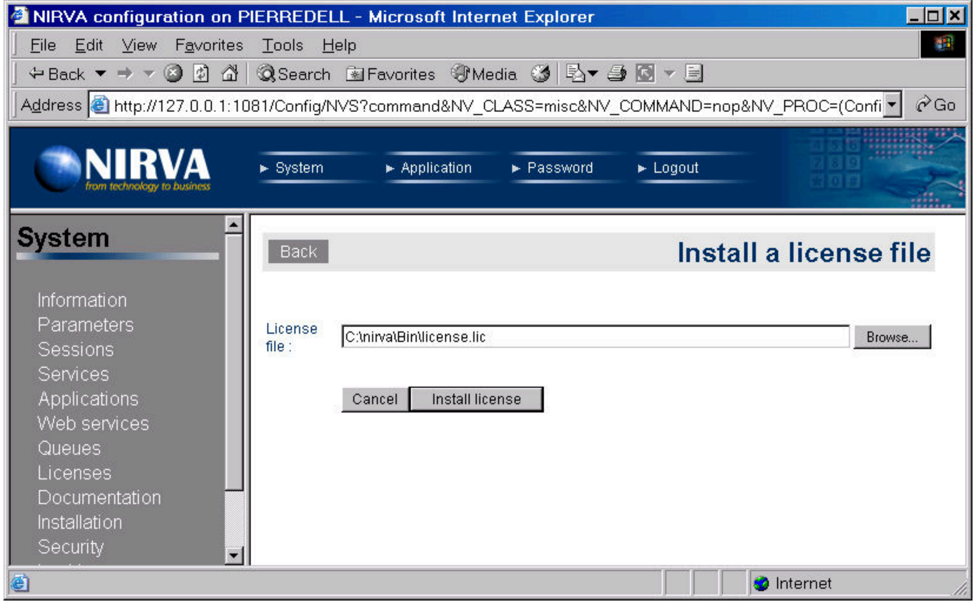
Please replace the SERVICE_ID parameter with your real service ID private key found in the "license.txt" file.

This command file first creates a license file on a server file object named "license" with the correct keys and then gets back this license file to the client in a file given as parameter. We use the following command to execute the license_create.txt file:

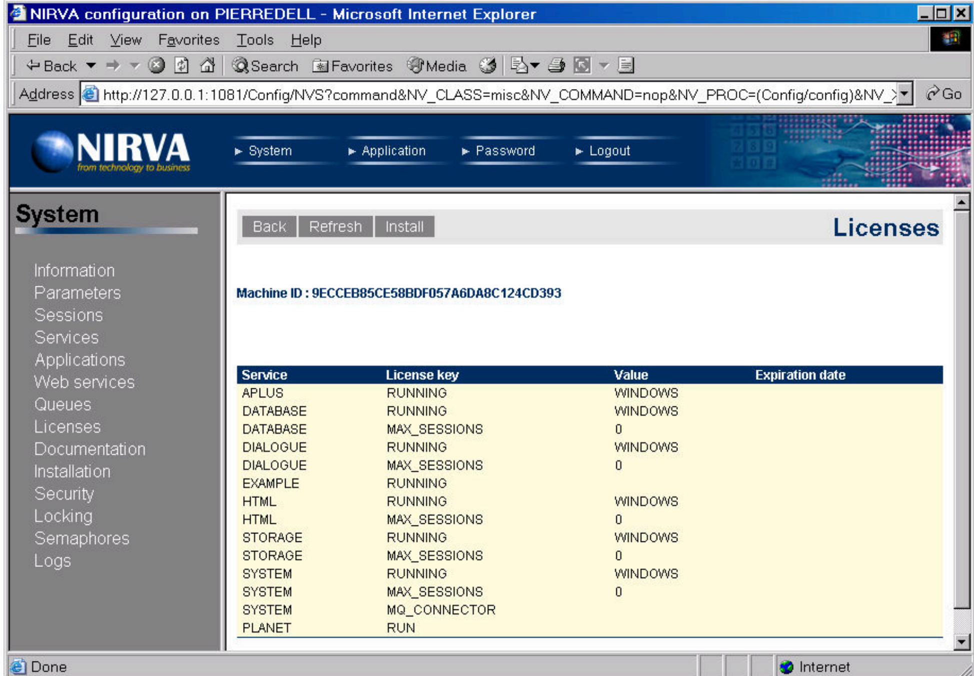
```
nvcc -i planet_license.txt 9ECCEB85CE58BDF057A6DA8C124CD393 license.lic
```

In this command you must replace the parameter "`9ECCEB85CE58BDF057A6DA8C124CD393`" with your real machine ID.

Now we have the license file in license.lic so we must import it to NIRVA. Let's do that from the configuration tool in the system/license menu by pressing the install button:



You must browse your just created license.lic file in order to install it to Nirva server. Then, the license channel has been added to Nirva:



Service	License key	Value	Expiration date
APLUS	RUNNING	WINDOWS	
DATABASE	RUNNING	WINDOWS	
DATABASE	MAX_SESSIONS	0	
DIALOGUE	RUNNING	WINDOWS	
DIALOGUE	MAX_SESSIONS	0	
EXAMPLE	RUNNING		
HTML	RUNNING	WINDOWS	
HTML	MAX_SESSIONS	0	
STORAGE	RUNNING	WINDOWS	
STORAGE	MAX_SESSIONS	0	
SYSTEM	RUNNING	WINDOWS	
SYSTEM	MAX_SESSIONS	0	
SYSTEM	MQ_CONNECTOR		
PLANET	RUN		

You can see the PLANET RUN license chanel added to the list.

License can also be created using the nvl.exe (windows tool) delivered in the Nirva/Bin directory. (Please see the chapter Tools/nvl in this documentation for further information).

Checking license

We know now how to generate license files and how to distribute them to the customers but we need to make some license control inside the service code itself.

A good place for doing that is when a new session is created on service side so we can add the license control code in the planetsession::OnInit method:

```
// Called one time when initializing the session
public boolean OnInit(nvcmd Command)
{

  // TODO
  // Insert eventual initialization code here

  // Checks the RUN license key
  String sCommand = "NV_CMD=|LICENSE:GET| SERVICE=|PLANET|";
  sCommand += " KEY=|RUN| SERVICE_ID=|21DE2F4F19C32C0A0A15C89D40D5B663|";

  if(Command.Command(sCommand) == 0)
         return false;  // No license found

  // All is OK
  return true;
}
```

Please replace the SERVICE_ID parameter value with you real service ID public key.

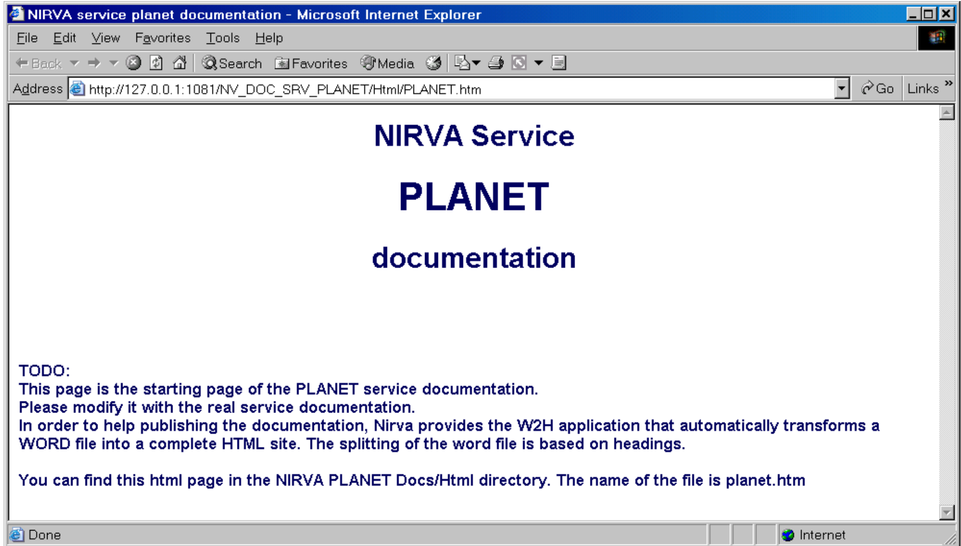
This is a simple way to check the license. Now it’s also possible to associate a value to a license key. We could use this feature in our example for controlling the maximum number of simultaneous sessions in the service planet.

Please consult the SYSTEM service LICENSE class for further information about licenses.

Documentation

The service documentation is directly available by an URL from a web browser. It’s the responsibility of the service provider to write this documentation.

The skeleton creates a file named “servicename.htm” where servicename is replaced by the name of the service (in lowercase) in the service Html documentation directory (in our example in Nirva/Service/PLANET/Docs/Html). This is the entry point of the service documentation. Here is the way to call it from the web browser:



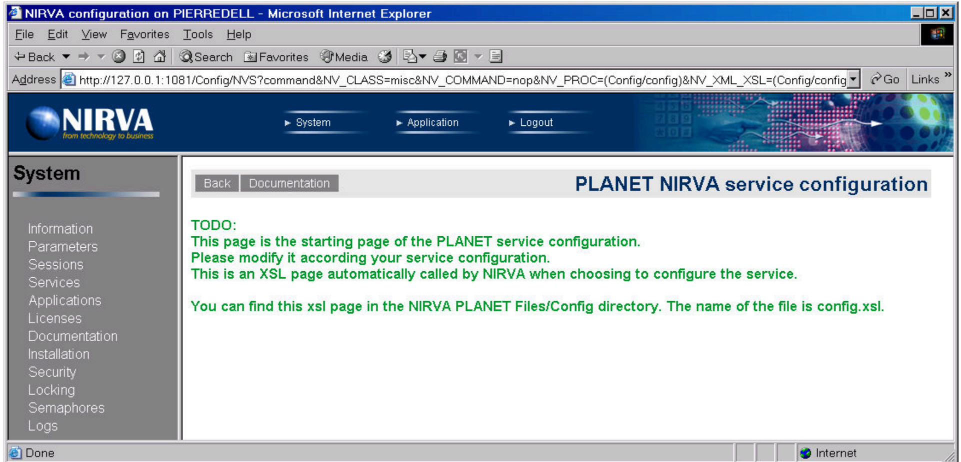
This URL points directly to Nirva/Service/PLANET/Docs/html/planet.htm

Configuration

In the same way than for the documentation, the service configuration is directly available by an URL from a web browser. It's the responsibility of the service provider to write the code of the configuration.

The configuration is made by using the XML and XSL capabilities of NIRVA.

The skeleton creates a file named "config.xsl" in the service config files directory (in our example in Nirva/Service/PLANET/Files/Config). This XSL file is used by NIRVA when calling the SYSTEM SERVICE CONFIG command from a web browser:



This commands first call a procedure named "config.nvp" in the service config procedure directory (here Nirva/Services/PLANET/Config/Procs directory) then prepares the content of the output container in XML data and finally calls the XML parser to transforms the XML data using the "config.xsl" file into viewable html code.

The service provider just has to modify the "config.xsl" and "config.nvp" files in order to set the service configuration.

Packaging the service

The SYSTEM SERVICE SKELETON command has also created a file named “package.lst” that resides in the service files directory (here Nirva/Services/PLANET/Files directory). This file is a package description file that will be used to create an installation package file with the help of the SYSTEM SERVICE PACKAGE command.

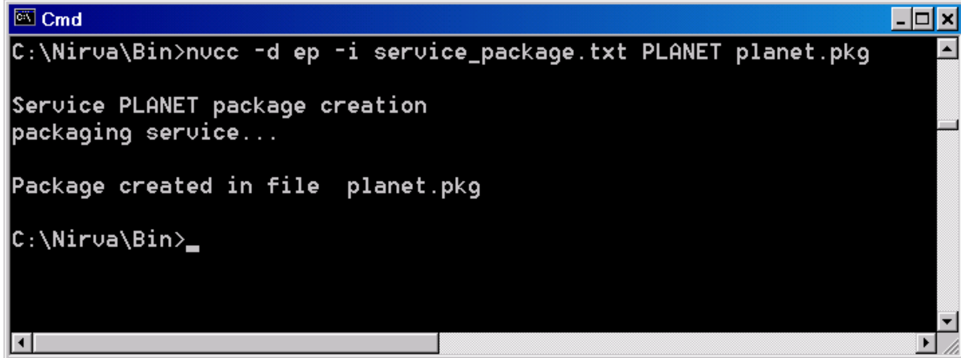
Here is the content of the package.lst file:

```
// package.lst : installation package listing
// for NIRVA SERVICE PLANET


// Header section
// This section is transmitted as it is to the package file
[HEADER]
SERVICE = PLANET

// All files and subdirectories of SERVICE Bin directory
[/Bin]
copydirsub:

// Description file
[/Files]
copyfile:service.dsc

// All files and subdirectories of SERVICE Files/Config directory
[/Files/Config]
copydirsub:

// Removes SERVICE Docs/Html directory
[/Docs/Html]
removedirsub:

// All files and subdirectories of SERVICE Docs directory
[/Docs]
copydirsub:

// All files and subdirectories of SERVICE Procs directory
[/Procs]
copydirsub:

// All files and subdirectories of SERVICE Wroot directory
[/Wroot]
copydirsub:
```

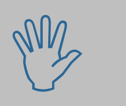
The format of the package file is described in the “installation packages” chapter.

We must have compiled the service in release mode and verify that the planet class files has been copied into the service Bin directory (here Nirva/Services/PLANET/Bin directory).

In order to create the package file, we can use the standard command file service_package.txt delivered in the NIRVA Bin directory. Here is the command:



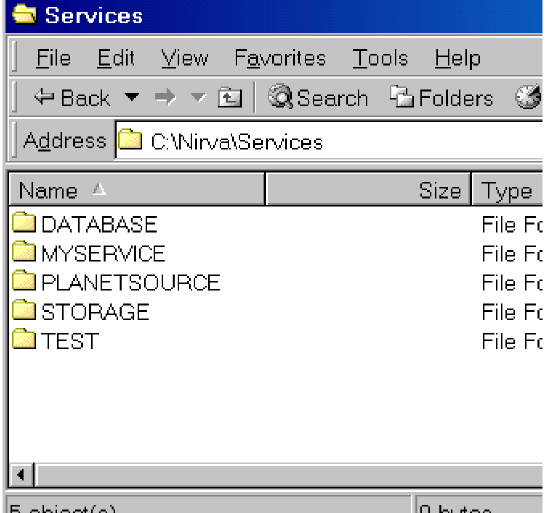
This command has created the local package file named “planet.pkg”. We can now use it to install the service.



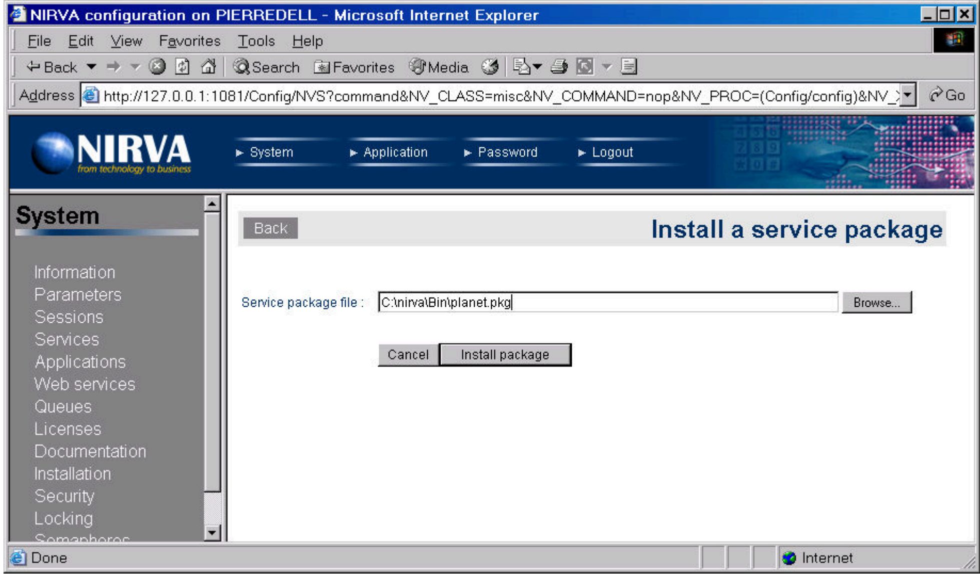
The package can also be created directly from the Nirva configuration tool.

Installing the service

For the service installation, we must consider that we are a customer receiving our service. For that, we will temporary rename the NIRVA PLANET service directory from PLANET to PLANETSOURCE. This is just for demonstrating the installation:



Now we use the Nirva configuration to do the service installation. Open the Nirva configuration tool, go to the systems/Services menu and press install:



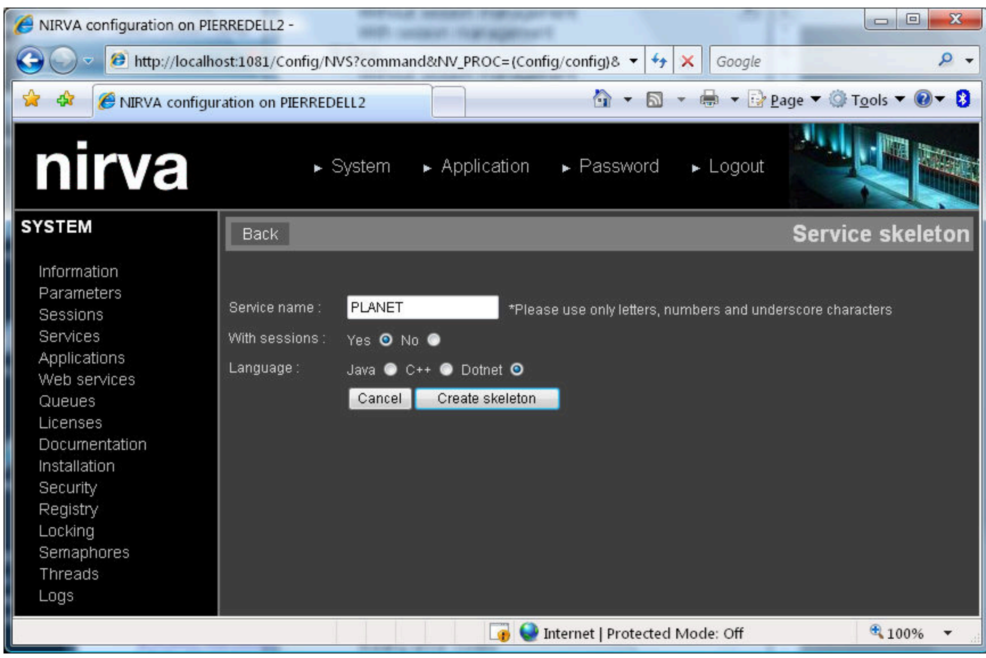
After confirmation, the service should have been installed on the target platform.

Dotnet service

For compilation, the example assumes that the Dotnet framework (at least version 3.5) is installed on your computer and that the csc.exe compiler is in your path.

Creating the skeleton

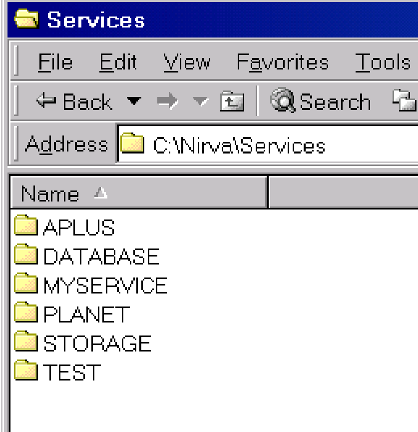
The first step is to create the service skeleton. For that one just need to run the Nirva configuration tool, go to the system/service menu and press the “Skeleton” button. Then edit the form as follow:



Don’t forget to set the service language as Dotnet.

Then press the “Create skeleton” button. If there is no error message, the skeleton has been created on the server side.

This can be verified by checking the NIRVA services directory. A new directory named PLANET should have been created:



Now the first step is finished

Compiling the service

In order to compile, just open a DOS console, go into the Nirva/Services/PLANET/Source directory and run

```
compile
```

```
Administrator: Command Prompt
C:\Nirva\Services\PLANET\Source>compile

C:\Nirva\Services\PLANET\Source>csc /target:library /r:../../../Bin/nirvadn.dll planet.cs
Microsoft (R) Visual C# 2008 Compiler version 3.5.30729.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.


C:\Nirva\Services\PLANET\Source>
```

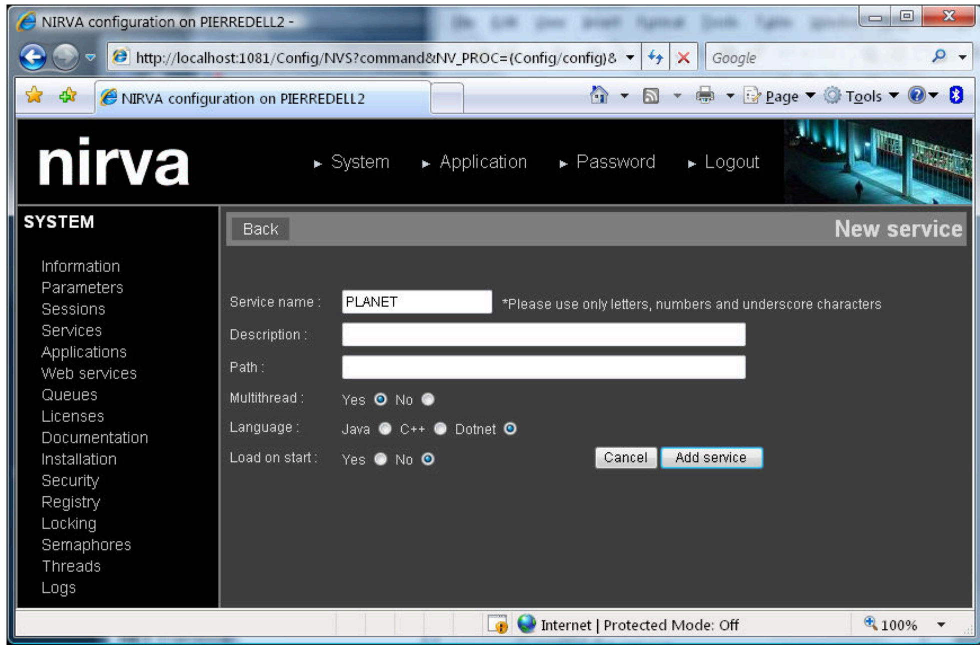
This creates a file named planet.dll. Please copy this file into the Nirva/Services/PLANET/Bin directory.

The tutorial is explained using the csc compiler from the command line. If the user has Visual C# installed, he can use it to open the project planet.sln in Nirva/Services/PLANET/Source directory.

Now the second step is finished

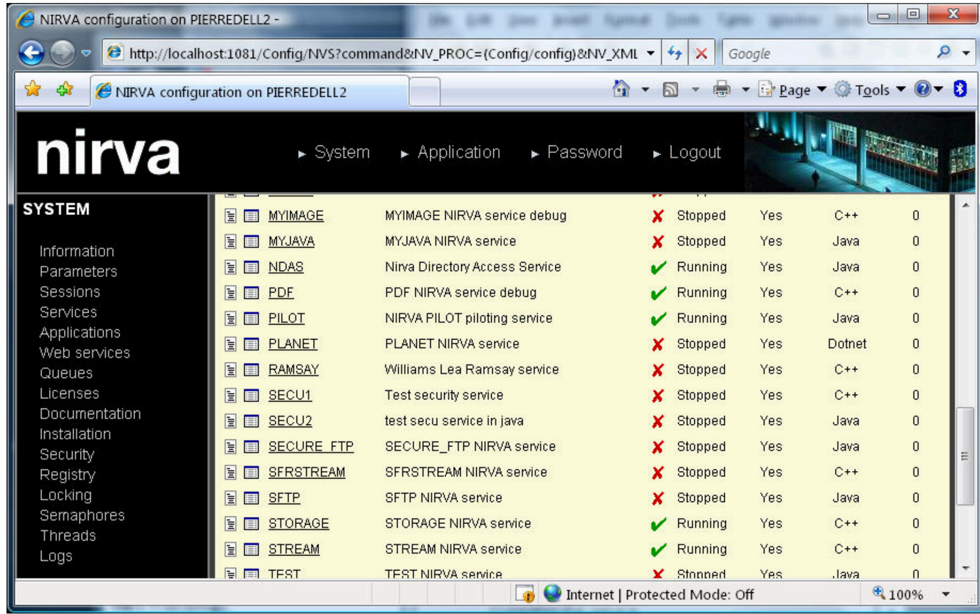
Mounting the service

We must now mount the service in the NIRVA service manager. For that, one can come back to the Nirva configuration tool in the systems/service menu and press the “Mount” button. Then edit the form as follow:



Don't forget to set the service language as Dotnet.

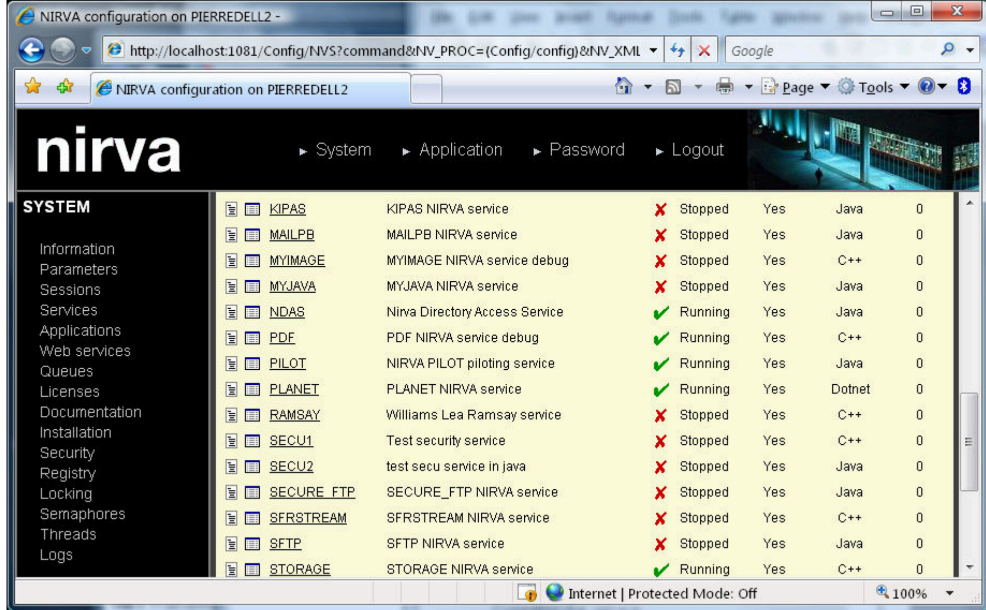
Then press the "Add service" button. If there is no error message, the service has been mounted and appears in the Nirva service list:



Now the third step is finished

Starting the service

For starting the service, just press the ✘ icon at the left of the service status in the service list.



Now the fourth step is finished

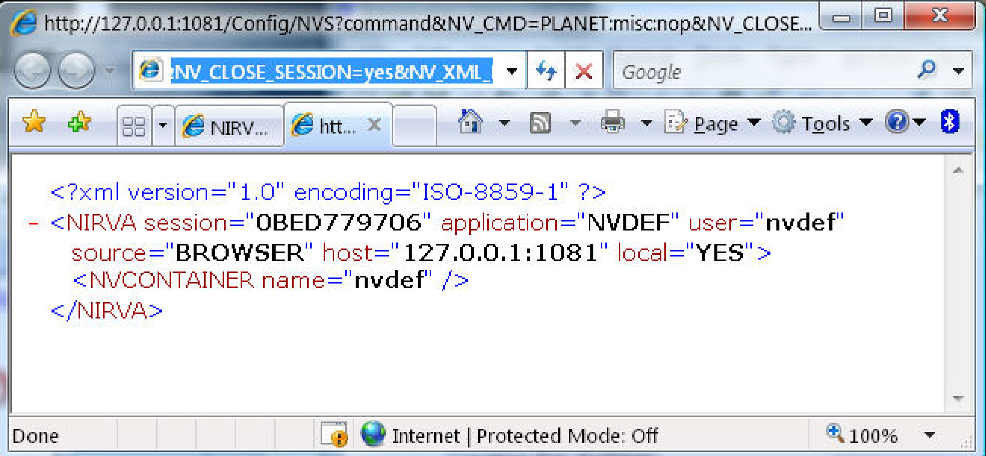
Testing the service

The final step is to test the service. Testing a service means sending a command to it. The service skeleton automatically creates the first service command named “NOP” that does nothing but allows this test. The command class to use is the same than the service name: “PLANET”.

For testing, one just has to send this URL from a browser (be careful, this requires that the default user has no password defined otherwise you’ll get an error message):

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:misc:nop&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no

This should display some XML data on your browser:



Now the last step is finished and our new PLANET service is functional.

The continuation of this example will show how to work with some more service functionality.

Adding service functionality

Now the service is created and runs but it does really nothing. We'll now implement a command named PLANET GET that creates a NIRVA string list object containing the planet names.

For that, let's modify the planetsession::OnCommand method in the following way:

```
// Called for each Nirva command to the service
// This is the command entry point for session
// return true if successful and false otherwise
public bool OnCommand(nvcmd Command)
{
  // TODO
  // Insert command processing here
   if (Command.IsCommand("PLANET", "GET", ""))
   {
     // Create the PLANETS string list object
     String sCommand = "NV_CMD=|OBJECT:CREATE| NAME=|PLANETS| TYPE=|STRINGLIST|";
     Command.Command(sCommand);

     // Populate the object with planet names
     sCommand ="NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NAME=|PLANETS| SEPARATOR=|;|";
     sCommand +=" VALUE=|MERCURY;VENUS;EARTH;MARS;JUPITER;SATURN;URANUS;NEPTUNE;PLUTON|";
     Command.Command(sCommand);
   }

  // Everything is OK
  return true;
}
```

The Command.IsCommand method tests a given command while the Command.Command function sends a command to another service (here the NIRVA SYSTEM service).

After the code modification and compilation, please stop the service from the nirva configuration tool (system/service menu), then copy the planetsession.class file into the service Bin directory and restart the service.

In order to view the result, we can send the command PLANET PLANET GET from a web browser. Here is the URL to call for that:

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:PLANET:GETt&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no&NV_XML_SIMPLE=yes

We can see the result from the web browser:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <nirva session="42B2D9478C" application="NVDEF" user="nvdef"
    source="BROWSER" host="127.0.0.1:1081" local="YES">
  - <planets type="stringlist">
      <data>MERCURY</data>
      <data>VENUS</data>
      <data>EARTH</data>
      <data>MARS</data>
      <data>JUPITER</data>
      <data>SATURN</data>
      <data>URANUS</data>
      <data>NEPTUNE</data>
      <data>PLUTON</data>
    </planets>
  </nirva>
```

We can see the “planets” object created by the PLANET PLANET GET command.

The result is in XML when the command has been sent from a browser. Let’s do now an html view. For that, we create the following XSL file named “planets.xsl” in the default application file directory (Nirva/Applications/NVDEF/Files):

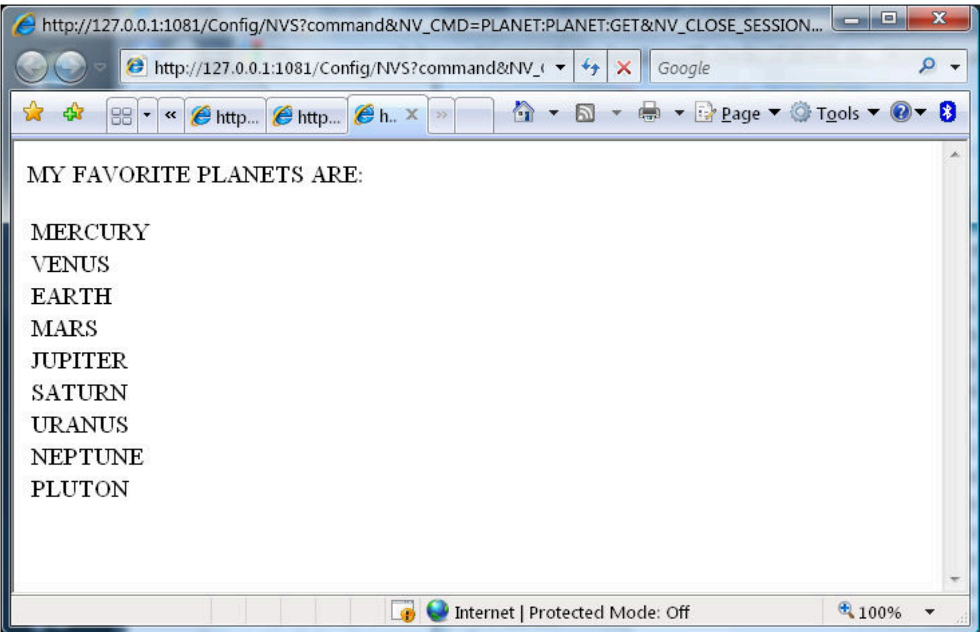
```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <html>
  <body>
  <xsl:apply-templates select="nirva"/>
  </body>
  </html>
</xsl:template>

<xsl:template match="nirva">
  <P>MY FAVORITE PLANETS ARE:</P>
  <table>
  <xsl:apply-templates select="planets/data"/>
  </table>
</xsl:template>

<xsl:template match="planets/data">
  <tr>
          <td><xsl:value-of select="."/></td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

The purpose of this file is not making a beautiful display but just to show how to arrange the NIRVA display using integrated XML features.

Now change the URL in order to display the result as html by calling the planets.xsl file:

http://127.0.0.1:1081/Config/NVS?command&NV_CMD=PLANET:PLANET:GET&NV_CLOSE_SESSION=yes&NV_XML_HTTP_HEADERS=no&NV_XML_VARIABLES=no&NV_XML_SIMPLE=yes&NV_XML_XSL=planets

Here is the result in the web browser:



Adding error codes

In this lesson, we'll add an error code for the error code class PLANET that will be returned if the command is not known by the service.

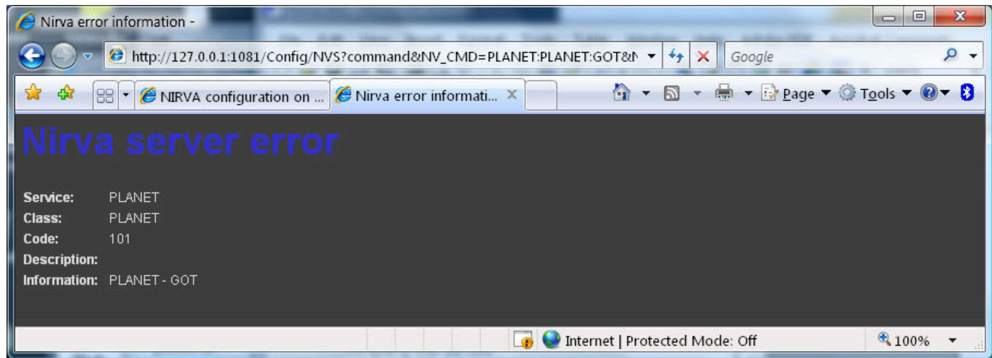
Let's modify again our PLANETSession::OnCommand function in the following way:

```
// Called for each Nirva command to the service
// This is the command entry point for session
// return true if successful and false otherwise
public bool OnCommand(nvcmd Command)
{
  // TODO
  // Insert command processing here

   if (Command.IsCommand("PLANET", "GET", ""))
   {
     // Create the PLANETS string list object
     String sCommand = "NV_CMD=|OBJECT:CREATE| NAME=|PLANETS| TYPE=|STRINGLIST|";
     Command.Command(sCommand);

     // Populate the object with planet names
     sCommand ="NV_CMD=|OBJECT:STRINGLIST_SET_VALUE| NAME=|PLANETS| SEPARATOR=|;|";
     sCommand +=" VALUE=|MERCURY;VENUS;EARTH;MARS;JUPITER;SATURN;URANUS;NEPTUNE;PLUTON|";
     Command.Command(sCommand);
     // Everything is OK
     return true;
```

```
    }

    // Unknown command
   // Prepare an error information and return it to NIRVA
   String ErrorInfo = Command.GetClass() + " - " + Command.GetCommand();

   // Set the error message
   Command.SetError("PLANET", 101, ErrorInfo);

   return false;
 }
```

After the code modification and compilation, please stop and restart the service form the nirva configuration tool (system/service menu).

Now we try to generate an error by sending a bad URL. We change the command name from “get” to “got”. Here is the result in the browser:

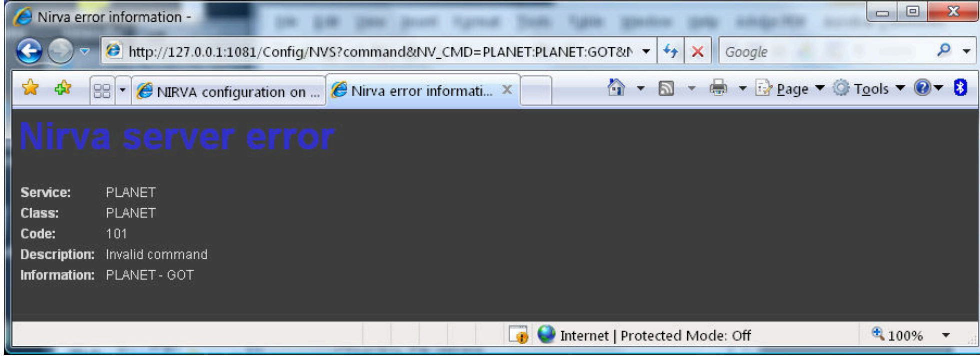


There is one thing more missing. We have the error class (PLANET), the error code (101), the error info (PLANET – GOT) but there is no error description. As we already saw, the error description is maintained in the service description file. Let’s modify the ERROR_CLASS_PLANET section of the service description file (in Nirva/Service/PLANET/Files):

```
[ERROR_CLASS_PLANET]
0   = No error;pas d'erreur;Keine Störung;Nessun errore;Ningún error
101 = Invalid command;commande non reconnue
```

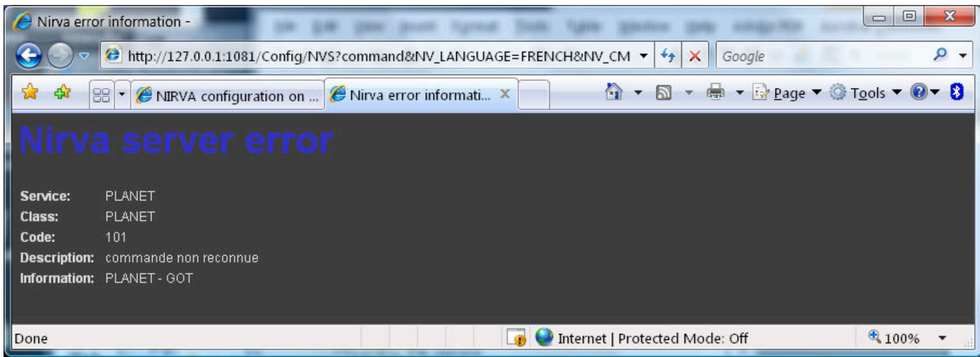
This section defines the error codes for the PLANET error class. Now we must stop and start again the NIRVA server nvs in order for the new content of the description file to be taken in care or we can just stop and restart the PLANET service from the nirva configuration tool (system/service menu).

Then, let’s try again our bad command from the web browser:



We have now the error description.

We can try it in French by adding the NV_LANGUAGE parameter (set to “FRENCH”) to the command:



Creating license

NIRVA provides all the mechanism for service providers to create and control their license.

A license is always linked to a unique machine ID. In order to get the local machine ID, we can use the getmachine.txt command file that resides in the Nirva/Bin directory. Here is the content of this file:

```
; NIRVA get machine identifier

NV_CMD=|license:info|
NV_CMD=|OBJECT:GET| NAME=|MACHINE_ID|
NV_CMD =|Local:OBJECT:STRING_GET_VALUE| NAME=|MACHINE_ID|
nvcc::printf \nMachine ID:
nvcc::printdata
nvcc::printf \n
```

```
Cmd
C:\nirva\Bin>nvcc -i getmachine.txt

Machine ID: 9ECCEB85CE58BDF057A6DA8C124CD393

C:\nirva\Bin>
```

This machine identifier must be kept somewhere.

When the service has been created, Nirva also created a file named “license.txt” in the PLANET/Source directory. This file contains a private and a public service ID. The private service ID is used to generate service license channels and the public service ID is used to check the license channels from inside the service code.

```
Here is an example of license.txt file:
PLANET Nirva service


This file contains the uniq service IDs you must use to create and check your license

Two service IDs are defined: the public service ID and the private service ID.

The public service ID must be used from inside your service in order to check
license channels that you have created. You check licenses by using the
SYSTEM LICENSE GET command.

The private service ID must be used only for creating new license channels by the way
of the Nirva nvl tool. The private service ID is confidential and must not be delivered
to anybody.


private service ID : 1E040EC4AB141DF7AD185B5206E90AA2
public service ID  : 21DE2F4F19C32C0A0A15C89D40D5B663
```

In our example, we will just create a license key for running the service. We’ll call it “RUN”.

Practically, a service provider distributes the license by giving a license file to the customer. The first step is for the customer to give its machine identifier to the service provider. This one then creates a license file and sends it back to the customer. Finally, the customer installs the license file to the NIRVA license manager.

We already saw the first step so we must now create the license file. For that, we use the following command file that we name “planet_license.txt”:

```
; NIRVA license creation for planet service
; 03/07/2002

NV_CMD=|license:set| /
  FILE=|license| /
  MACHINE_ID=|#param1| /
  SERVICE=|Planet| /
  SERVICE_ID=|1E040EC4AB141DF7AD185B5206E90AA2| /
  key=|RUN|
NV_CMD=|object:get| name=|license| Filename=|#param2|
```

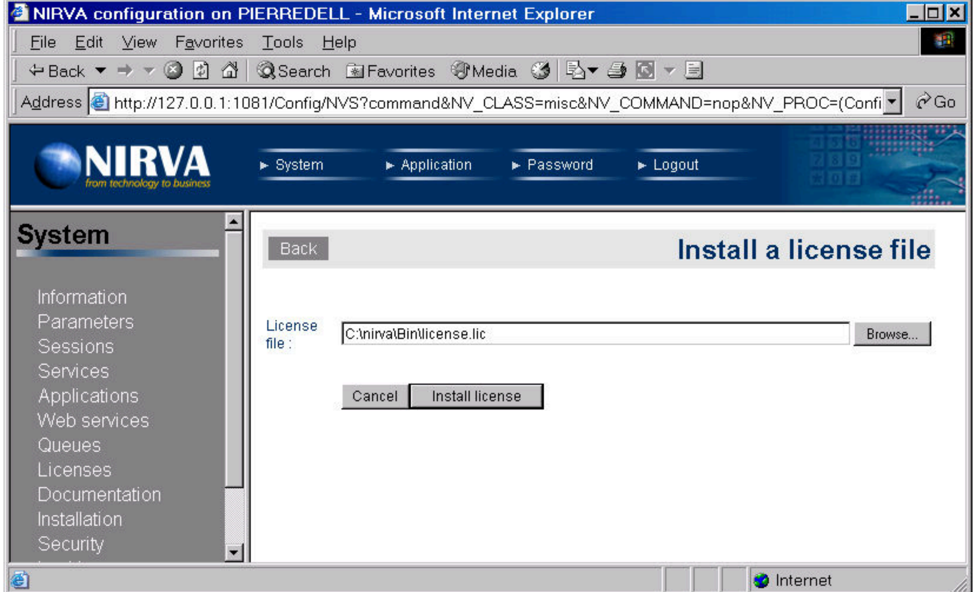
Please replace the SERVICE_ID parameter with your real service ID private key found in the “license.txt” file.

This command file first creates a license file on a server file object named “license” with the correct keys and then gets back this license file to the client in a file given as parameter. We use the following command to execute the license_create.txt file:

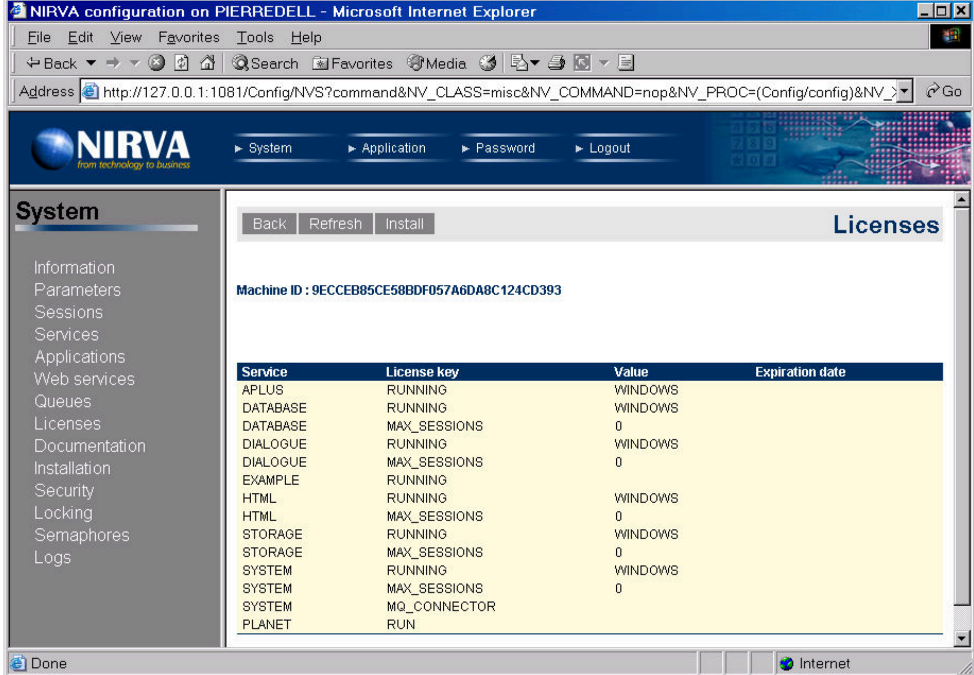
```
nvcc -i planet_license.txt 9ECCEB85CE58BDF057A6DA8C124CD393 license.lic
```

In this command you must replace the parameter "`9ECCEB85CE58BDF057A6DA8C124CD393`" with your real machine ID.

Now we have the license file in license.lic so we must import it to NIRVA. Let's do that from the configuration tool in the system/license menu by pressing the install button:



You must browse your just created license.lic file in order to install it to Nirva server. Then, the license channel has been added to Nirva:



Service	License key	Value	Expiration date
APLUS	RUNNING	WINDOWS	
DATABASE	RUNNING	WINDOWS	
DATABASE	MAX_SESSIONS	0	
DIALOGUE	RUNNING	WINDOWS	
DIALOGUE	MAX_SESSIONS	0	
EXAMPLE	RUNNING		
HTML	RUNNING	WINDOWS	
HTML	MAX_SESSIONS	0	
STORAGE	RUNNING	WINDOWS	
STORAGE	MAX_SESSIONS	0	
SYSTEM	RUNNING	WINDOWS	
SYSTEM	MAX_SESSIONS	0	
SYSTEM	MQ_CONNECTOR		
PLANET	RUN		

You can see the PLANET RUN license chanel added to the list.

License can also be created using the nvl.exe (windows tool) delivered in the Nirva/Bin directory. (Please see the chapter Tools/nvl in this documentation for further information).

Checking license

We know now how to generate license files and how to distribute them to the customers but we need to make some license control inside the service code itself.

A good place for doing that is when a new session is created on service side so we can add the license control code in the planetsession::OnInit method:

```
// Called one time when initializing the session
// return true if successful and false otherwise
public bool OnInit(nvcmd Command)
{
  // TODO
  // Insert eventual initialization code here

 // Checks the RUN license key
 String sCommand = "NV_CMD=|LICENSE:GET:PLANET|";
 sCommand += " KEY=|RUN| SERVICE_ID=|21DE2F4F19C32C0A0A15C89D40D5B663|";

  if(Command.Command(sCommand) == 0)
      return false;      // No license found

  // All is OK
  return true;
}
```

Please replace the SERVICE_ID parameter value with you real service ID public key.

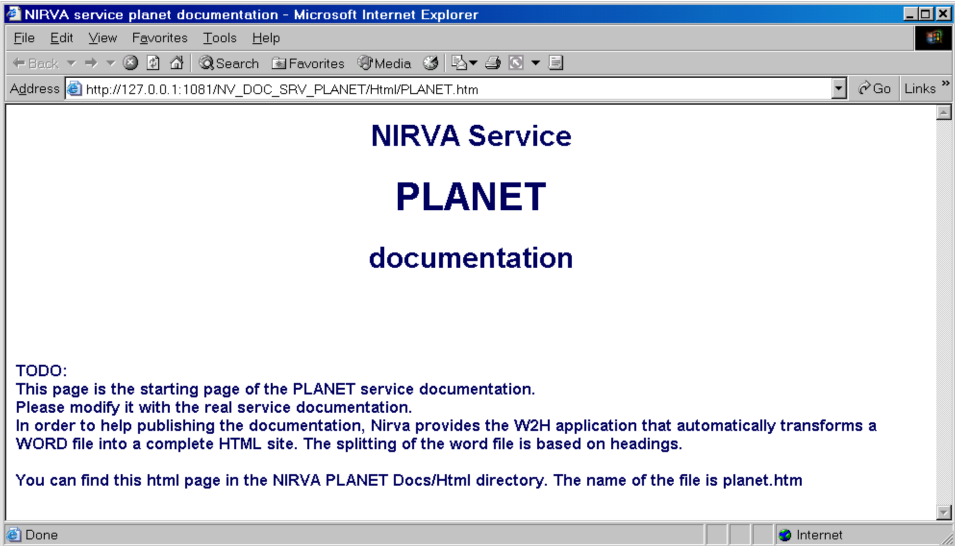
This is a simple way to check the license. Now it’s also possible to associate a value to a license key. We could use this feature in our example for controlling the maximum number of simultaneous sessions in the service planet.

Please consult the SYSTEM service LICENSE class for further information about licenses.

Documentation

The service documentation is directly available by an URL from a web browser. It’s the responsibility of the service provider to write this documentation.

The skeleton creates a file named “servicename.htm” where servicename is replaced by the name of the service (in lowercase) in the service Html documentation directory (in our example in Nirva/Service/PLANET/Docs/Html). This is the entry point of the service documentation. Here is the way to call it from the web browser:



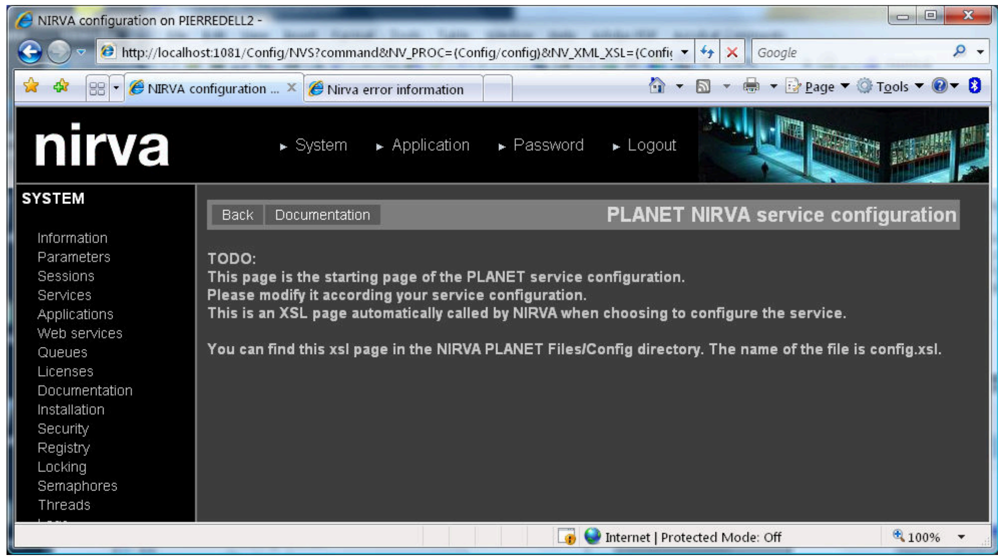
This URL points directly to Nirva/Service/PLANET/Docs/html/planet.htm

Configuration

In the same way that for the documentation, the service configuration is directly available by an URL from a web browser. It's the responsibility of the service provider to write the code of the configuration.

The configuration is made by using the XML and XSL capabilities of NIRVA.

The skeleton creates a file named “config.xsl” in the service config files directory (in our example in Nirva/Service/PLANET/Files/Config). This XSL file is used by NIRVA when calling the SYSTEM SERVICE CONFIG command from a web browser:



This commands first call a procedure named “config.nvp” in the service config procedure directory (here Nirva/Services/PLANET/Config/Procs directory) then prepares the content of the output container in XML data and finally calls the XML parser to transforms the XML data using the “config.xsl” file into viewable html code.

The service provider just has to modify the “config.xsl” and “config.nvp” files in order to set the service configuration.

Packaging the service

The SYSTEM SERVICE SKELETON command has also created a file named “package.lst” that resides in the service files directory (here Nirva/Services/PLANET/Files directory). This file is a package description file that will be used to create an installation package file with the help of the SYSTEM SERVICE PACKAGE command.

Here is the content of the package.lst file:

```
// package.lst : installation package listing
// for NIRVA SERVICE PLANET


// Header section
// This section is transmitted as it is to the package file
[HEADER]
SERVICE = PLANET

// All files and subdirectories of SERVICE Bin directory
[/Bin]
copydirsub:

// Description file
[/Files]
copyfile:service.dsc
textfile:service.dsc

// All files and subdirectories of SERVICE Files/Config directory
[/Files/Config]
copydirsub:

// Removes SERVICE Docs/Html directory
[/Docs/Html]
removedirsub:

// All files and subdirectories of SERVICE Docs directory
[/Docs]
copydirsub:

// All files and subdirectories of SERVICE Procs directory
[/Procs]
copydirsub:

// All files and subdirectories of SERVICE Wroot directory
[/Wroot]
copydirsub:
```

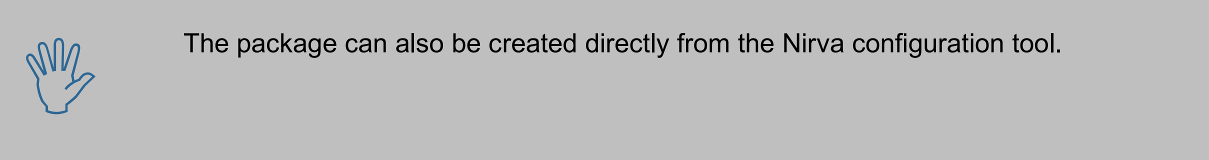
The format of the package file is described in the “installation packages” chapter.

We must have compiled the service in release mode and verify that the planet.dll file has been copied into the service Bin directory (here Nirva/Services/PLANET/Bin directory).

In order to create the package file, we can use the standard command file service_package.txt delivered in the NIRVA Bin directory. Here is the command:

```
C:\Nirva\Bin>nvcc -d ep -i service_package.txt PLANET planet.pkg

Service PLANET package creation
packaging service...

Package created in file  planet.pkg

C:\Nirva\Bin>_
```

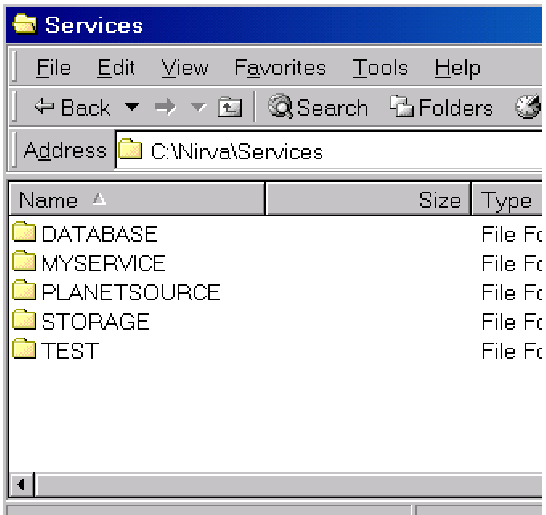
This command has created the local package file named “planet.pkg”. We can now use it to install the service.



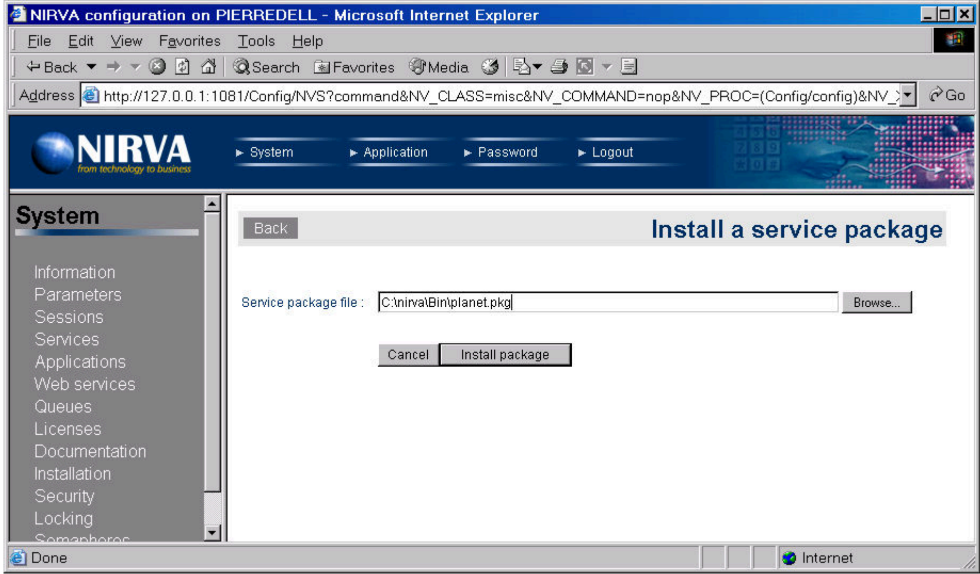
The package can also be created directly from the Nirva configuration tool.

Installing the service

For the service installation, we must consider that we are a customer receiving our service. For that, we will temporary rename the NIRVA PLANET service directory from PLANET to PLANETSOURCE. This is just for demonstrating the installation:



Now we use the Nirva configuration to do the service installation. Open the Nirva configuration tool, go to the systems/Services menu and press install:



After confirmation, the service should have been installed on the target platform.

Interface reference

This chapter gives the complete reference of the interface between NIRVA and an external service.

An external service is written in C++ or Java code.

C++

A C++ service consists of providing a dynamic library containing a set of well defined functions.

Nirva loads the library on memory and calls its functions on request.

Library entry point

The external service must implement the NvsCommand function. Here is the prototype of this function:

```
int NvsCommand(void* pCommand)
```

This function must return an integer value 1 in case of success and 0 otherwise.

It takes only one parameter which is a pointer to an NvServiceCommand class. This class is described in detail later in this chapter. The function should not delete this pointer.

The NvsCommand function is called for each command sent to the service including initialization and cleanup commands.

Library initialization and cleaning

When the external service is loaded in memory by NIRVA, it always receives a command SYSTEM NV_INIT_SERVICE as the first command. “SYSTEM” is the class name and “NV_INIT_SERVICE” is the command name. This command can be used to make the initialization of the service.

The service can catch this command by the following code:

```
if(Command->IsCommand("SYSTEM", "NV_INIT_SERVICE", NV_SOURCE_SERVER))
```

In the same way, when the external service is unloaded from memory by NIRVA, it always receives a command SYSTEM NV_EXIT_SERVICE as the last command. “SYSTEM” is the class name and “NV_EXIT_SERVICE” is the command name. This command can be used to make the cleanup of the service.

The service can catch this command by the following code:

```
if(Command->IsCommand("SYSTEM", "NV_EXIT_SERVICE", NV_SOURCE_SERVER))
```

The service skeleton that NIRVA creates redirects these two commands to the functions NvInitLib and NvExitLib.

Session management

NIRVA works always with sessions. Many services should implement their own session code. For example, a service that manages connections to databases should keep the database opened during all the session in order to keep the context.

For that, NIRVA provides a way for the service to work with sessions. A service session is always working with a single NIRVA session.

When a NIRVA session sends a command to the service for the first time, NIRVA first sends the command SYSTEM NV_INIT_SESSION to the service. “SYSTEM” is the class name and “NV_INIT_SESSION” is the command name. Typically, this command is used by the service to create its own session class.

The service can catch this command by the following code:

```
if(Command->IsCommand("SYSTEM", "NV_INIT_SESSION", NV_SOURCE_SERVER))
```

If the service creates its own session object, it can then send its pointer to the NIRVA session by using the `Command->SetServiceSession` function. For next commands received, the service will use the `Command->GetServiceSession` function to retrieve this pointer.

In the same way, before a NIRVA session ends, NIRVA sends the command SYSTEM NV_EXIT_SESSION to the service. “SYSTEM” is the class name and “NV_EXIT_SESSION” is the command name. Typically, this command is used by the service to delete its own session class.

The service can catch this command by the following code:

```
if(Command->IsCommand("SYSTEM", "NV_EXIT_SESSION", NV_SOURCE_SERVER))
```

When the service is unloaded from memory, NIRVA first sends a SYSTEM NV_EXIT_SESSION command for all the sessions connected to the service. This is done before sending the SYSTEM NV_EXIT_SERVICE command.

In order to see a typical implementation of service session management, please use the SYSTEM SERVICE SKELETON command to generate a service C++ minimal code or consult the service skeleton tutorial.

For a given service, Nirva locks the access to the NV_INIT_SERVICE, NV_EXIT_SERVICE, NV_INIT_SESSION and NV_EXIT_SESSION with the same lock object. In this way only one thread at a time can access one of these command implementations. So it's not necessary for the programmer to synchronize their code for these commands.

NvServiceCommand class

A pointer to NvServiceCommand class is given as parameter to the NvsCommand function of the external service.

This C++ class encapsulates the context of the command sent to the service. It provides functions for retrieving command information and for sending other commands to other services.

The NvServiceCommand class prototype is defined in the nvsext.h file. This file can be found in the Nirva/Sdk/Service directory or directly in the service source directory if the service code has been generated by the NIRVA SYSTEM SERVICE SKELETON command. Here is the declaration of the class:

```
class NvServiceCommand
{
public:
  virtual void SetError(const char *ErrorClass = "", int ErrorCode = 0, const char
*ErrorInfo = "") = 0 ;
  virtual bool Command(const char *CommandString, char *Buffer = NULL, int BufferSize =
0, int *DataSize = NULL) = 0 ;
  virtual void GetSessionId(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetClass(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetCommand(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetInContainer(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetOutContainer(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetLanguage(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetMachineName(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetMachineUser(char *Buffer, int BufferSize, int *DataSize = NULL) = 0 ;
  virtual void GetError(const char *InfoType, char *Buffer, int BufferSize, int
```

```
*DataSize = NULL) = 0 ;
  virtual int GetNumParameters() = 0 ;
  virtual int GetSource() = 0 ;
  virtual bool ParameterExist(const char *ParameterName) = 0 ;
  virtual void GetParameter(const char *ParameterName, char *Buffer, int BufferSize, int
*DataSize = NULL) = 0 ;
  virtual void GetParameterName(int index, char *Buffer, int BufferSize, int *DataSize =
NULL) = 0 ;
  virtual void AddHttpHeader(const char *HeaderName, const char *HeaderValue) = 0 ;
  virtual void SetServiceSession(void *ServiceSession) = 0 ;
  virtual void* GetServiceSession() = 0 ;
  virtual bool IsCommand(const char *Class = NULL, const char *Command = NULL, int
Source = -1) = 0 ;
};
```

The NvServiceCommand class contains only virtual functions that are implemented in the NIRVA kernel.

SetError

Syntax

virtual void SetError(const char *ErrorClass = "", int ErrorCode = 0, const char *ErrorInfo = "")

Description

This function sets the error code for the command. If a service command generates an error, the service NvsCommand function should return false and the service should use the NvServiceCommand::SetError function to set the error code.

Parameters

ErrorClass	Service error class. The service error classes are defined in the service description file. The error classes are entirely independent of the command classes.
ErrorCode	This is an integer that sets the error code of the service error class. This can be any positive value.
ErrorInfo	This parameter gives some more error information. NIRVA reports this string to the client. For example, if the error is “cannot open file”, the ErrorInfo parameter can report the name of the file. This will help support people or developers.

Return value

None.

Command

Syntax

virtual bool Command(const char *CommandString, char *Buffer = NULL, int BufferSize = 0, int *DataSize = NULL)

Description

This function sends a command to Nirva. This is the way for the service to communicate to NIRVA but also to the other external services.

This command receives the output buffer (whne there is one) into a buffer supplied by the service. If the supplied buffer is not large enough, one can reallocate a buffer to the required size (the size returned by the DataSize parameter) and then issue the command “COMMAND:GET_LAST_OUTPUT_BUFFER”:

Example:

```
int BufferSize = 100;
char *Buffer = (char*)malloc(BufferSize);
int DataSize = 0;
if(!Command->Command(CommandString, Buffer, BufferSize, &DataSize))
{
  free(Buffer);
  return false;
}
if(DataSize >= BufferSize)
{
  // Need to realloc
  char *NewBuffer = (char*)realloc(Buffer, DataSize+1);
  Buffer = NewBuffer;
  BufferSize = DataSize+1;
  Command->Command("NV_CMD=|COMMAND:GET_LAST_OUTPUT_BUFFER|",
                  Buffer, BufferSize, &DataSize);
}
free(Buffer);
return true;
```

Parameters

CommandString	Character buffer containing the command string. Please see the Nirva command syntax chapter for further information about the Nirva command.
Buffer	Output buffer. This is the address of a character buffer in which Nirva will write the command output if the command is using the output buffer. See the SYSTEM service reference to see which commands return information in the output buffer. This parameter can be NULL if no output buffer has to be used. Only the NIRVA SYSTEM service can return something in the output buffer.

BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data if the output buffer is to be used. This can be used to control that the output buffer is large enough. For example, if the output buffer size is 100 and the required data is 150, only the 100 first bytes (99 in fact) will be returned in the output buffer but the function will write 150 in the DataSize parameter. If DataSize is NULL, Nirva doesn't return the data size.

Return value

This function returns true if the command is successful and false otherwise. If not successful, the commands also sets the error information so the service doesn't need to set it itself.

GetSessionId

Syntax

virtual void GetSessionId(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the NIRVA session identifier of the session which has sent the command.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

GetClass

Syntax

virtual void GetClass(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the command class.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn’t return the data size.

Return value

None.

GetCommand

Syntax

virtual void GetCommand(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves command name.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn’t return the data size.

Return value

None.

GetInContainer

Syntax

virtual void GetInContainer(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the input container name.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

GetOutContainer

Syntax

virtual void GetOutContainer(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the output container name.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

GetLanguage

Syntax

virtual void GetLanguage(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the eventual language passed as parameter in the command.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

GetMachineName

Syntax

virtual void GetMachineName(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the name of the machine which has initiated the original command. If the command comes from a web browser, the machine name is empty.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

GetMachineUser

Syntax

virtual void GetMachineUser(char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the user name of the machine which has initiated the original command. If the command comes from a web browser, the user name is empty.

This user name has nothing to do with the NIRVA application user.

Parameters

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn’t return the data size.

Return value

None.

GetError

Syntax

virtual void GetError(const char *InfoType, char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the current command error information. It’s useful for the service to check an error after an NvServiceCommand::Command function call.

Parameters

InfoType	Kind of error information to return. This must be one of the strings “CODE”, “INFO”, “SERVICE”, “CLASS”, “DESCRIPTION”. For Nirva versions older than 2.5.024, “DESC” must be used instead “DESCRIPTION”. After version 2.5.024 both “DESC” and “DESCRIPTION” work. The default is “CODE” that returns the error code.
Buffer	Output buffer in which the function will write the result.

BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

GetNumParameters

Syntax

virtual int GetNumParameters()

Description

This function returns the number of parameters of the command. It can be used in conjunction with the NvServiceCommand::GetParameterName function to enumerate the parameters.

Parameters

None

Return value

The number of parameters.

GetParameterName

Syntax

virtual void GetParameterName(int index, char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves the name of a command parameter. It can be used in conjunction with the NvServiceCommand::GetNumParameters function to enumerate the parameters.

Parameters

index	Index of the parameter. This index starts at 1 and cannot be greater than the number of parameters.

Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

GetSource

Syntax

virtual int GetSource()

Description

This function returns the source of the command. The possible sources are defined in the nvsext.h file:

```
// Command source
#define NV_SOURCE_CLIENT0       // nvc source
#define NV_SOURCE_BROWSER       1       // web browser source
#define NV_SOURCE_PROCEDURE     2       // procedure source
#define NV_SOURCE_SERVICE       3       // service source
#define NV_SOURCE_SERVER4       // server source (used in init and close of services)
```

Parameters

None

Return value

The source of the command.

IsCommand

Syntax

virtual bool IsCommand(const char *Class = NULL, const char *Command = NULL, int Source = -1)

Description

This function checks the received command.

Parameters

Class	Class of the command. If the Class parameter is NULL, the function doesn't check the class name.
Command	Command name. If the Command parameter is NULL, the function doesn't check the command name.
Source	Source of the command. This parameter can take the value –1 or any of the values defined in the nvsext.h file. If the Source parameter is -1, the function doesn't check the command source.

Return value

True if the checking corresponds to the request and false otherwise.

ParameterExist

Syntax

virtual bool ParameterExist(const char *ParameterName)

Description

This function checks if the given parameter exists in the command parameters or not.

Parameters

ParameterName	Name of the parameter to check. The parameter name is case insensitive.

Return value

True if the parameter exists and false otherwise.

GetParameter

Syntax

virtual void GetParameter(const char *ParameterName, char *Buffer, int BufferSize, int *DataSize = NULL)

Description

This function retrieves a parameter value given a parameter name. If the parameter doesn't exist, the function returns a blank value.

Parameters

ParameterName	Name of the parameter to get. The parameter name is case insensitive.
Buffer	Output buffer in which the function will write the result.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data. This can be used to control that the output buffer is large enough. If DataSize is NULL, Nirva doesn't return the data size.

Return value

None.

AddHttpHeader

Syntax

virtual void AddHttpHeader(const char *HeaderName, const char *HeaderValue)

Description

This function adds a change an entry in the HTTP headers of the result of the original command.

This can be useful to transmit information in this way when using NIRVA XML features.

Parameters

HeaderName	HTTP header entry name.
HeaderValue	HTTP header entry value.

Return value

None.

SetServiceSession

Syntax

virtual void SetServiceSession(void *ServiceSession)

Description

This function can be used as answer to the SYSTEM NV_INIT_SESSION command sent to the service each time a NIRVA session calls the service for the first time. It allows for the service to create an object maintaining its own session context and to give a pointer to this object to NIRVA.

For each following command sent to the service, NIRVA will provide this pointer back to the service.

Parameters

ServiceSession	Pointer to the service session object to be maintained by NIRVA.

Return value

None.

GetServiceSession

Syntax

virtual void* GetServiceSession()

Description

This function retrieves the specific service session object previously sent by using the NvServiceCommand::SetServiceSession function. The caller must cast the result to the correct object session type.

Parameters

None

Return value

Pointer to the specific service session object.

Java

A Java service consists of providing a java class containing a set of well defined methods.

Nirva loads the class in memory and calls its methods on request.

This class is very simple to implement. The minimum requirement for the service class is to have an empty constructor, to provide 3 methods named Init, Exit and Command and to import the com.nirvasoft.nirva.nvcmd class. The nvcmd class allows Java service to communicate back to Nirva.

Command

The external service class must implement the Command method. Here is the prototype of this method:

```
public boolean Command()
```

This method must return a boolean value true in case of success and false otherwise.

It takes no parameter. In order to communicate back with NIRVA, the Command method can instantiate the nvcmd class. The nvcmd class is described later in this chapter.

Class initialization and cleaning

When NIRVA instantiates the external service class, it calls its Init method. This method can be used to make the initialization of the service. The Init method prototype is:

```
public boolean Init()
```

This method must return a boolean value true in case of success and false otherwise. If the return value is false, Nirva doesn't load the service.

It takes no parameter. In order to communicate back with NIRVA, the Command method can instantiate the nvcmd class. The nvcmd class is described later in this chapter.

In the same way, when the external service class is unloaded by NIRVA, NIRVA calls its Exit method. This method can be used to make the cleanup of the service. The Exit method prototype is:

```
public boolean Exit()
```

This method must return a boolean value true in case of success and false otherwise.

It takes no parameter. In order to communicate back with NIRVA, the Command method can instantiate the nvcmd class. The nvcmd class is described later in this chapter.

Session management

NIRVA works always with sessions. Many services should implement their own session code. For example, a service that manages connections to databases should keep the database opened during all the session in order to keep the context.

For that, NIRVA provides a way for the service to work with sessions. A service session is always working with a single NIRVA session.

When a NIRVA session sends a command to the service for the first time, NIRVA first sends the command SYSTEM NV_INIT_SESSION to the service. “SYSTEM” is the class name and “NV_INIT_SESSION” is the command name. Typically, this command is used by the service to create its own session class.

The service can catch this command by the following code:

```
nvcmd NvCommand = new nvcmd();

if(NvCommand.IsCommand("SYSTEM", "NV_INIT_SESSION", "SERVER"))
```

The service can use its own session management at this place.

In the same way, before a NIRVA session ends, NIRVA sends the command SYSTEM NV_EXIT_SESSION to the service. “SYSTEM” is the class name and “NV_EXIT_SESSION” is the command name. Typically, this command is used by the service to delete its own session class.

The service can catch this command by the following code:

```
nvcmd NvCommand = new nvcmd();

if(NvCommand.IsCommand("SYSTEM", "NV_EXIT_SESSION", "SERVER"))
```

When the service is unloaded from memory, NIRVA first sends a SYSTEM NV_EXIT_SESSION command for all the sessions connected to the service. This is done before sending the SYSTEM NV_EXIT_SERVICE command.

In order to see a typical implementation of service session management, please use the SYSTEM SERVICE SKELETON command to generate a service Java minimal code or consult the service skeleton tutorial. The skeleton creates a global hash table object that automatically maintains the session objects.

For a given service, Nirva locks the access to the NV_INIT_SERVICE, NV_EXIT_SERVICE, NV_INIT_SESSION and NV_EXIT_SESSION with the same lock object. In this way only one thread at a time can access one of these command implementations. So it’s not necessary for the programmer to synchronize their code for these commands.

nvcmd class

The nvcmd class is used by the service to communicate back to NIRVA. It's delivered in the nirva.jar file in the Nirva.Bin directory.

This Java class encapsulates the context of the command sent to the service. It provides functions for retrieving command information and for sending other commands to other services.

The nvcmd implements the following methods:

```
void SetError(String ErrorClass, int ErrorCode, String ErrorInfo)
boolean Command(String CommandString)
String GetResult()
String GetSessionId()
String GetSessionClass()
String GetCommand()
String GetInContainer()
String GetOutContainer()
String GetLanguage()
String GetMachineName()
String GetMachineUser()
String GetError(String InfoType)
int GetNumParameters()
String GetParameterName(int Index)
String GetSource()
boolean IsCommand(String Class, String Command, String Source)
boolean ParameterExist(String ParameterName)
String GetParameter(String ParameterName)
void AddHttpHeader(String HeaderName, String HeaderValue)
```

It also has a single property named “NvResult” that contains the result of some NIRVA commands. This is called the output buffer in this documentation. The NvResult property is always reset to an empty string before each command.

SetError

Syntax

void SetError(String ErrorClass, int ErrorCode, String ErrorInfo)

Description

This method sets the error code for the command. If a service command generates an error, the service NvsCommand function should return false and the service should use the NvServiceCommand::SetError function to set the error code.

Parameters

ErrorClass	Service error class. The service error classes are defined in the service description file. The error classes are entirely independent of the command classes.
ErrorCode	This is an integer that sets the error code of the service error class. This can be any positive value.
ErrorInfo	This parameter gives some more error information. NIRVA reports this string to the client. For example, if the error is “cannot open file”, the ErrorInfo parameter can report the name of the file. This will help support people or developers.

Return value

None.

Command

Syntax

boolean Command(String CommandString)

Description

This method sends a command to Nirva. This is the way for the service to communicate to NIRVA but also to the other external services.

Parameters

CommandString	Command string. Please see the Nirva command syntax chapter for further information about the Nirva command.

Return value

This function returns true if the command is successful and false otherwise. If not successful, the commands also sets the error information so the service doesn’t need to set it itself.

Some Nirva commands returns a string that is written into the nvcmd NvResult String member (This is called output buffer in command reference).

GetResult

Syntax

String GetResult()

Description

This method retrieves the value of the last successful Nirva command sent by the Command method.

Parameters

None.

Return value

NvResult string.

GetSessionId

Syntax

String GetSessionId()

Description

This method retrieves the NIRVA session identifier of the session which has sent the command.

Parameters

None.

Return value

Session ID.

GetClass

Syntax

String GetClass()

Description

This method retrieves the command class.

Parameters

None.

Return value

Command class.

GetCommand

Syntax

String GetCommand()

Description

This method retrieves command name.

Parameters

None

Return value

Command name.

GetInContainer

Syntax

String GetInContainer()

Description

This method retrieves the input container name.

Parameters

None.

Return value

Input container name.

GetOutContainer

Syntax

String GetOutContainer()

Description

This method retrieves the output container name.

Parameters

None

Return value

Output container name.

GetLanguage

Syntax

String GetLanguage()

Description

This method retrieves the eventual language passed as parameter in the command.

Parameters

None.

Return value

This method retrieves the eventual language passed as parameter in the command.

GetMachineName

Syntax

String GetMachineName()

Description

This method retrieves the name of the machine which has initiated the original command. If the command comes from a web browser, the machine name is empty.

Parameters

None.

Return value

Source machine name.

GetMachineUser

Syntax

String GetMachineUser()

Description

This method retrieves the user name of the machine which has initiated the original command. If the command comes from a web browser, the user name is empty.

This user name has nothing to do with the NIRVA application user.

Parameters

None.

Return value

Source machine user.

GetError

Syntax

String GetError(String InfoType)

Description

This method retrieves the current command error information. It’s useful for the service to check errors after a Command call.

Parameters

InfoType	Kind of error information to return. This must be one of the strings “CODE”, “INFO”, “SERVICE”, “CLASS”, “DESCRIPTION”. For Nirva versions older than 2.5.024, “DESC” must be used instead “DESCRIPTION”. After version 2.5.024 both “DESC” and “DESCRIPTION” work.

Return value

Error information.

GetNumParameters

Syntax

int GetNumParameters()

Description

This method returns the number of parameters of the command. It can be used in conjunction with the GetParameterName method to enumerate the parameters.

Parameters

None

Return value

The number of parameters.

GetParameterName

Syntax

String GetParameterName(int index)

Description

This method retrieves the name of a command parameter. It can be used in conjunction with the GetNumParameters method to enumerate the parameters.

Parameters

index	Index of the parameter. This index starts at 1 and cannot be greater than the number of parameters.

Return value

Parameter name.

GetSource

Syntax

String GetSource()

Description

This method returns the source of the command. The possible values are:

- CLIENT
- BROWSER
- PROCEDURE
- SERVICE
- SERVER

Parameters

None

Return value

The source of the command.

IsCommand

Syntax

boolean IsCommand(String Class, String Command, String Source)

Description

This method checks the received command.

Parameters

Class	Class of the command. If the Class parameter is empty, the function doesn't check the class name.
Command	Command name. If the Command parameter is empty, the function doesn't check the command name.
Source	Source of the command. If the Command parameter is empty, the function doesn't check the command name. This parameter can take the values “CLIENT”, “BROWSER”, “PROCEDURE” “SERVICE” or “SERVER”.

Return value

True if the checking corresponds to the request and false otherwise.

ParameterExist

Syntax

boolean ParameterExist(String ParameterName)

Description

This method checks if the given parameter exists in the command parameters or not.

Parameters

ParameterName	Name of the parameter to check. The parameter name is case insensitive.

Return value

true if the parameter exists and false otherwise.

GetParameter

Syntax

String GetParameter(String ParameterName)

Description

This method retrieves a parameter value given a parameter name. If the parameter doesn't exist, the function returns a blank value.

Parameters

ParameterName	Name of the parameter to get. The parameter name is case insensitive.

Return value

Parameter value or an empty string if the parameter doesn't exist.

AddHttpHeader

Syntax

void AddHttpHeader(String HeaderName, String HeaderValue)

Description

This method adds or changes an entry in the HTTP headers of the result of the original command.

Parameters

HeaderName	HTTP header entry name.
HeaderValue	HTTP header entry value.

Return value

None.

Dotnet

A Dotnet service consists of providing a dotnet class containing a set of well defined methods.

Nirva loads the class in memory and calls its methods on request.

Nirva creates a dotnet application domain for each service. This domain is loaded when the service starts and is released when the service stops.

The dotnet class is very simple to implement. The minimum requirement for the service class is to provide 3 methods named Init, Exit and Command. The class must override a class named “ServiceEntryPoint” defined in the assembly “nirvadn.dll” delivered in the Nirva/Bin directory. This assembly must ba added in the references of the source and the Nirva namespace must be used (“using Nirva” instruction)

The 3 methods receive a parameter of type nvcmd that is a reference to an object allowing the service to communicate back to Nirva. Nvcmd contains the command context and is specific to each command. The code should never keep a copy of this class across 2 commands.

Here is the minimal code for a dotnet service class:

```
using System;
using Nirva;

// class instancied when the service is started
public class myservice : ServiceEntryPoint
{
  public override bool Init(nvcmd NvCommand)
  {
          return true;
  }

  public override bool Exit(nvcmd NvCommand)
  {
          return true;
  }

  public override bool Command(nvcmd NvCommand)
  {
          return true;
  }
}
```

Class declaration

The class must override the ServiceEntryPoint class defined in the nirvadn assembly delivered in the Nirva/Bin directory.

```
public class myservice : ServiceEntryPoint
```

Command

The external service class must implement the Command method. Here is the prototype of this method:

```
public override bool Init(nvcmd NvCommand)
```

This method must return a boolean value true in case of success and false otherwise.

The nvcmd type parameter is used to communicate back to Nirva. The nvcmd class is described later in this chapter.

Class initialization and cleaning

When NIRVA instantiates the external service class, it calls its Init method. This method can be used to make the initialization of the service. The Init method prototype is:

```
public override bool Init(nvcmd NvCommand)
```

This method must return a boolean value true in case of success and false otherwise. If the return value is false, Nirva doesn't load the service.

The nvcmd type parameter is used to communicate back to Nirva. The nvcmd class is described later in this chapter.

In the same way, when the external service class is unloaded by NIRVA, NIRVA calls its Exit method. This method can be used to make the cleanup of the service. The Exit method prototype is:

```
override bool Exit(nvcmd NvCommand)
```

This method must return a boolean value true in case of success and false otherwise.

The nvcmd type parameter is used to communicate back to Nirva. The nvcmd class is described later in this chapter.

Session management

NIRVA works always with sessions. Many services should implement their own session code. For example, a service that manages connections to databases should keep the database opened during all the session in order to keep the context.

For that, NIRVA provides a way for the service to work with sessions. A service session is always working with a single NIRVA session.

When a NIRVA session sends a command to the service for the first time, NIRVA first sends the command SYSTEM NV_INIT_SESSION to the service. “SYSTEM” is the class name and “NV_INIT_SESSION” is the command name. Typically, this command is used by the service to create its own session class.

The service can catch this command by the following code:

```
if(NvCommand.IsCommand("SYSTEM", "NV_INIT_SESSION", "SERVER"))
```

The service can use its own session management at this place.

In the same way, before a NIRVA session ends, NIRVA sends the command SYSTEM NV_EXIT_SESSION to the service. “SYSTEM” is the class name and “NV_EXIT_SESSION” is the command name. Typically, this command is used by the service to delete its own session class.

The service can catch this command by the following code:

```
if(NvCommand.IsCommand("SYSTEM", "NV_EXIT_SESSION", "SERVER"))
```

When the service is unloaded from memory, NIRVA first sends a SYSTEM NV_EXIT_SESSION command for all the sessions connected to the service. This is done before sending the SYSTEM NV_EXIT_SERVICE command.

In order to see a typical implementation of service session management, please use the SYSTEM SERVICE SKELETON command to generate a service Dotnet minimal code or consult the service skeleton tutorial. The skeleton creates a global hash table object that automatically maintains the session objects.

For a given service, Nirva locks the access to the NV_INIT_SERVICE, NV_EXIT_SERVICE, NV_INIT_SESSION and NV_EXIT_SESSION with the same lock object. In this way only one thread at a time can access one of these command implementations. So it's not necessary for the programmer to synchronize their code for these commands.

nvcmd class

The nvcmd class is used by the service to communicate back to NIRVA. It's delivered in the nirvadn.dll assembly file in the Nirva.Bin directory.

This Dotnet class encapsulates the context of the command sent to the service. It provides functions for retrieving command information and for sending other commands to other services.

The nvcmd implements the following methods:

```
void SetError(String ErrorClass, int ErrorCode, String ErrorInfo)
bool Command(String CommandString)
String GetResult()
String GetSessionId()


String GetSessionClass()
String GetCommand()
String GetInContainer()
String GetOutContainer()
String GetLanguage()
String GetMachineName()
String GetMachineUser()
String GetError(String InfoType)
int GetNumParameters()
String GetParameterName(int Index)
String GetSource()
bool IsCommand(String Class, String Command, String Source)
bool ParameterExist(String ParameterName)
String GetParameter(String ParameterName)
void AddHttpHeader(String HeaderName, String HeaderValue)
```

It also has a single property named “NvResult” that contains the result of some NIRVA commands. This is called the output buffer in this documentation. The NvResult property is always reset to an empty string before each command.

SetError

Syntax

void SetError(String ErrorClass, int ErrorCode, String ErrorInfo)

Description

This method sets the error code for the command. If a service command generates an error, the service NvsCommand function should return false and the service should use the NvServiceCommand::SetError function to set the error code.

Parameters

ErrorClass	Service error class. The service error classes are defined in the service description file. The error classes are entirely independent of the command classes.
ErrorCode	This is an integer that sets the error code of the service error class. This can be any positive value.
ErrorInfo	This parameter gives some more error information. NIRVA reports this string to the client. For example, if the error is “cannot open file”, the ErrorInfo parameter can report the name of the file. This will help support people or developers.

Return value

None.

Command

Syntax

bool Command(String CommandString)

Description

This method sends a command to Nirva. This is the way for the service to communicate to NIRVA but also to the other external services.

Parameters

CommandString	Command string. Please see the Nirva command syntax chapter for further information about the Nirva command.

Return value

This function returns true if the command is successful and false otherwise. If not successful, the commands also sets the error information so the service doesn't need to set it itself.

Some Nirva commands returns a string (output buffer) that is kept by the nvcmd object and can be retrieved using the GetResult() method.

GetResult

Syntax

String GetResult()

Description

This method retrieves the value of the last successful Nirva command sent by the Command method.

Parameters

None.

Return value

Last command result string if the command was using the output buffer. Otherwise it's empty.

GetSessionId

Syntax

String GetSessionId()

Description

This method retrieves the NIRVA session identifier of the session which has sent the command.

Parameters

None.

Return value

Session ID.

GetClass

Syntax

String GetClass()

Description

This method retrieves the command class.

Parameters

None.

Return value

Command class.

GetCommand

Syntax

String GetCommand()

Description

This method retrieves command name.

Parameters

None

Return value

Command name.

GetInContainer

Syntax

String GetInContainer()

Description

This method retrieves the input container name.

Parameters

None.

Return value

Input container name.

GetOutContainer

Syntax

String GetOutContainer()

Description

This method retrieves the output container name.

Parameters

None

Return value

Output container name.

GetLanguage

Syntax

String GetLanguage()

Description

This method retrieves the eventual language passed as parameter in the command.

Parameters

None.

Return value

This method retrieves the eventual language passed as parameter in the command.

GetMachineName

Syntax

String GetMachineName()

Description

This method retrieves the name of the machine which has initiated the original command. If the command comes from a web browser, the machine name is empty.

Parameters

None.

Return value

Source machine name.

GetMachineUser

Syntax

String GetMachineUser()

Description

This method retrieves the user name of the machine which has initiated the original command. If the command comes from a web browser, the user name is empty.

This user name has nothing to do with the NIRVA application user.

Parameters

None.

Return value

Source machine user.

GetError

Syntax

String GetError(String InfoType)

Description

This method retrieves the current command error information. It’s useful for the service to check errors after a Command call.

Parameters

InfoType	Kind of error information to return. This must be one of the strings “CODE”, “INFO”, “SERVICE”, “CLASS”, “DESCRIPTION”.

Return value

Error information.

GetNumParameters

Syntax

int GetNumParameters()

Description

This method returns the number of parameters of the command. It can be used in conjunction with the GetParameterName method to enumerate the parameters.

Parameters

None

Return value

The number of parameters.

GetParameterName

Syntax

String GetParameterName(int index)

Description

This method retrieves the name of a command parameter. It can be used in conjunction with the GetNumParameters method to enumerate the parameters.

Parameters

index	Index of the parameter. This index starts at 1 and cannot be greater than the number of parameters.

Return value

Parameter name.

GetSource

Syntax

String GetSource()

Description

This method returns the source of the command. The possible values are:

- CLIENT
- BROWSER
- PROCEDURE
- SERVICE
- SERVER

Parameters

None

Return value

The source of the command.

IsCommand

Syntax

bool IsCommand(String Class, String Command, String Source)

Description

This method checks the received command.

Parameters

Class	Class of the command. If the Class parameter is empty, the function doesn't check the class name.
Command	Command name. If the Command parameter is empty, the function doesn't check the command name.
Source	Source of the command. If the Command parameter is empty, the function doesn't check the command name. This parameter can take the values "CLIENT", "BROWSER", "PROCEDURE" "SERVICE" or "SERVER".

Return value

true if the checking corresponds to the request and false otherwise.

ParameterExist

Syntax

bool ParameterExist(String ParameterName)

Description

This method checks if the given parameter exists in the command parameters or not.

Parameters

ParameterName	Name of the parameter to check. The parameter name is case insensitive.

Return value

true if the parameter exists and false otherwise.

GetParameter

Syntax

String GetParameter(String ParameterName)

Description

This method retrieves a parameter value given a parameter name. If the parameter doesn't exist, the function returns a blank value.

Parameters

ParameterName	Name of the parameter to get. The parameter name is case insensitive.

Return value

Parameter value or an empty string if the parameter doesn't exist.

AddHttpHeader

Syntax

void AddHttpHeader(String HeaderName, String HeaderValue)

Description

This method adds or changes an entry in the HTTP headers of the result of the original command.

Parameters

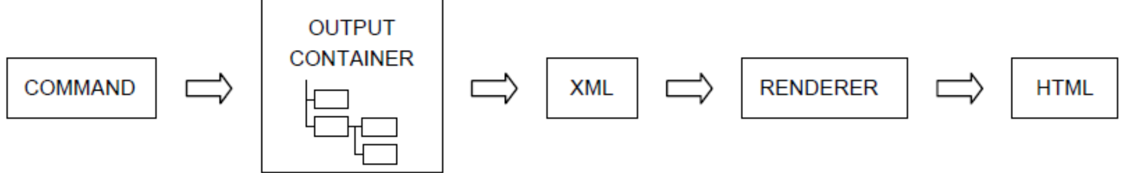
HeaderName	HTTP header entry name.
HeaderValue	HTTP header entry value.

Return value

None.

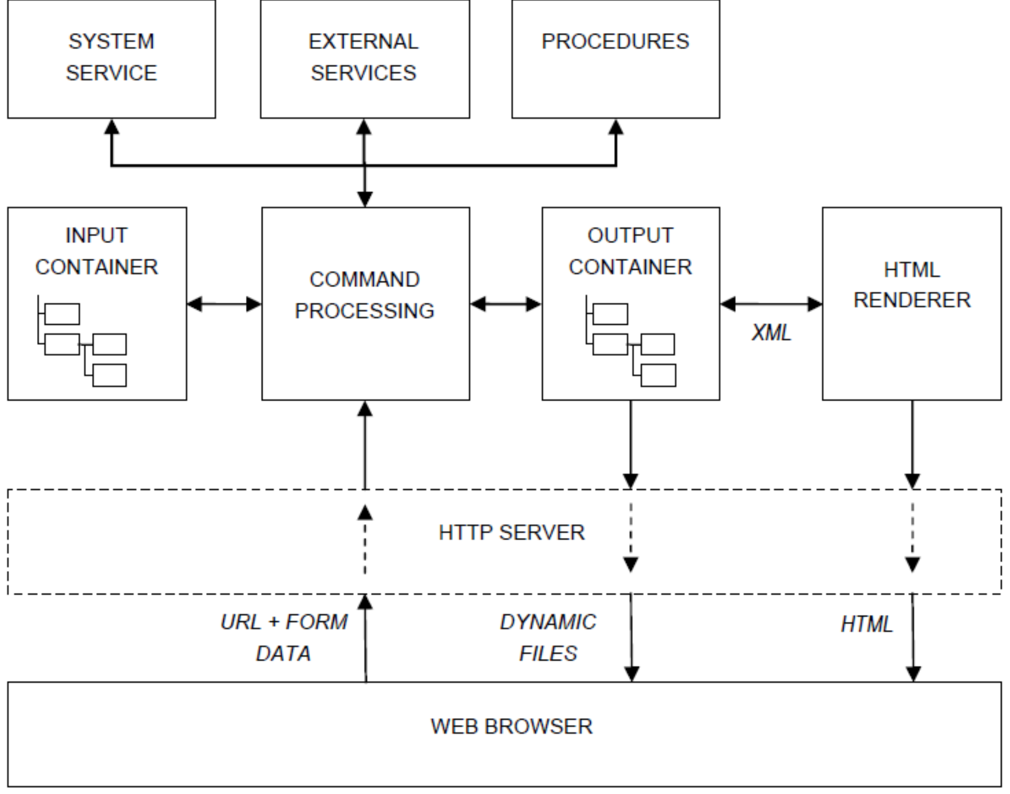
Renderers

The presentation layer works with HTML renderers that transform the content of an output container to HTML flow.



Nirva provides built-in renderers and delivers an interface for third parties to write their own renderers.

This is the typical request for a dynamic page.



The web browser sends a command to Nirva as a GET or POST HTTP request with optional form data. A typical Nirva URL command is the following:

```
http://myserver:1081/nv_app_myapp/NVS?command&NV_CMD=MYSERVICE:MYCLASS:MYCOMMAND&NV_PROC
=java:myproc&NV_XML_XSL=MYXSL&NV_SESSION_ID=1234567890
```

On receiving this request, Nirva first connects the session identified by the NV_SESSION_ID parameter. If this parameter is not given, Nirva opens a new session for the application “MYAPP” (other parameters authenticating the user must then be given).

The optional URL form data is transformed into session variables or file objects if there is a file upload and are reachable from the procedures and services.

As the NV_PROC parameter gives a name of a Java procedure (java:myproc), Nirva executes this procedure. The NV_PROC parameter contains the name of one or several procedures that are executed before the command itself. Another parameter allows execution of other procedures after the command. A procedure contains itself some Nirva commands and some calls to other procedures. Procedures are the place to build the business logic.

Then Nirva executes the command given in parameter NV_CMD. This is the command named “MYCOMMAND” for the class “MYCLASS” of the service “MYSERVICE”. If the NV_CMD parameter is not given, Nirva executes no command. This is generally the case when using Nirva as a web application server where the URLs just instruct Nirva to execute procedures.

A command gets information from an input container and delivers its data to an output container. The container names are given as parameters of the command but there are some default values. In this exemple, both input and output containers point to the same default container. The procedures inherit the name of the input and output containers.

When the command has finished, Nirva transforms the output container into XML and sends this XML stream to the renderer engine that generates the HTML code. The name of the renderer and the renderer page are respectively given by the NV_HREND and NV_HREND_PAGE parameters.

External renderers are mainly used with applications. They can also be used for service web pages (ie configuration pages) but this is not recommended since this creates a dependency between a service and a renderer. So it best to write service web pages using the built-in XSLT renderer.

Building a renderer

A render is a special Nirva service that implements a dedicated function named Render. This function receives the XML data and must deliver the HTML flow that will be directly sent back as HTTP content response to the browser.

Building a new renderer occurs in 3 steps:

- Create a service

- Implement the Render function
- Modify the service description file

Create a service

Just create a normal Java, C++ or Dotnet service as described in the services chapter.

Implement the render function

C++

In C++ the render function has the following prototype:

```
int NvsRender(void *RenderInterface, const char *ServiceName)
```

Where

- ServiceName is the name of the service
- RenderInterface is a pointer to a RenderInt class

The function must return 1 in case of success and 0 otherwise.

The RenderInterface is defined in the following way:

```
class NvRendererInt
{
public:
  virtual const char* GetInputData() = 0 ;
  virtual unsigned long GetInputDataSize() = 0 ;
  virtual const char* GetInputEncoding() = 0 ;
  virtual const char* GetPageName() = 0 ;
  virtual char* AllocOutputBuffer(unsigned long Size) = 0 ;
  virtual char* GetOutputBuffer() = 0 ;
  virtual unsigned long GetOutputBufferSize() = 0 ;
  virtual const char* GetSessionId() = 0 ;
  virtual const char* GetAppName() = 0 ;
  virtual void SetReason(const char* Reason) = 0 ;
  virtual void SetContentType(const char* ContentType) = 0 ;
  virtual void SetHttpRet(const char* HttpRet) = 0 ;
  virtual void* GetExtraData(const char* DataName) = 0 ;
  virtual void FreeInputData() = 0 ;
};
```

Where:

- GetInputData() retrieves a pointer to the XML data delivered by Nirva.

- GetInputDataSize() returns the size of the input data.
- GetInputEncoding() returns the encoding of the input. It can be “UTF-8” or “ISO-8859-1”.
- GetPageName() returns the name of the rendrer page as given by the NV_HREND_PAGE parameter.
- AllocOutputBuffer() must be used to allocate the buffer for storing the resulting Html code.
- GetOutputBuffer() returns a pointer to the output buffer (NULL if not allocated).
- GetSessionId() returns the current session ID.
- GetAppName() returns the application name.
- SetReason() sets the error reason in case the NvsRender function returns 0.
- SetContentType() sets the ouptut content type (default is “text/html”).
- SetHttpRet() sets the output http return code (default is “200”).
- GetExtraData() is not used.
- FreeInputData() can be used to free the input data if not more needed. Nirva automatically frees the input data if not done when the NvsRender function returns.

Example

```
int NvsRender(void *RenderInterface, const char *ServiceName)
{
  NvRendererInt *Render = (NvRendererInt*)RenderInterface;
  string MyResult = "<html><body><p>";
  MyResult += "Hello from renderer<br/>session ID is ";
  MyResult += Render->GetSessionId();
  MyResult += "<br/>page is ";
  MyResult += Render->GetPageName();
  MyResult += "</p></body></html>";
  char *OutputData = Render->AllocOutputBuffer(MyResult.length());
  memcpy(OutputData, MyResult.c_str(), Render->GetOutputBufferSize());
  return 1;
}
```

This is a basic example. A more realistic rendered should at least implement functions to get data from the input XML.

Java

In Java the render function has the following prototype:

```
public boolean Render(nvrend Rend)
```

It must be implemented in the service class.

The function must return true in case of success and false otherwise.

The nvrend object is defined in the following way:

```
public class nvrend
{
  public java.io.ByteArrayInputStream Input;
  public java.io.ByteArrayOutputStream Output;
  public java.lang.String PageName;
  public java.lang.String InputEncoding;
  public java.lang.String SessionId;
  public java.lang.String AppName;
  public java.lang.String Reason;
  public java.lang.String ContentType;
  public java.lang.String HttpRet;
  public int InputLength;
  public byte[] GetOutputBuffer()
```

Where:

- Input is an input stream containing the XML data delivered by Nirva.
- Output is the resulting output stream.
- InputEncoding is the encoding of the input. It can be “UTF-8” or “ISO-8859-1”.
- PageName is the name of the rendrer page as given by the NV_HREND_PAGE parameter.
- SessionId is the current session ID.
- AppName is the application name.
- Reason can be used by the renderer to set the error reason in case the Render function returns false.
- ContentType can be used by the renderer to change the output content type (default is “text/html”).
- HttpRet is the optional http return code (default is “200”).
- InputLength is the input length.
- GetOutputBuffer() Returns the output data as a byte array.

Example

There is an example on http://redmine.nirva-systems.com/projects/rnd-jsp/wiki

Dotnet

In Dotnet the render function has the following prototype:

```
public override bool Render(ref nvrend Rend)
```

It must be implemented in the service class.

The function must return true in case of success and false otherwise.

The nvrend object is defined in the following way:

```
[Serializable]
public class nvrend
{
    public byte[] Input;
    public byte[] Output;
    public String PageName;
    public String InputEncoding;
    public String SessionId;
    public String AppName;
    public String Reason;
    public String ContentType;
    public String HttpRet;
    public int InputLength;
}
```

Where:

- Input is a byte array containing the XML data delivered by Nirva.
- Output is a byte array containing the resulting output stream.
- InputEncoding is the encoding of the input. It can be “UTF-8” or “ISO-8859-1”.
- PageName is the name of the rendrer page as given by the NV_HREND_PAGE parameter.
- SessionId is the current session ID.
- AppName is the application name.
- Reason can be used by the renderer to set the error reason in case the Render function returns false.
- HttpRet is the optional http return code (default is “200”).
- ContentType can be used by the renderer to change the output content type (default is “text/html”).
- InputLength is the input length.

Example

```
public override bool Render(ref nvrend Rend)
{
    // Renderer
    String Output = "<head><meta http-equiv=\"Content-Type\"";
    Output += " content=\"text/html; charset=utf-8\"></head>";
    Output += "<html>";
    Output += "<h1>This is a dotnet renderer</h1>";
    Output += "</html>";

    UTF8Encoding enc = new UTF8Encoding();

    Rend.Output = enc.GetBytes(Output);
```

```
            return true;
        }
```

This is a basic example. A more realistic rendered should at least implement functions to get data from the input XML.

Modify the service description file

The file service.dsc in the service Files directory must be modified by adding a “RENDERER=YES” entry in the SETTINGS sections.

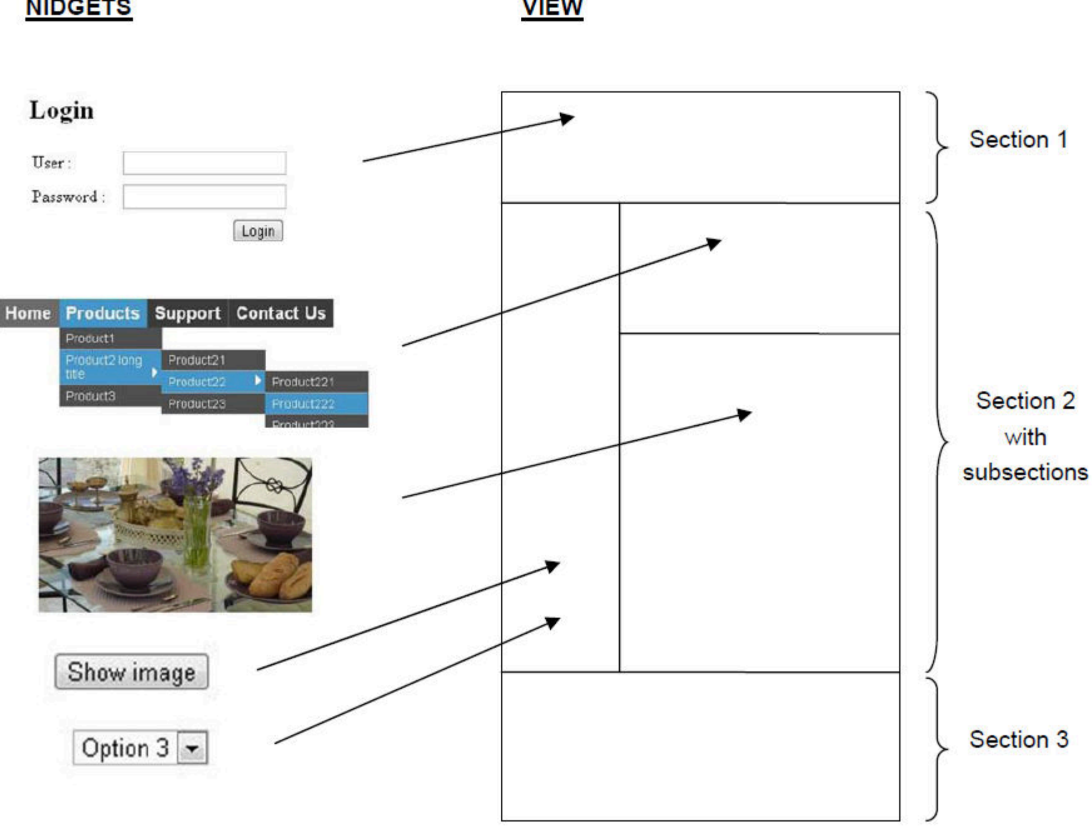
```
// SETTINGS section
[SETTINGS]
LANGUAGE = JAVA
PATH =
RENDERER = YES
```

Nidgets framework

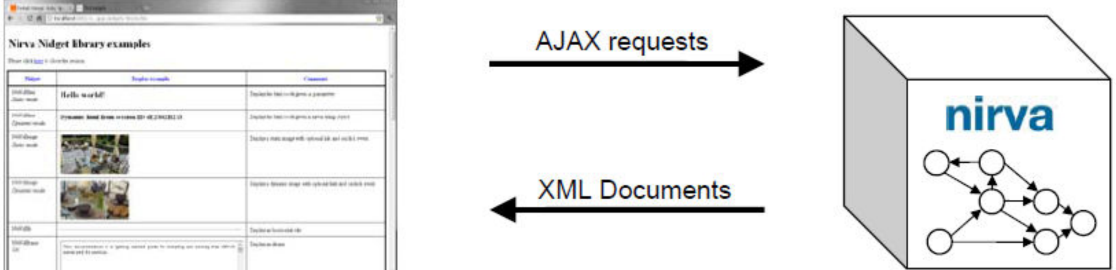
Overview

The Nirva nidget framework is based on JavaScript and Ajax technologies. It is a set of functionality allowing fast creation of Nirva web applications. “Nidgets” is the contraction of “Nirva” and “Widgets”. A Nidget is a standalone and customizable web component written in JavaScript. Nidgets are organized in libraries.

The framework defines the concept of view. A view is a web page divided in sections and subsections like nested tables in an html page. Each cell of the view can be populated with one or several nidgets.



The dynamic data exchange between Nirva and the browser is Ajax based. A view defines one or several XML documents generated by Nirva. Each nidget can be associated to one or several documents. If a document changes, an event is sent to the attached nidgets allowing them to change their display.



The html code of the views is automatically created by Nirva from the framework definition files that reside in the Files/Framework directory of the application. The html code is kept in a memory cache for being displayed in a very fast way. The code of the view is very short because it only contains the layout and names of the nidgets used in the view. All the real code is inside the nidgets themselves in Javascript files directly loaded and cached by the browser.

Once compiled and published, the framework code doesn't change so the display of web pages is fast. In debug mode Nirva checks if changes were made in the Files/Framework directory and automatically recompiles and publishes the appropriated views. If not in debug mode and if something has changed, all the views are compiled and published at application start time in order to accelerate the display.

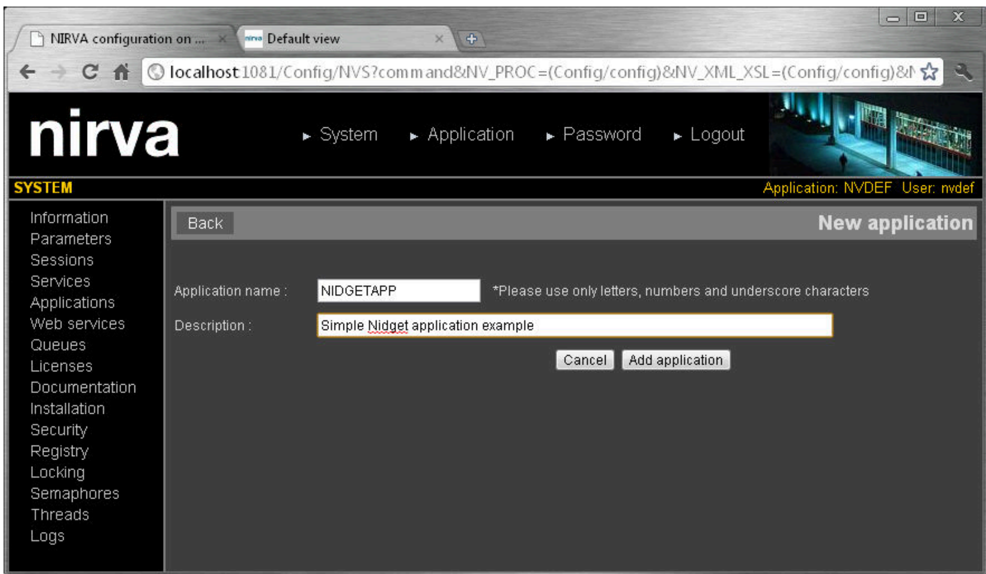
The framework files are located in the Files/Framework directory of the application and published in the Wroot/Framework directory. The developer always works in the Files/Framework directory and delivers the views of the application in this directory. He never writes or delivers files in the Wroot/Framework directory.

Getting started

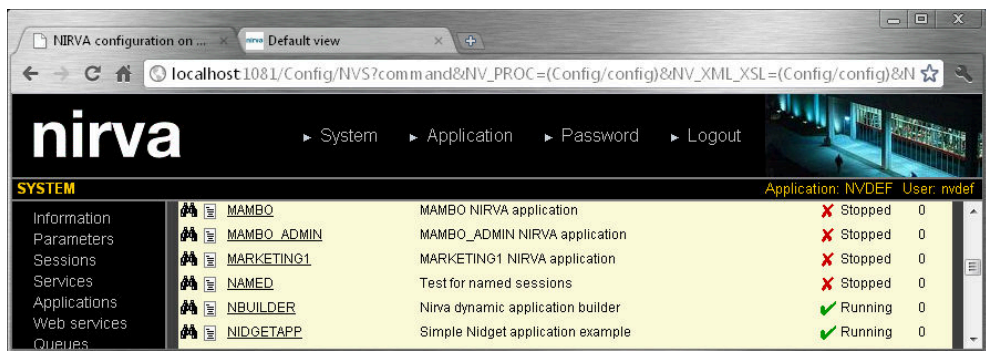
Hello world

In this section we'll create a very simple hello world application using the Nidget framework.

First create an application named NIDGETAPP from the Nirva configuration tool:



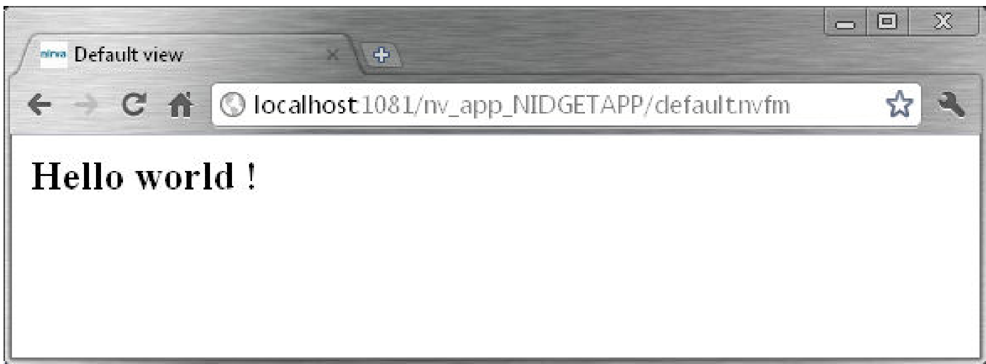
And start it:



Then edit the file NirvaDir/Applications/NIDGETAPP/Files/Framework/Params/default.xml and modify the line “<h2>Application default view</h2>” with “<h2>Hello world !</h2>”.

Verify that Nirva is in debug mode. If not turn it to debug mode in the Parameters sections of the configuration tool.

Then run the url http://localhost:1081/nv_app_NIDGETAPP/default.nvfm from your browser:



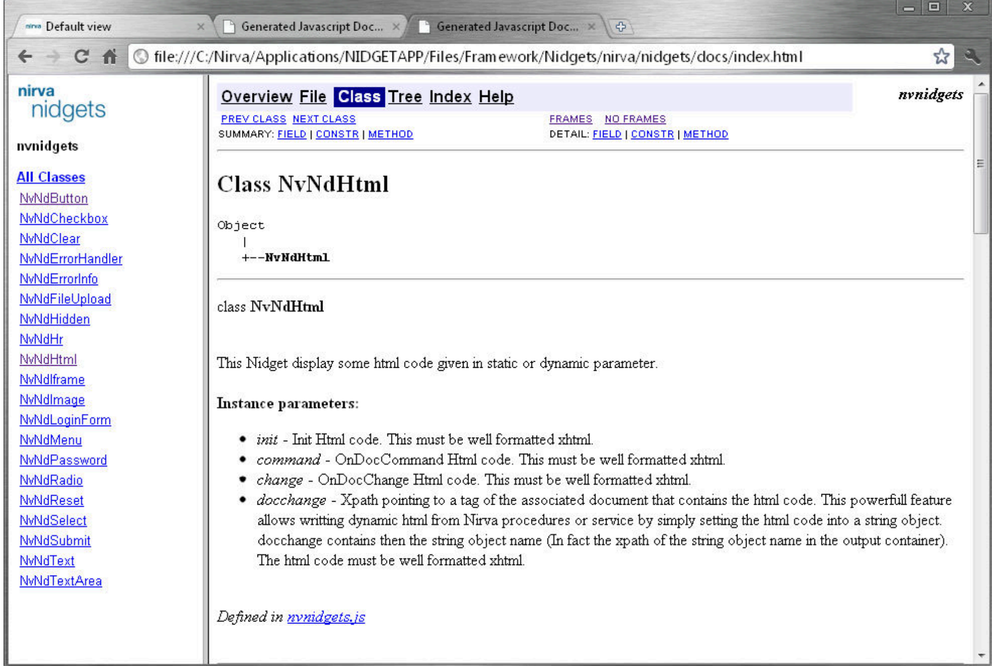
How it works

The previous url tells Nirva to display the view named “default” (the nvfm extension is associated to the nidget framework). When you create an application, Nirva automatically creates a default view. The definition

of the view can be found in the Files/Framework/Views/default.xml file (in the application directory). This is a static view (a view that doesn't require a nirva session) containing one Nidget of type “NvNdHtml” named “htmlcontent”. The layout of the view is defined in the “sections” tag:

```
<sections>
  <section name="global">
    <row>
      <col type="nidgets">
        <line>
          <nidget type="NvNdHtml">htmlcontent</nidget>
        </line>
      </col>
    </row>
  </section>
</sections>
```

The NvNdHtml Nidget is part of a standard Nirva Nidget library delivered in the file Files/Framework/Nidgets/nirva/nidgets/nvnidgets.js. This Nidget just displays the html code given as parameter (in fact this must be xhtml code). The Nidget library documentation can be found in: Files/Framework/Nidgets/nirva/nidgets/docs/index.html:



So in order to display our own xhtml code, you just have to write this code in the init parameter of the Nidget named “'htmlcontent”. This is done in the parameter file associated to the View named here default.xml. This file is declared in the “parameters” tag of the view definition file and resides in the Files/Framework/Params directory.

Declaration of the parameter file (in file Files/Framework/Views/default.xml):

```
<!-- nidgets parameters -->
<parameters>default.xml</parameters>
```

Content of the parameter file (Files/Framework/Params/default.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<parameters>
  <htmlcontent>
    <init>
      <h2>Hello world !</h2>
    </init>
  </htmlcontent>
</parameters>
```

The declaration of the views and the Nidget libraries to use are in the file config.xml in the Files/Framework directory. This file also contains some global application framework definitions.

```
<?xml version="1.0" encoding="UTF-8"?>

<framework>
  <!-- framework information -->
  <info>
         <name></name>
         <description></description>
         <version></version>
  </info>
  <!-- debug level -->
  <debug>0</debug>
  <!-- Error view -->
  <error>error</error>
  <!-- Header views -->
  <headers>
  </headers>
  <!-- Footer views -->
  <footers>
  </footers>
  <!-- javascript includes -->
  <includes>
  </includes>
  <!-- application icon -->
  <icon>Resources/nirva_icon.ico</icon>
  <!-- theme -->
  <theme></theme>
  <!-- Nidget packages -->
  <nidget_packages>
         <package>nirva/nidgets/nvnidgets.js</package>
  </nidget_packages>
  <!-- Views -->
  <views>
         <view>default</view>
         <view>error</view>
  </views>
</framework>
```

Dynamic code

In this chapter you'll create the same hello world view but this time in a dynamic way.

Just create another view by copying the file Files/Framework/Views/default.xml into a file named dynamic.xml in the same directory.

Then edit this file as follows:

```
Change the line: <type>STATIC</type>
With: <type>DYNAMIC</type>

Change the line: <title>Default view</title>
With: <title>Dynamic view</title>

Change the line: <parameters>default.xml</parameters>
With: <parameters>dynamic.xml</parameters>

Modify the docs tag as follows:

<docs>
  <doc>
    <name>dynhtml</name>
    <type>dynamic</type>
    <load>NV_PROC=|perl:gethtml| NV_CLOSE_SESSION=|YES|</load>
  </doc>
</docs>


Modify the nidget tag as follows:

<nidget type="NvNdHtml" docs="dynhtml">htmlcontent</nidget>
```

In the config.xml file, declare the new view:

```
<views>
  <view>default</view>
  <view>error</view>
  <view>dynamic</view>
</views>
```

Copy the file Files/Framework/Parameters/default.xml into a file named dynamic.xml in the same directory.

Then edit this file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<parameters>
  <htmlcontent>
    <docchange>myhtml</docchange>
```

```
  </htmlcontent>
</parameters>
```

Create a perl file named “gethtml.pl” in the application Procs directory and edit it as follows:

```
my $html = "<h2>Hello world";
NV::Command("NV_CMD=|VARIABLE:GET| NAME=|NV_SESSION_ID|");
$html = $html." from session $NV::RESULT</h2>";
NV::Command("NV_CMD=|OBJECT:CREATE| NAME=|myhtml| TYPE=|STRING| VALUE=|$html|");
```

Then you can display the view using the url http://localhost:1081/nv_app_NIDGETAPP/dynamic.nvfm from your browser (Nirva must be in debug mode for your changes to take effect):

In this example you’ve declared an Ajax xml document named “dynhtml” and you have associated this document to the “htmlcontent” nidget. When loading the view, Nirva is asked to execute the “gethtml” perl procedure. This procedure creates a string object named “myhtml” containing the html code to display. Then in the dynamic view parameter file the “htmlcontent” nidget uses the content of the “myhtml” object as html code.

For the purpose of the example, the session is closed just after loading the html code (see the docs/doc/load tag in the dynamic view definition). Of course, this is not mandatory and in a true application, you’d continue to work with an opened session.

More examples

Nirva/Bin/nirva_app_NIDGETS.pkg is a Nirva application example of the framework and the use of the standard nidget library. You can install it as a Nirva application, run Nirva in debug and console mode (to see some messages in the console) and then run the url http://localhost:1081/nv_app_NIDGETS/test.nvfm.

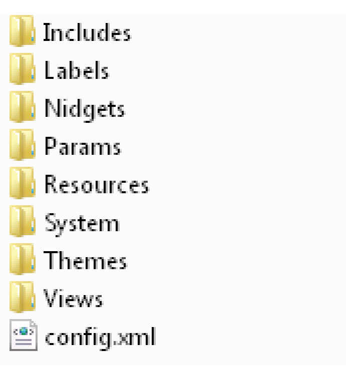
It’s advisable to study this application code to understand the basics of Nirva Nidgets.

Detailed concepts

Directory structure

Al the files of the Nidget framework reside in the Files/Framework subdirectory of the application directory. A publishing mechanism copies the necessary files from this directory to the application Wroot/Framework directory at run time. The programmer should never directly write or deliver files into the Wroot/Framework directory.

The framework source directory is organized as follows:



Item	Type	Description
config.xml	File	Main framework configuration file. Xml format.
Includes	Directory	Extra JavaScript include files. Those application specific JavaScript files will be published. The framework automatically publishes some standard JavaScript files that don't need to be included in the Includes directory. These standard JavaScript files are jquery.js (jquery library), nvajax.js (nirva Ajax connector), jquery.hoverIntent.minified.js (jquery hoverIntent extension) and nvfmw.js (nirva Nidget framework functions).
Labels	Directory	Label files. Label files are used to manage localisation of Nidget parameters.
Nidgets	Directory	Nidget files for the application. When creating an application, Nirva automatically copies the standard Nirva Nidget library files into this directory. Other Nidget libraries must be deployed here (generally in subdirectories).
Params	Directory	View parameter files (xml). The parameter files contain the parameters of the Nidget instances used in views.

Item	Type	Description
Resources	Directory	Application resources. The content of this directory will be published as it is in the Wroot/Framework/Resource directory. It contains application specific static files (images, static documents, etc…).
System	Directory	System files. This directory contains framework specific files and should not be modified.
Themes	Directory	Css files.
Views	Directory	View definition files (xml).

Configuration file

The file config.xml in the Framework irectory allows defining main framework parameters. This is where you declare views and Nidget libraries used in the application. The config.xml file should be utf-8 encoded.

An example of such a file:

```
<?xml version="1.0" encoding="UTF-8"?>

<framework>
  <!-- framework information -->
  <info>
    <name>Nidgets</name>
    <description>Nidget framework example</description>
    <version>1.00</version>
  </info>
  <!-- debug level -->
  <debug>0</debug>
  <!-- Error view -->
  <error>error</error>
  <!-- Header views -->
  <headers>
    <header>myheader</header>
  </headers>
  <!-- Footer views -->
  <footers>
    <footer>myfooter</footer>
  </footers>
  <!-- javascript includes -->
  <includes>
    <javascript>extra.js</javascript>
  </includes>
  <!-- application icon -->
  <icon>Resources/Images/nirva_icon.ico</icon>
  <!-- theme -->
  <theme>mytheme</theme>
  <!-- Nidget packages -->
  <nidget_packages>
    <package>nirva/nidgets/nvnidgets.js</package>
    <package>mynidgets/mynidgets.js</package>
  </nidget_packages>
```

```
  <!-- Views -->
  <views>
    <view>login</view>
    <view>login_test</view>
    <view>myheader</view>
    <view>myfooter</view>
    <view>myview</view>
    <view>myview_static</view>
    <view>test</view>
    <view>error</view>
    <view>frameset</view>
    <view>frame1</view>
    <view>frame2</view>
  </views>
</framework>
```

info tag

The info tag has 3 tags named “name”, “description” and “version”. They contain free text and are not mandatory. They are just here to help the programmer and external tools to identify the application.

debug tag

The debug tag gives the debug level of the framework. The debug level is transmitted to the nidgets. Nidgets can display debug information in a dedicated browser area when debug level is not 0.

error tag

The error tag gives the name of the view that will be used as an error view. The error view is called whenever an error occurs during access to a view.

Defining an error view is optional so this tag may be empty. When used, the view must be declared in the views section. An error view is a static type view. It can use framework functions to retrieve the error code.

headers and footers tags

The headers and footers tags respectively declare the views used as headers or footers of other views. Using headers and footers allows defining in one only place part of display which is common to several views (ex a banner, a menu, etc…). Each header must be declared inside a “header” tag and each footer must be declared inside a “footer” tag. When used, the header and footer views must also be declared in the views tag.

includes tag

The includes tag contains the name of extra JavaScript files the application uses. These files must physically reside in the Includes subdirectory. Each file must be declared in a “javascript” tag.

icon tag

This tag stores the name of application icon file. The application icon file is usually put in the Resource subdirectory.

theme tag

This tag gives the name of a css file (without the css extension) used for the application. The css file must be in the Theme subdirectory. There is only one theme per application. If several css files are necessary, use the css import statement. Css files can also be added dynamically using JavaScript code.

nidget_packages tag

This tag declares the nidget library packages used by the application. Nidget packages are JavaScript files. They physically reside in the Nidget subdirectory. Each package must be declared inside a “package” tag.

views tag

The views tag declares all the views of the application. The name of the view is the name of an xml file (without the .xml extension) residing in the Views subdirectory. Each view must be declared in a “view” tag.

Views

Each view declared in the views section of the configuration file must have a corresponding xml file in the Views subdirectory. For example if a view named “myview” has been declared in the views section, a file named “myview,xml” must exist in the Views subdirectory. The view file should be utf-8 encoded.

An example of such a file:

```
<?xml version="1.0" encoding="UTF-8"?>

<view>
  <!-- view information -->
  <info>
          <name>default</name>
          <description>default view</description>
          <version>1.00</version>
  </info>
  <!-- type STATIC or DYNAMIC (default)-->
  <type>STATIC</type>
  <!-- default language for static views -->
  <language>ENGLISH</language>
  <!-- view title -->
  <title>Default view</title>
  <!-- view global css class -->
  <class></class>
  <!-- header view -->
```

```
  <header></header>
  <!-- footer view -->
  <footer></footer>
  <!-- nidgets parameters -->
  <parameters>default.xml</parameters>
  <!-- nidget parameter labels -->
  <labels></labels>
  <!-- Ajax documents -->
  <docs>
  </docs>
  <!-- form parameters -->
  <forms>
  </forms>
  <!-- View sections (may contain subsections)-->
  <sections>
        <section name="global"> <!-- A section is converted to an html table -->
              <row>   <!-- Table row -->
                    <col type="nidgets">   <!-- Table column -->
                          <line>  <!-- The cell may contain several lines -->
                                <nidget type="NvNdHtml">htmlcontent</nidget>
                          </line>
                    </col>
              </row>
        </section>
  </sections>
</view>
```

info tag

The info tag has 3 tags named “name”, “description” and “version”. They contain free text and are not mandatory. They are just here to help the programmer and external tools to identify the view.

type tag

The type tag contains the view type. There are 3 possible values: “STATIC”, “DYNAMIC” or “HTML”:

- A static view can be called and displayed without a valid Nirva session. It contains Nirva Nidgets. When called in the context of a Nirva session, it can have some associated Ajax documents (see docs section). It’s usually used for login page or for error, footer and header views. The view named “myview_static” in the NIDGETS application is an example of a static view.
- A dynamic view can be called and displayed only in the context of a Nirva session so there is always a valid session ID in a dynamic view. A dynamic view contains Nirva Nidgets and can have some associated Ajax documents (see docs section). The view named “test” in the NIDGETS application is an example of a dynamic view.
- An html view is a special view containing only XHTML code. It cannot contain any Nirva Nidgets and cannot have associated Ajax documents. It’s an easy way to directly write html code in a view. This is a good place to put a frameset if needed. The view named “frameset” in the NIDGETS application is an example of an html view.

language tag

The language tag gives the default language for a static view. When there is a valid session ID, the language is the language of the Nirva session. When there is no session ID available (in a static view), the language is the one defined in this section. The default value is “ENGLISH”.

title tag

This is the view title displayed in the browser. Optional.

class tag

Optional global view css class. This parameter becomes the content of the “class” attribute of the view “body” tag when building the view. This parameter is not used with HTML type views.

header and footer tags

Name of a header and a footer to be used within the view. Using headers and footers allows defining in one only place part of display which is common to several views (ex a banner, a menu, etc…). Headers and footers are standard static or dynamic views. Headers and footers must be declared in the headers and footers section of the config file.

Here are some constraints with headers and footers:

- They must be a valid static or dynamic view declared as a view in the views section of the config file.
- They cannot be used with html type view.
- Header and footer cannot contain a header nor a footer.
- Header and footer must be declared respectively in the headers and footers section of the config file.
- A static view cannot use a dynamic header nor footer.
- Ajax documents must have unique names across header, footer and the view using them.
- Nidget instances must have unique names across header, footer and the view using them.
- Sections must have unique names across header, footer and the view using them.

The view named “myview” in the NIDGETS application is an example of a view using a header and a footer.

metas tag

Use this tag to add metas informations in the generated html code of the view. The “metas” tag must contain a list of “meta” tags. These are added directly in the output html header as “meta” tags.

parameters tag

A dynamic or static view contains Nidgets in the “sections” section. The Nidgets are instance of graphical components. Each Nidget can have several instance parameters. Those parameters values are defined in a dedicated parameter file that must be declared in the parameters tag. See the chapter about nidget parameters further in this documentation for more information about parameters.

The content of the parameters tag must be the name of a valid xml file (including extension) residing in the “Params” subdirectory.

There is an example of a parameter file in the NIDGETS application. The view name is “test” and the parameter file is “test_params.xml”.

labels tag

The Nidget instance parameters declared in the parameters file can be localized to match the language of the view. To use a localized parameter, one must use a label in the parameters file instead of a string value. The label must then be declared in the label file inside sections corresponding to the view language.

The labels tag contains the name of the optional label xml file (including extension) to be used with the view. This must be an existing file in the “Labels” subdirectory.

Labels are not available for html type views.

There is an example of a label file in the NIDGETS application. The view name is “myview_static”, the parameter file is “myview_static_params.xml” and the label file is “labels.xml”.

docs tag

This tag declares the Ajax xml documents used in the view (dynamic or static view). There are 3 kinds of documents:

- Dynamic - A dynamic document is produced dynamically by nirva (XML result of a nirva command).
- Static - A static document is an xml file residing in the Resources subdirectory.
- Inline - An inline document content is directly defined in the view file.

The docs tag is a collection of “doc” tags. Each doc tag contains itself the following tags:

- “name” is the document name. It must be unique across the view (and across header and footer when used). The document name is case insensitive and cannot contain space or special characters. The document name is mandatory.
- “type” is the document type. It must be “dynamic”, “static” or “inline”. It’s mandatory.
- “load” is used for dynamic and static documents. For a dynamic document, it containsthe nirva command to call when the view is loaded by the browser. When empty, no nirva command is sent. For a static document, it contains the path to the corresponding xml file. The file is relative to the Framework directory. Usually the xml static file resides in the Resources subdirectory so the load

parameter should be set to “Resources/myxml.xml” where myxml.xml is the static file to load. The load tag is not used for an inline document.

- “content” is reserved for inline documents. It contains the document xml code itself.

See example of documents in the “test” view of the NIDGETS application.

forms tag

This tag defines the form parameters of the view. A view (static or dynamic) can have a form associated to each top level section.

The “forms” tag is a collection of “form” tags. Each form tag has an attribute “name” storing the form’s name. The form’s name must be the same as the corresponding section name. Each form tag contains the following tags:

- “action” is the form action attribute. This is usually a request to an Ajax document.
- “validation” is the name of the JavaScript code for validating the form. Optional.

See example of forms in the “test” view of the NIDGETS application.

content tag

This tag is reserved for html type views. It contains the xhtml code that will be taken as it is in the generated html, just after the html head tag.

See example of content tag in the “frameset” view of the NIDGETS application.

sections tag

This tag defines the layout of the page by dividing it in sections and subsections. Each section or subsection contains a table defining cells. Each cell can contain one or several nidgets.

The “sections” tag contains a collection of “section” tags, each of them representing a top level section. Top level section may be associated to a form (see the “forms” tag).

Each section has a unique name defined in the “name” attribute of the “section” tag.

The structure is as follows:

```
<sections>
	<section name="name1" class="cs1" width="ws1" >
		<row class="cr1">
			<col class="cc1" width="wc1">
			</col>
		</row>
	</section>
```

```
        ...
</sections>
```

Such a section is transformed into the following html code when nirva builds the view:

```
<table id="name1" class="cs1" width="ws1" >
        <tr class="cr1">
                <td class="cc1" width="wc1" >
                </td>
        </tr>
</table>
```

The “class” and “width” attributes are optional. The section “name” attribute is mandatory and must be unique across the view (including header and footer if they are defined).

A cell of a section is located in the col tag. The col tag has another mandatory attribute named “type” that can take the value “nidgets” if the cell contains one or several nidgets or “subsection” if the cell contains a subsection.

When the cell contains nidgets, they must be enclosed in one or several line tags.

Example of a cell with horizontal nidgets:

```
<col type="nidgets">
        <line>
                <nidget type="NvNdHtml">html0</nidget>
                <nidget type="NvNdHtml">html1</nidget>
        </line>
</col>
```

Example of a cell with vertical nidgets:

```
<col type="nidgets">
        <line>
                <nidget type="NvNdHtml">html0</nidget>
        </line>
        <line>
                <nidget type="NvNdHtml">html1</nidget>
        </line>
</col>
```

Example of a cell with a subsection:

```
<col type="subsection">
  <section name="formerror" class="formerror" width="100%">
        <row>
                <col type="nidgets">
                        <line>
```

```
                                    <nidget type="NvNdHtml">errortext</nidget>
                            </line>
                    </col>
            </row>
    </section>
    <section name="form" class="form" width="100%">
            <row>
                    <col type="nidgets" width="20%">
                            <line>
                                    <nidget type="NvNdHtml">html22</nidget>
                            </line>
                    </col>
                    <col type="nidgets">
                            <line>
                                    <nidget type="NvNdText" docs="form">text1</nidget>
                            </line>
                    </col>
            </row>
    </section>
  </col>
```

The nidget tag itself has two attributes:

“type” is the type of nidget. The Nidget type name is given in the documentation of the nidget library that contains the nidget. The type attribute is mandatory.

“docs” is a semicolon separated list of Ajax documents associated to the nidget. These documents must be declared in the “docs” tag of the view. When a document is associated to a nidget, the nidget receives events when the document is first loaded and when it’s changing.

The nidget tag content is the instance name of the nidget. The instance name is transmitted to the nidget code and is used in the parameter file to identify the nidget parameters. The instance name must be unique across the view (and header and footer if used). The nidget tag can also have a “class” attribute.

At compilation time, the nidget tag is transformed to an html span tag with an “id” attribute set to the nidget instance name.

See example of layout in the “test” view of the NIDGETS application.

Nidgets

Nidget code

Nidgets are located in the Files/Framework/Nidgets directory. The Nidgets are JavaScript classes provided in JavaScript files. A single JavaScript file usually contains several nidgets.

Nirva provides a very basic Nidget library located in Files/Framework/Nidgets/nirva/nidgets. Any third party nidget library can be used. It just has to be added in the Files/Framework/Nidgets directory of the application and referenced in the framework configuration file.

Here is an example of a Nidget displaying a button:

```
NvNdButton = function (name) {
  this.ndname = name;
}

NvNdButton.prototype.ndname = null;

NvNdButton.prototype.OnInit = function (sectionname, formname, parameters) {
  var text = parameters.find('text').text();
  var onclick = parameters.find('onclick').text();
  var title = parameters.find('title').text();
  var MyHtml = '<button type="button" onclick="' + onclick + '" title="' + title + '">'
+ text + '</button>';
  $("#"+this.ndname).html(MyHtml);
}

NvNdButton.prototype.OnDocCommand = function (docname, xml) {
}

NvNdButton.prototype.OnDocChange = function (docname, xml) {
}
```

A single Nidget must provide a constructor and 3 functions named OnInit, OnDocCommand and OnDocChange.

Constructor

The constructor receives the name of the Nidget instance as parameter (String type).

Example:

```
NvNdButton = function (name) {
  this.ndname = name;
}
```

OnInit

The OnInit function is called by the framework when the page has been completely loaded. It has 3 parameters:

- *sectionname*: name of the section that owns the instance of the Nidget (direct parent section). Type String.
- *formname*: name of the parent form. Type String.
- *parameters*: static parameters of the Nidget instance. This is a jQuery object containing xml data that come from the view parameter file.

Example:

```
NvNdButton.prototype.OnInit = function (sectionname, formname, parameters) {
  var text = parameters.find('text').text();
  var onclick = parameters.find('onclick').text();
  var title = parameters.find('title').text();
  var MyHtml = '<button type="button" onclick="' + onclick + '" title="' + title + '">'
+ text + '</button>';
  $("#"+this.ndname).html(MyHtml);
}
```

OnDocCommand

The OnDocCommand function is called before a document associated to a Nidget changes. It can be used to change the display while Nirva processes the order and builds the XML document. It has 2 parameters:

- *docname*: name of the document that is about to change. Type String.
- *xml*: xml document. Type document (dom).

Example:

```
NvNdButton.prototype.OnDocCommand = function (docname, xml) {
}
```

OnDocChange

The OnDocChange function is called after a document associated to a Nidget has changed. This is where the display can be changed according to the content of the received Ajax document. It has 2 parameters:

- *docname*: name of the document that has changed. Type String.
- *xml*: xml document. Type document (dom).

Example:

```
NvNdText.prototype.OnDocChange = function (docname, xml)
{
  if(xml && this.docval)
  {
          var xpath = this.docval;
          xpath = xpath.replace(/\//g,'>');
          var value = $(xml).find(xpath);
          if(value.size())
                  $('#' + this.ndname + '_text').val(value.text());
  }
}
```

Third party providers building Nidget libraries should deliver their Nidgets as zip or tar file within a subdirectory specific to their company. For example mycompany/mynidgets. It is also a good practice to deliver the Nidget documentation. For that, Nirva embeds a perl script for Jsdoc. Here is an example for building a documentation of a Nidget library:

```
nvcc -z "NV_PROC=|(perl:System/JSDoc/jsdoc)| OUTPUT_DIR=|C:/mynidgets/docs|
FILENAME=|C:/mynidgets/mynidgets.js| LOGO=|C:/mynidgets/mynidgets.png|
PROJECT=|nvnidgets|"
```

Jsdoc documentation is available on http://jsdoc.sourceforge.net/. There is an example of nidget source file using the Jsdoc syntax in Nirvadir\Files\Framework\Nidgets\nirva\nidgets\nvnidgets.js

Parameters

Each Nidget instance may have associated parameters. The Nidget parameters are located in a parameter file in the “Params” subdirectory. The parameter file is referenced in the “parameters” tag of the view definition file. The parameter file is an xml file (utf-8 encoded) that contains a collection of nidget parameters in the “parameters” section.

Here is an example of such a file with parameters for 2 NvNdButton Nidgets named “button1” and “button2”:

```
<?xml version="1.0" encoding="UTF-8"?>

<parameters>
  <button1>
          <text>Hide frame</text>
          <onclick>NvFmHideNidget('iframe2')</onclick>
  </button1>
  <button2>
          <text>Show frame</text>
          <onclick>NvFmShowNidget('iframe2')</onclick>
  </button2>
</parameters>
```

The entire content of each Nidget parameter section is transmitted to the Nidget as a Jquery object in the “OnInit” function of the Nidget. The Nidget parameter section may contain any valid xml tree. This is up to the Nidget provider to detail the parameters of the Nidget in its documentation.

There is an example of a parameter file in the NIDGETS application. The view name is “test” and the parameter file is “test_params.xml”.

Label files

Label files are used to manage localisation of Nidget parameters. In a Nidget parameter file, instead of writing the parameter value, one can use labels using this syntax:

[NV_LABEL:LABELNAME]

Where LABELNAME is the name of the label. The label name is case insensitive. It should not contain any space or special character.

The label text for each managed language is stored in the label file.

Here is an example of a parameter file using labels:

```
<?xml version="1.0" encoding="UTF-8"?>

<parameters>
  <button1>
          <text>[NV_LABEL:MYBUTTONTEXT]</text>
          <onclick>NvFmHideNidget('iframe2')</onclick>
  </button1>
</parameters>
```

And the corresponding label file:

```
<?xml version="1.0" encoding="UTF-8"?>
<labels>
  <labels lang="ENGLISH">
          <mybuttontext>My button</mybuttontext>
  </labels>
  <labels lang="FRENCH">
          <mybuttontext>Mon bouton</mybuttontext>
  </labels>
</labels>
```

Each “labels” subsection of a label file corresponds to one language and contains all the labels defined for this language.

Standard nidgets library

Nirva provides a standard Nidget library that is automatically delivered in the Files/Framework/Nidgets/nirva/nidgets directory of a new application. The documentation for this library is available in the docs subdirectory (index.html file).

Framework functions

The Nidget framework provides a set of basic JavaScript functions defined in the file nvfmw.js. This file is automatically published so its functions are directly accessible from the Nidgets code.

This chapter is dedicated to Nidget providers.

We encourage the reader to study the file nvnidgets.js located in the Nirva/Files/Framework/Nidgets/nirva/nidgets directory. This file contains a library of standard nidgets. They are good examples of using framework functions.

NvFmAsyncDocCommand

NvFmAsyncDocCommand(docname, procedure, extraparam, formname)

Description

Sends an Ajax order to Nirva and retrieves a document. The function is asynchronous so it doesn't wait for the Nirva answer. The document must have been declared in the view. Any Nidget associated to the document receives an event before and after the order.

Parameters

docname	Document name as defined in the view. Required.
procedure	Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.
extraparam	Extra parameters. Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.
formname	Optional form name. If this parameter is given and the view contains a form with the given name, the function sends the form parameters to Nirva. Optional.

NvFmCommand

NvFmCommand(procedure, extraparam)

Description

Sends a command via Ajax to Nirva. This function is used to send a Nirva command without using Nidget documents. It doesn't return any xml document but the content of the Nirva output buffer that is available via the NvFmCommandGetResult function.

Parameters

procedure	Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.

extraparam	Extra Nirva parameters. This is a string containing Nirva command parameters in the usual syntax (name=|value|). Optional.

NvFmCommandGetResult

NvFmCommandGetResult()

Description

Returns the Nirva Output buffer as a string after a NvFmCommand call.

Parameters

none

NvFmDisplayNirvaError

NvFmDisplayNirvaError()

Description

Displays an alert message box with the last Ajax error message. This function is the default display error handler for Ajax communication.

Parameters

None

NvFmDocCommand

NvFmDocCommand(docname, procedure, extraparam, formname)

Description

Sends an Ajax order to Nirva and retrieves a document. The function waits for Nirva to answer. The document must have been declared in the view. Any Nidget associated to the document receives an event before and after the order.

Parameters

docname	Document name as defined in the view. Required.

procedure Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.

extraparam Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.

formname Optional form name. If this parameter is given and the view contains a form with the given name, the function sends the form parameters to Nirva. Optional.

NvFmFileUpload

NvFmFileUpload(inputid, procedure, extraparam)

Description

Uploads a file to Nirva. There is an example of using this function in the Nidget “NvNdFileUpload” in the nvnidgets.js file.

Parameters

inputid Id of an html input file type. Required.

procedure Procedure name to execute when uploading the file. This is the same content than the standard NV_PROC Nirva parameter. Optional.

extraparam Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.

NvFmGetApplication

NvFmGetApplication()

Description

Returns the current application name.

Parameters

none

NvFmGetDebug

NvFmGetDebug()

Description

Returns the current Nidget framework debug level.

Parameters

none

NvFmGetFileObject

NvFmGetFileObject(objname, procedure, extraparam, target, mime, mode, filename, cache)

Description

Sends an OBJECT:GET command to Nirva for retrieving and displaying a file object.

Parameters

objname	Name of the object to display. Required.
procedure	Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.
extraparam	Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.
target	This is the usual target attribute of HTML form (_blank, _self, _parent, _top, framename). Optional.
mime	Mime option. This parameter allows setting the mime type of the file object. When this parameter is not provided, NIRVA tries to set itself the mime type according to the file extension. Optional.
mode	If provided, can take values “ATTACHMENT” or “INLINE”. These values correspond to the “ATTACHMENT” and “INLINE” parameters of the OBJECT:GET command (see nirva user’s guide). Optional.
filename	The filename parameter has a meaning only if mode is not blank or null. This allows setting the name of the file when downloading from a web browser. Optional.
cache	Cache option. If this parameter is not “YES”, Nirva will add necessary code to the HTML output in order for the browser to not cache the object. The default is “NO”. Optional.

NvFmGetFileObjectUrl

NvFmGetFileObjectUrl(objname, procedure, extraparam, mime, mode, filename, cache)

Description

Constructs a url to get a file from Nirva using the OBJECT:GET command. Returns the url as a String.

Parameters

objname	Name of the object to get. Required.
procedure	Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.
extraparam	Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.
mime	Mime option. This parameter allows setting the mime type of the file object. When this parameter is not provided, NIRVA tries to set itself the mime type following the file extension. Optional.
mode	If provided, can take values “ATTACHMENT” or “INLINE”. These values correspond to the “ATTACHMENT” and “INLINE” parameters of the OBJECT:GET command (see nirva user’s guide). Optional.
filename	The filename parameter has a meaning only if mode is not blank or null. This allows setting the name of the file when downloading from a web browser. Optional.
cache	Cache option. If this parameter is not “YES”, Nirva will add necessary code to the HTML output in order for the browser not to cache the object. The default is “NO”. Optional.

NvFmGetLanguage

NvFmGetLanguage()

Description

Returns the current language of the view.

Parameters

none

NvFmGetLastError

NvFmGetLastError(type)

Description

Returns the required error information as a string.

Parameters

type	Requested error information type (String). This can take values “service”, “class”, “code”, “description” or “info”.

NvFmGetLoginViewUrl

NvFmGetLoginViewUrl(viewname, procedure, extraparam, formname)

Description

Constructs a url for getting a view (should be a static view) from Nirva without session ID. The function sets the session ID to an empty value. Returns the url as a String.

Parameters

viewname	Name of the view to display. Required.
procedure	Procedure name to execute. This is the same content as the standard NV_PROC Nirva parameter. Optional.
extraparam	Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.
formname	Optional for name. If this parameter is given and the view contains a form with the given name, the function adds the form parameters to the url. Optional.

NvFmGetNidgetObject

NvFmGetNidgetObject(NidgetName)

Description

Returns the Nidget instance object from its name.

Parameters

NidgetName	Nidget name (String).

NvFmGetSessionId

NvFmGetSessionId()

Description

Returns the current session Id as a string. If there is no session Id, returns an empty string.

Parameters

none

NvFmGetView

NvFmGetView(viewname, procedure, extraparam, formname, target)

Description

Displays a new view.

Parameters

viewname	Name of the view to display. Required.
procedure	Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.
extraparam	Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.
formname	Optional form name. If this parameter is given and the current view contains a form with the given name, the function sends the form parameters to Nirva. Optional.
target	This is the usual target attribute of HTML form (_blank, _self, _parent, _top, framename). Optional.

NvFmGetViewUrl

NvFmGetViewUrl(viewname, procedure, extraparam, formname)

Description

Constructs a url for getting a view from Nirva. The function automatically adds the session id parameter if there is one. Returns the url as a String.

Parameters

viewname	Name of the view to display. Required.
procedure	Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.
extraparam	Extra Nirva parameters. This is a string that contains Nirva command parameters in the usual syntax (name=|value|). Optional.
formname	Optional form name. If this parameter is given and the view contains a form with the given name, the function adds the form parameters to the url. Optional.

NvFmHideNidget

NvFmHideNidget(nidgetname)

Description

Hides a Nidget. All Nidgets are visible by default.

Parameters

nidgetname	Instance name of the nidget to hide. Required.

NvFmHideSection

NvFmHideSection(sectionname)

Description

Hides an entire section or subsection. All sections are visible by default.

Parameters

sectionname Name of the section to hide. Required.

NvFmHtmlEntities

NvFmHtmlEntities(htmltext)

Description

Replaces the &, <, >, ”, ’ characters in the given html text code to their corresponding HTML entities (& < > " '). Returns the htmltext string with replaced characters.

Parameters

htmltext Html code to process (type String). Required.

NvFmLogout

NvFmLogout(viewname, with_language, procedure, target)

Description

Closes the session and displays a new view (typically a static view).

Parameters

viewname Name of the view to display. Required.

with_language Boolean. When set to true, the current session language is transmitted as parameter to the view given in viewname. Optional.

procedure Procedure name to execute. This is the same content than the standard NV_PROC Nirva parameter. Optional.

target This is the usual target attribute of HTML form (_blank, _self, _parent, _top, framename). Optional.

NvFmSetErrorHandler

NvFmSetErrorHandler(ErrorHandler)

Description

Sets the default error handler for Ajax errors. The default error handler is to call the function NvFmDisplayNirvaError().

Parameters

ErrorHandler	Error code to call when an Ajax error occurs. This is a string containing code that will be passed to the JavaScript eval function. The error handler can use the NvFmGetLastError function to retrieve error information.

NvFmSetExitHandler

NvFmSetExitHandler(ExitHandler)

Description

Sets the default exit handler for the view. The exit handler is some JavaScript code called by the NvFmGetView function before calling the new view. The default exit handler does nothing.

Parameters

ExitHandler	Exit function for the view. This is a string containing code that will be passed to the JavaScript eval function.

NvFmSetInitHandler

NvFmSetInitHandler(InitHandler)

Description

Sets the default init handler for the view. The init handler is JavaScript code called at the end of the init process when all the documents and nidgets are built. The default init handler does nothing.

Parameters

InitHandler	Init code for the view. This is a string containing code that will be passed to the JavaScript eval function.

NvFmShowNidget

NvFmShowNidget(nidgetname)

Description

Shows a Nidget. All Nidgets are visible by default.

Parameters

nidgetname	Instance name of the nidget to show. Required.

NvFmShowSection

NvFmShowSection(sectionname)

Description

Shows an entire section or subsection. All sections are visible by default.

Parameters

sectionname	Name of the section to show. Required.

NvFmWriteDebug

NvFmWriteDebug(Line, Level)

Description

Writes debug information to the debug output when the debug level has been set in the config file.

Parameters

Line	Debug information to write (String).
Level	Debug level (Integer). If the current debug level defined in the config file is less than this parameter, then the debug line is not written in the debug console.

NvFmWriteDebugLn

NvFmWriteDebugLn(Line, Level)

Description

Writes a line debug information to the debug output when the debug level has been set in the config file. Adds a line feed at the end of the debug information.

Parameters

Line	Debug line to write (String).
Level	Debug level (integer). If the current debug level defined in the config file is less than this parameter, then the debug line is not written in the debug console.

innerXHTML

innerXHTML(element)

Description

Returns the XHTML code contained in a DOM element as a string. Examples available in the file nvnidgets.js.

Parameters

element	DOM element of an XML document. Required.

Web services

What is a web service

Web services represent a new architectural paradigm for applications. Web services implement capabilities that are available to other applications (or even other web services) via industry standard network and application interfaces and protocols. An application can use the capabilities of a web service by simply invoking it across a network without having to integrate it. As such, web services represent software building blocks that are URL addressable.

The capabilities provided by a web service can fall into a variety of categories, including:

- Functions, such as routine for calculating the integral square root of a number.
- Data, such as fetching the quantity of a particular widget a vendor has on band.
- Business processes, such as accepting an order for a widget, shipping the desired quantity of widgets and sending an invoice.

Some of these capabilities are difficult or impractical to integrate within third party applications. When these capabilities are exposed as web services, they can be loosely coupled together, thereby achieving the benefits of integration without incurring the difficulties thereof.

Web services expose their capabilities to client applications, not their implementation. This allows web services to be implemented in any language and on any platform and still be compatible with all client applications.

Each building block (web service) is self-contained. It describes its own capabilities, publishes its own programming interface and implements its own functionality that is available as a hosted service. The business logic of the web service runs on a remote machine that is accessible by other applications through a network. The client application simply invokes the functionality of a web service by sending it messages, receives return messages from the web service and then uses the result within the application. Since there is no need to integrate the web service within the client application into a single monolithic block, development and testing times, maintenance costs, and overall errors are thereby reduced.

Practically, a web service is defined as set of operations, each of them having a well defined structured pair of input and output messages. The description of the web service operations and messages is done into an XML data flow that respects the WSDL standard and is and URL accessible. In this way, a web service client application knows what the web service operations are and how to invoke them.

A client application invokes a web service operation by sending it an XML message formatted in respect to the SOAP standard. Then the web service executes the operation and returns back the result message also SOAP formatted.

The protocol used for exchanging messages between a web service and its clients is most often HTTP.

NIRVA implementation of web services

NIRVA allows users to create web services in just few clicks by the way of the NIRVA configuration tool.

A nirva web service is defined at system level but is always executed by a NIRVA application. In this way, a single web service can be used by several NIRVA applications.

A permission associated to each operation of a web service allows NIRVA applications to control web service security access.

Any NIRVA command puts incoming data into an input container and delivers resulting data from an output container. A NIRVA web service operation is very similar to a single NIRVA command except that the input and output message structures are well defined.

A NIRVA web service message is a well defined NIRVA container having subcontainers and NIRVA objects.

A NIRVA web service operation is a NIRVA procedure (written in native, perl, dotnet or java language) that takes data from the input container and delivers data into the output container.

A NIRVA web service itself is a collection of web service operations.

A NIRVA web service is accessible to external application like any other web service using HTTP, XML and SOAP standards but also from a NIRVA procedure or service by the way of a dedicated NIRVA command. In this way, it’s possible to integrate the web service business blocks into NIRVA applications or to create web services that call other web services.

Web service example

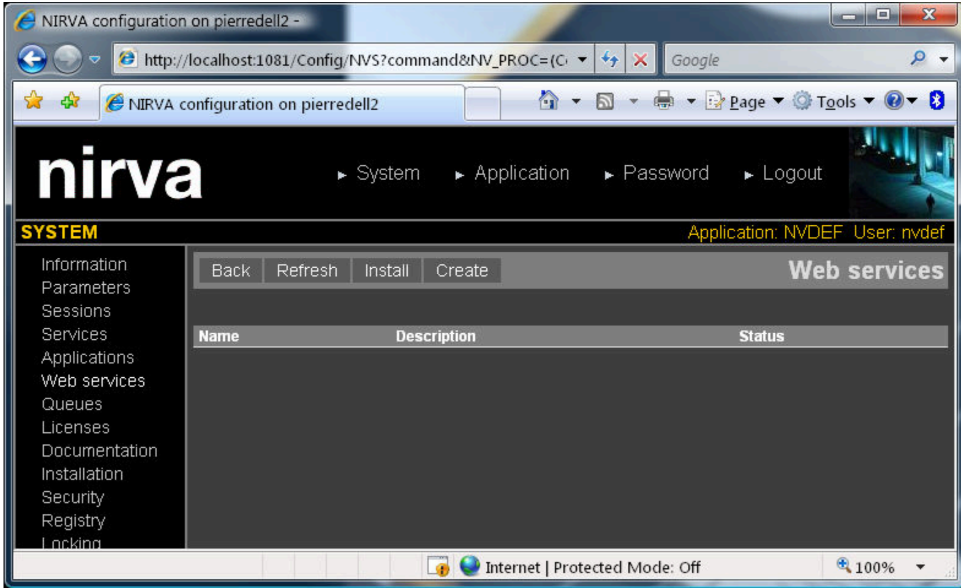
The following example creates a web service that takes in input the name of a person (first name and last name) and returns a welcome message.

Before trying this example, the NIRVA server must be running and the user must have enough rights to submit the necessary commands. The example runs on the NIRVA default application (NVDEF) and user (nvdef). Please consult the configuration chapter in order to give all the necessary permissions to the nvdef user in the NVDEF application. We can suggest to give the nvdef user all permissions.

The example shows nearly all steps in detail. In reality, these steps are carried out in a few seconds.

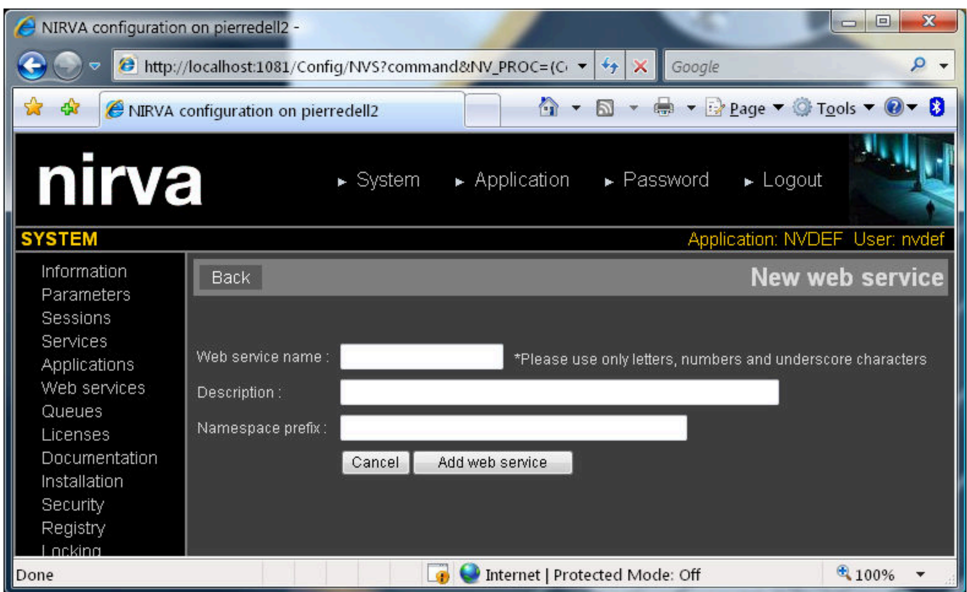
Creating the web service

For creating a web service, first run the configuration tool from your web browser (http://127.0.0.1:1081/Config/login.htm from the local machine) and use “nvadmin” user with default password “nirva” if you didn’t change it. Then go in the System/Web service menu. This should display the following screen:

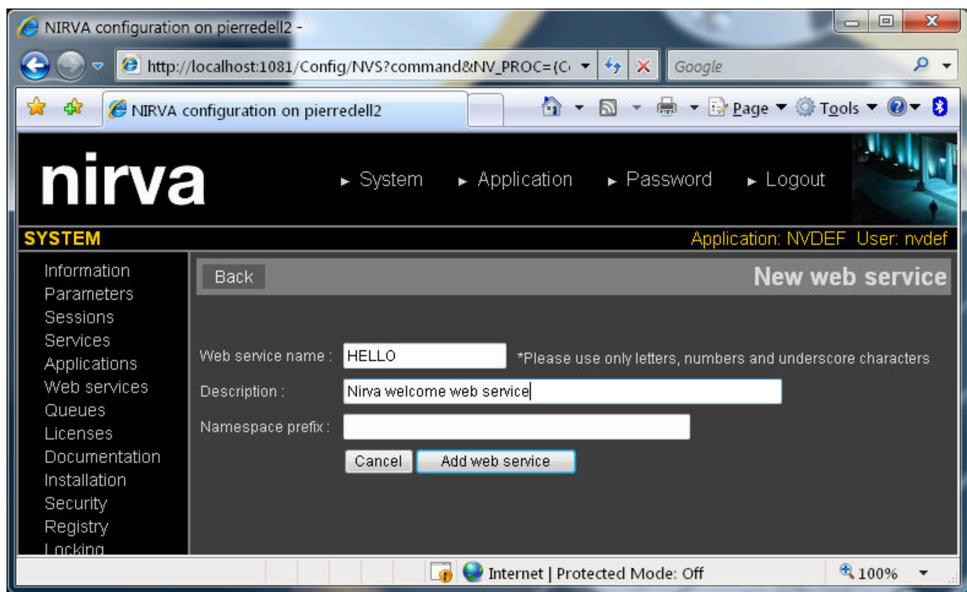


This is the list of available web services on your NIRVA server. The list should be empty if it is the first time you work with NIRVA web services.

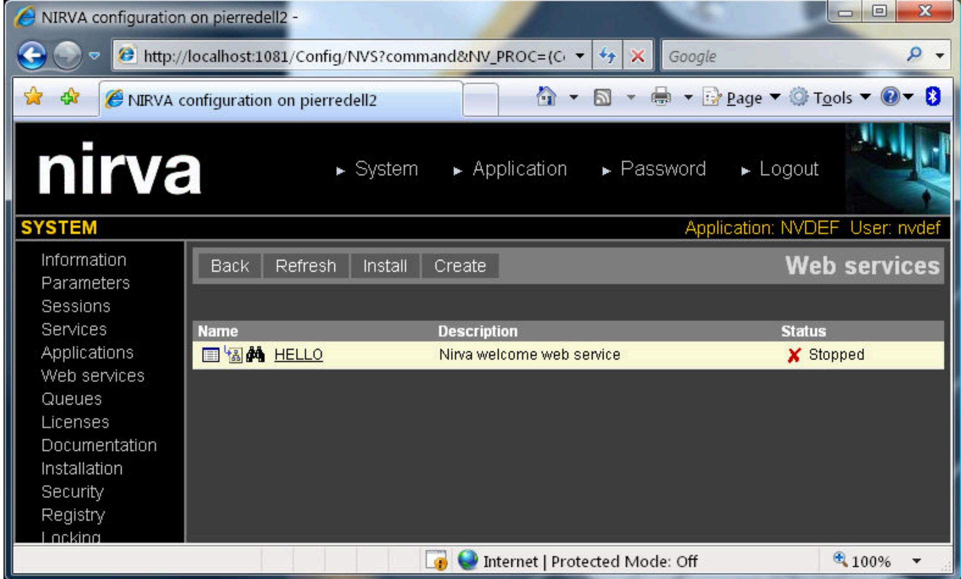
For creating a web service, press the “Create” button at the top of the screen:



In the Web service name, enter “HELLO” and in the description field, enter “Nirva welcome web service”: Leave the “Namespace prefix” field blank.



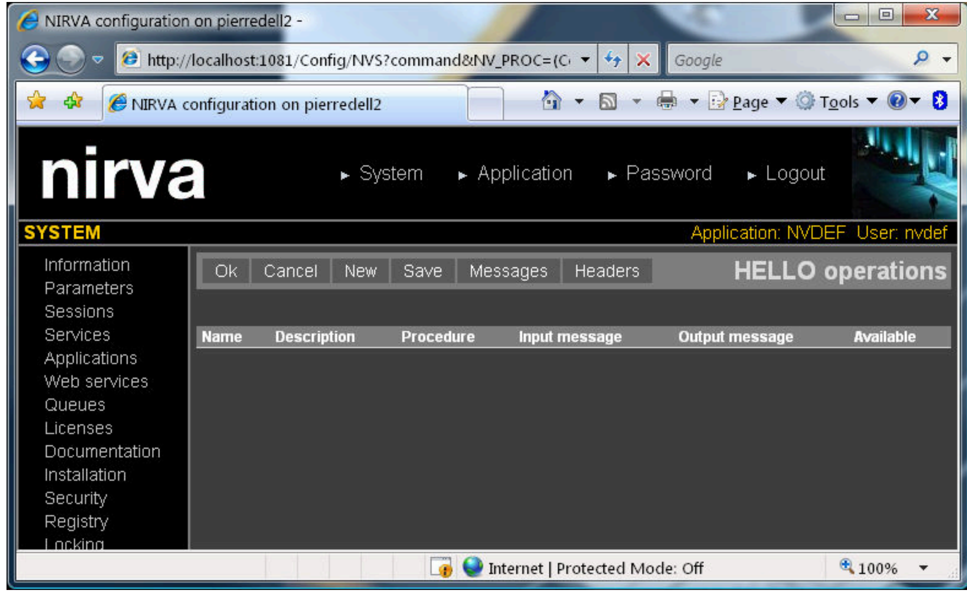
Press the “Add web service” button in order to create your “HELLO” web service. This returns to the web service list with the new HELLO web service listed:



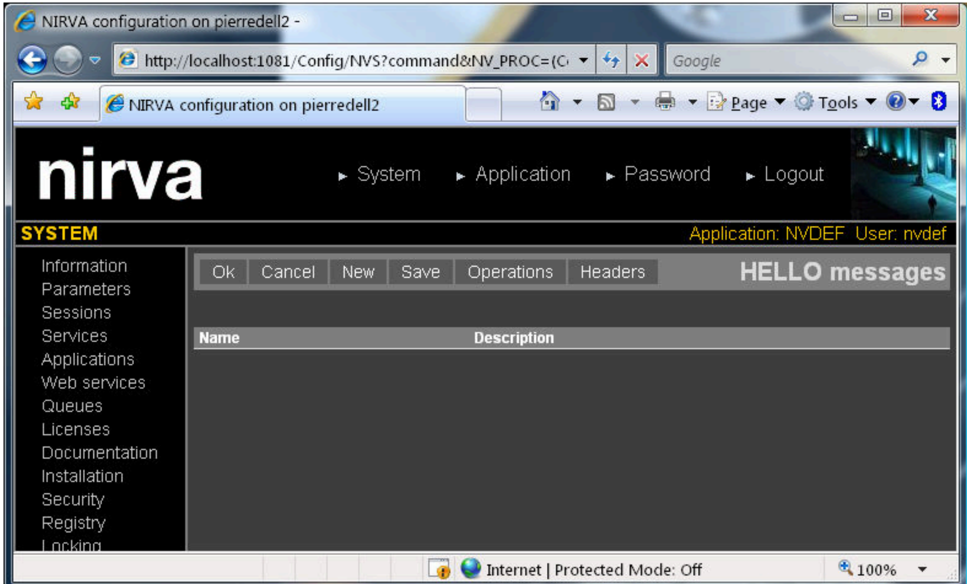
Editing the web service

After creating the web service we edit it in order to define its messages and operations. We will simply create an operation named “Welcome” accepting an input message named “You” that contains two string objects “firstname” and “lastname” and delivering an output message named “Message” containing a string object named “Welcome”.

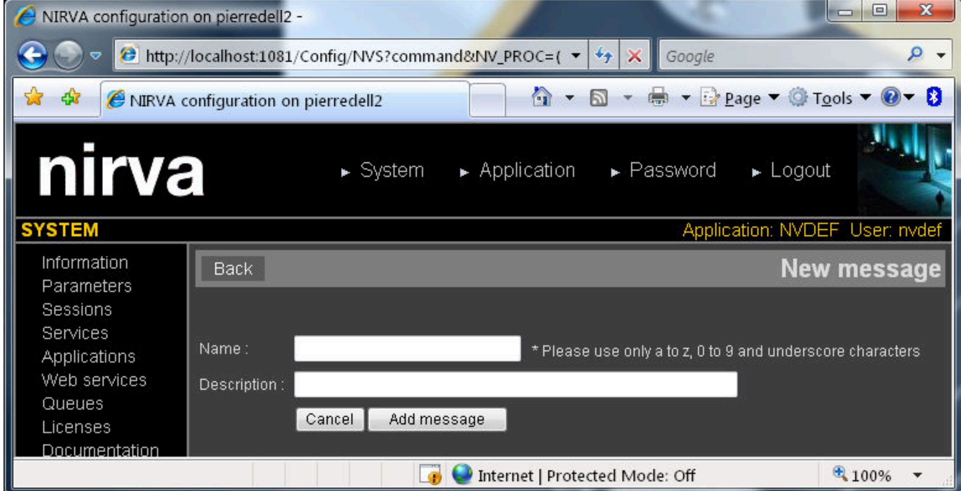
Enter the editing mode (the web service should be stopped for being able to enter editing mode). For that, click on the icon near the service name. This displays the following screen:



This is the list of operations. We'll create the "Welcome" operation later. Define the messages first by clicking on the "Messages" button:

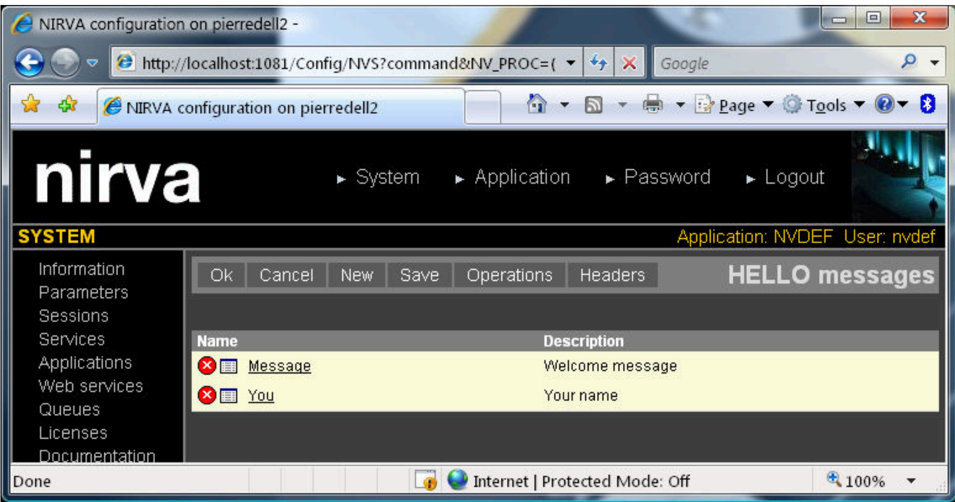


This is the list of defined messages for the HELLO web service. We now create a new message by pressing the "New" button:

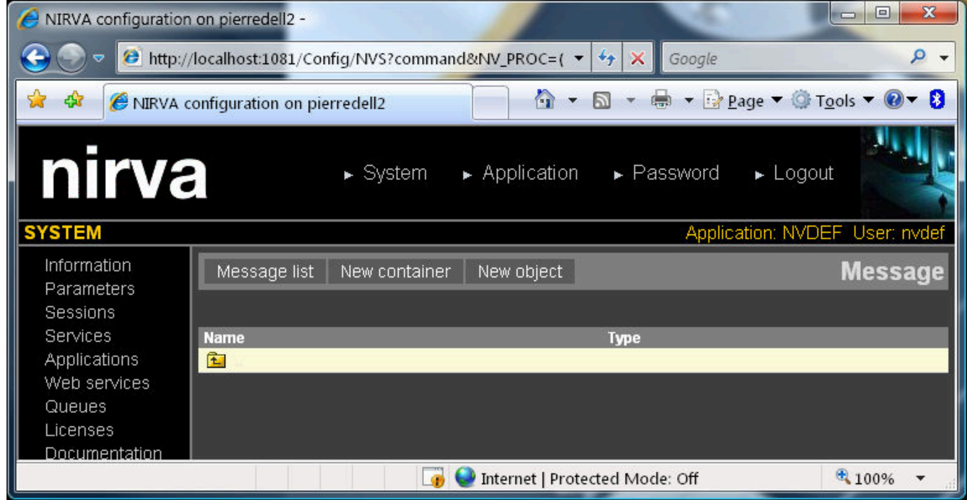


For the first message, enter “You” as the message name and “Your name” as description and press the “Add message” button. Repeat the operation for the second message with “Message” as name and “Welcome message” as description.

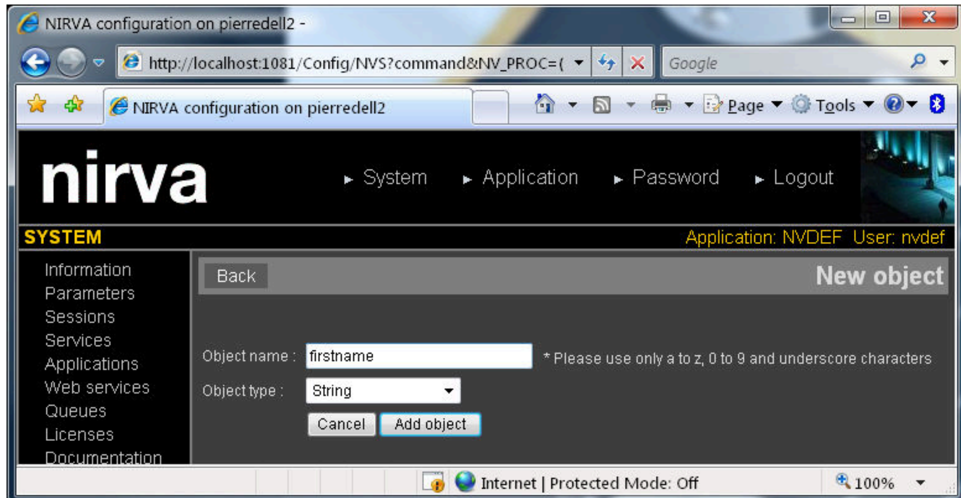
The message list should look like this:



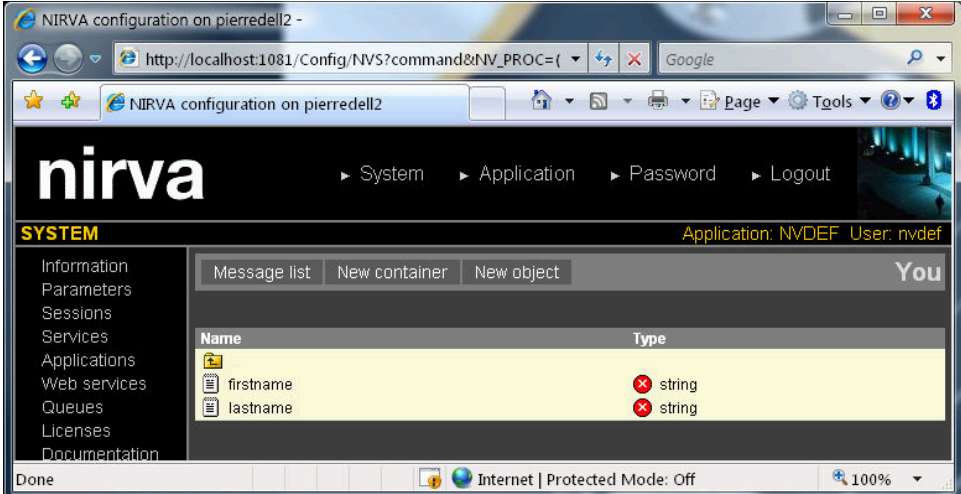
We have now the messages but no content. For editing message content: click on the message name. This enters into the structure of the message content. Let’s do it first for the “You” message:



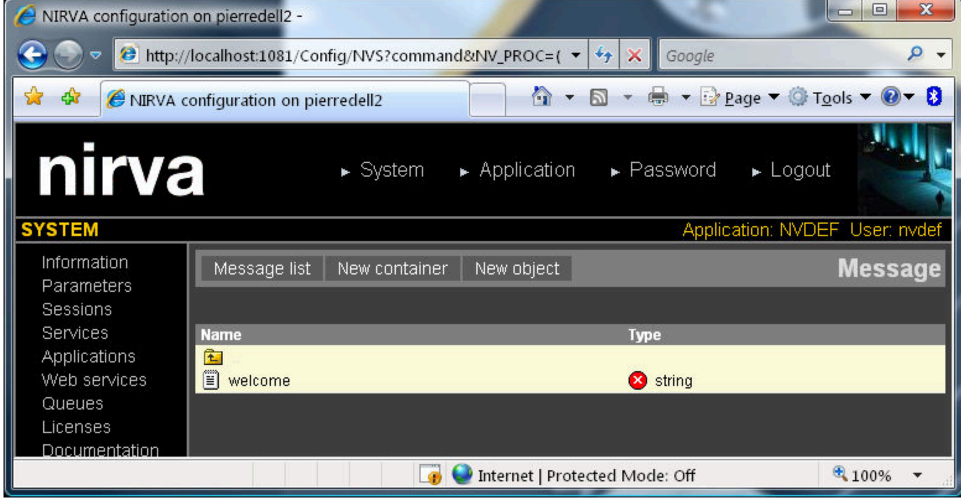
Into this message, create two string objects named respectively “firstname” and “lastname”. For that, press the “New object” button:



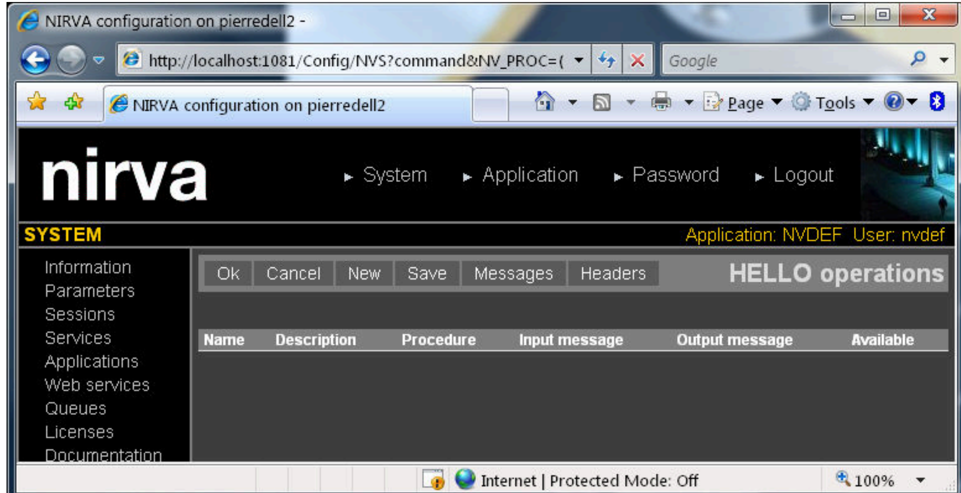
Then press the “Add object” button. Repeat the operation with the second object. The “You” message structure should look like this:



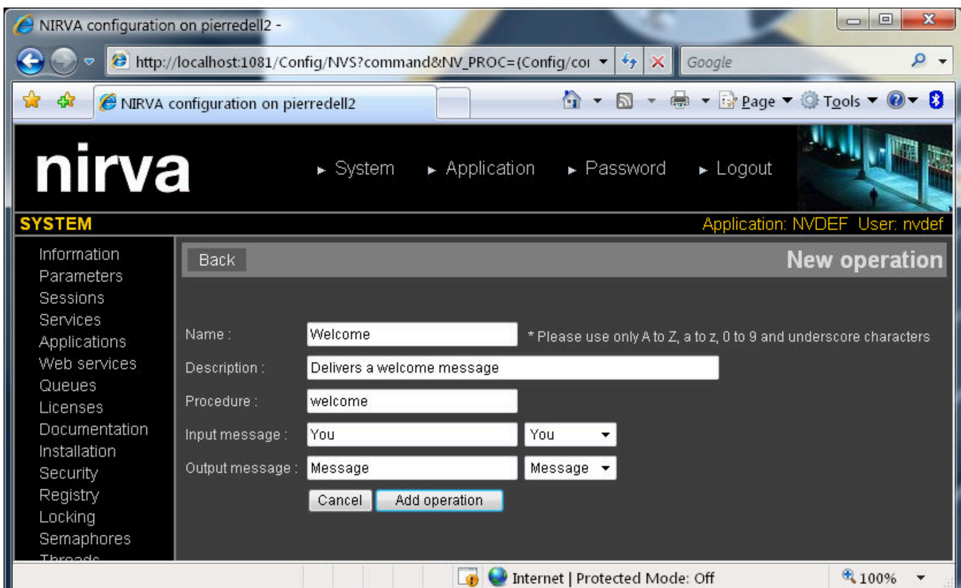
Do the same with the “Message” message by creating a string object named “welcome”:



Now we can come back to the message list by pressing the “Message list” button and switch to the operation list by clicking the “Operations” button:



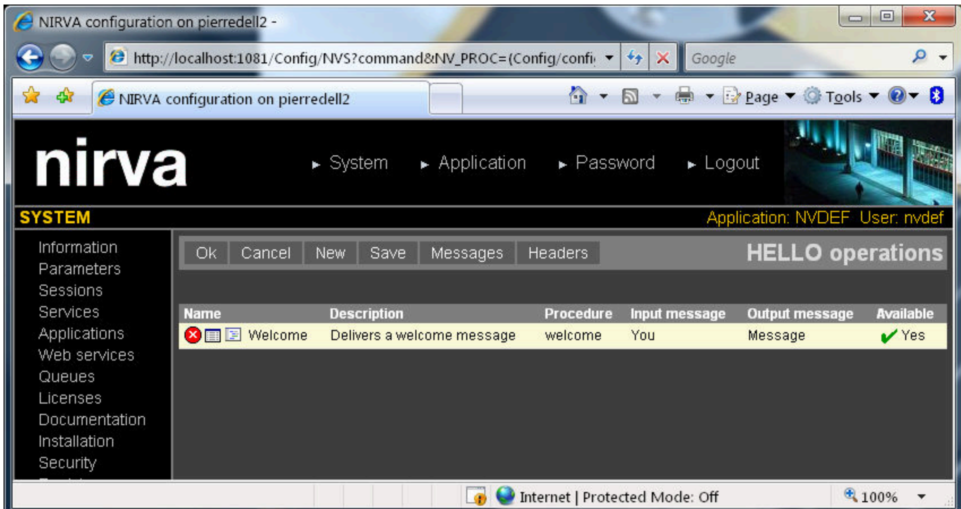
Create a new operation called “Welcome” by clicking on the “New” button:



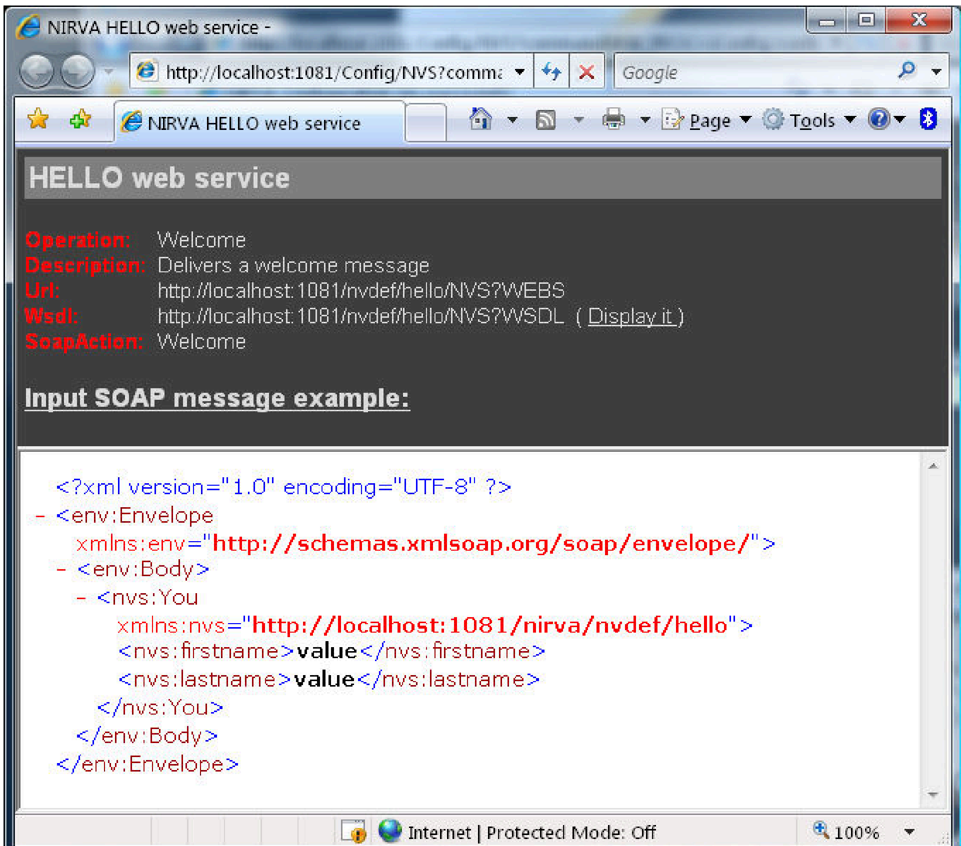
Enter the following information:

- Name is “Welcome”.
- Description is “Delivers a welcome message”.
- Procedure is “welcome”.
- Input message is “You” (chosen from the dropdown list).
- Output message is “Message” (chosen from the dropdown list).

And press the “Add operation” button. The operation list should then look like this:

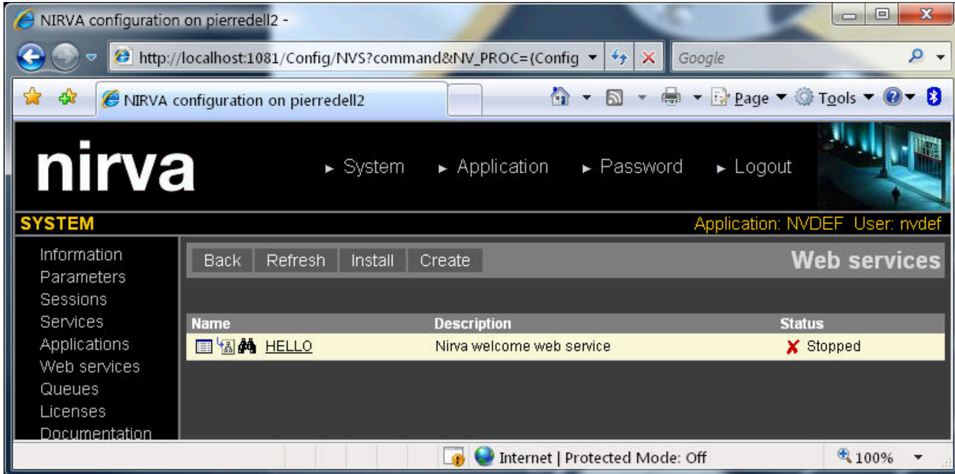


Our web service is now nearly ready. We can get some detailed information about the “Welcome” operation by clicking on the icon near the operation name:



The WSDL URL (in our example http://localhost:1081/nvdef/hello/NVS?WSDL) is available by clicking the “Display it” link from this screen. This URL will be useful for your web service client to automatically construct the messages or code for the web service (e.g. Java AXIS).

Go back to the web service list by clicking on the “Ok” button of the operation list. This also saves the HELLO web service. Nirva then displays the list of web services:



Note: the WSDL is also available from this screen clicking on the icon near the web service name.

We have now finished with the web service definition itself but the web service doesn't do anything. We must create the procedure that processes input data and delivers output data. For that, we create a file named "welcome.nvp" (we have defined the procedure name as "welcome" in the web service operation) in the Nirva/Webservices/HELLO/Procs directory with the following content:

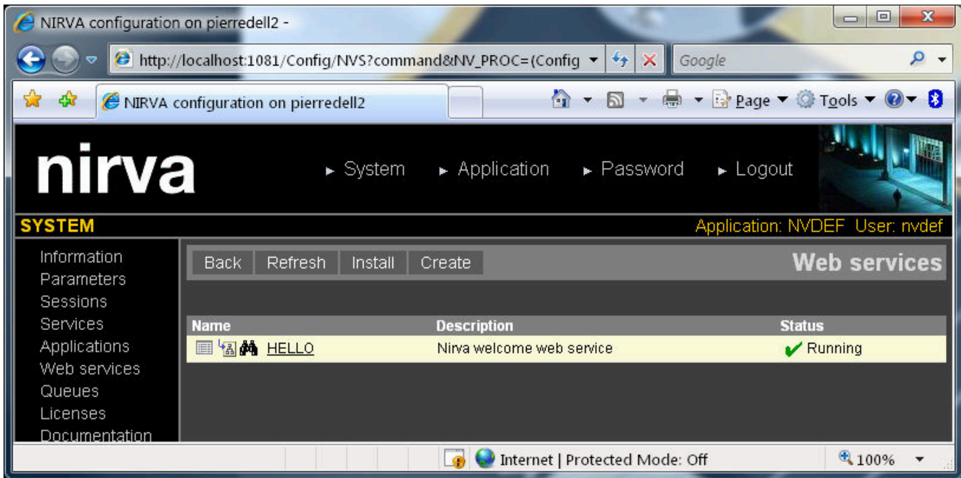
```
NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|firstname| NV_VAR=|FirstName|
NV_CMD=|OBJECT:STRING_GET_VALUE| NAME=|lastname| NV_VAR=|LastName|
NV_CMD=|OBJECT:CREATE| NAME=|welcome| TYPE=|STRING|
NV_CMD=|OBJECT:STRING_SET_VALUE| NAME=|welcome| VALUE=|Welcome to NIRVA web services, |
+ |#FirstName| + | | + |#LastName|
```

Please respect the syntax and put one command on an entire line.

This simple procedure just constructs the welcome message from the given first and last names. In this example, we use the NIRVA native language, but the procedure can be also written directly in Perl, .Net or Java language.

Starting the web service

For starting, the web service, click on the icon in the web service status column. The web service is now ready to run:



Running the web service

Testing the web service can be done with any dedicated web service tool on the market or the NIRVA nvcc tool. The first test shown is done with nvcc. The second tests is done with soapUI.

With nvcc tool

We must first create an input XML file named “welcomein.xml” with the following content:

```
<?xml version="1.0" encoding="UTF-8" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <nvs:You xmlns:nvs="http://pierredell:1081/nirva/nvdef/hello">
      <nvs:firstname>John</nvs:firstname>
      <nvs:lastname>SMITH</nvs:lastname>
    </nvs:You>
  </env:Body>
</env:Envelope>
```

This is the input SOAP message for our hello web service. Please copy this file into the Nirva/Bin directory.

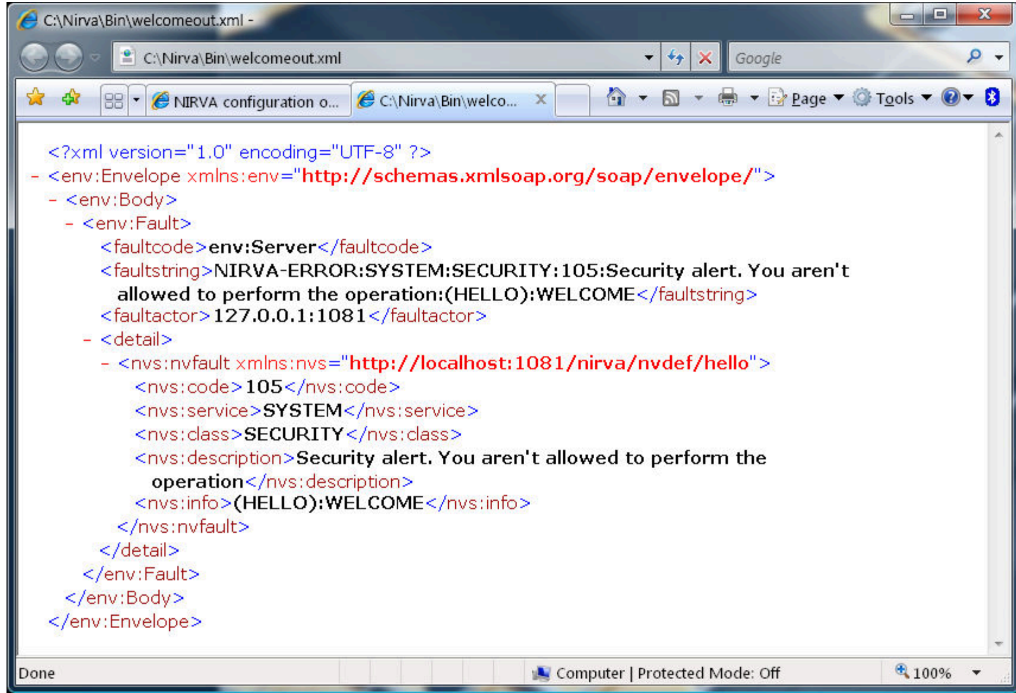
For testing the web service, open a console window, go into the Nirva/Bin directory and type the following command line:

```
nvcc -i testwebs.txt HELLO Welcome welcomein.xml welcomeout.xml
```

This instructs NIRVA to execute the Welcome operation of the HELLO web service with the input data of the “welcomin.xml”.

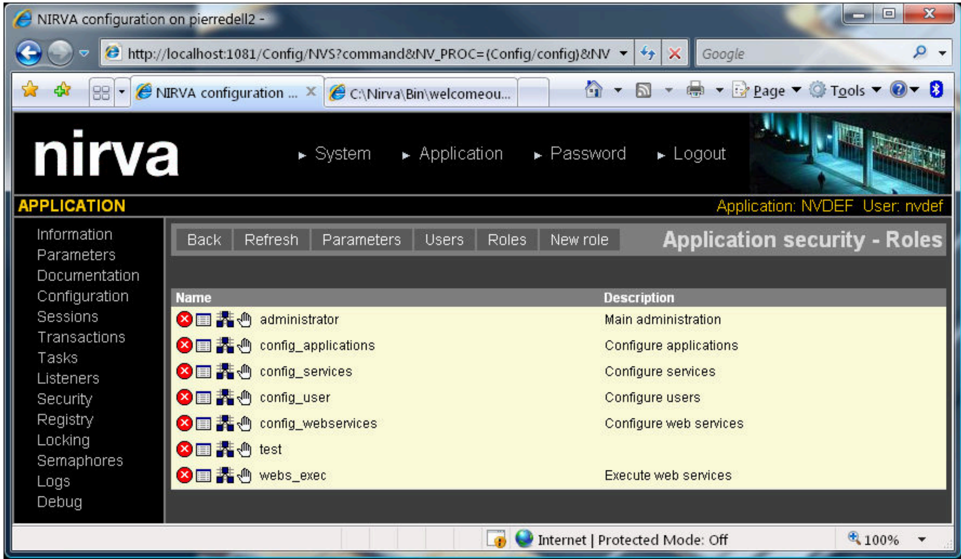
NIRVA delivers the output message into the welcomeout.xml file.

After running the command, the resulting welcomeout.xml file looks like this:

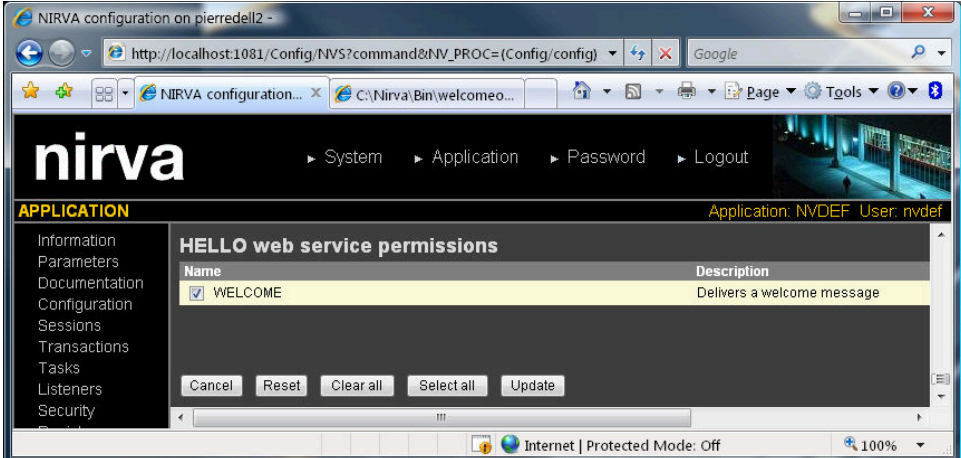


In fact, the returned message is a SOAP fault message because we forgot to allow the default user of the default application to use the web service.

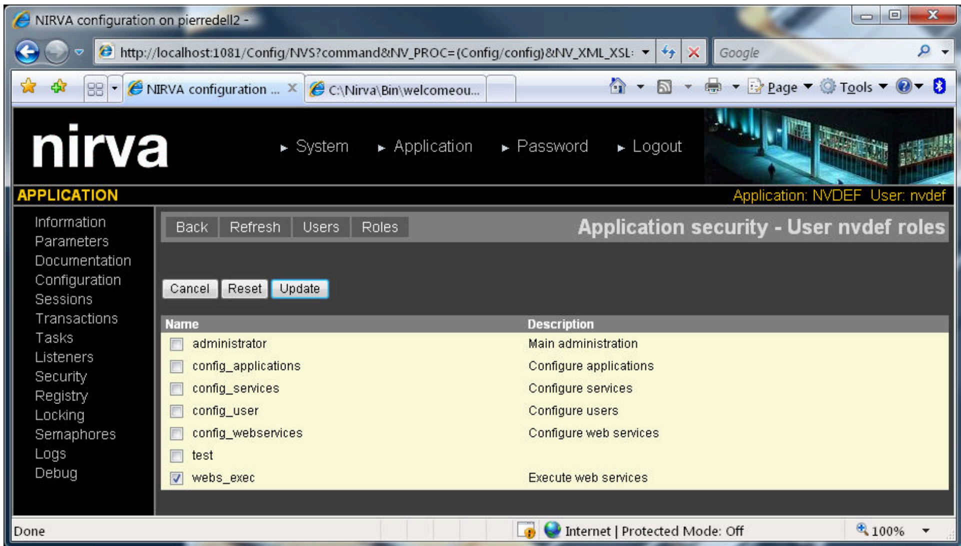
For that, we must create a new role to the default user and add the HELLO/Welcome web service permission to this role (we can also just add the permission to an existing role). Go into the Nirva configuration tool (http://localhost:1081/Config/login.htm) and login to the default application (do not enter anything in the Application field of the login screen). Then go into the application security and add a new role named “webs_exec”:



In this role, display the list of permissions, check the WELCOME permission for the HELLO web service (this should be at the bottom of the permission screen) and press the update button:



Now give to “nvdef” user the webs_exec role and press the update button:

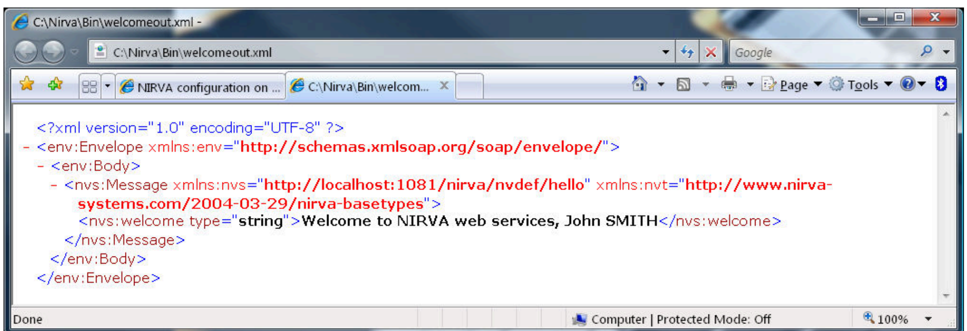


Nota: the screens may differ on your computer following the security rights you have already set.

Now, everything is OK and we can run again the web service with the command line:

```
nvcc -i testwebs.txt HELLO Welcome welcomein.xml welcomeout.xml
```

Now the welcomeout.xml file looks like this:

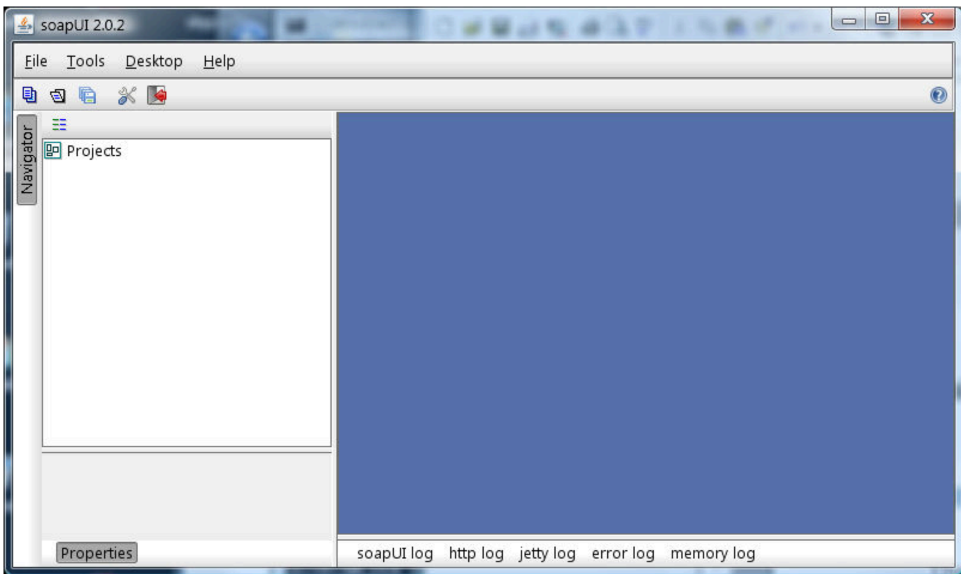


This is now the expected result of our simple HELLO web service.

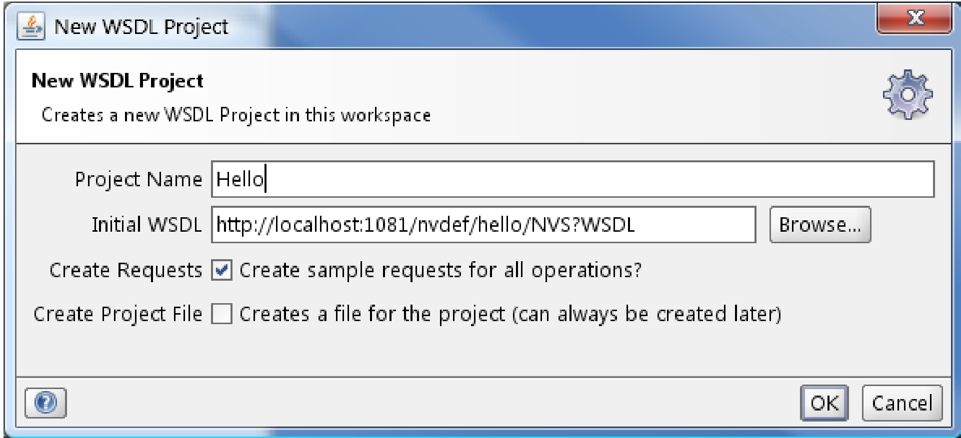
With SoapUI standard tool

The soapUI is a free tool for testing web services. It can be found on http://www.soapui.org/. We use version 2.0.2 in this documentation.

First start soapUI. This should display the following screen:

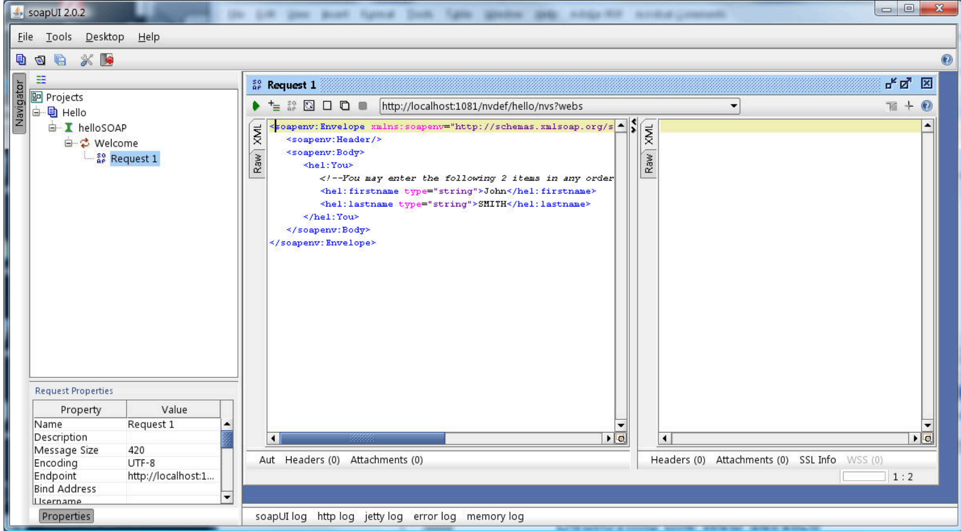


Right click on Projects and choose “New WSDL Project”:



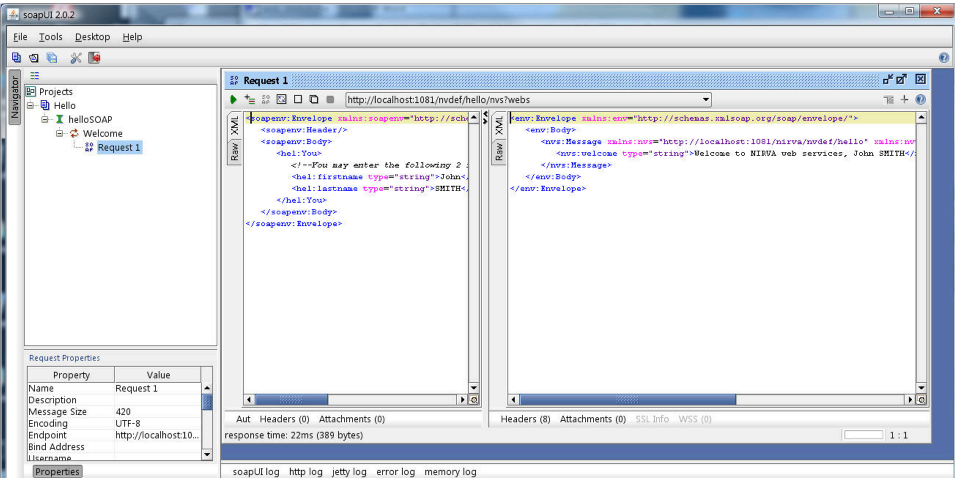
Name your project “Hello” and enter the URL of the web service as found in the Nirva configuration tool. If you run soapUI on the same computer than Nirva, this will be “http://localhost:1081/nvdef/hello/NVS?WSDL”, otherwise set the correct computer name and port in the URL.

Click “Ok”. This creates your Hello project. Expand the Hello project and double click on “Request 1”. SoapUI then automatically generates your input soap message:



Change the message by adding your first and last names in the correct places.

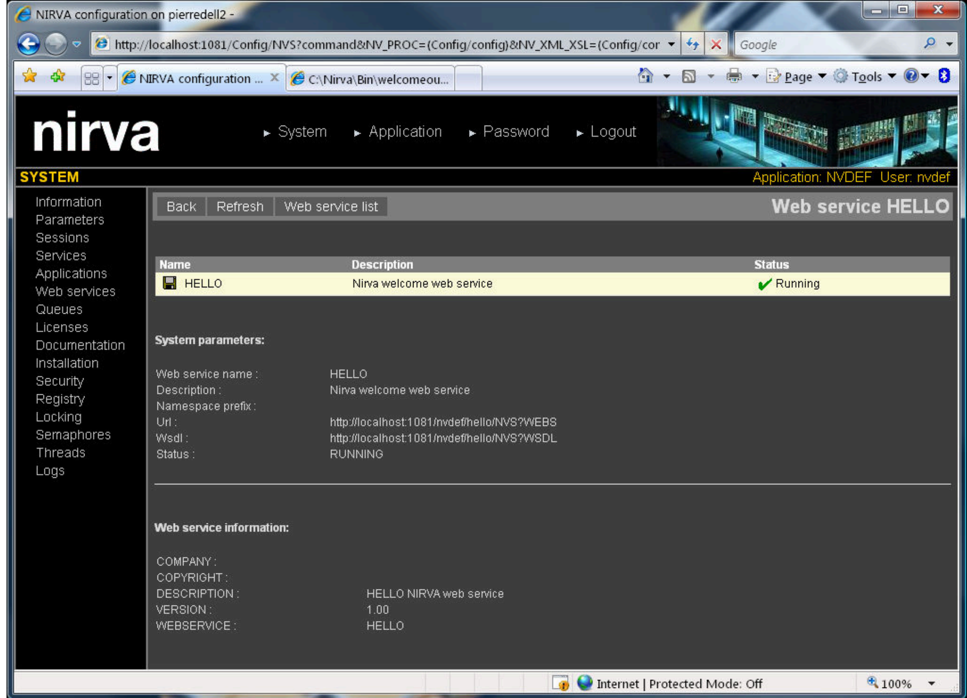
Then press the button from the Request 1 window. This sends the input message to Nirva and returns back the result message:



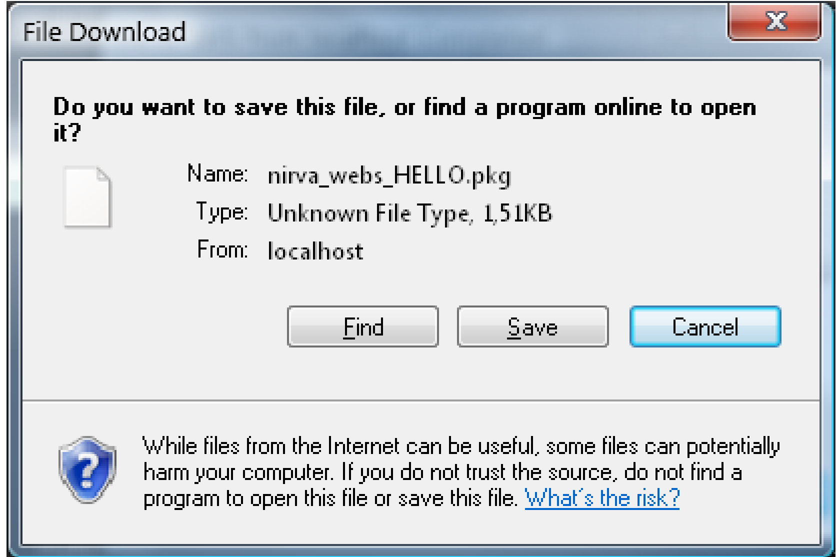
Deploying the web service

Deploying a web service means creating an installation package on one side and installing the application package on another side.

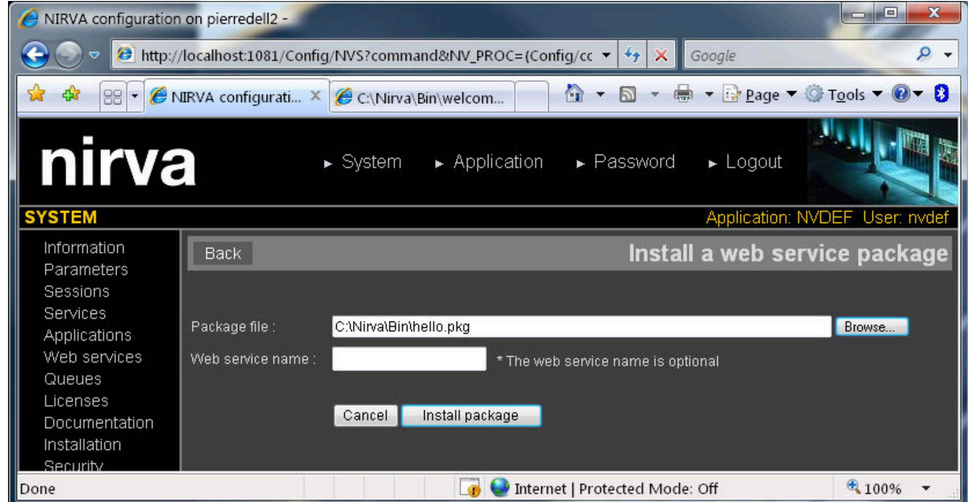
For creating the package file for the web service HELLO, go to the web service list in the configuration tool and click on the “HELLO” web service. This displays the following screen:



just click on the icon near the web service name. Then save your package file:



For installing this web service package on the target NIRVA, use the configuration tool of this target NIRVA, go to the System/Web services menu and press the install button. Then enters the path of your package file an press the “Install package” button:



The web service name is not mandatory since NIRVA is able to get it from the package file itself. Changing the web service name allows to create a copy of a web service.

After a confirmation message, the HELLO web service is installed on the target NIRVA.

Description file

The web service description file is a text file named "webservice.dsc". The description file must reside in the webservice "Files" directory.

The description file is composed of well defined sections. A new section begins with a new line starting with the '[' character followed by the section name and terminating with the ']' character.

Each section contains a succession of lines with a meaning depending of the section itself.

The description file can include comments. A comment is a line starting with ';', "//" or "\\".

Any blank line is ignored.

Here are the description file available sections:

INFO	General web service information.
SETTINGS	General settings.

Here is an example of a description file for the service "HELLO":

```
// hello.dsc : description file
// NIRVA web service
// This file should reside in the NIRVA/Webs/HELLO/Files directory

// This file contains the HELLO NIRVA web service description
// NIRVA tries to read it when loading the web service
// The hello.dsc file should be installed in the web service File director
// This file is not required but is very usefull for NIRVA configuration
```

```
// INFO section
// The INFO section gives some general web service information on the form
// infoname =info value
// Any new string can be added, removed or modified
[INFO]
WEBSERVICE = HELLO
VERSION = 1.00
DESCRIPTION = HELLO NIRVA web service
COMPANY =
COPYRIGHT =

[SETTINGS]
NSPREFIX =
```

INFO section

The INFO section gives some general web service information.

There is a single INFO section in the description file.

The section is composed of several entries of the form *infoname* = *infovalue*.

The WEBSERVICE entry is the name of the web service. It’s used by Nirva as default web service name during installation.

SETTINGS section

The SETTINGS section gives some web service information.

There is a single SETTINGS section in the description file.

The section is composed of several entries of the form *paramname* = *paramvalue*.

The NSPREFIX section gives the default namespace prefix that will be used in the web service WSDL if not given, Nirva is using the web service host parameter defined at system level (see config/system parameters). This parameter is only used at installation time if the web service doesn’t exist. After install the namespace prefix can be changed directly from the configuration tool.

Client library nvc

How to use

The nvc library allows sending any command to a Nirva server.

It’s the first connector available on client side. All other client connectors use the nvc library.

The nvc library is a full multithread library available on Window, AIX, Hpux, Solaris and Linux platforms.

The name of the dynamic library is nvc.dll for WINDOWS, libnvc.a for AIX, libnvc.sl for HPUX PA-RISC and libnvc.so for LINUX, SOLARIS and HPUX Itanium. On the SOLARIS platform, there is also a GCC version of the library named libnvcg.so for programmers using gcc as SOLARIS compiler. This gcc version of the library is given as it is without any warranty.

Client applications built with the nvc library must link with it if they use implicit linking. For windows, the “nvc.lib” file must be added with the linked libraries and for UNIX, the “–lnvc” linker option must be given.

Request

All the communication with the Nirva server is made in the context of a request.

The first thing a user must do is to open a request by giving all the necessary connection parameters for communicating with the Nirva server. This operation is done by using the *NvOpenRequest* function that gives back a request handle.

Then the user can use the request to prepare objects and send commands by using the *NvCommand* function.

Finally after finishing the operations, the user must use the *NvCloseRequest* function to close the request and free any resource maintained by the request.

Since the nvc library is a multithread library, several requests can be maintained in the same time.

Session ID

When creating a request by using the *NvOpenRequest* function, the user can give a Nirva session ID as parameter. Then, the request will work in the context of the given session.

In this way, it’s possible to connect a Nirva server session from different programs at different time.

If the session ID is not given, Nirva will create a new session when the user will send the first command.

Network connections

The client library maintains a list of network connections to the Nirva servers. This list is independent from the list of requests. When sending a command to the server, a request uses a free connection or creates a new one if there is no free connection available.

The Network connections are maintained alive for about 200 seconds. This avoids reconnecting the server for each command.

Local container

Each request encapsulates a local container. The local container is a Nirva container that is not hierarchical. It maintains a collection of Nirva objects.

The objects to be processed by the Nirva server must be first prepared in the local container, then the user sends these objects to the server, sends the commands that process the objects and gets back the resulting objects in the local container.

The local container is not able to use cached files. For using cached files from the client workstation or web server, one must use the Nirva server cached files in a share directory that is visible from the client.

Local service

The Nirva LOCAL service is a client side service that provides basic functionalities for manipulating local container objects and sending commands to the Nirva server.

The LOCAL service reference is documented in this chapter.

Command buffer

Each request maintains a command buffer. There are two ways to send commands:

The commands can be sent directly to the server.

The commands can be queued in the command buffer. Then the command buffer can be sent to the server that executes the commands. The command buffer also manages an error object for each command in order to get back the command status.

Sending a command

The *NvCommand* function allows sending a command. It uses a simple string as input that contains the command string (see the chapter describing the Nirva command syntax) and an eventual array of characters as output. This output is called output buffer.

The output buffer is generally a very small buffer even if the information to retrieve is important. In fact the retrieved information that comes from server processing is encapsulated into the objects of the local container and the local commands access only a little part of these objects at a time. For example, a current output data may be the content of a Nirva table object cell or the content of a Nirva string object.

Sending a command to the command buffer

For recording a new command in the command buffer, the command must have a parameter named “NV_COMMAND_INDEX” (that can be eventually blank) and the command must be a server command (NV_SERVICE parameter different of “LOCAL”).

The command is then inserted in the command buffer before the index given by the “NV_COMMAND_INDEX” parameter. If the “NV_COMMAND_INDEX” parameter is blank or equal to 0 or greater than the current command buffer size, the command is appended at the end of the command buffer.

Sending a command directly to the server

For sending the command directly to the Nirva server, the “NV_COMMAND_INDEX” parameter must not be given in the command.

If there are some unsent commands in the command buffer, the LOCAL service produces an error so the command buffer must be empty before sending directly a command to the server.

Sending and retrieving objects

For exchanging objects between local and server containers, the user must send the SYSTEM OBJECT SEND and SYSTEM OBJECT GET commands of the SYSTEM service. These commands are described in detail in the SYSTEM service reference in this documentation.

Variables

A request manages a list of local variables having a request scope. A variable is a named string object. The local service provides some commands for working with variables (VARIABLE class of the LOCAL service).

These variables can be used directly as a parameter value on the Nirva commands (see the Nirva command syntax for further information).

A variable name should never start with the ‘#’ character because this character is used as a variable identifier in the Nirva command. A variable name can contain spaces.

Error management

The LOCAL service provides global error information and error information for each command of the command buffer. Each error information is encapsulated in an error object.

The error object contains several strings:

CODE	Error code.
SERVICE	Service that produced the error.
CLASS	Error class. The error class can be different than the command class. Each service defines its own error classes.
INFO	Eventual information associated with the error.
DESCRIPTION	Description of the error in the language of the session.

When there is no error, the error object CODE is “0” and the other strings are blank. This is the default state.

The LOCAL service command ERROR_INFO of the REQUEST class allows getting the content of an error object.

An error is entirely identified by the SERVICE, CLASS and CODE values.

Global error

The global error object is in the default state (error CODE is “0”).

The content of the global error object depends of the command sent:

When sending a command to the LOCAL service, the global error contains the error information of the command if it has failed.

When sending directly a command to the Nirva server, the global error contains the error information of the command if it has failed.

When sending the command buffer to the server, the global error contains the error 107 (REQUEST error class) and further information can be get in the command error objects.

Command error

The command error has meaning only when using the command buffer.

There is one command error object for each command of the command buffer.

When a command is added to the command buffer, its associated error object contains 108 (REQUEST error class) as error code.

When the command buffer is sent to the server, all the command error codes have 101 (REQUEST error class) as error code.

When the command buffer has been sent to the server, there are 2 possibilities:

All the commands completed successfully. At this time, the global error code is 0 and all the command error codes also.

There was at least an error in one of the command sent. At this time, the global error code is 107 (REQUEST error class) and each command error object contains the error of its associated command. If the stop on error flag has been used when sending the command buffer, all the unprocessed commands (starting after the command that produced the error) keep the error code 101 (REQUEST class).

Example

This is a C++ example that makes a simple file copy but using the Nirva server.

The program creates 2 local file objects, a source one and a destination one. Then it sends the source file object to the Nirva server and requests back this object to put it into the destination file object.

This example works with the local Nirva server (the default TCP/IP address is 127.0.0.1). In order to change the default connection parameters, one must give them to the NvOpenRequest function.

This file must be compiled by linking the nvc library.

This example is to be taken as a school case for showing the basis of nvc library. In fact, if only a file copy has to be done, there are other simplest ways.

```
// nvcopy.cpp : Nirva example. Copy a file by sending it to the Nirva server
// Usage: nvcopy filein fileout


#include <stdio.h>
#include <string.h>

#include "nvc.h"        // Nirva nvc header


// Global variables
char CommandBuffer[2048];
char OutputBuffer[2048];

// Function declarations
void Cleanup(NVREQUEST Request);
bool CheckError(NVREQUEST Request, int Result);


// Entry point
int main(int argc, char* argv[])
{
  int Result;

  // test the command line
  if(argc < 3)
  {
         printf("\n\nUsage: nvcopy filein fileout\n");
         return -1;     // bad number of argument
  }

  // Open the Nirva request with default connection string
```

```
NVREQUEST Request = NvOpenRequest("");
if(Request == NULL)
{
        printf("\n\nError opening request\n");
        return -1;
}

// Creates the source file object and attach the source file to it
sprintf(CommandBuffer, "NV_CMD =|LOCAL:OBJECT:CREATE| NAME=|Source| TYPE=|FILE| \
                        FILENAME=|%s|", argv[1]);
 // Sends the Nirva command
 Result = NvCommand(Request, CommandBuffer, NULL, 0, NULL);
if(!CheckError(Request, Result))      // Check the result
{
        Cleanup(Request);
        return -1;
}

// Test if the source file exist
sprintf(CommandBuffer, "NV_CMD =|LOCAL:OBJECT:FILE_EXIST| NAME=|Source|");

// This command use the output buffer
Result = NvCommand(Request, CommandBuffer, OutputBuffer, sizeof(OutputBuffer), NULL);
if(!CheckError(Request, Result))
{
        Cleanup(Request);
        return -1;
}

// If the source file exist, the output buffer contains "YES"
if(strcmp(OutputBuffer, "YES") != 0)
{
        printf("\n\nSource file doesn't exist\n");
        Cleanup(Request);
        return -1;
}

// Creates the destination file object and attach the destination file to it
sprintf(CommandBuffer, "NV_CMD =|LOCAL:OBJECT:CREATE| NAME=|Destination| TYPE=|FILE| \
                        FILENAME=|%s|", argv[2]);

Result = NvCommand(Request, CommandBuffer, NULL, 0, NULL);
if(!CheckError(Request, Result))
{
        Cleanup(Request);
        return -1;
}


// Sends the source object to the server
sprintf(CommandBuffer, "NV_CMD=|OBJECT:SEND| NAME=|Source|");
Result = NvCommand(Request, CommandBuffer, NULL, 0, NULL);
if(!CheckError(Request, Result))
{
        Cleanup(Request);
        return -1;
}

// Gets back the server source object and put it into the destination object
```

```
  sprintf(CommandBuffer, "NV_CMD=|OBJECT:GET| NAME=|Source| LNAME=|Destination|");
   Result = NvCommand(Request, CommandBuffer, NULL, 0, NULL);
  if(!CheckError(Request, Result))
  {
          Cleanup(Request);
          return -1;
  }

  Cleanup(Request);

  // Everything OK
  return 0;
} // End main


// Cleanup the request and the library before exit
void Cleanup(NVREQUEST Request)
{
  // First test if there is an open session
  strcpy(CommandBuffer, "NV_CMD=|LOCAL:REQUEST:GET_SESSION_ID|");
  int Result = NvCommand(Request, CommandBuffer, OutputBuffer, sizeof(OutputBuffer),
NULL);
  if((Result == 1) && strlen(OutputBuffer))
  {
          // If there is one, close it
          sprintf(CommandBuffer, "NV_CMD=|Session:close|");

          NvCommand(Request, CommandBuffer, NULL, 0, NULL);
  }

  // Close the request
  NvCloseRequest(Request);

  // Cleanup the library
  NvExitLib();
}

// Checks the Nirva command error
// Result is -1 if bad request, 0 if error and 1 in case of success
bool CheckError(NVREQUEST Request, int Result)
{
  if(Result == -1) // bad request
  {
          printf("\nInvalid request");
          return false;
  }
  if(Result == 0) // Error in command
  {
          // Error in the command
          // We get and display error information
          char ErrorString[256];



          // Get error service
          NvCommand(Request, "NV_CMD=|LOCAL:REQUEST|:ERROR_INFO| ERROR_STRING=|SERVICE|",
                              ErrorString, sizeof(ErrorString), NULL);
          printf("\n\tError service     : ");
          printf(ErrorString);
```

```
        // Get error class
        NvCommand(Request, "NV_CMD=|LOCAL:REQUEST|:ERROR_INFO| ERROR_STRING=|CLASS|",
                          ErrorString, sizeof(ErrorString), NULL);
        printf("\n\tError class       : ");
        printf(ErrorString);

        // Get error code
        NvCommand(Request, "NV_CMD=|LOCAL:REQUEST|:ERROR_INFO| ERROR_STRING=|CODE|",
                          ErrorString, sizeof(ErrorString), NULL);
        printf("\n\tError code        : ");
        printf(ErrorString);


        // Get error description
        NvCommand(Request, "NV_CMD=|LOCAL:REQUEST|:ERROR_INFO| \
                              ERROR_STRING=|DESCRIPTION|",
                        ErrorString, sizeof(ErrorString), NULL);
        printf("\n\tError description : ");
        printf(ErrorString);


        // Get error information
        NvCommand(Request, "NV_CMD=|LOCAL:REQUEST|:ERROR_INFO| ERROR_STRING=|INFO|",
                          ErrorString, sizeof(ErrorString), NULL);
        printf("\n\tError info        : ");
        printf(ErrorString);
        printf("\n\n");

        return false;
  }

  return true;
} // End CheckError
```

Function reference

Overview

The Nirva client library provides only 4 functions:

- *NvOpenRequest* creates a new request and returns a request handle.
- *NvCloseRequest* closes a previously opened request.
- *NvCommand* sends commands to Nirva in the context of a request.
- *NvExitLib* frees the resources of the library.

Functions

NvOpenRequest

Syntax

NVREQUEST NvOpenRequest(const char *ConnectString)

Description

This function opens a new client request. The parameters of the request are given in the ConnectString parameter. The function returns a handle that is used with other functions to communicate with the request.

Parameters

ConnectString

Connection string. This string contains the request parameters. This is a succession of pairs `ParameterName="ParameterValue"`. The pair separator is the blank character. The parameter name is case insensitive. The '=' character must follow the parameter name. The parameter value must be enclosed in double quote characters. If the parameter value contains itself a double quote character, it must be preceded by another double quote character.
Here are the available connection string parameters:

- *Server* is the Nirva server TCP/IP address or machine name. This can be followed by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is installed. For example `Server="135.12.13.14:2035"` is a valid Server address. The default value for this parameter is `"127.0.0.1:1081"`.
  It's possible to define several server addresses (but with a same port) separated by a semicolon character (;). Then the connection will occur in a random way on one of the servers. This allows a simple load balancing solution without requiring a load balancing box. This is a simple load balancing solution and the connected servers must all run. Example: Server="192.168.20.17;192.168.20.19:80" will connect one of the servers 192.168.20.18 or 192.168.20.18 on the port 80.
  A proxy server can also be defined in a single address. At this time the format is the following:
  `proxy::`*`protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address`* where *protocol* is the protocol for the proxy server. It can be "http" or "https"; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for

the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.

- *ConnectionTimeOut* is the time out for establishing the TCP/IP connection to the Nirva server. The default value is 10 seconds.
- *Session* is the Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn't exist. If the parameter is not given, Nirva creates a new session. The newly opened session stays opened until the time out occurs or until it's explicitly closed (by a SYSTEM CLOSE command) or after sending a command (or the command buffer) if the auto close mode has been set.
- *AutoClose* is the auto close mode. If this parameter is set to "YES", the session is automatically closed after sending a command or the content of the command buffer to the server. Otherwise, the session stays open until it's explicitly closed by a SYSTEM CLOSE command or when the time out occurs.
- *SessionTimeOut* is the time out value in seconds for a session. It's used only when a new session is to be created (so when the *Session* parameter is not provided or blank). If *SessionTimeOut* is "0", the default Nirva time out will be used. This is the default.
- *Application* is the name of the Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In fact, a session is always opened in the context of an application. If the application parameter is not provided, Nirva uses the default application named "NVDEF".
- *User* is the application user name. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *Password* is the application user password. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank).
- *NewPassword* is the application new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
- *NewPasswordConfirm*. This parameter may be provided when the *NewPassword* parameter has been given. It allows Nirva to check if the new password is correct.
- *Open* is the name of the NIRVA procedure to call when opening the new session. The default is "session_open". This parameter is

significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”.

- *Close* is the name of the NIRVA procedure to call when closing the new session. The default is “session_close”. This parameter is significant only when a new session is to be created (so when the *Session* parameter is not provided or blank). In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”.
- *Ssl* is the SSL mode. If this parameter is set to “YES”, the request will use the HTTPS protocol to communicate to the NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the NIRVA server.
- *Certificate* is the optional client certificate (with SSL connection only). If the server is configured to require a client certificate, this one must be given using this option. The certificate parameter is the name of a certificate file in pem format that contains both the certificate and the private key (concatenated). If a password is required it may be added at the end of the parameter with a semicolon character separator.
- *MaxMsgSize* is the maximum amount of memory (in kilobytes) that a message (sent to NIRVA server) can use. If the message overrides this limit, NIRVA creates a temporary file to manage its data. This allows transmitting important information into messages (especially big files) without consuming too much memory. The default value is 20 Mbytes (20 480 kilobytes) that should be a good compromise between performance and memory management on the client side. The minimum value is 10 kilobytes.
- *TempDir* is the directory name where the nvc library can write the temporary files. If this parameter is not provided, NIRVA uses the current directory.
- *Unicode* tells if client is Unicode. When this parameter is set to “YES”, all the data sent to the server are supposed to be UTF-8 encoded and all data coming from server are automatically converted to UTF-8 by the server. When not in Unicode mode, the client is supposed to work in ISO-8859-1 character set.
- *Sso* is the SSO (Single sign-on) flag. If this parameter is set to “YES” and if the Nirva application has been configured to authenticate users via SSO based on client choice (Nirva or single sign-on mode), Nirva will authenticate the user via SSO.
- *SsoPrincipal* is the SSO (Single sign-on) principal name when using Kerberos protocol. This parameter has no meaning when not using SSO or when using SSO with NTLM.

- *SocketLog* is the name of an optional log file at socket (http) level.

Return value

This function returns a handle on the newly created request and NULL in case of error. This returned handle must be used as first parameter of the other functions. The handle is defined as void*.

NvCloseRequest

Syntax

void NvCloseRequest(NVREQUEST Request)

Description

This function closes a request previously opened with the *NvOpenRequest* function.

It frees any resource used by the request.

Parameters

Request	Handle of the request as returned by the *NvOpenRequest* function.

Return value

None.

NvCommand

Syntax

int NvCommand(NVREQUEST Request, const char *Command, const char *Buffer, int BufferSize, int *DataSize)

Description

This function sends a command to Nirva. The command can be a LOCAL service command, a server command sent directly to the Nirva server or a server command sent to the command buffer.

This command receives the output buffer (when there is one) into a buffer supplied by the service. If the supplied buffer is not large enough, one can reallocate a buffer to the required size (the size returned by the DataSize parameter) and then issue the command “LOCAL:GET_LAST_OUTPUT_BUFFER”:

Example:

```
int BufferSize = 100;
char *Buffer = (char*)malloc(BufferSize);
int DataSize = 0;
if(NvCommand(hRequest, Command, Buffer, BufferSize, &DataSize) != 1)
{
  free(Buffer);
  return false;
}
if(DataSize >= BufferSize)
{
  // Need to realloc
  char *NewBuffer = (char*)realloc(Buffer, DataSize+1);
  Buffer = NewBuffer;
  BufferSize = DataSize+1;
  NvCommand("NV_CMD=|COMMAND:GET_LAST_OUTPUT_BUFFER|",
                     Buffer, BufferSize, &DataSize);
}
free(Buffer);
return true;
```

Parameters

Request	Handle of the request as returned by the *NvOpenRequest* function.
Command	Character buffer containing the command string. Please see the Nirva command syntax chapter for further information about the Nirva command.
Buffer	Output buffer. This is the address of a character buffer in which Nirva will write the command output if the command delivers output. See the LOCAL and SYSTEM references to see which commands return information in the output buffer. This parameter can be NULL if no output buffer has to be used.
BufferSize	Size of the output buffer pointed by the *Buffer* parameter.
DataSize	This is a pointer to an integer that will receive the real size of the output data if the output buffer is used. This can be used to control that the output buffer is large enough. For example, if the output buffer size is 100 and the required data is 150, only the 100 first bytes (99 in fact) will be returned in the output buffer but the function will write 150 in the DataSize parameter. If DataSize is NULL, Nirva doesn't return the data size.

Return value

This function returns –1 if the Request parameter doesn't point to a valid request, 0 in case of error and 1 in case of success.

NvExitLib

Syntax

void NvExitLib()

Description

This function frees all the resources allocated by the library including the current opened requests.

It must be used just one time in the program exit code. There is no corresponding function for initializing the library nvc. In fact this is done automatically when creating the first request.

Parameters

None.

Return value

None.

Local service reference

Overview

This chapter describes in detail all the Nirva LOCAL service commands. The Nirva LOCAL service is a client side service that provides basic functionalities for manipulating objects and sending commands to the Nirva server. It's accessible from all NIRVA connectors except the XML and SOAP connectors. It's implemented in the client library nvc.

For each command, the reference gives the command name, the command description and the parameter list.

The parameters described in this chapter are command specific parameters. For general parameters, please refer to the Nirva command syntax chapter.

Output buffer

Some of the LOCAL or SYSTEM service commands return string information.

This information is first returned in the output buffer given in the *NvCommand* function.

The returned string information is also written in a request variable named "NV_RESULT" by default. The name of this variable can be changed by using the command parameter named "NV_VAR". See the Nirva command syntax for further information.

Classes

Here are the available LOCAL service classes:

REQUEST	Manages server commands.
VARIABLE	Variables management.
OBJECT	Local Container object management.

Error codes

OBJECT Class

Value	Description
101	No object type
102	Unknown object type
103	Cannot create local object
104	No file name defined for a local file object
105	Cannot set the local object data
106	Cannot get the local object
107	Cannot get the local object data
108	The local object already exist
109	Invalid object index
110	The object doesn't exist
111	The objects are the same
112	The objects are not compatible
113	Cannot copy objects
114	Invalid string list index
115	Invalid indexed string list key
116	Error in search table
117	Error in save table
118	Error in load table
119	Error in import table
120	Error in export table
121	Invalid table cell
122	Error in set cell line
123	Error in insert cell line
124	Error in remove rows
125	Error in clear rows
126	Bad column name
127	Error in modify columns
128	Cannot create file for file object
129	Cannot get file size
130	Cannot clear file for file object
131	Cannot append file to file object

Value	Description
132	No file name in file object
133	Cannot append binary data
134	Cannot save binary data to file
135	Cannot load binary data from file
136	Error in table select from
137	Cannot set the primary key
138	Cannot set row data
139	Bad object type
140	Cannot join tables
141	Cannot add table
142	Cannot get row index from primary value
143	Cannot add records to the table
144	Cannot compress local file
145	Cannot decompress local file

REQUEST Class

Value	Description
101	The command has been sent to the server but not executed.
102	Bad command parameter
103	Cannot get a connection to the Nirva server
104	Network error
105	Bad server. The server is not a Nirva server
106	Synchronization error
107	Error in one of the commands sent to the server
108	This command has not been sent to the server
109	The command buffer contains some un-sent commands
110	Unknown command
111	Unknown class
112	Invalid command index
113	Proxy authentication error

Commands

OBJECT class

The OBJECT class provides an important set of commands to manipulate the Nirva container objects.

The LOCAL container is a non-hierarchical container so it cannot contain subcontainers.

CLEAR_ALL

Description

This command removes all objects from the local container.

Use output buffer

No.

Parameters

None.

COPY

Description

This command copies an object to another object in the local container. Both source and destination objects must exist and must be of the same type.

Use output buffer

No.

Parameters

SNAME	Source object name. The object must exist on the local container. A file object must have a file name attached to it.
NAME	Destination object name. The object must exist on the local container and must have the same type than the source object. A file object must have a file name attached to it.
OFFSET	Offset. This parameter is used for file and binary objects. It gives the offset in bytes of the source object from which the copy will start.

	The default value is ”0” (beginning of file or binary data).
NUM_BYTES	Number of bytes to copy. This parameter is used for file and binary objects. It gives the number of bytes of the source object to copy. The default value is ”-1” (until the end of file or binary data).

CREATE

Description

This command creates a new object in the local container.

Use output buffer

No.

Parameters

NAME — Object name. The object name is case insensitive is case insensitive and cannot contain any of the ‘/’, ‘\’, ‘.’ or space characters. The object name cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. The default value for this parameter is “NV_OBJ_DEFAULT_NAME”.

TYPE — Object type. This parameter is mandatory.
The object type can take the following values:

- “BOOLEAN” for a boolean object.
- “INTEGER” for an integer object.
- “STRING” for a string object.
- “STRINGLIST” for a string list object.
- “INDSTRINGLIST” for an indexed string list object.
- “TABLE” for a table object.
- “FILE” for a file object.
- “BINARY” for a binary object.

REPLACE — Replace mode. If this parameter is set to “YES”, Nirva replaces an object with the same name if it exists. Otherwise, Nirva sends an error message if the object already exists. The default is “NO”.

CASE_SENSITIVE — This parameter is only used with indexed string list object type. If it’s set to “YES”, the indexed string list keys will be case sensitive. The default is “NO”.

PERSIST — This parameter gives the persistent value for file objects. It’s only used if the object to create is a file object.

The parameter can take value “0” for temporary files, or “-1” for persistent files. Cached files are not authorized on the local container.
The default value is “-1” (persistent) except if the FILENAME parameter is not given or is blank. At this time, the default value is “0” (temporary).

FILENAME

This parameter gives the file name of the file object.
The FILENAME parameter is only used if the object to create is a file object.
If this parameter is not provided, Nirva creates itself a file name following information of the EXTENSION, PREFIX, SUFFIX and DIRECTORY parameters.

DIRECTORY

This is the name of a directory where Nirva will put the local file if the object to create is a file object and the FILENAME parameter is not given.
If this parameter is not provided or is blank, Nirva uses the current directory.

EXTENSION

This is the file extension to use. The EXTENSION parameter is only used if the object to create is a file object. If this parameter is not provided or is blank, Nirva uses “.obj” for persistent files and “.tmp” for temporary. If the point character is omitted in the EXTENSION parameter, Nirva adds it.

PREFIX

This is the file prefix to use. The PREFIX parameter is only used if the object to create is a file object.
The prefix, if provided is used by Nirva to automatically create the file name.

SUFFIX

This is the file suffix to use. The SUFFIX parameter is only used if the object to create is a file object.
The suffix, if provided is used by Nirva to automatically create the file name.

VALUE

Object value. Allows to directly setting a value for objects BOOLEAN, INTEGER, STRING, STRINGLIST, INDSTRINGLIST and TABLE.

- For a BOOLEAN object this parameter can take values “TRUE” or “FALSE”. If another value is given, Nirva uses “FALSE” as object value.
- For an INTEGER object this is directly the object value. This must be a numeric value.
- For a STRING object this is directly the object value.
- For a STRINGLIST object, this is the values controlled by the SEPARATOR parameter.
- For an INDSTRINGLIST object, this is the values controlled by the KEYSEP and VALSEP parameters.
- For a TABLE object, this is the values controlled by the ROWSEP, COLSEP and LINESEP parameters.

SEPARATOR

Multi string separator for STRINGLIST object value. This parameter has meaning only when a VALUE parameter is given. It defines a character or string separator in the given value. Then NIRVA will add as many strings as the number of separators found (plus one). For example, if the value is “`v1;v2;v3`” and the separator is “`;`”, NIRVA will add the strings “`v1`”, “`v2`”

	and "v3" to the string list object. The SEPARATOR parameter may also take special values "NVLINE", "NVTAB" or "NVSPACE" for respectively line (or pair CR/LF) separator, tabulation separator or space separator. In the case of a space separator, several successive spaces are assimilated to a single separator.
KEYSEP	Key separator for INDSTRINGLIST object value. This parameter has meaning only when a VALUE parameter is given. It defines the key separator in the given VALUE parameter. It can be any string. The default is the ';' character (semicolon).
VALSEP	Value separator for INDSTRINGLIST object value. This parameter has meaning only when a VALUE parameter is given. It defines the value separator in the given VALUE parameter. It can be any string. The default is the '=' character.
ROWSEP	Row separator for TABLE object value. This parameter has meaning only when a VALUE parameter is given. It defines the string that separates the rows in the value. The default is the line feed character (\n).
COLSEP	Column separator for TABLE object value. This parameter has meaning only when a VALUE parameter is given. It defines the string that separates the columns in the value. The default is ";".
LINESEP	Line separator for TABLE object value. This parameter has meaning only when a VALUE parameter is given. It defines the string that separates the lines for columns containing several lines in the value. The default is "|".
COLUMNS	Name and descriptions of the columns for a table object. The names must be separated by a semicolon character (";"). Each value is composed of 3 parts separated by a ":" character: NAME:DESCRIPTION:NUMERIC. NAME is the column name, it cannot contain any blank or special character (they will be removed without error if there are), DESCRIPTION is a free text describing the column and NUMERIC, when set to "Y" tells that the column is numeric. The DESCRIPTION and NUMERIC part are optional. Here are some valid entries: "COL1;COL2;COL3" or "COL1;COL2:My column 2:Y;COL3:My column 3".

EXIST

Description

This command returns "YES" in the output buffer if the object exists on the local container and "NO" otherwise.

Use output buffer

Yes.

Parameters

NAME	Object name.

GET_NAME

Description

This command returns the name of an object of the local container given an object index.

In conjunction with the OBJECT GET_NUM commands, this command can be used to enumerate the container object names.

Use output buffer

Yes.

Parameters

INDEX	This is a number starting at one and having the maximum value equal to the number of objects returned by the GET_NUM command.

GET_NUM

Description

This command returns the number of objects of the local container.

Use output buffer

Yes.

Parameters

none

GET_TYPE

Description

This command returns the type of an object of the local container given an object name.

The result is in the output buffer. It can be BOOLEAN, INTEGER, STRING, STRINGLIST, INDSTRINGLIST, TABLE, FILE, BINARY or UNKNOWN.

Use output buffer

Yes.

Parameters

NAME	Object name. The object must exist on the input container.

REMOVE

Description

This command removes the given object from the local container.

Use output buffer

No.

Parameters

NAME	Object name.

SET_NAME

Description

This command allows changing the name of an existing object. The given object must exist.

If an object having the same name that the new object name exists, the command fails.

Use output buffer

No.

Parameters

NAME	Object name. The object must exist on the input container.
NEW_NAME	New object name. If an object having the same name that the new object name exists, the command fails.

BOOLEAN_GET_VALUE

Description

This command returns the value of a boolean object in the output buffer. The object must exist and must be of boolean type. The return value is “TRUE” or “FALSE”.

Use output buffer

Yes.

Parameters

NAME	Object name.

BOOLEAN_SET_VALUE

Description

This command sets the value of a boolean object. The object must exist and must be of boolean type.

Use output buffer

No.

Parameters

NAME	Object name.
VALUE	Object value. This parameter can take values “TRUE” or “FALSE”. If another value is given, Nirva uses “FALSE” as object value. The default is “FALSE”.

INTEGER_GET_VALUE

Description

This command returns the value of an integer object in the output buffer (as a string). The object must exist and must be of integer type.

Use output buffer

Yes.

Parameters

NAME	Object name.

INTEGER_SET_VALUE

Description

This command sets the value of an integer object. The object must exist and must be of integer type.

Use output buffer

No.

Parameters

NAME	Object name.
VALUE	Object value. This parameter must be a numeric value.

STRING_GET_VALUE

Description

This command returns the value of a string object in the output buffer. The object must exist and must be of string type.

Use output buffer

Yes.

Parameters

NAME	Object name.

STRING_SET_VALUE

Description

This command sets the value of a string object. The object must exist and must be of string type.

Use output buffer

No.

Parameters

NAME Object name.

VALUE Object value.

STRINGLIST_ADD_STRINGLIST

Description

This command adds all entries of a string list object to another string list object. Both objects must exist and must be of string list type.

Use output buffer

No.

Parameters

SNAME	Source object name.
NAME	Destination object name. All the source object entries are added to the destination object.

STRINGLIST_GET_SIZE

Description

This command returns the total number of entries of a string list object in the output buffer. The object must exist and must be of string list type.

Use output buffer

Yes.

Parameters

NAME	Object name.

STRINGLIST_GET_VALUE

Description

This command returns the value of a string list object entry in the output buffer. The object must exist and must be of string list type.

Use output buffer

Yes.

Parameters

NAME	Object name.
INDEX	Index of the entry. This index starts at 1. The INDEX value can also take the value “FIRST” for retrieving the first entry value or “LAST” for retrieving the last entry value. The default value is “LAST”.

STRINGLIST_GET_VALUES

Description

This command all or a range of values of a string list object in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.
WHAT	Range of index entries to get. If not given or blank or set to the value “NV_ALL”, the command retrieves all entries. If set to a value “m-n”, the command retrieves the entries from index m to index n (included). If a single index value is given, the command retrieves only the corresponding value so the command is then similar to the STRINGLIST_GET_VALUE command.
SEPARATOR	Separator for the output string. The default is “;”.

STRINGLIST_INSERT

Description

This command inserts a new entry in a string list. The entry is inserted before the given index. The object must exist and must be of string list type.

Use output buffer

No.

Parameters

NAME	Object name.
VALUE	New object entry value.
INDEX	Index of the entry where the insertion must occurs. The new entry is inserted before the given index. This index starts at 1. If the INDEX parameter is not provided or is blank, Nirva just adds the entry value to the end of the list. This is the default.
SEPARATOR	Multi string separator. This parameter has meaning only when the INDEX parameter is not given or is blank. It defines a character or string separator in the given value. Then NIRVA will add as many strings as the number of separators found (plus one). For example, if the value is "`v1;v2;v3`" and the separator is "`;`", NIRVA will add the strings "`v1`", "`v2`" and "`v3`" to the string list object. The SEPARATOR parameter may also take special values "NVLINE", "NVTAB" or "NVSPACE" for respectively line (or pair CR/LF) separator, tabulation separator or space separator. In the case of a space separator, several successive spaces are assimilated to a single separator.

STRINGLIST_REMOVE

Description

This command removes one or all entries of a string list object. The object must exist and must be of string list type.

Use output buffer

No.

Parameters

NAME	Object name.

INDEX	Index of the entry. This index starts at 1. The INDEX value can also take the value “FIRST” for removing the first entry, “LAST” for removing the last entry or “ALL” for removing all entries. The default value is “ALL”.

STRINGLIST_SEARCH

Description

This command searches for a given value in the string list and returns the found indexes in the output buffer. They are separated with the ‘;’ character.

Use output buffer

Yes.

Parameters

NAME	Object name.
VALUE	Value to search. If the string terminates with the * character, the command searches for all values starting with the string before the * character. For example if value is “t*”, the command searches all values starting with “t”
CASE	Case sensitive search. This can be “YES” or “NO” (default).
FIRST_ONLY	If this parameter is set to “YES”, the command retrieves only the first index of the found ones. If set to “NO, it retreives all the found indexes. The default is “YES”.

STRINGLIST_SET_VALUE

Description

This command sets the value of an entry of a string list. The list entry is referenced by an index starting at 1. The object must exist and must be of string list type.

Use output buffer

No.

Parameters

NAME	Object name.
VALUE	Object entry value.

INDEX Index of the entry. This index starts at 1. If the INDEX parameter is not provided or is blank, Nirva just adds the entry value to the end of the list. This feature can be used to add a new entry to the string list instead of using the INSERT command.

SEPARATOR Multi string separator. This parameter has meaning only when the INDEX parameter is not given or is blank. It defines a character or string separator in the given value. Then NIRVA will add as many strings as the number of separators found (plus one). For example, if the value is “`v1;v2;v3`” and the separator is “`;`”, NIRVA will add the strings “`v1`”, “`v2`” and “`v3`” to the string list object.
The SEPARATOR parameter may also take special values “NVLINE”, “NVTAB” or “NVSPACE” for respectively line (or pair CR/LF) separator, tabulation separator or space separator. In the case of a space separator, several successive spaces are assimilated to a single separator.

STRINGLIST_SORT

Description

This command sorts the entries of a string list object. The object must exist and must be of string list type.

The sort is in ascending order and can be numeric if the NUMERIC option has been chosen.

Use output buffer

No.

Parameters

NAME Object name.

NUMERIC Numeric option. If this parameter is set to “YES”, Nirva considers that the string list contains numeric value and tries to sort them accordingly.

NATURAL Natural option. If this parameter is set to “YES”, the command sorts items in a “natural” way. This is useful to list items containing both alphabetic characters and numbers. This option has no effect when NUMERIC is set to “YES”.

LOCALE Locale name. Internally the command uses unicode to sort strings so the sorting should be ok. However, if the sorting is not correct in your language you can set the locale name using this parameter. Ex: “FRENCH”.

STRINGLIST_SWAP

Description

This command swaps 2 entries of a string list object. The object must exist and must be of string list type.

Use output buffer

No.

Parameters

NAME	Object name.
INDEX1	Index of the first entry to swap. This index starts at 1. This parameter is mandatory.
INDEX2	Index of the second entry to swap. This index starts at 1. This parameter is mandatory.

INDSTRINGLIST_ADD_INDSTRINGLIST

Description

This command adds all entries of an indexed string list object to another indexed string list object. Both objects must exist and must be of indexed string list type.

The destination entries having the same key than the source entries are replaced by the source entries.

Use output buffer

No.

Parameters

SNAME	Source object name.
NAME	Destination object name. All the source object entries are added to the destination object.

INDSTRINGLIST_GET_KEY

Description

This command returns a KEY corresponding to an index. It allows enumerating the keys of the indexed string list object.

Use output buffer

Yes.

Parameters

NAME	Object name.
INDEX	Key index. The index starts at 1 and cannot be greater that the number of entries of the indexed string list object.

INDSTRINGLIST_GET_SIZE

Description

This command returns the total number of entries of an indexed string list object in the output buffer. The object must exist and must be of indexed string list type.

Use output buffer

Yes.

Parameters

NAME	Object name.

INDSTRINGLIST_GET_VALUE

Description

This command returns the value of an indexed string list object entry in the output buffer. The object must exist and must be of indexed string list type.

Use output buffer

Yes.

Parameters

NAME	Object name.
KEY	Entry key. This is a string that indexes the entry. The KEY can contain space characters.
INDEX	Key index. This parameter can be given to directly access the value by index instead of the key. The index starts at 1 and cannot be greater that the

number of entries of the indexed string list object.
However, accessing an indexed string list by the way of a numeric index is slower than accessing by key.

INDSTRINGLIST_GET_VALUES

Description

This command writes the complete content of an indexed string list object into the output buffer or in session variables.

Use output buffer

Yes.

Parameters

NAME	Object name.
MODE	Can be set to VARIABLES (default) or STRING. If set to VARIABLES, the command creates as many variables as the number of keys. The name of the variables is the name of the key eventually prefixed with a value given in PREFIX parameter. If the MODE is set to STRING, the command returns the content in the output buffer using KEYSEP and VALSEP parameters respectively as key and value separators.
KEYSEP	Key separator. This can be any string. The default is “;”. Used only if the mode has been set to STRING.
VALSEP	Value separator. This can be any string. The default is “=”. Used only if the mode has been set to STRING.
PREFIX	Prefix of created variables when the mode is VARIABLES. Default is no prefix.

INDSTRINGLIST_KEY_EXIST

Description

This command returns “YES” in the output buffer if the given key exists in the indexed string list object and “NO” otherwise.

Use output buffer

Yes.

Parameters

NAME	Object name.
KEY	Entry key. This is a string that indexes the entry. The KEY can contain space characters.

INDSTRINGLIST_REMOVE

Description

This command removes one or all entries of an indexed string list object. The object must exist and must be of indexed string list type.

Use output buffer

No.

Parameters

NAME	Object name.
KEY	Entry key. This is a string that indexes the entry. The KEY can contain space characters. If this parameter is not provided or is blank, Nirva removes all the entries of the indexed string list object.

INDSTRINGLIST_SET_VALUE

Description

This command sets the value of an entry of an indexed string list. The list entry is indexed by a string key. The object must exist and must be of indexed string list type.

If the entry already exists, Nirva sets its new value. If the entry doesn't exist, Nirva adds it to the object.

The command also allows setting several keys from a single string. For that, the KEY parameter must me empty and the VALUE parameter must contain key and value separators. For example, the value "`key1=val1;key2=val2`" will add the keys "`key1`" and "`key2`" to the object with respective values of "`val1`" and "`val2`". The default key and value separators (';' and '=' characters) can be changed by the command.

Use output buffer

No.

Parameters

NAME	Object name.
VALUE	Object entry value.
KEY	Entry key. This is a string that indexes the entry. The KEY can contain space characters.
KEYSEP	Key separator. This parameter has meaning only when the KEY parameter is empty. It defines the key separator in the given VALUE parameter. It can be any string. The default is the ';' character (semicolon).
VALSEP	Value separator. This parameter has meaning only when the KEY parameter is empty. It defines the value separator in the given VALUE parameter. It can be any string. The default is the '=' character (semicolon).

TABLE_ADD_COLUMNS

Description

This command adds several columns to a table object. The columns are inserted after the last column.

Use output buffer

No.

Parameters

NAME	Table object name.
COLUMNS	Name and descriptions of the columns to add. The names must be separated by a semicolon character (";"). Each value is composed of 3 parts separated by a ":" character: NAME:DESCRIPTION:NUMERIC. NAME is the column name, it cannot contain any blank or special character (they will be removed without error if there are), DESCRIPTION is a free text describing the column and NUMERIC, when set to "Y" tells that the column is numeric. The DESCRIPTION and NUMERIC part are optional. Here are some valid entries: "COL1;COL2;COL3" or "COL1;COL2:My column 2:Y;COL3:My column 3".

TABLE_ADD_ROWS

Description

This command adds several rows with data to a table object.

If there is a primary key defined for the table and there are some duplicates on the primary key, the command fails.

Use output buffer

No.

Parameters

NAME	Object name.
DATA	Row data. The row separator is the line feed character (\n), the column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the ROWSEP, COLSEP and LINESEP parameters. The column data must be in the same order than the column definition in the table. The last columns can be omitted if they are blank. If a cell contains an empty line, Nirva does not write this empty line. In order to force writing an empty line, the line value must be set to "NV_EMPTY".
ROWSEP	Row separator. This defines the string that separates the rows in the data. The default is the line feed character (\n).
COLSEP	Column separator. This defines the string that separates the columns in the data. The default is ";".
LINESEP	Line separator. This defines the string that separates the lines for columns containing several lines in the data. The default is "|".

TABLE_ADD_TABLE

Description

This command adds all rows of a table object to another table object. Both objects must exist and must have the same number of columns.

3 modes are available: Normal, Replace, No replace. These modes changes the way the command manages duplicates when the destination table has a primary column. In normal mode, any duplicate generates an error. In Replace mode, the new row having a primary key that already exists replaces the old row. In no replace mode, the new row having a primary key that already exists is skipped.

Use output buffer

No.

Parameters

SNAME	Source object name.
NAME	Destination object name. All the source object rows are added to the destination object.
MODE	Controls the way the command manages the duplicate when a primary key has been defined (see description). The possible values are: NORMAL, REPLACE and NO_REPLACE. The default is NORMAL.

TABLE_CLEAR

Description

This command clears the entire table including its column definition, description, and rows. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.

TABLE_CLEAR_CELL

Description

This command clears a cell of a table object.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_CLEAR_COLUMN

Description

This command clears a single column from a table object. Clearing a column means to remove any data from the column so the column still exists but is empty.

Use output buffer

No.

Parameters

NAME	Object name.
COL	Column index to clear. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_CLEAR_DATA

Description

This command removes all table rows. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.

TABLE_CLEAR_ROW

Description

This command clears a single row from a table object. Clearing a row means to remove any data from the row so the row still exists but is empty.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

TABLE_CLEAR_ROWS

Description

This command clears a selection of rows from a table object. Clearing a row means to remove any data from the row so the row still exists but is empty.

The selection is made by giving a query as parameter.

Use output buffer

No.

Parameters

NAME	Object name.
QUERY	Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query. The default value is “*” (search all rows).

TABLE_CREATE_FROM

Description

This command clears a table object and re-creates it with the column information of a source table object. Both objects must exist and must be of table type.

This command doesn't copy rows from the source object. For that, one can use the COPY command of the OBJECT class.

Use output buffer

No.

Parameters

SNAME	Source object name.
NAME	Destination object name. The destination object is a table. The command clears this table and initializes it with the table and column information of the source object. The destination object has 0 rows after the completion of this command.

TABLE_EXPORT

Description

This command exports data rows of a table object to an ASCII file.

Use output buffer

No.

Parameters

NAME	Object name.
FILENAME	Pathname of the file to export rows to. Here is the description of the file format: Each line corresponds to 1 row. The column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the COLSEP and LINESEP parameters. Each line is limited to 1024 characters (may be changed by the LINE_LENGTH parameter). If a row is longer than 1024 characters, the last line character is '\' (backslash) and the row content continues on the next line. The column data has the same order than the column definition in the

	table. The backslash and line feed characters are controlled by the way of WITH_LF parameter.
COLSEP	Column separator. This defines the string that separates the columns in the export file. The default is ";".
LINESEP	Line separator. This defines the string that separates the lines for columns containing several lines in the export file. The default is "|".
LINE_LENGTH	This is the maximum line length of the export file. The default is 1024. The minimum value is 1024.
WITH_LF	When this parameter is set to YES, Nirva writes any line feed character as the string "\n" and any backslash characters as the string "\\". When this parameter is set to NO, Nirva just replaces any line feed character with a space. In any case, the carriage return characters are removed. The default is NO.
WITH_BOM	When this parameter is set to "YES", Nirva writes the utf8 BOM at the start of the export file. The value can also be set to "AUTO". At this time Nirva writes the BOM only if the server has been started in unicode mode. The default is NO.

TABLE_FILTER_COLUMNS

Description

This command allows keeping only specified columns of a table. The other ones are removed. The name and order of the columns can be changed.

Use output buffer

No.

Parameters

NAME	Object name.
COLUMNS	Columns to keep. When not provided, the command has no effect. The COLUMNS parameter is a list of values separated by a semicolon character. Each value itself has the format *NewColName*:*ColName* where *ColName* is the name of the column and *NewColName* is its new name. If the column name doesn't have to be changed, the value is simply *ColName*. The columns will be delivered in given order.

TABLE_GET_CELL

Description

This command returns the complete cell data in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINESEP	Line separator. This can be a string (not only a single character). The default is “;”.

TABLE_GET_CELL_LINE

Description

This command returns a cell line data in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. The default is 1 (First line).
DEFAULT	Default value to return if the line index is not valid.

TABLE_GET_CELL_NUM_LINES

Description

This command returns the number of lines of a cell in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_GET_COLUMN

Description

This command gets the cells values for a given column and all rows.

Use output buffer

Yes.

Parameters

NAME	Object name.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
ROWSEP	Row separator. This can be a string (not only a single character). The default is “;”.
LINESEP	Line separator. This can be a string (not only a single character) The default is “|”.

TABLE_GET_COLUMN_DESCRIPTION

Description

This command returns the column description corresponding to the given column index in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.
COL	Column index. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_GET_COLUMN_INDEX

Description

This command returns the column index given a column name. The result is written in the output buffer.

Use output buffer

Yes.

Parameters

NAME Object name.

COLNAME Column name.

TABLE_GET_COLUMN_NAME

Description

This command returns the column name corresponding to the given column index in the output buffer.

Use output buffer

Yes.

Parameters

NAME Object name.

COL Column index. The index starts at 1.

TABLE_GET_COLUMN_NAMES

Description

This command returns the column names. All values are semicolon separated (;).

Use output buffer

Yes.

Parameters

NAME Object name.

TABLE_GET_DESCRIPTION

Description

This command returns the description of a table object in the output buffer. The object must exist and must be of table type.

Use output buffer

Yes.

Parameters

NAME	Object name.

TABLE_GET_NUM_COLUMNS

Description

This command returns the number of columns of a table object in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.

TABLE_GET_NUM_ROWS

Description

This command returns the number of rows of a table object in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.

TABLE_GET_PRIMARY_INFO

Description

This command returns information about the primary column of a table object.

Use output buffer

No.

Parameters

NAME	Object name.

Objects created

PRIMARY_COLUMN	This is a string object that contains the name of the primary column. It's empty if there is no primary column.
PRIMARY_COLUMN_CS	This is a boolean object set to TRUE if the primary column is case sensitive and to FALSE otherwise.

TABLE_GET_ROW

Description

This command extracts a single record from the table object and writes it into the output buffer or in session variables.

Use output buffer

Yes.

Parameters

NAME	Object name.
ROW	Index of the first row to get. The index starts at 1.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
MODE	Can be set to VARIABLES (default), STRING or INDSTRINGLIST. If set to VARIABLES, the command creates as many variables as the number of columns. The name of the variables is the name of the column eventually prefixed with a value given in PREFIX parameter. If the MODE is set to STRING, the command returns the row content in the output buffer. If the MODE is INDSTRINGLIST the command creates an indexed string list object and writes a value for each column name.
COLSEP	Column separator. The default is ";". Used only if the mode has been set to STRING.
LINESEP	Line separator. The default is "|" if mode is STRING or INDSTRINGLIST and ";" if mode is VARIABLES.

PREFIX Prefix of created variables when the mode is VARIABLES. Default is no prefix.

RESULT Name of the resulting object when the mode is INDSTRINGLIST. The resulting object is an indexed string list object containing the requested row. The default is “RESULT”.

Objects created

RESULT This is a Nirva indexed string list object that contains the requested row. The name of this object can be changed by using the RESULT parameter. This object is created only when the mode has been set to INDSTRINGLIST.

TABLE_GET_ROWS

Description

This command extracts some records from the table object and writes them to a new table or string object. The rows can be searched by a FIRST and LAST pair parameters or by a PAGE and PAGE_SIZE pair.

Use output buffer

Yes. It contains the number of found rows.

Parameters

NAME Object name.

MODE Output mode can be “TABLE” (default) or “STRING”. The command will create a corresponding output object type.

RESULT Name of the resulting object. The resulting object is a table or string object containing the requested rows. The default is “RESULT”.

FIRST Index of the first row to get. The index starts at 1. Default is 1.

LAST Index of the last row to get. The index starts at 1. A value of 0 means last page. Default is 0.

PAGE Page number to get. First page is page 1. If the PAGE parameter is given the FIRST and LAST parameters are not taken in care.

PAGE_SIZE Page size. Used only when the PAGE parameter has been given. If the page size is 0, no records are returned. The default page size is 10.

COLUMNS This parameter allows selecting only some of the columns. When not provided, all the columns are retrieved. The COLUMNS parameter is a list of values separated by a semicolon character. Each value itself has the format *NewColName*:*ColName* where *ColName* is the name of the column and *NewColName* is its new name. If the column name doesn’t have to be

	changed, the value is simply *ColName*. The columns will be delivered in given order.
ROWSEP	Row separator when the mode is “STRING”. This defines the string that separates the rows in the data. The default is the line feed character (\n).
COLSEP	Column separator when the mode is “STRING”. This defines the string that separates the columns in the data. The default is “;”.
LINESEP	Line separator when the mode is “STRING”. This defines the string that separates the lines for columns containing several lines in the data. The default is “|”.

Objects created

RESULT	This is a Nirva table or string object that contains the requested rows. The name of this object can be changed by using the RESULT parameter. When the mode is “TABLE”, the column definition of the resulting table is similar to the one of the searched table.

TABLE_GET_ROW_INDEX

Description

This command returns the row index given a primary key value. If there is no primary key defined for the table or if the given key is not found, the command returns 0.

The found row index is written into the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.
PRIMARY	Primary key value to find.

TABLE_IMPORT

Description

This command imports data rows from an ASCII file to a table object. The file can be the result of a TABLE_EXPORT command.

If there is a primary key defined for the table and if the command found some data in the file with already existing primary key, the record is skipped (not imported).

If the primary key value to import contains several lines, only the first line is imported.

Use output buffer

No.

Parameters

NAME	Object name.
FILENAME	Pathname of the file from which to import rows. Here is the description of the file format: Each line corresponds to 1 row. The column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the COLSEP and LINESEP parameters. Each line is limited to 1024 characters (may be changed by the LINE_LENGTH parameter). If a row is longer than 1024 characters, the last line character is '\' (backslash) and the row content continues on the next line. The column data must be in the same order than the column definition in the table. The last columns can be omitted if they are blank. The backslash and line feed characters are controlled by the way of WITH_LF parameter. If a cell contains an empty line, Nirva does not write this empty line. In order to force writing an empty line, the line value must be set to "NV_EM.PTY"
APPEND	Append option. If this parameter is set to "YES", Nirva appends the rows to the table. Otherwise, the imported rows replace the current rows. The default is "NO".
COLSEP	Column separator. This defines the string that separates the columns in the import file. The default is ";".
LINESEP	Line separator. This defines the string that separates the lines for columns containing several lines in the import file. The default is "|".
LINE_LENGTH	This is the maximum line length of the import file. The default is 1024. The minimum value is 1024.

OFFSET	This allows skipping the given number of bytes before importing the file to the table. This can be useful if the import file has some specific header. The default is 0.
SKIPLINES	This allows skipping the given number of lines before importing the file to the table. This can be useful if the import file has some specific header. If both the OFFSET and SKIPLINES parameters are provided, the OFFSET parameter is evaluated before the SKIPLINES parameters so the number of lines to skip will start from the given offset. The default is 0.
WITH_LF	This tells Nirva if the input data contains line feeds in cell content. If set to YES, any line feed in the table data to import must be written as the "\n" string. Any real backslash character must be written as the "\\" string. In any case, the carriage return characters are removed. The default is NO.

TABLE_INSERT_CELL_LINE

Description

This command inserts a new cell line.

The command fails when attempting to add a cell line to a primary key that is not empty.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. The new line is inserted before the given line index. If the LINE_INDEX is equal to "0", the new line is added at the end of the cell. The default is "0".

LINE	Line data.

TABLE_INSERT_COLUMN

Description

This command inserts a new column to a table object.

The insertion occurs before the given column index.

Use output buffer

No.

Parameters

NAME	Object name.
COLNAME	Column name. This parameter is mandatory and should not be blank. The column name cannot contain spaces.
DESCRIPTION	Column description.
NUMERIC	Numeric option. If this parameter is set to “YES”, the column is considered as numeric. The default is “NO”.
COL	Column index. The index starts at 1. The insertion occurs before the given column index. For inserting a new column at the end of the table, the COL parameter must be set to “0”. This is the default.
PRIMARY	Primary column. If PRIMARY is set to “YES”, the column becomes the primary column. The default is “NO”.
PRIMARY_CS	Primary column case sensitive. This parameter has meaning only when the PRIMARY parameter has been set to “YES”. If PRIMARY_CS is set to “YES”, the primary key will be case sensitive. The default is “NO” (case insensitive).

TABLE_INSERT_ROWS

Description

This command inserts a given number of empty rows to a table object.

The insertion occurs before the given row index.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index. The index starts at 1. The insertion occurs before the given row index. For inserting new rows at the end of the table, the ROW parameter must be set to “0”. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
NUMROW	Number of rows to add. The default is “1”.

TABLE_IS_COLUMN_NUMERIC

Description

This command tells if a column is numeric or not. The result is written in the output buffer.

If the Column is numeric, the returned value is “YES” and “NO” otherwise.

Use output buffer

Yes.

Parameters

NAME	Object name.
COL	Column index. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_JOIN

Description

This command does a join of 2 tables following a foreign key. All or just a part of the source columns can be added to the object.

The link between the 2 tables is the name of the column containing the foreign key.

Use output buffer

No.

Parameters

NAME	Object name.
SNAME	Source table object name. This is the name of the object to get columns from and to add to the current object.
FOREIGN	Foreign key name. This parameter is mandatory and has the format *src_col:dst_col* where *src_col* is the name of the column containing the foreign key in the source table and *dst_col* is the name of the column containing the foreign key in the destination table (the current object). If both source and destination column have the same name, this unique name can be given directly without any ':' separator.
COLUMNS	Name of the columns from the source object to add to the destination object (the current object). This parameter, if provided, consists of several column pairs separated by a semicolon character (';'). Each column pair has the format *src_col:dst_col* where *src_col* is the name of the column of the source object to add and *dst_col* is the name of the column in the destination object. If both source and destination column have the same name, this unique name can be given directly without any ':' separator. If this parameter is not provided, Nirva adds all the columns of the source object to the destination object.
REPLACE	Replace option. This parameter controls if Nirva must replace the destination columns having the same name than the source column or not. The possible values are "YES" or "NO". The default is "NO".

TABLE_LOAD

Description

This command loads the table object from a file. The file must be the result of a TABLE_SAVE command.

Use output buffer

No.

Parameters

NAME	Object name.
FILENAME	Pathname of the file from which to load the table.

TABLE_MODIFY_COLUMN

Description

This command allows setting the value of a table object column for a selection of rows.

The selection is made by giving a query as parameter.

The command fails if the column to modify is the primary column.

Use output buffer

No.

Parameters

NAME	Object name.
COL	Column index to remove. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
VALUE	New column value. The default is blank.
QUERY	Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query. The default value is “*” (search all rows).

TABLE_REMOVE_CELL_LINE

Description

This command removes cell line.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. The default is “1” (first line).

TABLE_REMOVE_COLUMN

Description

This command removes a single column from a table object.

Use output buffer

No.

Parameters

NAME	Object name.
COL	Column index to remove. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_REMOVE_ROW

Description

This command removes a single row from a table object.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

TABLE_REMOVE_ROWS

Description

This command removes a selection of rows from a table object. The selection is made by giving a query as parameter.

Use output buffer

No.

Parameters

NAME	Object name.
QUERY	Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query. The default value is “*” (search all rows).

TABLE_SAVE

Description

This command saves permanently the table object into the given file. The save format is proprietary.

Use output buffer

No. Parameters

NAME	Object name.
FILENAME	Pathname of the file for saving the table.

TABLE_SEARCH

Description

This command searches rows in the table object following a given query string. If the search is successful, the command creates a new table object that contains the resulting rows. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.
RESULT	Name of the resulting object. The resulting object is a table object containing the requested rows. The default is “RESULT”.
QUERY	Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query. An empty cell can be searched by setting a value of “NV_CELL_EMPTY_VALUE” (for example MYCOLUMN= NV_CELL_EMPTY_VALUE). The default value is “*” (search all rows).

SORT	Sort column. Name of the sort column. The default is to not sort. The sort is in ascending order but can be in descending order if the sort column name is preceded by the '-' (minus) character.
MAXDOCS	Maximum number of rows to get. This parameter can limit the number of found rows. Nirva stops the search process when the number of requested rows corresponding to the search criteria is reached. The default is "-1" (all found rows).
COLUMNS	This parameter allows selecting only some of the columns. When not provided, all the columns are retrieved. The COLUMNS parameter is a list of values separated by a semicolon character. Each value itself has the format *NewColName*:*ColName* where *ColName* is the name of the column and *NewColName* is its new name. If the column name doesn't have to be changed, the value is simply *ColName*. The columns will be delivered in given order.
WITH_ROW_INDEX	If this parameter is set to "YES", the command adds a column named "NV_SRC_ROW_INDEX" in the result table. This column contains the row index of the source table for the found records. If this column already exists, the command replaces its content. The NV_SRC_ROW_INDEX column cannot be removed from display when using the COLUMNS parameter but its name and its order can be changed as other columns. For example, in order to display only this column by renaming it INDEX, the COLUMNS parameter must be set to "INDEX:NV_SRC_ROW_INDEX".

Objects created

RESULT	This is a Nirva table object that contains the found rows. The name of this object can be changed by using the RESULT parameter. The column definition of the resulting table is similar to the one of the searched table.

TABLE_SELECT_FROM

Description

This command allows marking some of the table rows.

For each row, it searches in a given column if this column contains the values found in a given column of another table object.

Use output buffer

No.

Parameters

NAME	Object name. This must be a table object.
SOURCE	Source object name. This must be a table object. The command will use the source object to get the values to search.
SOURCE_COL	Index of the column of the source object from which to get the values to search for.
SOURCE_COLNAME	The column name can be given instead of the column index (SOURCE_COL parameter).
COL	Index of the column where the command will search the values.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
SELECTED_COL	Index of the column in which the command will write the result of the search. This cannot be a primary key.
SELECTED_COLNAME	The column name can be given instead of the column index (SELECTED_COL parameter).
SELECTED_YES	Value to set in the SELECTED_COLUMN if one of the values has been found in the record. The default is “YES”.
SELECTED_NO	Value to set in the SELECTED_COLUMN if the values have not been found in the record. The default is “NO”.
CASE_SENSITIVE	If set to “NO”, the search will not be case sensitive. The default is “YES”.

TABLE_SET_CELL

Description

This command sets the value of an existing cell.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINESEP	Line separator. This can be a string (not only a single character) The default is “;”.
DATA	Cell data.

TABLE_SET_CELL_LINE

Description

This command sets the value of an existing cell line.

The command fails when attempting to set the primary key cell line when the line index is greater than 1.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. The default is “1” (first line). If the line index is 0 or is greater than the number of lines of the cell, a new line is added at the end of the cell.
LINE	Line data.

TABLE_SET_COLUMN

Description

This command sets the cells values for a given column and all rows. If the requested column is a primary column and the column data contains duplicate values, the command fails.

Use output buffer

No.

Parameters

NAME	Object name.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
ROWSEP	Row separator. This can be a string (not only a single character) The default is “;”.
LINESEP	Line separator. This can be a string (not only a single character) The default is “|”.
DATA	Column data. Use column and line separators defined in LINESEP and COLSEP. If the data doesn’t contain values for all rows, remaining rows will be cleared (no value).

TABLE_SET_COLUMN_DESCRIPTION

Description

This command changes a table column description. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.
COL	Column index. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

DESCRIPTION	New table column description. The default is a blank string.

TABLE_SET_COLUMN_NAME

Description

This command changes a table column name. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.
COL	Column index. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
NEW_NAME	New table column name. If the new column name already exists, the command doesn't fail but does nothing.

TABLE_SET_COLUMN_TYPE

Description

This command changes a table column type. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.
COL	Column index. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
TYPE	New table column type. This can be “NUMERIC” or “ALPHANUMERIC” (default).

TABLE_SET_DESCRIPTION

Description

This command sets the description of a table object. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.
DESCRIPTION	New table description. The default is a blank string.

TABLE_SET_PRIMARY_COLUMN

Description

This command allows setting or changing the primary column of the table. The primary column must contain a unique string that identifies it. If there are some duplicates, the command fails.

The command also fails if the primary key column of a record contains more that one cell line.

If the primary column cell data is empty, the command doesn't fail but the row containing the empty column cannot be retrieved by is primary key.

A table object can have only one primary key, so if there was another primary key before sending this command, it's removed.

Use output buffer

No.

Parameters

NAME	Object name.
COLNAME	Column name. If this parameter is blank or is not given, Nirva considers that there is no more primary column.
PRIMARY_CS	Primary column case sensitive. If PRIMARY_CS is set to “YES”, the primary key will be case sensitive. The default is “NO” (case insensitive).

TABLE_SET_ROW

Description

This command writes a complete row data.

If there is a primary key defined for the table and if the entered primary key already exists, the command fails.

If the primary key value to import contains several lines, only the first line is imported.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
DATA	Row data. The column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the COLSEP and LINESEP parameters. The column data must be in the same order than the column definition in the table. The last columns can be omitted if they are blank. If a cell contains an empty line, Nirva does not write this empty line. In order to force writing an empty line, the line value must be set to “NV_EMPTY”.
COLSEP	Column separator. This defines the string that separates the columns in the data. The default is “;”.
LINESEP	Line separator. This defines the string that separates the lines for columns containing several lines in the data. The default is “|”.

TABLE_SORT

Description

This command sorts the rows of the table object. The object must exist and must be of table type.

Use output buffer

No.

Parameters

NAME	Object name.
SORT	Sort column. Name of the sort column. The sort is in ascending order but can be in descending order if the sort column name is preceded by the '-' (minus) character.
NATURAL	Natural option. If this parameter is set to "YES", the command sorts items in a "natural" way. This is useful to list items containing both alphabetic characters and numbers. This option has no effect when NUMERIC is set to "YES".
LOCALE	Locale name. Internally the command uses unicode to sort strings so the sorting should be ok. However, if the sorting is not correct in your language you can set the locale name using this parameter. Ex: "FRENCH".

TABLE_SORT_CELL

Description

This command sorts a cell of a table object. A table object cell may contain several lines of numeric or alphanumeric value. This command sorts these lines.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
ASCENDING	If this parameter is set to "YES", the sorting occurs in ascending order. If set to "NO" ", the sorting occurs in descending order.

	The default is “YES”.
NATURAL	Natural option. If this parameter is set to “YES”, the command sorts items in a “natural” way. This is useful to list items containing both alphabetic characters and numbers.
LOCALE	Locale name. Internally the command uses unicode to sort strings so the sorting should be ok. However, if the sorting is not correct in your language you can set the locale name using this parameter. Ex: “FRENCH”.

TABLE_SWAP_CELL_LINES

Description

This command swaps 2 cell lines.

Use output buffer

No.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX1	Index of the first cell line to swap. The index starts at 1.
LINE_INDEX2	Index of the second cell line to swap. The index starts at 1.

TABLE_SWAP_COLUMNS

Description

This command swaps 2 columns.

Use output buffer

No.

Parameters

NAME	Object name.
COL1	Index of the first column to swap. The index starts at 1.
COL2	Index of the second column to swap. The index starts at 1.

TABLE_SWAP_ROWS

Description

This command swaps 2 rows.

Use output buffer

No.

Parameters

NAME	Object name.
ROW1	Index of the first row to swap. The index starts at 1.
ROW2	Index of the second row to swap. The index starts at 1.

FILE_APPEND

Description

This command appends the content of a source file (or a part of it) to a destination file.

Use output buffer

No.

Parameters

NAME	Object name. This is the destination object (appended file).
SNAME	Object source name.

OFFSET	Offset. This parameter gives the offset in bytes of the source object from which append will start. The default value is ”0” (beginning of file).
NUM_BYTES	Number of bytes to copy. This parameter gives the number of bytes of the source object to append. The default value is ”-1” (until the end of file).

FILE_CLEAR

Description

This command physically clears the file associated with the file object if it exists.

Clearing a file means setting its size to 0 by removing all its content.

Use output buffer

No.

Parameters

NAME	Object name.

FILE_COMPRESS

Description

This command compresses the content of a source file to a destination file. The compression is for internal use. Compressed files can be only decompressed using the FILE_DECOMPRESS command.

Use output buffer

No.

Parameters

NAME	Object name. This is the file object to compress.
DNAME	Object destination name (compressed file).
MODE	Compression mode. Can be “HUFF” for huffman compression or “LZMA” for lzma compression. The default is “HUFF”.

FILE_CREATE

Description

This command physically creates the file associated with the file object if it doesn’t exist.

Generally, if the file object is not persistent, the file has been created at object creation time so this command will be useful only for persistent files.

Use output buffer

No.

Parameters

NAME	Object name.

FILE_DECOMPRESS

Description

This command decompresses the content of a source file to a destination file. The file must have been compressed using the FILE_COMPRESS command.

Use output buffer

No.

Parameters

NAME	Object name. This is the file object to decompress.
DNAME	Object destination name (decompressed file).

FILE_EXIST

Description

This command tells if the file associated with the file object physically exists or not.

The result is written in the output buffer. The return value is “YES” if the file exists and “NO” otherwise.

Use output buffer

Yes.

Parameters

NAME Object name.

FILE_GET_DIRNAME

Description

This command returns the directory name of the file associated with the file object.

The result is in the output buffer.

Use output buffer

Yes.

Parameters

NAME Object name.

FILE_GET_EXTENSION

Description

This command returns the extension of the file associated with the file object.

The result is in the output buffer.

The '.' (point) character is not returned.

Use output buffer

Yes.

Parameters

NAME Object name.

FILE_GET_FILENAME

Description

This command returns the name of the file associated with the file object.

The result is in the output buffer.

Only the file name is returned without the complete path. For the complete file path, one can use the FILE_GET_PATHNAME command.

Use output buffer

Yes.

Parameters

NAME	Object name.
WITH_EXT	If this parameter is set to "NO, the command doesn't return the extension of the file. The default is "YES".

FILE_GET_PATHNAME

Description

This command returns the complete path name of the file associated with the file object.

The result is in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.

FILE_GET_SIZE

Description

This command returns the size in bytes of the file associated with the file object.

The result is in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.

FILE_REMOVE

Description

This command removes the file associated with the file object.

By default, a persistent file cannot be removed by this function. In order to remove a persistent file, the FORCE parameter must be set to “YES”.

If the file is a temporary file, Nirva removes it immediately.

If the file is a cached file, Nirva takes it under the control of the cache mechanism to be deleted when it expires.

Use output buffer

No.

Parameters

NAME	Object name.
FORCE	If this parameter is set to “YES” and the file is persistent, Nirva will remove it. This is the only possibility to remove a persistent file.

FILE_SET_ASCII

Description

This command compares the file object with the local platform (WINDOWS or UNIX) and adjusts the carriage return / line feed pairs if necessary. The maximum line length must be 65530 bytes.

Use output buffer

No.

Parameters

NAME	Object name.

FILE_SET_FILENAME

Description

This command changes the file associated with a file object. If the file object is not persistent, the previous associated file is removed (and goes under the control of Nirva cache mechanism if this is a cached object).

Use output buffer

No.

Parameters

NAME	Object name.
FILENAME	This parameter gives the new file name to associate to the file object. If this parameter is not provided, Nirva creates itself a file name following information of the EXTENSION, PREFIX, SUFFIX and DIRECTORY parameters.
DIRECTORY	This is the name of a directory where Nirva will put the local file if the received object is a file object and the FILENAME parameter is not given. For information, Nirva generates an error message when trying to create a persistent file in the application work directory. If this parameter is not provided or is blank, Nirva uses the application work directory for temporary and cached files and the application file directory for persistent files.
EXTENSION	This is the file extension to use. The EXTENSION parameter is only used if the object to create is a file object. If this parameter is not provided or is blank, Nirva uses “.obj” for persistent files and “.tmp” for temporary and cached files. If the point character is omitted in the EXTENSION parameter, Nirva adds it.
PREFIX	This is the file prefix to use. The PREFIX parameter is only used if the object to create is a file object. The prefix, if provided is used by Nirva to automatically create the server file name.
SUFFIX	This is the file suffix to use. The SUFFIX parameter is only used if the object to create is a file object. The suffix, if provided is used by Nirva to automatically create the server file name.

FILE_SET_PERSIST

Description

This command changes the kind of file object to temporary, or persistent.

On the client side, cached files are not authorized.

Use output buffer

No.

Parameters

NAME	Object name.
PERSIST	This parameter gives the new persistent value. The parameter can take value “0” for temporary files and “-1” for persistent files. Cached files are not authorized on client side. The default value is “-1” (persistent).

BINARY_APPEND

Description

This command appends the content of a source binary object (or a part of it) to a destination binary object.

Use output buffer

No.

Parameters

NAME	Object name. This is the destination object (appended binary object).
SNAME	Object source name.
OFFSET	Offset. This parameter gives the offset in bytes of the source object from which append will start. The default value is ”0” (beginning of binary data).
NUM_BYTES	Number of bytes to copy. This parameter gives the number of bytes of the source object to append. The default value is ”-1” (until the end of binary data).

BINARY_CLEAR

Description

This command resets the binary object size to 0 freeing its data from memory.

Use output buffer

No.

Parameters

NAME	Object name.

BINARY_GET_SIZE

Description

This command returns the size in bytes of the binary object.

The result is in the output buffer.

Use output buffer

Yes.

Parameters

NAME	Object name.

BINARY_LOAD

Description

This command loads the binary object data from a file.

Use output buffer

No.

Parameters

NAME	Object name.
FILENAME	Name of the file to load.
OFFSET	Offset. This parameter gives the offset in bytes of the file from which the load will start. The default value is ”0” (beginning of file).
NUM_BYTES	Number of bytes to copy. This parameter gives the number of bytes of the file to load. The default value is ”-1” (until the end of file).

BINARY_SAVE

Description

This command saves the binary object data to a file.

Use output buffer

No.

Parameters

NAME	Object name.
FILENAME	Name of the save file.
OFFSET	Offset. This parameter gives the offset in bytes of the object from which the save will start. The default value is ”0” (beginning of binary data).
NUM_BYTES	Number of bytes to copy. This parameter gives the number of bytes of the object to save. The default value is ”-1” (until the end of binary data).

REQUEST class

The request class manages the commands sent to a Nirva server.

These commands can be sent directly one by one or queued in a command buffer controlled by the request object.

Each command of the command buffer is accessible by its index starting at 1.

ERROR_INFO

Description

This command returns one of the strings of an error object.

The result is in the output buffer.

Use output buffer

Yes.

Parameters

NV_COMMAND_INDEX	Index of the command for which to get the error information. The index starts at 1. If the NV_COMMAND_INDEX parameter is not provided or set to "0" or set to a value greater than the last command index, the command returns the global error information.
ERROR_STRING	Kind of error information to return. This must be one of the strings "CODE", "INFO", "DESCRIPTION", "SERVICE", "CLASS". The default is "CODE" that returns the error code.

GET_NUM_COMMANDS

Description

This command returns the number of commands in the command buffer. The result is in the output buffer.

Use output buffer

Yes.

Parameters

None

GET_SESSION_ID

Description

This command returns the current session ID of the request. If the session has been closed, the session ID is blank.

The result is in the output buffer.

Use output buffer

Yes.

Parameters

None

REMOVE_COMMAND

Description

This command removes one command from the command buffer.

Use output buffer

No.

Parameters

NV_COMMAND_INDEX	Index of the command to remove. The index starts at 1. If the NV_COMMAND_INDEX parameter is not provided or set to "0", the last command of the command buffer is removed.

RESET_COMMAND_BUFFER

Description

This command removes all the commands from the command buffer so the command buffer becomes empty.

Use output buffer

No.

Parameters

None

SEND_COMMANDS

Description

This command sends the command of the command buffer to the server. The commands are then executed on the server side.

The SEND_COMMANDS command then wait for the completion of the server commands.

Use output buffer

No.

Parameters

NV_STOP_ON_ERROR	Stop on error flag. If this parameter is set to “YES”, the server will stop processing the commands at the first command error. Otherwise, all commands are executed even if one of them produces an error.

XML_COMMAND

Description

This command is a client for the XML or SOAP connector. It allows to send XML or XML/SOAP data to the NIRVA server and to get back the XML result.

It can also be used for rapidly populating the input container.

This command doesn’t use the usual command mechanism of other local service command. In particular, it’s entirely independent from the command buffer.

Use output buffer

No.

Parameters

COMMAND	Command parameters to send to NIRVA. The parameter must have the usual parameter format of a CGI command. Special characters must be encoded, the parameter separator is the ‘&’ character and the name/value separator is the ‘=’ character. Generally, it’s not necessary to use this COMMAND parameter since the command parameters can be also given in the input XML data. The only exception is for the NV_XML_XSL_IN parameter, if one is needed, that must be given in the COMMAND parameter. A typical example for the COMMAND parameter should be “&NV_XML_XSL_IN=test”. Also the ‘&’ delimiter character for the first parameter must be given.
XMLIN	Input XML. This must be an existing NIRVA local object file. The associated file contains the XML or SOAP input data.
XMLOUT	Output XML. This must be an existing NIRVA local object file. The command will write the XML or SOAP output in the associated file of the XMLOUT object.
SOAP	SOAP mode. If this parameter is set to YES, NIRVA considers that the input message is SOAP.

WEBS	Webs service mode. If this parameter is set to “YES”, NIRVA considers that the input message is an SOAP message for a web service.
WEBSERVICE	Webs service name. Gives the name of the web service. This parameter has meaning only if the WEBS parameter has been set to “YES”.
OPERATION	Webs service operation name. Gives the name of the web service operation to launch. This parameter has meaning only if the WEBS parameter has been set to “YES”.
SOAPACTION	SOAP action HTTP header. This parameter has meaning only if the SOAP or WEBS parameters have been set to “YES”.

VARIABLE class

The VARIABLE class provides commands for managing request variables.

EXIST

Description

This command returns “YES” in the output buffer if the given request variable exists and “NO” otherwise.

Use output buffer

Yes.

Parameters

NAME	Request variable name. A variable name is case insensitive and can contain space characters.

GET

Description

This command gets the value of the given request variable. If the variable doesn’t exist, the command returns an empty string.

Use output buffer

Yes.

Parameters

NAME	Request variable name. A variable name is case insensitive and can contain space characters.

REMOVE

Description

This command removes the given request variable or all the request variables.

Use output buffer

No.

Parameters

NAME	Request variable name. A variable name is case insensitive and can contain space characters. If this parameter is not provided, all the request variables are removed.

SET

Description

This command sets the value of the given request variable. If the request variable doesn’t exist, the command creates it.

Use output buffer

No.

Parameters

NAME	Request variable name. A variable name is case insensitive and can contain space characters.
VALUE	Request variable value. The value can be any string.

System service reference

Overview

This chapter describes in detail all the Nirva SYSTEM service commands. The Nirva SYSTEM service is a server side internal service that provides basic functionalities.

For each command, the reference gives the command name, the sources for which the command may be used, the command description, the eventual command permissions, the parameter list and the eventual list of objects created by the command.

The parameters described in this chapter are command specific parameters. For general parameters, please refer to the Nirva command syntax chapter.

The available sources are:

- Client for all Nirva client connectors including Nirva client library (nvc).
- Web for commands from a web browser or any HTTP client. This includes XML, SOAP and Web service connectors and the MQ connector.
- Procedure for commands from a Nirva procedure.
- Service for commands from service to service.

Output buffer

Some of the SYSTEM service commands return string information. For all these commands, a parameter named “NV_VAR” gives the name of a session variable that will receive the result string. The default value for NV_VAR is “NV_RESULT”.

If these commands are called from a C++ service, the returning of the string information is done with a C function that requires a character buffer as parameter in which the command will write also the result string.

If these commands are called from a Java service, the returning of the string information is done directly into a Java string object.

If these commands are called from a Dotnet service, the returning of the string information is done directly into a Dotnet string object.

This possibility for a command to return string information is called “output buffer”.

Classes

Here are the available SYSTEM service classes:

APPLICATION	Application management.
COMMAND	Command management.
CONTAINER	Container commands.
DEBUG	Debug specific commands.
MISC	Miscellaneous commands.
FILEDB	Sqlite3 commands.
LICENSE	License management.
LOCK	Locking management.
LOG	Log commands.
MQ	MQ connector
OBJECT	Container object management and access.
PACKAGE	Installation package management.
REGISTRY	Registry access.
REQUEST	Connection to distant NIRVA servers.
SCHEDULER	Scheduler commands.
SEMAPHORE	Semaphore management.
SESSION	Session management.
SERVICE	Service management.
SYSTEM	System commands
TEST	Test commands.
TIME	Date and time management.
THREAD	Thread management.
TRANSCATION	Transaction management.
VARIABLE	Variable commands.
WEBSERVICE	Web services
XML	XML management.

Error codes

APPLICATION Class

Value	Description
101	Cannot create directory
102	Cannot remove directory
103	Cannot get directory
104	Cannot remove application
105	Cannot start application
106	Cannot stop application
107	Cannot get the list of application connected sessions
108	Cannot create application
109	Cannot create installation package
110	Cannot install application
111	Cannot get application information
112	The package file is not for the current platform
113	The application doesn’t exist
114	Bad application name

COMMAND Class

Value	Description
101	Command not available
102	Bad command parameter
103	Exception occurred in command
104	Command not available from this source
105	Bad parameter date format

CONTAINER Class

Value	Description
101	Cannot create the container
102	Cannot get the container
103	Cannot copy the container
104	Bad container index

Value	Description
105	Cannot export container
106	Cannot import container
107	Cannot move container

FILEDB Class

Value	Description
101	Cannot open database file
102	Database file is not opened
103	SQL time out
104	SQL error

LICENSE Class

Value	Description
101	Cannot load license file
102	Cannot set license key
103	Cannot remove license key
104	Cannot import license file
105	Cannot export license file
106	Bad or expired license key
107	Cannot get license information

LISTENER Class

Value	Description
101	A listener with the same name exists
102	Cannot create the listener
103	The listener must be stopped first
104	Cannot remove the listener
105	Cannot start the listener
106	Cannot stop the listener
107	Cannot set listeners parameters
108	Cannot get listener information

LOCK Class

Value	Description
101	Cannot lock
102	Cannot unlock
103	Cannot get lock list

LOG Class

Value	Description
101	Bad log name
102	Cannot create log
103	Cannot get log information
104	Error in log search
105	Error in log erase

MAIL Class

Value	Description
101	No sender defined
102	Cannot create the multipart container
103	Cannot attach the message
104	Cannot attach the file
105	Cannot connect to mail host
106	Authentication error
107	No recipient defined
108	Error sending mail

MQ Class

Value	Description
101	Bad MQ message type
102	The queue already exists
103	Cannot create the queue
104	Unknown queue
105	This operation requires the queue to be stopped
106	Cannot update queue parameters

Value	Description
107	Cannot remove queue
108	Cannot start queue
109	Cannot stop queue
110	Cannot list queue
111	The MQ connector is not running

OBJECT Class

Value	Description
101	No object name
102	No object type
103	Unknown object type
104	No object data
105	Cannot create object
106	Cannot set the object data
107	Cannot get the object
108	Cannot get this kind of object
109	Cannot get the object data
110	Cannot create temporary file for the file object
111	Cannot open file for the file object
112	The local object already exist
113	Invalid object index
114	The object doesn't exist
115	The objects are the same
116	The objects are not compatible
117	Cannot copy objects
118	Invalid string list index
119	Invalid indexed string list key
120	Error in search table
121	Error in save table
122	Error in load table
123	Error in import table
124	Error in export table
125	Invalid table cell
126	Error in set cell line
127	Error in insert cell line

Value	Description
128	Error in remove rows
129	Error in clear rows
130	Bad column name
131	Error in modify columns
132	Cannot create file for file object
133	Cannot get file size
134	Cannot clear file for file object
135	Cannot append file to file object
136	Cannot set this file object persistent
137	No file name in file object
138	Cannot append binary data
139	Cannot save binary data to file
140	Cannot load binary data from file
141	Bad object type
142	Error in select table from
143	Cannot set the primary key
144	Cannot set row data
145	Cannot join tables
146	Cannot add tables
147	Cannot get row index from primary value
148	Cannot add records to the table
149	Cannot compress file
150	Cannot decompress file
151	Cannot move the object

PACKAGE Class

Value	Description
101	Cannot create installation package
102	Cannot install package
103	The package file is not for the current platform

PRGM Class

Value	Description
101	Cannot run external program

Value	Description
102	Time out condition in external program
103	Error code returned by external program
104	Bad parameter in executable path

PROCEDURE Class

Value	Description
101	Cannot get procedure file
102	Invalid procedure name
103	Cannot open procedure file
104	Bad command parameter in procedure file
105	Error in the Perl script
106	Error in the Java method
107	Error in the Dotnet method

REGISTRY Class

Value	Description
101	Cannot get registry key
102	Cannot set registry key
103	Cannot remove registry key
104	Cannot create registry key
105	Cannot export registry key
106	Cannot import registry key
107	Cannot open user registry

REQUEST Class

Value	Description
101	Cannot get connection to NIRVA server
102	Network error
103	Bad request identifier
104	Cannot create new request

SCHEDULER Class

Value	Description
101	Exception occurred when running the task
102	Bad task name
103	The task already exist
104	Cannot create task
105	Cannot remove task
106	Cannot get task list
107	Cannot get task parameters

SECURITY Class

Value	Description
101	Bad user name
102	Cannot get the security context for this user
103	Invalid password
104	The application for checking security is not existing or is disabled
105	Security alert. You aren't allowed to perform the operation
106	The user already exist
107	Cannot add the user
108	Cannot get user list
109	Invalid user name
110	Bad password syntax
111	Cannot set the password
112	The role already exist
113	Cannot add the role
114	Cannot get role list
115	Invalid role name
116	Cannot get permission list
117	Password confirmation failed
118	Cannot remove default user
119	Cannot enable or disable user
120	Security is defined on another application and server
121	The password has expired. The user must change it.
122	The security is defined on a dedicated service
123	Cannot load user context

Value	Description
124	Cannot save user context
125	Cannot access a session not owned by the client
126	Authentication error (SSO)
127	A command has been detected in parameter or variable value

SEMAPHORE Class

Value	Description
101	Cannot lock
102	Cannot unlock
103	Cannot remove
104	Cannot get semaphore list

SEQUENCE Class

Value	Description
101	The sequence doesn't exist
102	Cannot save the sequence value to the registry

SERVICE Class

Value	Description
101	Cannot initialize service
102	Cannot get service from service list
103	The service cannot initialize session
104	The service SYSTEM class is reserved
105	Cannot stop the service (probably still in use)
106	Cannot get the list of service connected sessions
107	Cannot mount the service
108	Cannot unmount the service
109	Cannot create service skeleton
110	Cannot get service information
111	Cannot create service installation package
112	Cannot install service
113	Service is stopped

Value	Description
114	The service package file is not for the current platform
115	Bad service name
116	Unknown error in service command

SESSION Class

Value	Description
101	The session doesn't exist
102	Cannot create the session
103	Cannot login the session to the application
104	The session tries to be accessed by an unauthorized user
105	The session has been closed
106	The commands has been stopped because of a previous error
107	The session has not been initialized
108	The session cannot be closed
109	Cannot find session (probably expired)
110	Too many sessions
111	Error in the session init procedure (cannot login)
112	Cannot attach console

SOAP Class

Value	Description
101	Parsing error (not well formed SOAP message ?)

SYSTEM Class

Value	Description
101	Cannot stop server when running server as a window service
102	Cannot get system information
103	Too much sessions
104	Your license does not allow Nirva to run at this time

THREAD Class

Value	Description
101	Cannot create the session thread

TRANSACTION Class

Value	Description
101	Cannot create transaction
102	Cannot get transaction
103	Cannot save transaction parameters to disk
104	Cannot create transaction directory
105	Cannot save session context
106	Cannot restart transaction
107	Cannot load session context
108	Cannot remove transaction
109	Cannot start a transaction from another transaction
110	Cannot get transaction information
111	Cannot validate the transaction
112	Cannot rollback the transaction
113	The transaction is in process

WEBSERVICE Class

Value	Description
101	Cannot remove web service
102	Cannot start web service
103	Cannot stop web service
104	Cannot get web service information
105	Cannot create web service
106	Cannot create installation package
107	Cannot install web service
108	The web service package file is not for the current platform
109	Cannot update the web service content
110	Cannot turn the web service to edit mode
111	The web service is not in edit mode
112	Cannot enable web service operation

Value	Description
113	Cannot disable web service operation
114	Cannot create web service operation
115	Cannot remove web service operation
116	Cannot set web service operation parameters
117	Cannot set the SOAP input example for this service
118	Cannot create web service message
119	Cannot remove web service message
120	Cannot set web service message parameters
121	Cannot set web service message content
122	Cannot get web service message content
123	Cannot set the WSDL definition for this service
124	Unknown web service
125	The web service is not available
126	Cannot get the web service
127	The operation doesn't exist
128	Invalid web service name

XML Class

Value	Description
101	Cannot get object XML data
102	The XSL name is not correct
103	Cannot get the XSL file
104	No XSL processor defined
105	Invalid XSL processor command
106	Error running XSL processor
107	Cannot parse XML document
108	Bad XML input format
109	Error parsing command parameters from XML input
110	Cannot connect requested server
111	Error when sending the XML data to server
112	Invalid XML data (when using XML data validation feature)
113	Unsupported encoding

Permissions

This is the list of system permissions. Specific service or application permissions are defined at application and service levels.

For web services, NIRVA automatically creates a permission for each operation of each web service allowing controlling security at operation level for web services.

Permission	Description
STOP_SERVER	Stop NIRVA server
SET_SYSTEM_PARAM	Change general system parameters
GET_SYSTEM_PARAM	View general system parameters
REGISTRY_SYSTEM_WRITE_CLIENT	Access system or service registry in write mode from client or browser
REGISTRY_SYSTEM_READ_CLIENT	Access system or service registry in read mode from client or browser
REGISTRY_APPLICATION_WRITE_CLIENT	Access application registry in write mode from client or browser
REGISTRY_APPLICATION_READ_CLIENT	Access application registry in read mode from client or browser
APPLICATION_LIST	Get application list
APPLICATION_REMOVE	Remove application
APPLICATION_RUN	Start and stop application
APPLICATION_CREATE	Create new application
APPLICATION_CONFIG	Configure application
APPLICATION_PACKAGE	Package application
APPLICATION_INSTALL	Install application
APPLICATION_SESSION_LIST	List application sessions
WEBSERVICE_LIST	Get web service list
WEBSERVICE_REMOVE	Remove web service
WEBSERVICE_RUN	Start and stop web service
WEBSERVICE_CREATE	Create new web service
WEBSERVICE_PACKAGE	Package web service
WEBSERVICE_INSTALL	Install web service
WEBSERVICE_EDIT	Edit web service
SESSION_CLOSE_OTHER	Close another session
SERVICE_LIST	Get service list
SERVICE_RUN	Start and stop service

Permission	Description
SERVICE_MOUNT	Mount and unmount service
SERVICE_SKELETON	Create service skeleton
SERVICE_CONFIG	Configure services
SERVICE_PACKAGE	Package service
SERVICE_INSTALL	Install service
SERVICE_SESSION_LIST	List sessions using a given service
LICENSE_UPDATE	Update licenses
SYSTEM_PACKAGE	Package system
SYSTEM_INSTALL	Install system
SYSTEM_LOG_ADMIN	Administrate system logs
SYSTEM_SESSION_INFO	Get information about nirva sessions
SERVICE_LOG_ADMIN	Administrate service logs
APPLICATION_LOG_ADMIN	Administrate application logs
SYSTEM_LOG_READ	Read and list system logs (used only when SYSTEM_LOG_ADMIN is not set)
SERVICE_LOG_READ	Read and list service logs (used only when SERVICE_LOG_ADMIN is not set)
APPLICATION_LOG_READ	Read and list application logs (used only when APPLICATION_LOG_ADMIN is not set)
SYSTEM_SECURITY_ADMIN	Administrate system security
APPLICATION_SECURITY_ADMIN	Administrate application security
SECURITY_CHANGE_OWN_PASSWORD	Change own user password
SECURITY_CHANGE_OTHER_PASSWORD	Change password of other users
DISPLAY_CONFIG_SYSTEM	Display system menu in configuration tool
DISPLAY_CONFIG_APPLICATION	Display application menu in configuration tool
SCHEDULER_ADMIN	Administrate scheduler
SCHEDULER_LIST	List tasks (used only when SCHEDULER_ADMIN is not set)
SCHEDULER_RUN_OTHER_USER	Run scheduled tasks under another user account
SCHEDULER_RUN_MANUAL	manually and immediately run a scheduled task
MAIL_SEND	Send E-Mails
MQ_ADMIN	MQ connector administration
PASSWORD_GET	Can read session password from procedure or service
LISTENER_ADMIN	Administrate listeners
LISTENER_LIST	List listeners (used only when LISTENER_ADMIN is not set)

Permission	Description
DISPLAY_XML_BROWSER	Allow displaying xml output in browser when the application flag for blocking xml output in browser has been set
ACCESS_NOT_OWNED_SESSIONS	Allow clients to send commands to sessions they don't own when the application flag for blocking session access has been set
BYPASS_COMMAND_CLIENT_FILTER	When a client command filter is defined at application level, this permission allows the user to bypass the command filter
BYPASS_COMMAND_WEB_FILTER	When a web command filter is defined at application level, this permission allows the user to bypass the command filter
DISPLAY_ERROR_INFO	Allow displaying the info field of the error message in external commands (client, browser)
REST_CONNECTOR	Allow access to the rest connector

Commands

APPLICATION class

The APPLICATION class provides commands for working with applications.

CONFIG

APPLICATION:CONFIG

Source	Use Input Container	Use Output Container	Use Output buffer
Web	No	Yes	No

Description

Configures the connected application. This command calls the application configuration entry page. This page is an XSL page named “config.xsl” that should be in the application files config directory.

The command also calls an application procedure named “config.nvp” that should be in the application procedure config directory. If this procedure doesn’t exist, the command doesn’t fail. The procedure is executed before the command.

This command can be used only from a web browser. The eventual NV_XML_XSL parameter given in the command has no meaning and is internally replaced by the name of the config xslt page.

It's not possible to configure another application than the current connected one.

Permissions

APPLICATION_CONFIG

Parameters

None

CREATE

APPLICATION:CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Creates a new application.

The create function can also be used to change the parameters of an existing application. For that, the application must be stopped with the STOP command otherwise, this command returns an error. Only the application description can be changed.

The CREATE command creates the application directory and some default files for configuration, documentation and packaging.

Here is the list of created files in the application directory if they don't exist:

application.dsc	This is the application description file that contains some information that describes the application. This file is used by NIRVA when starting the application. This file is created in the application files directory.
config.xsl	This is the xslt file used to create the application configuration WEB page when using the SYSTEM APPLICATION CONFIG command. This page should be modified by the application programmer. This file is created in the subdirectory Config of the application files directory.
config.nvp	This is an empty procedure called by the SYSTEM APPLICATION CONFIG command. This procedure should be modified by the application

	programmer if necessary. This file is created in the subdirectory Config of the application procedure directory.
index.html	Starting html page for application documentation. This page should be modified by the application programmer if necessary. This file is created in the application documentation html directory.
package.lst	This is the installation package file. This file contains information for creating a NIRVA installation package for the application.

Permissions

APPLICATION_CREATE

Parameters

APPLICATION	Name of the application to create or to change. An application name should contain only alphanumeric characters and the underscore character.
DESCRIPTION	Application description.
TIMEOUT	Application default time out.
SECUAPP	Application for checking the security. If this parameter is let blank, the security is the security of the application NVDEF itself, if it's set to "(SYSTEM)" the system level security will be used, if it's set to a valid application, the given application security will be used. When set to another application, one can define another server where the security application resides. For example `TEST[secu_server:1083]` tells that the application uses the security of the TEST application installed on the server secu_server with TCP/IP port 1083. The default port is 1081. One can also use SSL for communicating with the distant security, at this time, the same example will be: `TEST[secu_server:1083(SSL)]`.
USE_SCHEDULED_TRANSAC	Tells if the application can use scheduled transactions or not. The possible values are "YES" or "NO". The default is "YES".
ALLOW_WEB_XML	When set to "NO", the output XML to the browser is controlled by a dedicated permission in the security of the application. The default is "YES".

CREATE_DIR

APPLICATION:CREATE_DIR

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	No

Description

This command creates a directory. For security reasons, this command is available only from procedures or services.

Parameters

DIR Complete path of the directory name to create, starting from the application directory. For example DIR="Test" will create a subdirectory named "Test" in the current application directory. The command only creates the last directory. The previous ones in the given path must exist. For example if DIR is "Test/MyDir", the command assumes that the "Test" directory already exists.
The DIR parameter may point to a full directory if the RELATIVE parameter is set to "NO".

RELATIVE If this parameter is set to "NO", the directory pointed by DIR is considered to be a full directory path and not a relative application directory path. The default is "YES".

DIR

APPLICATION:DIR

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	Yes	No

Description

This command populates the output container with the structure of a given directory. The command creates a subcontainer for each subdirectory and an object file for each file.

The names of created file objects and subcontainers depend of the NAMING parameter. There 2 possible naming: standard and topic based.

In standard naming, all subdirectories are named "dirx" where x is a number starting from 1 and incremented by one for each new subcontainer. In the same way all file objects are named "filex" where x is a number starting from 1 and incremented by one for each file object.

In the topic based naming, the name of subcontainers and file objects is based on the real file and directory names. They are translated to lower case, the spaces are removed and the '.' Characters are replaced by the string "_p_".

In both naming conventions, the original name of the file is written in the “ORIGIN” member of the file object, it can be retrieved with the SYSTEM OBJECT FILE_GET_ORIGIN command. The command also creates a string object named DIR_NAME in each created subcontainer and puts into it the real subdirectory name.

The created file objects point to the real files found (and not to a copy of them).

The DIR command first clears the output container.

Parameters

DIR	Complete path of the directory name to get, starting from the application directory. For example DIR=”Test” will get a subdirectory named “Test” from the current application directory. The DIR parameter may point to a full directory if the RELATIVE parameter is set to “NO”.
RELATIVE	If this parameter is set to “NO”, the directory pointed by DIR is considered to be a full directory path and not a relative application directory path. The default is “YES”.
FILES	This parameter gives the beginning of the file names to get. For example, if FILES is “tst”, the command will get all the files starting with “tst”. If FILES is “*”, all the files of DIR directory are retrieved. If FILES is blank, no files are retrieved. The default is “*” (all the files).
SUBDIRS	This parameter gives the beginning of the subdirectory names to get. For example, if SUDIRS is “tst”, the command will get all the subdirectories starting with “tst”. If SUDIRS is “*”, all the subdirectories of DIR directory are retrieved. If SUDIRS is blank, no subdirectories are retrieved. The default is “*” (all the subdirectories).
RECURSE	This parameter has meaning only when the SUBDIRS parameter is not blank. If recurse is set to “YES”, the command will descend also in subdirectories. The default is “YES” (descend in subdirectories).
PERSIST	This parameter gives the persistent value for created file objects. By default, the file objects created with the DIR command are persistent. The parameter can take value “0” for temporary files, “-1” for persistent files (default) and any other value (in seconds) for cached files.
NAMING	This parameter gives the naming convention of created file objects and subcontainers. See description for further information about naming. This parameter can take values “STANDARD” for a standard naming or “TOPIC” for a topic based naming. The default is “TOPIC”.

GET_DIR

APPLICATION:GET_DIR

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the complete path of the application directory.

Parameters

None

GET_FILE_DIR

APPLICATION:GET_FILE _DIR

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the complete path of the application file directory. The file directory is the place where the application stores persistent files. The work directory also contains XSLT files for using with the Nirva XML capabilities.

Parameters

None

GET_NAME

APPLICATION:GET_NAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the name of the current application

Parameters

None

GET_WORK_DIR

APPLICATION:GET_WORK_DIR

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the complete path of the application work directory. The work directory is the place where the application stores temporary and cached files.

Parameters

None

GET_WROOT_DIR

APPLICATION:GET_WROOT_DIR

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the complete path of the application web root directory. The web root directory is the place where the application stores its web sites.

Parameters

None

INFO

APPLICATION:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information about one or several applications:

- Name.
- Description.
- Default time out.
- Security application.
- Scheduled transaction flag.
- Allow browser xml output flag.
- Status
- Number of connected sessions
- Allow not owned session flag
- Authentication mode
- Nirva is sso user flag

- Client filter mode
- Web filter mode
- Client command filter
- Web command filter
- External command trigger
- Required services
- Use trigger procedure

The INFO command returns information in a NIRVA table object named “INFO” in the output container.

If the parameter “APPLICATION” is given, the command just reports information on the requested application.

Permissions

APPLICATION_LIST

Parameters

APPLICATION Name of the application to get information from. If this parameter is not given or is blank, the command retrieves information about all applications.

Objects created

INFO This is a Nirva table object containing the following columns:

- “NAME” is the application name.
- “DESCRIPTION” is the application description.
- “TIMEOUT” is the application default time out.
- “SECUAPP” is the name of the application that checks the security. A blank value means that the security is the security of the application itself and a value “(SYSTEM)” means that the security is the global system security.
- “USE_SCHEDULED_TRANSAC” tells if the application can use scheduled transactions or not. The possible values are “YES” or “NO”.
- “ALLOW_WEB_XML” tells if the application output XML in browser is always authorized (YES) or controlled by security permission (NO). The possible values are “YES” or “NO”.
- “STATUS” is the application status. It has the value “RUNNING” if the application is running and “STOPPED” otherwise.
- “NUM_SESSIONS” is the number of sessions currently connected to the application.

- “START_PAGE” is the application start web page as defined in the application description file (.dsc file).
- “ALLOW_NOT_OWNED_SESSIONS” allows authorizing Nirva clients to send commands to sessions they didn’t create themselves. The possible values are “YES” or “NO”.
- “AUTH_MODE” is the application authentication mode. This can be “NIRVA”, “SSO” or “NIRVA_SSO”.
- “NIRVA_IS_SSO_USER” flag. Tells if, in case of SSO, the nirva user is the SSO user or not. The possible values are “YES” or “NO”.
- “COMMAND_CLIENT_FILTER_MODE” is the application filter mode for client commands. The possible values are “NEVER”, “ALWAYS” or “BY_SECU”.
- “COMMAND_WEB_FILTER_MODE” is the application filter mode for web commands. The possible values are “NEVER”, “ALWAYS” or “BY_SECU”.
- “COMMAND_CLIENT_FILTER” is the application filter for client commands.
- “COMMAND_WEB_FILTER” is the application filter for web commands.
- “TRIGGER” is procedure ran as trigger for external commands.
- “REQUIRED_SERVICES” is the application list of required services. This is a comma separated list of services required by the application.
- “USE_TRIGGER_PROCEDURE” is a flag telling if the application is using the trigger procedure if one has been defined in the application.dsc file. The possible values are “YES” or “NO”.

INFOEX

APPLICATION:INFOEX

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some application specific information. This is the information found in the INFO section of the application description file. The description file automatically created by the SYSTEM:APPLICATION:CREATE command in the application files directory.

The application description file has the same format than the service description file but with only one section named INFO. Please see the external service chapter for further information about the description file.

Parameters

APPLICATION	Name of the application to get information from. This parameter is mandatory.

Objects created

APPLICATION_INFO	This is a Nirva indexed string list object corresponding to the pairs found in the INFO section of the application file. If there is no description file for the given application, the SERVICE_INFO object is created but is empty.

INSTALL

APPLICATION:INSTALL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

Installs an application package file previously created by the SYSTEM APPLICATION PACKAGE command.

The application package file is a binary file that itself contains other files.

This command can only install files in the given application directory.

The application must have been previously stopped in order for the command to succeed.

If the package file has a header entry PLATFORM = *Platform* where Platform is the target platform (WIN32, WIN64, AIX, LINUX, LINUX64, HPUX, HPUXI or SOLARIS), the INSTALL command checks if the target platform corresponds to the package file and returns an error if it’s not the case. See the chapter “Installation packages” for information about package file headers.

Permissions

APPLICATION_INSTALL

Parameters

APPLICATION	Name of the application for which to install the package. The application name is case insensitive. If this parameter is not provided, the command tries to get it from the package file in the header entry "*APPLICATION = AppName*" where *AppName* is the name of the application. An application name should contain only alphanumeric characters and the underscore character.
FILE	Name of the NIRVA file object that contains the package file to install. This object must be in the input container. The default value is "PACKAGE_FILE".

PACKAGE

APPLICATION:PACKAGE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

Creates an application package file that will be usable by the SYSTEM APPLICATION INSTALL command.

The application package file is a binary file that itself contains other files. The package file is created following the content of a package description file that must reside on the application files directory. The package file is compressed.

For security reasons, the application packages can only install application files on the target application directory.

Permissions

APPLICATION_PACKAGE

Parameters

APPLICATION	Name of the application for which to create an installation package. The application name is case insensitive.
FILE	Name of the NIRVA file object that will contain the resulting package file. This object is in the output container. If the object doesn't exist, the command creates it. The default value is "PACKAGE_FILE".

DESC_FILE	Name of package description file. This must correspond to an existing file that resides in the application files directory. The default value is “package.lst”. The format of the package description file is given in the “installation packages” chapter. It’s important (but not necessary) if the description file has a header entry “*APPLICATION* = *AppName*” where *AppName* is the name of the application.

REMOVE

APPLICATION:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Removes an application from the list of NIRVA application.

This command doesn’t remove the application files but just the link of the application in the NIRVA application list. In this way, the application can be created again.

In order to completely remove an application, one must use the REMOVE command and then manually removes the application form the application directory.

This command fails if the application is running so it must be previously stopped by using the SYSTEM:APPLICATION:STOP command.

Permissions

APPLICATION_REMOVE

Parameters

APPLICATION	Name of the application to remove. The application name is case insensitive.

REMOVE_DIR

APPLICATION:REMOVE_DIR

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	No

Description

This command removes a directory of the application directory or a part of it. For security reasons, this command is available only from procedures or services.

Parameters

DIR

Complete path of the directory name to remove, starting from the application directory. For example DIR="Test" will remove a subdirectory named "Test" from the current application directory.
The DIR parameter may point to a full directory if the RELATIVE parameter is set to "NO".

RELATIVE

If this parameter is set to "NO", the directory pointed by DIR is considered to be a full directory path and not a relative application directory path. The default is "YES".

SELF

If this parameter is set to "NO", the command doesn't remove the directory itself but only some of its files or subdirectories. The files and subdirectories to remove are given by the following parameters.
When self is set to "YES", all the files and subdirectories of the DIR directory are removed and then the DIR directory itself is removed.
The default is "YES" (remove the entire directory).

FILES

This parameter has a meaning only when the SELF parameter is "NO". This parameter gives the beginning of the file names to remove. For example, if FILES is "tst", the command will remove all the files starting with "tst". If FILES is "*", all the files of DIR directory are removed. If FILES is blank, no files are removed.
The default is "*" (all the files).

SUBDIRS

This parameter has a meaning only when the SELF parameter is "NO". This parameter gives the beginning of the subdirectory names to remove. For example, if SUDIRS is "tst", the command will remove all the subdirectories starting with "tst". If SUDIRS is "*", all the subdirectories of DIR directory are removed. If SUDIRS is blank, no subdirectories are removed.
The default is "*" (all the subdirectories).

SESSIONS

APPLICATION:SESSIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the list of session identifiers currently connected to the application.

Permissions

APPLICATION_SESSION_LIST

Parameters

APPLICATION Name of the application.

Objects created

SESSIONS This is a Nirva string list object that contains the list of session identifiers currently connected to the application.

START

APPLICATION:START

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command starts the given application if this one is not started. Since an application is not a physical executable but just a context, starting an application just means enabling it.

Permissions

APPLICATION_RUN

Parameters

APPLICATION Name of the application to start. The application name is case insensitive.

STOP

APPLICATION:STOP

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command stops the given application. Stopping an application means disabling it and closing all sessions connected to it.

This command may be long because it closes all the sessions using the application and waits for them to be completely terminated.

It’s not possible to stop its own application. At this time, the command fails. It’s also not possible to stop the NIRVA default application (NVDEF application).

Permissions

APPLICATION_RUN

Parameters

APPLICATION Name of the application to stop. The application name is case insensitive.

COMMAND class

The test class contains commands for managing NIRVA commands.

GET_PARAMETER

COMMAND:GET_PARAMETER

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure (native only)	No	No	Yes

Description

This command retrieves a command parameter and writes it into the output buffer. It's available only when called from inside a native procedure.

Parameters

NAME Name of the command parameter to get.

ORIGIN If this parameter is set to "YES", the command retrieves a parameter of the original command. Otherwise, it gets a parameter from the command which has launched the native procedure.

GET_HTTP_HEADERS

COMMAND:GET_HTTP_HEADERS

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	Yes	No

Description

This command retrieves the http headers of the orginal command. If the original command doesn't come from the web, the command fails.

Parameters

None

Objects created

HTTP_HEADERS This is an indexed string list object containing the http headers of the original command.

SET_HTTP_HEADER

COMMAND:SET_HTTP_HEADER

Source	Use Input Container	Use Output Container	Use Output buffer
Web Procedure Service	No	No	No

Description

This command allows setting or changing an HTTP header for the original command. It works only if the original command comes from a web browser. It's not working in the session open procedure.The given HTTP header will be inserted in the resulting HTTP output of the command.

Parameters

HEADER Http header name.

VALUE Header value. If the value is blank, the HTTP header is removed (only if APPEND is set to NO). The header value may contain a line feed character. When there are line feeds, Nirva splits the value according to line feeds and creates an http header with the same name for each splitted value. This is useful for defining several cookies.

APPEND If this parameter is set to YES, the VALUE is added to the current header value instead of replacing it. If the previous value wasn't empty, a line feed (\n) is inserted before the new value. The default is “NO”.

SET_HTTP_RET_CODE

COMMAND:SET_HTTP_RET_CODDE

Source	Use Input Container	Use Output Container	Use Output buffer
Web	No	No	No

Description

This command allows setting the HTTP return code for the original command. It works only if the original command comes from a web browser.

Parameters

VALUE HTTP code value. This is one of the standard HTTP error codes (example: “500”).

SET_OUTPUT_BUFFER

COMMAND:SET_OUTPUT_BUFFER

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	Yes

Description

Procedures and services can write directly the output buffer using this command. This allows procedure or service commands returning string information directly accessible without having to create an object.

For services this command works only in implemented service commands. It doesn't work in service init/exit code and in session init/exit code.

For procedures this command works only for procedures called by the NV_PROC or NV_POST_PROC parameters.

Since the output buffer is erased after each Nirva command this command must be the last one before the procedure or service command ends.

Parameters

VALUE String information to write into the output buffer.

Example

File Applications/NVDEF/Procs/outputbuf1.pl:

```
NV::Command("NV_PROC=|perl:outbuf2|");
print "OutputBuffer of outbuf2 = $NV::RESULT\n";
```

File Applications/NVDEF/Procs/outputbuf2.pl:

```
NV::Command("NV_CMD=|COMMAND:SET_OUTPUT_BUFFER| VALUE=|My value|");
```

Run from console:

```
nvcc -r perl:outbuf1
```

Result in Nirva console:

```
17:02:54 43DDDEE2BC      NVDEF --- Start Perl procedure: perl:outbuf1
17:02:54 43DDDEE2BC      NVDEF --- Start Perl procedure: perl:outbuf2
17:02:54 43DDDEE2BC      NVDEF --- End perl procedure: perl:outbuf2
OutputBuffer of outbuf2 = My value
17:02:54 43DDDEE2BC      NVDEF --- End perl procedure: perl:outbuf1
```

REMOVE_PARAMETER

COMMAND:REMOVE_PARAMETER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command allows removing a parameter of the original command. The original command is this one that comes from a client or from the web browser.

Parameters

NAME Name of the original command parameter to remove.

SET_PARAMETER

COMMAND:SET_PARAMETER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command allows setting or changing a parameter of the original command. The original command is this one that comes from a client or from the web browser. This is very useful to change the original command parameter from a procedure that is executed before the current command.

For example, if a command SYSTEM:OBJECT:GET is sent from a web browser in order to download a file object, the SET_PARAMETER command can then be used in a procedure called before executing the OBJECT:GET (by using the NV_PROC parameter) in order to set the name of the file returned as attachment.

Parameters

NAME Name of the original command parameter to set or change.

VALUE Parameter value.

CONTAINER class

The CONTAINER class provides commands for working with session containers.

COPY

CONTAINER:COPY

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command copies the given input container (or a part of it) in the given output container. The input and output container names are defined by the general command parameters NV_CONTAINER, NV_IN_CONTAINER and NV_OUT_CONTAINER. The input and output container can be a complete container or subcontainer. The command verifies that the operation is possible and return an error if not. For example if trying to copy the container c1 to c1.c2 with the subcontainers option this fails because the command results in an infinite loop.

Parameters

APPEND Append mode. If this parameter is set to “YES”, the command will append the input container to the output container. In this case, the eventual replacement of existing objects is controlled by the parameter REPLACE.
When not in append mode, the output container is first released before the copy. This is the default.

REPLACE This parameter has a meaning only in append mode (APPEND parameter set to “YES”). If REPLACE is set to “YES” all input container objects will replace output container objects having the same name.

	The default is “NO” so the output container objects having the same name than the input container objects will not be replaced.
SUBCONTAINERS	This parameter allows to also copying the subcontainers of the input container to the output container. To copy subcontainers, the parameter must be set to “YES”. The default value is “NO”.
OBJECTS	This parameter allows specifying only some of the input container objects to copy to the output container. If used, the parameter should contain the names of valid input container objects, separated by semicolon character (‘;’). The default is to include all input container objects.
WITH_DATA	This parameter allows to also copy the object data (and not only the object description) of the input container to the output container. To not copy object data, the parameter must be set to “NO”. The default value is “YES”.
PERSIST	This parameter gives the new persistent value for persistent file objects that have to be copied. By default, the new file objects will not be persistent. The parameter can take value “0” for temporary files, “-1” for persistent files and any other value (in seconds) for cached files.
FILES	This parameter, if set to “YES”, tells Nirva to also copy files with file objects. If this parameter is set to “NO”, the file objects are copied but not the file they point to. When files are copied, Nirva automatically creates the new files in the application file directory if the file object is persistent and in the application work directory otherwise. This destination directory can be changed by the FILEDIR parameter if the command has been sent from a procedure or from an external service. The default value is “YES”.
FILEDIR	This parameter has meaning only if the FILES parameter has been set to “YES”. It allows changing the destination directory for files of file objects. If the FILEDIR is set to the application work directory and the PERSIST parameter is set to “-1” (persistent), the command will return an error because the application work directory should not contain any persistent file. The FILEDIR parameter can be used only if the command source is procedure or service. Otherwise, it’s ignored.

ENCODE

CONTAINER:ENCODE

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command changes the encoding of the input container (objects and subcontainers). If the container contains objects only the STRING, STRINGLIST, INDSTRINGLIST and TABLE objects will be encoded because encoding doesn't have meaning on other kind of objects.

Parameters

SRC_ENCODING Current encoding of the container. This can be "UTF-8", "ISO-8859-1" or "INTERNAL". If "INTERNAL" is used, then the container is considered to have the same encoding than the internal NIRVA encoding ("UTF-8" or "ISO-8859-1") . The default value is "INTERNAL".

DEST_ENCODING Required new encoding of the container. This can be "UTF-8", "ISO-8859-1" or "INTERNAL". If "INTERNAL" is used, then the container will be converted to the same encoding than the internal NIRVA encoding ("UTF-8" or "ISO-8859-1"). If both SRC_ENCODING and DEST_ENCODING are the same, then the command does nothing. The default value is "INTERNAL".

WITH_SUBCONTAINERS If this parameter is set to "YES", then NIRVA also changes the encoding of the subcontainers. The default is "YES".

WITH_NAME If this parameter is set to "YES", then NIRVA also changes the encoding of the container and object names (and eventually subcontainer names if WITH_SUBCONTAINERS parameter has been set to "YES"). The default is "NO".

EXPORT

CONTAINER:EXPORT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command exports the input container (or a part of it) in the given file. The exported file format is proprietary and should not be modified manually.

Parameters

FILENAME	Name of the export file. All container information is written into this file in a proprietary format.
SUBCONTAINERS	This parameter allows to also exporting the subcontainers of the input container. To copy subcontainers, the parameter must be set to "YES". The default value is "NO".
OBJECTS	This parameter allows specifying only some of the input container objects to export. If used, the parameter should contain the names of valid input container objects, separated by semicolon character (';'). The default is to include all input container objects.

GET_NUM_SUBCONTAINERS

CONTAINER:GET_NUM_SUBCONTAINERS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the number of subcontainers of the input container.

Parameters

none

GET_PARENT_NAME

CONTAINER:GET_PARENT_NAME

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the name of the parent of the input container given in the NV_CONTAINER or NV_IN_CONTAINER parameters.

If the input container is a root container, the parent is a blank string.

Parameters

none

GET_SUBCONTAINER_NAME

CONTAINER:GET_SUBCONTAINER_NAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the name of a subcontainer of the input container given a subcontainer index.

In conjunction with the GET_NUM_SUBCONTAINERS commands, this command can be used to enumerate the subcontainer names.

Parameters

INDEX	This is a number starting at one and having the maximum value equal to the number of subcontainers returned by the GET_NUM_SUBCONTAINERS command.

IMPORT

CONTAINER:IMPORT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	Yes

Description

This command imports the content of the given file into the output container.

The command also allows importing only a specific object embedded in the given file.

The import file must be the result of a previous CONTAINER EXPORT command.

The IMPORT command returns size information in the output buffer. If only a single object has to be imported, this is the size of this object. Otherwise, the size is the size of the last imported object.

The size content depends of the object type:

- For a boolean object the size is always 1.
- For an integer object the size is the size in bytes of a C++ integer (generally 4).
- For a string object, the size is the string length.
- For a string list object, the size is the list size.
- For an indexed string list object, the size is the list size.
- For a table object, the size is the number of rows.
- For a file object, the size is the file length. The IMPORT command allows to not really importing files if requested (see the FILES parameters). At this time, the size information is also the real file length. This allows using the import function just to require the length of a file embedded in the importation file. In the same way, if only a part of the files has to be imported (see OFFSET and NUM_BYTES parameters), the size returned in the output buffer is still the real file size.
- For a binary object, the size is the size of the binary data.

Parameters

FILENAME	Name of the import file. This must be the result of a previous CONTAINER EXPORT command.
APPEND	Append mode. If this parameter is set to “YES”, the command will append the imported objects and subcontainers to the output container. In this case, the eventual replacement of existing objects is controlled by the parameter REPLACE. When not in append mode, the output container is first released before the copy. This is the default.

REPLACE	This parameter has a meaning only in append mode (APPEND parameter set to “YES”). If REPLACE is set to “YES” all imported objects will replace container objects having the same name. The default is “NO” so the container objects having the same name than the imported objects will not be replaced.
PERSIST	This parameter gives the new persistent value for persistent file objects that have to be imported. By default, the new file objects will not be persistent. The parameter can take value “0” for temporary files, “-1” for persistent files and any other value (in seconds) for cached files.
FILES	This parameter, if set to “YES”, tells Nirva to also import files with file objects. If this parameter is set to “NO”, the file objects are imported but not the file they point to. When files are really imported, Nirva automatically creates the new files in the application file directory if the file object is persistent and in the application work directory otherwise. This destination directory can be changed by the FILEDIR parameter if the command has been sent from a procedure or from an external service. The default value is “YES”.
FILEDIR	This parameter has meaning only if the FILES parameter has been set to “YES”. It allows changing the destination directory for files of file objects. If the FILEDIR is set to the application work directory and the PERSIST parameter is set to “-1” (persistent), the command will return an error because the application work directory should not contain any persistent file. The FILEDIR parameter can be used only if the command source is procedure or service. Otherwise, it’s ignored.
OBJECT	If this parameter is not blank, it must contain the name of a single object to import from the import file to the output container. The single object may point to a subcontainer of the import file. At this time it must contain the entire path to the requested object. For example: “`Sub1.Sub11.MyObject`” tells NIRVA to get only the object “`MyObject`” of the subcontainer “`Sub11`” of the subcontainer “`Sub1`”. The subcontainer and object names are case insensitive.
OFFSET	Offset. This parameter is used only for file and binary objects to import. It gives the offset in bytes from which the copy will start. The default value is ”0” (beginning of file or binary data).
NUM_BYTES	Number of bytes to copy. This parameter is used only for file and binary objects to import. It gives the number of bytes of the object to import. The default value is ”-1” (until the end of file or binary data).

LIST

CONTAINER:LIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command lists the containers (root containers) at session, application, service or system levels.

Parameters

LIST Name of the output object. The default is “LIST”.

LEVEL Level. This can be “SESSION” (default) to get session containers, “APPLICATION” to get application containers, “SYSTEM” to get system containers, “SERVICE” to get service containers (the name of the service must then be given in the SERVICE parameter) and “PARENT” to get the parent containers if the current session is a named session.

SERVICE Service name when level is “SERVICE”.

Objects created

LIST This is a Nirva string list object that contains the list of containers.

MOVE

CONTAINER:MOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	Yes	No

Description

This command moves the given container (or input container if the NAME parameter is not given) to the output container. There is no copy of object or subcontainers, only their reference are moved from one container to the other. If the output container is in a branch of the container to move, the command fails. Attention: the output container is cleared before the move operation. If the container to move is the input container, the command doesn’t remove it but just clears it.

Parameters

NAME

Name of container or subcontainer to move.
Since a container is a hierarchical structure, the moving of a subcontainer is made by giving the complete subcontainer path. For example "`Container.sub1`" moves the subcontainer "`sub1`" of the container "`Container`".
A container has a session, application, service or system scope. By default, the container has a session scope. In order to access an application container, the container name must be preceded by the character ':' without any other string in front. In order to access a service container, the container name must be preceded by the service name and the character ':' (for example `Myservice:Myscontainer`). In order to access a system container, the container name must be preceded by the string 'SYSTEM' and the character ':' (for example `SYSTEM:Myscontainer`).
In some situations, a session may have a parent session. This occurs when a session calls a command to be executed by another session. This is the case, for example, when using named sessions. At this time, the called session can access directly the containers of the caller session. For a called session to access its parent containers, the container name must be preceded with the string "[NV_PARENT]:". If the session doesn't have parent, this syntax simply calls the current session container. For example, `[NVPARENT]:MyContainer` will access a container named MyContainer in the caller session.
If the container name is equal to the input container, the container is not removed after move but simply cleared. Clearing a container means removing all its objects and subcontainers.
The default value for this parameter is the input container name.

REMOVE

CONTAINER:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command deletes the given container or subcontainer. Files associated with file objects that are not persistent are removed or sent to cache control if they have a persistent time.

Parameters

NAME

Name of container or subcontainer to remove.
Since a container is a hierarchical structure, the removing of a subcontainer is made by giving the complete subcontainer path. For example "`Container.sub1`" removes the subcontainer "`sub1`" of the container "`Container`".
A container has a session, application, service or system scope. By default, the container has a session scope. In order to access an application container, the container name must be preceded by the character ':' without any other string in front. In order to access a service container, the container name must be preceded by the service name and the character ':' (for example `Myservice:Myscontainer`). In order to access a system container, the container name must be preceded by the string 'SYSTEM' and the character ':' (for example `SYSTEM:Myscontainer`).
In some situations, a session may have a parent session. This occurs when a session calls a command to be executed by another session. This is the case, for example, when using named sessions. At this time, the called session can access directly the containers of the caller session. For a called session to access its parent containers, the container name must be preceded with the string "[NV_PARENT]:". If the session doesn't have parent, this syntax simply calls the current session container. For example, `[NVPARENT]:MyContainer` will access a container named MyContainer in the caller session.
If the container name is equal to the input or output container, the container is simply cleared but not removed. Clearing a container means removing all its objects and subcontainers.
The default value for this parameter is "nvdef" that clears the default container.

VIEW

CONTAINER:VIEW

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command lists the subcontainers and objects of the input container. The result is written in a table object in the output container.

Warning: if the input and output containers are the same, the object created by the command is also listed in itself.

Parameters

INFO — Name of the table object that the command will create in the output container. The default is “INFO”.

Objects created

INFO — This is a Nirva table object containing the following columns:

- “NAME” is the parent, subcontainer or object name, depending of the TYPE column.
- “TYPE” is set to “PARENT” for the parent container, to “CHILD” for a subcontainer and to “OBJECT” for an object.
- “INFO”. This is extra information depending of the TYPE column. If type is set to “PARENT”, INFO contains the parent complete path. If type is set to “CHILD”, INFO is the subcontainer complete path. If type is set to “OBJECT”, INFO contains the object’s type that can be “BOOLEAN”, “INTEGER”, “STRING”, “STRINGLIST”, “INDSTRINGLIST”, “TABLE”, “FILE” or “BINARY”.

DEBUG class

The DEBUG class provides commands for sending debug information to the console. It works only when nirva is in console and debug mode.

DISPLAY_CONTAINER

DEBUG:DISPLAY_CONTAINER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command displays the content of a container (object and subcontainer names) to the console when Nirva is in debug mode.

Parameters

NAME Name of the container to display. Default is output container.

DISPLAY_MESSAGE

DEBUG:DISPLAY_MESSAGE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command displays a message to the console when Nirva is in debug mode.

Parameters

MESSAGE Text to display.

DISPLAY_OBJECT

DEBUG:DISPLAY_OBJECT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command displays the content of a given object of the input container to the console when Nirva is in debug mode.

Parameters

NAME Name of the object to display.

COLSEP Table column separator for a table object. Default is “;”.

LINESEP Line separator for a table object. Default is “|”.

ROWS	Rows to display for a table object. The default is to display all rows. Row can give a specific row index (starting from 1) or a range of rows separated by a "-" character. For example "2-31" will display rows 2 to 31. If the value is "-1", the last row is displayed. For example "-1" displays the last row, "2- -1" displays from second to last row.

DISPLAY_VARIABLES

DEBUG:DISPLAY_VARIABLES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command displays the list of session variables to the console when Nirva is in debug mode.

Parameters

None

FILEDB class

The FILEDB class provides commands for working with SqLite 3 databases. Sqlite database is a fast single file database engine with no configuration.

Nirva provides an interface to SqlLite with only 3 commands:

- OPEN opens a database and creates it if not exists.
- CLOSE closes the database previously opened with OPEN command.
- SQL sends a sql order to an opened database.

Several databases can be opened at the same time and several users can concurrently access a database.

CLOSE

FILEDB:CLOSE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Closes a database previously opened by the OPEN command. All databases opened by a session are automatically closed when the session closes.

Parameters

NAME	Name that identifies the connected database as given in the OPEN command.

OPEN

FILEDB:OPEN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Opens and eventually creates a database. The database is identified by a single file. One can give a name to the opened database in the case the same session has to access different databases. This name is valid only for the duration of the session.

Parameters

FILENAME	File name of the database to open and eventually create. This parameter is mandatory.
NAME	Name that identifies the connected database to be used by the CLOSE and SQL commands. This name is not mandatory except when a session needs to access several databases.

If a database with the same name is already opened, Nirva doesn't reopen it except if it points to a different file name.

SQL

FILEDB:SQL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

Sends an SQL order to an opened database. The command returns a Nirva table object containing result data if the SQL order has it. For example, a SELECT order will give back all the requested records.

Parameters

NAME Name that identifies the connected database as given in the OPEN command.

SQL Sql order. The SQL syntax can be found on Sqlite official web site. This is standard SQL but some Sqlite specific SQL is also available.

RESULT Name of the Nirva table object that the command will create to send result data if there is some. If RESULT value is "NONE", no sql data will be returned. The default value is "RECORDS".

TIMEOUT Time out in seconds for the command to execute. The default value is 60 seconds. If the time out condition occurs, the command will fail with an error FILEDB:103.

Objects created

RECORDS This is a Nirva table object containing eventual data resulting from the SQL order.

LICENSE class

The LICENSE class provides commands for creating and controlling licenses. NIRVA manages licenses for itself and for external services or applications.

There are some commands dedicated to service and application providers in order for them to create their own license and to import them to NIRVA.

NIRVA maintains the license keys in a file named “nirva.lsc” that resides in the Nirva bin directory. A license key is composed of the following items:

The service name is the name of the service concerned by this license.

The service ID or application ID is a unique identifier known only by the service or application provider. This is the service or application private identifier that should never be given to anybody. The provider uses this private service or application ID to create license channels with the Nirva nvl tool. A public service or application ID is also available. The provider uses the public service or application ID for checking a license channel from its code.

The license key name is the name of the license key.

The license key value is an optional value associated to the license key. For example, it should be the maximum number of users.

The license key date is an optional expiration date of the license key.

The NIRVA license manager uses a unique machine identifier to assume that any installation has unique license keys.

Here is the usual procedure for a customer to request a license from a service provider:

The customer uses the LICENSE INFO command to request the unique machine identifier of its installation.

It then communicates this machine identifier to the service provider.

The service provider uses the LICENSE SET and LICENSE REMOVE commands to create a license file containing the requested licenses.

The service provider sends the license file to the customer.

The customer uses the LICENSE INSTALL command to install the license file in the NIRVA license manager.

Nirva provides a tool named nvl (windows tool) allowing providers to create their license files.

EXPORT

LICENSE:EXPORT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command exports some or all the license keys from the NIRVA license manager to a license file. The license file is in a NIRVA file object in the output container.

Parameters

SERVICE	Name of the external service to export license information from. If this parameter is not given or is blank, the command exports license information about all services.
FILE	Name of NIRVA file object containing a license file. If this parameter is not provided, NIRVA uses "LICENSE_FILE" as file object name. If the given object doesn't exist, the command creates it.

Objects created

LICENSE_FILE	This is a Nirva file object containing the license file with the exported license keys from the NIRVA license manager.

GET

LICENSE:GET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command is used only from an application or external service for checking a license key. If the license key is OK, the command also returns its value into the output buffer.

The command fails if the given key doesn't exist in the NIRVA license manager or if it has expired.

Parameters

SERVICE_ID	Unique service identifier. This can be the private or public service or application ID. For security reasons, it's preferable to give here only the public identifier.
SERVICE	Name of the service that owns the key.
KEY	License key name.

GET_TYPE

LICENSE:GET_TYPE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the current global Nirva license type. The result is in the output buffer. This command is deprecated. It stays only for compatibility reasons and always returns the value “PROD”.

Parameters

None.

INFO

LICENSE:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information about the licenses:

- Service.
- Name.
- Value
- Date

The INFO command returns information in a NIRVA table object named “INFO” in the output container. It also creates a string object named “MACHINE_ID” that gives back the unique machine identifier used by service providers to generate the licenses.

If the parameter “SERVICE” is given, the command just reports license information on the requested service.

If the parameter “FILE” is given, the command reports license information contained in a license file instead of the license information of the NIRVA license manager. This feature can be used by service providers to check the content of a license file.

Parameters

SERVICE Name of the external service or application to get license information from. If this parameter is not given or is blank, the command retrieves license information about all services and applications.

FILE Name of NIRVA file object containing a license file. If this parameter is provided and not blank, NIRVA returns license information from the license file. Otherwise, NIRVA returns information from the license manager.

Objects created

INFO This is a Nirva table object containing 4 columns:

- “SERVICE” is the service name (or application name).
- “KEY” is the license key name.
- “VALUE” is the optional license key value.
- “DATE” is the optional expiration date. The DATE column has the format YYYYMMDD and is blank if there is no date.
- *MACHINE_ID* This is a Nirva string object containing the unique machine identifier used by service providers to generate the licenses.

INSTALL

LICENSE:INSTALL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command installs the license keys contained into a license file in the NIRVA license manager. The existing keys with the same identification are replaced by the new ones.

Permissions

LICENSE_UPDATE

Parameters

FILE	Name of NIRVA file object containing the license file to install. If this parameter is not provided, NIRVA uses “LICENSE_FILE” as file object name. The object must exist for the command to be successful.

ISDEMO

LICENSE:ISDEMO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

Tests if the server is in demonstration mode (no global running authorization). If the server is in demo mode, the output buffer is filled with the value “YES”, otherwise it’s set to “NO”. This command is deprecated. It stays only for compatibility reasons and always returns the value “NO”.

Parameters

none.

REMOVE

LICENSE:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes a license key from the NIRVA license manager or from a license file.

Permissions

LICENSE_UPDATE

Parameters

FILE	Name of NIRVA file object containing the license file to work with. If this parameter is not provided, NIRVA works on the license manager directly. If this parameter is provided, the file object must exist.
SERVICE_ID	Unique service or application identifier. This identifier is secret and should be known only by the service provider. This is the private identifier. It's generated automatically by the SYSTEM SERVICE SKELETON command or explicitly by the SYSTEM SERVICE IDENT command. The service identifier must be exactly 32 characters long.
SERVICE	Name of the external service. If this parameter is not given or is blank, the command removes all license information from the license file or from the NIRVA license manager. Otherwise, the command only removes the license keys of the given service.
KEY	License key name to remove. This parameter is meaningful only when the SERVICE parameter has been given. It allows removing only a single license key from the license file or from the NIRVA license manager.

SET

LICENSE:SET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command creates or changes a license key in a license file. It's reserved to service or application providers who must manage their customer licenses.

The SET command uses the unique service or application identifier that must be known only by the service or application provider.

Permissions

LICENSE_UPDATE

Parameters

FILE	Name of NIRVA file object containing the license file to work with. If this parameter is not provided, NIRVA uses “LICENSE_FILE” as file object name. The object is created (in the input container) if it doesn't exist.

MACHINE_ID	Unique machine identifier. This string is given by the LICENSE INFO command. If it's provided, the current machine identifier of the license file will be replaced by the given one. The machine identifier must be exactly 32 characters long.
SERVICE_ID	Unique service or application identifier. This identifier is secret and should be known only by the service provider. This is the private identifier. It's generated automatically by the SYSTEM SERVICE SKELETON command or explicitly by the SYSTEM SERVICE IDENT command. The service identifier must be exactly 32 characters long.
SERVICE	Name of the service or application that owns the key. Nirva truncates it to 40 characters if it's too long.
KEY	License key name. This can be any string that identifies the license key. Nirva truncates it to 40 characters if it's too long. In fact, a license key is entirely identified by the SERVICE_ID, the SERVICE and the KEY parameters.
VALUE	License key value. This is an optional string that can be used by the service provider. For example a service provider should limit the number of concurrent users to 20 by setting a license key named MAXUSERS with a value of 20. Nirva truncates the key value to 40 characters if it's too long.
DATE	Optional license key expiration date. If there is no expiration date, this parameter should not be provided or set to blank. If provided, the DATE parameter must have the format YYYYMMDD.

LISTENER class

The LISTENER class provides commands for managing Nirva listeners.

Nirva listeners are threaded sessions that have the application life and call a procedure in a loop. After each execution of the procedure, the listener waits for the given sleep time. Several threads can be defined for a single listener allowing controlling the power given to some Nirva processing.

Listeners are good candidate for processing the Nirva workflow queue activities.

Listeners are attached to application so each application maintains its own list of listeners.

When a thread listener starts, it creates a Nirva session that is maintained during all the thread life. The same session is used for different instance of the listener procedure in the loop so one can keep a session context across procedure instances.

CREATE

LISTENER:CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command creates a new listener if it doesn’t exist. If the listener already exists, the command fails.

Permissions

LISTENER_ADMIN

Parameters

NAME	Listener name. This name identifies the listener. It can contain space characters. This parameter is mandatory.
DESCRIPTION	Optional description.
PROC	Name of the procedure that will be executed by the listener. If this parameter is not provided, the listener will be created with the given number of threads but the execution will do nothing. The listener procedure will be executed in loop, each occurrence separated by a sleep time given in the SLEEP parameter. If the execution time of the procedure is long, it should include some calls to the SESSION:CHECK_CLOSE_REQUEST command and leave if there is a request to close. The listener procedure can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name.
USER	User name. This is the name of the user of the listener session. This must be a valid user. If this parameter is not provided, Nirva uses the default user (nvdef).
OPEN	Name of the procedure that will be called when the listener session is opened. If the value is blank, no procedure will be called (default). If a procedure is given and the procedure returns an error, the listener session will not be created. This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).

CLOSE	Name of the procedure that will be called when the listener session is closed. If the value is blank, no procedure will be called (default). This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
THREADS	Number of threads to execute for this listener. The minimum value is 1 (default)
SLEEP	Sleep time in milliseconds between each occurrence of the thread. The default is 100 milliseconds. If set to 0, no sleep time will occur.
PROCF	Name of a procedure that will be ran in case of failure. This parameter is not mandatory. The failed procedure can be used for error reporting. It receives the error information as following parameters: NV_ERROR_CODE, NV_ERROR_SERVICE, NV_ERROR_CLASS, NV_ERROR_DESC and NV_ERROR_INFO. If defined, the failed procedure is called each time the main procedure fails.

LIST

LISTENER:LIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information about one or several listeners:

The LIST command returns information in a NIRVA table object named “LISTENER_LIST” in the output container.

If the parameter “NAME” is given, the command just reports information on the requested listener.

Permissions

LISTENER_ADMIN

LISTENER_LIST (used only when LISTENER_ADMIN is not set)

Parameters

NAME Listener name. If this parameter is provided with a valid listener name, the command returns information about this listener. Otherwise, the command returns information about all listeners of the current application.

Objects created

LISTENER_LIST This is a Nirva table object containing the following columns:

- “NAME” is the listener name.
- “DESCRIPTION” is the listener description.
- “PROC” is the listener procedure.
- “USER” is the user name.
- “OPEN” is the session open procedure.
- “CLOSE” is the session close procedure.
- “THREAD” is the number of threads.
- “SLEEP” is the sleep time in milliseconds.
- “STATUS” is set to “RUNNING” or “STOPPED”.

REMOVE

LISTENER:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes an existing listener. To remove a listener, this one must have been previously stopped otherwise the command fails.

Permissions

LISTENER_ADMIN

Parameters

NAME Listener name. This name identifies the listener. This parameter is mandatory.

SET_PARAM

LISTENER:SET_PARAM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command sets one or several parameters of the listener. The listener must not run otherwise the command fails.

Only the given parameters will be modified.

Permissions

LISTENER_ADMIN

Parameters

NAME	Listener name. This name identifies the listener. It can contain space characters. This parameter is mandatory.
DESCRIPTION	Optional description.
PROC	Name of the procedure that will be executed by the listener. If this parameter is not provided, the listener will be created with the given number of threads but the execution will do nothing. The listener procedure will be executed in loop, each occurrence separated by a sleep time given in the SLEEP parameter. If the execution time of the procedure is long, it should include some calls to the SESSION:CHECK_CLOSE_REQUEST command and leave if there is a request to close. The procedure receives the parameter NV_LISTENER_NAME that contains the listener name The listener procedure can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name.
USER	User name. This is the name of the user of the listener session. This must be a valid user. If the value is blank, Nirva uses the default user (nvdef).
OPEN	Name of the procedure that will be called when the listener session is opened. If the value is blank, no procedure will be called. If a procedure is given and the procedure returns an error, the listener session will not be created. This can be a native, java, dotnet or perl procedure. See the description of

	the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
CLOSE	Name of the procedure that will be called when the listener session is closed. If the value is blank, no procedure will be called. This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
THREADS	Number of threads to execute for this listener. The minimum value is 1.
SLEEP	Sleep time in milliseconds between each occurrence of the thread. If set to 0, no sleep time will occur.
PROCF	Name of a procedure that will be ran in case of failure. This parameter is not mandatory. The failed procedure can be used for error reporting. It receives the error information as following parameters: NV_LISTENER_NAME, NV_ERROR_CODE, NV_ERROR_SERVICE, NV_ERROR_CLASS, NV_ERROR_DESC and NV_ERROR_INFO. If defined, the failed procedure is called each time the main procedure fails.

START

LISTENER:START

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command starts a listener. The listener will run until stopped by the STOP command or until the application that owns it stops.

Permissions

LISTENER_ADMIN

Parameters

NAME	Listener name. This name identifies the listener. This parameter is mandatory.

STOP

LISTENER:STOP

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command stops a listener.

Permissions

LISTENER_ADMIN

Parameters

NAME	Listener name. This name identifies the listener. This parameter is mandatory.
WAIT	Wait time in seconds. The command will wait for the listener to stop within the given number of seconds. If after this time out, the listener stills work, the command will fail except if the FORCE parameter has been set to “YES”. At this time, when FORCE parameter has been set to “YES”, Nirva hardly kills the pending threads attached to the listener. The default wait time is 10 seconds.
FORCE	Force flag. See the description of the WAIT parameter. The default is “NO”.

LOCK class

The LOCK class provides commands for managing lock objects.

A Nirva lock object is a named object having a status locked or unlocked. A lock is owned by a session so only the session who locked a lock object can unlock it.

When a session is closed, all the lock objects it owns are automatically free.

A session who wants to access a lock object can tell the system to wait for the object to be free for a given time, to not wait (return error if the lock is locked) or to wait infinitely.

String information can be associated to a lock object.

Nirva also provides a command to test the state of a lock object and to return the associated information if there is one.

Nirva maintains lock lists at system and application levels.

Since NIRVA checks the session time outs every 30 seconds. An object owned by a timeouted session can stay locked a maximum of 30 seconds after the time out occurs.

INFO

LOCK:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the current system or application lock list (or a part of it). For each lock object, the command returns the following information:

- Name.
- Owner.
- Information associated

The INFO command returns information in a NIRVA table object named "INFO" in the output container.

It's possible to reduce the list by searching for some specific lock object names.

Parameters

NAME — Name of the lock object. This parameter, when provided, is used to restrict the list to specific lock object names. The wildcard character '*' can be used for searching object names starting with a given value. If NAME is "*", NIRVA doesn't restrict the list based on the lock name. This is the default.

SYSTEM — System level lock. By default, this parameter is set to "NO" so Nirva returns the application lock list. If SYSTEM is set to "YES", Nirva returns the system lock list.

Objects created

INFO — This is a Nirva table object containing the following columns:

- "NAME" is the lock object name.
- "OWNER" is the session identifier of the session who has locked the object.
- "INFO" gives the eventual information associated with the lock object when the object has been locked. It can be empty.

ISLOCKED

LOCK:ISLOCKED

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	Yes

Description

This command tests of a lock object is locked or not and returns information about it. If the object is locked, the output buffer returns “YES”. Otherwise it returns “NO”.

Parameters

NAME Name of the lock object. The lock name is case insensitive.
This parameter is mandatory.

SYSTEM System level lock. By default, this parameter is set to “NO” so Nirva searches for the lock object in the application lock list. If SYSTEM is set to “YES”, Nirva searches for the lock object in the system lock list.

Objects created

LOCKED This is a boolean object set to TRUE if the object is currently locked and to FALSE otherwise.

OWNER This is a string object that gives the owner of the lock object when the object is locked. The owner is the session identifier of the session who has locked the object.
If the object is not locked, the OWNER string is empty.

INFO This is a string object that gives the eventual information associated with the lock object when the object has been locked.
If the object is not locked, the INFO string is empty.

LOCK

LOCK:LOCK

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command tries to gain access to the given lock object. If the lock object doesn't exist, Nirva creates it automatically. If the lock is successful, the session becomes the lock owner. It will own the lock until it unlocks it or until the session closes.

If the command cannot get the object locked, it returns an error message or a boolean object depending if the GOT_LOCK parameter is given or not.

Parameters

NAME	Name of the lock object. The lock name is case insensitive. This parameter is mandatory.
INFO	Lock information string. This parameter allows associating a string to the lock object. This string can be retrieved with the ISLOCKED command. The string is associated only if the lock has been successful.
WAIT	Maximum number of seconds to wait for getting the lock of the lock object. If this value is -1, the command waits infinitely for the object to be locked. While waiting for a lock object to be locked, the command doesn't require any machine time. The default WAIT value is “300”.
SYSTEM	System level lock. By default, this parameter is set to “NO” so the lock object is at application level. If SYSTEM is set to “YES”, the lock object is at system level. The application and system lock lists are completely independent so an object of the same name can be at application and system level.
SPEED	Lock waiting speed. This parameter can take values from 1 to 10. 10 is faster but requires more CPU time. The default is 1.
GOT_LOCK	Name of a boolean object in which NIRVA will write the result of the command. If the GOT_LOCK parameter is given, the command always successes (expect in case of major failure) and sets the object which name is given by “GOT_LOCK” to TRUE or FALSE depending if the ownership of the lock has been got or not. The boolean object is created by the command in the output container. If the GOT_LOCK parameter is not given, the command returns an error if the user cannot get the ownership of the lock.

Objects created

<boolean> The command creates a boolean object if the GOT_LOCK parameter is not empty. The object name is the one given by the GOT_LOCK parameter. Please see the description of the GOT_LOCK parameter.

UNLOCK

LOCK:UNLOCK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command tries to unlock a previously locked object. Normally, only the session who has locked the lock object can unlock it but the UNLOCK command allows to also unlock a lock object of another session if the FORCE parameter is set to “YES”. This force mode should be reserved only for administration function.

If the command cannot unlock the object, it returns an error message.

Parameters

NAME Name of the lock object. The lock name is case insensitive.
This parameter is mandatory.

SYSTEM System level lock. By default, this parameter is set to “NO” so Nirva searches for the lock object in the application lock list. If SYSTEM is set to “YES”, Nirva searches for the lock object in the system lock list.

FORCE When this parameter is set to “YES”, the command accepts to unlock a lock object that has been locked from another session.
By default, this parameter is set to “NO”.

LOG class

The LOG class provides commands for manipulating log information.

Nirva provides some complex log features. Logs exist at system, application or service levels.

The logs can be consulted directly from a web browser with search facilities.

A NIRVA log is a flat file with a proprietary format (textual format with some specific separators and header) that contains log records. A log record itself is composed of the following information:

- DATE is the record write date (format YYYYMMDD). The Date is automatically set by the system.
- TIME is the record write time (format HHMMSS). The Time is automatically set by the system.
- IDENTIFIER is a free text that identifies a user entity. For example, for the system log, the identifier is the NIRVA session identifier.
- TYPE is a numeric code that defines the kind of record. It can be 0 for information record, 1 for a warning record and 2 for an error record.
- MESSAGE is the log message itself. It's textual information.
- EXTRA is an optional second message. It can be used to separate some information from the message itself. It's textual information.
- SOURCE is a free text that can be used to give information about the origin of the log record (who has written the record ?).
- LEVEL is the log level. It's numeric information that takes values from 1 to 6. This is the responsibility to the log writer to manage the log levels. The NIRVA LOG SEARCH command allows specifying the maximum log level to search for.

A log record is limited to 65536 characters.

NIRVA provides some functionality to limit the size of log files and to save them in a dedicated directory structure. This directory structure is date based allowing easily retrieve the log information for a date range.

All the logs can be controlled and searched from the NIRVA configuration tool. Please refer to the configuration chapter for further information.

NIRVA provides a system log that cannot be removed and displays information at system level. This log is named “SYSTEM”. Following the chosen level, it displays the start of each command, the end of the each command, the command errors and the command parameters. When logging command parameters, any parameter starting with “PASS”, “PSW” or “PWD” is discarded from the display. This is to avoid logging passwords.

CREATE

LOG:CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command creates a new log if it doesn't exist. If the log already exists, the command tries to clean some free directories in the log data and changes the eventual log parameters by the ones given in the command.

Permissions

SYSTEM_LOG_ADMIN

SERVICE_LOG_ADMIN

APPLICATION_LOG_ADMIN

Parameters

LOG	Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character.
SERVICE	This parameter tells if the log is a system, service or application log. By default, if the SERVICE parameter is not provided or is blank, the log is considered to be an application log. If SERVICE is set to "SYSTEM", the log is a system log. If SERVICE is set to the name of an external service, the log is considered to be a service log.
ERASE	Erase mode. If this parameter is set to "YES", any eventual data of a previous log with the same name is erased. This occurs only if the log did not exist before sending the command. The default is "NO".
FILE	Import file. This is the name of a NIRVA file object that resides in the input container. It allows importing some data to the newly created log. This is useful for support people who can create and consult a log with data received from a customer. The physical import file must be the result of an exported log from the LOG SEARCH command. By default, this parameter is blank so the new log is created without any data. The FILE parameter has meaning only when the log is effectively created. So if the log already exists, it's not possible to import data to it. When importing data, the ERASE parameter is automatically set to "YES".
DESCRIPTION	Free text that describes the log.
LEVEL	Initial log level. This can be a numeric value from 0 to 6. If set to "0", the LOG WRITE command will not do anything. The default value is "1".
MAXTIME	Maximum time in minutes that a single log file must record. The default is 10 minutes. The minimum value is 1 minute.
SAVE	Save option. When this parameter is set to "YES", NIRVA saves all the log files to a dedicated directory based on a date structure. The save occurs when the log file has overflowed the maximum record time defined by the MAXTIME parameter. The default is "YES".
MAXSAVEDAYS	Maximum number of days for saved log data.

ERASE

LOG:ERASE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes some log records. The records are selected on a date or a date range.

Permissions

SYSTEM_LOG_ADMIN

SERVICE_LOG_ADMIN

APPLICATION_LOG_ADMIN

Parameters

LOG

Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character.

SERVICE

This parameter tells if the log is a system, service or application log. By default, if the SERVICE parameter is not provided or is blank, the log is considered to be an application log. If SERVICE is set to “SYSTEM”, the log is a system log. If SERVICE is set to the name of an external service, the log is considered to be a service log.

DATE

From date. This parameter allows limiting the selection to a given date or to a date range when used with the “TO_DATE” parameter. The general format of the DATE parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only DD/MM is given, NIRVA is using the current year.
If only DD is given, NIRVA is using the current month and year.
NIRVA can also accept the year on 2 digits.
If the DATE is preceded by a minus sign (‘-‘) followed by an integer. NIRVA considers that as a number of days before the current day. For example, “-3” is 3 days before the current day.
The default value for the DATE field depends of the parameter “TO_DATE”.

	If TO_DATE is not provided, the command doesn't remove anything. If TO_DATE is provided and DATE is empty, NIRVA sets DATE to "01.01.2000". This allows removing all log data to a given date.
TO_DATE	To date. This parameter allows limiting the selection to a date range. If it's not provided, the selection occurs for a fixed date given by the DATE parameter. If the TO_DATE parameter is given but the DATE parameter is blank (or not given), NIRVA removes all the log data until the given TO_DATE. The general format of the TO_DATE parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year. Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format. The separator can be '/', '\', '.', ' '(space) or ':'. It's also possible to not give the separator. If only DD/MM is given, NIRVA is using the current year. If only DD is given, NIRVA is using the current month and year. NIRVA can also accept the year on 2 digits. If the TO_DATE is preceded by a minus sign ('-') followed by an integer. NIRVA considers that as a number of days before the current day. For example, "-3" is 3 days before the current day. The default value for the TO_DATE field is the value of the DATE field.
CLEAR	Clears current log data. This parameter, when set to "YES" allows to also removing the current log data. The current log data is this one that is not saved. Physically, it's stored in 2 files named "nvs.log" and "nvs1.log". The default value is "NO".

GET_LEVEL

LOG:GET_LEVEL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the log level in the output buffer.

Parameters

LOG	Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character.
SERVICE	This parameter tells if the command has to return system, service or application log level. By default, if the SERVICE parameter is not provided or is blank, the command returns the application log level. If SERVICE is set to “SYSTEM”, the command returns system log level. If SERVICE is set to the name of an external service, the command returns service log level.

INFO

LOG:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information about one or several logs:

- Name.
- Description.
- Level
- Maximum record time
- Save option

The INFO command returns information in a NIRVA table object named “LOG_INFO” in the output container.

If the parameter “LOG” is given, the command just reports information on the requested log.

Permissions

SYSTEM_LOG_ADMIN or SYSTEM_LOG_READ

SERVICE_LOG_ADMIN or SERVICE_LOG_READ

APPLICATION_LOG_ADMIN or APPLICATION_LOG_READ

Parameters

LOG Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character. If this parameter is provided, the command returns information about the given log. Otherwise, it returns information about all the logs.

SERVICE This parameter tells if the command has to return system, service or application log information. By default, if the SERVICE parameter is not provided or is blank, the command returns the application log information. If SERVICE is set to “SYSTEM”, the command returns system log information. If SERVICE is set to the name of an external service, the command returns service log information.

Objects created

LOG_INFO This is a Nirva table object containing the following columns:

- “NAME” is the log name.
- “DESCRIPTION” is the log description.
- “LEVEL” is the current log level.
- “MAXTIME” is the maximum time in minutes that a single log file must record.
- “SAVE” is the log save option. It can be “YES” or “NO”.
- “MAXSAVEDAYS” is the maximum number of log days saved.

SEARCH

LOG:SEARCH

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command searches records in a log. The result is given as a NIRVA table object containing log records.

This table object has the following columns:

- DATE is the record write date (format YYYYMMDD.
- TIME is the record write time (format HHMMSS.mmm) where mmm is the number of milliseconds.

- IDENTIFIER is a free text that identifies a user entity. For example, for the system log, the identifier is the NIRVA session identifier.
- TYPE is a numeric code that defines the kind of record. It can be 0 for information record, 1 for a warning record and 2 for an error record.
- MESSAGE is the log message itself. It's textual information.
- EXTRA is the optional second message.
- SOURCE is a free text that gives the origin of the log record.
- LEVEL is the log level. It's numeric information that takes values from 1 to 6.
- THREAD is the ID of the thread that produced the log entry.

Optionally, the result can also be sent to an export file. This export file can then be used as input to the LOG CREATE function.

Permissions

SYSTEM_LOG_ADMIN or SYSTEM_LOG_READ

SERVICE_LOG_ADMIN or SERVICE_LOG_ READ

APPLICATION_LOG_ADMIN or APPLICATION_LOG_ READ

Parameters

LOG	Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character.
SERVICE	This parameter tells if the log is a system, service or application log. By default, if the SERVICE parameter is not provided or is blank, the log is considered to be an application log. If SERVICE is set to “SYSTEM”, the log is a system log. If SERVICE is set to the name of an external service, the log is considered to be a service log.
RESULT	Name of the NIRVA table object that will contain the result of the search. The default is “LOG_VIEW”. The command creates the object if it doesn't exist.
FILE	Name of the NIRVA file object that will contain the exported result. This export file can then be used as an import to the LOG CREATE command. If this parameter is not provided, the command doesn't export the result. If this parameter is provided but is blank, the command is using “LOG_FILE” as object name. If the object doesn't exist, the command creates it.
DATE	From date. This parameter allows limiting the search to a given date or to a date range when used with the “TO_DATE” parameter. The general format of the DATE parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year. Now, if the input format is different than this one, NIRVA may accept it by

trying to convert it to the required format.
The separator can be '/', '\', '.', ' '(space) or ':'. It's also possible to not give the separator.
If only DD/MM is given, NIRVA is using the current year.
If only DD is given, NIRVA is using the current month and year.
NIRVA can also accept the year on 2 digits.
If the DATE is preceded by a minus sign ('-') followed by an integer. NIRVA considers that as a number of days before the current day. For example, "-3" is 3 days before the current day.
The default value for the DATE field depends of the parameter "TIME". If TIME is provided, the default DATE parameter is the current date. If TIME is not provided, the default DATE parameter is date of the day 15 minutes before the current date (so it's the current day if the command is sent after 00:15).

TO_DATE

To date. This parameter allows limiting the search to a date range. If it's not provided, the search occurs for a fixed date given by the DATE parameter.
If the TO_DATE parameter is given but the DATE parameter is blank (or not given), NIRVA resets the TO_DATE parameter to blank. At this time, the search occurs only on the current date.
The general format of the TO_DATE parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be '/', '\', '.', ' '(space) or ':'. It's also possible to not give the separator.
If only DD/MM is given, NIRVA is using the current year.
If only DD is given, NIRVA is using the current month and year.
NIRVA can also accept the year on 2 digits.
If the TO_DATE is preceded by a minus sign ('-') followed by an integer. NIRVA considers that as a number of days before the current day. For example, "-3" is 3 days before the current day.
The default value for the TO_DATE field is the value of the DATE field.

TIME

From time. This parameter allows limiting the search to a given date/time or to a date/time range when used with the "TO_DATE" and TO_TIME parameters. The general format of the TIME parameter is HH:MM:SS where HH is the hour from 0 to 23, MM is the minute from 0 to 59 and SS is the second from 0 to 59.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be '/', '\', '.', ' '(space) or ':'. It's also possible to not give the separator.
If only HHMM is given, NIRVA is using 0 for the seconds.
If only HH is given, NIRVA is using 0 for the seconds and for the minutes.
If the TIME is preceded by a minus sign ('-') followed by an integer. NIRVA considers that as a number of hours before the current time. For example, "-

3” is 3 hours before the current time. If the integer is followed by an “m” character, NIRVA considers that as a number of minutes before the current time. For example, “-10m” is 10 minutes before the current time.
The default value for the TIME field depends of the parameter “DATE”. If DATE is provided, the default TIME parameter is 00:00:00. If DATE is not provided, the default TIME parameter is time of the day 15 minutes before the current time.

TO_TIME To time. This parameter allows limiting the search to a given date/time or to a date/time range when used with the “DATE” and TIME parameters.
The general format of the TO_TIME parameter is HH:MM:SS where HH is the hour from 0 to 23, MM is the minute from 0 to 59 and SS is the second from 0 to 59.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only HHMM is given, NIRVA is using 0 for the seconds.
If only HH is given, NIRVA is using 0 for the seconds and for the minutes.
If the TO_TIME is preceded by a minus sign (‘-‘) followed by an integer. NIRVA considers that as a number of hours before the current time. For example, “-3” is 3 hours before the current time. If the integer is followed by a “m” character, NIRVA considers that as a number of minutes before the current time. For example, “-10m” is 10 minutes before the current time.
The default value for the TO_TIME field depends of the parameter “DATE”. If DATE is provided, the default TO_TIME parameter is 23:59:59. If DATE is not provided, the default TO_TIME parameter is the current time.

MESSAGE Log message. This parameter allows to limit the search to a given log message string. The wildcard character '*' can be used for searching messages starting with a given value or ending with a given value. If the ‘*’ wildcard character is used before and after the search value, NIRVA will find messages containing the requested value.

EXTRA Log extra information. This parameter allows to limit the search to a given log extra message string. The wildcard character '*' can be used for searching messages starting with a given value or ending with a given value. If the ‘*’ wildcard character is used before and after the search value, NIRVA will find extra information containing the requested value.

TYPE Log record type. This parameter allows to limit the search to a given log type. This is a numeric value. It can take values “0” for information type, “1” for warning type and “2” for error type. If any other value is given, NIRVA doesn’t restrict the search for a record type (this is the default).

LEVEL Log level. This parameter allows limiting the search to a maximum log level. This is a numeric value. It can take values “1” to “6”.

IDENT Identifier. This parameter allows to limit the search to a given log identifier string (for example a specific session). The wildcard character '*' can be

	used for searching messages starting with a given value or ending with a given value. If the '*' wildcard character is used before and after the search value, NIRVA will find identifiers containing the requested value.
SOURCE	Log record source. This parameter allows to limit the search to a given source string. The wildcard character '*' can be used for searching messages starting with a given value or ending with a given value. If the '*' wildcard character is used before and after the search value, NIRVA will find sources containing the requested value.
MAXSIZE	Maximum intermediate file size. In order to avoid starting too big searches, NIRVA limits the size of an internal log data file (constructed with only the date range criteria) to the first 1024 kilobytes. This limit can be changed by setting the MAXSIZE parameter to another value.

Objects created

LOG_VIEW	This is a Nirva table object containing the log information (see description). The name of this object can be changed with the RESULT parameter.

SET_OPTIONS

LOG:SET_OPTIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command allows changing some of the log options.

If some of the SET_OPTION parameters are not provided, their corresponding log option will not be changed.

Permissions

SYSTEM_LOG_ADMIN

SERVICE_LOG_ADMIN

APPLICATION_LOG_ADMIN

Parameters

LOG	Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character.
SERVICE	This parameter tells if the log is a system, service or application log. By default, if the SERVICE parameter is not provided or is blank, the log is considered to be an application log. If SERVICE is set to "SYSTEM", the log is a system log. If SERVICE is set to the name of an external service, the log is considered to be a service log.
DESCRIPTION	Free text that describes the log.
LEVEL	Current log level. This can be a numeric value from 0 to 6. If set to "0", the LOG WRITE command will not do anything.
MAXTIME	Maximum time in minutes that a single log file must record. The minimum value is 1 minute.
SAVE	Save option. When this parameter is set to "YES", NIRVA saves all the log files to a dedicated directory based on a date structure. The save occurs when the log file has overflowed the maximum record time defined by the MAXTIME parameter.
MAXSAVEDAYS	Maximum number of days for saved log data.

REMOVE

LOG:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes a log.

Permissions

SYSTEM_LOG_ADMIN

SERVICE_LOG_ADMIN

APPLICATION_LOG_ADMIN

Parameters

LOG	Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character.
SERVICE	This parameter tells if the log is a system, service or application log. By default, if the SERVICE parameter is not provided or is blank, the log is considered to be an application log. If SERVICE is set to “SYSTEM”, the log is a system log. If SERVICE is set to the name of an external service, the log is considered to be a service log.
ERASE	Erase mode. If this parameter is set to “YES”, the data associated with the log is also removed. The default is “NO”.

WRITE

LOG:WRITE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command writes a new log record. If the maximum time for a log file has elapsed, NIRVA saves the previous log file if the save option has been set. In any case, NIRVA always maintains at least the log information of the maximum time defined for the log. For example, if the maximum time has been set to 10 minutes, NIRVA will always keep at least the last 10 minutes of log. This is done by working permanently with 2 log files named “nvs.log” and “nvs1.log”. “nvs.log” maintains the log for the current period while “nvs1.log” maintains the log for the previous period.

Parameters

LOG	Log name. This is the unique name that identifies the log at system, application or service level. The log name should not contain space or any special character.
SERVICE	This parameter tells if the log is a system, service or application log. By default, if the SERVICE parameter is not provided or is blank, the log is considered to be an application log. If SERVICE is set to “SYSTEM”, the log is a system log. If SERVICE is set to the name of an external service, the log is considered to be a service log.

MESSAGE	Log message. The log message may contain several lines. The line separator is the “/n” (newline) character or the “µ” character.
EXTRA	Log extra information. This textual information can be used to separate some information from the message itself. The log extra may contain several lines. The line separator is the “/n” (newline) character or the “µ” character
TYPE	Log record type. This can take the value “INFO” for an information record, “WARNING” for a warning record and ”ERROR” for an error record. The default is “INFO”.
LEVEL	Log level. This can be a value from 1 to 6. This value is compared with the current log level and the log record is written only if the given level is less or equal to the current log level. If LEVEL is set to “0”, the log record is not written. The default is “1”.
IDENT	Identifier. This is a free text that identifies a user entity. For example, for the system log, the identifier is the NIRVA session identifier. If this parameter is not provided, NIRVA uses the session identifier.
SOURCE	Log record source. This is a free text that can be used to give information about the origin of the log record (who has written the record ?). If this parameter is not provided or is blank, NIRVA uses the values “client”, “browser”, “procedure”, “service” or “system” following the origin of the LOG WRITE command.

MAIL class

The MAIL class provides a command for sending an email to an external SMTP server. The good and simple SMTP server “James” has been tested successfully with NIRVA.

SEND

MAIL:SEND

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sends an email with eventual file attachments.

Permissions

MAIL_SEND

Parameters

FROM	Sender email. If this parameter is not provided, NIRVA uses the address USER@MAILHOST where USER and MAILHOST are other command parameters or default values from NIRVA configuration.
TO	Recipient addresses. This can be a set of addresses separated by a comma character (‘,’). This parameter is mandatory.
CC	Copy addresses. This can be a set of addresses separated by a comma character (‘,’).
BCC	Hidden copy addresses. This can be a set of addresses separated by a comma character (‘,’).
MAILHOST	Address of the SMTP server. If this parameter is not given, NIRVA uses the default one from the configuration.
AUTH	Flag for using authentication. If this parameter is set to “YES”, NIRVA will try to authenticate the user to the SMTP server. The default value is “YES”. If the SMTP server has been configured with authentication and the AUTH parameter is “NO”, the command will fail.
USER	User name for authentication to the SMTP server. If this parameter is not provided, NIRVA tries to use the first part of the FROM parameter (before the @ character) as user name. If the FROM parameter is not given or if the USER parameter is set to “NV_DEFAULT”, NIRVA uses the default mail user from the configuration. This parameter has meaning only if the AUTH parameter has been set to “YES” (default).
PASSWORD	Password for authentication to the SMTP server. If this parameter is not provided, NIRVA uses the default mail user from the configuration. This parameter has meaning only if the AUTH parameter has been set to “YES” (default).
TIMEOUT	Connection time out in seconds for the SMTP server. Default is 30 seconds.
SUBJECT	Message subject.
BODY	Message body.
MIME	Mime type of the body. The default is text/plain. This can be used to send html body, at this time the mime type must be set to “text/html”. The encoding can also be added. For example “text/html; charset=iso-8859-1”.
DEBUG	If this parameter is set to “YES”, NIRVA sends debugging information in the console if NIRVA has been ran in console mode.
WITH_ATTACHMENT	This parameter must be set to “YES” if there is only one file attachment. For multiple attachments, it must be set to “MULTIPLE_INLINE” or “MULTIPLE_TABLE”. Default is "NO".

ATTACHMENT	This parameter is only used when WITH_ATTACHMENT parameter is set to “YES”. Name of a file object containing the attachment. The associated file name extension is used to set the mime type when the MIME parameter is not provided. For example, if extension is “.pdf”, the mime type will be set to “application/pdf”.
ATTACHMENT_MIME	This parameter is only used when WITH_ATTACHMENT parameter is set to “YES”. Mime type of the optional attachment.
ATTACHMENT_NAME	This parameter is only used when WITH_ATTACHMENT parameter is set to “YES”. Display name of the optional attachment. If this parameter is not provided, NIRVA uses the object associated file name.
ATTACHMENT_ID	This parameter is only used when WITH_ATTACHMENT parameter is set to “YES”. Id of the attachment, for embedded images. If this parameter is specified, attachment will be considered as 'inline'.
ATTACHMENTS	This parameter is only used when WITH_ATTACHMENT parameter is set to “MULTIPLE_INLINE”. Name of Nirva file objects to add as attachments in the email. The names must be separated by a semicolon character (“;”). Mime type and name of attachments will be computed from file extension and file name.
ATTACHMENT_TABLE	This parameter is only used when WITH_ATTACHMENT parameter is set to “MULTIPLE_TABLE”. Name of the Nirva table in the input container that configures the attachments. The table must have 4 columns: ATTACHMENT, MIME_TYPE, NAME and ID, in this order, corresponding to: ■ “ATTACHMENT” name of the Nirva File object in the input container. This is mandatory. ■ “MIME_TYPE” Mime type of the attachment. It is optional, and is computed from the file extension if not given. (application/<file ext>). ■ “NAME” name of the attachment. If it is not provided, NIRVA uses the object associated file name. ■ “ID” id of the attachment, for embedded images.

MQ class

The MQ class provides commands for IBM MQSeries queue management.

ISAVAILABLE

MQ:ISAVAILABLE

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

Tests if the MQ connector is available on the server. If the MQ connector is available, the output buffer is filled with the value “YES”, otherwise it’s set to “NO”.

The MQ connector may be un-available for the following reasons:

- NIRVA server started with the “–q” option.
- No NIRVA license for MQ connector.
- The MQ client is not installed properly on the computer.

Parameters

none.

MISC class

The MISC class provides miscellaneous useful commands

EXEC

MISC:EXEC

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	Yes	No

Description

This command executes an external program on the server side. For security raisons, it’s only available for commands sent from procedure or from service. If a client or browser application needs to use this command, it must do it via a procedure.

The executed program will run in the environment of the Nirva server with the access rights defined for the user who started Nirva.

The EXEC command is able to transmit any kind of parameter to the executable. In this way, the EXEC command is a great open door for implementing some external technology or object processing.

Parameters

PATH

Complete command line for the executable. If the PATH parameter contains a string encapsulated in ‘%’ characters, the string will be replaced by Nirva with the corresponding parameter name of the current command. For example if path is “`cp %source% %dest%`”, Nirva will search for parameters named “source” and “dest” on the EXEC command and will replace the “%source%” and %dest%” strings with the corresponding parameter value. The parameter value itself can be a Nirva variable. In this way, any kind of parameters can be sent to the executable command line.
If the command line must contain a real ‘%’ character, this one must be doubled.
The PATH parameter is mandatory. It’s limited to 8192 characters under UNIX (not limited under WINDOWS).

LOCK

Optional name of a lock object. If this parameter is provided, Nirva will first gain access to the given lock object before executing the external program and will release it after. This allows serializing launching of external programs that are not able to run in multiple instances at the same time.

TIME_OUT

Optional time out in seconds for the execution of the external program.
If this parameter is “-1”, Nirva will wait indefinitely until the external program has finished.
If the time out is “0”, Nirva will not wait for the external program to return.
Otherwise, Nirva will wait for the given number of seconds and will signal the time out condition if it occurs by an error.
The default value is “-1” (infinite wait).

RET

Name of the Nirva integer object on which Nirva will write the return code of the external program.
The default is “RETURN_CODE”.

RET_OK

This parameter is used for the checking of the return code. If the RET_OK parameter is not provided or is blank, Nirva doesn’t check the external program return code.
If this parameter is used, it must contain return codes that are considered as successful, each of them separated by a ‘;’ (semicolon) character. One can use the ‘<’ and ‘>’ characters to test several values. For example, if RET_OK is “`0;1;>10;<20`”, Nirva will consider that values 0,1,11 to 19 as correct return codes and that the other codes are bad.
If the external executable doesn’t return a correct code, Nirva will send an error message.

Objects created

RETURN_CODE

This is a Nirva integer object containing the return code. The name of this object can be changed with the RET parameter.

GET_CRASH

MISC:GET_CRASH

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the crash condition flag in the output buffer. If the value returned is “YES”, that means that the last time the NIRVA server has been stopped, it was a hard stop (power off or hard kill of the process).

This command is useful during service initialization. A service can request the crash condition flag and then do its own cleanup.

Parameters

None

HASH

MISC:HASH

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns in the output buffer the computed MD5 hash value of a given input string or file. The return value is 32 characters length and contains only the characters “01234567890ABCDEF”.

Parameters

STRING	String to convert to MD5 hash value.
FILENAME	A filename can be given instead of a string. At this time, the returned value is the computed MD5 string for the file.

INFOEX

MISC:INFOEX

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some server information. This is the information found in the INFO section of the server description file.

The INFOEX command returns information in a NIRVA indexed string list object named “SYSTEM_INFO” in the output container.

Parameters

None

Objects created

SYSTEM_INFO	This is a Nirva indexed string list object corresponding to the pairs found in the INFO section of the server description file. If there is no description file for the server, the SYSTEM_INFO object is created but is empty.

NOP

MISC:NOP

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command does nothing but it can be used for resetting the timeout of a session and / or to run a procedure by using the general NV_PROC parameter.

The NOP command is often used from web applications that directly connect to the Nirva server.

Parameters

none

SLEEP

MISC:SLEEP

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command sleeps the session for the given time. While sleeping, the session does nothing and doesn't take any machine time.

The SLEEP command can be used for synchronizing some tasks.

Parameters

TIME Number of milliseconds to sleep. The default value is “0” (no sleep).

SEQUENCE

MISC:SEQUENCE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

Get a one or a series of sequential numbers (integers). A sequence is defined by its name and the last generated sequence number is persistent (saved in Nirva registry).

The minimum value for a sequence is 1.

A sequence can be defined at system or application level.

Parameters

NAME	Sequence name. The sequence name is case insensitive. It should only contain alphabetic and numeric characters and the "_" (underscore) character. This parameter is mandatory.
MAX	Maximum value of the sequence (from 1 to 2 000 000 000). When the sequence reaches the maximum value, it restarts from 1. The default value is 2 000 000 000.
NUM	Number of integers to return. The default value is 1. The maximum value for NUM is the value of MAX.
RESET	Resets the sequence to the given value (from 1 to 2 000 000 000). If the sequence doesn't exist and the RESET parameter is not given, the sequence starts at 1.
MIN_DIGITS	Minimum number of digits. Allows left leading 0s to be applied until the given number of digits. If this parameter is not provided, there are no leadings 0s.
ERROR_IF_NOT_EXIST	If set to "YES", the command returns an error if the sequence doesn't exist. Otherwise the command creates the sequence.
SYSTEM	If set to "YES", the command uses a system level sequence. Otherwise it uses an application level sequence.
RESULT	Name of the returned stringlist object. The default is "RESULT".

Objects created

RESULT	This is a Nirva stringlist object containing the requested sequence numbers. The name of this object can be changed with the RESULT parameter.

UNIQ

MISC:UNIQ

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns in the output buffer a unique string. The return value is 32 characters length and contains only the characters "01234567890ABCDEF".

The string is guaranty to be unique at system level. In fact, this string is computed with a combination of the computer name, the process id, a unique number stored in the registry at each call and a value coming from the random generator.

Parameters

None

OBJECT class

The OBJECT class provides an important set of commands to manipulate the Nirva container objects.

CLEAR_ALL

OBJECT:CLEAR_ALL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes all objects from the input container.

Parameters

None.

COPY

OBJECT:COPY

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command copies an object from the input to the output container. Both source and destination objects must exist and must be of the same type.

Parameters

SNAME	Source object name. The object must exist on the input container. A file object must have a file name attached to it.
NAME	Destination object name. The object must exist on the output container and must have the same type than the source object. A file object must have a file name attached to it.
OFFSET	Offset. This parameter is used for file and binary objects. It gives the offset in bytes of the source object from which the copy will start. The default value is ”0” (beginning of file or binary data).
NUM_BYTES	Number of bytes to copy. This parameter is used for file and binary objects. It gives the number of bytes of the source object to copy. The default value is ”-1” (until the end of file or binary data).

CREATE

OBJECT:CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command creates a new object in the input container.

Parameters

NAME	Object name. The object name is case insensitive and cannot contain any of the ‘/’, ‘\’, ‘.’ or space characters. The object name cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. The default value for this parameter is “NV_OBJ_DEFAULT_NAME”.
TYPE	Object type. This parameter is mandatory. The object type can take the following values: ■ “BOOLEAN” for a boolean object.

- “INTEGER” for an integer object.
- “STRING” for a string object.
- “STRINGLIST” for a string list object.
- “INDSTRINGLIST” for an indexed string list object.
- “TABLE” for a table object.
- “FILE” for a file object.
- “BINARY” for a binary object.

REPLACE Replace mode. If this parameter is set to “YES”, Nirva replaces an object with the same name if it exists. Otherwise, Nirva sends an error message if the object already exist. The default is “NO”.

CASE_SENSITIVE This parameter is only used with indexed string list object type. If it’s set to “YES”, the indexed string list keys will be case sensitive. The default is “NO”.

PERSIST This parameter gives the persistent value for file objects. It’s only used if the object to create is a file object.
The parameter can take value “0” for temporary files, “-1” for persistent files and any other value (in seconds) for cached files.
Persistent files are automatically created in the application file directory while other kinds of files are created in the application work directory.
The default value is “0” (not persistent).

FILENAME This parameter gives the file name of the file object.
The FILENAME parameter is only used if the object to create is a file object.
If the command comes from client or browser, the FILENAME parameter is automatically reset to “” (blank).
If this parameter is not provided, Nirva creates itself a file name following information of the EXTENSION, PREFIX, SUFFIX and DIRECTORY parameters.

DIRECTORY This is the name of a directory where Nirva will put the local file if the object to create is a file object and the FILENAME parameter is not given.
For information, Nirva generates an error message when trying to create a persistent file in the application work directory.
If this parameter is not provided or is blank, Nirva uses the application work directory for temporary and cached files and the application file directory for persistent files.

EXTENSION This is the file extension to use. The EXTENSION parameter is only used if the object to create is a file object. If this parameter is not provided or is blank, Nirva uses “.obj” for persistent files and “.tmp” for temporary and cached files. If the point character is omitted in the EXTENSION parameter, Nirva adds it.

PREFIX This is the file prefix to use. The PREFIX parameter is only used if the object to create is a file object.

The prefix, if provided is used by Nirva to automatically create the server file name.

SUFFIX

This is the file suffix to use. The SUFFIX parameter is only used if the object to create is a file object.
The suffix, if provided is used by Nirva to automatically create the server file name.

CREATE

Create file option. This parameter has meaning only when the object is a file object and the FILENAME parameter has been given. By default, Nirva then physically creates the file if it doesn't exist. When this parameter is set to "NO", the object is attached to the given file but the file itself is not created if it doesn't exist. The default is "YES" (create the file if it doesn't exist).

VALUE

Object value. Allows to directly setting a value for objects BOOLEAN, INTEGER, STRING, STRINGLIST, INDSTRINGLIST and TABLE.
For a BOOLEAN object this parameter can take values "TRUE" or "FALSE". If another value is given, Nirva uses "FALSE" as object value.
For an INTEGER object this is directly the object value. This must be a numeric value.
For a STRING object this is directly the object value.
For a STRINGLIST object, this is the values controlled by the SEPARATOR parameter.
For an INDSTRINGLIST object, this is the values controlled by the KEYSEP and VALSEP parameters.
For a TABLE object, this is the values controlled by the ROWSEP, COLSEP and LINESEP parameters.

SEPARATOR

Multi string separator for STRINGLIST object value. This parameter has meaning only when a VALUE parameter is given. It defines a character or string separator in the given value. Then NIRVA will add as many strings as the number of separators found (plus one). For example, if the value is "`v1;v2;v3`" and the separator is "`;`", NIRVA will add the strings "`v1`", "`v2`" and "`v3`" to the string list object.
The SEPARATOR parameter may also take special values "NVLINE", "NVTAB" or "NVSPACE" for respectively line (or pair CR/LF) separator, tabulation separator or space separator. In the case of a space separator, several successive spaces are assimilated to a single separator.

KEYSEP

Key separator for INDSTRINGLIST object value. This parameter has meaning only when a VALUE parameter is given. It defines the key separator in the given VALUE parameter. It can be any string. The default is the ';' character (semicolon).

VALSEP

Value separator for INDSTRINGLIST object value. This parameter has meaning only when a VALUE parameter is given. It defines the value separator in the given VALUE parameter. It can be any string. The default is the '=' character.

ROWSEP

Row separator for TABLE object value. This parameter has meaning only when a VALUE parameter is given. It defines the string that separates the

	rows in the value. The default is the line feed character (\n).
COLSEP	Column separator for TABLE object value. This parameter has meaning only when a VALUE parameter is given. It defines the string that separates the columns in the value. The default is “;”.
LINESEP	Line separator for TABLE object value. This parameter has meaning only when a VALUE parameter is given. It defines the string that separates the lines for columns containing several lines in the value. The default is “|”.
COLUMNS	Name and descriptions of the columns for a table object. The names must be separated by a semicolon character (“;”). Each value is composed of 3 parts separated by a “:” character: NAME:DESCRIPTION:NUMERIC. NAME is the column name, it cannot contain any blank or special character (they will be removed without error if there are), DESCRIPTION is a free text describing the column and NUMERIC, when set to “Y” tells that the column is numeric. The DESCRIPTION and NUMERIC part are optional. Here are some valid entries: “COL1;COL2;COL3” or “COL1;COL2:My column 2:Y;COL3:My column 3”.

ENCODE

OBJECT:ENCODE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command changes the encoding of the requested object. It can be useful to convert for example a table object data that comes from a database and that is known to be “ISO-8859-1” encoded while NIRVA has been defined to internally encode objects as Unicode (“UTF-8”).

The ENCODE command works only on STRING, STRINGLIST, INDSTRINGLIST and TABLE objects. If attempt to use it with another object, the COMMAND doesn’t produce any error but does nothing.

Parameters

NAME	Object name.

SRC_ENCODING	Current encoding of the object. This can be “UTF-8”, “ISO-8859-1” or “INTERNAL”. If “INTERNAL” is used, then the object is considered to have the same encoding than the internal NIRVA encoding (“UTF-8” or “ISO-8859-1”) . The default value is “INTERNAL”.
DEST_ENCODING	Required new encoding of the object. This can be “UTF-8”, “ISO-8859-1” or “INTERNAL”. If “INTERNAL” is used, then the object will be converted to the same encoding than the internal NIRVA encoding (“UTF-8” or “ISO-8859-1”). If both SRC_ENCODING and DEST_ENCODING are the same, then the command does nothing. The default value is “INTERNAL”.
WITH_NAME	If this parameter is set to “YES”, then NIRVA also changes the encoding of the object name. The default is “NO”.

EXIST

OBJECT:EXIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns “YES” in the output buffer if the object exists on the input container and “NO” otherwise.

Parameters

NAME	Object name.

GET

OBJECT:GET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web	No	Yes	No

Description

This command is used from Nirva clients or web browsers to get an object from the server. The command also works for getting objects from one Nirva server to the other.

From a Nirva client, the object is placed into the local container. If the object to get is a file object and a local file object with the given name already exists, Nirva will use this local object to receive the file. In any other case, Nirva will remove the local object of the given name if it exists. If the local object doesn't exist, Nirva creates it.

From a Nirva server, the object is placed into the input container. Nirva will remove the local object of the given name if it exists. If the local object doesn't exist, Nirva creates it.

If this command is used from a web browser (or XML, SOAP, Web service and MQ connectors), the object to get can be only a file or binary object.

Parameters

NAME

Server (or distant server) object name. The object name is case insensitive. The object name cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. The default value for this parameter is "NV_OBJ_DEFAULT_NAME".

LNAME

Local (or local server) object name.
This parameter has meaning only if the command comes from a client or from a server.
This is the name of the local object in client container (nvc library) or in the input container when the command is used between 2 servers. The object name is case insensitive.
If this parameter is not provided, Nirva assumes that the local name is the same than the server name given by the "NAME" parameter.

CASE_SENSITIVE

This parameter is only used with indexed string list object type. If it's set to "YES", the indexed string list keys will be case sensitive. The default is "NO".

FILENAME

This parameter gives the local file name of the file object.
The FILENAME parameter is only used if the object received is a file object.
When the command comes from a web browser, the FILENAME parameter has a meaning only if the ATTACHEMENT or INLINE parameters have been set to yes. This allows setting the name of the file when downloading from a web browser.
When the command comes from a NIRVA client, and this parameter is not provided, Nirva creates itself a file name following information of the EXTENSION, PREFIX, SUFFIX and DIRECTORY parameters or uses the file name of an eventual existing local object of the given name (LNAME).
The FILENAME parameter is ignored for GET commands from server to server.

PERSIST

This parameter allows changing the persistent value for the local file object.
This parameter has meaning only if the command comes from a client or from a server.
The PERSIST parameter is only used if the object received is a file object.

	For a client command, the parameter can take value "0" for temporary files, or "-1" for persistent files. Cached files are not authorized on the local container. If this parameter is not provided and the local object has just been created by the command, the default value is "-1" (persistent) except if the FILENAME parameter is not given or is blank. At this time, the default value is "0" (temporary). For a server command, the parameter can take value "0" for temporary files, or "-1" for persistent files and any other positive value (in minutes) for cached files. If this parameter is not provided the default value is "0" (temporary file).
DIRECTORY	This is the name of a directory where Nirva will put the local file if the received object is a file object and the FILENAME parameter is not given. This parameter has meaning only if the command comes from a client.
EXTENSION	This is the file extension to use. The EXTENSION parameter is only used if the received object is a file object and the FILENAME parameter is not given. If this parameter is not provided or is blank, Nirva uses ".obj" for persistent files and ".tmp" for temporary and cached files. If the point character is omitted in the EXTENSION parameter, Nirva adds it. This parameter has meaning only if the command comes from a client.
PREFIX	This is the file prefix to use. The PREFIX parameter is only used if the received object is a file object and the FILENAME parameter is not given. The prefix, if provided is used by Nirva to automatically create the local file name. This parameter has meaning only if the command comes from a client or from a server.
SUFFIX	This is the file suffix to use. The SUFFIX parameter is only used if the received object is a file object and the FILENAME parameter is not given. The suffix, if provided is used by Nirva to automatically create the local file name. This parameter has meaning only if the command comes from a client or a server.
REMOVE	Remove option. This parameter has meaning only if the command comes from a client. If this parameter is set to "YES". The server object is removed after being sent to the client. If the file has to be removed and is a cached file, it goes under the control of the Nirva server cache mechanism. The default is "NO" (no removing).
NOCACHE	No cache option. This parameter has meaning only if the command comes from a web browser. In this case and if NOCACHE is set to "YES", Nirva will add necessary code to the HTML output in order for the browser to not cache the object. When used to get PDF data flow from NIRVA with an internet explorer client, the NOCACHE parameter must be set to "NO". The default is "YES" (no cache).

ATTACHMENT	Attachment option. This parameter has meaning only if the command comes from a web browser. In this case and if ATTACHMENT is set to "YES", Nirva will add necessary code to the HTML output in order for the browser to download the file instead of displaying it. The parameters INLINE and ATTACHMENT are exclusives, only one of them can be used in the command. The default is "NO" (the file will be displayed by the browser if it can display it).
INLINE	In line option. This parameter has meaning only if the command comes from a web browser. In this case and if INLINE is set to "YES", Nirva will add necessary code to the HTML output in order for the browser to first try to display the object and to download it if it cannot display. The parameters INLINE and ATTACHMENT are exclusives, only one of them can be used in the command. The default is "NO".
MIME	Mime option. This parameter has meaning only if the command comes from a web browser. It allows setting the mime type of the file object. When this parameter is not provided, NIRVA tries to set itself the mime type following the file extension.
ASCII	If this parameter is set to "YES" and the object is a file object, NIRVA adjusts the received file on the local side with the correct carriage return / line feed pair if necessary, following the local platform (WINDOWS or UNIX). This feature doesn't work from a web browser.

GET_NAME

OBJECT:GET_NAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the name of an object of the input container given an object index.

In conjunction with the OBJECT GET_NUM commands, this command can be used to enumerate the container object names.

Parameters

INDEX This is a number starting at one and having the maximum value equal to the number of objects returned by the OBJECT GET_NUM command.

GET_NUM

OBJECT:GET_NUM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the number of objects of the input container.

Parameters

none

GET_TYPE

OBJECT:GET_TYPE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the type of an object of the input container given an object name.

The result is in the output buffer. It can be BOOLEAN, INTEGER, STRING, STRINGLIST, INDSTRINGLIST, TABLE, FILE, BINARY or UNKNOWN.

Parameters

NAME Object name. The object must exist on the input container.

MOVE

OBJECT:MOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command moves an object from input to output container. Internally, the object remains the same and only its reference is moved from the input container to the output container.

Parameters

NAME Object name. The object must exist on the input container.

REPLACE Replace mode. If set to “NO” and if the object already exists in the output container, the command fails. Otherwise, the eventual existing object is replaced.

REMOVE

OBJECT:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes the given object from the input container.

Parameters

NAME Object name.

SEND

OBJECT:SEND

Source	Use Input Container	Use Output Container	Use Output buffer
Client	Yes	No	No

Description

This command is used from Nirva clients to send an object to the server. The command also works for sending objects from one Nirva server to the other.

From a client, the object is placed into the session input container. If an object of the given name already exists in the input container, Nirva first removes it and then replaces it with the new one.

Form a server to another NIRA server, the object is taken from the local output container and placed into the distant session input container. If an object of the given name already exists in the distant input container, Nirva first removes it and then replaces it with the new one.

Parameters

NAME	Server (or distant server) object name. The object name is case insensitive. The object name cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. The default value for this parameter is “NV_OBJ_DEFAULT_NAME”.
LNAME	Local (or local server) object name. This is the name of the local object in client container (nvc library) when the command is sent from a client or the name of the local object in the output container when the command is sent from a local server to another server. The object name is case insensitive. The object name cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. If this parameter is not provided, Nirva assumes that the local name is the same than the server name given by the “NAME” parameter.
CASE_SENSITIVE	This parameter is only used with indexed string list object type. If it’s set to “YES”, the indexed string list keys will be case sensitive. The default is “NO”.
PERSIST	This parameter gives the persistent value for file objects. It’s only used if the object sent is a file object. The parameter can take value “0” for temporary files, “-1” for persistent files and any other value (in seconds) for cached files. Persistent files are automatically created in the application file directory while other kinds of files are created in the application work directory. The default value is “0” (not persistent).
EXTENSION	This is the file extension to use. The EXTENSION parameter is only used if the object sent is a file object. If this parameter is not provided or is blank, Nirva uses “.obj” for persistent files and “.tmp” for temporary and cached

	files. If the point character is omitted in the EXTENSION parameter, Nirva adds it.
PREFIX	This is the file prefix to use. The PREFIX parameter is only used if the object sent is a file object. The prefix, if provided is used by Nirva to automatically create the server file name.
SUFFIX	This is the file suffix to use. The SUFFIX parameter is only used if the object sent is a file object. The suffix, if provided is used by Nirva to automatically create the server file name.
ASCII	If this parameter is set to "YES" and the object is a file object, NIRVA adjusts the received file on the server side with the correct carriage return / line feed pair if necessary, following the server platform (WINDOWS or UNIX).

SET_NAME

OBJECT:SET_NAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command allows changing the name of an existing object. The given object must exist.

If an object having the same name that the new object name exists in the input container, the command fails.

Parameters

NAME	Object name. The object must exist on the input container.
NEW_NAME	New object name. If an object having the same name that the new object name exists in the input container, the command fails.

BOOLEAN_GET_VALUE

OBJECT:BOOLEAN_GET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the value of a boolean object in the output buffer. The object must exist and must be of boolean type. The return value is “TRUE” or “FALSE”.

Parameters

NAME Object name.

BOOLEAN_SET_VALUE

OBJECT:BOOLEAN_SET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the value of a boolean object. The object must exist and must be of boolean type.

Parameters

NAME Object name.

VALUE Object value. This parameter can take values “TRUE” or “FALSE”. If another value is given, Nirva uses “FALSE” as object value.
The default is “FALSE”.

INTEGER_GET_VALUE

OBJECT:INTEGER_GET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the value of an integer object in the output buffer (as a string). The object must exist and must be of integer type.

Parameters

NAME Object name.

INTEGER_SET_VALUE

OBJECT:INTEGER_SET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the value of an integer object. The object must exist and must be of integer type.

Parameters

NAME Object name.

VALUE Object value. This parameter must be a numeric value.

STRING_GET_VALUE

OBJECT:STRING_GET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the value of a string object in the output buffer. The object must exist and must be of string type.

Parameters

NAME	Object name.

STRING_SET_VALUE

OBJECT:STRING_SET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the value of a string object. The object must exist and must be of string type.

Parameters

NAME	Object name.
VALUE	Object value.

STRINGLIST_ADD_STRINGLIST

OBJECT:STRINGLIST_ADD_STRINGLIST

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command adds all entries of a string list object to another string list object. Both objects must exist and must be of string list type.

Parameters

SNAME Source object name.

NAME Destination object name. All the source object entries are added to the destination object.

USE_OUT_CONTAINER If set to “YES” the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.

STRINGLIST_GET_SIZE

OBJECT:STRINGLIST_GET_SIZE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the total number of entries of a string list object in the output buffer. The object must exist and must be of string list type.

Parameters

NAME Object name.

STRINGLIST_GET_VALUE

OBJECT:STRINGLIST_GET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the value of a string list object entry in the output buffer. The object must exist and must be of string list type.

Parameters

NAME Object name.

INDEX Index of the entry. This index starts at 1. The INDEX value can also take the value “FIRST” for retrieving the first entry value or “LAST” for retrieving the last entry value.
The default value is “LAST”.

STRINGLIST_GET_VALUES

OBJECT:STRINGLIST_GET_VALUES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command all or a range of values of a string list object in the output buffer.

Parameters

NAME Object name.

WHAT Range of index entries to get. If not given or blank or set to the value “NV_ALL”, the command retrieves all entries. If set to a value “m-n”, the command retrieves the entries from index m to index n (included). If a single

	index value is given, the command retrieves only the corresponding value so the command is then similar to the STRINGLIST_GET_VALUE command.
SEPARATOR	Separator for the output string. The default is ";".

STRINGLIST_INSERT

OBJECT:STRINGLIST_INSERT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command inserts a new entry in a string list. The entry is inserted before the given index. The object must exist and must be of string list type.

Parameters

NAME	Object name.
VALUE	New object entry value.
INDEX	Index of the entry where the insertion must occurs. The new entry is inserted before the given index. This index starts at 1. If the INDEX parameter is not provided or is blank, Nirva just adds the entry value to the end of the list. This is the default.
SEPARATOR	Multi string separator. This parameter has meaning only when the INDEX parameter is not given or is blank. It defines a character or string separator in the given value. Then NIRVA will add as many strings as the number of separators found (plus one). For example, if the value is "`v1;v2;v3`" and the separator is "`;`", NIRVA will add the strings "`v1`", "`v2`" and "`v3`" to the string list object. The SEPARATOR parameter may also take special values "NVLINE", "NVTAB" or "NVSPACE" for respectively line (or pair CR/LF) separator, tabulation separator or space separator. In the case of a space separator, several successive spaces are assimilated to a single separator.

STRINGLIST_REMOVE

OBJECT:STRINGLIST_REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes one or all entries of a string list object. The object must exist and must be of string list type.

Parameters

NAME Object name.

INDEX Index of the entry. This index starts at 1. The INDEX value can also take the value “FIRST” for removing the first entry, “LAST” for removing the last entry or “ALL” for removing all entries.
The default value is “ALL”.

STRINGLIST_SEARCH

OBJECT:STRINGLIST_SEARCH

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command searches for a given value in the string list and returns the found indexes in the output buffer. They are separated with the ‘;’ character.

Parameters

NAME Object name.

VALUE	Value to search. If the string terminates with the * character, the command searches for all values starting with the string before the * character. For example if value is "t*", the command searches all values starting with "t"
CASE	Case sensitive search. This can be "YES" or "NO" (default).
FIRST_ONLY	If this parameter is set to "YES", the command retrieves only the first index of the found ones. If set to "NO, it retreives all the found indexes. The default is "YES".

STRINGLIST_SET_VALUE

OBJECT:STRINGLIST_SET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the value of an entry of a string list. The list entry is referenced by an index starting at 1. The object must exist and must be of string list type.

Parameters

NAME	Object name.
VALUE	Object entry value.
INDEX	Index of the entry. This index starts at 1. If the INDEX parameter is not provided or is blank, Nirva just adds the entry value to the end of the list. This feature can be used to add a new entry to the string list instead of using the INSERT command.
SEPARATOR	Multi string separator. This parameter has meaning only when the INDEX parameter is not given or is blank. It defines a character or string separator in the given value. Then NIRVA will add as many strings as the number of separators found (plus one). For example, if the value is "`v1;v2;v3`" and the separator is "`;`", NIRVA will add the strings "`v1`", "`v2`" and "`v3`" to the string list object. The SEPARATOR parameter may also take special values "NVLINE", "NVTAB" or "NVSPACE" for respectively line (or pair CR/LF) separator, tabulation separator or space separator. In the case of a space separator, several successive spaces are assimilated to a single separator.

STRINGLIST_SORT

OBJECT:STRINGLIST_SORT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sorts the entries of a string list object. The object must exist and must be of string list type.

The sort is in ascending order and can be numeric if the NUMERIC option has been chosen.

Parameters

NAME Object name.

NUMERIC Numeric option. If this parameter is set to “YES”, Nirva considers that the string list contains numeric value and tries to sort them accordingly.

NATURAL Natural option. If this parameter is set to “YES”, the command sorts items in a “natural” way. This is useful to list items containing both alphabetic characters and numbers. This option has no effect when NUMERIC is set to “YES”.

LOCALE Locale name. Internally the command uses unicode to sort strings so the sorting should be ok. However, if the sorting is not correct in your language you can set the locale name using this parameter. Ex: “FRENCH”.

STRINGLIST_SWAP

OBJECT:STRINGLIST_SWAP

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command swaps 2 entries of a string list object. The object must exist and must be of string list type.

Parameters

NAME	Object name.
INDEX1	Index of the first entry to swap. This index starts at 1. This parameter is mandatory.
INDEX2	Index of the second entry to swap. This index starts at 1. This parameter is mandatory.

INDSTRINGLIST_ADD_INDSTRINGLIST

OBJECT:INDSTRINGLIST_ADD_INDSTRINGLIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command adds all entries of an indexed string list object to another indexed string list object. Both objects must exist and must be of indexed string list type.

The destination entries having the same key than the source entries are replaced by the source entries.

Parameters

SNAME	Source object name.
NAME	Destination object name. All the source object entries are added to the destination object.
USE_OUT_CONTAINER	If set to “YES” the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.

INDSTRINGLIST_GET_KEY

OBJECT:INDSTRINGLIST_GET_KEY

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns a KEY corresponding to an index. It allows enumerating the keys of the indexed string list object.

Parameters

NAME Object name.

INDEX Key index. The index starts at 1 and cannot be greater that the number of entries of the indexed string list object.

INDSTRINGLIST_GET_SIZE

OBJECT:INDSTRINGLIST_GET_SIZE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the total number of entries of an indexed string list object in the output buffer. The object must exist and must be of indexed string list type.

Parameters

NAME Object name.

INDSTRINGLIST_GET_VALUE

OBJECT:INDSTRINGLIST_GET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the value of an indexed string list object entry in the output buffer. The object must exist and must be of indexed string list type.

Parameters

NAME Object name.

KEY Entry key. This is a string that indexes the entry. The KEY can contain space characters.

INDEX Key index. This parameter can be given to directly access the value by index instead of the key. The index starts at 1 and cannot be greater that the number of entries of the indexed string list object.
However, accessing an indexed string list by the way of a numeric index is slower than accessing by key.

INDSTRINGLIST_GET_VALUES

OBJECT:INDSTRINGLIST_GET_VALUES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command writes the complete content of an indexed string list object into the output buffer or in session variables.

Parameters

NAME Object name.

MODE Can be set to VARIABLES (default) or STRING. If set to VARIABLES, the command creates as many variables as the number of keys. The name of the variables is the name of the key eventually prefixed with a value given in

	PREFIX parameter. If the MODE is set to STRING, the command returns the content in the output buffer using KEYSEP and VALSEP parameters respectively as key and value separators.
KEYSEP	Key separator. This can be any string. The default is “;”. Used only if the mode has been set to STRING.
VALSEP	Value separator. This can be any string. The default is “=”. Used only if the mode has been set to STRING.
PREFIX	Prefix of created variables when the mode is VARIABLES. Default is no prefix.

INDSTRINGLIST_KEY_EXIST

OBJECT:INDSTRINGLIST_KEY_EXIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns “YES” in the output buffer if the given key exists in the indexed string list object and “NO” otherwise.

Parameters

NAME	Object name.
KEY	Entry key. This is a string that indexes the entry. The KEY can contain space characters.

INDSTRINGLIST_REMOVE

OBJECT:INDSTRINGLIST_REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes one or all entries of an indexed string list object. The object must exist and must be of indexed string list type.

Parameters

NAME	Object name.
KEY	Entry key. This is a string that indexes the entry. The KEY can contain space characters. If this parameter is not provided or is blank, Nirva removes all the entries of the indexed string list object.

INDSTRINGLIST_SET_VALUE

OBJECT:INDSTRINGLIST_SET_VALUE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the value of an entry of an indexed string list. The list entry is indexed by a string key. The object must exist and must be of indexed string list type.

If the entry already exists, Nirva sets its new value. If the entry doesn't exist, Nirva adds it to the object.

The command also allows setting several keys from a single string. For that, the KEY parameter must me empty and the VALUE parameter must contain key and value separators. For example, the value "`key1=val1;key2=val2`" will add the keys "`key1`" and "`key2`" to the object with respective values of "`val1`" and "`val2`". The default key and value separators (';' and '=' characters) can be changed by the command.

Parameters

NAME	Object name.
VALUE	Object entry value.
KEY	Entry key. This is a string that indexes the entry. The KEY can contain space characters.
KEYSEP	Key separator. This parameter has meaning only when the KEY parameter is empty. It defines the key separator in the given VALUE parameter. It can be any string. The default is the ';' character (semicolon).

VALSEP Value separator. This parameter has meaning only when the KEY parameter is empty. It defines the value separator in the given VALUE parameter. It can be any string. The default is the '=' character (semicolon).

TABLE_ADD_COLUMNS

OBJECT:TABLE_ADD_COLUMNS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command adds several columns to a table object. The columns are inserted after the last column.

Parameters

NAME Table object name.

COLUMNS Name and descriptions of the columns to add. The names must be separated by a semicolon character (";"). Each value is composed of 3 parts separated by a ":" character: NAME:DESCRIPTION:NUMERIC. NAME is the column name, it cannot contain any blank or special character (they will be removed without error if there are), DESCRIPTION is a free text describing the column and NUMERIC, when set to "Y" tells that the column is numeric. The DESCRIPTION and NUMERIC part are optional. Here are some valid entries: "COL1;COL2;COL3" or "COL1;COL2:My column 2:Y;COL3:My column 3".

TABLE_ADD_ROWS

OBJECT:TABLE_ADD_ROWS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command adds several rows with data to a table object.

If there is a primary key defined for the table and there are some duplicates on the primary key, the command fails.

Parameters

NAME	Object name.
DATA	Row data. The row separator is the line feed character (\n), the column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the ROWSEP, COLSEP and LINESEP parameters. The column data must be in the same order than the column definition in the table. The last columns can be omitted if they are blank. If a cell contains an empty line, Nirva does not write this empty line. In order to force writing an empty line, the line value must be set to “NV_EMPTY”.
ROWSEP	Row separator. This defines the string that separates the rows in the data. The default is the line feed character (\n).
COLSEP	Column separator. This defines the string that separates the columns in the data. The default is “;”.
LINESEP	Line separator. This defines the string that separates the lines for columns containing several lines in the data. The default is “|”.

TABLE_ADD_TABLE

OBJECT:TABLE_ADD_TABLE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command adds all rows of a table object to another table object. Both objects must exist and must have the same number of columns.

3 modes are available: Normal, Replace, No replace. These modes changes the way the command manages duplicates when the destination table has a primary column. In normal mode, any duplicate

generates an error. In Replace mode, the new row having a primary key that already exists replaces the old row. In no replace mode, the new row having a primary key that already exists is skipped.

Parameters

SNAME	Source object name.
NAME	Destination object name. All the source object rows are added to the destination object.
MODE	Controls the way the command manages the duplicate when a primary key has been defined (see description). The possible values are: NORMAL, REPLACE and NO_REPLACE. The default is NORMAL.
USE_OUT_CONTAINER	If set to "YES" the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.

TABLE_CLEAR

OBJECT:TABLE_CLEAR

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command clears the entire table including its column definition, description, and rows. The object must exist and must be of table type.

Parameters

NAME	Object name.

TABLE_CLEAR_CELL

OBJECT:TABLE_CLEAR_CELL

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command clears a cell of a table object.

Parameters

NAME Object name.

ROW Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

COL Column index of the cell. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_CLEAR_COLUMN

OBJECT:TABLE_CLEAR_COLUMN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command clears a single column from a table object. Clearing a column means to remove any data from the column so the column still exists but is empty.

Parameters

NAME Object name.

COL Column index to clear. The index starts at 1.

COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_CLEAR_DATA

OBJECT:TABLE_CLEAR_DATA

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes all table rows. The object must exist and must be of table type.

Parameters

NAME	Object name.

TABLE_CLEAR_ROW

OBJECT:TABLE_CLEAR_ROW

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command clears a single row from a table object. Clearing a row means to remove any data from the row so the row still exists but is empty.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

TABLE_CLEAR_ROWS

OBJECT:TABLE_CLEAR_ROWS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command clears a selection of rows from a table object. Clearing a row means to remove any data from the row so the row still exists but is empty.

The selection is made by giving a query as parameter.

Parameters

NAME Object name.

QUERY Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query.
The default value is “*” (search all rows).

TABLE_CREATE_FROM

OBJECT:TABLE_CREATE_FROM

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command clears a table object and re-creates it with the column information of a source table object. Both objects must exist and must be of table type.

This command doesn't copy rows from the source object. For that, one can use the COPY command of the OBJECT class.

Parameters

SNAME Source object name.

NAME Destination object name. The destination object is a table. The command clears this table and initializes it with the table and column information of the source object.
The destination object has 0 rows after the completion of this command.

USE_OUT_CONTAINER If set to "YES" the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.

TABLE_EXPORT

OBJECT:TABLE_EXPORT

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	No	No

Description

This command exports data rows of a table object to an ASCII file.

This command is not available from client or browser source.

Parameters

NAME Object name.

FILENAME Pathname of the file to export rows to.
Here is the description of the file format:

	Each line corresponds to 1 row. The column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the COLSEP and LINESEP parameters. Each line is limited to 1024 characters (may be changed by the LINE_LENGTH parameter). If a row is longer than 1024 characters, the last line character is '\' (backslash) and the row content continues on the next line. The column data has the same order than the column definition in the table. The backslash and line feed characters are controlled by the way of WITH_LF parameter.
COLSEP	Column separator. This defines the string that separates the columns in the export file. The default is “;”.
LINESEP	Line separator. This defines the string that separates the lines for columns containing several lines in the export file. The default is “|”.
LINE_LENGTH	This is the maximum line length of the export file. The default is 1024. The minimum value is 1024.
WITH_LF	When this parameter is set to YES, Nirva writes any line feed character as the string “\n” and any backslash characters as the string “\\”. When this parameter is set to NO, Nirva just replaces any line feed character with a space. In any case, the carriage return characters are removed. The default is NO.
WITH_BOM	When this parameter is set to “YES”, Nirva writes the utf8 BOM at the start of the export file. The value can also be set to “AUTO”. At this time Nirva writes the BOM only if the server has been started in unicode mode. The default is NO.

TABLE_FILTER_COLUMNS

OBJECT:TABLE_FILTER_COLUMNS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command allows keeping only specified columns of a table. The other ones are removed. The name and order of the columns can be changed.

Parameters

NAME	Object name.
COLUMNS	Columns to keep. When not provided, the command has no effect. The COLUMNS parameter is a list of values separated by a semicolon character. Each value itself has the format *NewColName*:*ColName* where *ColName* is the name of the column and *NewColName* is its new name. If the column name doesn't have to be changed, the value is simply *ColName*. The columns will be delivered in given order.

TABLE_GET_CELL_LINE

OBJECT:TABLE_GET_CELL_LINE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns a cell line data in the output buffer.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. The default is 1 (First line).
DEFAULT	Default value to return if the line index is not valid.

TABLE_GET_CELL

OBJECT:TABLE_GET_CELL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the complete cell data in the output buffer.

Parameters

NAME Object name.

ROW Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

COL Column index of the cell. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

LINESEP Line separator. This can be a string (not only a single character) The default is “;”.

TABLE_GET_CELL_NUM_LINES

OBJECT:TABLE_GET_CELL_NUM_LINES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the number of lines of a cell in the output buffer.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_GET_COLUMN

OBJECT:TABLE_GET_COLUMN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command gets the cells values for a given column and all rows. The result is in the output buffer.

Parameters

NAME	Object name.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
ROWSEP	Row separator. This can be a string (not only a single character). The default is “;”.
LINESEP	Line separator. This can be a string (not only a single character). The default is “|”.

TABLE_GET_COLUMN_DESCRIPTION

OBJECT:TABLE_GET_COLUMN_DESCRIPTION

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the column description corresponding to the given column index in the output buffer.

Parameters

NAME Object name.

COL Column index. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_GET_COLUMN_INDEX

OBJECT:TABLE_GET_COLUMN_INDEX

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the column index given a column name. The result is written in the output buffer.

Parameters

NAME Object name.

COLNAME Column name.

TABLE_GET_COLUMN_NAME

OBJECT:TABLE_GET_COLUMN_NAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the column name corresponding to the given column index in the output buffer.

Parameters

NAME Object name.

COL Column index. The index starts at 1.

TABLE_GET_COLUMN_NAMES

OBJECT:TABLE_GET_COLUMN_NAMES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the column names. All values are semicolon separated (;).

Parameters

NAME Object name.

TABLE_GET_DESCRIPTION

OBJECT:TABLE_GET_DESCRIPTION

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the description of a table object in the output buffer. The object must exist and must be of table type.

Parameters

NAME Object name.

TABLE_GET_NUM_COLUMNS

OBJECT:TABLE_GET_NUM_COLUMNS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the number of columns of a table object in the output buffer.

Parameters

NAME Object name.

TABLE_GET_NUM_ROWS

OBJECT:TABLE_GET_NUM_ROWS

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the number of rows of a table object in the output buffer.

Parameters

NAME	Object name.

TABLE_GET_PRIMARY_INFO

OBJECT:TABLE_GET_PRIMARY_INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command returns information about the primary column of a table object.

Parameters

NAME	Object name.

Objects created

PRIMARY_COLUMN	This is a string object that contains the name of the primary column. It’s empty if there is no primary column.
PRIMARY_COLUMN_CS	This is a boolean object set to TRUE if the primary column is case sensitive and to FALSE otherwise.

TABLE_GET_ROW

OBJECT:TABLE_GET_ROW

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	Yes

Description

This command extracts a single record from the table object and writes it into the output buffer or in session variables.

Parameters

NAME	Object name.
ROW	Index of the first row to get. The index starts at 1.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
MODE	Can be set to VARIABLES (default), STRING or INDSTRINGLIST. If set to VARIABLES, the command creates as many variables as the number of columns. The name of the variables is the name of the column eventually prefixed with a value given in PREFIX parameter. If the MODE is set to STRING, the command returns the row content in the output buffer. If the MODE is INDSTRINGLIST the command creates an indexed string list object and writes a value for each column name.
COLSEP	Column separator. The default is “;”. Used only if the mode has been set to STRING.
LINESEP	Line separator. The default is “|” if mode is STRING or INDSTRINGLIST and “;” if mode is VARIABLES.
PREFIX	Prefix of created variables when the mode is VARIABLES. Default is no prefix.
RESULT	Name of the resulting object when the mode is INDSTRINGLIST. The resulting object is an indexed string list object containing the requested row. The default is “RESULT”.

Objects created

RESULT	This is a Nirva indexed string list object that contains the requested row. The name of this object can be changed by using the RESULT parameter. This object is created only when the mode has been set to INDSTRINGLIST.

TABLE_GET_ROWS

OBJECT:TABLE_GET_ROWS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	Yes

Description

This command extracts some records from the table object and writes them to a new table object or a string object. The rows can be searched by a FIRST and LAST pair parameters or by a PAGE and PAGE_SIZE pair. The number of rows is written in the output buffer.

Parameters

NAME	Object name.
MODE	Output mode can be “TABLE” (default) or “STRING”. The command will create a corresponding output object type.
RESULT	Name of the resulting object. The resulting object is a table or string object containing the requested rows. The default is “RESULT”.
FIRST	Index of the first row to get. The index starts at 1. Default is 1.
LAST	Index of the last row to get. The index starts at 1. A value of 0 means last page. Default is 0.
PAGE	Page number to get. First page is page 1. If the PAGE parameter is given the FIRST and LAST parameters are not taken in care.
PAGE_SIZE	Page size. Used only when the PAGE parameter has been given. If the page size is 0, no records are returned. The default page size is 10.
COLUMNS	This parameter allows selecting only some of the columns. When not provided, all the columns are retrieved. The COLUMNS parameter is a list of values separated by a semicolon character. Each value itself has the format *NewColName*:*ColName* where *ColName* is the name of the column and *NewColName* is its new name. If the column name doesn't have to be changed, the value is simply *ColName*. The columns will be delivered in given order.
ROWSEP	Row separator when the mode is “STRING”. This defines the string that separates the rows in the data. The default is the line feed character (\n).
COLSEP	Column separator when the mode is “STRING”. This defines the string that separates the columns in the data. The default is “;”.

LINESEP Line separator when the mode is “STRING”. This defines the string that separates the lines for columns containing several lines in the data.
The default is “|”.

Objects created

RESULT This is a Nirva table or string object that contains the requested rows. The name of this object can be changed by using the RESULT parameter.

TABLE_GET_ROW_INDEX

OBJECT:TABLE_GET_ROW_INDEX

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the row index given a primary key value. If there is no primary key defined for the table or if the given key is not found, the command returns 0.

The found row index is written into the output buffer.

Parameters

NAME Object name.

PRIMARY Primary key value to find.

TABLE_IMPORT

OBJECT:TABLE_IMPORT

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	No	No

Description

This command imports data rows from an ASCII file to a table object. The file can be the result of a TABLE_EXPORT command.

This command is not available from client or browser source.

If there is a primary key defined for the table and if the command found some data in the file with already existing primary key, the record is skipped (not imported).

If the primary key value to import contains several lines, only the first line is imported.

Parameters

NAME	Object name.
FILENAME	Pathname of the file from which to import rows. Here is the description of the file format: Each line corresponds to 1 row. The column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the COLSEP and LINESEP parameters. Each line is limited to 1024 characters (may be changed by the LINE_LENGTH parameter). If a row is longer than 1024 characters, the last line character is '\' (backslash) and the row content continues on the next line. The column data must be in the same order than the column definition in the table. The last columns can be omitted if they are blank. The backslash and line feed characters are controlled by the way of WITH_LF parameter. If a cell contains an empty line, Nirva does not write this empty line. In order to force writing an empty line, the line value must be set to “NV_EMPTY”.
APPEND	Append option. If this parameter is set to “YES”, Nirva appends the rows to the table. Otherwise, the imported rows replace the current rows. The default is “NO”.
COLSEP	Column separator. This defines the string that separates the columns in the import file. The default is “;”.
LINESEP	Line separator. This defines the string that separates the lines for columns containing several lines in the import file. The default is “|”.
LINE_LENGTH	This is the maximum line length of the import file. The default is 1024. The minimum value is 1024.
OFFSET	This allows skipping the given number of bytes before importing the file to the table. This can be useful if the import file has some specific header. The default is 0.
SKIPLINES	This allows skipping the given number of lines before importing the file to the table. This can be useful if the import file has some specific header. If both the OFFSET and SKIPLINES parameters are provided, the OFFSET parameter is evaluated before the SKIPLINES parameters so the number of lines to skip will start from the given offset. The default is 0.
WITH_LF	This tells Nirva if the input data contains line feeds in cell content. If set to YES, any line feed in the table data to import must be written as the “\n”

string. Any real backslash character must be written as the “\\” string.
In any case, the carriage return characters are removed.
The default is NO.

TABLE_INSERT_CELL_LINE

OBJECT:TABLE_INSERT_CELL_LINE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command inserts a new cell line.

The command fails when attempting to add a cell line to a primary key that is not empty.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. The new line is inserted before the given line index. If the LINE_INDEX is equal to “0”, the new line is added at the end of the cell. The default is “0”.
LINE	Line data.

TABLE_INSERT_ROWS

OBJECT:TABLE_INSERT_ROWS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command inserts a given number of empty rows to a table object.

The insertion occurs before the given row index.

Parameters

NAME	Object name.
ROW	Row index. The index starts at 1. The insertion occurs before the given row index. For inserting new rows at the end of the table, the ROW parameter must be set to “0”. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
NUMROW	Number of rows to add. The default is “1”.

TABLE_INSERT_COLUMN

OBJECT:TABLE_INSERT_COLUMN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command inserts a new column to a table object.

The insertion occurs before the given column index.

Parameters

NAME	Object name.
COLNAME	Column name. This parameter is mandatory and should not be blank. The column name cannot contain spaces.
DESCRIPTION	Column description.
NUMERIC	Numeric option. If this parameter is set to “YES”, the column is considered as numeric. The default is “NO”.
COL	Column index. The index starts at 1. The insertion occurs before the given column index. For inserting a new column at the end of the table, the COL parameter must be set to “0”. This is the default.
PRIMARY	Primary column. If PRIMARY is set to “YES”, the column becomes the primary column. The default is “NO”.
PRIMARY_CS	Primary column case sensitive. This parameter has meaning only when the PRIMARY parameter has been set to “YES”. If PRIMARY_CS is set to “YES”, the primary key will be case sensitive. The default is “NO” (case insensitive).

TABLE_IS_COLUMN_NUMERIC

OBJECT:TABLE_IS_COLUMN_NUMERIC

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command tells if a column is numeric or not. The result is written in the output buffer.

If the Column is numeric, the returned value is “YES” and “NO” otherwise.

Parameters

NAME	Object name.
COL	Column index. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_JOIN

OBJECT:TABLE_JOIN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command does a join of 2 tables following a foreign key. All or just a part of the source columns can be added to the object.

The link between the 2 tables is the name of the column containing the foreign key.

Parameters

NAME — Object name.

SNAME — Source table object name. This is the name of the object to get columns from and to add to the current object.

USE_OUT_CONTAINER — If set to “YES” the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.

FOREIGN — Foreign key name. This parameter is mandatory and has the format *src_col:dst_col* where *src_col* is the name of the column containing the foreign key in the source table and *dst_col* is the name of the column containing the foreign key in the destination table (the current object). If both source and destination column have the same name, this unique name can be given directly without any ‘:’ separator.

COLUMNS — Name of the columns from the source object to add to the destination object (the current object). This parameter, if provided, consists of several column pairs separated by a semicolon character (‘;’). Each column pair has the format *src_col:dst_col* where *src_col* is the name of the column of the source object to add and *dst_col* is the name of the column in the destination object. If both source and destination column have the same name, this unique name can be given directly without any ‘:’ separator.
If this parameter is not provided, Nirva adds all the columns of the source object to the destination object.

REPLACE — Replace option. This parameter controls if Nirva must replace the destination columns having the same name than the source column or not. The possible values are “YES” or “NO”. The default is “NO”.

TABLE_LOAD

OBJECT:TABLE_LOAD

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	No	No

Description

This command loads the table object from a file. The file must be the result of a TABLE_SAVE command.

This command is not available from client or browser source.

Parameters

NAME	Object name.
FILENAME	Pathname of the file from which to load the table.

TABLE_MODIFY_COLUMN

OBJECT:TABLE_MODIFY_COLUMN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command allows setting the value of a table object column for a selection of rows.

The selection is made by giving a query as parameter.

The command fails if the column to modify is the primary column.

Parameters

NAME	Object name.
COL	Column index to modify. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
VALUE	New column value. The default is blank.

QUERY	Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query. The default value is "*" (search all rows).

TABLE_REMOVE_CELL_LINE

OBJECT:TABLE_REMOVE_CELL_LINE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes cell line.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. The default is "1" (first line).

TABLE_REMOVE_COLUMN

OBJECT:TABLE_REMOVE_COLUMN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes a single column from a table object.

Parameters

NAME Object name.

COL Column index to remove. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TABLE_REMOVE_ROW

OBJECT:TABLE_REMOVE_ROW

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes a single row from a table object.

Parameters

NAME Object name.

ROW Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY

If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

TABLE_REMOVE_ROWS

OBJECT:TABLE_REMOVE_ROWS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes a selection of rows from a table object. The selection is made by giving a query as parameter.

Parameters

NAME

Object name.

QUERY

Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query.
The default value is “*” (search all rows).

TABLE_SAVE

OBJECT:TABLE_SAVE

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	No	No

Description

This command saves permanently the table object into the given file. The save format is proprietary.

This command is not available from client or browser source.

Parameters

NAME Object name.

FILENAME Pathname of the file for saving the table.

TABLE_SEARCH

OBJECT:TABLE_SEARCH

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command searches rows in the table object following a given query string. If the search is successful, the command creates a new table object that contains the resulting rows. The object must exist and must be of table type.

Parameters

NAME Object name.

RESULT Name of the resulting object. The resulting object is a table object containing the requested rows. The default is “RESULT”.

QUERY Query string. The syntax of the query is quiet simple. It is a succession of expressions separated by the operators "OR" or "AND". Each expression itself is composed of a column name, an operator and a value. The valid operators are "=" and "!=" for alphanumeric fields and "=", "!=", "<", ">", "<="and ">=" for numeric columns. For alphanumeric columns, the wildcard character '*' can be used for searching columns starting with a given value or ending with a given value. If the ‘*’ wildcard character is used before and after the search value, NIRVA will find columns containing the requested value. For searching all rows of a table, the query string must be set to "*". Example: "F1=Test* AND F2>15" is a valid query. An empty cell can be searched by setting a value of “NV_CELL_EMPTY_VALUE” (for example MYCOLUMN= NV_CELL_EMPTY_VALUE).
The default value is “*” (search all rows).

SORT	Sort column. Name of the sort column. The default is to not sort. The sort is in ascending order but can be in descending order if the sort column name is preceded by the '-' (minus) character.
MAXDOCS	Maximum number of rows to get. This parameter can limit the number of found rows. Nirva stops the search process when the number of requested rows corresponding to the search criteria is reached. The default is "-1" (all found rows).
COLUMNS	This parameter allows selecting only some of the columns. When not provided, all the columns are retrieved. The COLUMNS parameter is a list of values separated by a semicolon character. Each value itself has the format *NewColName*:*ColName* where *ColName* is the name of the column and *NewColName* is its new name. If the column name doesn't have to be changed, the value is simply *ColName*. The columns will be delivered in given order.
WITH_ROW_INDEX	If this parameter is set to "YES", the command adds a column named "NV_SRC_ROW_INDEX" in the result table. This column contains the row index of the source table for the found records. If this column already exists, the command replaces its content. The NV_SRC_ROW_INDEX column cannot be removed from display when using the COLUMNS parameter but its name and its order can be changed as other columns. For example, in order to display only this column by renaming it INDEX, the COLUMNS parameter must be set to "INDEX:NV_SRC_ROW_INDEX".

Objects created

RESULT	This is a Nirva table object that contains the found rows. The name of this object can be changed by using the RESULT parameter.

TABLE_SELECT_FROM

OBJECT:TABLE_SELECT_FROM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command allows marking some of the table rows.

For each row, it searches in a given column if this column contains the values found in a given column of another table object.

Parameters

NAME	Object name. This must be a table object.
SOURCE	Source object name. This must be a table object. The command will use the source object to get the values to search.
USE_OUT_CONTAINER	If set to “YES” the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.
SOURCE_COL	Index of the column of the source object from which to get the values to search for.
SOURCE_COLNAME	The column name can be given instead of the column index (SOURCE_COL parameter).
COL	Index of the column where the command will search the values.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
SELECTED_COL	Index of the column in which the command will write the result of the search. This cannot be a primary key.
SELECTED_COLNAME	The column name can be given instead of the column index (SELECTED_COL parameter).
SELECTED_YES	Value to set in the SELECTED_COLUMN if one of the values has been found in the record. The default is “YES”.
SELECTED_NO	Value to set in the SELECTED_COLUMN if the values have not been found in the record. The default is “NO”.
CASE_SENSITIVE	If set to “NO”, the search will not be case sensitive. The default is “YES”.

TABLE_SET_CELL

OBJECT:TABLE_SET_CELL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the value of an existing cell.

Parameters

NAME Object name.

ROW Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

COL Column index of the cell. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

LINESEP Line separator. This can be a string (not only a single character). The default is ";".

DATA Cell data.

TABLE_SET_CELL_LINE

OBJECT:TABLE_SET_CELL_LINE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the value of an existing cell line. If the line index is greater than the current number of lines, a new line is added at the end of the cell.

The command fails when attempting to set the primary key cell line when the line index is greater than 1.

Parameters

NAME Object name.

ROW Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

COL Column index of the cell. The index starts at 1.

COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
LINE_INDEX	Index of the cell line. The index starts at 1. If the line index is 0 or is greater than the number of lines of the cell, a new line is added at the end of the cell. The default is “1” (first line).
LINE	Line data.

TABLE_SET_COLUMN

OBJECT:TABLE_SET_COLUMN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the cells values for a given column and all rows. If the requested column is a primary column and the column data contains duplicate values, the command fails.

Parameters

NAME	Object name.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
ROWSEP	Row separator. This can be a string (not only a single character). The default is “;”.
LINESEP	Line separator. This can be a string (not only a single character). The default is “|”.
DATA	Column data. Use column and line separators defined in LINESEP and COLSEP. If the data doesn’t contain values for all rows, remaining rows will be cleared (no value).

TABLE_SET_COLUMN_DESCRIPTION

OBJECT:TABLE_SET_COLUMN_DESCRIPTION

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command changes a table column description. The object must exist and must be of table type.

Parameters

NAME Object name.

COL Column index. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

DESCRIPTION New table column description. The default is a blank string.

TABLE_SET_COLUMN_NAME

OBJECT:TABLE_SET_COLUMN_NAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command changes a table column name. The object must exist and must be of table type.

Parameters

NAME Object name.

COL Column index. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

NEW_NAME New table column name. If the new column name already exists, the command doesn't fail but does nothing.

TABLE_SET_COLUMN_TYPE

OBJECT:TABLE_SET_CLUMN_TYPE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command changes a table column type. The object must exist and must be of table type.

Parameters

NAME Object name.

COL Column index. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

TYPE New table column type. This can be “NUMERIC” or “ALPHANUMERIC” (default).

TABLE_SET_DESCRIPTION

OBJECT:TABLE_SET_DESCRIPTION

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sets the description of a table object. The object must exist and must be of table type.

Parameters

NAME	Object name.
DESCRIPTION	New table description. The default is a blank string.

TABLE_SET_PRIMARY_COLUMN

OBJECT:TABLE_SET_PRIMARY_COLUMN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command allows setting or changing the primary column of the table. The primary column must contain a unique string that identifies it. If there are duplicates, the command fails.

The command also fails if the primary key column of a record contains more that one cell line.

If the primary column cell data is empty, the command doesn't fail but the row containing the empty column cannot be retrieved by is primary key.

A table object can have only one primary key, so if there was another primary key before sending this command, it's removed.

Parameters

NAME	Object name.
COLNAME	Column name. If this parameter is blank or is not given, Nirva considers that there is no primary column.
PRIMARY_CS	Primary column case sensitive. If PRIMARY_CS is set to “YES”, the primary key will be case sensitive. The default is “NO” (case insensitive).

TABLE_SET_ROW

OBJECT:TABLE_SET_ROW

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command writes a complete row data.

If there is a primary key defined for the table and if the entered primary key already exists, the command fails.

If the primary key value to import contains several lines, only the first line is imported.

Parameters

NAME Object name.

ROW Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

DATA Row data. The column separator is the character ';' and the line separator for columns containing several lines is '|'. These separators can be changed by using the COLSEP and LINESEP parameters. The column data must be in the same order than the column definition in the table. The last columns can be omitted if they are blank. If a cell contains an empty line, Nirva does not write this empty line. In order to force writing an empty line, the line value must be set to “NV_EMPTY”.

COLSEP Column separator. This defines the string that separates the columns in the data.
The default is “;”.

LINESEP Line separator. This defines the string that separates the lines for columns containing several lines in the data.
The default is “|”.

TABLE_SORT

OBJECT:TABLE_SORT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sorts the rows of the table object. The object must exist and must be of table type.

Parameters

NAME	Object name.
SORT	Sort column. Name of the sort column. The sort is in ascending order but can be in descending order if the sort column name is preceded by the '-' (minus) character.
NATURAL	Natural option. If this parameter is set to "YES", the command sorts items in a "natural" way. This is useful to list items containing both alphabetic characters and numbers. This option has no effect when NUMERIC is set to "YES".
LOCALE	Locale name. Internally the command uses unicode to sort strings so the sorting should be ok. However, if the sorting is not correct in your language you can set the locale name using this parameter. Ex: "FRENCH".

TABLE_SORT_CELL

OBJECT:TABLE_SORT_CELL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command sorts a cell of a table object. A table object cell may contain several lines of numeric or alphanumeric value. This command sorts these lines.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.
PRIMARY	If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.
COL	Column index of the cell. The index starts at 1.
COLNAME	The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.
ASCENDING	If this parameter is set to “YES”, the sorting occurs in ascending order. If set to “NO” ”, the sorting occurs in descending order. The default is “YES”.
NATURAL	Natural option. If this parameter is set to “YES”, the command sorts items in a “natural” way. This is useful to list items containing both alphabetic characters and numbers.
LOCALE	Locale name. Internally the command uses unicode to sort strings so the sorting should be ok. However, if the sorting is not correct in your language you can set the locale name using this parameter. Ex: “FRENCH”.

TABLE_SWAP_CELL_LINES

OBJECT:TABLE_SWAP_CELL_LINES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command swaps 2 cell lines.

Parameters

NAME	Object name.
ROW	Row index of the cell. The index starts at 1. If this value is 0, the row index is the last row of the table. This is the default.

PRIMARY If the table has a primary index, the primary value of the row to get can be given in the PRIMARY parameter instead of the ROW parameter. If both ROW and PRIMARY parameters are given, ROW has the priority.

COL Column index of the cell. The index starts at 1.

COLNAME The column name can be given in COLNAME instead of the COL parameter. If both COL and COLNAME parameters are given, COLNAME has the priority.

LINE_INDEX1 Index of the first cell line to swap. The index starts at 1.

LINE_INDEX2 Index of the second cell line to swap. The index starts at 1.

TABLE_SWAP_COLUMNS

OBJECT:TABLE_SWAP_COLUMNS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command swaps 2 columns.

Parameters

NAME Object name.

COL1 Index of the first column to swap. The index starts at 1.

COL2 Index of the second column to swap. The index starts at 1.

TABLE_SWAP_ROWS

OBJECT:TABLE_SWAP_ROWS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command swaps 2 rows.

Parameters

NAME	Object name.
ROW1	Index of the first row to swap. The index starts at 1.
ROW2	Index of the second row to swap. The index starts at 1.

FILE_APPEND

OBJECT:FILE_APPEND

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command appends the content of a source file (or a part of it) to a destination file.

Parameters

NAME	Object name. This is the destination object (appended file).
SNAME	Object source name.
USE_OUT_CONTAINER	If set to "YES" the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.
OFFSET	Offset. This parameter gives the offset in bytes of the source object from which appending will start. The default value is "0" (beginning of file).
NUM_BYTES	Number of bytes to copy. This parameter gives the number of bytes of the source object to append. The default value is "-1" (until the end of file).

FILE_CLEAR

OBJECT:FILE_CLEAR

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command physically clears the file associated with the file object if it exists.

Clearing a file means setting its size to 0 by removing all its content.

Parameters

NAME Object name.

FILE_COMPRESS

OBJECT:FILE_COMPRESS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command compresses the content of a source file to a destination file. The compression is for internal use. Compressed files can be only decompressed using the FILE_DECOMPRESS command.

Parameters

NAME Object name. This is the object to compress.

DNAME Object destination name (compressed file).

MODE Compression mode. Can be “HUFF” for huffman compression or “LZMA” for lzma compression. The default is “HUFF”.

FILE_CREATE

OBJECT:FILE_CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command physically creates the file associated with the file object if it doesn’t exist.

Generally, if the file object is not persistent, the file has been created at object creation time so this command will be useful only for persistent files.

Parameters

NAME Object name.

FILE_DECOMPRESS

OBJECT:FILE_DECOMPRESS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command decompresses the content of a source file to a destination file.

Parameters

NAME Object name. This is the object to decompress.

DNAME Object destination name (decompressed file).

FILE_EXIST

OBJECT:FILE_EXIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command tells if the file associated with the file object physically exists or not.

The result is written in the output buffer. The return value is “YES” if the file exists and “NO” otherwise.

Parameters

NAME Object name.

FILE_GET_DIRNAME

OBJECT:FILE_GET_DIRNAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the directory name of the file associated with the file object.

The result is in the output buffer.

Parameters

NAME Object name.

FILE_GET_EXTENSION

OBJECT:FILE_GET_EXTENSION

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the extension of the file associated with the file object.

The result is in the output buffer.

The '.' (point) character is not returned.

Parameters

NAME	Object name.

FILE_GET_FILENAME

OBJECT:FILE_GET_FILENAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the name of the file associated with the file object.

The result is in the output buffer.

Only the file name is returned without the complete path. For the complete file path, one can use the FILE_GET_PATHNAME command.

Parameters

NAME	Object name.
WITH_EXT	If this parameter is set to "NO, the command doesn't return the extension of the file. The default is "YES".

FILE_GET_ORIGIN

OBJECT:FILE_GET_ORIGIN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

Retrieves the origin of a file. The origin is a string maintained by the file object. It's generally used to keep the origin of the file. When sending a file from a web browser to the NIRVA server by a HTTP POST command, the server writes the origin file name into the origin string.

Parameters

NAME Object name.

FILE_GET_PATHNAME

OBJECT:FILE_GET_PATHNAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the complete path name of the file associated with the file object.

The result is in the output buffer.

Parameters

NAME Object name.

FILE_GET_SIZE

OBJECT:FILE_GET_SIZE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the size in bytes of the file associated with the file object.

The result is in the output buffer.

Parameters

NAME Object name.

FILE_REMOVE

OBJECT:FILE_REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command removes the file associated with the file object.

By default, a persistent file cannot be removed by this function. In order to remove a persistent file, the FORCE parameter must be set to “YES”.

If the file is a temporary file, Nirva removes it immediately.

If the file is a cached file, Nirva takes it under the control of the cache mechanism to be deleted when it expires.

Parameters

NAME Object name.

FORCE If this parameter is set to “YES” and the file is persistent, Nirva will remove it. This is the only possibility to remove a persistent file.

FILE_SET_ASCII

OBJECT:FILE_SET_ASCII

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command compares the file object with the server platform (WINDOWS or UNIX) and adjusts the carriage return / line feed pairs if necessary. The maximum line length must be 65530 bytes.

Parameters

NAME Object name.

FILE_SET_FILENAME

OBJECT:FILE_SET_FILENAME

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	No	No

Description

This command changes the file associated with a file object. If the file object is not persistent, the previous associated file is removed (and goes under the control of Nirva cache mechanism if this is a cached object).

This command is not available from client or browser source.

Parameters

NAME Object name.

FILENAME This parameter gives the new file name to associate to the file object. If this parameter is not provided, Nirva creates itself a file name following

	information of the EXTENSION, PREFIX, SUFFIX and DIRECTORY parameters.
DIRECTORY	This is the name of a directory where Nirva will put the local file if the received object is a file object and the FILENAME parameter is not given. For information, Nirva generates an error message when trying to create a persistent file in the application work directory. If this parameter is not provided or is blank, Nirva uses the application work directory for temporary and cached files and the application file directory for persistent files.
EXTENSION	This is the file extension to use. The EXTENSION parameter is only used if the object to create is a file object. If this parameter is not provided or is blank, Nirva uses ".obj" for persistent files and ".tmp" for temporary and cached files. If the point character is omitted in the EXTENSION parameter, Nirva adds it.
PREFIX	This is the file prefix to use. The PREFIX parameter is only used if the object to create is a file object. The prefix, if provided is used by Nirva to automatically create the server file name.
SUFFIX	This is the file suffix to use. The SUFFIX parameter is only used if the object to create is a file object. The suffix, if provided is used by Nirva to automatically create the server file name.

FILE_SET_ORIGIN

OBJECT:FILE_SET_ORIGIN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

Sets the origin of a file. The origin is a string maintained by the file object. It's generally used to keep the origin of the file. When sending a file from a web browser to the NIRVA server by a HTTP POST command, the server writes the origin file name into the origin string.

Parameters

NAME	Object name.

ORIGIN	Origin. This can be any kind of text

FILE_SET_PERSIST

OBJECT:FILE_SET_PERSIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command changes the kind of file object to temporary, cached or persistent.

If the associated file is in the application work directory, the command returns an error when trying to make the object persistent.

Parameters

NAME	Object name.
PERSIST	This parameter gives the new persistent value. The parameter can take value “0” for temporary files, “-1” for persistent files and any other value (in seconds) for cached files. The default value is “0” (not persistent).

BINARY_APPEND

OBJECT:BINARY_APPEND

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command appends the content of a source binary object (or a part of it) to a destination binary object.

Parameters

NAME	Object name. This is the destination object (appended binary object).
SNAME	Object source name.
USE_OUT_CONTAINER	If set to “YES” the destination object is taken from the output container. Otherwise from the input container. The source object is always taken from the input container.
OFFSET	Offset. This parameter gives the offset in bytes of the source object from which appending will start. The default value is ”0” (beginning of binary data).
NUM_BYTES	Number of bytes to copy. This parameter gives the number of bytes of the source object to append. The default value is ”-1” (until the end of binary data).

BINARY_CLEAR

OBJECT:BINARY_CLEAR

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command resets the binary object size to 0 freeing its data from memory.

Parameters

NAME	Object name.

BINARY_GET_SIZE

OBJECT:BINARY_GET_SIZE

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	Yes

Description

This command returns the size in bytes of the binary object.

The result is in the output buffer.

Parameters

NAME Object name.

BINARY_LOAD

OBJECT:BINARY_LOAD

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	No	No

Description

This command loads the binary object data from a file.

This command is not available from client or browser source.

Parameters

NAME Object name.

FILENAME Name of the file to load.

OFFSET Offset. This parameter gives the offset in bytes of the file from which the load will start.
The default value is ”0” (beginning of file).

NUM_BYTES Number of bytes to copy. This parameter gives the number of bytes of the file to load.
The default value is ”-1” (until the end of file).

BINARY_SAVE

OBJECT:BINARY_SAVE

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	Yes	No	No

Description

This command saves the binary object data to a file.

This command is not available from client or browser source.

Parameters

NAME Object name.

FILENAME Name of the save file.

OFFSET Offset. This parameter gives the offset in bytes of the object from which the save will start.
The default value is ”0” (beginning of binary data).

NUM_BYTES Number of bytes to copy. This parameter gives the number of bytes of the object to save.
The default value is ”-1” (until the end of binary data).

PACKAGE class

The PACKAGE class provides some commands for packaging and installing files on the NIRVA server.

The commands of this class only allow installing files in the NIRVA directory and subdirectories.

In fact, NIRVA defines package files. A package file is a compressed file that contains the files to install and the relative path of the files to install.

When installing a package, NIRVA uses a base path from which the content of the package will be installed. With the SYSTEM:PACKAGE classes, the base path is always the NIRVA root directory.

The package file itself is built following the information given in a package description file. Please refer to the “installation package” chapter for further information about package description files.

The class provides the PACKAGE command to prepare a package file and the INSTALL command to install it.

INFO

PACKAGE:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

Returns the package file header information. The header part of a package file contains some entries of the form EntryName = EntryValue. This command creates a NIRVA indexed string list object containing these entries.

Parameters

FILE — Name of the NIRVA file object that contains the package file to get information from. This object must be in the input container. The default value is "PACKAGE_FILE".

FILE_LIST — Name of a string list file object. If this parameter is given, the command will fill the object with file names found in the package file.

Objects created

PACKAGE_INFO — This is a Nirva indexed string list object corresponding to the pairs found in the HEADER section of the package file. If there is no HEADER section, the PACKAGE_INFO object is created but is empty.

FILE_LIST — List of file names found in the package file. The object name is given by the FILE_LIST parameter. No object created if this parameter is empty.

INSTALL

PACKAGE:INSTALL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

Installs a package file previously created by the SYSTEM PACKAGE PACKAGE command.

The package file is a binary file that itself contains other files.

This command can only install files in the NIRVA directory (and subdirectories).

If the package file has a header entry "*SERVICE=SrvName*", the command installs the package as a service package for the *SrvName* service.

If the package file has a header entry "*APPLICATION=AppName*", the command installs the package as an application package for the *AppName* application.

If the package file has a header entry PLATFORM = *Platform* where Platform is the target platform (WIN32, WIN64, AIX, LINUX, LINUX64, HPUX, HPUXI or SOLARIS), the INSTALL command checks if the target platform corresponds to the package file and returns an error if it's not the case. See the chapter "Installation packages" for information about package file headers.

Permissions

SYSTEM_INSTALL

SERVICE_INSTALL

APPLICATION_INSTALL

Parameters

FILE	Name of the NIRVA file object that contains the package file to install. This object must be in the input container. The default value is "PACKAGE_FILE".

PACKAGE

PACKAGE:PACKAGE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

Creates a package file that will be usable by the SYSTEM PACKAGE INSTALL command.

The package file is a binary file that itself contains other files. The package file is created following the content of a package description file that must reside on the NIRVA files directory. The package file is compressed.

Permissions

SYSTEM_PACKAGE

Parameters

FILE	Name of the NIRVA file object that will contain the resulting package file. This object is in the output container. If the object doesn't exist, the command creates it. The default value is "PACKAGE_FILE".
DESC_FILE	Name of package description file. This must correspond to an existing file that resides in the NIRVA files directory. The default value is "package.lst". The format of the package description file is given in the "installation packages" chapter.

REGISTRY class

The REGISTRY class provides commands for working registries.

The Nirva registries allow maintaining persistent information at server level. There are two kinds of registries: standard registries and user registries. Standard registries are written in fixed directories in the Nirva disk tree while user registries can be written in a directory given by the user. Standard registries are mainly used for configuration information while user registries are designed to maintain application or service data. The registries can be defined at 3 different levels:

System registry for system level information. This registry is available to all applications and services.

Application registry for persistent information specific to each application. Having a separate registry for each application assumes the complete independence between the Nirva applications.

Service registry is used for external service persistent information.

The Nirva registry is a very powerful registry because it can store in a hierarchical structure any kind of Nirva object. For example, a Nirva table can be stored as a registry entry.

The access to the registry is simple. Only 7 commands allow the registry manipulation:

- The *GET* command copies a branch (or some single entries) of the registry into a session container (or sub-container).
- The *SET* command copies a branch of a session container (or some objects) into the registry.
- The *REMOVE* command removes a branch of the registry (or some single entries).
- The *CREATE* command creates a new branch (or key).
- The *EXIST* command tests if a branch of the registry exists.
- The *EXPORT* command exports a registry branch into a file.
- The *IMPORT* command imports registry information from a file previously created with the EXPORT function.

These 7 commands have also some options to operate on dedicated objects, on a complete branch including sub-containers or not.

The usual way to modify the registry is to call the *GetRegistryKey*, then to modify the necessary entries with the system object commands on the session container and finally to write the result back to the registry by the use of the *SetRegistryKey* function.

The access to the registries is serialized automatically by Nirva server avoiding conflicts. Despite of that, it's also possible to use Nirva locking mechanisms (see next chapter) for controlling registry transactional operations when several registry accesses must be controlled in a transactional way.

Nirva maintains a read/write cache for each registry. This cache is automatically cleared every 3 hours for freeing memory resources.

For accessing and/or creating a user registry, the OPEN_REGISTRY must be used giving the registry name and the registry base directory. After a user registry has been successfully opened, it can be accessed with the other registry commands but giving its name as parameter. When finishing working with a user registry, this one must be close using the CLOSE_REGISTRY command. Like standard registries, user registries can be defined at system, application or service levels.

For security reasons, the access to the registry key named “system” (and all its subkeys) is protected. At system level, the “system” registry key is only accessible from the NIRVA code itself so it's not possible to access it directly from any command. At service level, the “system” key is accessible only if the command to access it comes from the service itself. At application level, the “system” key is accessible only from an application procedure. This is not true for user registries where all the keys are accessible.

CLEAR_CACHE

REGISTRY:CLEAR_CACHE

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	No

Description

This command manually clears a registry cache.

All registry caches are periodically and automatically cleared every hour by the server.

Parameters

SERVICE	This parameter tells if the registry is the system, service or application registry. By default, if the SERVICE parameter is not provided or is blank, the registry is considered to be the application registry. If SERVICE is set to “SYSTEM”, the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.

NAME — Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.

CLOSE_REGISTRY

REGISTRY:CLOSE_REGISTRY

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command closes a user registry previously opened by the OPEN_REGISTRY command. In order to use a user registry, this one must have been previously opened. It must be close after finishing working with it. There must be the same number of close than open.

Parameters

SERVICE — This parameter tells if the registry is the system, service or application registry. By default, if the SERVICE parameter is not provided or is blank, the registry is considered to be the application registry. If SERVICE is set to “SYSTEM”, the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.

NAME — User registry name. A registry name is case insensitive. It uniquely identifies the registry at system, application or service level. We can have the same name for an application, system or service registry. This parameter is mandatory.

CREATE

REGISTRY:CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command creates a registry key if doesn't exist.

Permissions

These permissions apply only to standard registries (no permission on user registries)

REGISTRY_SYSTEM_WRITE_CLIENT

REGISTRY_APPLICATION_WRITE_CLIENT

Parameters

KEY	Registry key. The registry key is the equivalent of a Nirva subcontainer. The registry key must give the entire path to the subcontainer. For example "`MySubKey.MyKey`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the application registry. The registry key cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. By default, the registry used is the application registry. In order to access the system registry, the KEY parameter must be enclosed in brackets. For example "`(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the system registry. In the same way, in order to access a service registry, the KEY parameter must be enclosed in brackets. For example "`MyService(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the "`MyService`" service registry. An alternate way to access system or service registry is to use the SERVICE parameter. At this time, the KEY parameter should contain only the key without any brackets or service name. The registry key is case insensitive.
SERVICE	This parameter can be used for accessing system or service registry instead of using brackets in the KEY parameter. If it's blank, or not given, Nirva will use the syntax of the KEY parameter to determine if it's a system, service or application registry. If SERVICE is set to "SYSTEM", the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.
NAME	Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.

EXIST

REGISTRY:EXIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command tests if a registry key/entry exists or not. The result is written in the output container in a boolean object named “EXIST”.

Permissions

These permissions apply only to standard registries (no permission on user registries)

REGISTRY_SYSTEM_READ_CLIENT

REGISTRY_APPLICATION_READ_CLIENT

Parameters

KEY

Registry key. The registry key is the equivalent of a Nirva subcontainer. The registry key must give the entire path to the subcontainer. For example “`MySubKey.MyKey`” points to the key “`MyKey`” of the subkey “`MySubKey`” of the application registry. The registry key cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character.
By default, the registry used is the application registry. In order to access the system registry, the KEY parameter must be enclosed in brackets. For example “`(MySubKey.MyKey)`” points to the key “`MyKey`” of the subkey “`MySubKey`” of the system registry.
In the same way, in order to access a service registry, the KEY parameter must be enclosed in brackets. For example “`MyService(MySubKey.MyKey)`” points to the key “`MyKey`” of the subkey “`MySubKey`” of the “`MyService`” service registry.
An alternate way to access system or service registry is to use the SERVICE parameter. At this time, the KEY parameter should contain only the key without any brackets or service name.
The registry key is case insensitive.

SERVICE

This parameter can be used for accessing system or service registry instead of using brackets in the KEY parameter. If it’s blank, or not given, Nirva will use the syntax of the KEY parameter to determine if it’s a system, service or application registry. If SERVICE is set to “SYSTEM”, the registry is the

	system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.
NAME	Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.
ENTRY	Optionnal entry name. If this parameter is given the command checks the existence of the corresponding entry.

Objects created

EXIST	Result of the command. This object has the value TRUE if the registry key exists and FALSE otherwise.

EXPORT

REGISTRY:EXPORT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command exports the given registry key (or a part of it) in the given file. The exported file format is proprietary and should not be modified manually.

Permissions

These permissions apply only to standard registries (no permission on user registries)

REGISTRY_SYSTEM_READ_CLIENT

REGISTRY_APPLICATION_READ_CLIENT

Parameters

KEY	Registry key. The registry key is the equivalent of a Nirva subcontainer. The registry key must give the entire path to the subcontainer. For example “`MySubKey.MyKey`” points to the key “`MyKey`” of the subkey “`MySubKey`” of the application registry. The registry key cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. By default, the registry used is the application registry. In order to access the

	system registry, the KEY parameter must be enclosed in brackets. For example "`(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the system registry. In the same way, in order to access a service registry, the KEY parameter must be enclosed in brackets. For example "`MyService(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the "`MyService`" service registry. An alternate way to access system or service registry is to use the SERVICE parameter. At this time, the KEY parameter should contain only the key without any brackets or service name. The registry key is case insensitive.
SERVICE	This parameter can be used for accessing system or service registry instead of using brackets in the KEY parameter. If it's blank, or not given, Nirva will use the syntax of the KEY parameter to determine if it's a system, service or application registry. If SERVICE is set to "SYSTEM", the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.
NAME	Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.
FILENAME	Name of the export file. All registry key information is written into this file in a proprietary format.
SUBKEYS	This parameter also allows exporting the subkeys of the registry key. To copy subkeys, the parameter must be set to "YES". The default value is "NO".

GET

REGISTRY:GET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command copies the given registry key (or a part of it) in the given output container.

Permissions

These permissions apply only to standard registries (no permission on user registries)

REGISTRY_SYSTEM_READ_CLIENT

REGISTRY_APPLICATION_READ_CLIENT

Parameters

KEY

Registry key. The registry key is the equivalent of a Nirva subcontainer. The registry key must give the entire path to the subcontainer. For example "`MySubKey.MyKey`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the application registry. The registry key cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character.
By default, the registry used is the application registry. In order to access the system registry, the KEY parameter must be enclosed in brackets. For example "`(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the system registry.
In the same way, in order to access a service registry, the KEY parameter must be enclosed in brackets. For example "`MyService(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the "`MyService`" service registry.
An alternate way to access system or service registry is to use the SERVICE parameter. At this time, the KEY parameter should contain only the key without any brackets or service name.
The registry key is case insensitive.

SERVICE

This parameter can be used for accessing system or service registry instead of using brackets in the KEY parameter. If it's blank, or not given, Nirva will use the syntax of the KEY parameter to determine if it's a system, service or application registry. If SERVICE is set to "SYSTEM", the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.

NAME

Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.

APPEND

Append mode. If this parameter is set to "YES", the command will append the registry key to the output container. In this case, the eventual replacement of existing objects is controlled by the parameter REPLACE.
When not in append mode, the output container is first released before the copy. This is the default.

REPLACE

This parameter has a meaning only in append mode (APPEND parameter set to "YES"). If REPLACE is set to "YES" all registry key entries will replace output container objects having the same name.
The default is "NO" so the output container objects having the same name than the registry key entries will not be replaced.

SUBKEYS

This parameter also allows copying the subkeys of the registry key to the output container.

	To copy subkeys, the parameter must be set to "YES". The default value is "NO".
RECURSE	This parameter has meaning only when SUBKEYS is set to YES. If RECURSE is set to "YES", the command gets all the child hierarchy. When set to "NO", only the direct subkeys are listed with their objects. The default value is "YES".
ENTRIES	This parameter allows specifying only some of the registry key entries to copy to the output container. If used, the parameter should contain the names of valid registry key entries, separated by semicolon character (';'). The default is to include all registry key entries.
WITH_DATA	This parameter allows to also copy the entry data (and not only the entry description) of the registry key to the output container. To not copy entry data, the parameter must be set to "NO". The default value is "YES".
PERSIST	This parameter gives the new persistent value for persistent file entries that have to be copied. By default, the new file objects will not be persistent. The parameter can take value "0" for temporary files, "-1" for persistent files and any other value (in seconds) for cached files.
FILES	This parameter, if set to "YES", tells Nirva to also copy files with file entries. If this parameter is set to "NO", the file entries are copied but not the file they point to. When files are copied, Nirva automatically creates the new files in the application file directory if the file object is persistent and in the application work directory otherwise. This destination directory can be changed by the FILEDIR parameter if the command has been sent from a procedure or from an external service. The default value is "YES".
FILEDIR	This parameter has meaning only if the FILES parameter has been set to "YES". It allows changing the destination directory for files of file objects. If the FILEDIR is set to the application work directory and the PERSIST parameter is set to "-1" (persistent), the command will return an error because the application work directory should not contain any persistent file. The FILEDIR parameter can be used only if the command source is procedure or service. Otherwise, it's ignored.

IMPORT

REGISTRY:IMPORT

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command imports the content of the given file into the given registry key. The registry key is created if it doesn't exist.

The import file must be the result of a previous REGISTRY EXPORT command.

Permissions

These permissions apply only to standard registries (no permission on user registries)

REGISTRY_SYSTEM_WRITE_CLIENT

REGISTRY_APPLICATION_WRITE_CLIENT

Parameters

KEY

Registry key. The registry key is the equivalent of a Nirva subcontainer. The registry key must give the entire path to the subcontainer. For example "`MySubKey.MyKey`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the application registry. The registry key cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character.
By default, the registry used is the application registry. In order to access the system registry, the KEY parameter must be enclosed in brackets. For example "`(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the system registry.
In the same way, in order to access a service registry, the KEY parameter must be enclosed in brackets. For example "`MyService(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the "`MyService`" service registry.
An alternate way to access system or service registry is to use the SERVICE parameter. At this time, the KEY parameter should contain only the key without any brackets or service name.
The registry key is case insensitive.

SERVICE

This parameter can be used for accessing system or service registry instead of using brackets in the KEY parameter. If it's blank, or not given, Nirva will use the syntax of the KEY parameter to determine if it's a system, service or application registry. If SERVICE is set to "SYSTEM", the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.

NAME	Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.
FILENAME	Name of the import file. This must be the result of a previous REGISTRY EXPORT command.
APPEND	Append mode. If this parameter is set to “YES”, the command will append the imported objects and subkeys to the registry key. In this case, the eventual replacement of existing objects is controlled by the parameter REPLACE. When not in append mode, the registry key is first released before the copy. This is the default.
REPLACE	This parameter has a meaning only in append mode (APPEND parameter set to “YES”). If REPLACE is set to “YES” all imported objects will replace registry key entries having the same name. The default is “NO” so the registry key entries having the same name than the imported objects will not be replaced.

OPEN_REGISTRY

REGISTRY:OPEN_REGISTRY

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command opens and eventually creates a user registry pointing to a dedicated directory. In order to use a user registry, this one must have been previously opened. It must be close after finishing working with it. There must be the same number of close than open.

Nirva has locking mechanisms for accessing registries but they do not apply if two registries has been defined to access the same directory so there should never have 2 user registries with a different name pointing to the same directory.

Parameters

SERVICE	This parameter tells if the registry is the system, service or application registry. By default, if the SERVICE parameter is not provided or is blank, the registry is considered to be the application registry. If SERVICE is set to “SYSTEM”, the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.

NAME	Registry name. A registry name is case insensitive. It uniquely identifies the registry at system, application or service level. We can have the same name for an application, system or service registry. This parameter is mandatory.
DIRECTORY	Directory path. If the directory doesn't exist, the command creates it. If the creation fails, the command also fails. This parameter is mandatory.

REMOVE

REGISTRY:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes a registry key (or part of it).

Permissions

These permissions apply only to standard registries (no permission on user registries)

REGISTRY_SYSTEM_WRITE_CLIENT

REGISTRY_APPLICATION_WRITE_CLIENT

Parameters

KEY

Registry key. The registry key is the equivalent of a Nirva subcontainer. The registry key must give the entire path to the subcontainer. For example "`MySubKey.MyKey`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the application registry. The registry key cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character.

By default, the registry used is the application registry. In order to access the system registry, the KEY parameter must be enclosed in brackets. For example "`(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the system registry.

In the same way, in order to access a service registry, the KEY parameter must be enclosed in brackets. For example "`MyService(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the "`MyService`" service registry.

An alternate way to access system or service registry is to use the SERVICE

	parameter. At this time, the KEY parameter should contain only the key without any brackets or service name. The registry key is case insensitive.
SERVICE	This parameter can be used for accessing system or service registry instead of using brackets in the KEY parameter. If it's blank, or not given, Nirva will use the syntax of the KEY parameter to determine if it's a system, service or application registry. If SERVICE is set to "SYSTEM", the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.
NAME	Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.
SUBKEYS	This parameter also allows removing the subkeys of the registry key. To remove subkeys, the parameter must be set to "YES". The default value is "NO".
ENTRIES	This parameter allows specifying only some of the key entries to remove from the registry key. If used, the parameter should contain the names of valid registry key entries, separated by semicolon character (';'). The default is to remove all entries of the registry key.

SET

REGISTRY:SET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command copies the input container (or a part of it) in the given registry key. The registry key is created if it doesn't exist.

Permissions

These permissions apply only to standard registries (no permission on user registries)

REGISTRY_SYSTEM_WRITE_CLIENT

REGISTRY_APPLICATION_WRITE_CLIENT

Parameters

KEY	Registry key. The registry key is the equivalent of a Nirva subcontainer. The registry key must give the entire path to the subcontainer. For example "`MySubKey.MyKey`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the application registry. The registry key cannot start with a number or punctuation character. A good practice is to use only letters, numbers and underscore character. By default, the registry used is the application registry. In order to access the system registry, the KEY parameter must be enclosed in brackets. For example "`(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the system registry. In the same way, in order to access a service registry, the KEY parameter must be enclosed in brackets. For example "`MyService(MySubKey.MyKey)`" points to the key "`MyKey`" of the subkey "`MySubKey`" of the "`MyService`" service registry. An alternate way to access system or service registry is to use the SERVICE parameter. At this time, the KEY parameter should contain only the key without any brackets or service name. The registry key is case insensitive.
SERVICE	This parameter can be used for accessing system or service registry instead of using brackets in the KEY parameter. If it's blank, or not given, Nirva will use the syntax of the KEY parameter to determine if it's a system, service or application registry. If SERVICE is set to "SYSTEM", the registry is the system registry. If SERVICE is set to the name of an external service, the registry is considered to be a service registry.
NAME	Registry name for a user registry. If this parameter is given, the registry is a user registry. It must have been previously opened by the OPEN_REGISTRY command at service, system or application level.
APPEND	Append mode. If this parameter is set to "YES", the command will append the input container to the registry key. In this case, the eventual replacement of existing objects is controlled by the parameter REPLACE. When not in append mode, the registry key is first released before the copy. This is the default.
REPLACE	This parameter has a meaning only in append mode (APPEND parameter set to "YES"). If REPLACE is set to "YES" all input container objects will replace registry key entries having the same name. The default is "NO" so the registry key entries having the same name than the input container objects will not be replaced.
SUBKEYS	This parameter also allows copying the subcontainers of the input container to the registry key. To copy subcontainers, the parameter must be set to "YES". The default value is "NO".

ENTRIES	This parameter allows specifying only some of the input container objects to copy to the registry key. If used, the parameter should contain the names of valid input container objects, separated by semicolon character (‘;’). The default is to include all objects.
WITH_DATA	This parameter allows to also copy the entry data (and not only the entry description) of the input container objects to the registry key. To not copy entry data, the parameter must be set to “NO”. The default value is “YES”.

REQUEST class

The REQUEST class allows to establish connections to other NIRVA servers and to send commands to them.

A connection to another NIRVA server is represented by a request. A request is a named object that defines the connection parameters. Once a request has been opened, any command sent to the session can be redirected to the distant NIRVA server by giving the request name in the NV_REQUEST parameter of the command.

If a command is executed via a request and returns something in the output buffer, the content of the distant output buffer is immediately available as the local output buffer.

The requests have a session scope so 2 sessions can have a request with the same name pointing to different servers.

CLOSE

REQUEST:CLOSE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command closes a request previously opened with the OPEN command.

Parameters

NV_REQUEST	Request name as returned by the SYSTEM REQUEST OPEN command. This is a string that uniquely identifies the request. It’s case independent. It must correspond to an existing request.

OPEN

REQUEST:OPEN

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command opens a new request for connecting distant NIRVA server.

Parameters

NAME

Request name. This is a string that uniquely identifies the request. It's case independent. If not provided, NIRVA creates itself the request name and returns it in the output buffer.

REOPEN

Reopen option. If this parameter is set to "YES" and a request with the same name already exists, the command closes it first and reopens it. If this parameter is set to "NO" and a request with the same name already exists, the command fails. The default is "NO".

SERVER

Distant Nirva server TCP/IP address or machine name. This can be follow by the TCP/IP port if this one is different than the default port (1081 for HTTP and 1082 for HTTPS). The TCP/IP address and port must correspond to a valid machine on which the Nirva server is installed. For example `Server="135.12.13.14:2035"` is a valid Server address. The default value for this parameter is `"127.0.0.1:1081"`.
A proxy server can also be defined in a single address. At this time the format is the following: `proxy::protocol://proxyserver:proxyport(proxyuser;proxypassword)::target_address` where *protocol* is the protocol for the proxy server. It can be "http" or "https"; *proxyserver* is the address of the proxy server, *proxyport* is the TCP/IP port of the proxy server, *proxyuser* and *proxypassword* are the user and password for the proxy if the proxy requires authentication (if no authentication required, the parentheses can be omitted) and *target_address* is the final address of the server as defined previously.
If the SERVER parameter is enclosed in brackets, Nirva considers it as a virtual host and will connect to one of the real hosts defined for this virtual host. Virtual hosts are defined from

	the Nirva configuration tool. They are very useful in multi server architecture in failover or load balancing mode
APPLICATION	Name of the distant Nirva application to work on. This parameter is significant only when a new session is to be created (so when the *SESSION* parameter is not provided or blank). In fact, a session is always opened in the context of an application. If the application parameter is not provided, Nirva uses the same application than the currentlt connected one (on the local server).
USER	Application user name for the distant Nirva server. This parameter is significant only when a new session is to be created (so when the *SESSION* parameter is not provided or blank). If not provided, NIRVA uses the current connected user name (on the local server).
PASSWORD	Application user password for the distant Nirva server. This parameter is significant only when a new session is to be created (so when the *SESSION* parameter is not provided or blank). If not provided, NIRVA uses the current connected user password (on the local server).
NEW_PASSWORD	This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
NEW_PASSWORD_CONFIRM	This parameter may be provided when the NEW_PASSWORD parameter has been given. It allows Nirva to check if the new password is correct.
SESSION	Distant Nirva session ID. If the parameter is given, Nirva tries to connect to an existing session (when sending the first server command) and produces an error if the session doesn't exist. If the parameter is not given, Nirva creates a new session. The closing of the session is controlled by the CLOSE_MODE parameter.
OPEN	Name of the procedure that NIRVA calls when opening this session. The default value is “session_open”. This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar). This parameter is significant only when a new session is to be created (so when the SESSION parameter is not provided or blank).
CLOSE	Name of the procedure that NIRVA calls when closing the session. The default value is “session_close”. This can be a native, java, dotnet or perl procedure. See the

	description of the NV_PROC parameter for the syntax of the procedure name. This parameter is significant only when a new session is to be created (so when the SESSION parameter is not provided or blank). For a java procedure, only a class file is accepted (no jar).
CLOSE_MODE	Defines the way to close the distant Nirva server session. If the CLOSE_MODE session is set to "MANUAL", the user must explicitly close the distant session by issuing a SYSTEM SESSION CLOSE command (or by a time out). If the CLOSE_MODE session is set to "COMMAND", the distant session is automatically close after sending a command so a new distant session is created for each command. If the CLOSE_MODE session is set to "SESSION", the distant session stays opened until the local session is not closed or the request is closed (except if the user sends an explicit SYSTEM SESSION CLOSE command for the distant session). The default value is "CLOSE_SESSION".
SESSION_TIME_OUT	Time out value in seconds for the distant session. It's used only when a new session is to be created (so when the *SESSION* parameter is not provided or blank). If *SESSION_TIME_OUT* is "0", the default Nirva time out will be used. This is the default
CONNECTION_TIME_OUT	Time out for establishing the TCP/IP connection to the Distant Nirva server. The default value is 10 seconds.
SSL	SSL mode. If this parameter is set to "YES", the request will use the HTTPS protocol to communicate to the distant NIRVA server, assuming encrypted data. The HTTPS server must have been enabled on the distant NIRVA server.
SOCKET_LOG	Complete path name of a log file where Nirva will write all read/writes on the used socket.

SCHEDULER class

The NIRVA scheduler allows running tasks with a defined periodicity.

A scheduled task is simply a NIRVA procedure written in native, perl, dotnet or java language. A task is always defined at application level but the task procedure can be also system or service procedures.

When the scheduler allows the task execution, it creates a session in a separate thread in order to run the task in background. This sessions runs in the application context under the user account defined in the task parameters.

A task may also use some procedures for reporting a successful completion or an error. This can be use for example to send an E-Mail to the administrator or to write log entries.

The scheduler defines several types of frequency for running a single task:

- One time
- Daily
- Weekly
- Monthly day based
- Monthly week based

For example, one can define a task that will run every last working week day of a month between 21:30:00 to 03:00:00 the day after.

The frequency type sets the days where the task should start. Now, along these days, a task is authorized to run from a given time and with a given time span that can be nearly one day long. This defines a period of time from 1 second to 23 hours 59 minutes and 59 seconds to run the task. During this period, the task can be repeated a given number of times with a defined delay between each occurrence.

In any case, the day for running a task is the day of the starting period. So if the period reaches midnight, the day is considered to be the day before.

For example, if a task is defined to run the 1st of March between 23:00:00 and 01:00:00 with a repeat mode set to infinite occurrences and a delay of 10 seconds between occurrences. Then, some of the tasks occurrences will run after midnight so on the March 2, but their starting date will be considered to be March1st.

A task can be disabled temporarily from the scheduler but stays defined. This allows avoiding some task to be started if necessary.

Any task can also be immediately run by a dedicated command even if it’s disabled for the scheduler. This can be used to run any kind of process in background.

Nirva provides some visual tools for creating, viewing and controlling the application scheduled tasks.

CREATE_TASK

SCHEDULER:CREATE_TASK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command creates a new scheduled task. After creation the task is registered but it has no associated procedure so it’s necessary to use the SCHEDULER:SET_TASK_PARAM command in order to set the task parameters.

The command fails if the task already exists.

Permissions

SCHEDULER_ADMIN

Parameters

NAME	Task name. The task name should not contain any special character. The task name is case insensitive and is mandatory.
DESCRIPTION	Task description.

ENABLE_TASK

SCHEDULER:ENABLE_TASK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command enables or disables a task. A disabled task is not candidate for the scheduler but can be executed manually (in background) with the SCHEDULER:RUN_TASK command.

If the task instance currently runs while disabling, it continue to run normally and only the next instance will be disabled.

Permissions

SCHEDULER_ADMIN

Parameters

NAME	Name of the task.
ENABLE	“YES” to enable the task and “NO” to disable it.

GET_TASK_PARAM

SCHEDULER:GET_TASK_PARAM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns complete information about a specific task.

The GET_TASK_PARAM command returns information in a NIRVA indexed string list object named "TASK_PARAM" in the output container.

Please see the SET_TASK_PARAM command description for further information about task parameters.

Permissions

SCHEDULER_ADMIN

Parameters

NAME Name of the task.

Objects created

TASK_PARAM This is a Nirva indexed string list object containing the complete task information. Here are the available strings:

- "NAME" is the task name.
- "DESCRIPTION" is the task description.
- "STATUS" is the current task status. It can be "NOT_STARTED", "IN_QUEUE", "RUNNING", "FAILED", "SUCCESS" or "STOPPED".
- "ENABLE" is set to "YES" if the task is enabled and to "NO" otherwise.
- "LAST_START_TIME" is the date and time of the last task start. If it's the same day than the current one, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS. It can be empty if the task has never been started.
- "LAST_END_TIME" is the date and time of the last task end. If it's the same day than the current one, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS. It can be empty if the task has never been started.

- “START_DATE” is the scheduled start date. The returned format is YYYY/MM/DD.
- “START_TIME” is the scheduled start time. The returned format is HH:MM:SS.
- “TIME_SPAN” is the time span in seconds.
- “ERROR” gives task error information if there is one available. This is the error information of the last task occurrence.
- “USER” is the user name used to run the task.
- “PROCT” is the task run procedure.
- “PROCF” is the task failed procedure.
- “PROCS” is the task success procedure.
- “PROCI” is the task init session procedure.
- “PROCE” is the task cleanup session procedure.
- “FREQUENCY” is the scheduled frequency it can be “ONETIME”, “DAILY”, “WEEKLY” or “MONTHLY”.
- “REPEAT” is the repeat mode. “YES” or “NO”.
- “DELAY” is the number of seconds between each occurrence in repeat mode.
- “START_REPEAT_MODE” is the start repeat mode. It can be “END_RUN”, “START_RUN” or “START_RUN_FIXED”.
- “OCCURENCES” is the maximum number of occurrences in repeat mode (0 means not limited).
- “STOP_REPEAT_MODE” is the stop repeat mode. It can be “NOSTOP”, “ONERROR” or “ONSUCCESS”.
- “DAY_EVERY” is the day span for a DAILY frequency.
- “WEEK_EVERY” is the week span for a WEEKLY frequency.
- “WEEK_DAYS” is the authorized days for a WEEKLY frequency. It’s a collection of values from 1 to 7 separated by a semicolon character (‘;’). 1 is Monday and 7 is Sunday.
- “MONTHS_MONTHS” is the authorized months for a MONTHLY frequency. It’s a collection of values from 1 to 12 separated by a semicolon character (‘;’). 1 is January and 12 is December.
- “MONTH_DAY_BASED” is set to “YES” if the MONTHLY frequency is day based and to “NO” if it’s week based.
- “MONTH_DAYS” is the authorized month days for a MONTHLY DAY BASED frequency. It’s a collection of values from 1 to 31 separated by a semicolon character (‘;’).

- “MONTH_WEEK” is the month week for a MONTHLY WEEK BASED frequency. It can be “FIRST”, “SECOND”, “THIRD”, “FOURTH” or “LAST”.
- “MONTH_WEEK_DAY” is the month week day for a MONTHLY WEEK BASED frequency. It’s a value from 1 to 7 where 1 is Monday and 7 is Sunday.

REMOVE_TASK

SCHEDULER:REMOVE_TASK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes the given task.

The removing is possible only if the task is not in use.

Permissions

SCHEDULER_ADMIN

Parameters

NAME	Name of the task to remove.
WAIT	Number of seconds to wait for the task to be free if it’s in use. The default is 10 seconds.

RUN_TASK

SCHEDULER:RUN_TASK

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command immediately runs a task independently of the scheduler time parameters and even if the task has been disabled. This feature can be used to run any kind of process in background.

Permissions

SCHEDULER_ADMIN

SCHEDULER_RUN_MANUAL (only if there is no SCHEDULER_ADMIN permission)

Parameters

NAME	Name of the task.

SET_TASK_PARAM

SCHEDULER:SET_TASK_PARAM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command sets one several of the task parameters. If a parameter is not given, it will not be changed.

Permissions

SCHEDULER_ADMIN

SCHEDULER_RUN_OTHER_USER (for USER parameter only)

Parameters

NAME	Task name.
DESCRIPTION	This is a free string describing the task.

USER	User name that will run the task. The task is always run in background by creating a new session. By default, the task runs on the name of the user who has created it but it's possible to change to another user name. This possibility is subject to a special permission in the user security profile. If the User field is let blank, NIRVA uses the current user.
PROCT	Name of the procedure to be ran by the task. This parameter is required or the task will never run. The task procedure receives the parameter NV_TASK_NAME that contains the task name. Please see the Calling a procedure chapter for description of the procedure syntax.
PROCF	Name of a procedure that will be ran in case of failure. This parameter is not mandatory. The failed procedure can be used for error reporting. It receives the error information as following parameters: NV_TASK_NAME, NV_ERROR_CODE, NV_ERROR_SERVICE, NV_ERROR_CLASS, NV_ERROR_DESC and NV_ERROR_INFO.
PROCS	Name of a procedure that will be ran in case of success. This parameter is not mandatory. The task success procedure receives the parameter NV_TASK_NAME that contains the task name
PROCI	Name of a procedure that will be ran before starting the task procedure itself. In fact, when the scheduler starts an instance of a task, it creates a new session and closes it after the end of the task. This parameter is used as the session open procedure name and is executed in the same conditions than for a normal user session (possibility to set specific session permission, for example). This parameter is not mandatory.
PROCE	Name of a procedure that will be ran after ending the task procedure itself. In fact, when the scheduler starts an instance of a task, it creates a new session and closes it after the end of the task. This parameter is used as the session close procedure name and is executed in the same conditions than for a normal user session. This parameter is not mandatory.
FREQUENCY	This defines the frequency of the task. It can be "ONETIME" for a one shot task, "DAILY", "WEEKLY" or "MONTHLY".
START_DATE	Minimum starting date of the task. The task will not run before this date. The general format of the START_DATE parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year. Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format. The separator can be '/', '\', '.', ' '(space) or ':'. It's also possible to not give the separator. If only DD/MM is given, NIRVA is using the current year. If only DD is given, NIRVA is using the current month and year. NIRVA can also accept the year on 2 digits. If the START_DATE is preceded by a plus sign ('+') followed by an integer. NIRVA considers that as a number of days after the current day. For

	example, “+3” is 3 days after the current day. If the START_DATE parameter is given but blank, NIRVA uses the current date. The initial task start date is the date of task creation.
START_TIME	Starting time of the task. For each day where the task is able to run, the scheduler defines a starting time so the task will not run before this time. The general format of the START_TIME parameter is HH:MM:SS where HH is the hour from 0 to 23, MM is the minute from 0 to 59 and SS is the second from 0 to 59. Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format. The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator. If only HHMM is given, NIRVA is using 0 for the seconds. If only HH is given, NIRVA is using 0 for the seconds and for the minutes. If the START_TIME is preceded by a plus sign (‘+‘) followed by an integer. NIRVA considers that as a number of hours after the current time. For example, “+3” is 3 hours after the current time. If the integer is followed by an “m” character, NIRVA considers that as a number of minutes after the current time. For example, “+10m” is 10 minutes after the current time. If the START_TIME parameter is given but blank, NIRVA uses the current time. The initial task start time is the time of task creation.
TIME_SPAN	Amount of time in seconds for the task to start after the start time. The parameters START_TIME and TIME_SPAN define a time window into which the task can run. For example, if START_TIME has been set 22:30:00 and TIME_SPAN to 03:00, the task will be able to run from 22:30:00 to 01:30:00 the next day. The minimum value is 1 second and the maximum is 86399 seconds. The initial task time span is 3600 seconds.
REPEAT	Repeat mode. Can be “YES” or “NO”. During the time span, the task can be eventually run several times. For example, it’s possible to schedule a task running each day from 23:00:00 to 01:00:00 with infinite occurrences and a delay of 10 seconds between each occurrence. The repeat mode can be disabled (So there is only one occurrence each day). The initial repeat mode is “NO”.
START_REPEAT_MODE	3 modes are available. “END_RUN”: The next occurrence starts after the defined delay of the last occurrence stop time. “START_RUN”: the next occurrence starts after the defined delay of the last occurrence start time. If the current occurrence is not terminated at the time of start, the next occurrence starts when the current one has finished. “START_RUN_FIXED”: the next occurrence starts after the defined delay of the last occurrence start time. If the current occurrence is not terminated at the time of start, the next occurrence doesn’t start.

DELAY	Minimum delay in seconds between 2 successive occurrences on the same day. This parameter is meaningful only if the repeat mode has been set. The minimum value is 1 second and the maximum is 82799 seconds. If the given value is blank, NIRVA assumes 60 seconds. The initial delay is 60 seconds.
OCCURENCES	Maximum number of occurrences on the same day. This parameter is meaningful only if the repeat mode has been set. If Occurrences is set to 0, the number of occurrences is not limited (in fact it's limited by the time span). If the given value is blank, NIRVA assumes 1 occurrence. The initial number of occurrences is 1.
CLEAR_OCC	Clear occurrence counter. The scheduler maintains a counter of the number of occurrences for each day. It's automatically reset to 0 at the end of the time span. It's possible to manually clear the counter by setting the CLEAR_OCC parameter to "YES".
STOP_REPEAT_MODE	Stop repeat mode. When the repeat mode is set, one can stop the task at first success or failure occurrence. This parameter can take values "NOSTOP", "ONERROR", "ONSUCCESS".
DAY_EVERY	This parameter has meaning only for a DAILY frequency. It defines a day span for running task. The minimum value is 1 day and the maximum is 365 days. If the given value is blank, NIRVA assumes 1 day. The initial value is 1 day.
WEEK_EVERY	This parameter has meaning only for a WEEKLY frequency. It defines a week span for running task. The minimum value is 1 week and the maximum is 52 weeks. If the given value is blank, NIRVA assumes 1 week. The initial value is 1 week.
WEEK_DAYS	This parameter has meaning only for a WEEKLY frequency. It defines the week days for running the task. This is a list of values from 1 to 7 separated by a semicolon character (';'). 1 is Monday and 7 is Sunday. The initial value is "1" (Monday).
MONTH_MONTHS	This parameter has meaning only for a MONTHLY frequency. It defines the allowed months for running the task. This is a list of values from 1 to 12 separated by a semicolon character (';'). 1 is January and 12 is December. The initial value is all months.
MONTH_DAY_BASED	This parameter has meaning only for a MONTHLY frequency. It allows choosing between a month's day or a month's week based frequency. If it's set to "YES", a month day based frequency is used, otherwise, a month week based frequency is used. The initial value is "YES".
MONTH_DAYS	This parameter has meaning only for a MONTHLY DAY BASED frequency. It defines the allowed month days for running the task. This is a list of values from 1 to 31 separated by a semicolon character (';'). The initial value is 1 (first day of the month).

MONTH_WEEK	This parameter has meaning only for a MONTHLY WEEK BASED frequency. It defines the single month week on which the task is authorized to run. The authorized values are “FIRST”, “SECOND”, “THIRD”, “FOURTH” and “LAST”. The initial value is “FIRST” (first week of the month).
MONTH_WEEK_DAY	This parameter has meaning only for a MONTHLY WEEK BASED frequency. It defines the day of the week on which the task is authorized to run. The authorized values are 1 to 7 where 1 is Monday and 7 is Sunday.

TASK_LIST

SCHEDULER:TASK_LIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the current application scheduled task list. For each task, the command returns the following information:

- Name.
- Description.
- Status
- Enable flag
- Frequency
- Last start time
- Last end time
- Error information

The TASK_LIST command returns information in a NIRVA table object named “TASK_LIST” in the output container.

Permissions

SCHEDULER_ADMIN

SCHEDULER_LIST (used only when SCHEDULER_ADMIN is not set)

Parameters

None

Objects created

TASK_LIST This is a Nirva table object containing the following columns:

- “NAME” is the task name.
- “DESCRIPTION” is the task description.
- “STATUS” is the current task status. It can be “NOT_STARTED”, “IN_QUEUE”, “RUNNING”, “FAILED”, “SUCCESS” or “STOPPED”.
- “ENABLE” is set to “YES” if the task is enabled and to “NO” otherwise.
- “FREQUENCY” is the scheduled frequency it can be “ONETIME”, “DAILY”, “WEEKLY” or “MONTHLY”.
- “LASTSTART” is the date and time of the last task start. If it’s the same day than the current one, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS. It can be empty if the task has never been started. If the task has been ran manually a “(M)” string follows the last start time.
- “LASTEND” is the date and time of the last task end. If it’s the same day than the current one, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS. It can be empty if the task has never been started.
- “ERROR” gives task error information if there is one available. This is the error information of the last task occurrence.

SECURITY class

The SECURITY class provides commands for manipulating system and application security.

The security model is described in the “Security model” chapter in this documentation.

ADD_ROLE

SECURITY:ADD_ROLE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Adds a new role to the global role list of the security. If the role already exists, the command fails.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	New role name. The role name is case insensitive and should not contain any space or special character. The role name is mandatory.
DESCRIPTION	Optional role description.

ADD_SESSION_PERMISSION

SECURITY:ADD_SESSION_PERMISSION

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure	No	No	No

Description

This command adds a permission or a role to the list of permissions of the user of the current session. This command can be called only from the application open session procedure.

This command is useful when the application get permissions or role names from an external system (LDAP for example).

If a permission is given, Nirva just add the permission to the list of permissions of the session. If a role is given, Nirva first gets the role permissions from the role defined at system or application security and then adds all the found permissions to the user session.

Parameters

SERVICE	Service permission or role. This parameter must be set to the name of a service for adding a service permission, to the value “SYSTEM” for adding a system permission or role, to the name of a web service enclosed in brackets for adding a web service permission and must be empty for adding an application permission.

PERMISSION	Permission to add.
ROLE	Role to add. If this parameter is provided, Nirva adds all the permissions of the role to the user session. This parameter cannot be used if the security of the application has been defined on another server or in an external security service.

ADD_USER

SECURITY:ADD_USER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Adds a new user. If the user already exists, the command fails.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	New user name. The user name is case insensitive and should not contain any space or special character. The user name is mandatory.
DESCRIPTION	Optional user description.
FULLNAME	Optional user full name.
PASSWORD	Optional user password.
PASSWORD_EXPIRE	Set the expiration time in days for the new user. This can be set to “-1” for no expiration, to a positive value or to the value

	"DEFAULT" in order to use the default defined at security level. The default value is "DEFAULT".
PASSWORD_CHANGE_FIRST	If this parameter is set to "YES", the user must change its password at the next logon. If set to "NO", he will not have to change its password at next logon. If this parameter is not provided, Nirva uses the default value defined at security level.

CHECK

SECURITY:CHECK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command verifies the given permission for the connected user.

Parameters

SERVICE	Service permission. This parameter must be set to the name of a service for checking a service permission, to the value "SYSTEM" for checking a system permission, to the name of a web service enclosed in brackets for checking a web service permission and must be empty for checking an application permission.
PERMISSION	Permission to check. The permission names are given in the SYSTEM service reference (in this documentation) for the system permissions, in the service documentations for the service permissions and in the application documentations for the application permissions. For web service, the permission names are directly the web service operation names.
PROCEDURE	Optional name of a procedure that will do a second check of the permission. The second check occurs only if the first check is OK so if the user has effectively the permission. Before calling the procedure, NIRVA creates a boolean object named "NV_PERMISSION_OK" in the input container and initializes it to a TRUE value. The procedure must set the value of the "NV_PERMISSION_OK" boolean object to FALSE if the permission is not to be accepted. This feature allows extending some security permissions to external systems (for example by checking a flag in a database).

	If the PROCEDURE parameter is not given or is blank, NIRVA doesn't proceed to a second check.
AUTHORIZED	Name of a boolean object in which NIRVA will write the result of the command. If the AUTORIZED parameter is given, the command always successes (expect in case of major failure) and sets the object which name is given by "AUTHORIZED" to TRUE or FALSE depending if the user has the required permission or not. The boolean object is created by the command in the output container. If the AUTORIZED parameter is not given, the command returns an error if the user doesn't have the required permission.

Objects created

<boolean>	The command creates a boolean object if the AUTHORIZED parameter is not empty. The object name is the one given by the AUTHORIZED parameter. Please see the description of the AUTHORIZED parameter.

ENABLE_USER

SECURITY:ENABLE_USER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Enables or disables a user. The "nvadmin" user cannot be disabled.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).

USER	Name of the user to enable or disable.
ENABLE	If this parameter is set to “NO”, the command is to disable the user. Otherwise, the command is to enable the user (default).

INFO

SECURITY:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information at security level:

- Default password expiration time
- Change password at next logon flag
- Number of bad logon attempts before the user to be locked
- The name of a procedure for checking the new password syntax. This procedure, if it exists, will be launched by Nirva when the user is changing its password. It will receive 3 parameters OLD_PASSWORD containing the old password, NEW_PASSWORD containing the new password and USER containing the user name. If the procedure produces an error (SetError function from Perl, Dotnet or Java procedure), Nirva will return a bad password syntax error to the user.
- The number of days after which unused users are automatically removed from the security

The INFO command returns information in a NIRVA indexed string list object named “INFO” in the output container.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).

Objects created

INFO This is a Nirva indexed string list object containing the following keys:

- "PASSWORD_EXPIRE" is the default expiration time in days for new users. If the value is "-1", the password never expires.
- "PASSWORD_CHANGE_NEXT" flag tells if the new users must change their password at next logn ("1") or not ("0").
- "USER_TIME_OUT" is the number of days after which unused users are automatically removed from the security (-1 means no automatic removal).
- "NEW_PASSWORD_CHECK_PROC" is the name of a procedure for checking the new password syntax.
- "LOCKING_ATTEMPTS" is the number of bad logon attempts before the user to be locked.

LOAD_USER_CONTEXT

SECURITY:LOAD_USER_CONTEXT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command retrieves from the output container the user context previously saved by the SAVE_USER_CONTEXT command on the security system.

The user context is a specific container associated to the user saved in the registry. It allows keeping user specific information.

The user context works even if security is defined on another application or server. If the security used is not the Nirva security but an external security service, the LOAD_USER_CONTEXT may not work if it has not been implemented in the third party security service.

The output container will be erased first and then populated by the entire user context.

Parameters

none

PERMISSION_LIST

SECURITY:PERMISSION_LIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the list of available permissions.

The PERMISSION_LIST command returns information in a NIRVA table object named “PERMISSION_LIST” in the output container.

If the target security is the system security, the returned permissions are the system permissions and the service permissions.

If the target security is the application security, the returned permissions are the system permissions, the service permissions and the application permissions.

This returns always the permissions of the connected application, even if the application is using the security of another application.

Parameters

SYSTEM — If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).

Objects created

PERMISSION_LIST — This is a Nirva table object containing the following columns:

- “NAME” is the permission string. A permission string has the format *Service:PermissionName* where *Service* is the eventual service name or web service name (enclosed in brackets) and *PermissionName* is the permissions name as defined in the system, service or application description file (dsc file). If the permission is an application permission, the permission string is directly the permission name. If the permission is a web service permission, *Service* is the name of the web service enclosed in brackets and *PermissionName* is the name of

the web service operation. See the chapter about the security model for further information.

- "DESCRIPTION" is the permission description.

REMOVE_ROLE

SECURITY:REMOVE_ROLE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Removes a role if it exists.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Name of the role to remove.

REMOVE_ROLE_INHERITS

SECURITY:REMOVE_ROLE_INHERITS

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Remove some of the role inherited roles.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Role name.
INHERITS	List of inherited roles to remove. If there are several roles, they must be separated by a semicolon character (‘;’).
ALL	If this parameter is set to “YES”, the command removes all the inherited roles whatever the content of the INHERITS parameter.

REMOVE_ROLE_PERMISSIONS

SECURITY:REMOVE_ROLE_PERMISSIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Remove some of the role permissions.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Role name.
PERMISSIONS	List of permission strings to remove. If there are several permissions, they must be separated by a semicolon character (‘;’). A permission string has the format *Service:PermissionName* where *Service* is the eventual service name or web service name (enclosed in brackets) and *PermissionName* is the permissions name as defined in the system, service or application description file (dsc file). If the permission is an application permission, the permission string is directly the permission name. If the permission is a web service permission, *Service* is the name of the web service enclosed in brackets and *PermissionName* is the name of the web service operation. See the chapter about the security model for further information.
ALL	If this parameter is set to “YES”, the command removes all the permissions whatever the content of the PERMISSIONS parameter.

REMOVE_SESSION_PERMISSION

SECURITY:REMOVE_SESSION_PERMISSION

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure	No	No	No

Description

This command removes a permission from the list of permissions of the session’s user. This command can be called only from the application open session procedure.

Parameters

SERVICE	Service permission. This parameter must be set to the name of a service for removing a service permission, to the value “SYSTEM” for removing a system permission, to the name of a web service enclosed in brackets for

	removing a web service permission and must be empty for removing an application permission.
PERMISSION	Permission to remove.

REMOVE_USER

SECURITY:REMOVE_USER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Removes a user if it exists. The “nvdef” and “nvadmin” users cannot be removed.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	Name of the user to remove.

REMOVE_USER_ROLES

SECURITY:REMOVE_USER_ROLES

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Remove some of the user roles.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	User name.
ROLES	List of roles to remove. If there are several roles, they must be separated by a semicolon character (‘;’).
ALL	If this parameter is set to “YES”, the command removes all the user roles whatever the content of the ROLES parameter.

ROLE_EXIST

SECURITY:ROLE_EXIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

Tests if the given role exists. The output buffer is set to “YES” if the given role exists and to “NO” otherwise.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Name of the role to test.

ROLE_LIST

SECURITY:ROLE_LIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information about one or several roles:

- Name.
- Description.
- Inherited roles
- Permissions

The ROLE_LIST command returns information in a NIRVA table object named "ROLE_LIST" in the output container.

If the parameter "ROLE" is given, the command just reports information on the requested role.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Name of the role to get information. If this parameter is not given, the command returns information for all the roles.

Objects created

ROLE_LIST This is a Nirva table object containing the following columns:

- "NAME" is the role name.
- "DESCRIPTION" is the role description.
- "INHERITS" is the list of inherited roles for the role.
- "PERMISSIONS" is the list of permissions for the role.

SAVE_USER_CONTEXT

SECURITY:SAVE_USER_CONTEXT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command saves input container as the user context on the security system.

The user context is a specific container associated to the user saved in the registry. It allows keeping user specific information.

The user context works even if security is defined on another application or server. If the security used is not the Nirva security but an external security service, the SAVE_USER_CONTEXT may not work if it has not been implemented in the third party security service.

The input container must point to the container to save as user context.

Parameters

none

SESSION_PERMISSIONS

SECURITY:SESSION_PERMISSIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the list of permissions for the user session.

The SESSION_PERMISSIONS command returns information in a NIRVA string list object named "PERMISSION_LIST" in the output container.

If the connected user is nvadmin, this command doesn't fail but returns an empty list.

Parameters

APPONLY — By default the command retrieves only the application permissions. If this parameter is set to "NO", all the permissions are retrieved.

Objects created

PERMISSION_LIST — This is a Nirva string list object containing the list of session permissions. A permission string has the format *Service:PermissionName* where *Service* is the eventual service or web service name (enclosed in brackets) and *PermissionName* is the permissions name as defined in the system, service or application description file (dsc file). If the permission is an application permission, the permission string is directly the permission name. If the permission is a web service permission, *Service* is the name of the web service enclosed in brackets and *PermissionName* is the name of the web service operation. See the chapter about the security model for further information.

SET_OTHER_USER_PASSWORD

SECURITY:SET_OTHER_USER_PASSWORD

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Changes the password of another user. The user must exist. Only the nvadmin user can change its password so the command will fail if attempted to change the nvadmin password.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

SECURITY_CHANGE_OTHER_PASSWORD

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	User name.
PASSWORD_NEW	New user password.

SET_PARAM

SECURITY:SET_PARAM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Set some security level parameters. Only the parameters given will be changed.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
PASSWORD_EXPIRE	Set the default expiration time in days for the new users. This can be set to "-1" for no expiration or to a positive value.
PASSWORD_CHANGE_FIRST	If this parameter is set to "YES", the new user must change its password at the next logon. If set to "NO", he will not have to change its password at next logon.
USER_TIME_OUT	The number of days after which unused users are automatically removed from the security. If set to -1, no user removal occurs.
NEW_PASSWORD_CHECK_PROC	The procedure name for checking the new password syntax. This procedure, if it exists, will be launched by Nirva when the user is changing its password. It will receive 3 parameters OLD_PASSWORD containing the old password, NEW_PASSWORD containing the new password and USER containing the user name. If the procedure produces an error (SetError function from Perl, Dotnet or Java procedure), Nirva will return a bad password syntax error to the user.
LOCKING_ATTEMPTS	Number of bad logon attempts before the user to be locked. If set to -1, no locking occurs.

SET_ROLE_DESCRIPTION

SECURITY:SET_ROLE_DESCRIPTION

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Changes the role description. The role must exist.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Role name.
DESCRIPTION	New role description.

SET_ROLE_INHERITS

SECURITY:SET_ROLE_INHERITS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Appends inherited roles to the role.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Role name.
INHERITS	List of inherited roles to append. If there are several inherited roles, they must be separated by a semicolon character (';').
APPEND	Append mode. If this parameter is set to "YES", the command appends the given inherited roles. Otherwise, it first removes all the role inherited roles and then adds the given inherited roles. The default value is "NO" (no append).

SET_ROLE_PERMISSIONS

SECURITY:SET_ROLE_PERMISSIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Appends permissions to the role.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
ROLE	Role name.
PERMISSIONS	List of permission strings to append. If there are several permissions, they must be separated by a semicolon character (';'). A permission string has the format *Service:PermissionName* where *Service* is the eventual service name or web service name (enclosed in brackets) and *PermissionName* is the permissions name as defined in the system, service or application description file (dsc file). If the permission is an application permission, the permission string is directly the permission name. If the permission is a web service permission, *Service* is the name of the web service enclosed in brackets and *PermissionName* is the name of the web service operation. See the chapter about the security model for further information.
APPEND	Append mode. If this parameter is set to "YES", the command appends the given permissions. Otherwise, it first removes all the role permissions and then adds the given permissions. The default value is "NO" (no append).

SET_USER_DESCRIPTION

SECURITY:SET_USER_DESCRIPTION

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Changes the user description. The user must exist.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	User name.
DESCRIPTION	New user description.

SET_USER_FULLNAME

SECURITY:SET_USER_FULLNAME

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Changes the user full name. The user must exist.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	User name.
FULLNAME	New user full name.

SET_USER_PARAM

SECURITY:SET_USER_PARAM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Change some user parameters. This command can be used instead of the SET_USER_FULLNAME and SET_USER_DESCRIPTION commands.

Only the given parameters will be changed.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	User name. The user name is case insensitive and should not contain any space or special character. The user name is mandatory.
DESCRIPTION	User description.
FULLNAME	User full name.
PASSWORD_EXPIRE	Set the expiration time in days for the user. This can be set to “-1” for no expiration, to a positive value or to the value “DEFAULT” in order to use the default defined at security level.
PASSWORD_CHANGE_FIRST	If this parameter is set to “YES”, the user must change its password at the next logon. If set to “NO”, he will not have to change its password at next logon.

SET_USER_PASSWORD

SECURITY:SET_USER_PASSWORD

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Changes the user password.

Permissions

SECURITY_CHANGE_OWN_PASSWORD

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
PASSWORD_OLD	Old user password. The command checks the old user password that must be the same than the current connected user.
PASSWORD_NEW	New user password. Can be blank. The password is case sensitive but cannot contain blanks or special characters.
PASSWORD_NEW_CONFIRM	New user password. Can be blank. This value is compared to the value given in the PASSWORD_NEW parameter.

SET_USER_ROLES

SECURITY:SET_USER_ROLES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Appends roles to the user.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	User name.
ROLES	List of roles to append. If there are several roles, they must be separated by a semicolon character (‘;’).
APPEND	Append mode. If this parameter is set to “YES”, the command appends the given roles. Otherwise, it first removes all the user roles and then adds the given roles. The default value is “NO” (no append).

USER_EXIST

SECURITY:USER_EXIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

Tests if the given user exists. The output buffer is set to “YES” if the given user exists and to “NO” otherwise.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to “YES”, the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	Name of the user to test.

USER_LIST

SECURITY:USER_LIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information about one or several users:

- Name.
- Description.
- Full name
- Roles
- Enable flag
- Expiration time
- Change password at next logon flag

The USER_LIST command returns information in a NIRVA table object named “USER_LIST” in the output container.

If the parameter “USER” is given, the command just reports information on the requested user.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security).
USER	Name of the user to get information. If this parameter is not given, the command returns information for all the users.

Objects created

USER_LIST This is a Nirva table object containing the following columns:

- "NAME" is the user name.
- "DESCRIPTION" is the user description.
- "FULLNAME" is the user full name.
- "ROLE" is the list of roles for the user.
- "ENABLE" is the enable flag. It has the value "YES" if the user is enabled and "NO" otherwise.
- "PASSWORD_EXPIRE" is the expiration time in days. If the value is "-1", the password never expires.
- "PASSWORD_CHANGE_NEXT" flag tells if the user must change its password at next logon ("1") or not ("0").

USER_PERMISSIONS

SECURITY:USER_PERMISSIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the list of permissions for the given user.

The USER_PERMISSIONS command returns information in a NIRVA table object named "PERMISSION_LIST" in the output container.

This command fails for the nvadmin user because this user is defined outside the security tables and has all rights.

This command cannot be used if the security of the application has been defined on another server or by an external security service. It then produces an error message.

Permissions

SYSTEM_SECURITY_ADMIN

APPLICATION_SECURITY_ADMIN

Parameters

SYSTEM	If this parameter is set to "YES", the command will address the system security. Otherwise, it will use the application security (that may be itself redirected to system or another application security). The default is "NO".
USER	Name of the user.

Objects created

PERMISSION_LIST This is a Nirva table object containing the following columns:

- "NAME" is the permission string. A permission string has the format *Service:PermissionName* where *Service* is the eventual service or web service name (enclosed in brackets) and *PermissionName* is the permissions name as defined in the system, service or application description file (dsc file). If the permission is an application permission, the permission string is directly the permission name. If the permission is a web service permission, *Service* is the name of the web service enclosed in brackets and *PermissionName* is the name of the web service operation. See the chapter about the security model for further information.
- "DESCRIPTION" is the permission description.

SEMAPHORE class

The SEMAPHORE class provides commands for managing semaphore objects.

The semaphores allow to control simultaneous accesses to a shared limited resource. It's a very good tool for tuning very complex applications on a machine with a limited number of resources.

A Nirva semaphore object is a named object having a status locked or unlocked and an initial value. Each time a thread wants to access the resource, the value is decreased. When the value reaches 0, the semaphore object is locked and all threads requesting it are suspended while the value stays at 0. When a thread has got the access to the semaphore, it does its processing and unlocks the semaphore. Unlocking a semaphore means increasing its value by one.

When a session is closed, all the semaphore objects it owns are automatically free.

Nirva maintains semaphore lists at system and application levels.

Since NIRVA checks the session time outs every 30 seconds. An semaphore owned by a timeouted session can stay locked a maximum of 30 seconds after the time out occurs.

INFO

SEMAPHORE:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the current system or application semaphore list (or a part of it). For each semaphore object, the command returns the following information:

- Name.
- In use (number of threads having locked the semaphore).
- Value (maximum simultaneous accesses)

The INFO command returns information in a NIRVA table object named “INFO” in the output container.

It’s possible to reduce the list by searching for some specific semaphore object names.

Parameters

NAME Name of the semaphore object. This parameter, when provided, is used to restrict the list to specific semaphore object names. The wildcard character '*' can be used for searching object names starting with a given value. If NAME is “*”, NIRVA doesn’t restrict the list based on the semaphore name. This is the default.

SYSTEM System level lock.
By default, this parameter is set to “NO” so Nirva returns the application semaphore list. If SYSTEM is set to “YES”, Nirva returns the system semaphore list.

Objects created

INFO This is a Nirva table object containing the following columns:

- “NAME” is the semaphore object name.

- “INUSE” is the number of sessions which want to access the semaphore.
- “VALUE” is the maximum number of sessions allowed to gain access to the semaphore at the same time.

LOCK

SEMAPHORE:LOCK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command tries to gain access to the given semaphore object. If the semaphore object doesn’t exist, Nirva creates it automatically. If the lock is successful, the session becomes one of the lock owners. It will own the lock until it unlocks it or until the session closes.

If the semaphore value is greater than 0, the session immediately gets the access to the semaphore. Otherwise, it waits until the semaphore value to be greater than 0.

If the command cannot get the object locked, it returns an error message.

Parameters

NAME	Name of the semaphore object. The semaphore name is case insensitive. This parameter is mandatory.
VALUE	Semaphore initial value. This is the maximum number of simultaneous accesses to the resource. This parameter has meaning only when the semaphore is to be created.
SYSTEM	System level semaphore. By default, this parameter is set to “NO” so the semaphore object is at application level. If SYSTEM is set to “YES”, the semaphore object is at system level. The application and system semaphore lists are completely independent so an object of the same name can be at application and system level.

REMOVE

SEMAPHORE:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes an unused semaphore. A semaphore is unused when no thread has locked it.

If the semaphore is in used, the command fails.

Generally, it's not necessary to remove a semaphore because NIRVA does that automatically every 60 seconds.

Parameters

NAME	Name of the semaphore object. The semaphore name is case insensitive. This parameter is mandatory.
SYSTEM	System level semaphore. By default, this parameter is set to "NO" so Nirva searches for the semaphore object in the application semaphore list. If SYSTEM is set to "YES", Nirva searches for the lock object in the system semaphore list.

UNLOCK

SEMAPHORE:UNLOCK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command unlocks a previously locked semaphore object. Unlocking a semaphore means increasing its value by one. Only a session having already locked the semaphore can unlock it.

If the command cannot unlock the semaphore, it returns an error message.

Parameters

NAME Name of the semaphore object. The semaphore name is case insensitive. This parameter is mandatory.

SYSTEM System level semaphore.
By default, this parameter is set to “NO” so Nirva searches for the semaphore object in the application semaphore list. If SYSTEM is set to “YES”, Nirva searches for the lock object in the system semaphore list.

SERVICE class

The SERVICE class provides commands for working with external services.

CONFIG

SERVICE:CONFIG

Source	Use Input Container	Use Output Container	Use Output buffer
Web	No	Yes	No

Description

Configures a service. This command calls the service configuration entry page. This page is an XSL page named “config.xsl” that should be in the service files config directory.

The command also calls a service procedure named “config.nvp” that should be in the service procedure config directory. If this procedure doesn’t exist, the command doesn’t fail. The procedure is executed before the command.

This command can be used only from a web browser. The eventual NV_XML_XSL parameter given in the command has no meaning and is internally replaced by the name of the config xslt page.

The service must have been mounted for this command to work.

Permissions

SERVICE_CONFIG

Parameters

SERVICE Name of the external service to configure. The service name is case insensitive.

IDENT

SERVICE:IDENT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command creates and returns a unique identifier that can be used by the service provider in the license management. The SERVICE SKELETON command also generates automatically this identifier.

Parameters

SERVICE Name of the external service. The service name is not controlled in this command. In fact, the string given in this parameter is simply used to generate the unique identifier. This parameter is not mandatory.

Objects created

IDENTIFIER Unique identifier created by the command. This is a string object of 32 characters length containing letters A to F or digits 0 to 9.

INFO

SERVICE:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some information about one or several services:

- Name.
- Description.
- Path

- Status
- Multithread flag
- Loads on start flag
- Language
- Number of connected sessions
- Application for which the service is required

The INFO command returns information in a NIRVA table object named “INFO” in the output container.

If the parameter “SERVICE” is given, the command just reports information on the requested service.

Permissions

SERVICE_LIST

Parameters

SERVICE	Name of the external service to get information from. If this parameter is not given or is blank, the command retrieves information about all services.

Objects created

INFO — This is a Nirva table object containing the following columns:

- “NAME” is the service name.
- “DESCRIPTION” is the service description.
- “PATH” is the library service path name (relative or absolute).
- “STATUS” is the service status. It has the value “RUNNING” if the service is running and “STOPPED” otherwise.
- “MULTITHREAD” is the multithread flag. It has the value “YES” if the service is multithread and “NO” otherwise.
- “LOAD_ON_START” is the load on start flag. It has the value “YES” if the service is configured to be loaded when starting Nirva and “NO” otherwise.
- “LANGUAGE” is the service language. It can be “C++”, “DOTNET” or “JAVA”.
- “NUM_SESSIONS” is the number of sessions currently connected to the service.
- “REQUIRED_FOR_APPLICATIONS” List of running applications for which the service is required. This is a comma separated list.

INFOEX

SERVICE:INFOEX

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns some service specific information. This is the information found in the INFO section of the service description file. See the external service chapter for further information about the service description file.

The INFOEX command returns information in a NIRVA indexed string list object named “SERVICE_INFO” in the output container.

Parameters

SERVICE Name of the external service to get information from. This parameter is mandatory.

Objects created

SERVICE_INFO This is a Nirva indexed string list object corresponding to the pairs found in the INFO section of the service description file. If there is no description file for the given service, the SERVICE_INFO object is created but is empty.

INSTALL

SERVICE:INSTALL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Installs a service package file previously created by the SYSTEM SERVICE PACKAGE command.

The service package file is a binary file that itself contains other files.

This command can only install files in the given service directory.

The service must have been previously stopped in order for the command to succeed.

The command also calls a service procedure named “install.nvp” that should be in the service procedure directory. If this procedure doesn’t exist, the command doesn’t fail. The procedure is executed after the command.

For security reasons, the service packages can only install service files on the target service directory. If some other files have to be installed in other server directories, this can be done by using the “install.nvp” procedure.

If the package file has a header entry PLATFORM = *Platform* where Platform is the target platform (WIN32, WIN64, AIX, LINUX, LINUX64, HPUX, HPUXI or SOLARIS), the INSTALL command checks if the target platform corresponds to the package file and returns an error if it’s not the case. See the chapter “Installation packages” for information about package file headers.

Permissions

SERVICE_INSTALL

Parameters

SERVICE	Name of the external service for which to install the package. The service name is case insensitive. If this parameter is not provided, the command tries to get it from the package file in the header entry “*SERVICE = SrvName*” where *SrvName* is the name of the service. A service name should contain only alphanumeric characters and the underscore character.
FILE	Name of the NIRVA file object that contains the package file to install. This object must be in the input container. The default value is "PACKAGE_FILE".
MOUNT	If this parameter is set to “YES”, the command also mounts the service, if it’s not already mounted, in the NIRVA service list. The mounting is made by running the SERVICE:MOUNT command with the default parameters. The command fails if the mounting fails but the installation part itself stays OK. At this time, the user will have to mount directly the service with the SERVICE:MOUNT command. The default value is “NO” (no mounting).

MOUNT

SERVICE:MOUNT

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Mounts a service. A service is a library. Mounting a service means telling to Nirva what is the name and path of the library to use.

The mount function can also be used to change the parameters of an existing and mounted service. For that, the service must be stopped with the STOP command otherwise, this command returns an error.

The service library must implement a function named NvsCommand.

When NIRVA starts, it automatically mounts the services that were previously mounted when NIRVA has been stopped.

Permissions

SERVICE_MOUNT

Parameters

SERVICE	Name of the external service to mount. The service name is case insensitive.
DESCRIPTION	External service description.
PATH	Complete path name of the library implementing the service. This can be a relative (to the service directory) or absolute path. If the PATH parameter is not given or is blank, NIRVA tries to locate the library in the service Bin directory.
MULTITHREAD	If this parameter is set to “NO”, the library is considered as not multithread and Nirva will serialize all the commands sent to it. The default is “YES” (multithread).
LOAD_ON_START	If this parameter is set to “YES”, the library will be loaded when starting the Nirva server. Otherwise, the library is loaded only when the first command is sent to it. The default is “NO”. Generally, a library should not be loaded at start time if not necessary in order to save memory space.
LANGUAGE	This can be “JAVA”, “DOTNET” or “C++”. The default is “C++”.

PACKAGE

SERVICE:PACKAGE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

Creates a service package file that will be usable by the SYSTEM SERVICE INSTALL command.

The service package file is a binary file that itself contains other files. The package file is created following the content of a package description file that must reside on the service files directory. The package file is compressed.

For security reasons, the service packages can only install service files on the target service directory.

Permissions

SERVICE_PACKAGE

Parameters

SERVICE — Name of the external service for which to create an installation package. The service name is case insensitive.

FILE — Name of the NIRVA file object that will contain the resulting package file. This object is in the output container. If the object doesn't exist, the command creates it. The default value is "PACKAGE_FILE".

DESC_FILE — Name of package description file. This must correspond to an existing file that resides in the service files directory. The default value is “package.lst”. The format of the package description file is given in the “installation packages” chapter. It's important (but not necessary) if the description file has a header entry “*SERVICE = SrvName*” where *SrvName* is the name of the service.

PROTECT_CONTAINERS

SERVICE:PROTECT_CONTAINERS

Source	Use Input Container	Use Output Container	Use Output buffer
Service	No	No	No

Description

Protects access to service containers from outside the service.

This command is generally called one time at service initialization.

This feature is available only from Nirva 3.0.005. Service providers should implement this command using the parameter NV_NO_ERROR set to “YES” in order to avoid an error message if the service runs on an older Nirva version.

Parameters

None

SESSIONS

SERVICE:SESSIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the list of session identifiers currently connected to the service. A session is connected to a service when the service has been started and the session made at least one request to the service.

Permissions

SERVICE_SESSION_LIST

Parameters

SERVICE Name of the external service.

Objects created

SESSIONS This is a Nirva string list object that contains the list of session identifiers currently used by the external service.

SKELETON

SERVICE:SKELETON

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Creates a skeleton of the service source code. This skeleton is created in the source directory of the service directory.

If the service is C++, this command creates the following files in the service source directory if they don't exist:

servicename.cpp	where *servicename* is replaced by the service name is the service C++ source file that contains the NvsCommand entry point and the eventual session class if the session option has been requested. The file extension is ".C" under AIX instead of ".cpp".
servicename.h	where *servicename* is replaced by the service name is the corresponding header file containing declarations of the Nirva command interface.
nirva.def	is a WINDOWS specific file that exports the NvsCommand function.
makefile.aix	is the AIX make file.
makefile.linux	is the LINUX make file.
makefile.solaris	is the SOLARIS make file.
makefile.hp	is the HPUX PA-RISCmake file.
makefile.hpi	is the HPUX Itanium make file.
servicename.dsp	where *servicename* is replaced by the service name is the Microsoft Visual Studio 6 project file.
servicename.dsw	where *servicename* is replaced by the service name is the Microsoft Visual Studio 6 workspace file.
nvthread.h	Header file that defines the class NirvaMutex that can be used by the programmer for managing accesses to shared resources from the multithreaded service. This file also defines a macro named SLEEP that allows to sleep the current thread for the given number of milliseconds.
nvthread.cpp	Implementation file of the NirvaMutex class. The file extension is ".C" under AIX instead of ".cpp".

If the service is Java, this command creates the following files in the service source directory if they don't exist:

servicename.java	where *servicename* is replaced by the service name is the service java source file that contains the service class and the eventual session class if the session option has been requested.

If the service is Dotnet, this command creates the following files in the service source directory if they don't exist:

servicename.cs	where *servicename* is replaced by the service name is the service c# source file that contains the service class and the eventual session class if the session option has been requested.
servicename.sln	where *servicename* is replaced by the service name is the service solution file for Visual C# 2008.
servicename.csproj	where *servicename* is replaced by the service name is the service project file for Visula C# 2008.
compile.bat	This is a batch file for compiling the service if Visual C# 2008 is not used.
Properties/assemblyinfo.cs	Assembly information file used by Visual C# 2008 to describe the assembly file.

It also creates the following files if they don't exist:

license.txt	This file is also in the Source directory. It contains the service private and public Ids for creating and checking licenses.
service.dsc	This is the service description file that contains some information that describe the service. This file is used by NIRVA when mounting the service. This file is created in the service files directory.
config.xsl	This is the xslt file used to create the service configuration WEB page when using the SYSTEM SERVICE CONFIG command. This page should be modified by the service programmer. This file is created in the subdirectory Config of the service files directory.
config.nvp	This is an empty procedure called by the SYSTEM SERVICE CONFIG command. This procedure should be modified by the service programmer if necessary. This file is created in the subdirectory Config of the service procedure directory.
servicename.htm	where *servicename* is replaced by the service name is the starting html page for service documentation. This page should be modified by the service programmer if necessary. This file is created in the service documentation directory.
package.lst	This is the installation package file. This file contains information for creating a NIRVA installation package for the service.

install.nvp	This is an empty procedure called by the SYSTEM SERVICE INSTALL command. This procedure should be modified by the service programmer if necessary. This file is created in the service procedure directory.

The service doesn't need to be mounted for this command to work.

Please read the comments in the generated files and refer to the external service chapter for further information about Nirva external service interface.

Permissions

SERVICE_SKELETON

Parameters

SERVICE	Name of the external service. A service name should contain only alphanumeric characters and the underscore character.
WITH_SESSIONS	If this parameter is set to "YES", Nirva creates the necessary code for service session management. If set to "NO", only the NvsCommand function is implemented. The default is "YES".
LANGUAGE	This can be "JAVA", "DOTNET" or "C++". The default is "C++".

START

SERVICE:START

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command starts the given external service if this one is not started. Starting a service consists of loading an external library. A service can be started immediately or on next command request to the service.

Permissions

SERVICE_RUN

Parameters

SERVICE	Name of the external service to start. The service name is case insensitive.
IMMEDIATE	Immediate mode. If this parameter is set to “YES”, the service is started immediately. If this parameter is set to “NO”, the service is started the next time a command requests an access to it. If the parameter is not provided, Nirva uses the “load on start” flag of the service to decide to run it immediately or not.

STOP

SERVICE:STOP

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command tries to stop the given external service. Stopping a service consists of unloading its code from memory. If a service is required by an application (listed in the Setting/required entry of the application dsc file), it cannot be stopped while the application is running.

Permissions

SERVICE_RUN

Parameters

SERVICE	Name of the external service to stop. The service name is case insensitive.
WAIT	Number of seconds to wait for the service to terminate. The service may need some time to terminate because it may have some commands in process. When stopping the service, Nirva blocks all new commands for the service and waits the given number of seconds for the service to finish its processing. If the service cannot terminate in the given number of seconds, it’s considered as not terminated and one can send some new commands to it. For waiting indefinitely, the WAIT parameter should be set to “-1”. The default for this parameter is “60” (60 seconds).

UNMOUNT

SERVICE:UNMOUNT

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

Unmounts a service. A service is a library. Unmounting a service means removing the link between Nirva and the library or java class or dotnet assembly.

When a service is mounted, Nirva creates some subdirectories in the Nirva service directory. These subdirectories are not removed when unmounting a service.

If the service is running, NIRVA tries to stop it for the given number of seconds. The command fails if it cannot stop the service.

Permissions

SERVICE_MOUNT

Parameters

SERVICE	Name of the external service to unmount. The service name is case insensitive.
WAIT	Number of seconds to wait for the service to terminate if it runs. The service may need some time to terminate because it may have some commands in process. When stopping the service, Nirva blocks all new commands for the service and waits the given number of seconds for the service to finish its processing. If the service cannot terminate in the given number of seconds, it’s considered as not terminated and one can send some new commands to it. For waiting indefinitely, the WAIT parameter should be set to “-1”. The default for this parameter is “60” (60 seconds).

SESSION class

The SESSION class provides commands for working with sessions.

CHECK_CLOSE_REQUEST

SESSION:CHECK_CLOSE_REQUEST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns “YES” in the output buffer if the session has been requested to close by an external user or by Nirva itself (when Nirva stops).

The CHECK_CLOSE_REQUEST command should be used especially in potentially long procedures or service commands, typically inside a loop to be able to break the loop in case of close request.

Parameters

None

CLOSE

SESSION:CLOSE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web	No	No	No

Description

This command closes a session. If the general parameter NV_SESSION_ID is given, the command closes the given session.

If the general parameter NV_SESSION_ID is not given, the command creates the session and then closes it. This can be very useful if the command calls one or several procedures. At this time, a session is created only for calling these procedures and immediately closed after.

It’s not possible to close the calling session from inside a service or a procedure.

If the session to close is in use, NIRVA cannot close it directly so it resets its time out to 0 in order for the session to close itself when it has finished its work.

Parameters

LOGOUT_PAGE

This parameter has meaning only when the command comes from a browser. The logout page is the name of the html page that nirva must send back after logout. If this parameter is blank, nirva tries to get the application default page.

CLOSE_OTHER

SESSION:CLOSE_OTHER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command closes another session that the calling one.

If the session to close is in use, NIRVA cannot close it directly so it resets its time out to 0 in order for the session to close itself when it has finished its work.

Permissions

SESSION_CLOSE_OTHER. This permission is checked only when the order comes from Client or Web (not from procedure or service).

Parameters

SESSION

Session to close (session ID). If the SESSION parameter is the same than the calling session, the command returns an error message. This parameter is mandatory.

WAIT

Maximum wait time if the session is in use. The default is 10 seconds. The SESSION_CLOSE_OTHER command will wait a certain time for the session to be closed. After this time, the session time out is well reset to 0 but it’s still in use. It will close itself automatically after it has finished its processing. The default is to wait 10 seconds. A value of 0 means to wait infinitely.

CREATE

SESSION:CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	Yes

Description

This command creates a new session. It can be used only from procedures or from a service. In other cases, session creation is controlled directly by the NV_SESSION_ID parameter or automatically for client connectors.

In fact, this is generally used in 2 cases:

- For an application to communicate with another application.
- For an application to create named sessions.

The newly created session ID is returned in the output buffer.

Parameters

APPLICATION
Name of the application to connect. If this parameter is not provided the application will be the same as the current one. If the parameter is provided but blank, the application will be the default application (NVDEF).

USER
User name for the connected application. The default is the same user as the current session.

PASSWORD
Password for the connected application. The default is the same password as the current session.

TIMEOUT
This parameter gives the time out value in seconds for the session.
The time out is given in seconds but in fact, NIRVA checks for timeout sessions every 30 seconds.
A time out of 0 closes the session after the command. A time out of –1 sets a time out for the life of the application (this is the default).

NAME
Session name. This parameter is optional and allows naming the session. Named session allows session group management. When executing a command from a procedure or from a service, one can tell NIRVA to do it in the context of a named session. Several sessions can have the same name.
The named sessions can be used for example to maintain a

	stack of shared database connections accessible by other sessions during the application life.
OPEN	Name of the procedure that NIRVA calls when opening this session. The default value is “session_open”. In order to not execute any open procedure, the open procedure name must be set “NV_SESSION_OPEN_NONE”. This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
CLOSE	Name of the procedure that NIRVA calls when closing this session. The default value is “session_close”. In order to not execute any close procedure, the close procedure name must be set “NV_SESSION_CLOSE_NONE”. This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
NEW_PASSWORD	*A*pplication new user password. This parameter must be provided when a logon returned an error requiring the user to change its password. The new password must be different than the old one.
NEW_PASSWORD_CONFIRM	This parameter may be provided when the *NEW_PASSWORD* parameter has been given. It allows Nirva to check if the new password is correct.
IP	Optional IP address of the session creator. If the session has to be accessed directly by a client and the security has been set to prevent client to access sessions they don’t own, you must set the session IP to the TCP/IP address of the client. Otherwise it can be let blank (default).

GET_NAMED

SESSION:GET_NAMED

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	No

Description

This command manually gets the ownership of a named session to the calling session. This ownership occurs until the calling session calls the SESSION:FREE_NAMED command.

When a session has the ownership of a named session, it can execute several commands in the context of this named session by using the NV_SESSION_NAME parameter. Nirva then guaranties that the named session is always the same.

A single session can own several named session with different names. A session must call the same number of SESSION:FREE_NAMED command than calls to the SESSION:GET_NAMED command.

The GET_NAMED command is itself executed in the context of the named session so any procedure attached to this command (NV_PROC or NV_POST_PROC parameters) is also executed in the context of the named session.

The pair GET_NAMED and FREE_NAMED must be called from the same thread. It is good practice to have both GET_NAMED and corresponding FREE_NAMED in the same procedure or service command.

Parameters

NV_SESSION_NAME Name of the named session to get.

FREE_NAMED

SESSION:FREE_NAMED

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	No

Description

This command manually frees a named session previously got with the SESSION:GET_NAMED command. A session must call the same number of SESSION:FREE_NAMED command than calls to the SESSION:GET_NAMED command.

The GET_NAMED command is itself executed in the context of the named session so any procedure attached to this command (NV_PROC or NV_POST_PROC parameters) is also executed in the context of the named session. The named session is freed after the command.

When a session exists it automatically free all the named sessions it owns.

The pair GET_NAMED and FREE_NAMED must be called from the same thread. It is good practice to have both GET_NAMED and corresponding FREE_NAMED in the same procedure or service command.

Parameters

NV_SESSION_NAME Name of the named session to free.

GET_NUM

SESSION:GET_NUM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the current number of sessions in the output buffer.

Parameters

None

GET_PASSWORD

SESSION:GET_PASSWORD

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	Yes

Description

This command returns the session password. For security reason, it’s available only from a procedure or service and subject to a permission.

Permissions

PASSWORD_GET

Parameters

None

GET_SESSION_ID

SESSION:GET_SESSION_ID

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the current session ID. There are generally some other ways to get the current session ID depending of the source of the commands.

Parameters

None

GET_USER

SESSION:GET_USER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the session user name.

Parameters

ALL_INFO — If this parameter is set to "YES", the command returns a nirva indexed string list object named USER_INFO and containing several user information.

Objects created

USER_INFO — This object is created only if the ALL_INFO parameter is set to "YES". This is a Nirva indexed string list object containing the following keys:

- "NAME" is the session user name.
- "FULLNAME" is the session user full name.

LIST

SESSION:LIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the list of NIRVA sessions currently running. The list can be limited to a specific service, to a specific application or to a single session.

The command creates a table object returning complete session information:

- Identifier.
- Connected application.
- Name.
- Application user.
- Number of connections using the session.
- Number of seconds since the last command.
- Time out value.
- Total number of commands processed.
- Current command.
- Original command.
- Number of seconds when the last command started.
- Creation time.

- Services used by the session.
- Session type.
- Command stack (debug mode only).

Permissions

SYSTEM_SESSION_INFO

Parameters

WHAT	The WHAT parameter allows to limit the session list. In order to limit the list to a given service, the WHAT parameter must be set to “SRV:*service*” where *service* is the service name. In order to limit the list to a given application, the WHAT parameter must be set to “APP:*application*” where *application* is the application name. In order to limit the list to a single session, the WHAT parameter must be set to “ID:*sessionID*” where *sessionID* is the session identifier.
WITH_CURRENT	By default, the LIST command doesn’t return the current session that generated the LIST command. In order to include it in the list, the WITH_CURRENT parameter must be set to “YES”. The default is “NO”.
TYPE	Limit the list to the given session type (optional). This can be one of the values CLIENT, NAMED, SCHEDULER, TRANSACTION, LISTENER, THREAD, INTERNAL or SYSTEM.
SORT	Allows sorting of the result list following the given column. This can be the name of one of the columns as described below.
ASCENDING	Tells if the SORT is ascending (default) or descending (value “NO”).

Objects created

SESSIONS

This is a Nirva table object containing following columns:

- “IDENTIFIER” is the session unique identifier.
- “APPLICATION” is the name of the application connected by the session.
- “NAME” is the session name (named sessions only).
- “USER” is the application user name.
- “IN_USE” is the number of connections currently using the session. This number will be generally 0 or 1.
- “LAST_USE_TIME” is the number of seconds from the end of the last command.
- “TIME_OUT” is the session time out in seconds. A value of –1 means the session has an infinite time out and will close when the corresponding application is stopped.

- “TOTAL_COMMANDS” is the number of commands already processed by the session since it has been created.
- “ACTUAL_COMMAND” is the command currently processed by the session. The ACTUAL_COMMAND is given on the form SERVICE:CLASS:COMMAND.
- “ORIGINAL_COMMAND” is the original command which has generated the current command. The original command is always a command sent from a client (browser or service or internal) while the current command may come from a procedure or a service.
- “COMMAND_TIME” is the number of seconds from the starting of the current command if there is one.
- “CREATION_TIME” is the date and time of the session creation. If the session has been created the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- “SERVICES” is the list of services used by the session.
- “TYPE” is the session type. This can be one of the values CLIENT, NAMED, SCHEDULER, TRANSACTION, LISTENER, THREAD, INTERNAL or SYSTEM.
- “STACK” is the command stack for the current session. It May contains several lines. This information is available in debug mode only.

SERVICES

SESSION:SERVICES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the list of external services currently used by the session.

Parameters

None

Objects created

SERVICES This is a Nirva string list object that contains the list of external services currently used by the session.

SET_CREATOR_IP

SESSION:SET_CREATOR_IP

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	No

Description

This command allows setting or changing the TCP/IP address of the client who has created the session. Normally if the session has been created by a client, the IP address is automatically set. The TCP/IP address of the creator is used in the security for controlling that a client user that sends a command to Nirva is the one who has created the session.

Now there may have some cases where the session has been created by a client but must be accessed by another client. At this time one must modify the creator IP address with the address of the client that should access the session.

If the application has been configured to not control the session access with TCP/IP, the creator IP address is not used.

The current session TCP/IP creator address and the address of the sender of one command are available from the session variables NV_CLIENT_CREATOR_IP and NV_CLIENT_ IP.

Parameters

IP New TCP/IP address of the session creator.

SET_DEFAULT_ERROR_PROC

SESSION:SET_DEFAULT_ERROR_PROC

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command allows defining the default error procedure. An error procedure is called when any nirva command fails.

Nirva maintains a stack of error procedures. When the stack is empty, Nirva uses the default error procedure if there is one.

Error procedures are described in chapter Error management.

Parameters

NAME	Name of the default error procedure. This parameter can be set to blank (or omitted) in order to reset the default error procedure. Please see the Calling a procedure chapter for description of the procedure syntax.

SET_DEFAULT_ERROR_XSL

SESSION:SET_DEFAULT_ERROR_XSL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command allows defining the default error xslt for the entire session.

The error xslt file is used for commands from browser in order to personalize the display of error information. By default, the error xslt is a file named “error.xsl” in the application directory. This default can be changed by using this command for setting the default error xslt for the current session. Finally the NV_XML_XSL_ERROR parameter can be used to change the default for a single command. An xslt error defined at session level will be taken in care only if the session is available at the time of the error.

Error procedures are described in chapter Error management.

Parameters

NAME	Name of the default error procedure. This parameter can be set to blank (or omitted) in order to reset the default error procedure. Please see the Calling a procedure chapter for description of the procedure syntax.

SET_LANGUAGE

SESSION:SET_LANGUAGE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command sets the default session language.

This is the language used for error descriptions. When an error occurs, an error description is given following the required language.

The default session language can also be set by giving the NV_LANGUAGE parameter when opening a new session.

The language becomes the default session language and it's not necessary to give the NV_LANGUAGE parameter for other session commands except if the user wants to use another language for his current command.

Parameters

LANGUAGE Language. The default is “ENGLISH”.

USE_ID_COOKIE

SESSION:USE_ID_COOKIE

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure	No	No	No

Description

This command allows using an http cookie for transmitting the session ID between a browser and the server. The command is available only in the session open procedure (or a procedure called by the session open procedure). A single call to this command is enough to validate the use of the session cookie for the entire session life.

The cookie is named NvSessionId. When using this cookie, the Nirva server first tries to detect the NV_SESSION_ID parameter from the command and, if not found, tries to get it from the cookie sent back by the browser.

If the session is not found or closed when ending the command, Nirva asks the browser to reset the cookie.

One can set a time out to the cookie.

Parameters

TIMEOUT	Time out in seconds for the cookie. If set to 0, the cookie will be sent to the browser without expiration date so it will be valid until the browser is closed (or manually deleted on the browser side). If set to “SESSION” the time out will be the same as the Nirva session time out. The default is 0.
HTTPONLY	When set to “YES”, adds the HttpOnly option to the session ID cookie. The default is “NO”.
SECURE	When set to “YES”, adds the Secure option to the session ID cookie. The default is “NO”.

SYSTEM class

The test class contains system level commands.

GET_ENCODING

SYSTEM:GET_ENCODING

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the nirva current encoding in the output buffer.

This can be “ISO-8859-1” or “UTF-8”.

Parameters

None

GET_NUM_PROCS

SYSTEM:GET_NUM_PROCS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the number of processors of the machine.

Parameters

None

GET_NUM_EXCEPTIONS

SYSTEM:GET_NUM_EXCEPTIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the number of exceptions occurred on the system since the process has stared. If an exception occurs, the process nvs.exe may become unstable. This command may be used in a scheduler task in order to alert an administrator when an exception occurred.

Parameters

None

GET_PLATFORM

SYSTEM:GET_PLATFORM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the operating system running Nirva in the output buffer.

The possible values are:

- “WIN32” (windows 32 bits)
- “WIN64” (windows 64 bits)
- “LINUX” (linux 32 bits)
- “LINUX64” (linux 64 bits)
- “AIX” (IBM AIX risc)
- “HPUX” (HP-UX Risc)
- “HPUXI” (HP-UX Itanium)
- “SOLARIS” (SUN Solaris)

Parameters

None

GET_TYPE

SYSTEM:GET_TYPE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the nirva version type in the output buffer.

Parameters

None

GET_VERSION

SYSTEM:GET_VERSION

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns the nirva version in the output buffer.

Parameters

None

TEST class

The test class contains commands for testing Nirva server.

CRASH

TEST:CRASH

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

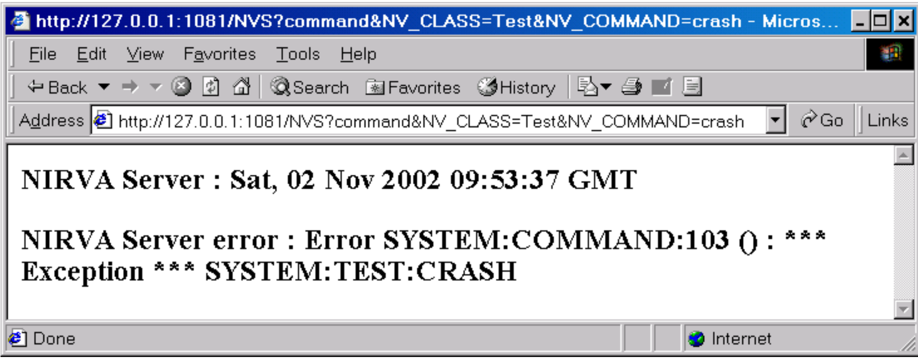
Description

This strange command produces a real crash on the server by trying to write information at a bad memory address. This command is used by development team in order to test the Nirva reaction to a real crash.

In fact, Nirva catches the crash, displays the exception message and closes the thread that produced the crash (and only this thread).

In this way, a crash inside a session has no influence for other opened sessions.

As answer to this command, Nirva server sends back an html page giving the error exception message:



Parameters

None

TEST_BROWSER

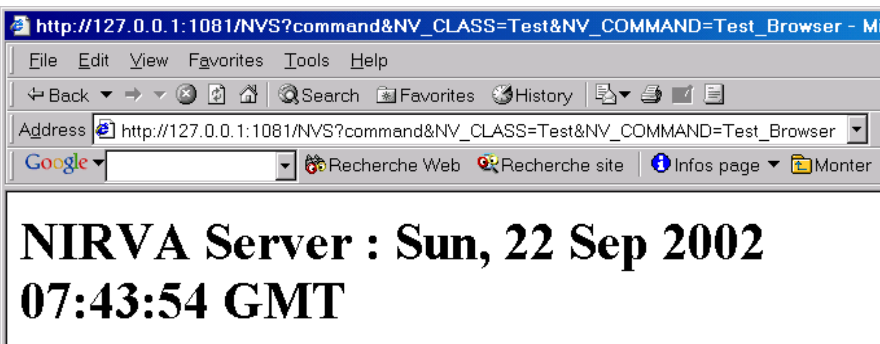
TEST:TEST_BROWSER

Source	Use Input Container	Use Output Container	Use Output buffer
Web	No	No	No

Description

This command allows testing the communication from a web browser to the Nirva server. It's available only from a web browser.

If the connection is successful, the Nirva server will send back an html page giving the GMT time:



The standard XML output management of Nirva is not available in this command because the command directly sends the HTML output to the browser.

Parameters

TEXT	This is a small text that the command will report back in the resulting HTML page. This parameter is optional.

THREAD class

The Thread class allows creating a session thread.

A session thread executes a procedure in a separate thread and then closes.

CREATE

THREAD:CREATE

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	Yes

Description

This command creates a new session thread, executes the given procedure in a separate thread and then closes the created thread and its associated session. The session thread ID is returned in the output buffer.

Once the new thread has been successfully started, the command returns to the caller.

This command is only available from procedure or service.

Parameters

PROC	Name of the procedure that will be executed by the session thread. If this parameter is not provided, the session thread will be created but it will end immediately. If the execution time of the procedure is long, it should include some calls to the SESSION:CHECK_CLOSE_REQUEST command and leave if there is a request to close. The procedure receives a parameter named “NV_CALLING_SESSION” that gives the session ID of the calling session (the one that calls the THREAD:CREATE command). This allows the session thread eventually sending data to the caller The procedure can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name.

USER	User name. This is the name of the user of the session thread. This must be a valid user. If this parameter is not provided, Nirva takes the user of the calling session.
OPEN	Name of the procedure that will be called when the session thread is opened. If the value is blank, no procedure will be called (default). If a procedure is given and the procedure returns an error, the session thread will not be created. The open procedure is called from the calling thread and not from the created thread. This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
CLOSE	Name of the procedure that will be called when the listener session is closed. If the value is blank, no procedure will be called (default). The close procedure is called from the created thread. This can be a native, java, dotnet or perl procedure. See the description of the NV_PROC parameter for the syntax of the procedure name. For a java procedure, only a class file is accepted (no jar).
PROCF	Name of a procedure that will be ran in case of failure. This parameter is not mandatory. The failed procedure can be used for error reporting. It receives the error information as following parameters: NV_ERROR_CODE, NV_ERROR_SERVICE, NV_ERROR_CLASS, NV_ERROR_DESC and NV_ERROR_INFO. If defined, the failed procedure is called when the main procedure fails.
LANGUAGE	Default language for the thread session. If not provided, this is the language of the current command.

TIME class

The TIME class provides commands for managing date and time.

GET

TIME:GET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the current time and date.

Parameters

GMT	If this parameter is set to “YES”, the command returns the GMT time instead of the local time. The default is “NO”.

Objects created

TIME This is a Nirva indexed string list object containing the following strings:

- “YEAR” is the year on 4 digits.
- “MONTH” is the month from 1 to 12.
- “DAY” is the day from 1 to 31.
- “HOUR” is the hour from 0 to 23.
- “MIN” is the minutes from 0 to 59.
- “SEC” is the seconds from 0 to 59.

TRANSACTION class

Nirva allows to group commands in a transactional context. A NIRVA transaction is simply an application procedure that is executed by a special command named SYSTEM TRANSACTION START.

When executing this command, NIRVA saves the initial session context in a persistent place and executes the transaction commands by continuously saving the current context.

At the end of the transaction commands, NIRVA sets the transaction status to “COMPLETED” or “FAILED”.

A completed transaction can then be validated by a dedicated command and then removed from the transaction list. The validation process is made by calling an external procedure defined with the transaction. The validation allows a user to validate a set of NIRVA commands.

A transaction can also be rolled back or restarted in case of failure.

The starting of transaction can be scheduled by setting its start time. An embedded transaction server periodically checks for transactions ready to run and then, runs them.

Nirva provides some visual tools for viewing and controlling the application transactions.

A transaction is created and ran by using the SYSTEM TRANSACTION START. This command creates a new transaction and returns (in the output buffer) a transaction unique identifier. The SYSTEM TRANSACTION START command requires a parameter that gives the name of the application procedure to run for this transaction.

The transaction finishes when all the commands of the procedure have been processed successfully or when there is an error in one of these commands.

When starting a transaction, NIRVA first saves the initial session context in the registry of the application and then starts to process the commands of the transaction procedure. During all the transaction, the session context is permanently saved in the application registry so when the transaction stops (successfully or with errors), both initial and current session contexts are saved in the application registry.

This feature slows the processing of the commands but is necessary to be able to eventually rollback later.

Now, it's possible to disable the saving of both initial and current contexts. This can be useful when maximum performance is required and when transactions are used only to control some NIRVA processing.

When saving a session context, NIRVA in fact saves the following things:

- Session containers.
- Session variables.
- Application user name (always saved even if the option to not save a context has been requested).
- Input and output container names.

While working with transaction, NIRVA maintains different transaction status:

- The status “NOT_STARTED” means that the transaction commands haven't been processed yet.
- The status “RUNNING” means that NIRVA is currently processing the transaction commands.
- The status “COMPLETED” means that all the transaction commands have been processed successfully.
- The status “FAILED” means that one transaction command has failed. When this occurs, NIRVA maintains the error information in the transaction and the current saved context is the one at the time of the error.
- The status “VALIDATED” means that the transaction has been validated correctly.
- The status “NOT_VALIDATED” means that the validation procedure has been called but this procedure failed.

The transaction validation is an optional step allowing the user to control the transaction. The validation process consists of calling a validation procedure that will do some specific job when the transaction has been completed. If this procedure is successful, the transaction is validated.

When a transaction has the status “COMPLETED”, “FAILED” or “NOT_VALIDATED”, it's possible to restart it by sending the SYSTEM TRANSACTION RESTART command. This command first runs an optional rollback procedure and then restarts the transaction by creating a new session and working with the saved session initial context (if there is one).

A transaction is always defined at application level but the transaction procedures can be also system or service procedures.

GET_CONTAINER

TRANSACTION:GET_CONTAINER

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command allows accessing a container (or a part of it) from the saved transaction context. It's very useful to get the result of scheduled transactions.

The requested container is copied in the output container.

Parameters

TID — Identifier of the transaction to get container from.

NAME — Name of container or subcontainer to get.
Since a container is a hierarchical structure, the access of a subcontainer is made by giving the complete subcontainer path. For example "`Container.sub1`" gets the subcontainer "`sub1`" of the container "`Container`". If this parameter is not provided, the default container is retrieved.

CONTEXT — Transaction context from which to get the container.
This parameter can take the values "INITIAL" or "CURRENT" in order to get the container from the initial or current transaction context. The default is "CURRENT".

APPEND — Append mode. If this parameter is set to "YES", the command will append the transaction container to the output container. In this case, the eventual replacement of existing objects is controlled by the parameter REPLACE.
When not in append mode, the output container is first released before the copy. This is the default.

REPLACE — This parameter has a meaning only in append mode (APPEND parameter set to "YES"). If REPLACE is set to "YES" all transaction container objects will replace output container objects having the same name.
The default is "NO" so the output container objects having the same name than the transaction container objects will not be replaced.

SUBCONTAINERS — This parameter allows to also copying the subcontainers of the transaction container to the output container.
To copy subcontainers, the parameter must be set to "YES".
The default value is "NO".

OBJECTS	This parameter allows specifying only some of the transaction container objects to copy to the output container. If used, the parameter should contain the names of valid transaction container objects, separated by semicolon character (‘;’). The default is to include all transaction container objects.
WITH_DATA	This parameter allows to also copy the object data (and not only the object description) of the transaction container to the output container. To not copy object data, the parameter must be set to “NO”. The default value is “YES”.
PERSIST	This parameter gives the new persistent value for persistent file objects that have to be copied. By default, the new file objects will not be persistent. The parameter can take value “0” for temporary files, “-1” for persistent files and any other value (in seconds) for cached files.
FILES	This parameter, if set to “YES”, tells Nirva to also copy files with file objects. If this parameter is set to “NO”, the file objects are copied but not the file they point to. When files are copied, Nirva automatically creates the new files in the application file directory if the file object is persistent and in the application work directory otherwise. This destination directory can be changed by the FILEDIR parameter if the command has been sent from a procedure or from an external service. The default value is “YES”.
FILEDIR	This parameter has meaning only if the FILES parameter has been set to “YES”. It allows changing the destination directory for files of file objects. If the FILEDIR is set to the application work directory and the PERSIST parameter is set to “-1” (persistent), the command will return an error because the application work directory should not contain any persistent file. The FILEDIR parameter can be used only if the command source is procedure or service. Otherwise, it’s ignored.

GET_VARIABLES

TRANSACTION:GET_VARIABLES

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command allows accessing the list of session variables from the saved transaction context. It's very useful to get the result of scheduled transactions.

The requested variables are copied in the output container in a NIRVA indexed string list object named "VARIABLES".

Parameters

TID	Identifier of the transaction to get variables from.
CONTEXT	Transaction context from which to get the variables. This parameter can take the values "INITIAL" or "CURRENT" in order to get the variables from the initial or current transaction context. The default is "CURRENT".

Objects created

VARIABLES	This is a Nirva indexed string list object containing the requested session variables.

INFO

TRANSACTION:INFO

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns the current application transaction list (or a part of it). For each transaction, the command returns the following information:

- Unique identifier.
- Name.
- Description.
- Status
- Error information
- Creation time
- Last start time

- Transaction procedure

The INFO command returns information in a NIRVA table object named “INFO” in the output container.

It’s possible to reduce the list by searching for some specific transaction names or creation dates.

Parameters

NAME
Transaction name. This parameter, when provided, is used to restrict the list to specific transaction names. The wildcard character '*' can be used for searching transactions names starting with a given value. If NAME is “*”, NIRVA doesn’t restrict the list based on the transaction name. This is the default.

DATE
Transaction creation date. If this parameter is given alone, it’s used to restrict the list to specific transaction creation dates. The date must be given on the format DD/MM/YYYY. The space, point and backslash separators are also accepted.
When this parameter is given with the TO_DATE parameter, it’s used as a starting date to restrict the list of transaction to a specific date range (from DATE to TO_DATE).
The default is blank (no date restriction).

TO_DATE
This parameter has meaning only when the DATE parameter is a valid date. It’s used as an ending date to restrict the list of transaction to a specific date range (from DATE to TO_DATE).
The default is blank (no end date).

Objects created

INFO
This is a Nirva table object containing the following columns:

- “IDENT” is the unique transaction identifier.
- “NAME” is the transaction name (can be empty).
- “DESCRIPTION” is the transaction description.
- “STATUS” is the current transaction status. It can be “NOT_STARTED”, “RUNNING”, “FAILED”, “COMPLETED”, “VALIDATED” or “NOT_VALIDATED”.
- “ERROR” gives transaction error information if there is one available.
- “CTIME” is the date and time of the transaction creation. If the transaction has been created the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- “STIME” is the date and time of the transaction last start. If the transaction has been started (or restarted) the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- “PROCT” is the transaction procedure name.

INFOEX

TRANSACTION:INFOEX

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command returns complete information about a specific transaction.

The INFOEX command returns information in a NIRVA indexed string list object named "TRANSACTION_INFO" in the output container.

Parameters

TID Identifier of the transaction to get information from.

Objects created

TRANSACTION_INFO This is a Nirva indexed string list object containing the complete transaction information. Here the available strings:

- "IDENT" is the unique transaction identifier.
- "NAME" is the transaction name (can be empty).
- "DESCRIPTION" is the transaction description.
- "STATUS" is the current transaction status. It can be "NOT_STARTED", "RUNNING", "FAILED", "COMPLETED", "VALIDATED" or "NOT_VALIDATED".
- "ERROR" gives transaction error information if there is one available.
- "CTIME" is the date and time of the transaction creation. If the transaction has been created the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- "STIME" is the date and time of the transaction last start. If the transaction has been started (or restarted) the same day, only the time is returned. The returned format is YYYY/MM/DD HH:MM:SS.
- "PROCT" is the transaction procedure name.
- "PROCV" is the validation procedure name (executed to validate a transaction).
- "PROCRC" is the rollback completed procedure name (executed to rollback a successfully completed transaction).

- “PROCRF” is the rollback failed procedure name (executed to rollback a failed transaction).
- “PROCF” is the failed procedure name (executed when the transaction has failed).
- “PROCC” is the completed procedure name (executed when the transaction has completed successfully).
- “PROCVOK” is the validated OK procedure name (executed when validation is successful).
- “PROCVKO” is the validated KO procedure name (executed when validation failed).

REMOVE

TRANSACTION:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes the given transaction.

The removing is possible only if the transaction is not in use.

Parameters

TID	Identifier of the transaction to remove.
WAIT	Number of seconds to wait for the transaction to be free if it’s in use. The default is 10 seconds.

RESTART

TRANSACTION:RESTART

Source	Use Input Container	Use Output Container	Use Output buffer

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command restarts a given transaction.

It’s not possible to restart a transaction having the status “RUNNING” or “VALIDATED”.

The restart is made by creating a new session specific to the transaction. This session is automatically closed after the end of the restart. The new session starts with one of the transaction saved context: initial or current context.

If the status of the TRANSACTION is “NOT_STARTED”, the restart directly occurs with the initial saved context.

If the status of the TRANSACTION is “FAILED”, the restart first restores the current context, runs the rollback procedure for failed transaction (if there is one defined) and restarts the transaction procedure after having restored the initial context.

If the status of the TRANSACTION is “COMPLETED” or “NOT_VALIDATED”, the restart first restores the current context, runs the rollback procedure for completed transaction (if there is one defined) and restarts the transaction procedure after having restored the initial context.

The restart always occurs with the user name used when the transaction has been created.

The restarting can be scheduled. This concerns only the restarting itself. In any case, the command executes the rollback if requested and restores the context.

This command fails if the transaction is in use.

Parameters

TID Identifier of the transaction to restart.

NO_REMOVE If this parameter is set to “YES”, NIRVA will not remove the transaction (after successful restart and validation) even if the remove mode has been set to “AUTO”.

AUTO_ROLLBACK Rollback if error.
If this parameter is set to “YES”, the transaction will automatically rollback in case of error (by executing the failed rollback procedure if there is one). At this time, the final status will be “NOT_STARTED”. The RESTART command will still return an error.
The default is “NO”.

START_MODE Start mode.
If this parameter is set to “AUTO”, NIRVA restarts the transaction immediately.
If this parameter is set to “MANUAL”, the user will need to send an explicit

RESTART command with a START_MODE set to “AUTO” in order to restart the transaction.
If this parameter is set to “SCHEDULED”, NIRVA returns immediately and the control of the restarting of the transaction is given to the embedded transaction server.

START_DATE

Start date. This parameter has meaning only when the START_MODE parameter has been set to “SCHEDULED”. It gives the transaction starting date. The general format of the START_DATE parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only DD/MM is given, NIRVA is using the current year.
If only DD is given, NIRVA is using the current month and year.
NIRVA can also accept the year on 2 digits.
If the START_DATE is preceded by a plus sign (‘+‘) followed by an integer. NIRVA considers that as a number of days after the current day. For example, “+3” is 3 days after the current day.
The default value for the START_DATE field is the current date.

START_TIME

Start time. This parameter has meaning only when the START_MODE parameter has been set to “SCHEDULED”. It gives the transaction starting time. The general format of the START_TIME parameter is HH:MM:SS where HH is the hour from 0 to 23, MM is the minute from 0 to 59 and SS is the second from 0 to 59.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only HHMM is given, NIRVA is using 0 for the seconds.
If only HH is given, NIRVA is using 0 for the seconds and for the minutes.
If the TIME is preceded by a plus sign (‘+‘) followed by an integer. NIRVA considers that as a number of hours after the current time. For example, “+3” is 3 hours after the current time. If the integer is followed by an “m” character, NIRVA considers that as a number of minutes after the current time. For example, “+10m” is 10 minutes after the current time.
The default value for the START_TIME field is the current time.

ROLLBACK

TRANSACTION:ROLLBACK

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command rollback a given transaction if there are some rollback procedures defined.

It’s not possible to rollback a transaction having the status “RUNNING” or “VALIDATED”.

After a successful rollback, the transaction goes to the status “NOT_STATED”.

The rollback is made by creating a new session specific to the transaction. This session is automatically closed after the end of the rollback. The new session starts with the transaction current saved context.

If the status of the TRANSACTION is “NOT_STARTED”, the rollback commands do nothing.

If the status of the TRANSACTION is “FAILED”, the restart first restores the current context and then runs the rollback procedure for failed transaction (if there is one defined)

If the status of the TRANSACTION is “COMPLETED” or “NOT_VALIDATED”, the restart first restores the current context and then runs the rollback procedure for completed transaction (if there is one defined).

The rollback always occurs with the user name used when the transaction has been created.

This command fails if the transaction is in use.

Parameters

TID	Identifier of the transaction to rollback.

SET_OPTIONS

TRANSACTION:SET_OPTIONS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command allows changing some of the transaction options.

It will fail if the given transaction is in use except if the transaction is the current one. This allows defining the transaction options directly inside the transaction procedure instead of defining them at creation time. For example, the rollback procedure can be changed at different steps of the transaction.

If some of the SET_OPTION parameters are not provided, their corresponding transaction option will not be changed.

Parameters

TID	Identifier of the transaction to set options.
NAME	Transaction name. This is a friendly name that can be used to recognize the transaction but it does not identify it. Only the transaction identifier uniquely identifies a transaction.
DESCRIPTION	Transaction description.
PROC_VALI	Name of the validation procedure. NIRVA runs the validation procedure when the transaction status is “COMPLETED” in order to try to validate the transaction. If this parameter is blank, the validation process will just change the status of the transaction from “COMPLETED” to “VALIDATED”. The procedure name can correspond to an application, system or service procedure.
PROC_ROLLC	Name of the callback procedure for a completed transaction. NIRVA runs this procedure when the transaction status is “COMPLETED” or “NOT_VALIDATED” and the user has sent a RESTART order. If this parameter is blank, no callback occurs. The procedure name can correspond to an application, system or service procedure.
PROC_ROLLF	Name of the callback procedure for a failed transaction. NIRVA runs this procedure when the transaction status is “FAILED” and the user has sent a RESTART order. If this parameter is blank, no callback occurs. The procedure name can correspond to an application, system or service procedure.
PROC_FAIL	Name of the procedure for a failed transaction. NIRVA runs this procedure when a transaction has failed. This can be useful to make reporting of the transaction processing. The procedure name can correspond to an application, system or service procedure.
PROC_COMP	Name of the procedure for a completed transaction. NIRVA runs this procedure when a transaction has been completed successfully. This can be useful to make reporting of the transaction processing.

	The procedure name can correspond to an application, system or service procedure.
PROC_VOK	Name of the procedure for a validated transaction. NIRVA runs this procedure when a transaction has been validated successfully. This can be useful to make reporting of the transaction processing. The procedure name can correspond to an application, system or service procedure.
PROC_VKO	Name of the procedure for a completed transaction. NIRVA runs this procedure when a transaction validation has failed. This can be useful to make reporting of the transaction processing. The procedure name can correspond to an application, system or service procedure.
VALIDATION_MODE	Validation mode. If this parameter is set to “AUTO”, NIRVA will start the validation immediately after a successful completion of the transaction. If this parameter is set to “MANUAL”, the user will need to send an explicit VALIDATE command in order to validate the transaction.
REMOVE_MODE	Remove mode. If this parameter is set to “AUTO”, NIRVA will automatically remove the transaction from the list when it has been validated successfully. If this parameter is set to “MANUAL”, the user will need to send an explicit REMOVE command in order to remove the transaction.

SET_STATUS

TRANSACTION:SET_STATUS

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command allows forcing the status of a transaction. It does no processing except changing the status so it must be used carefully.

It’s not possible to change the status to “RUNNING”.

This command fails if the transaction is in use.

Parameters

TID Identifier of the transaction to change status.

STATUS New transaction status. This can be “NOT_STARTED”, “COMPLETED”, “FAILED”, “VALIDATED” or “NOT_VALIDATED”.

START

TRANSACTION:START

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command creates a new transaction and runs it if requested. The command requires a parameter named "PROC_TRAN" that gives the name of application procedure to run. This application procedure contains the commands of the transaction.

When starting a transaction, NIRVA first saves the initial session context in the registry of the application and then starts to process the commands of the transaction procedure. During all the transaction, the session context is permanently saved in the application registry so when the transaction stops (successfully or with errors), both initial and current session contexts are saved in the application registry.

The starting of transaction can be scheduled by setting its start time. An embedded transaction server periodically checks for transactions ready to run and then, runs them.

The START command will fail if called from another transaction.

The command returns the transaction unique identifier in the output buffer.

Parameters

PROC_TRANS Name of the transaction procedure. This parameter is mandatory. The transaction procedure contains the transaction commands.
Please see the Calling a procedure chapter for description of the procedure syntax.

NAME Transaction name. This is a friendly name that can be used to recognize the transaction but it does not identify it. Only the transaction identifier uniquely identifies a transaction.

DESCRIPTION Transaction description.

START_MODE Start mode.
If this parameter is set to “AUTO”, NIRVA starts the transaction immediately.

This is the default.
If this parameter is set to “MANUAL”, the user will need to send an explicit RESTART command in order to start the transaction.
If this parameter is set to “SCHEDULED”, NIRVA returns immediately and the control of the starting of the transaction is given to the embedded transaction server.

START_DATE

Start date. This parameter has meaning only when the START_MODE parameter has been set to “SCHEDULED”. It gives the transaction starting date. The general format of the START_DATE parameter is DD/MM/YYYY where DD is the day from 1 to 31, MM is the month from 1 to 12 and YYYY is the complete year.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only DD/MM is given, NIRVA is using the current year.
If only DD is given, NIRVA is using the current month and year.
NIRVA can also accept the year on 2 digits.
If the START_DATE is preceded by a plus sign (‘+‘) followed by an integer. NIRVA considers that as a number of days after the current day. For example, “+3” is 3 days after the current day.
The default value for the START_DATE field is the current date.

START_TIME

Start time. This parameter has meaning only when the START_MODE parameter has been set to “SCHEDULED”. It gives the transaction starting time. The general format of the START_TIME parameter is HH:MM:SS where HH is the hour from 0 to 23, MM is the minute from 0 to 59 and SS is the second from 0 to 59.
Now, if the input format is different than this one, NIRVA may accept it by trying to convert it to the required format.
The separator can be ‘/’, ’\’, ’.’, ‘ ‘(space) or ‘:’. It’s also possible to not give the separator.
If only HHMM is given, NIRVA is using 0 for the seconds.
If only HH is given, NIRVA is using 0 for the seconds and for the minutes.
If the START_TIME is preceded by a plus sign (‘+‘) followed by an integer. NIRVA considers that as a number of hours after the current time. For example, “+3” is 3 hours after the current time. If the integer is followed by an “m” character, NIRVA considers that as a number of minutes after the current time. For example, “+10m” is 10 minutes after the current time.
The default value for the START_TIME field is the current time.

PROC_VALID

Name of the validation procedure.
NIRVA runs the validation procedure when the transaction status is “COMPLETED” in order to try to validate the transaction. If this parameter is not provided or is blank, the validation process will just change the status of the transaction from “COMPLETED” to “VALIDATED”.

	The procedure name can correspond to an application, system or service procedure. It can be a Perl script.
PROC_ROLLC	Name of the callback procedure for a completed transaction. NIRVA runs this procedure when the transaction status is “COMPLETED” or “NOT_VALIDATED” and the user has sent a RESTART order. If this parameter is not provided or is blank, no callback occurs. The procedure name can correspond to an application, system or service procedure. It can be a Perl script.
PROC_ROLLF	Name of the callback procedure for a failed transaction. NIRVA runs this procedure when the transaction status is “FAILED” and the user has sent a RESTART order. If this parameter is not provided or is blank, no callback occurs. The procedure name can correspond to an application, system or service procedure. It can be a Perl script.
PROC_FAIL	Name of the procedure for a failed transaction. NIRVA runs this procedure when a transaction has failed. This can be useful to make reporting of the transaction processing. The procedure name can correspond to an application, system or service procedure. It can be a Perl script.
PROC_COMP	Name of the procedure for a completed transaction. NIRVA runs this procedure when a transaction has been completed successfully. This can be useful to make reporting of the transaction processing. The procedure name can correspond to an application, system or service procedure. It can be a Perl script.
PROC_VOK	Name of the procedure for a validated transaction. NIRVA runs this procedure when a transaction has been validated successfully. This can be useful to make reporting of the transaction processing. The procedure name can correspond to an application, system or service procedure. It can be a Perl script.
PROC_VKO	Name of the procedure for a completed transaction. NIRVA runs this procedure when a transaction validation has failed. This can be useful to make reporting of the transaction processing. The procedure name can correspond to an application, system or service procedure. It can be a Perl script.
VALIDATION_MODE	Validation mode. If this parameter is set to “AUTO”, NIRVA will start the validation immediately after a successful completion of the transaction. This is the default. If this parameter is set to “MANUAL”, the user will need to send an explicit VALIDATE command in order to validate the transaction.
REMOVE_MODE	Remove mode. If this parameter is set to “AUTO”, NIRVA will automatically remove the

	transaction from the list when it has been validated successfully. This is the default. If this parameter is set to “MANUAL”, the user will need to send an explicit REMOVE command in order to remove the transaction.
SAVE_INITIAL	Save initial context mode. Normally, the transaction saves the initial context (containers and variables) in a persistent way. If this parameter is set to “NO”, NIRVA doesn’t save the initial context. If the initial context is not saved, a restart of the transaction will operate in a completely blank new context. The default is “YES”.
SAVE_CURRENT	Save current context mode. Normally, the transaction saves the current context (containers and variables) in a persistent way while running the transaction. This allows working on this current context for RESTART or VALIDATING commands. If this parameter is set to “NO”, NIRVA doesn’t save the current context. This disabling can be useful if a transaction is just used for a control operation. When the context is not saved, the transaction is faster. The default is “YES”.
AUTO_ROLLBACK	Rollback if error. If this parameter is set to “YES”, the transaction will automatically rollback in case of error (by executing the failed rollback procedure if there is one). At this time, the final status will be “NOT_STARTED”. The START command will still return an error. The default is “NO”.

VALIDATE

TRANSACTION:VALIDATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command validates a given transaction.

The command fails if the transaction status is “RUNNING”, “NOT_STARTED” or “FAILED”.

If the transaction is already validated (status = “VALIDATED”), the function returns successfully without doing anything.

The validation is made by creating a new session specific to the transaction. This session is automatically closed after the end of the restart. The new session starts with the current saved context.

It runs the validation procedure and changes the transaction status to “VALIDATED” or “NOT_VALIDATED” following the result of the validation procedure.

The validation always occurs with the user name used when the transaction has been created.

This command fails if the transaction is in use.

Parameters

TID	Identifier of the transaction to validate.
NO_REMOVE	If this parameter is set to “YES”, NIRVA will not remove the transaction (after successful validation) even if the remove mode has been set to “AUTO”.

VARIABLE class

The VARIABLE class provides commands for managing session variables.

ENCODE

VARIABLE:ENCODE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command changes the encoding of the requested variable.

Parameters

NAME	Session variable name. A variable name is case insensitive and can contain space characters. If the variable name is preceded with the string “[NV_PARENT]:”, the variable refers to the parent session. This can be used for accessing parent session variables from a named session. If there is no parent session, the string “[NV_PARENT]:” is ignored if it exists.
SRC_ENCODING	Current encoding of the variable. This can be “UTF-8”, “ISO-8859-1” or “INTERNAL”. If “INTERNAL” is used, then the variable is considered to have the same encoding than the internal NIRVA encoding (“UTF-8” or “ISO-8859-1”) . The default value is “INTERNAL”.

DEST_ENCODING Required new encoding of the variable. This can be “UTF-8”, “ISO-8859-1” or “INTERNAL”. If “INTERNAL” is used, then the variable will be converted to the same encoding than the internal NIRVA encoding (“UTF-8” or “ISO-8859-1”). If both SRC_ENCODING and DEST_ENCODING are the same, then the command does nothing. The default value is “INTERNAL”.

EXIST

VARIABLE:EXIST

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command returns “YES” in the output buffer if the given variable exists and “NO” otherwise.

Parameters

NAME Session variable name. A variable name is case insensitive and can contain space characters. If the variable name is preceded with the string “[NV_PARENT]:”, the variable refers to the parent session. This can be used for accessing parent session variables from a named session. If there is no parent session, the string “[NV_PARENT]:” is ignored if it exists.

GET

VARIABLE:GET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	Yes

Description

This command gets the value of the given session variable. If the variable doesn’t exist, the command returns an empty string. The returned variable value is written to another variable named “RESULT” by

default (but that can be changed). If the command is sent from a service (inter service communication), the variable value is returned in a buffer given as parameter of the C function that calls the command.

In fact, the GET command of the VARIABLE class is not very useful because the variable access is done by giving the variable name preceded by a '#' character as a command parameter value. For example, if a command includes the parameter "PARAM="#myvar"", Nirva will get the value of the session variable "myvar" as value of the parameter "PARAM".

Parameters

NAME — Session variable name. A variable name is case insensitive and can contain space characters. If the variable name is preceded with the string "[NV_PARENT]:", the variable refers to the parent session. This can be used for accessing parent session variables from a named session. If there is no parent session, the string "[NV_PARENT]:" is ignored if it exists.

GET_MULTI

VARIABLE:GET_MULTI

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command exports some of the session variables into an indexed string list object.

Parameters

NAME — String telling which variables to get. All the variables starting with the given value will be retrieved. If NAME is blank or not given, Nirva exports all session variables. A variable name is case insensitive and can contain space characters. If the variable name is preceded with the string "[NV_PARENT]:", the variable refers to the parent session. This can be used for accessing parent session variables from a named session. If there is no parent session, the string "[NV_PARENT]:" is ignored if it exists.

VARIABLES — Name of the indstringlist object that the command returns. The default is "VARIABLES".

REMOVE

VARIABLE:REMOVE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command removes the given session variable or all the session variables.

The session variables are removed automatically when a new command comes from a NIRVA client or a browser. When this occurs, NIRVA creates a new set of standard variables including the command parameters and then process the command. A standard parameter named NV_KEEP_VAR allows to not removing the session variables for a new command.

Parameters

NAME Session variable name. A variable name is case insensitive and can contain space characters. If the variable name is preceded with the string "[NV_PARENT]:", the variable refers to the parent session. This can be used for accessing parent session variables from a named session. If there is no parent session, the string "[NV_PARENT]:" is ignored if it exists.
If this parameter is blank or not provided, all the session variables are removed.

SET

VARIABLE:SET

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	No	No

Description

This command sets the value of the given session variable. If the session variable doesn't exist, the command creates it.

Parameters

NAME Session variable name. A variable name is case insensitive and can contain space characters. If the variable name is preceded with the string “[NV_PARENT]:”, the variable refers to the parent session. This can be used for accessing parent session variables from a named session. If there is no parent session, the string “[NV_PARENT]:” is ignored if it exists.

VALUE Session variable value. The value can be any string.

SRC_ENCODING Current encoding of the variable value. This can be “UTF-8”, “ISO-8859-1” or “INTERNAL”. If “INTERNAL” is used, then the variable is considered to have the same encoding than the internal NIRVA encoding (“UTF-8” or “ISO-8859-1”) . The default value is “INTERNAL”.

DEST_ENCODING Required new encoding of the variable. This can be “UTF-8”, “ISO-8859-1” or “INTERNAL”. If “INTERNAL” is used, then the variable will be converted to the same encoding than the internal NIRVA encoding (“UTF-8” or “ISO-8859-1”). If both SRC_ENCODING and DEST_ENCODING are the same, then the command does nothing. The default value is “INTERNAL”.

WEBSERVICE class

The WEBSERVICE class provides commands for working with web services. In fact only one command is really useful for working with web services: the EXECUTE command.

The EXUCUTE command is automatically launched when accessing NIRVA by the way of its web service connector.

There are other web service commands dedicated to web service management but they are not documented here.

EXECUTE

WEBSERVICE:EXECUTE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command executes an operation of a web service. The input data must be in the input container and the output data is delivered in the output container following the structure of input and output messages for the operation.

If some part of the input data structure doesn't exist, the command automatically creates it but with empty value. For example, if the web service operation input message defines a string object named “mystring” as input, the command will create the object “mystring” if it doesn't exist but with no value inside).

In the same way if some part of the input data structure doesn't exist after executing the web service procedure, the command automatically creates it.

In this way, the command always warranty that input and output data correspond with the respective input and output messages defined in the web service operation requested.

Permissions

There is one permission associated to each operation of each web service. The user must have the permission enabled in order to run the corresponding web service operation.

Parameters

WEBSERVICE	Name of the web service. This parameter is mandatory.
OPERATION	Name of the web service operation to execute. This parameter is mandatory.

Objects created

The command creates all the objects and subcontainers defined in the output message of the web service operation.

XML class

The XML class provides commands for working with XML data.

The XML class commands are independent from the standard Nirva XML output management that allows for each web command to produce XML or HTML output directly onto the web browser.

GET_XML

XML:GET_XML

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command populates the output container with the XML content of a file or binary object. The XML data must respect the NIRVA XML grammar described in the XML chapter.

In order to adapt any kind of XML data to the NIRVA XML grammar, one can use the internal XSLT processor by the way of the XML TRANSFORM command.

Parameters

XMLOBJ

This is the name of the source object of the input container that contains the XML data. This can be a file or binary object.
If this parameter is not provided, Nirva uses "XML" as object name.

LOAD_XSL

XML:LOAD_XSL

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	No	Yes	No

Description

This command creates a NIRVA file object that contains a NIRVA XSL file defined at application, system or service level.

Parameters

XSL

This is the name of NIRVA file object that will contain the eventually found XSL file. If the object already exists and is not of file type, NIRVA first removes it. If the object doesn't exist, NIRVA creates it. The default value is "XSL".

XSL_NAME

This parameter must correspond to an existing server xslt file.
An xslt file can be at application, system or service level.
For an application xslt file, the xslt file name is given as it is. For example: "`file1.xsl`". This must correspond to an existing file of the application file directory. If there is no extension given, Nirva adds "`.xsl`" after the xslt name. The xslt file name is case insensitive even under UNIX.
For a system xslt file, the xslt file name must be enclosed in brackets and preceded by the service name. For example: "`(file1.xsl)`". This must correspond to an existing file of the system file directory. If there is no extension given, Nirva adds "`.xsl`" after the xslt name. The xslt file name is case insensitive even under UNIX.
For a service xslt file, the xslt file name must be enclosed in brackets. For example: "`MyService(file1.xsl)`". This must correspond to an existing file of the service file directory. If there is no extension given, Nirva adds

"`.xsl`" after the xslt name. The xslt file name is case insensitive even under UNIX.

SEND

XML:SEND

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This powerful command is an HTTP client for sending XML messages to any XML over HTTP server. It allows communicating with any application able to receive XML messages.

NIRVA maintains a pool of TCP/IP connections in order to reuse connections. Any un-used connection is closed after approximately 120 seconds.

Parameters

XMLIN — This is the name of NIRVA file or binary object containing the XML file or data to send. This object must be in the input container. The default name is "XMLIN".

XMLOUT — This is the name of NIRVA file or binary object that will contain the resulting XML data from the server. This object will be created in the output container. The default name is "XMLOUT". If the object already exists and is not of file or binary type, NIRVA first removes it. If the object doesn't exist, NIRVA creates it.

OUTPUT — Tells to create a file or binary result. If output is "FILE", the command writes the result in a file object. If output is "BINARY", the command writes the result in a binary object. The name of the result object is given by the XMLOUT parameter.
The default is "FILE".

URL — Url for the server that processes the XML data.

HTTP_USER — User name if the connected server requires authentication.

HTTP_PASSWORD — Password if the connected server requires authentication.

SSL — If the URL parameter starts with" https://" string, NIRVA automatically considers using the HTTPS protocol. Otherwise, one can force the HTTPS protocol by setting the SSL parameter to "YES".

PORT	The TCP/IP port can be given directly in the URL. If not found in the URL, one can set it by the way of the PORT parameter. If the port is not found, NIRVA assumes port 80 for HTTP and port 443 for HTTPS.
PROXY	Optional address of a proxy server to be used to connect the final target server. The format of this parameter is the following: `protocol://proxyserver:proxyport` where *protocol* is the protocol for the proxy server. It can be “http” or “https”; *proxiserver* is the address of the proxy server and *proxyport* is the TCP/IP port.
PROXY_USER	User name for proxy connection if the proxy requires authentication.
PROXY_PASSWORD	Password for proxy connection if the proxy requires authentication.
CTIMEOUT	Connection time out. This is the maximum time in seconds for establishing the connection to the server. The default is 10 seconds.
CLOSE	Normally, after sending the command, the TCP/IP connection is not closed in order for the connection to be reused if necessary. One can force the connection to close immediately after the command by setting the CLOSE parameter to “YES”. In any case the connection will be automatically closed by NIRVA after approximately 120 seconds if not used.
SOCKET_LOG	Complete path name of a log file where Nirva will write all read/writes on the used socket.
METHOD	HTTP method used to send the request. This can be “GET” or “POST”. Default is “POST”.
FULL	If set to “YES”, the full url is sent to the server after the HTTP method. Otherwise only the requested path is sent. The default is “NO”. If a proxy is to be used, the full path is always sent.
SOAP_ACTION	If given, the value is sent as SOAPAction http header for sending a SOAP 1.1 message to a web service provider.
CONTENT_TYPE	Content type of the sent object. The default is “text/xml”.
REUSE	Reuse connection. Nirva maintains a stack of open socket connections to peer servers and normally reuse the connections when several commands address the same server. This parameter can be set to “NO” in order to force the creation of a new connection for the XML:SEND command. The default is “YES”. If a client certificate is being used, it’s advised to set the REUSE parameter to “NO”.
CERTIFICATE	Name of a Nirva file object that points to a client certificate file if the server requires it. If given, the certificate must contain also contain the private key. Nirva accepts only PEM format certificates. You can use openssl tools in order to convert your certificate if it’s not in PEM format. If a client certificate is being used, it’s advised to set the REUSE parameter to “NO”.
CERTIFICATE_PASSWORD	Optional certificate password if your client certificate requires one.

CACERT	Name of a Nirva file object that points to a root CA certificate file if the server requires it. Nirva accepts only PEM format certificates. You can use openssl tools in order to convert your certificate if it's not in PEM format.

SET_XML

XML:SET_XML

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command produces a file or binary object in the output container that is an XML view of the input container.

It can be used instead of the standard Nirva XML mechanism for users who want to get the XML result into a file or binary object and without displaying it.

The command uses the input container given by default or in the general parameters NV_CONTAINER or NV_IN_CONTAINER for producing XML data file.

The XML data tags are described in the XML section of this documentation.

Parameters

XMLOBJ	This is the name of the resulting file or binary object that the command will put in the output container. If this parameter is not provided, Nirva uses "XML" as object name. If the object doesn't exist, Nirva creates it as a temporary file object if a file object is requested and as a binary object otherwise. If the requested object is file and the object exists but is not a file object, Nirva removes it before creating it. If the requested object is file and the object exists and is a file object, Nirva directly uses the object.
OUTPUT	Tells to create a file or binary result. If output is "FILE", the command writes the result in a file object. If output is "BINARY", the command writes the result in a binary object. The name of the result object is given by the XMLOBJ parameter. The default is "FILE".
OBJECTS	This parameter allows specifying only some of the input container objects to put to the resulting XML data.

If used, the parameter should contain the names of valid container objects, separated by semicolon character (';').
The default is to include all input container objects.

SUBCONTAINERS
This parameter allows to also sending the subcontainers of the input container to the resulting XML data.
To send subcontainers, the parameter must be set to "YES".
The default value is "NO".

WITH_DATA
This parameter allows to also send the object data (and not only the object description) of the specified container to the resulting XML data.
It can control also if the NIRVA file and binary data must be sent.
To not send object data, the parameter must be set to "NO". To send object data but without binary and file data, the parameter must be set to "YES". To send all data including binary and file data, the parameter must be set to "ALL". When NIRVA sends binary or file data, this data is base64 encoded.
The default is "YES".

HEADERS
This parameter allows to also send the HTTP headers received in the input HTTP order to the resulting XML data.
The possible values are "YES" or "NO". The default is "NO"

VARIABLES
This parameter allows to also sending the session variables to the resulting XML data.
The possible values are "YES" or "NO". The default is "NO"

NSREF
Name space URL. This is the URL of the namespace to use in the resulting XML data. If this parameter is blank, no namespace is used (default).

NSPREFIX
This parameter has meaning only if the NSREF parameter is not blank. It defines the prefix of the namespace. If the prefix is empty, the given namespace URL will be the default namespace. If the prefix is given, all the NIRVA generated XML tags will be namespace qualified.

ENCODING
Name of the character set to use in the generated XML data. The default is "ISO-8859-1" or "UTF-8" if NIRVA has been launched with the Unicode option. This parameter tells nirva what XML model to use for output: normal or simple model. The normal and simple models are described in the XML chapter in this documentation.

SIMPLE
This parameter tells nirva what XML model to use for output: normal or simple model. The normal and simple models are described in the XML chapter in this documentation. The default is normal model. This parameter is deprecated but stays for compatibility. Please use the MODEL parameter instead.

STRICT
When transforming container content to XML, Nirva encodes special characters <, >, ', " and & to respectively "<", ">", "'", """ and "&". In strict mode (when parameter value is "YES"), if the container data contains a string "<", this one will be transformed into "<". If strict mode is not set, the "<" string is unchanged. So if the strict mode is not

set, the strings “<”, “>”, “'”, “"” and “&” in the container data are unchanged. The default value is “NO”.

MODEL

This parameter tells nirva what XML model to use for output: NORMAL, SIMPLE, COMPACT or TINY. These models are described in the XML chapter in this documentation. The default is normal model.
The MODEL parameter must be used instead of the SIMPLE parameter. If both are used, MODEL has priority.

SET_XSL

XML:SET_XSL

Source	Use Input Container	Use Output Container	Use Output buffer
Procedure Service	No	No	No

Description

This command allows changing the name of the XSL file given in the original command by the NV_XML_XSL command parameter.

This feature allows procedures or services to target the command to a specific user interface that they deliver. In this way, it’s possible for NIRVA applications or services to deliver commands with a web user interface.

This command is available only from a service or a procedure and when the original command is a web command.

Parameters

XSL_NAME

This parameter must correspond to an existing server xslt file.
An xslt file can be at application, system or service level.
For an application xslt file, the xslt file name is given as it is. For example: “`file1.xsl`”. This must correspond to an existing file of the application file directory. If there is no extension given, Nirva adds “`.xsl`” after the xslt name. The xslt file name is case insensitive even under UNIX.
For a system xslt file, the xslt file name must be enclosed in brackets and preceded by the service name. For example: “`(file1.xsl)`”. This must correspond to an existing file of the system file directory. If there is no extension given, Nirva adds “`.xsl`” after the xslt name. The xslt file name is case insensitive even under UNIX.
For a service xslt file, the xslt file name must be enclosed in brackets. For example: “`MyService(file1.xsl)`”. This must correspond to an existing file of the service file directory. If there is no extension given, Nirva adds

"`.xsl`" after the xslt name. The xslt file name is case insensitive even under UNIX.

TRANSFORM

XML:TRANSFORM

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	Yes	No

Description

This command uses the NIRVA defined XSLT engine in order to transform XML file or data into another XML file or data using a given XSL file.

This command can be use with any kind of XML data and not only with XML data generated by NIRVA.

When using the internal xslt engine, all the parameters of the TRANSFORM command are sent to the XSLT engine, they can be accessed from inside the xslt style sheet in the following way (assuming TEST is a parameter of the command XML:TRANSFORM):

```
<xsl:param name="TEST" />
<xsl:value-of-select="$TEST" />
```

Parameters

XMLSRC — This is the name of NIRVA file or binary object containing the XML file or data to transform. This object must be in the input container. The default name is "XML".

XMLDEST — This is the name of NIRVA file or binary object that will contain the transformed XML data. This object will be created in the output container. The default name is "XMLDEST". If the object already exists and is not of file or binary type, NIRVA first removes it. If the object doesn't exist, NIRVA creates it.

OUTPUT — Tells to create a file or binary result. If output is "FILE", the command writes the result in a file object. If output is "BINARY", the command writes the result in a binary object. The name of the result object is given by the XMLDEST parameter.
The default is "FILE".

XSL	This is the name of NIRVA file object containing the XSL file used for the transformation. This object must be in the input container. The default name is “XSL”.
XSL_NAME	This parameter can be given instead of the XSL parameter (it has priority on it). It allows to directly giving the name of an xslt file that resides in the application, system or service Files directory. It avoids the use the LOAD_XSL command. This parameter must correspond to an existing server xslt file. An xslt file can be at application, system or service level. For an application xslt file, the xslt file name is given as it is. For example: “`file1.xsl`”. This must correspond to an existing file of the application file directory. If there is no extension given, Nirva adds “`.xsl`” after the xslt name. The xslt file name is case insensitive even under UNIX. For a system xslt file, the xslt file name must be enclosed in brackets and preceded by the service name. For example: “`(file1.xsl)`”. This must correspond to an existing file of the system file directory. If there is no extension given, Nirva adds “`.xsl`” after the xslt name. The xslt file name is case insensitive even under UNIX. For a service xslt file, the xslt file name must be enclosed in brackets. For example: “`MyService(file1.xsl)`”. This must correspond to an existing file of the service file directory. If there is no extension given, Nirva adds “`.xsl`” after the xslt name. The xslt file name is case insensitive even under UNIX.
SKIP_DTD	Skips the DTD loading phase when set to “YES”. Default is “NO”.

VALIDATE

XML:VALIDATE

Source	Use Input Container	Use Output Container	Use Output buffer
Client Web Procedure Service	Yes	No	No

Description

This command validates an XML against a schema or dtd. If the validation is not successful, the command returns an error and the error information field contains the detail of the errors.

Parameters

XML	This is the name of NIRVA file or binary object containing the XML file or data to validate. This object must be in the input container. The default name is “XML”.
MODE	Validation mode. Set this parameter to “DTD” for validating against a dtd or “SCHEMA” to validate against a schema. The default is “SCHEMA”.
DTD	Name of NIRVA file or binary object containing the dtd when the mode is set to “DTD”. The default name is “DTD”.
SCHEMA	Name of NIRVA file or binary object containing the schema when the mode is set to “SCHEMA”. The default name is “SCHEMA”.

XML

Overview

XML processing is directly included in the NIRVA's heart.

Any command sent to NIRVA from a HTTP client (including web browsers) may include input XML data and deliver output XML data.

In fact, the input XML is transformed by NIRVA into NIRVA objects and subcontainers into the input container and the output XML is a view of the output container.

The input and output XML respect the NIRVA XML grammar described in this chapter. In order for NIRVA to integrate to or from external applications, this XML grammar can be changed to any kind of other XML grammar by using the embedded XSLT parser.

When using the SOAP connector, the NIRVA XML data is in the Body element of the SOAP message. The XSLT processor cannot be used with the SOAP connector.

NIRVA accepts data in ISO-8859-1 and UTF-8 encodings and automatically convert the received characters into the internal encoding (UTF-8 or ISO-8859-1).

Nirva delivers by default its output XML data in the internal encoding (UTF-8 or ISO-8859-1). This can be changed by a command parameter, or automatically by detecting the input encoding or by the way of the embedded XSLT processor.

NIRVA provides 4 XML data model: normal, simple, compact and tiny. The choice of the model depends of the connector used and the value of the NV_XML_MODEL parameter. For input, it's possible to use the normal or simple models and eventually a mix of them.

Input XML

A HTTP client can directly send XML data in the body of the message. Then NIRVA transforms the data into objects and subcontainers into the input container.

A parameter of the command also gives the name of an XSLT file that processes the XML input data flow to transform it into an acceptable XML for NIRVA.

Here is an example of Nirva XML input:

Normal model:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<NIRVA>
  <NVCOMMAND>
    <NVPARAM name="NV_CLASS">MISC</NVPARAM>
    <NVPARAM name="NV_COMMAND">NOP</NVPARAM>
  </NVCOMMAND>
  <NVOBJ name="zut" type="STRING">
    <NVDATA>Test string cet été là</NVDATA>
  </NVOBJ>
    <NVCONTAINER  name="toto" />
</NIRVA>
```

Simple model:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<nirva>
  <nvcommand>
    <nv_class>MISC</nv_class>
    <nv_command>NOP</nv_command>
  </nvcommand >
  </zut>Test string cet été là</zut>
  <toto  type="container" />
</nirva>
```

Output XML

Each time a command is sent to NIRVA from a HTTP client, NIRVA delivers a XML representation of the output container (or another container if required).

This XML data flow contains first some session variables (optional) and then the hierarchical content of the output container.

A parameter of the command also gives the name of an XSLT file that processes the XML data flow to transform it into another XML file or a displayable HTML file.

In this way, one can build applications directly using NIRVA and XSLT scripts. All the configuration layer of NIRVA has been made using this technique.

The NIRVA SYSTEM:XML:SET_XML also allows to transform a container or a part of it as NIRVA XML data flow.

Here is an example of Nirva XML output before calling the parser:

Normal model:

```
<NIRVA session="BA1A640876" application="NVDEF" user="nvdef" source="BROWSER"
host="localhost:1081" local="YES">
  <NVCONTAINER name="nvdef">
    <NVOBJ name="myfile" type="FILE">
      <NVNAME>toto.txt</NVNAME>
        <NVEXTENSION>txt</NVEXTENSION>
        <NVDIRECTORY>C:\nirva\Applications\NVDEF\Files</NVDIRECTORY>
        <NVPATHNAME>C:\nirva\Applications\NVDEF\Files\toto.txt</NVPATHNAME>
        <NVSIZE>0</NVSIZE>
        <NVORIGIN></NVORIGIN>
    </NVOBJ>
    <NVOBJ name="mystring" type="STRING">
      <NVDATA>Hello world!</NVDATA>
    </NVOBJ>
    <NVCONTAINER name="subcontainer">
      <NVOBJ name="sondages" type="TABLE">
        <NVDESCRIPTION></NVDESCRIPTION>
        <NVSIZE>2</NVSIZE>
        <NVPRIMARY case-sensitive="no"></NVPRIMARY>
        <NVCOLDESC name="ident" type="ALPHANUMERIC"></NVCOLDESC>
        <NVCOLDESC name="title" type="ALPHANUMERIC"></NVCOLDESC>
        <NVCOLDESC name="description" type="ALPHANUMERIC"></NVCOLDESC>
        <NVROW>
          <NVCOL name="ident">
            <NVDATA>1</NVDATA>
          </NVCOL>
          <NVCOL name="title">
            <NVDATA>Title1</NVDATA>
          </NVCOL>
        </NVROW>
        <NVROW>
          <NVCOL name="ident">
            <NVDATA>2</NVDATA>
          </NVCOL>
          <NVCOL name="title">
            <NVDATA>Title2</NVDATA>
          </NVCOL>
        </NVROW>
      </NVOBJ>
    </NVCONTAINER>
  </NVCONTAINER>
</NIRVA>
```

Simple model:

```
<nirva session="9A1742DF10" application="NVDEF" user="nvdef" source="BROWSER"
host="localhost:1081" local="YES">
  <myfile type="file">
    <name>toto.txt</name>
    <extension>txt</extension>
    <directory>C:\nirva\Applications\NVDEF\Files</directory>
    <pathname>C:\nirva\Applications\NVDEF\Files\toto.txt</pathname>
```

```
    <size>0</size>
    <origin></origin>
  </myfile>
  <mystring type="string">Hello world!</mystring>
  <subcontainer type="container">
    <sondages type="table">
      <description></description>
      <size>2</size>
      <primary case-sensitive="no"></primary>
      <coldesc name="ident" type="ALPHANUMERIC"></coldesc>
      <coldesc name="title" type="ALPHANUMERIC"></coldesc>
      <row>
        <col name="ident">
          <data>1</data>
        </col>
        <col name="title">
          <data>Title1</data>
        </col>
      </row>
      <row>
        <col name="ident">
          <data>2</data>
        </col>
        <col name="title">
          <data>Title2</data>
        </col>
      </row>
    </sondages>
  </subcontainer>
</nirva>
```

Compact model:

```
<nirva session="1BF7B11144" application="NVDEF" user="nvdef" source="BROWSER"
host="localhost:1081" local="YES">
  <myfile type="file">
    <name>toto.txt</name>
    <extension>txt</extension>
    <directory>C:\nirva\Applications\NVDEF\Files</directory>
    <pathname>C:\nirva\Applications\NVDEF\Files\toto.txt</pathname>
    <size>0</size>
    <origin></origin>
  </myfile>
  <mystring type="string">Hello world!</mystring>
  <subcontainer type="container">
    <sondages type="table">
      <row>
        <ident>1</ident>
        <title>Title1</title>
      </row>
      <row>
        <ident>2</ident>
        <title>Title2</title>
      </row>
    </sondages>
```

```
  </subcontainer>
</nirva>
```

Tiny model:

```
<nirva session="1BF7B11144" application="NVDEF" user="nvdef" source="BROWSER"
host="localhost:1081" local="YES">
  <myfile>
    <name>toto.txt</name>
    <extension>txt</extension>
    <directory>C:\nirva\Applications\NVDEF\Files</directory>
    <pathname>C:\nirva\Applications\NVDEF\Files\toto.txt</pathname>
    <size>0</size>
    <origin></origin>
  </myfile>
  <mystring>Hello world!</mystring>
  <subcontainer>
    <sondages>
      <row>
        <ident>1</ident>
        <title>Title1</title>
      </row>
      <row>
        <ident>2</ident>
        <title>Title2</title>
      </row>
    </sondages>
  </subcontainer>
</nirva>
```

Error XML

In case of error, NIRVA may deliver XML error data with a dedicated grammar. This grammar can itself be transformed into another one by using the NV_XML_XSL_ERROR parameter of the command.

Here is an example of NIRVA XML error output:

Normal model:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<NIRVA>
  <NVERROR>
    <NVERRORCODE>101</NVERRORCODE>
    <NVERRORSERVICE>SYSTEM</NVERRORSERVICE>
    <NVERRORCLASS>COMMAND</NVERRORCLASS>
    <NVERRORDESC>Command not available</NVERRORDESC>
    <NVERRORINFO>SYSTEM:MISC:NOPU</NVERRORINFO>
  </NVERROR>
</NIRVA>
```

Simple, Compact and Tiny models:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<nvfault>
  <code>101</code>
  <service>SYSTEM</service>
  <class>COMMAND</class>
  <description>Command not available</description>
  <info>SYSTEM:MISC:NOPU</info>
</nvfault>
```

The error output for the SOAP connector is different but the given information is the same:

Normal model:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <env:faultcode>Server</env:faultcode>
    <env:faultstring>NIRVA-ERROR:SYSTEM:COMMAND:101:Command not
available:SYSTEM:MISC:NOPU</env:faultstring>
    <env:faultactor>127.0.0.1:1081</env:faultactor>
    <env:detail>
      <nvs:NVERROR xmlns:nvs="http://www.nirva-systems.com/2004-03-29/nirva-xml-fault">
      <nvs:NVERRORCODE>101</nvs:NVERRORCODE>
      <nvs:NVERRORSERVICE>SYSTEM</nvs:NVERRORSERVICE>
      <nvs:NVERRORCLASS>COMMAND</nvs:NVERRORCLASS>
      <nvs:NVERRORDESC>Command not available</nvs:NVERRORDESC>
      <nvs:NVERRORINFO>SYSTEM:MISC:NOPU</nvs:NVERRORINFO>
      </nvs:NVERROR>
    </env:detail>
  </env:Body>
</env:Envelope>
```

Simple model:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Body>
    <env:faultcode>Server</env:faultcode>
    <env:faultstring>NIRVA-ERROR:SYSTEM:COMMAND:101:Command not
available:SYSTEM:MISC:NOPU</env:faultstring>
    <env:faultactor>127.0.0.1:1081</env:faultactor>
    <env:detail>
      <nvs:nvfault xmlns:nvs="http://www.nirva-systems.com/2004-03-29/nirva-xml-fault">
       <nvs:code>101</nvs:code>
       <nvs:service>SYSTEM</nvs:service>
```

```
      <nvs:class>COMMAND</nvs:class>
      <nvs:description>Command not available</nvs:description>
      <nvs:info>SYSTEM:MISC:NOPU</nvs:info>
     </nvs:nvfault>
    </env:detail>
  </env:Body>
</env:Envelope>
```

Parsers

Nirva provides an internal XSLT parser for transforming the XML data to another XML data (or HTML data). NIRVA may aalso be configured to use external parsers (see configuration chapter). The parsers must implement the XSLT language.

Normal model

This chapter gives the list of XML elements of the NIRVA XML grammar for the normal model. We distinguish different grammars for input, output and error XML.

Input

Main structure

```
<NIRVA>
   <NVCOMMAND>
    <NVPARAM></ NVPARAM >
    <NVPARAM></ NVPARAM >
    ..
   </NVCOMMAND>
   <NVOBJ>
   </NVOBJ>
   <NVOBJ>
   </NVOBJ>
   ..
   <NVCONTAINER>
     <NVOBJ>
     </NVOBJ>
     <NVOBJ>
     </NVOBJ>
     ..
   </NVCONTAINER>
   <NVCONTAINER>
     <NVOBJ>
```

```
        </NVOBJ>
        <NVOBJ>
        </NVOBJ>
        ..
      </NVCONTAINER>
      ..
   <NIRVA>
```

The global XML element is named “NIRVA”.

The NIRVA element can contain 3 elements: NVCOMMAND (optional) contains command parameters, NVOBJ are objects for the root of the NIRVA input container and NVCONTAINER are subcontainers themselves containing other subcontainers or objects.

Reference

NIRVA

Description

This is the root XLM element of each NIRVA XML data. All the rest of XML data flow is encapsulated in this element.

Attributes

None

Data

This element has no data.

Elements

The NIRVA element can contain only NVCOMMAND, NVOBJ or NVCONTAINER elements. However, if it contains other elements, these one are not taken in care but NIRVA doesn't produce any error.

NVCOMMAND

Description

This element encapsulates NIRVA command parameters. It's optional because the parameter can also be given in the URL. If it's here, it must be the immediate child of the NIRVA element. Only one command can be send in an input XML data flow.

Attributes

None.

Data

This element has no data.

Elements

The NVCOMMAND element can contain only NVPARAM elements.

NVPARAM

Description

Command parameter. This element must be a child of the NVCOMMAND element. It gives a NIRVA command parameter. It overrides any command parameter having the same name given in the URL.

Attributes

name	Command parameter name.

Data

Command parameter value.

Elements

The NVPARAM element cannot contain other elements.

NVCONTAINER

Description

This element encapsulates a NIRVA subcontainer. The subcontainer itself is composed of NIRVA objects or other subcontainers. The NVCONTAINER elements that are direct children of the NIRVA element will become subcontainers of the input container.

Attributes

name	NIRVA subcontainer name. A subcontainer name is case independent. This attribute is optional. If the subcontainer name is not given, NIRVA automatically generates a name on the form “cont*x*” where *x* is a number

starting to 1. If a subcontainer with the same name already exists, NIRVA doesn't remove it but just appends objects to it.

Data

This element has no data.

Elements

The NVCONTAINER element can contain only NVOBJ or other NVCONTAINER elements.

NVOBJ

Description

This element encapsulates a NIRVA object. Its content depends of the type of object. If the object is a file object, it will be created as a temporary file object in the application work directory.

Attributes

name — NIRVA object name. An object name is case independent. This attribute is optional. If the object name is not given, NIRVA automatically generates a name on the form "obj*x*" where *x* is a number starting to 1. If an object with the same name already exists, NIRVA replaces it with the new one.

type — NIRVA object type. This attribute can take values "BOOLEAN", "INTEGER", "STRING", "STRINGLIST", "INDSTRINGLIST", "TABLE", "FILE" or "BINARY following the NIRVA object type.

case_sensitive — Case sensitive flag for indexed string list objects. This parameter has meaning only when the type is "INDSTRINGLIST". If set to "YES" (or "yes"), the created indexed string list object is case sensitive (for the key). The default is case insensitive.

extension — Extension for file objects. This attribute has meaning only when the type is "FILE". It gives the extension for the created file name. If no extension is given, NIRVA sets a default extension. The attribute is optional.

prefix — Prefix for file objects. This attribute has meaning only when the type is "FILE". If a prefix is given, NIRVA will use it for building the created file name. The attribute is optional.

suffix — Suffix for file objects. This attribute has meaning only when the type is "FILE". If a suffix is given, NIRVA will use it for building the created file name. The attribute is optional.

Data

This element has no data.

Elements

The NVOBJ element can contain several other elements following its type.

- A boolean NVOBJ contains one and only one NVDATA element.
- An integer NVOBJ contains one and only one NVDATA element.
- A string NVOBJ contains one and only one NVDATA element.
- A string list NVOBJ contains one NVDATA element for each string of the list.
- An indexed string list NVOBJ contains one NVDATA element for each string of the list.
- A table NVOBJ contains one NVDESCRIPTION element, one NVPRIMARY element, several NVCOLDESC elements (one for each column of the table) and several NVROW elements (one for each row).
- A file NVOBJ contains optionally one NVDATA element and one NVORIGIN element.
- A binary NVOBJ contains optionally one NVDATA element.

NVDATA

Description

This element gives some NIRVA object data. Its content depends of the type of object.

Attributes

key	This attribute is used only if the NVDATA element is for an indexed string list object. At this time it's the index of the list item. It can be any string.

Data

Data or part of the data of the object. The data depends of the object type.

- For a boolean object, the data can take values “true”, “false”, “TRUE”, “FALSE”,”0” or “1”. There is only one NVDATA element for one object.
- For an integer object, the data is the integer value. There is only one NVDATA element for one object.
- For a string object, the data is the string value. There is only one NVDATA element for one object.
- For a string list object, the data is the string value of each item. There can have several NVDATA elements for one object (one for each item of the list).
- For an indexed string list object, the data is the string value of each item. There can have several NVDATA elements for one object (one for each item of the list). The item key is then given in the key attribute of the NVDATA element. If the key attribute is empty, the entry is not added to the object.
- For a table object, the data is a cell line. There are several NVDATA elements for one object.
- For a file or a binary object, the NVDATA element is optional. If here, it must contain the base64 coded data of the file or binary object.

Elements

The NVDATA element cannot contain other elements.

NVDESCRIPTION

Description

This element gives the description of a NIRVA table object. It can be anywhere as child element of a NVOBJ element of table type. It’s not mandatory.

Attributes

None

Data

Table description.

Elements

The NVDESCRIPTION element cannot contain other elements.

NVPRIMARY

Description

This element gives the primary column for a table. It can be anywhere as child element of a NVOBJ element of table type. It’s not mandatory.

Attributes

case_sensitive	Tells if the primary column index is case sensitive or not. It can take values “yes” or “no”. The default is no.

Data

Table primary column.

Elements

The NVPRIMARY element cannot contain other elements.

NVCOLDESC

Description

This element gives the column name and description for a table object. All the table NVCOLDESC elements must precede any NVROW elements.

Attributes

name	Column name. Mandatory. The column name cannot contain spaces.
type	Column type. Can be “NUMERIC” or “ALPHANUMERIC”. Default is “ALPHANUMERIC”.

Data

Column description.

Elements

The NVCOLDESC element cannot contain other elements.

NVROW

Description

This element is a record of a table object.

Attributes

None

Data

This element has no data.

Elements

The NVROW element contains one NVCOL element for each column.

NVCOL

Description

This element encapsulates a cell of a table object.

Attributes

name	Column name.

Data

This element has no data.

Elements

The NVCOL element contains one NVDATA element for each cell line.

NVORIGIN

Description

This element contains the optional file origin name for a file object. It must be the first element of the NVOBJ element for a file object.

Attributes

None

Data

File origin name.

Elements

The NVORIGIN element cannot contain other elements.

Output

Main structure

```
<NIRVA>
  <NVHTTPHEADER></NVHTTPHEADER>
  <NVHTTPHEADER></NVHTTPHEADER>
  ..
  <NVSESSIONVAR></NVSESSIONVAR>
  <NVSESSIONVAR></NVSESSIONVAR>
  ..
  <NVCONTAINER>
    <NVOBJ>
    </NVOBJ>
    <NVOBJ>
```

```
      </NVOBJ>
      ..
      <NVCONTAINER>
        <NVOBJ>
        </NVOBJ>
        <NVOBJ>
        </NVOBJ>
        ..
      </NVCONTAINER>
      ..
    </NVCONTAINER>
  <NIRVA>
```

The global XML element is named “NIRVA”.

The NIRVA element can contain 3 elements: NVHTTPHEADER (optional) represents a HTTP header has sent by the WEB browser, NVSESSIONVAR (optional) is a NIRVA session variable and NVCONTAINER is a NIRVA container.

Each container itself is composed of NVOBJ elements representing the container objects or other NVCONTAINER elements representing the subcontainers.

The element attributes are always in lower case.

Reference

NIRVA

Description

This is the root XLM element of each NIRVA XML data. All the rest of XML data flow is encapsulated in this element.

Attributes

session	NIRVA session ID.
application	name of the connected NIRVA application.
user	User name of the application.
source	This attribute can take the values “BROWSER”, “CLIENT”, “PROCEDURE” or “SERVICE” following the origin of the command which has generated the XML data flow.
host	Server name or address. This attribute gives the name of the server as received in the URL.
local	This attribute is set to “YES” if the host is “127.0.0.1” or “localhost” and to “NO” otherwise.

Data

This element has no data.

Elements

The NIRVA element can contain only NVHTTPHEADER, NSESSIONVAR or NVCONTAINER elements.

NVHTTPHEADER

Description

This element encapsulates a HTTP header as received by the command. This is directly the http headers generated by the WEB browser when the command comes from a WEB browser (or any HTTP client). The NVHTTPHEADER elements are optional and are controlled by the NV_XML_HTTP_HEADERS command parameter.

Attributes

key	HTTP header name. Always uppercase

Data

HTTP header data.

Elements

The NVHTTPHEADER element cannot contain other elements.

NVSESSIONVAR

Description

This element encapsulates a NIRVA session variable. When a command comes from a WEB browser, all HTTP headers are also transformed into NIRVA session variables so, in this case, the information given in the NVHTTPHEADER elements is also given in NVSESSIONVAR elements.

The NVSESSIONVAR elements are optional and are controlled by the NV_XML_VARIABLES command parameter.

Attributes

key	NIRVA session variable name. Always uppercase.

Data

NIRVA session variable data.

Elements

The NVSESSIONVAR element cannot contain other elements.

NVCONTAINER

Description

This element encapsulates a NIRVA container or subcontainer. The container itself is composed of NIRVA objects or subcontainers.

Attributes

name	NIRVA container name. Always lowercase.

Data

This element has no data.

Elements

The NVCONTAINER element can contain only NVOBJ or other NVCONTAINER elements.

NVOBJ

Description

This element encapsulates a NIRVA object. Its content depends of the type of object.

Attributes

name	NIRVA object name. Always lowercase.
type	NIRVA object type. This attribute can take values “BOOLEAN”, “INTEGER”, “STRING”, “STRINGLIST”, “INDSTRINGLIST”, “TABLE”, “FILE” or “BINARY following the NIRVA object type.
case-sensitive	Set to yes or no for an indexed string list object. Not given for other objects.

Data

This element has no data.

Elements

The NVOBJ element can contain several other elements following its type:

- A boolean NVOBJ contains one and only one NVDATA element. If the general command parameter NV_XML_WITH_DATA has been set to “NO”, the boolean NVOBJ doesn’t contain any element.
- An integer NVOBJ contains one and only one NVDATA element. If the general command parameter NV_XML_WITH_DATA has been set to “NO”, the integer NVOBJ doesn’t contain any element.
- A string NVOBJ contains one and only one NVDATA element. If the general command parameter NV_XML_WITH_DATA has been set to “NO”, the string NVOBJ doesn’t contain any element.
- A string list NVOBJ contains one NVDATA element for each string of the list. If the general command parameter NV_XML_WITH_DATA has been set to “NO”, the string list NVOBJ doesn’t contain any element.
- An indexed string list NVOBJ contains one NVDATA element for each string of the list. If the general command parameter NV_XML_WITH_DATA has been set to “NO”, the string list NVOBJ doesn’t contain any element.
- A table NVOBJ contains one NVDESCRIPTION element, one NVPRIMARY element, one NVSIZE element, several NVCOLDESC elements (one for each column of the table) and several NVROW elements (one for each row). If the general command parameter NV_XML_WITH_DATA has been set to “NO”, the table NVOBJ doesn’t contain the NVROW elements.
- A file NVOBJ contains one NVNAME element, one NVEXTENSION element, one NVDIRECTORY element, one NVPATHNAME, one NVORIGIN element and one NVSIZE element. A file NVOBJ may contain optionally one NVDATA element if the general command parameter NV_XML_WITH_DATA has been set to “ALL”.
- A binary NVOBJ contains one NVSIZE element. A binary NVOBJ may contain optionally one NVDATA element if the general command parameter NV_XML_WITH_DATA has been set to “ALL”.

NVDATA

Description

This element gives some NIRVA object data. Its content depends of the type of object.

Attributes

key	This attribute is used only if the NVDATA element is for an indexed string list object. At this time it’s the index of the list item. It can be any string.

Data

Data or part of the data of the object. The data depends of the object type.

- For a boolean object, the data can take values “true” or “false”. There is only one NVDATA element for one object.

- For an integer object, the data is the integer value. There is only one NVDATA element for one object.
- For a string object, the data is the string value. There is only one NVDATA element for one object.
- For a string list object, the data is the string value of each item. There are several NVDATA elements for one object (one for each item of the list).
- For an indexed string list object, the data is the string value of each item. There are several NVDATA elements for one object (one for each item of the list). The item key is then given in the key attribute of the NVDATA element.
- For a table object, the data is a cell line. There are several NVDATA elements for one object.
- For a file or a binary object, the NVDATA element is optional. If here, it contains the base64 coded data of the file or binary object.

Elements

The NVDATA element cannot contain other elements.

NVDESCRIPTION

Description

This element gives the description of a NIRVA table object.

Attributes

None

Data

Table description.

Elements

The NVDESCRIPTION element cannot contain other elements.

NVPRIMARY

Description

This element gives the primary column for a table object.

Attributes

Case-sensitive	Tells if the primary column is case sensitive or not.

Data

Primary column name. Empty if no primary column.

Elements

The NVPRIMARY element cannot contain other elements.

NVSIZE

Description

This element gives the size of some of the NIRVA objects. In fact, it's used only for table, file and binary object.

Attributes

None

Data

Number of rows of the table for a table object.

File size in bytes for a file object.

Size of data (in bytes) for a binary object.

Elements

The NVSIZE element cannot contain other elements.

NVCOLDESC

Description

This element gives the column name and description for a table object.

Attributes

name	Column name.
type	Column type. Can be "NUMERIC" or "ALPHANUMERIC".

Data

Column description.

Elements

The NVCOLDESC element cannot contain other elements.

NVROW

Description

This element is a record of a table object.

Attributes

None

Data

This element has no data.

Elements

The NVROW element contains one NVCOL element for each column.

NVCOL

Description

This element encapsulates a cell of a table object.

Attributes

name	Column name.

Data

This element has no data.

Elements

The NVCOL element contains one NVDATA element for each cell line.

NVNAME

Description

File name of a file object.

Attributes

None

Data

File name (without path) associated with the file object.

NVEXTENSION

Description

File extension of a file object.

Attributes

None

Data

File extension (without the point character) of the file object.

Elements

The NVEXTENSION element cannot contain other elements.

NVDIRECTORY

Description

File directory of a file object.

Attributes

None

Data

File directory (without the file name) of the file object.

Elements

The NVDIRECTORY element cannot contain other elements.

NVPATHNAME

Description

File complete pathname of a file object.

Attributes

None

Data

File complete pathname of the file object.

Elements

The NVPATHNAME element cannot contain other elements.

NVORIGIN

Description

Origin of a file object.

Attributes

None

Data

Origin of the file object.

Elements

The NVORIGIN element cannot contain other elements.

Error

Main structure

```
<NIRVA>
  <NVERROR>
    <NVERRORCODE></NVERRORCODE>
    <NVERRORSERVICE></NVERRORSERVICE>
    <NVERRORCLASS></NVERRORCLASS>
    <NVERRORDESC></NVERRORDESC>
    <NVERRORINFO></NVERRORINFO>
  </NVERROR>
</NIRVA>
```

The global XML element is named “NIRVA”.

The NIRVA element contain a single element named NVERROR that itself contains the error information

Reference

NIRVA

Description

This is the root XLM element of each NIRVA XML data. All the rest of XML data flow is encapsulated in this element.

Attributes

session	NIRVA session ID. If there is no session ID, this attribute is not given.
application	name of the connected NIRVA application. If there is no session ID, this attribute is not given.
user	User name of the application. If there is no session ID, this attribute is not given.

Data

This element has no data.

Elements

The NIRVA element contains one and only one NVERROR element.

NVERROR

Description

This element encapsulates NIRVA error information. It’s always the immediate child of the NIRVA element.

Attributes

None.

Data

This element has no data.

Elements

The NVERROR element contains exactly one NVERRORCODE, one NVERRORSERVICE, one NVERRORCLASS, one NVERRORDESC and one NVERRORINFO elements.

NVERRORCODE

Description

Error code.

Attributes

None.

Data

Error code.

Elements

The NVERRORCODE element cannot contain other elements.

NVERRORSERVICE

Description

Error service. Name of the NIRVA service producing the error.

Attributes

None.

Data

Service name.

Elements

The NVERRORSERVICE element cannot contain other elements.

NVERRORCLASS

Description

Error class.

Attributes

None.

Data

Error class.

Elements

The NVERRORCLASS element cannot contain other elements.

NVERRORDESC

Description

Error description in the language of the session.

Attributes

None.

Data

Error description.

Elements

The NVERRORDESC element cannot contain other elements.

NVERRORINFO

Description

Error extra information.

Attributes

None.

Data

Error extra information.

Elements

The NVERRORINFO element cannot contain other elements.

Simple model

This chapter gives the list of XML elements of the NIRVA XML grammar for the simple model. We distinguish different grammars for input, output and error XML.

Input

Main structure

```
<nirva>
   <nvcommand>
    <Param1></Param1 >
    <Param2></Param2 >
    ..
   </nvcommand>
   <Obj1>
   </Obj1>
   <Obj2>
   </Obj2>
```

```
  ..
  <Container1>
    <Obj1>
    </Obj1>
    <Obj2>
    </Obj2>
    ..
    <Container11>
      ..
    </Container11>
  </Container1>
<nirva>
```

The global XML element is mandatory but can have any name (here “nirva”).

The nvcommand element if exists must be the direct child of the main XML element. It contains optional command parameters. Each parameter is an element where element name is the parameter name and element value is the parameter value.

Other elements are objects or subcontainers. The distinction is made by an attribute named “type”. The type attribute can take the values “container”, “boolean”, “integer”, “string”, “stringlist”, “indstringlist”, “table”, “file” or “binary”. If the element is a container, the type attribute can be omitted if the element has at least one child. If the element is a string object, the type attribute can be omitted.

Reference

Main element

Description

This is the root XLM element of each NIRVA XML data. All the rest of XML data flow is encapsulated in this element. The element name is free.

Attributes

None

Data

This element has no data.

Elements

The NIRVA element can contain the nvcommand element or containers or objects.

Command parameters

Description

The command parameters are encapsulated in the nvcommand element. It’s optional because the parameter can also be given in the URL. If it’s here, it must be the immediate child of the Main element. Only one command can be send in an input XML data flow.

Here is the structure of the nvcommand element:

```
<nvcommand>
  <paraname1>Value</paraname1>
  <paraname2>Value</paraname2>
  …
</nvcommand>
```

Where paranameX is the parameter name and Value is the parameter value.

Attributes

None.

Container

Description

A container is an element having the attribute type set to “container”. The container name is the element name (case insensitive).

A container can contain objects or other containers.

Attributes

type	Must be “container”. This attribute is not mandatory if the container element contains at least one child element.

Boolean object

Description

A boolean object has the following structure:

```
<objname type="boolean">Value</objname>
```

Where objname is the object name and Value can be “true”, “false”, “TRUE”, “FALSE”,”0” or “1”.

Attributes

type	Must be “boolean”.

Integer object

Description

An integer object has the following structure:

```
<objname type="integer">Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

type	Must be “integer”.

String object

Description

A string object has the following structure:

```
<objname type="string">Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

type	Must be “string”. This parameter is optional.

StringList object

Description

A stringlist object has the following structure:

```
<objname type="stringlist">
  <data>Value1</data>
  <data>Value2</data>
  …
  <data>Valuen</data>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the string list.

Attributes

type	Must be “stringlist”.

IndStringList object

Description

An indstringlist object has the following structure:

```
<objname type="indstringlist" case-sensitive="no">
  <data key="key1">Value1</data>
  <data key="key2">Value2</data>
  …
  <data key="keyn">Valuen</data>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the indexed string list. The index is given by the key attribute of each data element.

Attributes

type	Must be “indstringlist”.
case_sensitive	If this attribute is set to “YES” (or “yes”), the created indexed string list object is case sensitive (for the key). The default is case insensitive.

Table object

Description

A table object has the following structure:

```
<objname type="table">
  <description>Table description</description>
  <primary case-sensitive="no">Table primary column</primary>
  <coldesc name="column1 name" type="column1 type">Column1 description</coldesc >
  <coldesc name="column2 name" type="column2 type">Column2 description</coldesc >
  …
  <coldesc name="columnn name" type="columnn type">Columnn description</coldesc >
  <row>
    <col name="column1 name">
      <data>Value1</data>
      <data>Value2</data>
      …
      <data>Valuen</data>
    </col>
    <col name="column2 name">
      <data>Value1</data>
      <data>Value2</data>
      …
      <data>Valuen</data>
    </col>
    …
    <col name="columnn name">
     <data>Value1</data>
     <data>Value2</data>
     …
     <data>Valuen</data>
    </col>
  </row>
  <row>
  </row>
  …
  <row>
  </row>
</objname>
```

Where objname is the object name.

A table object can have a description given in the description element. This must be a direct child of the table object element.

The primary column can be given in the primary element. It’s not mandatory. The primary column index can be case sensitive or not.

The next tags define the column names of the table. Each column description is encapsulated into a coldesc element. The name attribute is mandatory. The column name cannot contain spaces. The type attribute can take values “ALPHANUMERIC” (default) or “NUMERIC”. The columns will be created in the same order than the coldesc element order.

Then, the table data follows. Each row is encapsulated in a row element. Into each row element, the col elements allow separating the column data. The name attribute of the column element is optional and should be used only if the column data is given in a different order than the column description. Into each col element, the cell lines are given in data elements.

Attributes

type	Must be “table”.

File object

Description

A file object has the following structure:

```
<objname type="file">
  <origin>Origin file name</origin>
  <data>File data</data>
< /objname>
```

Where objname is the object name and File data is the file data. The file data must be base64 encoded. The created file objects are temporary file objects (persistence 0). The persistence value can be changed inside a procedure by dedicated commands.

The origin tag is optional.

Attributes

type	Must be “file”.
extension	This attribute gives the extension for the created file name. If no extension is given, NIRVA sets a default extension. The attribute is optional.
prefix	Prefix for file name. If a prefix is given, NIRVA will use it for building the created file name. The attribute is optional.
suffix	Suffix for file name. If a suffix is given, NIRVA will use it for building the created file name. The attribute is optional.

Binary object

Description

A file object has the following structure:

```
<objname type="binary">
  <data>Binary data</data>
< /objname>
```

Where objname is the object name and Binary data is the binary data. The binary data must be base64 encoded.

Attributes

type	Must be “binary”.

Output

Main structure

```
<nirva>
   <httpheader></httpheader>
   <httpheader></httpheader>
   <variable></variable>
   <variable></variable>
   <Obj1>
   </Obj1>
   <Obj2>
   </Obj2>
   ..
   <Container1>
     <Obj1>
     </Obj1>
     <Obj2>
     </Obj2>
     ..
   </Container1>
   <Container2>
     <Obj1>
     </Obj1>
     <Obj2>
     </Obj2>
     ..
   </Container2>
   ..
<nirva>
```

The global XML element is mandatory but can have any name (here “nirva”).

The httpheader elements contain input http headers.

The variable elements contain session variables.

Other elements are objects or subcontainers. The distinction is made by an attribute named “type”. The type attribute can take the values “container”, “boolean”, “integer”, “string”, “stringlist”, “indstringlist”, “table”, “file” or “binary”.

Reference

Main element

Description

This is the root XLM element of each NIRVA XML data. All the rest of XML data flow is encapsulated in this element. The element name is always “nirva”.

Following the connector used and the parameter NV_XML_SESSION_INFO parameter value, the nirva main element main contains some attributes or not.

Attributes

session	NIRVA session ID.
application	name of the connected NIRVA application.
user	User name of the application.
source	This attribute can take the values “BROWSER”, “CLIENT”, “PROCEDURE” or “SERVICE” following the origin of the command which has generated the XML data flow.
host	Server name or address. This attribute gives the name of the server as received in the URL.
local	This attribute is set to “YES” if the host is “127.0.0.1” or “locahost” and to “NO” otherwise.

HTTP headers

Description

The HTTP headers are given in the httpheader elements that are immediate child of the main nirva element.

Following the connector used and the parameter NV_XML_HTTP_HEADERS parameter value, the HTTP headers may be displayed or not.

Attributes

key	HTTP header name.

Session variables

Description

The session variables are given in the variable elements that are immediate child of the main nirva element.

Following the connector used and the parameter NV_XML_VARIABLES parameter value, the session variables may be displayed or not.

Attributes

key	Variable name.

Container

Description

A container is an element having the attribute type set to “container”. The container name is the element name (always lowercase).

A container can contain objects or other containers.

Attributes

type	Always set to “container”.

Boolean object

Description

A boolean object has the following structure:

```
<objname type="boolean">Value</objname>
```

Where objname is the object name and Value can be “true”, “false”.

Attributes

type	Always set to “boolean”.

Integer object

Description

An integer object has the following structure:

```
<objname type="integer">Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

type Always set to “integer”.

String object

Description

A string object has the following structure:

```
<objname type="string">Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

type Always set to “string”.

StringList object

Description

A stringlist object has the following structure:

```
<objname type="stringlist">
  <data>Value1</data>
  <data>Value2</data>
  …
  <data>Valuen</data>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the string list.

Attributes

type	Always set to “stringlist”.

IndStringList object

Description

An indstringlist object has the following structure:

```
<objname type="indstringlist" case-sensitive="no">
  <data key="key1">Value1</data>
  <data key="key2">Value2</data>
  …
  <data key="keyn">Valuen</data>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the indexed string list. The index is given by the key attribute of each data element.

Attributes

type	Always set to “indstringlist”.
case-sensitive	Tells if the object has case sensitive keys or not (yes or no).

Table object

Description

A table object has the following structure:

```
<objname type="table">
  <description>Table description</description>
  <primary case-sensitive="no">Table primary column</primary>
  <size>Table size</size>
  <coldesc name="column1 name" type="column1 type">Column1 description</coldesc >
  <coldesc name="column2 name" type="column2 type">Column2 description</coldesc >
  …
  <coldesc name="columnn name" type="columnn type">Columnn description</coldesc >
  <row>
    <col name="column1 name">
```

```
      <data>Value1</data>
      <data>Value2</data>
      …
      <data>Valuen</data>
     </col>
     <col name="column2 name">
      <data>Value1</data>
      <data>Value2</data>
      …
      <data>Valuen</data>
     </col>
     …
     <col name="columnn name">
      <data>Value1</data>
      <data>Value2</data>
      …
      <data>Valuen</data>
     </col>
   </row>
   <row>
   </row>
   …
   <row>
   </row>
 </objname>
```

Where objname is the object name.

The description element gives the table description.

The primary element gives the table primary column.

The size element gives the number of rows in the table.

The next tags give the table column descriptions. Each column description is encapsulated into a coldesc element. The name attribute gives the column name. The type attribute can take values “ALPHANUMERIC” or “NUMERIC”.

Then, the table data follows. Each row is encapsulated in a row element. Into each row element, the col elements allow separating the column data. The name attribute of the column element gives the column name. The column order corresponds to the order of the column descriptions.

Attributes

type	Always set to “table”.

File object

Description

A file object has the following structure:

```
<objname type="file">
  <name>File name</name>
  <extension>File extension</extension>
  <directory>File directory</directory>
  <pathname>File complete path name</pathname>
  <size>File size</size>
  <origin>File origin name</origin>
  <data>File data</data>
< /objname>
```

Where objname is the object name, File name is the file name, File extension is the file extension (without the point character), file directory is the file directory, File complete path name is the complete file pathname, File size is the file size in bytes and File data is the file data base64 encoded.

Attributes

type Always set to “file”.

Binary object

Description

A file object has the following structure:

```
<objname type="binary">
  <size>Data size</size>
  <data>Binary data</data>
< /objname>
```

Where objname is the object name and Binary data is the binary data base64 encoded.

Attributes

type Always set to “binary”.

Error

Main structure

```
<nvfault>
  <code></code>
  <service></service>
  <class></class>
```

```
   <description></description>
   <info></info>
</nvfault>
```

The global XML element is named “nvfault” and contains the error information

Reference

nvfault

Description

This is the root XLM element of the NIRVA fault message. All the rest of XML error data flow is encapsulated in this element.

Attributes

session	NIRVA session ID. If there is no session ID, this attribute is not given.
application	name of the connected NIRVA application. If there is no session ID, this attribute is not given.
user	User name of the application. If there is no session ID, this attribute is not given.
Language	Language of the error information.

code

Description

Error code.

Attributes

None.

service

Description

Error service. Name of the NIRVA service producing the error.

Attributes

None.

---

class

Description

Error class.

Attributes

None.

---

description

Description

Error description in the language of the session.

Attributes

None.

---

info

Description

Error extra information.

Attributes

None.

Compact model

This chapter gives the list of XML elements of the NIRVA XML grammar for the compact model. We distinguish different grammars for output and error XML. This model is not used for input.

Output

Main structure

```
<nirva>
   <variable></variable>
   <variable></variable>
   <Obj1>
   </Obj1>
   <Obj2>
   </Obj2>
   ..
   <Container1>
     <Obj1>
     </Obj1>
     <Obj2>
     </Obj2>
     ..
   </Container1>
   <Container2>
     <Obj1>
     </Obj1>
     <Obj2>
     </Obj2>
     ..
   </Container2>
   ..
<nirva>
```

The global XML element is mandatory but can have any name (here “nirva”).

The variable elements contain session variables.

Other elements are objects or subcontainers. The distinction is made by an attribute named “type”. The type attribute can take the values “container”, “boolean”, “integer”, “string”, “stringlist”, “indstringlist”, “table”, “file” or “binary”.

Reference

Main element

Description

This is the root XLM element of each NIRVA XML data. All the rest of XML data flow is encapsulated in this element. The element name is always “nirva”.

Following the connector used and the parameter NV_XML_SESSION_INFO parameter value, the nirva main element main contains some attributes or not.

Attributes

session	NIRVA session ID.
application	name of the connected NIRVA application.
user	User name of the application.
source	This attribute can take the values “BROWSER”, “CLIENT”, “PROCEDURE” or “SERVICE” following the origin of the command which has generated the XML data flow.
host	Server name or address. This attribute gives the name of the server as received in the URL.
local	This attribute is set to “YES” if the host is “127.0.0.1” or “locahost” and to “NO” otherwise.

Session variables

Description

The session variables are given in the variable elements that are immediate child of the main nirva element.

Following the connector used and the parameter NV_XML_VARIABLES parameter value, the session variables may be displayed or not.

Attributes

key	Variable name.

Container

Description

A container is an element having the attribute type set to “container”. The container name is the element name (always lowercase).

A container can contain objects or other containers.

Attributes

type	Always set to “container”.

Boolean object

Description

A boolean object has the following structure:

```
<objname type="boolean">Value</objname>
```

Where objname is the object name and Value can be “true”, “false”.

Attributes

type	Always set to “boolean”.

Integer object

Description

An integer object has the following structure:

```
<objname type="integer">Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

type	Always set to “integer”.

String object

Description

A string object has the following structure:

```
<objname type="string">Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

type Always set to “string”.

StringList object

Description

A stringlist object has the following structure:

```
<objname type="stringlist">
  <data>Value1</data>
  <data>Value2</data>
  …
  <data>Valuen</data>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the string list.

Attributes

type Always set to “stringlist”.

IndStringList object

Description

An indstringlist object has the following structure:

```
<objname type="indstringlist">
  <key1>Value1</key1>
  <key2>Value2</key2>
  …
  <keyn>Valuen</keyn>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the indexed string list. The tag name is directly the key value (be careful to use keys that can be accepted as xml tag names).

Attributes

type Always set to “indstringlist”.

Table object

Description

A table object has the following structure:

```
<objname type="table">
  <description>Table description</description>
  <primary case-sensitive="no">Table primary column</primary>
  <size>Table size</size>
  <row>
    <col1>Value1</col1>
    <col2>Value2</col2>
     …
    <coln>Valuen</coln>
  </row>
  <row>
  </row>
  …
  <row>
  </row>
</objname>
```

Where objname is the object name.

The description element gives the table description.

The primary element gives the table primary column.

The size element gives the number of rows in the table.

Then, the table data follows. Each row is encapsulated in a row element. Into each row element, the column name is directly the tag name (be careful to use only authorized XML characters for column names).

Attributes

type	Always set to “table”.

File object

Description

A file object has the following structure:

```
<objname type="file">
  <name>File name</name>
  <extension>File extension</extension>
  <directory>File directory</directory>
```

```
    <pathname>File complete path name</pathname>
    <size>File size</size>
    <origin>File origin name</origin>
    <data>File data</data>
  < /objname>
```

Where objname is the object name, File name is the file name, File extension is the file extension (without the point character), file directory is the file directory, File complete path name is the complete file pathname, File size is the file size in bytes and File data is the file data base64 encoded.

Attributes

type	Always set to “file”.

Binary object

Description

A file object has the following structure:

```
  <objname type="binary">
    <size>Data size</size>
    <data>Binary data</data>
  < /objname>
```

Where objname is the object name and Binary data is the binary data base64 encoded.

Attributes

type	Always set to “binary”.

Error

Main structure

```
<nvfault>
  <code></code>
  <service></service>
  <class></class>
  <description></description>
  <info></info>
</nvfault>
```

The global XML element is named “nvfault” and contains the error information

Reference

---

nvfault

Description

This is the root XLM element of the NIRVA fault message. All the rest of XML error data flow is encapsulated in this element.

Attributes

session	NIRVA session ID. If there is no session ID, this attribute is not given.
application	name of the connected NIRVA application. If there is no session ID, this attribute is not given.
user	User name of the application. If there is no session ID, this attribute is not given.
Language	Language of the error information.

---

code

Description

Error code.

Attributes

None.

---

service

Description

Error service. Name of the NIRVA service producing the error.

Attributes

None.

---

class

Description

Error class.

Attributes

None.

---

description

Description

Error description in the language of the session.

Attributes

None.

---

info

Description

Error extra information.

Attributes

None.

Tiny model

This chapter gives the list of XML elements of the NIRVA XML grammar for the tiny model. We distinguish different grammars for output and error XML. This model is not used for input.

Output

Main structure

```
<nirva>
   <variable></variable>
   <variable></variable>
   <Obj1>
   </Obj1>
   <Obj2>
   </Obj2>
   ..
   <Container1>
     <Obj1>
     </Obj1>
     <Obj2>
     </Obj2>
     ..
   </Container1>
   <Container2>
     <Obj1>
     </Obj1>
     <Obj2>
     </Obj2>
     ..
   </Container2>
   ..
<nirva>
```

The global XML element is mandatory but can have any name (here “nirva”).

The variable elements contain session variables.

Other elements are objects or subcontainers. The distinction is made by an attribute named “type”. The type attribute can take the values “container”, “boolean”, “integer”, “string”, “stringlist”, “indstringlist”, “table”, “file” or “binary”.

Reference

Main element

Description

This is the root XLM element of each NIRVA XML data. All the rest of XML data flow is encapsulated in this element. The element name is always “nirva”.

Following the connector used and the parameter NV_XML_SESSION_INFO parameter value, the nirva main element main contains some attributes or not.

Attributes

session	NIRVA session ID.
application	name of the connected NIRVA application.
user	User name of the application.
source	This attribute can take the values “BROWSER”, “CLIENT”, “PROCEDURE” or “SERVICE” following the origin of the command which has generated the XML data flow.
host	Server name or address. This attribute gives the name of the server as received in the URL.
local	This attribute is set to “YES” if the host is “127.0.0.1” or “locahost” and to “NO” otherwise.

Session variables

Description

The session variables are given in the variable elements that are immediate child of the main nirva element.

Following the connector used and the parameter NV_XML_VARIABLES parameter value, the session variables may be displayed or not.

Attributes

key	Variable name.

Container

Description

A container is an element containg other object or container elements. The container name is the element name (always lowercase).

A container can contain objects or other containers.

Attributes

None

Boolean object

Description

A boolean object has the following structure:

```
<objname>Value</objname>
```

Where objname is the object name and Value can be “true”, “false”.

Attributes

None

Integer object

Description

An integer object has the following structure:

```
<objname>Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

None

String object

Description

A string object has the following structure:

```
<objname>Value</objname>
```

Where objname is the object name and Value is the object value.

Attributes

None

StringList object

Description

A stringlist object has the following structure:

```
<objname>
  <data>Value1</data>
  <data>Value2</data>
  …
  <data>Valuen</data>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the string list.

Attributes

None

IndStringList object

Description

An indstringlist object has the following structure:

```
<objname>
  <key1>Value1</key1>
  <key2>Value2</key2>
  …
  <keyn>Valuen</keyn>
</objname>
```

Where objname is the object name. Each data child element corresponds to one entry of the indexed string list. The tag name is directly the key value (be careful to use keys that can be accepted as xml tag names).

Attributes

None

Table object

Description

A table object has the following structure:

```
<objname>
  <description>Table description</description>
  <primary case-sensitive="no">Table primary column</primary>
  <size>Table size</size>
  <row>
    <col1>Value1</col1>


    <col2>Value2</col2>
     …
    <coln>Valuen</coln>
  </row>
  <row>
  </row>
  …
  <row>
  </row>
</objname>
```

Where objname is the object name.

The description element gives the table description.

The primary element gives the table primary column.

The size element gives the number of rows in the table.

Then, the table data follows. Each row is encapsulated in a row element. Into each row element, the column name is directly the tag name (be careful to use only authorized XML characters for column names).

Attributes

None

File object

Description

A file object has the following structure:

```
<objname>
  <name>File name</name>
  <extension>File extension</extension>
```

```
  <directory>File directory</directory>
  <pathname>File complete path name</pathname>
  <size>File size</size>
  <origin>File origin name</origin>
  <data>File data</data>
< /objname>
```

Where objname is the object name, File name is the file name, File extension is the file extension (without the point character), file directory is the file directory, File complete path name is the complete file pathname, File size is the file size in bytes and File data is the file data base64 encoded.

Attributes

None

Binary object

Description

A file object has the following structure:

```
<objname>
  <size>Data size</size>
  <data>Binary data</data>
< /objname>
```

Where objname is the object name and Binary data is the binary data base64 encoded.

Attributes

None

Error

Main structure

```
<nvfault>
  <code></code>
  <service></service>
  <class></class>
  <description></description>
  <info></info>
</nvfault>
```

The global XML element is named “nvfault” and contains the error information

Reference

---

nvfault

Description

This is the root XLM element of the NIRVA fault message. All the rest of XML error data flow is encapsulated in this element.

Attributes

session	NIRVA session ID. If there is no session ID, this attribute is not given.
application	name of the connected NIRVA application. If there is no session ID, this attribute is not given.
user	User name of the application. If there is no session ID, this attribute is not given.
Language	Language of the error information.

---

code

Description

Error code.

Attributes

None.

---

service

Description

Error service. Name of the NIRVA service producing the error.

Attributes

None.

class

Description

Error class.

Attributes

None.

description

Description

Error description in the language of the session.

Attributes

None.

info

Description

Error extra information.

Attributes

None.

Security model

Overview

NIRVA provide internally a RBAC (Role Based Access Control) security model. The security defines users, roles and permissions. The roles are hierarchical so given role can inherits the permissions of other roles.

The security is application based. Each application has its own security (its own users, roles and permissions). However, an application can decide to use the security of another application or the security defined at system level. When using the security of another application, this one can be on another server. This allows defining a single centralized security in load balancing environments.

Nirva security can be replaced by a specific security system implemented as a special Nirva service (ex LDAP), providing user authentication, password change and delivering some user permissions.

When a new session is created, it’s always done in the context of an application so there is a necessary login step on the security defined for the application. This login step can be hidden by using the NIRVA default user named “nvdef”. In fact, when no user name is given when creating the session, NIRVA automatically assumes the default user “nvdef”.

NIRVA defines a super user named “nvadmin” that has all rights and cannot be removed. Its initial password is “nirva”. It can be changed. The nvadmin user is always checked locally and cannot be defined on an external security system.

User authentication can be made by Nirva (or external security service if one has been associated with an application) or by Single Sign-On (SSO). SSO works only if both client and server are on a domain and if they are using Kerberos.

The “nvadmin” user cannot be authenticated via SSO.

NIRVA provides the necessary user interface to manage security at system or application level. Please see the configuration chapter in this documentation.

The Nirva security provides also the following functionality:

- User locking (disabled) after a given number of bad logons
- Procedure for validating password syntax
- Removal of unconnected users after a given number of days
- Password expiration management
- Force changing password at first or next logon

Outside the security system, Nirva also provides a way to limit access to sessions from clients. When this feature is enabled, a session can only be accessed by the client who has created it. Nirva uses the TCP/IP addresses of the sender and the session creator for controlling that. This feature is defined at application level (can be different for different applications) and is controlled by a dedicated permission in the security. That concerns only commands sent from clients (web or connectors) so any command sent from a procedure or service is not controlled. Also a local client (127.0.0.1) can access to any session.

Permissions

Permissions are defined by NIRVA itself (system permissions), by the application builder (application permissions), by the service builder (service permissions) or by the web service builder (web service permissions).

This documentation gives the list of system permissions (in the system service reference). For application or service permissions, please consult the corresponding documentations.

The web service permissions are directly the name of the operations defined for the web service.

For service, application and system permissions, the permissions are defined in the description files found in the “Files” directory of system, applications or services. They are simple pairs giving the permission name and the permission description.

In a dsc file, the PERMISSIONS section enumerates the service security permissions.

There is a single PERMISSIONS section in the description file.

The section is composed of several entries of the form *permissionname* = *permissiondescription*.

Here is an example of a PERMISSIONS section extracted from the NIRVA STORAGE service description file:

```
// PERMISSIONS section
// This section enumerates the STORAGE security roles on the form rolename =
roledescription
// If the security roles are not used by the service, this section can be removed or let
empty

[PERMISSIONS]
VOLUME_ADMIN = Administrate volumes
DOCUMENT_READ = Read documents
DOCUMENT_WRITE = Write documents
DOCUMENT_REMOVE = Remove documents
```

The permission name is case insensitive and should not contain space nor special characters.

It’s the responsibility of the application or service provider to define and check its permissions.

When a user creates a new session, NIRVA gets its permission list and keeps this list in memory for the duration of the session so any change to the user permissions is taken in care at the next user login.

Roles

A role is a named object containing a list of permissions.

The role name is case insensitive and should not contain space nor special characters.

A role can inherit permissions from other roles (hierarchical model). NIRVA takes care of the re-entrant roles (role1 inherits role2 that itself inherits role1) by stopping hierarchy when a re-entrant role is detected.

Users

A user is a named object having an optional password and containing a list of roles.

The user name is case insensitive and should not contain space nor special characters.

The user name is checked when creating a new session.

When a user creates a new session, NIRVA gets its permission list and keeps this list in memory for the duration of the session so any change to the user permissions is taken in care at the next user login.

Checking a permission

For checking a permission, the application, service or client programmer must use the SYSTEM:SECURITY:CHECK command described in the SYSTEM service reference in this documentation.

Security service

In place of Nirva security, one can use an external security system defined in a specific security service.

Overview

The security service is a standard Nirva external service that must implemented some well defined commands:

- User login (mandatory)
- User change password

- User get full name
- User get permissions
- User load and save context

Implementation

This chapter describes the command that must be implemented in a security service. All these commands must be implemented in class named “SECURITY”.

LOGIN

Description

This command is called when a user creates a new session to a Nirva application.

The command should fail if the login is not successful with one of the following error codes for the error class named SECURITY:

101 - Bad user. The user is unknown.

102 - Cannot get context (for example if a security LDAP service cannot connect to the LDAP server).

103 - Bad password. The password is not correct.

104 - Bad application. The application is unknown (if the service implements security based on applications)

110 - Bad password syntax (when the syntax of a new password is incorrect)

121 - Change password (when the user must change its password)

Any other error code will produce a SYSTEM:SESSION:103 (cannot login) error code.

The command may create 2 kinds of information:

- A string object named USER_FULL_NAME in the nvdef container. This is the user full name if there is one. This is not mandatory.
- A list of permissions for the user. This is done by using the SYSTEM:SECURITY:ADD_SESSION_PERMISSION command form the service implementation. All the permissions added in this way will be available to the user session. The permissions must correspond to existing system, application, service or web service permissions. This is not mandatory.

Parameters

APPLICATION	Nirva Application name.
USER	User name.
PASSWORD	User password. May be empty.

NEW_PASSWORD	Optional new password given when the user wants to change its password at login time.
HAVE_NEW_PASSWORD	Set to “YES” when the NEW_PASSWORD information is valid and to “NO” otherwise. If yes, the service must proceed to the password change with the new password given in NEW_PASSWORD and the old password given in PASSWORD.
CONTROL_PASSWORD	Set to “YES” when the password has to be checked and to “NO” otherwise. In some situations Nirva will just require to check the user and to get its permissions but without controlling the password. This is the case for example when running a procedure from the scheduler or when the user has been identified via Single Sign-On (SSO).

CHANGE_PASSWORD

Description

This command is called when a user changes its password after a login.

The command should fail if the password change is not successful with one of the following error codes for the error class named SECURITY:

102 - Cannot get context (for example if a security LDAP service cannot connect to the LDAP server).

103 - Bad password. The password is not correct.

110 - Bad password syntax (when the syntax of a new password is incorrect)

Any other error code is even accepted.

Parameters

APPLICATION	Nirva Application name.
USER	User name.
PASSWORD	User old password.
NEW_PASSWORD	New password.

LOAD_USER_CONTEXT

Description

This command is called when the SYSTEM:SECURITY:LOAD_USER_CONTEXT command is sent.

It should retrieve the user specific context previously saved with the SAVE_USER_CONTEXT command.

The command should write the entire user context into the output container. When returning to Nirva, all the content of the output container will be considered as the user context.

The storing and loading of the user context and the format for storing is under the responsibility of the service. A simple possibility is to use the CONTAINER:EXPORT and CONTAINER:IMPORT commands to store and retrieve the context as a file.

The command should fail if cannot load user context with one of the following error codes for the error class named SECURITY:

102 - Cannot get context (for example if a security LDAP service cannot connect to the LDAP server).

123 – Cannot load user context.

Any other error code is even accepted.

Parameters

APPLICATION	Nirva Application name.
USER	User name.

SAVE_USER_CONTEXT

Description

This command is called when the SYSTEM:SECURITY:SAVE_USER_CONTEXT command is sent.

It should store the user specific context.

The user context is given as the entire content of the input container.

The storing and loading of the user context and the format for storing is under the responsibility of the service. A simple possibility is to use the CONTAINER:EXPORT and CONTAINER:IMPORT commands to store and retrieve the context as a file.

The command should fail if cannot save user context with one of the following error codes for the error class named SECURITY:

102 - Cannot get context (for example if a security LDAP service cannot connect to the LDAP server).

124 – Cannot save user context.

Any other error code is even accepted.

Parameters

APPLICATION	Nirva Application name.
USER	User name.

Single Sign-On (SSO)

This chapter describes how to use the Single Sign-On (SSO) features of Nirva. It addresses specialists of computer security. It implies that the security configuration of your organization is defined properly and that involved components are correctly set up. Any support issue regarding SSO will be accepted from security specialists only and after proving that the problem doesn’t come from the environment

Overview

Single sign-on (SSO) is mechanism whereby a single action of user authentication and authorization can allow a user to access all computers and systems according to access permission, without the need to enter multiple passwords. Single sign-on reduces human error, a major component of systems failure and is therefore highly desirable.

Nirva supports Single Sign-On thereby avoiding already identified users of a domain to re-enter credentials to access Nirva applications.

SSO is currently available on Windows platforms only (client and server) for the following clients:

Web browser (IE and Firefox)

Nirva client connectors using the nvc.dll library (Java, command line, C, C++, Perl, PHP, .Net, Cold Fusion, ActiveX, Virtual Printer).

Web services or XML HTTP clients implementing the SSO standard protocols “Negotiate”, “Kerberos” or “NTLM”. The Negotiate protocol is a Microsoft specific protocol allowing to automatically choosing between Kerberos or NTLM following the configuration and availability of security components in the target environment.

When an access to Nirva requires the client to sign on, Nirva sends some special information to the client over HTTP, telling it that an authentication is required and that the “Negotiate”, “Kerberos” and “NTLM” protocols are accepted. According to available capabilities, the client will choose one of these protocols. The Nirva Windows clients using the nvc library will use the “Negotiate” protocol. The client browsers (IE and Firefox) also use the “Negotiate” protocol.

In this document whereas “Kerberos” is mentioned, one must understand “Kerberos” or “Kerberos over Negotiate”. When “NTLM” is mentioned, one must understand “NTLM over Negotiate”.

SSO involves three parts:

The domain controller. This is a Microsoft Active Directory server on this example. This domain controller also acts as a Kerberos server when Kerberos is to be used.

The Nirva server is the computer running the Nirva application to be used with SSO.

The client computer is the client connecting to the Nirva application using SSO.

This example uses the following information:

- Domain controller = NIRVADC

- Domain = nirvalyon.local
- Nirva server = BENJ
- Nirva service account = ssotest
- Nirva application = SSOTEST

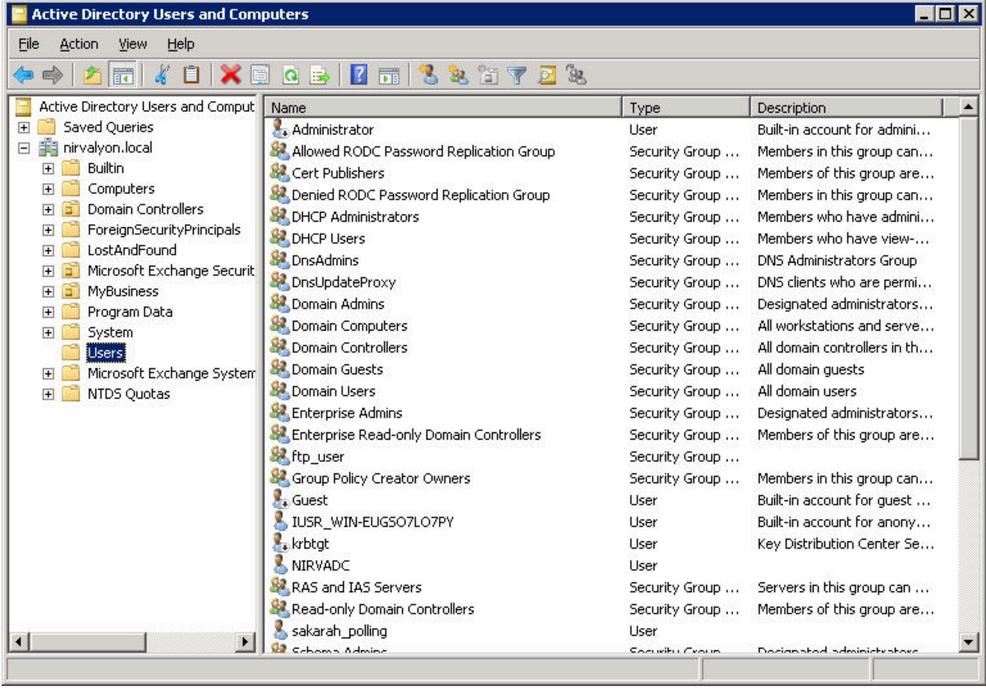
Configuring the domain controller

Creating a special user account

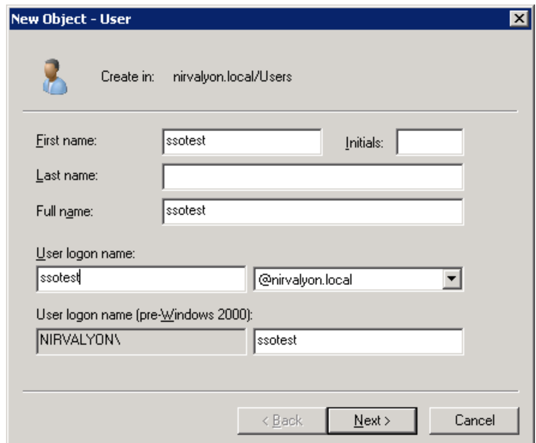
This part is required only when “Kerberos” protocol is to be used.

For each Nirva server that needs to be SSO (Kerberos) enabled, a special user account must be created. This user account is called a Service Principal Name account (SPN account). This user acts as the connection between the Kerberos server, the Active Directory and the Nirva server. This user account will be used to run the Nirva service on the Nirva server.

Start the Active Directory User and Computers from the Administration tools menu.



Right click the Users folder and select new - user



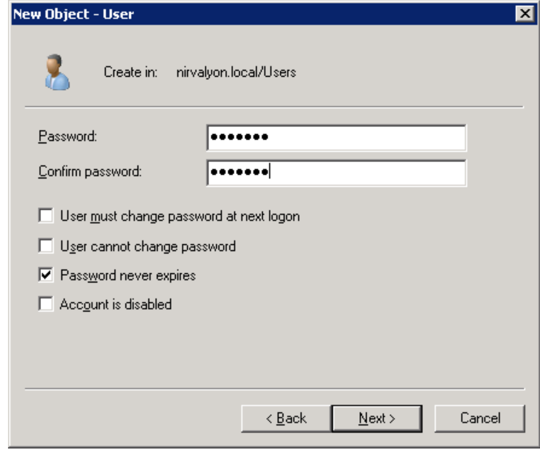
The user name should be a specific user name in the domain that will be used for starting the Nirva service. A computer or user with the same name must not exist on the domain.

Here the domain is nirvalyon.local and the user is ssotest.

Enter the name into the User logon name

Press next.

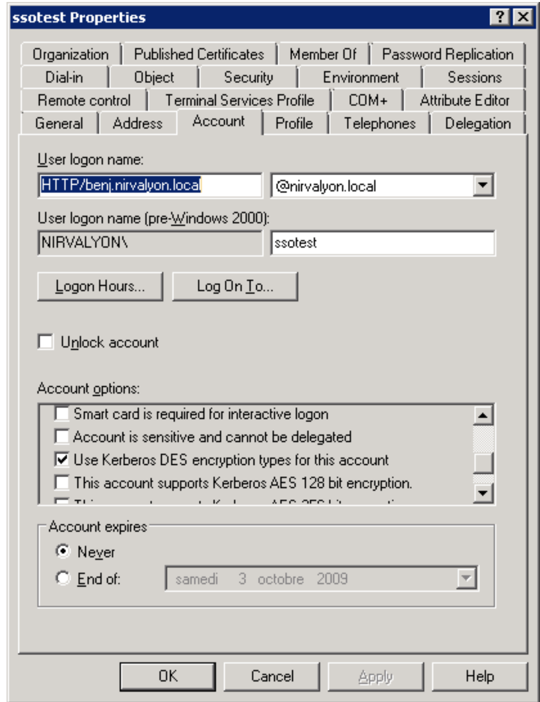
Enter the password and select Password never expires.



Click Next and then Finish to confirm the new user account

Now we need to modify the user account in order to specify the encryption algorithm.

Open the newly created user account. Select the account tab.



Scroll down to the bottom of the Accounts options section.

Select the Use DES encryption types for this account.

Note: When changing the ‘Use DES encryption…’ checkbox on a user account, the password MUST be reset after the change has been done.

To reset the password, right click on the user account and select the option Reset Password, re-enter the password and press the OK button.

Setting principal names

This part is only required if the “Kerberos” protocol is to be used.

Now that the user account has been created and updated, we need to create a service principal setting for the created user account.

The service principal name is a unique key identifying the SSO target

It has the following format: `Service/Server`

The service for Nirva clients using nvc library can be set to “NIRVA”.

The service for browsers connecting Nirva must be set to “HTTP”. For example if, from your web browser, you want to access via SSO to a Nirva resource located on the url “`http://benj:1081/....`” Then you must set the SPN to “`HTTP/benj`”.

The server must be set to the DNS name of the target server (Nirva server). This DNS name must be unique and the reverse lookup DNS table must also have a single entry for the server.

So in our example, enter the following commands:

```
setspn -A HTTP/benj nirvalyon.local\ssotest
setspn -A NIRVA/benj nirvalyon.local\ssotest
```

A given SPN must be associated with only one account otherwise the “Kerberos” protocol will fail. More generally, when using the “Negotiate” protocol, if a valid SPN corresponding to the requested resource exists, windows will use “Kerberos”. If not, windows may use “NTLM”.

Note: You can list the SPNs associated to a given account with the “setspn –L account“ command. For example:

```
setspn -L nirvalyon.local\ssotest.
```

For urls using a DNS name instead of a direct machine name, the same DNS name must be registered as SPN with the HTTP service. For example, if you have a url “`http://mysiteweb.com/`” and you have defined that “`mysiteweb.com`” points to your nirva server (benj) in the DNS, then the “`HTTP/mysiteweb.com`” must be registered as SPN via setspn:

```
setspn -A HTTP/mysiteweb.com nirvalyon.local\ssotest
```

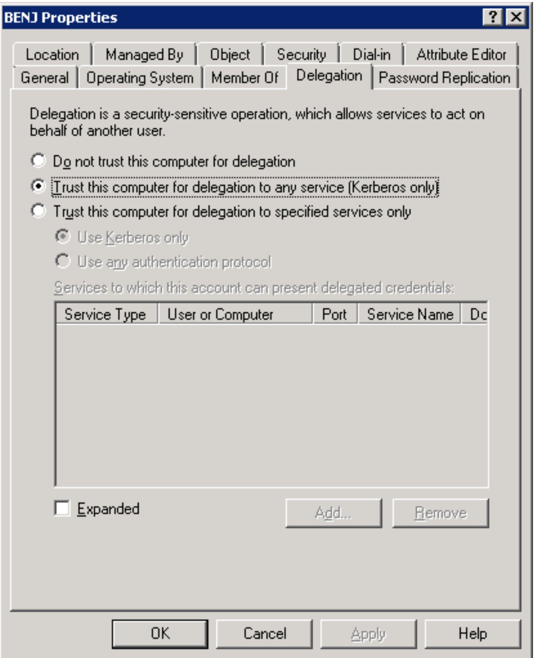
The port is not used by the browser to evaluate the principal name so if the url is “`http://benj:1081/`” the SPN will be “`HTTP/benj`”.

Allowing delegation

This part is only required if the “Kerberos” protocol is to be used.

If the nirva server is not the domain controller itself, make sure to set the nirva server machine as Trust computer for delegation:

Go to the Administrative Tools -> Active Directory Users and Computers. Expand the domain and click on Computers. Locate the nirva server computer and right click on “Properties”, then open the “Delegation” tab:



Enable the “Trust this computer for delegation to any service (Kerberos only)” check box.

Configuring the Nirva server

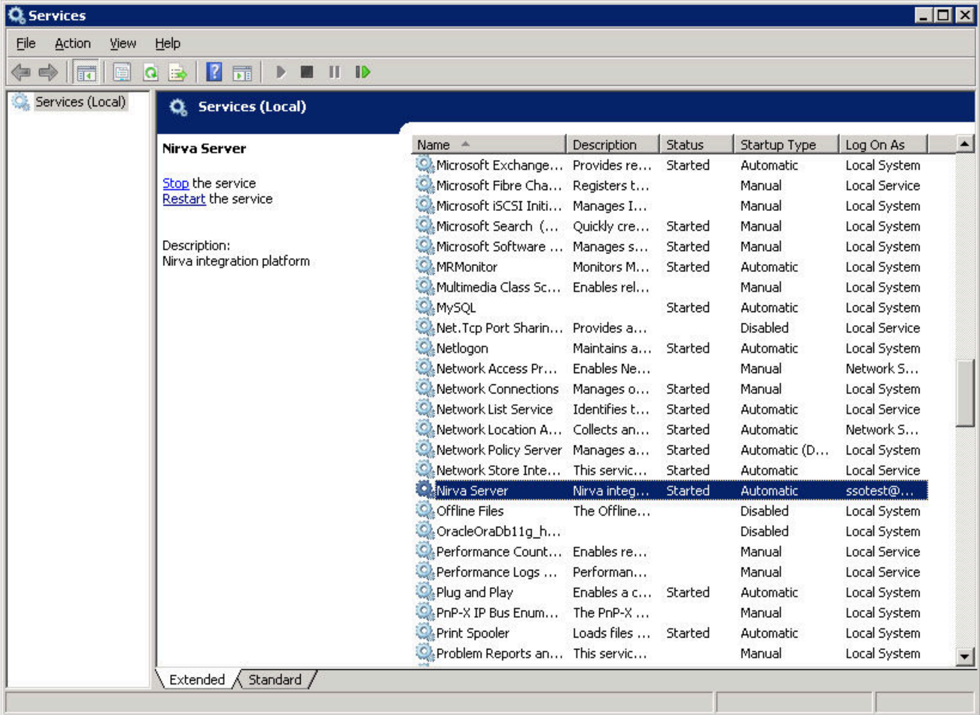
Running Nirva as the special user account.

This part is only required if the “Kerberos” protocol is to be used.

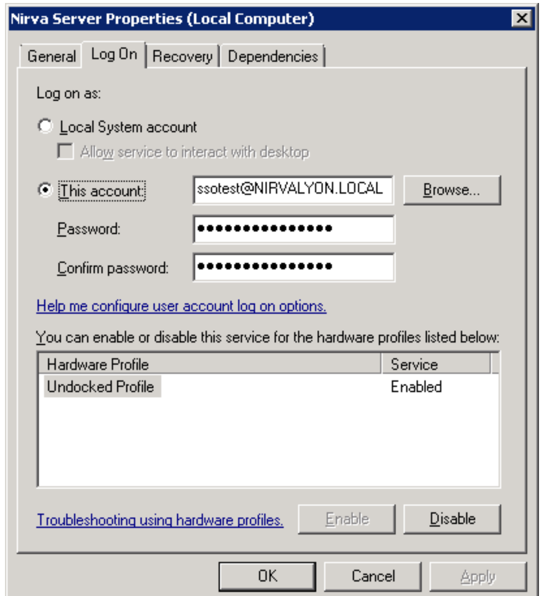
Nirva must be running in the context of the special user (ssotest) defined on the domain controller in previous step.

If you run Nirva in console mode, just start Nirva from this special user account session (ssotest).

If you run Nirva as a service, go to the control panel and open the “Services” window:



Then right click on the “Nirva server” entry and select “Properties” and the “Log On” tab:

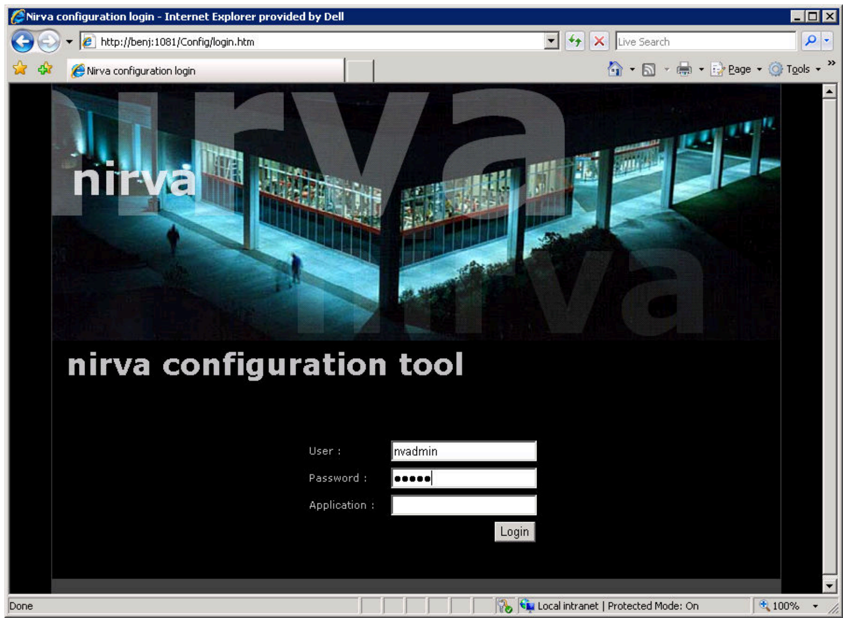


Enable the “This account” check box and enter the name of the special user you have defined in the domain control (fully qualified user name). Enter the password and click OK. Then you can start the service.

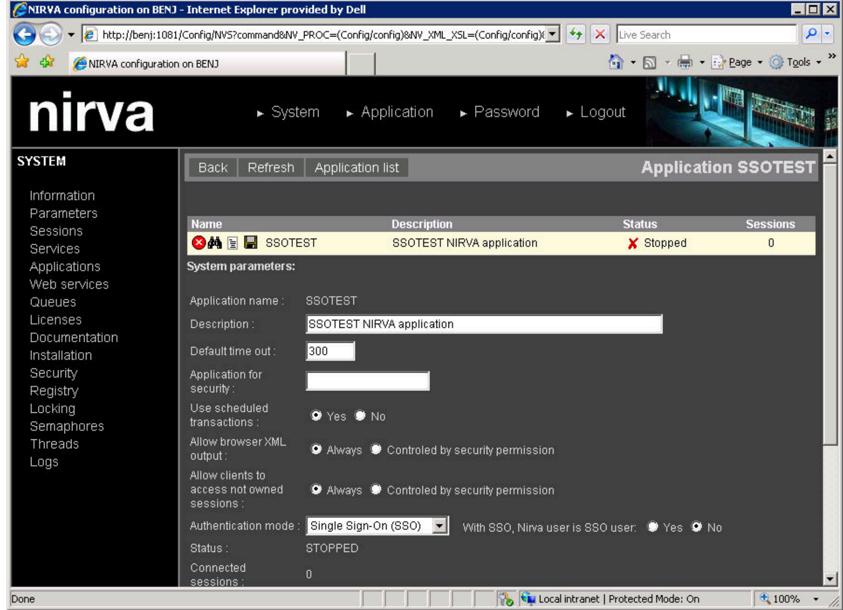
Configuring Nirva

On Nirva, you must enable SSO for your Nirva application(s).

For that, open the Nirva administration tool with nvadmin account:



Then go the main System menu, select the “Applications” item in order to display the application list, then click on your application name (here SSOTEST) and stop the application if it was running:



In order to enable SSO, you must modify the “Authentication mode” line. You have 3 modes:

- **Nirva**. SSO is disabled.
- **Single Sign-On (SSO)**. SSO is enabled and mandatory. This is the only mode allowing SSO from browsers.
- **Nirva or Single Sign-On**. SSO is enabled only on client request. In this mode, Nirva clients using the nvc library must give a parameter in order to enable SSO on the session. This is a parameter in the connection string (see the documentation for the Nirva connectors).

When using SSO, you can choose to have the SSO user equal to the Nirva user or not. If not, the SSO is only used for authentication and the security is based on the Nirva user. The password for the Nirva user is

not checked and can be omitted. If yes, the Nirva user becomes the SSO user and this SSO user must exists in the table of Nirva users. The nvadmin user never uses SSO.

In any case, the SSO user and domain is available in the session context as session variables named “NV_SSO_USER” and “NV_SSO_DOMAIN”.

In our example, we choose “Single Sign-On (SSO) mode” and the Nirva user is not the SSO user.

Then start the Nirva application.

Note: For the NVDEF application, you must go into the Systems parameters in order to enable SSO and you must stop and restart Nirva.

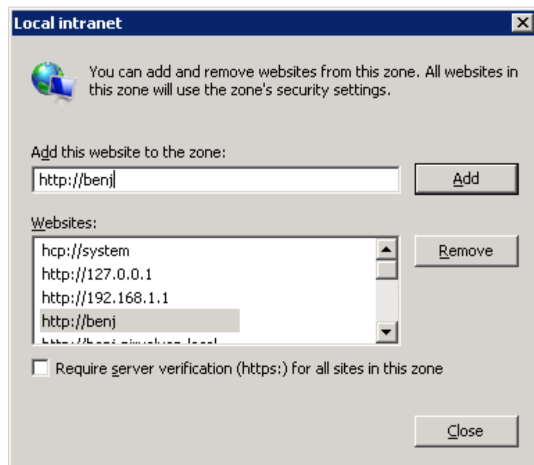
Configuring Clients

Browsers

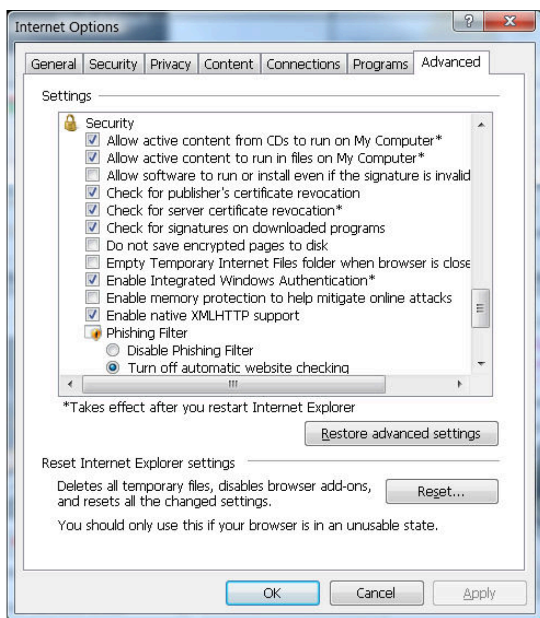
Following your browser, you may have to do some specific configuration.

With internet explorer for example, you should add the website to the list of trusted web site on your local intranet. For that, go to the Tools menu, Internet options, Security. Click on local intranet and then “Sites” button:

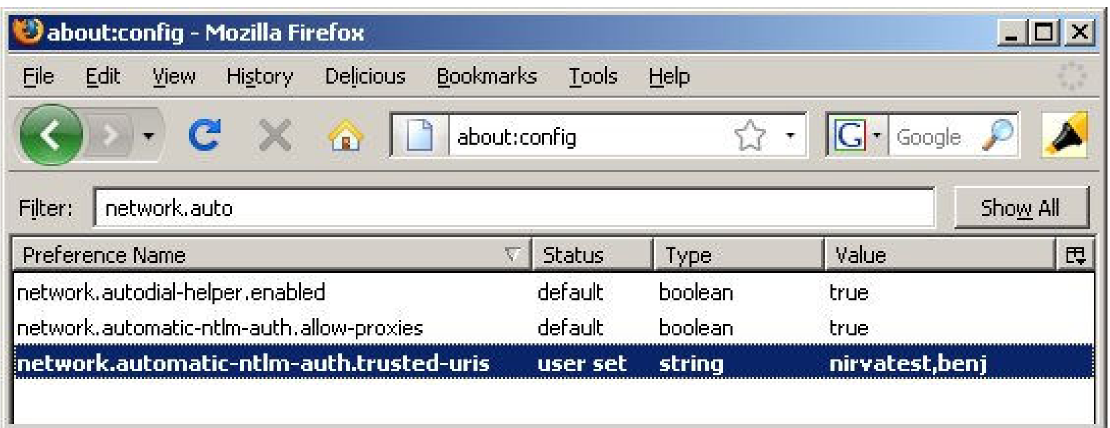
Click on the site button and add the website (you may have an intermediate window for local intranet options):



You must also be sure that in the internet options advanced tab, the line “Enable Integrated Windows Authentication” is checked:



With Firefox, type “about:config” in the navigation bar and then search for “network.automatic-ntlm-auth.trusted-uris”. Add the site server to it (separate from others with a comma character)



Nirva clients

Nirva clients have 2 options for SSO:

The first option is the flag for enabling SSO when the application mode has been set to “Nirva or Single Sign-On”. This is generally part of the connection string as “Sso=YES” or “Sso=NO”. For the nvcc (command line connector) this is a string “(SSO)” in the –a option after the server address.

The second option is the SSO principal name when using Kerberos. This is generally part of the connection string as “SsoPrincipal=SPN”. The SPN must be an existing SPN as defined on the domain controller using the setspn command (see chapter “Configuring the domain controller”). For the nvcc (command line connector) this is a string “(SSO:SPN)” in the –a option after the server address.

Examples with nvcc (when Nirva application SSOTEST is in mode “Nirva or Single Sign-On”):

SSO using Kerberos (the SPN “NIRVA/benj” must have been defined in the domain controller otherwise NTLM will be automatically used):

```
nvcc -p SSOTEST -a benj(SSO:NIRVA/benj) -z "NV_CMD=|MISC:NOP|"
```

SSO using NTLM:

```
nvcc -p SSOTEST -a benj(SSO) -z "NV_CMD=|MISC:NOP|"
```

No SSO:

```
nvcc -p SSOTEST -a benj -z "NV_CMD=|MISC:NOP|"
```

These examples use the Nirva default nvdef user.

In order to verify if SSO works, you can define a password to the nvdef user on the SSOTEST application. In this case the last command should return an error “invalid password” but the other ones should succeed. Any other error code is even accepted.

Installation packages

Overview

NIRVA provides some commands for installing files on the NIRVA server. This feature is especially used for external service and application installation.

For security reasons, NIRVA only accepts to install files in the NIRVA directory and subdirectories. It's not possible to install files outside the NIRVA directory with the NIRVA installation features.

In fact, NIRVA defines package files. A package file is a compressed file that contains the files to install and the relative path of the files to install.

When installing a package, NIRVA uses a base path from which the content of the package will be installed. This base path depends of the command used to install the package. Here are the 3 possible base paths:

- NIRVA root directory if the SYSTEM PACKAGE INSTALL is used.
- NIRVA target application directory if the SYSTEM APPLICATION INSTALL command is used.
- NIRVA target service directory if the SYSTEM SERVICE INSTALL command is used.

The package file itself is built following the information given in a package description file. This package description file is then given as parameter of one of the commands to create packages (system, service or application packages).

The package description file

Here is an example of package description file:

```
// package.lst : installation package listing
// for NIRVA SERVICE PLANET

// Header section
// This section is transmitted as it is to the package file
[HEADER]
SERVICE = PLANET
```

```
// All files and subdirectories of SERVICE Bin directory
[/Bin]
copydirsub:

// Description file
[/Files]
copyfile:service.dsc

// All files and subdirectories of SERVICE Files/Config directory
[/Files/Config]
copydirsub:

// Removes SERVICE Docs/Html directory
[/Docs/Html]
removedirsub:

// All files and subdirectories of SERVICE Docs directory
[/Docs]
copydirsub:

// All files and subdirectories of SERVICE Procs directory
[/Procs]
copydirsub:

// All files and subdirectories of SERVICE Wroot directory
[/Wroot]
copydirsub:
```

This is the default package description file generated by the SYSTEM SERVICE SKELETON command for a service named PLANET.

The first part of the package description file is the HEADER part. It contains several lines on the form *EntryName* = *EntryValue*. The header part is transmitted as it is to the package file. The SYSTEM PACKAGE INFO command reports the content of the header part of a package file. The header is generally used to set information about the package.

4 entries of the header have special meaning:

- If there is an entry *SERVICE=SrvName* where *SrvName* is the name of a service, the package will be considered as a service package.
- If there is an entry *APPLICATION=AppName* where *AppName* is the name of an application the package will be considered as an application package.
- If there is an entry *PLATFORM* = *Platform* where Platform is the target platform (WIN32, WIN64, AIX, LINUX, LINUX64, HPUX, HPUXI or SOLARIS), the INSTALL commands check if the target platform corresponds to the package file and returns an error if it’s not the case.
- If there is an entry *ENCODING* = *UTF-8* then NIRVA considers that the information written in the HEADER part is UTF-8 encoded. Otherwise NIRVA considers that the information in the HEADER part is coded with the “ISO-8859-1” character set.

The other parts of the package description file are sections corresponding to target directories. The section name is a target directory. When the package is installed, NIRVA creates these directories if they don't exist. They are relative directories that start from the base directory calculated by NIRVA. The directory name can be blank in order to put file in base directory.

For example, if the section name is "`/Test/Myfiles`", and the package is installed with the SYSTEM SERVICE INSTALL command, NIRVA will create the directories "`Test`" and "`Test/Myfiles`" in the target service directory. The directory names are case sensitive under UNIX.

Each section contains a list of lines corresponding to some actions to apply. Each action line has the following format:

```
actiontype:parameters
```

Where `actiontype` is the type of action and parameters are the `parameters` specific to this type of action.

Here are the possible actions:

`copyfile`	inserts a single source file in the package file that will be copied on the target directory at package installation time. The parameter for this action has the format: Targetfile = Sourcefile where `Targetfile` is the name of the file created at installation time and `Sourcefile` is the complete or relative pathname of the file to put into the package at package creation time. When `Sourcefile` is given Nirva first considers that it is a complete path and checks if this file exists. If not Nirva considers a relative path. If `Sourcefile` is starting with the # character, Nirva replaces this # character with the base directory of the component beeing packaged. If the source file name and directory is the same than the destination file name and directory (relative directory), it's not necessary to give the `Sourcefile` information. At this time, the separator '=' must also not be given. This action is the default one.
`copyperlfile`	this action is similar to the `copyfile` action but encrypts the perl file (works only on files having a ".pl" extension). Perl encryption allows hiding the perl code to users. This action can be used instead of the `perlencode` action when only some particular perl files have to be encrypted.
`copydir`	inserts all the files of a source directory in the package file. These file will be copied in the target directory at package installation time. The parameter for this action is the complete path of the source directory containing the files to include in the package file. If the source directory is the same than the target directory, the parameter can be omitted.

`copydirsub`	do the same action than `copydir` but also includes all files of all subdirectories of the source directory. The files will be copied in the target directory at package installation time by keeping the directory structure. The parameter for this action is the complete path of the source directory containing the files and subdirectories to include in the package file. If the source directory is the same than the target directory, the parameter can be omitted.
`excludefile`	excludes a file when copydir or copydirsub is used. If the section contains a copydir, the excludefile action excludes a file from the source directory. If the section contains a copydirsub, the exclude action excludes a file from the source directory and all its subdirectories. The parameter for this action is the name (just the filename, not the complete path) of the file to exclude. It may contain the '*' wildchar as first or last character in order to exclude all file names starting or terminating by the given string.
`excludedir`	excludes a subdirectory directory when copydirsub is used. If the section contains a copydirsub, the excludedir action excludes a subdirectory from the source directory and all its subdirectories. The parameter for this action is the name of the subdirectory to exclude. It may contain the '*' wildchar as first or last character in order to exclude all subdirectory names starting or terminating by the given string.
`removefile`	removes a file on the target directory at package installation time. The parameter for this action is the name (just the filename, not the complete path) of the file to remove. It may contain the '*' wildchar as first or last character in order to remove all files with name starting or terminating by the given string.
`removedir`	removes all the files of the target directory at package installation time. This action has no parameters.
`removedirsub`	do the same action than `removedir` but also removes all subdirectories of the target directory at package installation time. This action has no parameters.
`exec`	sets the file permissions as an executable file. This is meaningful only under UNIX platforms (ignored on windows). The parameter for this action is the name of the file created (target file).
`textfile`	sets the given file as a text file. This action is taken in care at installation time and allows setting or removing the carriage return character following the destination platform of the package. The parameter for this action is the name of the file created (target file).
`textdir`	do the same action than textfile for the complete directory. This action has no parameters.
`perlencode`	this is an option to encode perl files for `copyfile`, `copydir` and `copydirsub` actions. The scope of this option is only for the current section.

Only the files having a ".pl" extension will be encrypted. Perl encryption allows hiding the perl code to users. This action has no parameters.

When creating the package file, NIRVA processes all the sections of the package description file in the order they appear.

For each section, Nirva reorders the action lines by processing them in the following order:

- removedir
- removedirsub
- copyfile
- copyperlfile
- copydir + excludefile + excludedir
- copydirsub + excludefile + excludedir
- exec
- textfile
- textdir

Creating a package

The package creation can be done by using one of the following commands:

- SYSTEM:PACKAGE:PACKAGE for creating a system package.
- SYSTEM:SERVICE:PACKAGE for creating a service package.
- SYSTEM:APPLICATION:PACKAGE for creating an application package.

Please refer to these commands description in the system service reference for further information.

It's also possible to use some command line tools delivered in standard with NIRVA.

```
nvcc –i system_package.txt <Package file> [<Description file>] for creating a system package.

nvcc –i service_package.txt <Service name> <Package file> [<Description file>] for creating a service package.

nvcc –i application_package.txt <Application name> <Package file> [<Description file>] for creating an application package.
```

These command tools are described in the "Tools" chapter of this documentation.

Finally, an application, service or web service package can be created directly from the nirva configuration tool (see configuration chapter in this documentation).

Installing a package

The package installation can be done by using one of the following commands:

- SYSTEM:PACKAGE:INSTALL for installing a system, a service or an application package.
- SYSTEM:SERVICE:INSTALL for installing a service package.
- SYSTEM:APPLICATION:INSTALL for installing an application package.

Please refer to these commands description in the system service reference for further information.

It's also possible to use some command line tools delivered in standard with NIRVA.

```
nvcc –i system_install.txt <Package file> for installing a system, a service or an
application package.

nvcc –i service_install.txt <Service name> <Package file> for installing a service
package.

nvcc –i application_install.txt <Application name> <Package file> for installing an
application package.
```

These command tools are described in the “Tools” chapter of this documentation.

The installation of Nirva packages can be done also from the Nirva configuration tool (see configuration chapter in this documentation).

Test sets

Overview

Test sets are xml files that define some application or service level unit tests. There are 2 types of tests:

- Procedure. The test just executes the defined procedure. This procedure must be part of the application or service procedures.
- Url. Calls the defined Url in a separate browser window. This is relative Url called in the context of the connected application from the Nirva configuration tool.

Application level test sets files must be put in the Files/Test/sets subdirectory of the application directory.

Service level test sets files must be put in the Files/Test/sets subdirectory of the service directory.

In order for a new test set to be taken in care, the corresponding application or service must be stopped and restarted.

Tests can be started from the Nirva configuration tool (see configuration chapter).

The test set file

A single test set file is an xml file that must have a “.xml” extension.

Here is an example of a test set file:

```
<?xml version="1.0"?>
<set>
  <name>My first set</name>
  <description>Test for my first set</description>
  <tests>
          <test>
                  <name>Test 1</name>
                  <description>Simple perl procedure&#x0A;new line</description>
                  <type>PROC</type>
                  <procedure>perl:ptest</procedure
                  <parameters>
```

```
                        <param1>param1 value</param1>
                        <param2>param2 value</param2>
                        <param3>param3 value</param3>
                </parameters>
        </test>
        <test>
                <name>Test 2</name>
                <description>Simple java procedure</description>
                <type>PROC</type>
                <procedure>java:jtest</procedure>
        </test>
        <test>
                <name>Test 3</name>
                <description>Perl procedure from url</description>
                <type>URL</type>
                <url>NVS?command&amp;NV_PROC=perl:ptest</url>
        </test>
  </tests>
</set>
```

The set/name tag is mandatory and cannot be empty.

The set/description tag is mandatory but can be empty

The set/tests/test/name tag is mandatory and must uniquely identify the test.

The set/tests/test/description tag is mandatorybut can be empty.

The set/tests/test/type must be “PROC” for a test of type “procedure” and “URL” for a test of type “url”. It is mandatory.

The set/tests/test/ulr tag is mandatory when the type is “URL” and must contain a relative url. When Nirva calls the URL, it adds the current session ID to the URL making it running in the context of the current session.

The set/tests/test/procedure tag is mandatory when the type is “PROC” and must contain a procedure that is in the application or service directory (Please see the Calling a procedure chapter for description of the procedure syntax). The test procedure is called in the context of the current application and in a container named “TEST” (the NV_CONTAINER parameter is set to “TEST”). If the procedure fails, the test fails. When calling a test procedure, Nirva sets 2 sessions variables: “RESULT” is set to “1” and “REASON” is set to an empty value. If the procedure changes the “RESULT” session variable to something else than “1”, the test fails. The reason displayed in the Nirva config tool is the then the value of the “REASON” session variable.

Here is an example of a test procedure that fails:

```
# Do something
NV::Command("NV_CMD=|OBJECT:CREATE| NAME=|mystring| TYPE=|STRING| VALUE=|myvalue|");

# Make the test fail
NV::Command("NV_CMD=|VARIABLE:SET| NAME=|RESULT| VALUE=|0|");
NV::Command("NV_CMD=|VARIABLE:SET| NAME=|REASON| VALUE=|My reason|");
```

The set/tests/test/parameters tag is not mandatory. It contains some test parameters. Test parameters are used only with procedure type tests. The parameters are transmitted to the test procedure at execution time.

Since test sets are usually delivered in the application or service package, one can define the parameters in an external parameters file. This avoids replacing a test file containing specific parameter values at installation time. An external parameter file must have the same structure than the test file but with only the parameter tag for tests having parameters. If some parameters with the same name are defined in both the test and parameter files, the values in the parameter file has priority. The parameter file associated to a test file must have the same name with the string “_param” addes vefore the extension. For example if the test set file is “test1.xml”, the parameter file will be “test1_param.xml”.

Here is an example of a test parameter file:

```
<?xml version="1.0" encoding="UTF-8"?>
<set>
<tests>
  <test>
         <name>Test 1</name>
         <parameters>
                 <param2>param2 value modified</param2>
                 <param4>param4 value</param4>
         </parameters>
  </test>
  <test>
         <name>Test 2</name>
         <parameters>
                 <paramtest2>param1 test 2</paramtest2>
         </parameters>
  </test>
</tests>
/set>
```

Benchmarks

HTTP server

This benchmark compares the performances of HTTP servers for 100 simultaneous requests.

This is a comparison and not a performance test itself. What is important in this benchmark is the difference between the tested HTTP servers.

Test environment

Machine: DELL inspirion 8100 1.2 GHz with 512 Mbytes memory. W2K.

Test program: Apache jmeter version 1.8.1.

Apache server version 1.3.27.

IIS server version 4.0.2.4426.

NIRVA server version beta1 build 21.

The 3 tested HTTP servers and the test program are running on the same machine.

Test description

The test consists of sending in parallel 100 requests to a single html page containing a form and a gif logo. This test is repeated 100 times so the total number of accesses is 10000 with always 100 concurrent. The html page is exactly the same for all tested servers.

A first test is made by requesting the test program to keep open the network connections between the different loops and a second test is made by telling him to not keep the connections opened.

The test program (jmeter) is configured with a very fast ramp up period. This is a school case but this shows the capacity of the server to accept a big amount of simultaneous socket connections (100 in the exact same time in our case). This is a common difficulty on an HTTP server to manage the queue of input connections because this queue is often kernel limited so the server must answer to a connected request in a very fast way. Practically, the test will deliver an error rate (0% means no error and 100% means all in error) that gives information about the capacity of the server to manage the queue of incoming connection requests. The more the rate is low, the more the server has capabilities to accept simultaneous network connection requests.

Test results

With keep connections alive

	Average (ms)	Min (ms)	Max (ms)	Error (%)	Rate (requests/s)
Apache	601	0	54979	0,16	126.5
IIS	1537	0	3916	90.77	62.1
NIRVA	719	0	2734	0,31	137.0

Without keep connections alive

	Average (ms)	Min (ms)	Max (ms)	Error (%)	Rate (requests/s)
Apache	1565	20	6078	14.42	62.0
IIS	1802	0	2604	99.47	54.3
NIRVA	1671	0	3525	0.59	58.9

Comments

The IIS is nearly 2 times slower in the keep connection alive test and is particularly bad in the management of simultaneous network connection requests (important error rate).

NIRVA and Apache are similar in terms of performance and for the error rate (negligible for both) in the keep connections alive test, but NIRVA seems to be more regular because the maximum time for a request is 2.7 seconds on NIRVA and 55 seconds for apache. This is true also in the test without alive connections but the difference is then smaller. In this second test NIRVA is also better for the error rate.

Using nvcc

The NIRVA tool nvcc can be use to make performance measurements.

For that, nvcc must be launched in test mode by using the “-b” option. This option allows defining the number of threads and the number of loops.

In test mode, nvcc processes in the following way:

- Create the requested number of threads.
- For each created thread, establishes a connection to the NIRVA server by sending a SYSTEM MISC NOP command.
- Waits for all the threads to be connected to the server.

- Sends an event to all the threads for them to start processing. In this way, all the threads start processing simultaneously. The thread processing consists of executing the input file in loop mode with the number of loops given by the "–b" option.
- Waits for all threads to end their processing.
- Disconnects the threads from the NIRVA server.
- Waits for all threads to be disconnected.
- Reports the time of real processing (without the ramp up and ramp down periods).

The reporting gives the total number of requests, the total time and the number of requests per second. A request is an execution of the script (the input file). The number of requests per second is different than the number of commands per second because a single script may contain an important number of commands.

In this way, the processing time for a given operation can be easily measured.

Here is an example of nvcc test mode reporting:

```
Cmd
C:\nirva\Bin>nvcc -i test.txt -b l=100:r=2



Connecting sessions...
Waiting for all sessions to be connected...
Start processing commands...
Waiting for end processing...
Processing has finished...
Disconnecting sessions...
Waiting for all sessions to be disconnected...

TEST RESULTS

Number of threads = 2
Number of loops = 100
Total number of requests = 200
Processing time = 430 ms
Rate = 465 requests / second

Press enter to continue

C:\nirva\Bin>
```

This example executes 2 threads in parallel, each of them running 100 times the script test.txt.

Proprietary rights notice

All brands or product names cited in this documentation are trademarks or registered trademarks of their respective companies or organizations.

RSA Security

This product includes code licensed from RSA Security, Inc.

Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

Java Runtime Environment

CONTAINS IBM(R) Developer Kit for Linux(R), Java(TM) 2 Technology Edition, Version 1.3.1, 32-bit version for POWER Runtime Modules

(c) Copyright IBM Corporation 1997-2002

All Rights Reserved

CONTAINS IBM(R) 32-bit Runtime Environment for AIX(TM), Java(TM) 2 Technology Edition, Version 1.4 Modules

(c) Copyright IBM Corporation 1999, 2002

All Rights Reserved

CONTAINS IBM Runtime Environment for AIX(R), Java(TM) 2 Technology

Edition Runtime Modules

(c) Copyright IBM Corporation 1999, 2000

All Rights Reserved

OpenSSL

NIRVA utilize the "OpenSSL toolkit" functionality provided by "The Open SSL Project" at http://www.openssl.org. SDI Limited acknowledges all patent rights therein."

The OpenSSL toolkit is licensed under a dual-license (the OpenSSL license and the original SSLeay license). See the license text below.

OpenSSL License

Copyright (c) 1998-2000 The OpenSSL Project. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org)
4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org
5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.
6. Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

SSLeay license

Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. This package is an SSL implementation written by Eric Young (eay@cryptsoft.com). The implementation was written so as to conform with Netscapes SSL. This library is free for commercial and non-commercial use as long as the following conditions are aheared to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson

(tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
All advertising materials mentioning features or use of this software must display the following acknowledgement: "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)". The word 'cryptographic' can be left out if the rouines from the library being used are not cryptographic related :-).
If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement: "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.The licence and distribution terms for any publically available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution licence [including the GNU Public Licence.

The "Artistic License"

Preamble

The intent of this document is to state the conditions under which a
Package may be copied, such that the Copyright Holder maintains some
semblance of artistic control over the development of the package,
while giving the users of the package the right to use and distribute
the Package in a more-or-less customary fashion, plus the right to make
reasonable modifications.

Definitions:

"Package" refers to the collection of files distributed by the Copyright Holder, and derivatives of that collection of files created through textual modification.

"Standard Version" refers to such a Package if it has not been modified, or has been modified in accordance with the wishes of the Copyright Holder as specified below.

"Copyright Holder" is whoever is named in the copyright or copyrights for the package.

"You" is you, if you're thinking about copying or distributing this Package.

"Reasonable copying fee" is whatever you can justify on the basis of media cost, duplication charges, time of people involved, and so on. (You will not be required to justify it to the Copyright Holder, but only to the computing community at large as a market that must bear the fee.)

"Freely Available" means that no fee is charged for the item itself, though there may be fees involved in handling the item. It also means that recipients of the item may redistribute it under the same conditions they received it.

1. You may make and give away verbatim copies of the source form of the Standard Version of this Package without restriction, provided that you duplicate all of the original copyright notices and associated disclaimers.

2. You may apply bug fixes, portability fixes and other modifications derived from the Public Domain or from the Copyright Holder. A Package modified in such a way shall still be considered the Standard Version.

3. You may otherwise modify your copy of this Package in any way, provided

that you insert a prominent notice in each changed file stating how and when you changed that file, and provided that you do at least ONE of the following:

a) place your modifications in the Public Domain or otherwise make them Freely Available, such as by posting said modifications to Usenet or an equivalent medium, or placing the modifications on a major archive site such as uunet.uu.net, or by allowing the Copyright Holder to include your modifications in the Standard Version of the Package.

b) use the modified Package only within your corporation or organization.

c) rename any non-standard executables so the names do not conflict with standard executables, which must also be provided, and provide a separate manual page for each non-standard executable that clearly documents how it differs from the Standard Version.

d) make other distribution arrangements with the Copyright Holder.

4. You may distribute the programs of this Package in object code or executable form, provided that you do at least ONE of the following:

a) distribute a Standard Version of the executables and library files, together with instructions (in the manual page or equivalent) on where to get the Standard Version.

b) accompany the distribution with the machine-readable source of the Package with your modifications.

c) give non-standard executables non-standard names, and clearly document the differences in manual pages (or equivalent), together with instructions on where to get the Standard Version.

d) make other distribution arrangements with the Copyright Holder.

5. You may charge a reasonable copying fee for any distribution of this

Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own. You may embed this Package's interpreter within an executable of yours (by linking); this shall be construed as a mere form of aggregation, provided that the complete Standard Version of the interpreter is so embedded.

6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whoever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called "undump" or "unexec" methods of producing a binary executable image, then distribution of such an image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.

7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.

8. Aggregation of this Package with a commercial distribution is always permitted provided that the use of this Package is embedded; that is, when no overt attempt is made to make this Package's interfaces visible to the end user of the commercial distribution. Such use shall not be construed as a distribution of this Package.

9. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.

10. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The End

GNU GENERAL PUBLIC LICENSE

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Copyright (C) 1989 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it,

that you can change the software or use pieces of it in new free
programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have. You must make sure that they, too, receive or can get the
source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software. If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

The precise terms and conditions for copying, distribution and
modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any program or other work which
contains a notice placed by the copyright holder saying it may be
distributed under the terms of this General Public License. The
"Program", below, refers to any such program or work, and a "work based
on the Program" means either the Program or any work containing the

Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.

2. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

a) cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and

b) cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.

d) You may charge a fee for the physical act of transferring a

copy, and you may at your option offer warranty protection in
exchange for a fee.

Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

3. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

a) accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,

b) accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,

c) accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

4. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License.

Any attempt otherwise to copy, modify, sublicense, distribute or transfer
the Program is void, and will automatically terminate your rights to use
the Program under this License. However, parties who have received
copies, or rights to use copies, from you under this General Public
License will not have their licenses terminated so long as such parties
remain in full compliance.

5. By copying, distributing or modifying the Program (or any work based
on the Program) you indicate your acceptance of this license to do so,
and all its terms and conditions.

6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the original
licensor to copy, distribute or modify the Program subject to these
terms and conditions. You may not impose any further restrictions on the
recipients' exercise of the rights granted herein.

7. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number. If the Program
specifies a version number of the license which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation. If the Program does not specify a version number of
the license, you may choose any version ever published by the Free Software
Foundation.

8. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission. For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this. Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and

of promoting the sharing and reuse of software generally.

NO WARRANTY

9. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

10. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Perl

NIRVA contains an embedded perl version 5. Any third party can freely (except for a nominal charge for the cost of distribution) get a complete machine-readable copy of the corresponding source code, distributed under the terms of Paragraphs 1 and 2 of the GNU GENERAL PUBLIC LICENSE.

Perl Kit, Version 5

Copyright 1989-2002, Larry Wall

All rights reserved.

This program is free software; you can redistribute it and/or modify
it under the terms of either:

a) the GNU General Public License as published by the Free
Software Foundation; either version 1, or (at your option) any
later version, or

b) the "Artistic License" which comes with this Kit.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See either
the GNU General Public License or the Artistic License for more details.

You should have received a copy of the Artistic License with this
Kit, in the file named "Artistic". If not, I'll be glad to provide one.

You should also have received a copy of the GNU General Public License
along with this program in the file named "Copying". If not, write to the
Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
02111-1307, USA or visit their web page on the internet at
http://www.gnu.org/copyleft/gpl.html.

LIBXML/LIBXSLT

The MIT License

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including

without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Copyright © 2003 by the Open Source Initiative

Technical questions about the website go to Steve M.: webmaster at opensource.org / **Policy questions** about open source go to the Board of Directors.

The contents of this website are licensed under the Open Software License version 1.1.

OSI is a registered non-profit with 501(c)(3) status. Contact our Board for further donation information.