PyQGIS developer cookbook

Release 2.18

QGIS Project

April 08, 2019

Contents

1 Introducere **1**
- 1.1 Run Python code when QGIS starts . . . 1
- 1.2 Python Console . . . 2
- 1.3 Plugin-uri Python . . . 3
- 1.4 Aplicații Python . . . 3

2 Încărcarea proiectelor **7**

3 Încărcarea Straturilor **9**
- 3.1 Straturile Vectoriale . . . 9
- 3.2 Straturile Raster . . . 11
- 3.3 Map Layer Registry . . . 11

4 Utilizarea straturilor raster **13**
- 4.1 Detaliile stratului . . . 13
- 4.2 Render . . . 13
- 4.3 Refreshing Layers . . . 15
- 4.4 Interogarea valorilor . . . 15

5 Utilizarea straturilor vectoriale **17**
- 5.1 Obținerea informaţiilor despre atribute . . . 17
- 5.2 Selectarea entităților . . . 18
- 5.3 Iterații în straturile vectoriale . . . 18
- 5.4 Modificarea straturilor vectoriale . . . 20
- 5.5 Modificarea straturi vectoriale prin editarea unui tampon de memorie . . . 22
- 5.6 Crearea unui index spațial . . . 23
- 5.7 Writing Vector Layers . . . 23
- 5.8 Memory Provider . . . 24
- 5.9 Aspectul (simbologia) straturilor vectoriale . . . 26
- 5.10 Lecturi suplimentare . . . 33

6 Manipularea geometriei **35**
- 6.1 Construirea geometriei . . . 35
- 6.2 Accesarea geometriei . . . 36
- 6.3 Predicate și operațiuni geometrice . . . 36

7 Proiecții suportate **39**
- 7.1 Sisteme de coordonate de referință . . . 39
- 7.2 Projections . . . 40

8 Using Map Canvas **41**
- 8.1 Încapsularea suportului de hartă . . . 41
- 8.2 Folosirea instrumentelor în suportul de hartă . . . 42

8.3 Benzile elastice și marcajele nodurilor . . . 43
8.4 Dezvoltarea instrumentelor personalizate pentru suportul de hartă . . . 44
8.5 Dezvoltarea elementelor personalizate pentru suportul de hartă . . . 45

9 Randarea hărților și imprimarea **47**
9.1 Randarea simplă . . . 47
9.2 Randarea straturilor cu diferite CRS-uri . . . 48
9.3 Output using Map Composer . . . 48

10 Expresii, filtrarea și calculul valorilor **51**
10.1 Parsarea expresiilor . . . 52
10.2 Evaluarea expresiilor . . . 52
10.3 Exemple . . . 53

11 Citirea și stocarea setărilor **55**

12 Comunicarea cu utilizatorul **57**
12.1 Showing messages. The `QgsMessageBar` class . . . 57
12.2 Afișarea progresului . . . 58
12.3 Jurnalizare . . . 59

13 Dezvoltarea plugin-urilor Python **61**
13.1 Scrierea unui plugin . . . 62
13.2 Conținutul Plugin-ului . . . 63
13.3 Documentație . . . 67
13.4 Traducerea . . . 67

14 Infrastructura de autentificare **71**
14.1 Introducere . . . 71
14.2 Glosar . . . 71
14.3 QgsAuthManager the entry point . . . 72
14.4 Adapt plugins to use Authentication infrastructure . . . 75
14.5 Authentication GUIs . . . 75

15 Setările IDE pentru scrierea și depanarea de plugin-uri **79**
15.1 O notă privind configurarea IDE-ului în Windows . . . 79
15.2 Depanare cu ajutorul Eclipse și PyDev . . . 80
15.3 Depanarea cu ajutorul PDB . . . 84

16 Utilizarea straturilor plugin-ului **85**
16.1 Subclasarea QgsPluginLayer . . . 85

17 Compatibilitatea cu versiunile QGIS anterioare **87**
17.1 Meniul plugin-ului . . . 87

18 Lansarea plugin-ului dvs. **89**
18.1 Metadate și nume . . . 89
18.2 Codul și ajutorul . . . 89
18.3 Depozitul oficial al plugin-urilor python . . . 90

19 Fragmente de cod **93**
19.1 Cum să apelăm o metodă printr-o combinație rapidă de taste . . . 93
19.2 Inversarea Stării Straturilor . . . 93
19.3 Cum să accesați tabelul de atribute al entităților selectate . . . 93

20 Scrierea unui plugin Processing **95**
20.1 Creating a plugin that adds an algorithm provider . . . 95
20.2 Creating a plugin that contains a set of processing scripts . . . 95

21 Biblioteca de analiză a rețelelor **97**
21.1 Informații generale 97
21.2 Construirea unui graf 97
21.3 Analiza grafului 99

22 Plugin-uri Python pentru Serverul QGIS **105**
22.1 Arhitectura Plugin-urilor de Filtrare de pe Server 105
22.2 Tratarea excepțiilor provenite de la un plugin 107
22.3 Scrierea unui plugin pentru server 107
22.4 Plugin-ul de control al accesului 110

Index **113**

CHAPTER 1

Introducere

- Run Python code when QGIS starts
  - Variabila de mediu PYQGIS_STARTUP
  - Fişierul `startup.py`
- Python Console
- Plugin-uri Python
- Aplicații Python
  - Utilizarea PyQGIS în script-uri de sine stătătoare
  - Utilizarea PyQGIS în aplicații personalizate
  - Rularea Aplicațiilor Personalizate

This document is intended to work both as a tutorial and a reference guide. While it does not list all possible use cases, it should give a good overview of the principal functionality.

Starting from 0.9 release, QGIS has optional scripting support using Python language. We've decided for Python as it's one of the most favourite languages for scripting. PyQGIS bindings depend on SIP and PyQt4. The reason for using SIP instead of more widely used SWIG is that the whole QGIS code depends on Qt libraries. Python bindings for Qt (PyQt) are done also using SIP and this allows seamless integration of PyQGIS with PyQt.

There are several ways how to use Python bindings in QGIS desktop, they are covered in detail in the following sections:

- automatically run Python code when QGIS starts
- issue commands in Python console within QGIS
- create and use plugins in Python
- create custom applications based on QGIS API

Python bindings are also available for QGIS Server:

- starting from 2.8 release, Python plugins are also available on QGIS Server (see *Server Python Plugins*)
- starting from 2.11 version (Master at 2015-08-11), QGIS Server library has Python bindings that can be used to embed QGIS Server into a Python application.

There is a complete QGIS API reference that documents the classes from the QGIS libraries. Pythonic QGIS API is nearly identical to the API in C++.

A good resource when dealing with plugins is to download some plugins from plugin repository and examine their code. Also, the `python/plugins/` folder in your QGIS installation contains some plugin that you can use to learn how to develop such plugin and how to perform some of the most common tasks.

1.1 Run Python code when QGIS starts

Există două metode distincte de a rula cod Python de fiecare dată când porneşte QGIS.

1.1.1 Variabila de mediu PYQGIS_STARTUP

You can run Python code just before QGIS initialization completes by setting the `PYQGIS_STARTUP` environment variable to the path of an existing Python file.

This method is something you will probably rarely need, but worth mentioning here because it is one of the several ways to run Python code within QGIS and because this code will run before QGIS initialization is complete. This method is very useful for cleaning sys.path, which may have undesireable paths, or for isolating/loading the initial environ without requiring a virt env, e.g. homebrew or MacPorts installs on Mac.

1.1.2 Fişierul `startup.py`

Every time QGIS starts, the user's Python home directory (usually: `.qgis2/python`) is searched for a file named `startup.py`, if that file exists, it is executed by the embedded Python interpreter.

1.2 Python Console

For scripting, it is possible to take advantage of integrated Python console. It can be opened from menu: *Plugins* → *Python Console*. The console opens as a non-modal utility window:

Figure 1.1: Consola Python din QGIS

The screenshot above illustrates how to get the layer currently selected in the layer list, show its ID and optionally, if it is a vector layer, show the feature count. For interaction with QGIS environment, there is a `iface` variable, which is an instance of `QgsInterface`. This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application.

For convenience of the user, the following statements are executed when the console is started (in future it will be possible to set further initial commands)

```
from qgis.core import *
import qgis.utils
```

For those which use the console often, it may be useful to set a shortcut for triggering the console (within menu *Settings* → *Configure shortcuts...*)

1.3 Plugin-uri Python

QGIS allows enhancement of its functionality using plugins. This was originally possible only with C++ language. With the addition of Python support to QGIS, it is also possible to use plugins written in Python. The main advantage over C++ plugins is its simplicity of distribution (no compiling for each platform needed) and easier development.

Many plugins covering various functionality have been written since the introduction of Python support. The plugin installer allows users to easily fetch, upgrade and remove Python plugins. See the Python Plugin Repositories page for various sources of plugins.

Crearea de plugin-uri în Python este simplă, instrucțiuni detaliate găsindu-se în :ref: *developing_plugins*.

Note: Python plugins are also available in QGIS server (*label_qgisserver*), see *Plugin-uri Python pentru Serverul QGIS* for further details.

1.4 Aplicații Python

Often when processing some GIS data, it is handy to create some scripts for automating the process instead of doing the same task again and again. With PyQGIS, this is perfectly possible — import the `qgis.core` module, initialize it and you are ready for the processing.

Or you may want to create an interactive application that uses some GIS functionality — measure some data, export a map in PDF or any other functionality. The `qgis.gui` module additionally brings various GUI components, most notably the map canvas widget that can be very easily incorporated into the application with support for zooming, panning and/or any further custom map tools.

PyQGIS custom applications or standalone scripts must be configured to locate the QGIS resources such as projection information, providers for reading vector and raster layers, etc. QGIS Resources are initialized by adding a few lines to the beginning of your application or script. The code to initialize QGIS for custom applications and standalone scripts is similar, but examples of each are provided below.

Note: do *not* use `qgis.py` as a name for your test script — Python will not be able to import the bindings as the script's name will shadow them.

1.4.1 Utilizarea PyQGIS în script-uri de sine stătătoare

Pentru a starta un script independent, inițializați resursele QGIS la începutul script-ului, similar codului următor:

```
from qgis.core import *

# supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# create a reference to the QgsApplication, setting the
# second argument to False disables the GUI
qgs = QgsApplication([], False)

# load providers
qgs.initQgis()

# Write your code here to load some layers, use processing algorithms, etc.

# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

Începem prin importarea modulului `qgis.core`, urmată de configurarea prefixului căii. Prefixul căii este reprezentat de locația în care este instalat QGIS pe sistemul dumneavoastră. Configurarea sa în cadrul script-ului are loc prin apelarea metodei `setPrefixPath`. Al doilea argument al lui `setPrefixPath` este setat pe `TRUE`, care stabilește dacă sunt folosite căile implicite.

Calea de instalare a QGIS variază în funcție de platformă; cel mai simplu mod de a o identifica pe cea din sistemul dvs. este de a utiliza *Python Console* din interiorul QGIS și analizând rezultatul generat de execuția `QgsApplication.prefixPath()`.

După configurarea căii prefixului, în variabila `qgs` vom salva o referință către `QgsApplication`. Al doilea argument este setat la `False`, ceea ce indică faptul că nu avem de gând să utilizăm un GUI, atât timp cât dorim să scriem un script de sine stătător. Având `QgsApplication` configurată, vom încărca furnizorii de date QGIS și registrul stratului, prin apelarea metodei `qgs.initQgis()`. Aplicația QGIS fiind inițializată, suntem gata să scriem restul script-ul. În cele din urmă, vom încheia printr-un apel la `qgs.exitQgis()`, pentru a elimina din memorie furnizorii de date și registrul stratului.

1.4.2 Utilizarea PyQGIS în aplicații personalizate

Singura diferență dintre *Utilizarea PyQGIS în script-uri de sine stătătoare* și o aplicație PyQGIS particularizată este dată de al doilea argument, la instanțierea `QgsApplication`. Vom transmite `True` în loc de‘‘False‘‘, pentru a indica faptul că intenționăm să utilizăm o interfață grafică.

```
from qgis.core import *

# supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# create a reference to the QgsApplication
# setting the second argument to True enables the GUI, which we need to do
# since this is a custom application
qgs = QgsApplication([], True)

# load providers
qgs.initQgis()

# Write your code here to load some layers, use processing algorithms, etc.

# When your script is complete, call exitQgis() to remove the provider and
# layer registries from memory
qgs.exitQgis()
```

Now you can work with QGIS API — load layers and do some processing or fire up a GUI with a map canvas. The possibilities are endless :-)

1.4.3 Rularea Aplicațiilor Personalizate

Trebuie să indicați sistemului dvs. unde să caute bibliotecile QGIS și modulele Python corespunzătoare, atunci când acestea nu se află într-o locație standard — altfel, Python vă va notifica:

```
>>> import qgis.core
ImportError: No module named qgis.core
```

This can be fixed by setting the `PYTHONPATH` environment variable. In the following commands, `qgispath` should be replaced with your actual QGIS installation path:

- on Linux: **export PYTHONPATH=/qgispath/share/qgis/python**
- on Windows: **set PYTHONPATH=c:\qgispath\python**

The path to the PyQGIS modules is now known, however they depend on `qgis_core` and `qgis_gui` libraries (the Python modules serve only as wrappers). Path to these libraries is typically unknown for the operating system, so you get an import error again (the message might vary depending on the system):

```
>>> import qgis.core
ImportError: libqgis_core.so.1.5.0: cannot open shared object file: No such file or directory
```

Remediați acest lucru prin adăugarea directoarelor în care rezidă bibliotecile QGIS la calea de căutare a editorului de legături:

- on Linux: **export LD_LIBRARY_PATH=/qgispath/lib**
- on Windows: **set PATH=C:\qgispath;%PATH%**

Aceste comenzi pot fi puse într-un script bootstrap, care se va ocupa de pornire. Atunci când livrați aplicații personalizate folosind PyQGIS, există, de obicei, două variante:

- require user to install QGIS on his platform prior to installing your application. The application installer should look for default locations of QGIS libraries and allow user to set the path if not found. This approach has the advantage of being simpler, however it requires user to do more steps.
- să împachetați QGIS împreună cu aplicația dumneavoastră. Livrarea aplicației poate fi mai dificilă deoarece pachetul va fi foarte mare, dar utilizatorul va fi salvat de povara de a descărca și instala software suplimentar.

The two deployment models can be mixed - deploy standalone application on Windows and macOS, for Linux leave the installation of QGIS up to user and his package manager.

CHAPTER 2

Încărcarea proiectelor

Sometimes you need to load an existing project from a plugin or (more often) when developing a stand-alone QGIS Python application (see: *Aplicații Python*).

To load a project into the current QGIS application you need a `QgsProject instance()` object and call its `read()` method passing to it a `QFileInfo` object that contains the path from where the project will be loaded:

```
# If you are not inside a QGIS console you first need to import
# qgis and PyQt4 classes you will use in this script as shown below:
from qgis.core import QgsProject
from PyQt4.QtCore import QFileInfo
# Get the project instance
project = QgsProject.instance()
# Print the current project file name (might be empty in case no projects have been loaded)
print project.fileName()
u'/home/user/projects/my_qgis_project.qgs'
# Load another project
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
print project.fileName()
u'/home/user/projects/my_other_qgis_project.qgs'
```

In case you need to make some modifications to the project (for example add or remove some layers) and save your changes, you can call the `write()` method of your project instance. The `write()` method also accepts an optional `QFileInfo` that allows you to specify a path where the project will be saved:

```
# Save the project to the same
project.write()
# ... or to a new file
project.write(QFileInfo('/home/user/projects/my_new_qgis_project.qgs'))
```

Both `read()` and `write()` funtions return a boolean value that you can use to check if the operation was successful.

Note: If you are writing a QGIS standalone application, in order to synchronise the loaded project with the canvas you need to instanciate a `QgsLayerTreeMapCanvasBridge` as in the example below:

```
bridge = QgsLayerTreeMapCanvasBridge( \
         QgsProject.instance().layerTreeRoot(), canvas)
# Now you can safely load your project and see it in the canvas
project.read(QFileInfo('/home/user/projects/my_other_qgis_project.qgs'))
```

CHAPTER 3

Încărcarea Straturilor

- Straturile Vectoriale
- Straturile Raster
- Map Layer Registry

Haideți să deschidem mai multe straturi cu date. QGIS recunoaște straturile vectoriale și pe cele de tip raster. În plus, sunt disponibile și tipurile de straturi personalizate, dar pe acestea nu le vom discuta aici.

3.1 Straturile Vectoriale

To load a vector layer, specify layer's data source identifier, name for the layer and provider's name:

```
layer = QgsVectorLayer(data_source, layer_name, provider_name)
if not layer.isValid():
  print "Layer failed to load!"
```

Identificatorul sursei de date reprezintă un șir specific pentru fiecare furnizor de date vectoriale în parte. Numele stratului se va afișa în lista straturilor. Este important să se verifice dacă stratul a fost încărcat cu succes. În cazul neîncărcării cu succes, va fi returnată o instanță de strat nevalid.

The quickest way to open and display a vector layer in QGIS is the addVectorLayer function of the `QgisInterface`:

```
layer = iface.addVectorLayer("/path/to/shapefile/file.shp", "layer name you like", "ogr")
if not layer:
  print "Layer failed to load!"
```

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step. The function returns the layer instance or *None* if the layer couldn't be loaded.

Lista de mai jos arată modul de accesare a diverselor surse de date, cu ajutorul furnizorilor de date vectoriale:

- OGR library (shapefiles and many other file formats) — data source is the path to the file:
  - for shapefile:

    ```
    vlayer = QgsVectorLayer("/path/to/shapefile/file.shp", "layer_name_you_like", "ogr")
    ```

  - for dxf (note the internal options in data source uri):

    ```
    uri = "/path/to/dxffile/file.dxf|layername=entities|geometrytype=Point"
    vlayer = QgsVectorLayer(uri, "layer_name_you_like", "ogr")
    ```

- PostGIS database — data source is a string with all information needed to create a connection to PostgreSQL database. `QgsDataSourceURI` class can generate this string for you. Note that QGIS has to be compiled with Postgres support, otherwise this provider isn't available:

```
uri = QgsDataSourceURI()
# set host name, port, database name, username and password
uri.setConnection("localhost", "5432", "dbname", "johny", "xxx")
# set database schema, table name, geometry column and optionally
# subset (WHERE clause)
uri.setDataSource("public", "roads", "the_geom", "cityid = 2643")

vlayer = QgsVectorLayer(uri.uri(False), "layer name you like", "postgres")
```

Note: The `False` argument passed to `uri.uri(False)` prevents the expansion of the authentication configuration parameters, if you are not using any authentication configuration this argument does not make any difference.

- CSV or other delimited text files — to open a file with a semicolon as a delimiter, with field "x" for x-coordinate and field "y" for y-coordinate you would use something like this:

```
uri = "/some/path/file.csv?delimiter=%s&xField=%s&yField=%s" % (";", "x", "y")
vlayer = QgsVectorLayer(uri, "layer name you like", "delimitedtext")
```

Note: The provider string is structured as a URL, so the path must be prefixed with `file://`. Also it allows WKT (well known text) formatted geometries as an alternative to `x` and `y` fields, and allows the coordinate reference system to be specified. For example:

```
uri = "file:///some/path/file.csv?delimiter=%s&crs=epsg:4723&wktField=%s" % (";", "shape")
```

- Fișiere GPX — furnizorul de date "gpx" citește urme, rute și puncte de referință din fișiere gpx. Pentru a deschide un fișier, tipul (urmă/traseu/punct de referință) trebuie să fie specificat ca parte a url-ului:

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer name you like", "gpx")
```

- SpatiaLite database — Similarly to PostGIS databases, `QgsDataSourceURI` can be used for generation of data source identifier:

```
uri = QgsDataSourceURI()
uri.setDatabase('/home/martin/test-2.3.sqlite')
schema = ''
table = 'Towns'
geom_column = 'Geometry'
uri.setDataSource(schema, table, geom_column)

display_name = 'Towns'
vlayer = QgsVectorLayer(uri.uri(), display_name, 'spatialite')
```

- Geometrii MySQL bazate pe WKB, prin OGR — sursa de date o reprezintă șirul de conectare la tabelă:

```
uri = "MySQL:dbname,host=localhost,port=3306,user=root,password=xxx|layername=my_table"
vlayer = QgsVectorLayer( uri, "my table", "ogr" )
```

- Conexiune WFS:. conexiunea este definită cu un URI și cu ajutorul furnizorului `WFS`:

```
uri = "http://localhost:8080/geoserver/wfs?srsname=EPSG:23030&typename=union&version=1.0.0&re
vlayer = QgsVectorLayer(uri, "my wfs layer", "WFS")
```

Identificatorul URI poate fi creat folosindu-se biblioteca standard `urllib`:

```
params = {
    'service': 'WFS',
    'version': '1.0.0',
    'request': 'GetFeature',
    'typename': 'union',
    'srsname': "EPSG:23030"
```

```
}
uri = 'http://localhost:8080/geoserver/wfs?' + urllib.unquote(urllib.urlencode(params))
```

Note: You can change the data source of an existing layer by calling `setDataSource()` on a `QgsVectorLayer` instance, as in the following example:

```
# layer is a vector layer, uri is a QgsDataSourceURI instance
layer.setDataSource(uri.uri(), "layer name you like", "postgres")
```

3.2 Straturile Raster

For accessing raster files, GDAL library is used. It supports a wide range of file formats. In case you have troubles with opening some files, check whether your GDAL has support for the particular format (not all formats are available by default). To load a raster from a file, specify its file name and base name:

```
fileName = "/path/to/raster/file.tif"
fileInfo = QFileInfo(fileName)
baseName = fileInfo.baseName()
rlayer = QgsRasterLayer(fileName, baseName)
if not rlayer.isValid():
  print "Layer failed to load!"
```

Similarly to vector layers, raster layers can be loaded using the addRasterLayer function of the `QgisInterface`:

```
iface.addRasterLayer("/path/to/raster/file.tif", "layer name you like")
```

This creates a new layer and adds it to the map layer registry (making it appear in the layer list) in one step.

Straturile raster pot fi, de asemenea, create dintr-un serviciu WCS:

```
layer_name = 'modis'
uri = QgsDataSourceURI()
uri.setParam('url', 'http://demo.mapserver.org/cgi-bin/wcs')
uri.setParam("identifier", layer_name)
rlayer = QgsRasterLayer(str(uri.encodedUri()), 'my wcs layer', 'wcs')
```

detailed URI settings can be found in provider documentation

Alternativ, puteți încărca un strat raster de pe un server WMS. Cu toate acestea, în prezent, nu este posibilă accesarea din API a răspunsului GetCapabilities — trebuie să cunoașteți straturile dorite:

```
urlWithParams = 'url=http://irs.gis-lab.info/?layers=landsat&styles=&format=image/jpeg&crs=EPSG:4
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print "Layer failed to load!"
```

3.3 Map Layer Registry

If you would like to use the opened layers for rendering, do not forget to add them to map layer registry. The map layer registry takes ownership of layers and they can be later accessed from any part of the application by their unique ID. When the layer is removed from map layer registry, it gets deleted, too.

Adding a layer to the registry:

```
QgsMapLayerRegistry.instance().addMapLayer(layer)
```

Layers are destroyed automatically on exit, however if you want to delete the layer explicitly, use:

```
QgsMapLayerRegistry.instance().removeMapLayer(layer_id)
```

For a list of loaded layers and layer ids, use:

```
QgsMapLayerRegistry.instance().mapLayers()
```

CHAPTER 4

Utilizarea straturilor raster

- Detaliile stratului
- Render
  - Rastere cu o singură bandă
  - Rastere multibandă
- Refreshing Layers
- Interogarea valorilor

This sections lists various operations you can do with raster layers.

4.1 Detaliile stratului

A raster layer consists of one or more raster bands — it is referred to as either single band or multi band raster. One band represents a matrix of values. Usual color image (e.g. aerial photo) is a raster consisting of red, blue and green band. Single band layers typically represent either continuous variables (e.g. elevation) or discrete variables (e.g. land use). In some cases, a raster layer comes with a palette and raster values refer to colors stored in the palette:

```
rlayer.width(), rlayer.height()
(812, 301)
rlayer.extent()
<qgis._core.QgsRectangle object at 0x000000000F8A2048>
rlayer.extent().toString()
u'12.095833,48.552777 : 18.863888,51.056944'
rlayer.rasterType()
2  # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
rlayer.bandCount()
3
rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
rlayer.hasPyramids()
False
```

4.2 Render

Când un strat raster este încărcat, în funcție de tipul său, va moșteni un stil de desenare implicit. Acesta poate fi modificat, fie prin modificarea manuală a proprietăților rasterului, fie programatic.

To query the current renderer:

```
>>> rlayer.renderer()
<qgis._core.QgsSingleBandPseudoColorRenderer object at 0x7f471c1da8a0>
>>> rlayer.renderer().type()
u'singlebandpseudocolor'
```

To set a renderer use `setRenderer()` method of `QgsRasterLayer`. There are several available renderer classes (derived from `QgsRasterRenderer`):

- QgsMultiBandColorRenderer
- QgsPalettedRasterRenderer
- QgsSingleBandColorDataRenderer
- QgsSingleBandGrayRenderer
- QgsSingleBandPseudoColorRenderer

Straturile cu o singură bandă raster pot fi desenate fie în nuanțe de gri (valori mici = negru, valori ridicate = alb), sau cu un algoritm cu pseudoculori, care atribuie culori valorilor din banda singulară. Rasterele cu o singură bandă pot fi desenate folosindu-se propria paletă. Straturile multibandă sunt, de obicei, desenate prin maparea benzilor la culori RGB. Altă posibilitate este de a utiliza doar o singură bandă pentru desenarea în tonuri de gri sau cu pseudoculori.

The following sections explain how to query and modify the layer drawing style. After doing the changes, you might want to force update of map canvas, see *Refreshing Layers*.

TODO: contrast enhancements, transparency (no data), user defined min/max, band statistics

4.2.1 Rastere cu o singură bandă

Let's say we want to render our raster layer (assuming one band only) with colors ranging from green to yellow (for pixel values from 0 to 255). In the first stage we will prepare `QgsRasterShader` object and configure its shader function:

```
>>> fcn = QgsColorRampShader()
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
    QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn.setColorRampItemList(lst)
>>> shader = QgsRasterShader()
>>> shader.setRasterShaderFunction(fcn)
```

Nuanțatorul mapează culorile hărții, așa cum este specificat în harta sa de culori. Harta de culoare reprezintă o listă de elemente cu valorile pixelilor și culoarea asociată acestora.Există trei moduri de interpolare a valorilor:

- linear (`INTERPOLATED`): resulting color is linearly interpolated from the color map entries above and below the actual pixel value
- discrete (`DISCRETE`): color is used from the color map entry with equal or higher value
- exact (`EXACT`): color is not interpolated, only the pixels with value equal to color map entries are drawn

In the second step we will associate this shader with the raster layer:

```
>>> renderer = QgsSingleBandPseudoColorRenderer(layer.dataProvider(), 1, shader)
>>> layer.setRenderer(renderer)
```

The number 1 in the code above is band number (raster bands are indexed from one).

4.2.2 Rastere multibandă

By default, QGIS maps the first three bands to red, green and blue values to create a color image (this is the `MultiBandColor` drawing style. In some cases you might want to override these setting. The following code

interchanges red band (1) and green band (2):

```
rlayer.renderer().setGreenBand(1)
rlayer.renderer().setRedBand(2)
```

In case only one band is necessary for visualization of the raster, single band drawing can be chosen — either gray levels or pseudocolor.

4.3 Refreshing Layers

If you do change layer symbology and would like ensure that the changes are immediately visible to the user, call these methods

```
if hasattr(layer, "setCacheImage"):
  layer.setCacheImage(None)
layer.triggerRepaint()
```

The first call will ensure that the cached image of rendered layer is erased in case render caching is turned on. This functionality is available from QGIS 1.4, in previous versions this function does not exist — to make sure that the code works with all versions of QGIS, we first check whether the method exists.

Note: This method is deprecated as of QGIS 2.18.0 and will produce a warning. Simply calling `triggerRepaint()` is sufficient.

The second call emits signal that will force any map canvas containing the layer to issue a refresh.

With WMS raster layers, these commands do not work. In this case, you have to do it explicitly

```
layer.dataProvider().reloadData()
layer.triggerRepaint()
```

In case you have changed layer symbology (see sections about raster and vector layers on how to do that), you might want to force QGIS to update the layer symbology in the layer list (legend) widget. This can be done as follows (`iface` is an instance of `QgisInterface`)

```
iface.legendInterface().refreshLayerSymbology(layer)
```

4.4 Interogarea valorilor

To do a query on value of bands of raster layer at some specified point

```
ident = rlayer.dataProvider().identify(QgsPoint(15.30, 40.98), \
  QgsRaster.IdentifyFormatValue)
if ident.isValid():
  print ident.results()
```

The `results` method in this case returns a dictionary, with band indices as keys, and band values as values.

```
{1: 17, 2: 220}
```

CHAPTER 5

Utilizarea straturilor vectoriale

- Obținerea informaţiilor despre atribute
- Selectarea entităților
- Iterații în straturile vectoriale
  - Accesarea atributelor
  - Parcurgerea entităților selectate
  - Parcurgerea unui subset de entități
- Modificarea straturilor vectoriale
  - Adăugarea entităților
  - Ștergerea entităților
  - Modificarea entităților
  - Adăugarea și eliminarea câmpurilor
- Modificarea straturi vectoriale prin editarea unui tampon de memorie
- Crearea unui index spațial
- Writing Vector Layers
- Memory Provider
- Aspectul (simbologia) straturilor vectoriale
  - Render cu Simbol Unic
  - Render cu Simboluri Categorisite
  - Render cu Simboluri Graduale
  - Lucrul cu Simboluri
    - Lucrul cu Straturile Simbolului
    - Crearea unor Tipuri Personalizate de Straturi pentru Simboluri
  - Crearea renderelor Personalizate
- Lecturi suplimentare

Această secțiune rezumă diferitele acțiuni care pot fi efectuate asupra straturilor vectoriale.

5.1 Obținerea informaţiilor despre atribute

You can retrieve information about the fields associated with a vector layer by calling `pendingFields()` on a `QgsVectorLayer` instance:

```
# "layer" is a QgsVectorLayer instance
for field in layer.pendingFields():
    print field.name(), field.typeName()
```

Note: Starting from QGIS 2.12 there is also a `fields()` in `QgsVectorLayer` which is an alias to `pendingFields()`.

5.2 Selectarea entităților

In QGIS desktop, features can be selected in different ways, the user can click on a feature, draw a rectangle on the map canvas or use an expression filter. Selected features are normally highlighted in a different color (default is yellow) to draw user's attention on the selection. Sometimes can be useful to programmatically select features or to change the default color.

To change the selection color you can use `setSelectionColor()` method of `QgsMapCanvas` as shown in the following example:

```
iface.mapCanvas().setSelectionColor( QColor("red") )
```

To add add features to the selected features list for a given layer, you can call `setSelectedFeatures()` passing to it the list of features IDs:

```
# Get the active layer (must be a vector layer)
layer = iface.activeLayer()
# Get the first feature from the layer
feature = layer.getFeatures().next()
# Add this features to the selected list
layer.setSelectedFeatures([feature.id()])
```

To clear the selection, just pass an empty list:

```
layer.setSelectedFeatures([])
```

5.3 Iterații în straturile vectoriale

Iterating over the features in a vector layer is one of the most common tasks. Below is an example of the simple basic code to perform this task and showing some information about each feature. the `layer` variable is assumed to have a `QgsVectorLayer` object

```
iter = layer.getFeatures()
for feature in iter:
    # retrieve every feature with its geometry and attributes
    # fetch geometry
    geom = feature.geometry()
    print "Feature ID %d: " % feature.id()

    # show some information about the feature
    if geom.type() == QGis.Point:
        x = geom.asPoint()
        print "Point: " + str(x)
    elif geom.type() == QGis.Line:
        x = geom.asPolyline()
        print "Line: %d points" % len(x)
    elif geom.type() == QGis.Polygon:
        x = geom.asPolygon()
        numPts = 0
        for ring in x:
            numPts += len(ring)
        print "Polygon: %d rings with %d points" % (len(x), numPts)
    else:
        print "Unknown"

    # fetch attributes
    attrs = feature.attributes()

    # attrs is a list. It contains all the attribute values of this feature
    print attrs
```

5.3.1 Accesarea atributelor

Attributes can be referred to by their name.

```
print feature['name']
```

Alternatively, attributes can be referred to by index. This is will be a bit faster than using the name. For example, to get the first attribute:

```
print feature[0]
```

5.3.2 Parcurgerea entităților selectate

if you only need selected features, you can use the `selectedFeatures()` method from vector layer:

```
selection = layer.selectedFeatures()
print len(selection)
for feature in selection:
    # do whatever you need with the feature
```

Another option is the Processing `features()` method:

```
import processing
features = processing.features(layer)
for feature in features:
    # do whatever you need with the feature
```

By default, this will iterate over all the features in the layer, in case there is no selection, or over the selected features otherwise. Note that this behavior can be changed in the Processing options to ignore selections.

5.3.3 Parcurgerea unui subset de entități

If you want to iterate over a given subset of features in a layer, such as those within a given area, you have to add a `QgsFeatureRequest` object to the `getFeatures()` call. Here's an example

```
request = QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for feature in layer.getFeatures(request):
    # do whatever you need with the feature
```

With `setLimit()` you can limit the number of requested features. Here's an example

```
request = QgsFeatureRequest()
request.setLimit(2)
for feature in layer.getFeatures(request):
    # loop through only 2 features
```

If you need an attribute-based filter instead (or in addition) of a spatial one like shown in the examples above, you can build an `QgsExpression` object and pass it to the `QgsFeatureRequest` constructor. Here's an example

```
# The expression will filter the features where the field "location_name"
# contains the word "Lake" (case insensitive)
exp = QgsExpression('location_name ILIKE \'%Lake%\'')
request = QgsFeatureRequest(exp)
```

See *Expresii, filtrarea și calculul valorilor* for the details about the syntax supported by `QgsExpression`.

Cererea poate fi utilizată pentru a defini datele cerute pentru fiecare entitate, astfel încât iteratorul să întoarcă toate entitățile, dar să returneze datele parțiale pentru fiecare dintre ele.

```
# Only return selected fields
request.setSubsetOfAttributes([0,2])
# More user friendly version
request.setSubsetOfAttributes(['name','id'],layer.pendingFields())
# Don't return geometry objects
request.setFlags(QgsFeatureRequest.NoGeometry)
```

Tip: Speed features request

If you only need a subset of the attributes or you don't need the geometry information, you can significantly increase the **speed** of the features request by using `QgsFeatureRequest.NoGeometry` flag or specifying a subset of attributes (possibly empty) like shown in the example above.

5.4 Modificarea straturilor vectoriale

Most vector data providers support editing of layer data. Sometimes they support just a subset of possible editing actions. Use the `capabilities()` function to find out what set of functionality is supported

```
caps = layer.dataProvider().capabilities()
# Check if a particular capability is supported:
caps & QgsVectorDataProvider.DeleteFeatures
# Print 2 if DeleteFeatures is supported
```

For a list of all available capabilities, please refer to the API Documentation of QgsVectorDataProvider

To print layer's capabilities textual description in a comma separated list you can use `capabilitiesString()` as in the following example:

```
caps_string = layer.dataProvider().capabilitiesString()
# Print:
# u'Add Features, Delete Features, Change Attribute Values,
# Add Attributes, Delete Attributes, Create Spatial Index,
# Fast Access to Features at ID, Change Geometries,
# Simplify Geometries with topological validation'
```

Utilizând oricare dintre următoarele metode de editare a straturilor vectoriale, schimbările sunt efectuate direct în depozitul de date (un fișier, o bază de date etc). În cazul în care doriți să faceți doar schimbări temporare, treceți la secțiunea următoare, care explică efectuarea *modifications with editing buffer*.

Note: If you are working inside QGIS (either from the console or from a plugin), it might be necessary to force a redraw of the map canvas in order to see the changes you've done to the geometry, to the style or to the attributes:

```
# If caching is enabled, a simple canvas refresh might not be sufficient
# to trigger a redraw and you must clear the cached image for the layer
if iface.mapCanvas().isCachingEnabled():
    layer.setCacheImage(None)
else:
    iface.mapCanvas().refresh()
```

5.4.1 Adăugarea entităților

Create some `QgsFeature` instances and pass a list of them to provider's `addFeatures()` method. It will return two values: result (true/false) and list of added features (their ID is set by the data store).

To set up the attributes you can either initialize the feature passing a `QgsFields` instance or call `initAttributes()` passing the number of fields you want to be added.

```
if caps & QgsVectorDataProvider.AddFeatures:
    feat = QgsFeature(layer.pendingFields())
    feat.setAttributes([0, 'hello'])
    # Or set a single attribute by key or by index:
    feat.setAttribute('name', 'hello')
    feat.setAttribute(0, 'hello')
    feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
    (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

5.4.2 Ștergerea entităților

To delete some features, just provide a list of their feature IDs

```
if caps & QgsVectorDataProvider.DeleteFeatures:
    res = layer.dataProvider().deleteFeatures([5, 10])
```

5.4.3 Modificarea entităților

It is possible to either change feature's geometry or to change some attributes. The following example first changes values of attributes with index 0 and 1, then it changes the feature's geometry

```
fid = 100   # ID of the feature we will modify

if caps & QgsVectorDataProvider.ChangeAttributeValues:
    attrs = { 0 : "hello", 1 : 123 }
    layer.dataProvider().changeAttributeValues({ fid : attrs })

if caps & QgsVectorDataProvider.ChangeGeometries:
    geom = QgsGeometry.fromPoint(QgsPoint(111,222))
    layer.dataProvider().changeGeometryValues({ fid : geom })
```

Tip: Favor QgsVectorLayerEditUtils class for geometry-only edits

If you only need to change geometries, you might consider using the `QgsVectorLayerEditUtils` which provides some of useful methods to edit geometries (translate, insert or move vertex etc.).

Tip: Directly save changes using `with` based command

Using `with edit(layer):` the changes will be commited automatically calling `commitChanges()` at the end. If any exception occurs, it will `rollBack()` all the changes. See *Modificarea straturi vectoriale prin editarea unui tampon de memorie*.

5.4.4 Adăugarea și eliminarea câmpurilor

Pentru a adăuga câmpuri (atribute), trebuie să specificați o listă de definiții pentru acestea. Pentru ștergerea de câmpuri e suficientă furnizarea unei liste de indecși pentru câmpuri.

```
from PyQt4.QtCore import QVariant

if caps & QgsVectorDataProvider.AddAttributes:
    res = layer.dataProvider().addAttributes(
        [QgsField("mytext", QVariant.String),
        QgsField("myint", QVariant.Int)])

if caps & QgsVectorDataProvider.DeleteAttributes:
    res = layer.dataProvider().deleteAttributes([0])
```

După adăugarea sau eliminarea câmpurilor din furnizorul de date, câmpurile stratului trebuie să fie actualizate, deoarece modificările nu se propagă automat.

```
layer.updateFields()
```

5.5 Modificarea straturi vectoriale prin editarea unui tampon de memorie

When editing vectors within QGIS application, you have to first start editing mode for a particular layer, then do some modifications and finally commit (or rollback) the changes. All the changes you do are not written until you commit them — they stay in layer's in-memory editing buffer. It is possible to use this functionality also programmatically — it is just another method for vector layer editing that complements the direct usage of data providers. Use this option when providing some GUI tools for vector layer editing, since this will allow user to decide whether to commit/rollback and allows the usage of undo/redo. When committing changes, all changes from the editing buffer are saved to data provider.

To find out whether a layer is in editing mode, use `isEditable()` — the editing functions work only when the editing mode is turned on. Usage of editing functions

```
from PyQt4.QtCore import QVariant

# add two features (QgsFeature instances)
layer.addFeatures([feat1,feat2])
# delete a feature with specified ID
layer.deleteFeature(fid)

# set new geometry (QgsGeometry instance) for a feature
layer.changeGeometry(fid, geometry)
# update an attribute with given field index (int) to given value (QVariant)
layer.changeAttributeValue(fid, fieldIndex, value)

# add new field
layer.addAttribute(QgsField("mytext", QVariant.String))
# remove a field
layer.deleteAttribute(fieldIndex)
```

In order to make undo/redo work properly, the above mentioned calls have to be wrapped into undo commands. (If you do not care about undo/redo and want to have the changes stored immediately, then you will have easier work by *editing with data provider*.) How to use the undo functionality

```
layer.beginEditCommand("Feature triangulation")

# ... call layer's editing methods ...

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

The `beginEditCommand()` will create an internal "active" command and will record subsequent changes in vector layer. With the call to `endEditCommand()` the command is pushed onto the undo stack and the user will be able to undo/redo it from GUI. In case something went wrong while doing the changes, the `destroyEditCommand()` method will remove the command and rollback all changes done while this command was active.

To start editing mode, there is `startEditing()` method, to stop editing there are `commitChanges()` and `rollBack()` — however normally you should not need these methods and leave this functionality to be triggered by the user.

De asemenea, puteți utiliza expresia `with edit(layer)` - pentru a încorpora într-un bloc de cod semantic, pentru commit și rollback, așa cum se arată în exemplul de mai jos:

```
with edit(layer):
  feat = layer.getFeatures().next()
  feat[0] = 5
  layer.updateFeature(feat)
```

This will automatically call `commitChanges()` in the end. If any exception occurs, it will `rollBack()` all the changes. In case a problem is encountered within `commitChanges()` (when the method returns False) a `QgsEditError` exception will be raised.

5.6 Crearea unui index spațial

Indecșii spațiali pot îmbunătăți dramatic performanța codului dvs, în cazul în care este nevoie să interogați frecvent un strat vectorial. Imaginați-vă, de exemplu, că scrieți un algoritm de interpolare, și că, pentru o anumită locație, trebuie să aflați cele mai apropiate 10 puncte dintr-un strat, în scopul utilizării acelor puncte în calculul valorii interpolate. Fără un index spațial, singura modalitate pentru QGIS de a găsi cele 10 puncte, este de a calcula distanța tuturor punctelor față de locația specificată și apoi de a compara aceste distanțe. Această sarcină poate fi mare consumatoare de timp, mai ales în cazul în care trebuie să fie repetată pentru mai multe locații. Dacă pentru stratul respectiv există un index spațial, operațiunea va fi mult mai eficientă.

Gândiți-vă la un strat fără index spațial ca la o carte de telefon în care numerele de telefon nu sunt ordonate sau indexate. Singura modalitate de a afla numărul de telefon al unei anumite persoane este de a citi toate numerele, începând cu primul, până când îl găsiți.

Indecșii spațiali nu sunt creați în mod implicit pentru un strat QGIS vectorial, dar îi puteți genera cu ușurință. Iată ce trebuie să faceți:

- create spatial index — the following code creates an empty index

  ```
  index = QgsSpatialIndex()
  ```

- add features to index — index takes `QgsFeature` object and adds it to the internal data structure. You can create the object manually or use one from previous call to provider's `nextFeature()`

  ```
  index.insertFeature(feat)
  ```

- alternatively, you can load all features of a layer at once using bulk loading

  ```
  index = QgsSpatialIndex(layer.getFeatures())
  ```

- o dată ce ați introdus valori în indexul spațial, puteți efectua unele interogări

  ```
  # returns array of feature IDs of five nearest features
  nearest = index.nearestNeighbor(QgsPoint(25.4, 12.7), 5)

  # returns array of IDs of features which intersect the rectangle
  intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
  ```

5.7 Writing Vector Layers

You can write vector layer files using `QgsVectorFileWriter` class. It supports any other kind of vector file that OGR supports (shapefiles, GeoJSON, KML and others).

There are two possibilities how to export a vector layer:

- from an instance of `QgsVectorLayer`

```
error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_shapes.shp", "CP1250", None, "ESRI

if error == QgsVectorFileWriter.NoError:
    print "success!"

error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", "utf-8", None, "GeoJSO
if error == QgsVectorFileWriter.NoError:
    print "success again!"
```

The third parameter specifies output text encoding. Only some drivers need this for correct operation - shapefiles are one of those — however in case you are not using international characters you do not have to care much about the encoding. The fourth parameter that we left as `None` may specify destination CRS — if a valid instance of `QgsCoordinateReferenceSystem` is passed, the layer is transformed to that CRS.

For valid driver names please consult the supported formats by OGR — you should pass the value in the "Code" column as the driver name. Optionally you can set whether to export only selected features, pass further driver-specific options for creation or tell the writer not to create attributes — look into the documentation for full syntax.

- directly from features

```
from PyQt4.QtCore import QVariant

# define fields for feature attributes. A QgsFields object is needed
fields = QgsFields()
fields.append(QgsField("first", QVariant.Int))
fields.append(QgsField("second", QVariant.String))

""" create an instance of vector file writer, which will create the vector file.
Arguments:
1. path to new file (will fail if exists already)
2. encoding of the attributes
3. field map
4. geometry type - from WKBTYPE enum
5. layer's spatial reference (instance of
   QgsCoordinateReferenceSystem) - optional
6. driver name for the output file """
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Sh

if writer.hasError() != QgsVectorFileWriter.NoError:
    print "Error when creating shapefile: ",  w.errorMessage()

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes([1, "text"])
writer.addFeature(fet)

# delete the writer to flush features to disk
del writer
```

5.8 Memory Provider

Memory provider is intended to be used mainly by plugin or 3rd party app developers. It does not store data on disk, allowing developers to use it as a fast backend for some temporary layers.

Furnizorul suportă câmpuri de tip string, int sau double.

The memory provider also supports spatial indexing, which is enabled by calling the provider's `createSpatialIndex()` function. Once the spatial index is created you will be able to iterate over fea-

tures within smaller regions faster (since it's not necessary to traverse all the features, only those in specified rectangle).

A memory provider is created by passing `"memory"` as the provider string to the `QgsVectorLayer` constructor.

Constructorul are, de asemenea, un URI care definește unul din următoarele tipuri de geometrie a stratului: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"` sau `"MultiPolygon"`.

URI poate specifica, de asemenea, sistemul de coordonate de referință, câmpurile, precum și indexarea furnizorului de memorie. Sintaxa este:

crs=definiție Specifies the coordinate reference system, where definition may be any of the forms accepted by `QgsCoordinateReferenceSystem.createFromString()`

index=yes Specificați dacă furnizorul va utiliza un index spațial.

field=nume:tip(lungime,precizie) Specificați un atribut al stratului. Atributul are un nume și, opțional, un tip (integer, double sau string), lungime și precizie. Pot exista mai multe definiții de câmp.

Următorul exemplu de URI încorporează toate aceste opțiuni

```
"Point?crs=epsg:4326&field=id:integer&field=name:string(20)&index=yes"
```

Următorul exemplu de cod ilustrează crearea și popularea unui furnizor de memorie

```
from PyQt4.QtCore import QVariant

# create layer
vl = QgsVectorLayer("Point", "temporary_points", "memory")
pr = vl.dataProvider()

# add fields
pr.addAttributes([QgsField("name", QVariant.String),
                    QgsField("age",  QVariant.Int),
                    QgsField("size", QVariant.Double)])
vl.updateFields() # tell the vector layer to fetch changes from the provider

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes(["Johny", 2, 0.3])
pr.addFeatures([fet])

# update layer's extent when new features have been added
# because change of extent in provider is not propagated to the layer
vl.updateExtents()
```

În cele din urmă, să verificăm dacă totul a mers bine

```
# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
e = layer.extent()
print "extent:", e.xMiniminum(), e.yMinimum(), e.xMaximum(), e.yMaximum()

# iterate over features
f = QgsFeature()
features = vl.getFeatures()
for f in features:
    print "F:", f.id(), f.attributes(), f.geometry().asPoint()
```

5.9 Aspectul (simbologia) straturilor vectoriale

Când un strat vector este randat, aspectul datelor este dat de **render** și de **simbolurile** asociate stratului. Simbolurile sunt clase care au grijă de reprezentarea vizuală a tuturor entităților, în timp ce un render determină ce simbol va fi folosit doar pentru anumite entități.

The renderer for a given layer can obtained as shown below:

```
renderer = layer.rendererV2()
```

Și cu acea referință, să explorăm un pic

```
print "Type:", rendererV2.type()
```

There are several known renderer types available in QGIS core library:

Tipul	Clasa	Descrierea
singleSymbol	`QgsSingleSymbolRendererV2`	Asociază tuturor entităților același simbol
categorizedSymbol	`QgsCategorizedSymbolRendererV2`	Asociază entităților un simbol diferit, în funcție de categorie
graduatedSymbol	`QgsGraduatedSymbolRendererV2`	Asociază fiecărei entități un simbol diferit pentru fiecare gamă de valori

There might be also some custom renderer types, so never make an assumption there are just these types. You can query `QgsRendererV2Registry` singleton to find out currently available renderers:

```
print QgsRendererV2Registry.instance().renderersList()
# Print:
[u'singleSymbol',
u'categorizedSymbol',
u'graduatedSymbol',
u'RuleRenderer',
u'pointDisplacement',
u'invertedPolygonRenderer',
u'heatmapRenderer']
```

Este posibilă obținerea conținutului renderului sub formă de text — lucru util pentru depanare

```
print rendererV2.dump()
```

5.9.1 Render cu Simbol Unic

You can get the symbol used for rendering by calling `symbol()` method and change it with `setSymbol()` method (note for C++ devs: the renderer takes ownership of the symbol.)

You can change the symbol used by a particular vector layer by calling `setSymbol()` passing an instance of the appropriate symbol instance. Symbols for *point*, *line* and *polygon* layers can be created by calling the `createSimple()` function of the corresponding classes `QgsMarkerSymbolV2`, `QgsLineSymbolV2` and `QgsFillSymbolV2`.

The dictionary passed to `createSimple()` sets the style properties of the symbol.

For example you can replace the symbol used by a particular **point** layer by calling `setSymbol()` passing an instance of a `QgsMarkerSymbolV2` as in the following code example:

```
symbol = QgsMarkerSymbolV2.createSimple({'name': 'square', 'color': 'red'})
layer.rendererV2().setSymbol(symbol)
```

`nume`: indică forma markerului, aceasta putând fi oricare dintre următoarele:

- `cerc`
- `pătrat`

- cross
- dreptunghi
- diamant
- pentagon
- triunghi
- triunghi echilateral
- stea
- stea_regulată
- săgeată
- vârf_de_săgeată_plin
- x

To get the full list of properties for the first symbol layer of a simbol instance you can follow the example code:

```
print layer.rendererV2().symbol().symbolLayers()[0].properties()
# Prints
{u'angle': u'0',
u'color': u'0,128,0,255',
u'horizontal_anchor_point': u'1',
u'name': u'circle',
u'offset': u'0,0',
u'offset_map_unit_scale': u'0,0',
u'offset_unit': u'MM',
u'outline_color': u'0,0,0,255',
u'outline_style': u'solid',
u'outline_width': u'0',
u'outline_width_map_unit_scale': u'0,0',
u'outline_width_unit': u'MM',
u'scale_method': u'area',
u'size': u'2',
u'size_map_unit_scale': u'0,0',
u'size_unit': u'MM',
u'vertical_anchor_point': u'1'}
```

This can be useful if you want to alter some properties:

```
# You can alter a single property...
layer.rendererV2().symbol().symbolLayer(0).setName('square')
# ... but not all properties are accessible from methods,
# you can also replace the symbol completely:
props = layer.rendererV2().symbol().symbolLayer(0).properties()
props['color'] = 'yellow'
props['name'] = 'square'
layer.rendererV2().setSymbol(QgsMarkerSymbolV2.createSimple(props))
```

5.9.2 Render cu Simboluri Categorisite

You can query and set attribute name which is used for classification: use `classAttribute()` and `setClassAttribute()` methods.

Pentru a obține o listă de categorii

```
for cat in rendererV2.categories():
    print "%s: %s :: %s" % (cat.value().toString(), cat.label(), str(cat.symbol()))
```

Where `value()` is the value used for discrimination between categories, `label()` is a text used for category description and `symbol()` method returns assigned symbol.

The renderer usually stores also original symbol and color ramp which were used for the classification: `sourceColorRamp()` and `sourceSymbol()` methods.

5.9.3 Render cu Simboluri Graduale

Acest render este foarte similar cu renderul cu simbol clasificat, descris mai sus, dar în loc de o singură valoare de atribut per clasă el lucrează cu intervale de valori, putând fi, astfel, utilizat doar cu atribute numerice.

Pentru a afla mai multe despre gamele utilizate în render

```
for ran in rendererV2.ranges():
    print "%f - %f: %s %s" % (
        ran.lowerValue(),
        ran.upperValue(),
        ran.label(),
        str(ran.symbol())
      )
```

puteți folosi din nou `classAttribute()` pentru a afla numele atributului de clasificare, metodele `sourceSymbol()` și `sourceColorRamp()`. În plus, există metoda `mode()` care determină modul în care au fost create gamele: folosind intervale egale, cuantile sau o altă metodă.

Dacă doriți să creați propriul render cu simbol gradual, puteți face acest lucru așa cum este ilustrat în fragmentul de mai jos (care creează un simplu aranjament cu două clase)

```
from qgis.core import *

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2 myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)

myVectorLayer.setRendererV2(myRenderer)
QgsMapLayerRegistry.instance().addMapLayer(myVectorLayer)
```

5.9.4 Lucrul cu Simboluri

For representation of symbols, there is `QgsSymbolV2` base class with three derived classes:

- `QgsMarkerSymbolV2` — for point features
- `QgsLineSymbolV2` — for line features
- `QgsFillSymbolV2` — for polygon features

Every symbol consists of one or more symbol layers (classes derived from `QgsSymbolLayerV2`). The symbol layers do the actual rendering, the symbol class itself serves only as a container for the symbol layers.

Having an instance of a symbol (e.g. from a renderer), it is possible to explore it: `type()` method says whether it is a marker, line or fill symbol. There is a `dump()` method which returns a brief description of the symbol. To get a list of symbol layers

```
for i in xrange(symbol.symbolLayerCount()):
    lyr = symbol.symbolLayer(i)
    print "%d: %s" % (i, lyr.layerType())
```

To find out symbol's color use `color()` method and `setColor()` to change its color. With marker symbols additionally you can query for the symbol size and rotation with `size()` and `angle()` methods, for line symbols there is `width()` method returning line width.

Dimensiunea și lățimea sunt în milimetri, în mod implicit, iar unghiurile sunt în grade.

Lucrul cu Straturile Simbolului

As said before, symbol layers (subclasses of `QgsSymbolLayerV2`) determine the appearance of the features. There are several basic symbol layer classes for general use. It is possible to implement new symbol layer types and thus arbitrarily customize how features will be rendered. The `layerType()` method uniquely identifies the symbol layer class — the basic and default ones are SimpleMarker, SimpleLine and SimpleFill symbol layers types.

You can get a complete list of the types of symbol layers you can create for a given symbol layer class like this

```
from qgis.core import QgsSymbolLayerV2Registry
myRegistry = QgsSymbolLayerV2Registry.instance()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbolV2.Marker):
    print item
```

Output

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

`QgsSymbolLayerV2Registry` class manages a database of all available symbol layer types.

To access symbol layer data, use its `properties()` method that returns a key-value dictionary of properties which determine the appearance. Each symbol layer type has a specific set of properties that it uses. Additionally, there are generic methods `color()`, `size()`, `angle()`, `width()` with their setter counterparts. Of course size and angle is available only for marker symbol layers and width for line symbol layers.

Crearea unor Tipuri Personalizate de Straturi pentru Simboluri

Imaginați-vă că ați dori să personalizați modul în care se randează datele. Vă puteți crea propria dvs. clasă de strat de simbol, care va desena entitățile exact așa cum doriți. Iată un exemplu de marker care desenează cercuri roșii cu o rază specificată

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

  def __init__(self, radius=4.0):
      QgsMarkerSymbolLayerV2.__init__(self)
      self.radius = radius
      self.color = QColor(255,0,0)

  def layerType(self):
     return "FooMarker"

  def properties(self):
      return { "radius" : str(self.radius) }

  def startRender(self, context):
    pass

  def stopRender(self, context):
      pass

  def renderPoint(self, point, context):
      # Rendering depends on whether the symbol is selected (QGIS >= 1.5)
      color = context.selectionColor() if context.selected() else self.color
      p = context.renderContext().painter()
      p.setPen(color)
      p.drawEllipse(point, self.radius, self.radius)

  def clone(self):
      return FooSymbolLayer(self.radius)
```

The `layerType()` method determines the name of the symbol layer, it has to be unique among all symbol layers. Properties are used for persistence of attributes. `clone()` method must return a copy of the symbol layer with all attributes being exactly the same. Finally there are rendering methods: `startRender()` is called before rendering first feature, `stopRender()` when rendering is done. And `renderPoint()` method which does the rendering. The coordinates of the point(s) are already transformed to the output coordinates.

For polylines and polygons the only difference would be in the rendering method: you would use `renderPolyline()` which receives a list of lines, resp. `renderPolygon()` which receives list of points on outer ring as a first parameter and a list of inner rings (or None) as a second parameter.

De obicei, este convenabilă adăugarea unui GUI pentru setarea atributelor tipului de strat pentru simboluri, pentru a permite utilizatorilor să personalizeze aspectul: în exemplul de mai sus, putem lăsa utilizatorul să seteze raza cercului. Codul de mai jos implementează un astfel de widget

```
class FooSymbolLayerWidget(QgsSymbolLayerV2Widget):
    def __init__(self, parent=None):
        QgsSymbolLayerV2Widget.__init__(self, parent)

        self.layer = None

        # setup a simple UI
        self.label = QLabel("Radius:")
        self.spinRadius = QDoubleSpinBox()
        self.hbox = QHBoxLayout()
        self.hbox.addWidget(self.label)
        self.hbox.addWidget(self.spinRadius)
        self.setLayout(self.hbox)
        self.connect(self.spinRadius, SIGNAL("valueChanged(double)"), \
            self.radiusChanged)

    def setSymbolLayer(self, layer):
        if layer.layerType() != "FooMarker":
            return
        self.layer = layer
```

```
        self.spinRadius.setValue(layer.radius)

    def symbolLayer(self):
        return self.layer

    def radiusChanged(self, value):
        self.layer.radius = value
        self.emit(SIGNAL("changed()"))
```

Acest widget poate fi integrat în fereastra de proprietăți a simbolului. În cazul în care tipul de strat simbol este selectat în fereastra de proprietăți a simbolului, se creează o instanță a stratului simbol și o instanță a widget-ului stratului simbol. Apoi, se apelează metoda `setSymbolLayer()` pentru a aloca stratul simbol widget-ului. În acea metodă, widget-ul ar trebui să actualizeze UI pentru a reflecta atributele stratului simbol. Funcția `symbolLayer()` este utilizată la preluarea stratului simbol din fereastra de proprietăți, în scopul folosirii sale pentru simbol.

La fiecare schimbare de atribute, widget-ul ar trebui să emită semnalul `changed()` pentru a permite ferestrei de proprietăți să-și actualizeze previzualizarea simbolului.

Acum mai lipsește doar liantul final: pentru a face QGIS conștient de aceste noi clase. Acest lucru se face prin adăugarea stratului simbol la registru. Este posibilă utilizarea stratului simbol, de asemenea, fără a-l adăuga la registru, dar unele funcționalități nu vor fi disponibile: de exemplu, încărcarea de fișiere de proiect cu straturi simbol personalizate sau incapacitatea de a edita atributele stratului în GUI.

Va trebui să creăm metadate pentru stratul simbolului

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

  def __init__(self):
    QgsSymbolLayerV2AbstractMetadata.__init__(self, "FooMarker", QgsSymbolV2.Marker)

  def createSymbolLayer(self, props):
    radius = float(props[QString("radius")]) if QString("radius") in props else 4.0
    return FooSymbolLayer(radius)

  def createSymbolLayerWidget(self):
    return FooSymbolLayerWidget()

QgsSymbolLayerV2Registry.instance().addSymbolLayerType(FooSymbolLayerMetadata())
```

You should pass layer type (the same as returned by the layer) and symbol type (marker/line/fill) to the constructor of parent class. `createSymbolLayer()` takes care of creating an instance of symbol layer with attributes specified in the *props* dictionary. (Beware, the keys are QString instances, not “str” objects). And there is `createSymbolLayerWidget()` method which returns settings widget for this symbol layer type.

Ultimul pas este de a adăuga acest strat simbol la registru — și am încheiat.

5.9.5 Crearea renderelor Personalizate

Ar putea fi utilă crearea unei noi implementări de render, dacă doriți să personalizați regulile de selectare a simbolurilor pentru randarea entităților. Unele cazuri de utilizare: simbolul să fie determinat de o combinație de câmpuri, dimensiunea simbolurilor să depindă în funcție de scara curentă, etc

Urmatorul cod prezintă o simplă randare personalizată, care creează două simboluri de tip marker și apoi alege aleatoriu unul dintre ele pentru fiecare entitate

```
import random

class RandomRenderer(QgsFeatureRendererV2):
  def __init__(self, syms=None):
    QgsFeatureRendererV2.__init__(self, "RandomRenderer")
    self.syms = syms if syms else [QgsSymbolV2.defaultSymbol(QGis.Point), QgsSymbolV2.defaultSymb
```

```
  def symbolForFeature(self, feature):
    return random.choice(self.syms)

  def startRender(self, context, vlayer):
    for s in self.syms:
      s.startRender(context)

  def stopRender(self, context):
    for s in self.syms:
      s.stopRender(context)

  def usedAttributes(self):
    return []

  def clone(self):
    return RandomRenderer(self.syms)
```

The constructor of parent `QgsFeatureRendererV2` class needs renderer name (has to be unique among renderers). `symbolForFeature()` method is the one that decides what symbol will be used for a particular feature. `startRender()` and `stopRender()` take care of initialization/finalization of symbol rendering. `usedAttributes()` method can return a list of field names that renderer expects to be present. Finally `clone()` function should return a copy of the renderer.

Like with symbol layers, it is possible to attach a GUI for configuration of the renderer. It has to be derived from `QgsRendererV2Widget`. The following sample code creates a button that allows user to set symbol of the first symbol

```
class RandomRendererWidget(QgsRendererV2Widget):
  def __init__(self, layer, style, renderer):
    QgsRendererV2Widget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2()
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

  def setColor1(self):
    color = QColorDialog.getColor(self.r.syms[0].color(), self)
    if not color.isValid(): return
    self.r.syms[0].setColor(color);
    self.btn1.setColor(self.r.syms[0].color())

  def renderer(self):
    return self.r
```

The constructor receives instances of the active layer (`QgsVectorLayer`), the global style (`QgsStyleV2`) and current renderer. If there is no renderer or the renderer has different type, it will be replaced with our new renderer, otherwise we will use the current renderer (which has already the type we need). The widget contents should be updated to show current state of the renderer. When the renderer dialog is accepted, widget's `renderer()` method is called to get the current renderer — it will be assigned to the layer.

Ultimul bit lipsă este cel al metadatelor renderului și înregistrarea în registru, altfel încărcarea straturilor cu renderul nu va funcționa, iar utilizatorul nu va fi capabil să-l selecteze din lista de rendere. Să finalizăm exemplul nostru de RandomRenderer

```
class RandomRendererMetadata(QgsRendererV2AbstractMetadata):
  def __init__(self):
    QgsRendererV2AbstractMetadata.__init__(self, "RandomRenderer", "Random renderer")

  def createRenderer(self, element):
    return RandomRenderer()
  def createRendererWidget(self, layer, style, renderer):
    return RandomRendererWidget(layer, style, renderer)

QgsRendererV2Registry.instance().addRenderer(RandomRendererMetadata())
```

În mod similar cu straturile simbol, constructorul de metadate abstracte așteaptă numele renderului, nume vizibil pentru utilizatori și numele opțional al pictogramei renderului. Metoda `createRenderer()` transmite instanța `QDomElement` care poate fi folosită pentru a restabili starea renderului din arborele DOM. Metoda `createRendererWidget()` creează widget-ul de configurare. Aceasta nu trebuie să fie prezent sau ar putea returna *None*, dacă renderul nu vine cu GUI-ul.

To associate an icon with the renderer you can assign it in `QgsRendererV2AbstractMetadata` constructor as a third (optional) argument — the base class constructor in the RandomRendererMetadata `__init__()` function becomes

```
QgsRendererV2AbstractMetadata.__init__(self,
       "RandomRenderer",
       "Random renderer",
       QIcon(QPixmap("RandomRendererIcon.png", "png")))
```

The icon can be associated also at any later time using `setIcon()` method of the metadata class. The icon can be loaded from a file (as shown above) or can be loaded from a Qt resource (PyQt4 includes .qrc compiler for Python).

5.10 Lecturi suplimentare

DE EFECTUAT:

- creating/modifying symbols
- working with style (`QgsStyleV2`)
- working with color ramps (`QgsVectorColorRampV2`)
- rule-based renderer (see this blogpost)
- exploring symbol layer and renderer registries

CHAPTER 6

Manipularea geometriei

- Construirea geometriei
- Accesarea geometriei
- Predicate și operațiuni geometrice

Points, linestrings and polygons that represent a spatial feature are commonly referred to as geometries. In QGIS they are represented with the `QgsGeometry` class.

Uneori, o geometrie poate fi de fapt o colecție de simple geometrii (simple-părți). O astfel de geometrie poartă denumirea de geometrie multi-parte. În cazul în care conține doar un singur tip de geometrie simplă, o denumim multi-punct, multi-linie sau multi-poligon. De exemplu, o țară formată din mai multe insule poate fi reprezentată ca un multi-poligon.

Coordonatele geometriilor pot fi în orice sistem de coordonate de referință (CRS). Când extragem entitățile dintr-un strat, geometriile asociate vor avea coordonatele în CRS-ul stratului.

Description and specifications of all possible geometries construction and relationships are available in the OGC Simple Feature Access Standards for advanced details.

6.1 Construirea geometriei

There are several options for creating a geometry:

- din coordonate

```
gPnt = QgsGeometry.fromPoint(QgsPoint(1,1))
gLine = QgsGeometry.fromPolyline([QgsPoint(1, 1), QgsPoint(2, 2)])
gPolygon = QgsGeometry.fromPolygon([[QgsPoint(1, 1), QgsPoint(2, 2),
                                     QgsPoint(2, 1)]])
```

  Coordinates are given using `QgsPoint` class.

  Polyline (Linestring) is represented by a list of points. Polygon is represented by a list of linear rings (i.e. closed linestrings). First ring is outer ring (boundary), optional subsequent rings are holes in the polygon.

  Geometriile multi-parte merg cu un nivel mai departe: multi-punctele sunt o listă de puncte, multi-liniile o listă de linii iar multi-poligoanele sunt o listă de poligoane.

- din well-known text (WKT)

```
gem = QgsGeometry.fromWkt("POINT(3 4)")
```

- din well-known binary (WKB)

```
>>> g = QgsGeometry()
>>> wkb = '010100000000000000000045400000000000001440'.decode('hex')
>>> g.fromWkb(wkb)
```

```
>>> g.exportToWkt()
'Point (42 5)'
```

6.2 Accesarea geometriei

First, you should find out geometry type, `wkbType()` method is the one to use — it returns a value from `QGis.WkbType` enumeration

```
>>> gPnt.wkbType() == QGis.WKBPoint
True
>>> gLine.wkbType() == QGis.WKBLineString
True
>>> gPolygon.wkbType() == QGis.WKBPolygon
True
>>> gPolygon.wkbType() == QGis.WKBMultiPolygon
False
```

As an alternative, one can use `type()` method which returns a value from `QGis.GeometryType` enumeration. There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

To extract information from geometry there are accessor functions for every vector type. How to use accessors

```
>>> gPnt.asPoint()
(1, 1)
>>> gLine.asPolyline()
[(1, 1), (2, 2)]
>>> gPolygon.asPolygon()
[[(1, 1), (2, 2), (2, 1), (1, 1)]]
```

Note: The tuples (x,y) are not real tuples, they are `QgsPoint` objects, the values are accessible with `x()` and `y()` methods.

For multipart geometries there are similar accessor functions: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon()`.

6.3 Predicate și operațiuni geometrice

QGIS uses GEOS library for advanced geometry operations such as geometry predicates (`contains()`, `intersects()`, ...) and set operations (`union()`, `difference()`, ...). It can also compute geometric properties of geometries, such as area (in the case of polygons) or lengths (for polygons and lines)

Here you have a small example that combines iterating over the features in a given layer and performing some geometric computations based on their geometries.

```
# we assume that 'layer' is a polygon layer
features = layer.getFeatures()
for f in features:
  geom = f.geometry()
  print "Area:", geom.area()
  print "Perimeter:", geom.length()
```

Areas and perimeters don't take CRS into account when computed using these methods from the `QgsGeometry` class. For a more powerful area and distance calculation, the `QgsDistanceArea` class can be used. If projections are turned off, calculations will be planar, otherwise they'll be done on the ellipsoid.

```
d = QgsDistanceArea()
d.setEllipsoid('WGS84')
d.setEllipsoidalMode(True)
```

```
print "distance in meters: ", d.measureLine(QgsPoint(10,10),QgsPoint(11,11))
```

Puteți căuta mai multe exemple de algoritmi care sunt incluși în QGIS și să folosiți aceste metode pentru a analiza și a transforma datele vectoriale. Mai jos sunt prezente câteva trimiteri spre codul unora dintre ele.

Additional information can be found in following sources:

- Geometry transformation: Reproject algorithm
- Distance and area using the `QgsDistanceArea` class: Distance matrix algorithm
- Multi-part to single-part algorithm

CHAPTER 7

Proiecții suportate

- Sisteme de coordonate de referință
- Projections

7.1 Sisteme de coordonate de referință

Coordinate reference systems (CRS) are encapsulated by `QgsCoordinateReferenceSystem` class. Instances of this class can be created by several different ways:

- specifică CRS-ul după ID-ul său

  ```
  # PostGIS SRID 4326 is allocated for WGS84
  crs = QgsCoordinateReferenceSystem(4326, QgsCoordinateReferenceSystem.PostgisCrsId)
  ```

  QGIS folosește trei ID-uri diferite pentru fiecare sistem de referință:

  – `PostgisCrsId` — ID-uri folosite în interiorul bazei de date PostGIS.

  – `InternalCrsId` — ID-uri folosite în baza de date QGIS.

  – `EpsgCrsId` — ID-uri asignate de către organizația EPSG

  În cazul în care nu se specifică altfel în al doilea parametru, PostGIS SRID este utilizat în mod implicit.

- specifică CRS-ul prin well-known text (WKT)

  ```
  wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],'
         PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],'
         AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
  crs = QgsCoordinateReferenceSystem(wkt)
  ```

- create invalid CRS and then use one of the `create*()` functions to initialize it. In following example we use Proj4 string to initialize the projection

  ```
  crs = QgsCoordinateReferenceSystem()
  crs.createFromProj4("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")
  ```

It's wise to check whether creation (i.e. lookup in the database) of the CRS has been successful: `isValid()` must return `True`.

Note that for initialization of spatial reference systems QGIS needs to look up appropriate values in its internal database `srs.db`. Thus in case you create an independent application you need to set paths correctly with `QgsApplication.setPrefixPath()` otherwise it will fail to find the database. If you are running the commands from QGIS python console or developing a plugin you do not care: everything is already set up for you.

Accessing spatial reference system information

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.toProj4()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

7.2 Projections

You can do transformation between different spatial reference systems by using `QgsCoordinateTransform` class. The easiest way to use it is to create source and destination CRS and construct `QgsCoordinateTransform` instance with them. Then just repeatedly call `transform()` function to do the transformation. By default it does forward transformation, but it is capable to do also inverse transformation

```
crsSrc = QgsCoordinateReferenceSystem(4326)    # WGS 84
crsDest = QgsCoordinateReferenceSystem(32633)  # WGS 84 / UTM zone 33N
xform = QgsCoordinateTransform(crsSrc, crsDest)

# forward transformation: src -> dest
pt1 = xform.transform(QgsPoint(18,5))
print "Transformed point:", pt1

# inverse transformation: dest -> src
pt2 = xform.transform(pt1, QgsCoordinateTransform.ReverseTransform)
print "Transformed back:", pt2
```

CHAPTER 8

Using Map Canvas

- Încapsularea suportului de hartă
- Folosirea instrumentelor în suportul de hartă
- Benzile elastice și marcajele nodurilor
- Dezvoltarea instrumentelor personalizate pentru suportul de hartă
- Dezvoltarea elementelor personalizate pentru suportul de hartă

The Map canvas widget is probably the most important widget within QGIS because it shows the map composed from overlaid map layers and allows interaction with the map and layers. The canvas shows always a part of the map defined by the current canvas extent. The interaction is done through the use of **map tools**: there are tools for panning, zooming, identifying layers, measuring, vector editing and others. Similar to other graphics programs, there is always one tool active and the user can switch between the available tools.

Map canvas is implemented as `QgsMapCanvas` class in `qgis.gui` module. The implementation is based on the Qt Graphics View framework. This framework generally provides a surface and a view where custom graphics items are placed and user can interact with them. We will assume that you are familiar enough with Qt to understand the concepts of the graphics scene, view and items. If not, please make sure to read the overview of the framework.

Whenever the map has been panned, zoomed in/out (or some other action triggers a refresh), the map is rendered again within the current extent. The layers are rendered to an image (using `QgsMapRenderer` class) and that image is then displayed in the canvas. The graphics item (in terms of the Qt graphics view framework) responsible for showing the map is `QgsMapCanvasMap` class. This class also controls refreshing of the rendered map. Besides this item which acts as a background, there may be more **map canvas items**. Typical map canvas items are rubber bands (used for measuring, vector editing etc.) or vertex markers. The canvas items are usually used to give some visual feedback for map tools, for example, when creating a new polygon, the map tool creates a rubber band canvas item that shows the current shape of the polygon. All map canvas items are subclasses of `QgsMapCanvasItem` which adds some more functionality to the basic `QGraphicsItem` objects.

Pentru a rezuma, arhitectura suportului pentru hartă constă în trei concepte:

- suportul de hartă — pentru vizualizarea hărții
- map canvas items — additional items that can be displayed in map canvas
- map tools — for interaction with map canvas

8.1 Încapsularea suportului de hartă

Canevasul hărții este un widget ca orice alt widget Qt, așa că utilizarea este la fel de simplă ca și crearea și afișarea lui

```
canvas = QgsMapCanvas()
canvas.show()
```

This produces a standalone window with map canvas. It can be also embedded into an existing widget or window. When using .ui files and Qt Designer, place a `QWidget` on the form and promote it to a new class: set `QgsMapCanvas` as class name and set `qgis.gui` as header file. The `pyuic4` utility will take care of it. This is a very convenient way of embedding the canvas. The other possibility is to manually write the code to construct map canvas and other widgets (as children of a main window or dialog) and create a layout.

În mod implicit, canevasul hărții are un fundal negru și nu utilizează anti-zimțare. Pentru a seta fundalul alb și pentru a permite anti-zimțare pentru o redare mai bună

```
canvas.setCanvasColor(Qt.white)
canvas.enableAntiAliasing(True)
```

(In case you are wondering, `Qt` comes from `PyQt4.QtCore` module and `Qt.white` is one of the predefined `QColor` instances.)

Now it is time to add some map layers. We will first open a layer and add it to the map layer registry. Then we will set the canvas extent and set the list of layers for canvas

```
layer = QgsVectorLayer(path, name, provider)
if not layer.isValid():
  raise IOError, "Failed to open the layer"

# add layer to the registry
QgsMapLayerRegistry.instance().addMapLayer(layer)

# set extent to the extent of our layer
canvas.setExtent(layer.extent())

# set the map canvas layer set
canvas.setLayerSet([QgsMapCanvasLayer(layer)])
```

După executarea acestor comenzi, suportul ar trebui să arate stratul pe care le-ați încărcat.

8.2 Folosirea instrumentelor în suportul de hartă

The following example constructs a window that contains a map canvas and basic map tools for map panning and zooming. Actions are created for activation of each tool: panning is done with `QgsMapToolPan`, zooming in/out with a pair of `QgsMapToolZoom` instances. The actions are set as checkable and later assigned to the tools to allow automatic handling of checked/unchecked state of the actions – when a map tool gets activated, its action is marked as selected and the action of the previous map tool is deselected. The map tools are activated using `setMapTool()` method.

```
from qgis.gui import *
from PyQt4.QtGui import QAction, QMainWindow
from PyQt4.QtCore import SIGNAL, Qt, QString

class MyWnd(QMainWindow):
  def __init__(self, layer):
    QMainWindow.__init__(self)

    self.canvas = QgsMapCanvas()
    self.canvas.setCanvasColor(Qt.white)

    self.canvas.setExtent(layer.extent())
    self.canvas.setLayerSet([QgsMapCanvasLayer(layer)])

    self.setCentralWidget(self.canvas)

    actionZoomIn = QAction(QString("Zoom in"), self)
    actionZoomOut = QAction(QString("Zoom out"), self)
    actionPan = QAction(QString("Pan"), self)
```

```
    actionZoomIn.setCheckable(True)
    actionZoomOut.setCheckable(True)
    actionPan.setCheckable(True)

    self.connect(actionZoomIn, SIGNAL("triggered()"), self.zoomIn)
    self.connect(actionZoomOut, SIGNAL("triggered()"), self.zoomOut)
    self.connect(actionPan, SIGNAL("triggered()"), self.pan)

    self.toolbar = self.addToolBar("Canvas actions")
    self.toolbar.addAction(actionZoomIn)
    self.toolbar.addAction(actionZoomOut)
    self.toolbar.addAction(actionPan)

    # create the map tools
    self.toolPan = QgsMapToolPan(self.canvas)
    self.toolPan.setAction(actionPan)
    self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
    self.toolZoomIn.setAction(actionZoomIn)
    self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
    self.toolZoomOut.setAction(actionZoomOut)

    self.pan()

  def zoomIn(self):
    self.canvas.setMapTool(self.toolZoomIn)

  def zoomOut(self):
    self.canvas.setMapTool(self.toolZoomOut)

  def pan(self):
    self.canvas.setMapTool(self.toolPan)
```

You can put the above code to a file, e.g. `mywnd.py` and try it out in Python console within QGIS. This code will put the currently selected layer into newly created canvas

```
import mywnd
w = mywnd.MyWnd(qgis.utils.iface.activeLayer())
w.show()
```

Just make sure that the `mywnd.py` file is located within Python search path (`sys.path`). If it isn't, you can simply add it: `sys.path.insert(0, '/my/path')` — otherwise the import statement will fail, not finding the module.

8.3 Benzile elastice și marcajele nodurilor

To show some additional data on top of the map in canvas, use map canvas items. It is possible to create custom canvas item classes (covered below), however there are two useful canvas item classes for convenience: `QgsRubberBand` for drawing polylines or polygons, and `QgsVertexMarker` for drawing points. They both work with map coordinates, so the shape is moved/scaled automatically when the canvas is being panned or zoomed.

Pentru a afișa o polilinie

```
r = QgsRubberBand(canvas, False)  # False = not a polygon
points = [QgsPoint(-1, -1), QgsPoint(0, 1), QgsPoint(1, -1)]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Pentru a afișa un poligon

```
r = QgsRubberBand(canvas, True)  # True = a polygon
points = [[QgsPoint(-1, -1), QgsPoint(0, 1), QgsPoint(1, -1)]]
r.setToGeometry(QgsGeometry.fromPolygon(points), None)
```

Rețineți că punctele pentru poligon nu reprezintă o simplă listă: în fapt, aceasta este o listă de inele conținând inele liniare ale poligonului: primul inel reprezintă granița exterioară, în plus (opțional) inelele corespund găurilor din poligon.

Benzile elastice acceptă unele personalizări, și anume schimbarea culorii și a lățimii liniei

```
r.setColor(QColor(0, 0, 255))
r.setWidth(3)
```

The canvas items are bound to the canvas scene. To temporarily hide them (and show again, use the `hide()` and `show()` combo. To completely remove the item, you have to remove it from the scene of the canvas

```
canvas.scene().removeItem(r)
```

(În C + + este posibilă ștergerea doar a elementului, însă în Python `del r` ar șterge doar referința iar obiectul va exista în continuare, acesta fiind deținut de suport)

Rubber band can be also used for drawing points, however `QgsVertexMarker` class is better suited for this (`QgsRubberBand` would only draw a rectangle around the desired point). How to use the vertex marker

```
m = QgsVertexMarker(canvas)
m.setCenter(QgsPoint(0, 0))
```

This will draw a red cross on position [0,0]. It is possible to customize the icon type, size, color and pen width

```
m.setColor(QColor(0, 255, 0))
m.setIconSize(5)
m.setIconType(QgsVertexMarker.ICON_BOX) # or ICON_CROSS, ICON_X
m.setPenWidth(3)
```

For temporary hiding of vertex markers and removing them from canvas, the same applies as for the rubber bands.

8.4 Dezvoltarea instrumentelor personalizate pentru suportul de hartă

You can write your custom tools, to implement a custom behaviour to actions performed by users on the canvas.

Map tools should inherit from the `QgsMapTool` class or any derived class, and selected as active tools in the canvas using the `setMapTool()` method as we have already seen.

Iată un exemplu de instrument pentru hartă, care permite definirea unei limite dreptunghiulare, făcând clic și trăgând cursorul mouse-ului pe canevas. După ce este definit dreptunghiul, coordonatele sale sunt afișate în consolă. Se utilizează elementele benzii elastice descrise mai înainte, pentru a arăta dreptunghiul selectat, așa cum a fost definit.

```
class RectangleMapTool(QgsMapToolEmitPoint):
  def __init__(self, canvas):
      self.canvas = canvas
      QgsMapToolEmitPoint.__init__(self, self.canvas)
      self.rubberBand = QgsRubberBand(self.canvas, QGis.Polygon)
      self.rubberBand.setColor(Qt.red)
      self.rubberBand.setWidth(1)
      self.reset()

  def reset(self):
      self.startPoint = self.endPoint = None
      self.isEmittingPoint = False
      self.rubberBand.reset(QGis.Polygon)
```

```
def canvasPressEvent(self, e):
    self.startPoint = self.toMapCoordinates(e.pos())
    self.endPoint = self.startPoint
    self.isEmittingPoint = True
    self.showRect(self.startPoint, self.endPoint)

def canvasReleaseEvent(self, e):
    self.isEmittingPoint = False
    r = self.rectangle()
    if r is not None:
      print "Rectangle:", r.xMinimum(), r.yMinimum(), r.xMaximum(), r.yMaximum()

def canvasMoveEvent(self, e):
    if not self.isEmittingPoint:
      return

    self.endPoint = self.toMapCoordinates(e.pos())
    self.showRect(self.startPoint, self.endPoint)

def showRect(self, startPoint, endPoint):
    self.rubberBand.reset(QGis.Polygon)
    if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
      return

    point1 = QgsPoint(startPoint.x(), startPoint.y())
    point2 = QgsPoint(startPoint.x(), endPoint.y())
    point3 = QgsPoint(endPoint.x(), endPoint.y())
    point4 = QgsPoint(endPoint.x(), startPoint.y())

    self.rubberBand.addPoint(point1, False)
    self.rubberBand.addPoint(point2, False)
    self.rubberBand.addPoint(point3, False)
    self.rubberBand.addPoint(point4, True)    # true to update canvas
    self.rubberBand.show()

def rectangle(self):
    if self.startPoint is None or self.endPoint is None:
      return None
    elif self.startPoint.x() == self.endPoint.x() or self.startPoint.y() == self.endPoint.y():
      return None

    return QgsRectangle(self.startPoint, self.endPoint)

def deactivate(self):
    super(RectangleMapTool, self).deactivate()
    self.emit(SIGNAL("deactivated()"))
```

8.5 Dezvoltarea elementelor personalizate pentru suportul de hartă

DE EFECTUAT: how to create a map canvas item

```
import sys
from qgis.core import QgsApplication
from qgis.gui import QgsMapCanvas

def init():
  a = QgsApplication(sys.argv, True)
  QgsApplication.setPrefixPath('/home/martin/qgis/inst', True)
```

```
  QgsApplication.initQgis()
  return a

def show_canvas(app):
  canvas = QgsMapCanvas()
  canvas.show()
  app.exec_()
app = init()
show_canvas(app)
```

Randarea hărților și imprimarea

- Randarea simplă
- Randarea straturilor cu diferite CRS-uri
- Output using Map Composer
  - Output to a raster image
  - Output to PDF

There are generally two approaches when input data should be rendered as a map: either do it quick way using `QgsMapRenderer` or produce more fine-tuned output by composing the map with `QgsComposition` class and friends.

9.1 Randarea simplă

Render some layers using `QgsMapRenderer` — create destination paint device (`QImage`, `QPainter` etc.), set up layer set, extent, output size and do the rendering

```
# create image
img = QImage(QSize(800, 600), QImage.Format_ARGB32_Premultiplied)

# set image's background color
color = QColor(255, 255, 255)
img.fill(color.rgb())

# create painter
p = QPainter()
p.begin(img)
p.setRenderHint(QPainter.Antialiasing)

render = QgsMapRenderer()

# set layer set
lst = [layer.id()]  # add ID of every layer
render.setLayerSet(lst)

# set extent
rect = QgsRectangle(render.fullExtent())
rect.scale(1.1)
render.setExtent(rect)

# set output size
render.setOutputSize(img.size(), img.logicalDpiX())

# do the rendering
render.render(p)
```

```
p.end()

# save image
img.save("render.png","png")
```

9.2 Randarea straturilor cu diferite CRS-uri

If you have more than one layer and they have a different CRS, the simple example above will probably not work: to get the right values from the extent calculations you have to explicitly set the destination CRS and enable OTF reprojection as in the example below (only the renderer configuration part is reported)

```
...
# set layer set
layers = QgsMapLayerRegistry.instance().mapLayers()
lst = layers.keys()
render.setLayerSet(lst)

# Set destination CRS to match the CRS of the first layer
render.setDestinationCrs(layers.values()[0].crs())
# Enable OTF reprojection
render.setProjectionsEnabled(True)
...
```

9.3 Output using Map Composer

Map composer is a very handy tool if you would like to do a more sophisticated output than the simple rendering shown above. Using the composer it is possible to create complex map layouts consisting of map views, labels, legend, tables and other elements that are usually present on paper maps. The layouts can be then exported to PDF, raster images or directly printed on a printer.

The composer consists of a bunch of classes. They all belong to the core library. QGIS application has a convenient GUI for placement of the elements, though it is not available in the GUI library. If you are not familiar with Qt Graphics View framework, then you are encouraged to check the documentation now, because the composer is based on it. Also check the Python documentation of the implementation of QGraphicView.

The central class of the composer is `QgsComposition` which is derived from `QGraphicsScene`. Let us create one

```
mapRenderer = iface.mapCanvas().mapRenderer()
c = QgsComposition(mapRenderer)
c.setPlotStyle(QgsComposition.Print)
```

Note that the composition takes an instance of `QgsMapRenderer`. In the code we expect we are running within QGIS application and thus use the map renderer from map canvas. The composition uses various parameters from the map renderer, most importantly the default set of map layers and the current extent. When using composer in a standalone application, you can create your own map renderer instance the same way as shown in the section above and pass it to the composition.

It is possible to add various elements (map, label, ...) to the composition — these elements have to be descendants of `QgsComposerItem` class. Currently supported items are:

- harta — acest element indică bibliotecilor unde să pună harta. Vom crea o hartă și o vom întinde peste întreaga dimensiune a hârtiei

  ```
  x, y = 0, 0
  w, h = c.paperWidth(), c.paperHeight()
  composerMap = QgsComposerMap(c, x ,y, w, h)
  c.addItem(composerMap)
  ```

- eticheta — permite afișarea textelor. Este posibilă modificarea fontului, culoarea, alinierea și marginea

  ```
  composerLabel = QgsComposerLabel(c)
  composerLabel.setText("Hello world")
  composerLabel.adjustSizeToText()
  c.addItem(composerLabel)
  ```

- legenda

  ```
  legend = QgsComposerLegend(c)
  legend.model().setLayerSet(mapRenderer.layerSet())
  c.addItem(legend)
  ```

- scara grafică

  ```
  item = QgsComposerScaleBar(c)
  item.setStyle('Numeric') # optionally modify the style
  item.setComposerMap(composerMap)
  item.applyDefaultSize()
  c.addItem(item)
  ```

- săgeată
- imagine
- basic shape
- nodes based shape

  ```
  polygon = QPolygonF()
  polygon.append(QPointF(0.0, 0.0))
  polygon.append(QPointF(100.0, 0.0))
  polygon.append(QPointF(200.0, 100.0))
  polygon.append(QPointF(100.0, 200.0))

  composerPolygon = QgsComposerPolygon(polygon, c)
  c.addItem(composerPolygon)

  props = {}
  props["color"] = "green"
  props["style"] = "solid"
  props["style_border"] = "solid"
  props["color_border"] = "black"
  props["width_border"] = "10.0"
  props["joinstyle"] = "miter"

  style = QgsFillSymbolV2.createSimple(props)
  composerPolygon.setPolygonStyleSymbol(style)
  ```

- tabelă

By default the newly created composer items have zero position (top left corner of the page) and zero size. The position and size are always measured in millimeters

```
# set label 1cm from the top and 2cm from the left of the page
composerLabel.setItemPosition(20, 10)
# set both label's position and size (width 10cm, height 3cm)
composerLabel.setItemPosition(20, 10, 100, 30)
```

A frame is drawn around each item by default. How to remove the frame

```
composerLabel.setFrame(False)
```

Besides creating the composer items by hand, QGIS has support for composer templates which are essentially compositions with all their items saved to a .qpt file (with XML syntax). Unfortunately this functionality is not yet available in the API.

Once the composition is ready (the composer items have been created and added to the composition), we can proceed to produce a raster and/or vector output.

The default output settings for composition are page size A4 and resolution 300 DPI. You can change them if necessary. The paper size is specified in millimeters

```
c.setPaperSize(width, height)
c.setPrintResolution(dpi)
```

9.3.1 Output to a raster image

The following code fragment shows how to render a composition to a raster image

```
dpi = c.printResolution()
dpmm = dpi / 25.4
width = int(dpmm * c.paperWidth())
height = int(dpmm * c.paperHeight())

# create output image and initialize it
image = QImage(QSize(width, height), QImage.Format_ARGB32)
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)

# render the composition
imagePainter = QPainter(image)
c.renderPage( imagePainter, 0 )
imagePainter.end()

image.save("out.png", "png")
```

9.3.2 Output to PDF

The following code fragment renders a composition to a PDF file

```
printer = QPrinter()
printer.setOutputFormat(QPrinter.PdfFormat)
printer.setOutputFileName("out.pdf")
printer.setPaperSize(QSizeF(c.paperWidth(), c.paperHeight()), QPrinter.Millimeter)
printer.setFullPage(True)
printer.setColorMode(QPrinter.Color)
printer.setResolution(c.printResolution())

pdfPainter = QPainter(printer)
paperRectMM = printer.pageRect(QPrinter.Millimeter)
paperRectPixel = printer.pageRect(QPrinter.DevicePixel)
c.render(pdfPainter, paperRectPixel, paperRectMM)
pdfPainter.end()
```

Expresii, filtrarea și calculul valorilor

- Parsarea expresiilor
- Evaluarea expresiilor
  - Expresii de bază
  - Expresii cu entități
  - Tratarea erorilor
- Exemple

QGIS are un oarecare suport pentru analiza expresiilor, cum ar fi SQL. Doar un mic subset al sintaxei SQL este acceptat. Expresiile pot fi evaluate fie ca predicate booleene (returnând True sau False), fie ca funcții (care întorc o valoare scalară). Parcurgeți *vector_expressions* din Manualul Utilizatorului, pentru o listă completă a funcțiilor disponibile.

Trei tipuri de bază sunt acceptate:

- — număr atât numere întregi cât și numere zecimale, de exemplu, `123`, `3.14`
- șir — acesta trebuie să fie cuprins între ghilimele simple: `'hello world'`
- referință către coloană — atunci când se evaluează, referința este substituită cu valoarea reală a câmpului. Numele nu sunt protejate.

Următoarele operațiuni sunt disponibile:

- operatori aritmetici: `+`, `-`, `*`, `/`, `^`
- paranteze: pentru forțarea priorității operatorului: `(1 + 1) * 3`
- plus și minus unari: `-12`, `+5`
- funcții matematice: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- funcții de conversie: `to_int`, `to_real`, `to_string`, `to_date`
- funcții geometrice: `$area`, `$length`
- funcții de manipulare a geometriei: `$x`, `$y`, `$geometry`, `num_geometries`, `centroid`

Și următoarele predicate sunt suportate:

- comparație: `=`, `!=`, `>`, `>=`, `<`, `<=`
- potrivirea paternurilor: `LIKE` (folosind % și _), `~` (expresii regulate)
- predicate logice: `AND`, `OR`, `NOT`
- verificarea valorii NULL: `IS NULL`, `IS NOT NULL`

Exemple de predicate:

- `1 + 2 = 3`
- `sin(angle) > 0`

- `'Hello' LIKE 'He%'`
- `(x > 10 AND y > 10) OR z = 0`

Exemple de expresii scalare:

- `2 ^ 10`
- `sqrt(val)`
- `$length + 1`

10.1 Parsarea expresiilor

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> exp.hasParserError()
False
>>> exp = QgsExpression('1 + 1 = ')
>>> exp.hasParserError()
True
>>> exp.parserErrorString()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

10.2 Evaluarea expresiilor

10.2.1 Expresii de bază

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> value = exp.evaluate()
>>> value
1
```

10.2.2 Expresii cu entități

The following example will evaluate the given expression against a feature. "Column" is the name of the field in the layer.

```
>>> exp = QgsExpression('Column = 99')
>>> value = exp.evaluate(feature, layer.pendingFields())
>>> bool(value)
True
```

You can also use `QgsExpression.prepare()` if you need check more than one feature. Using `QgsExpression.prepare()` will increase the speed that evaluate takes to run.

```
>>> exp = QgsExpression('Column = 99')
>>> exp.prepare(layer.pendingFields())
>>> value = exp.evaluate(feature)
>>> bool(value)
True
```

10.2.3 Tratarea erorilor

```
exp = QgsExpression("1 + 1 = 2 ")
if exp.hasParserError():
  raise Exception(exp.parserErrorString())
```

```
value = exp.evaluate()
if exp.hasEvalError():
  raise ValueError(exp.evalErrorString())

print value
```

10.3 Exemple

Următorul exemplu poate fi folosit pentru a filtra un strat și pentru a întoarce orice entitate care se potrivește unui predicat.

```
def where(layer, exp):
  print "Where"
  exp = QgsExpression(exp)
  if exp.hasParserError():
    raise Exception(exp.parserErrorString())
  exp.prepare(layer.pendingFields())
  for feature in layer.getFeatures():
    value = exp.evaluate(feature)
    if exp.hasEvalError():
      raise ValueError(exp.evalErrorString())
    if bool(value):
      yield feature

layer = qgis.utils.iface.activeLayer()
for f in where(layer, 'Test > 1.0'):
  print f + " Matches expression"
```

CHAPTER 11

Citirea și stocarea setărilor

De multe ori, pentru un plugin, este utilă salvarea unor variabile, astfel încât utilizatorul să nu trebuiască să le reintroducă sau să le reselecteze, la fiecare rulare a plugin-ului.

Aceste variabile pot fi salvate cu ajutorul Qt și QGIS API. Pentru fiecare variabilă ar trebui să alegeți o cheie care va fi folosită pentru a accesa variabila — pentru culoarea preferată a utilizatorului ați putea folosi o cheie de genul "culoare_favorită" sau orice alt șir semnificativ. Este recomandabil să folosiți o oarecare logică în denumirea cheilor.

We can make difference between several types of settings:

- **global settings** — they are bound to the user at particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. This functionality is provided directly by Qt framework by the means of `QSettings` class. By default, this class stores settings in system's "native" way of storing settings, that is — registry (on Windows), .plist file (on macOS) or .ini file (on Unix). The QSettings documentation is comprehensive, so we will provide just a simple example

  ```
  def store():
    s = QSettings()
    s.setValue("myplugin/mytext", "hello world")
    s.setValue("myplugin/myint",  10)
    s.setValue("myplugin/myreal", 3.14)

  def read():
    s = QSettings()
    mytext = s.value("myplugin/mytext", "default text")
    myint  = s.value("myplugin/myint", 123)
    myreal = s.value("myplugin/myreal", 2.71)
  ```

  The second parameter of the `value()` method is optional and specifies the default value if there is no previous value set for the passed setting name.

- **project settings** — vary between different projects and therefore they are connected with a project file. Map canvas background color or destination coordinate reference system (CRS) are examples — white background and WGS84 might be suitable for one project, while yellow background and UTM projection are better for another one. An example of usage follows

  ```
  proj = QgsProject.instance()

  # store values
  proj.writeEntry("myplugin", "mytext", "hello world")
  proj.writeEntry("myplugin", "myint", 10)
  proj.writeEntry("myplugin", "mydouble", 0.01)
  proj.writeEntry("myplugin", "mybool", True)

  # read values
  mytext = proj.readEntry("myplugin", "mytext", "default text")[0]
  myint = proj.readNumEntry("myplugin", "myint", 123)[0]
  ```

As you can see, the `writeEntry()` method is used for all data types, but several methods exist for reading the setting value back, and the corresponding one has to be selected for each data type.

- **map layer settings** — these settings are related to a particular instance of a map layer with a project. They are *not* connected with underlying data source of a layer, so if you create two map layer instances of one shapefile, they will not share the settings. The settings are stored in project file, so if the user opens the project again, the layer-related settings will be there again. This functionality has been added in QGIS v1.4. The API is similar to QSettings — it takes and returns QVariant instances

```
# save a value
layer.setCustomProperty("mytext", "hello world")

# read the value again
mytext = layer.customProperty("mytext", "default text")
```

CHAPTER 12

Comunicarea cu utilizatorul

- Showing messages. The `QgsMessageBar` class
- Afișarea progresului
- Jurnalizare

Această secțiune prezintă câteva metode și elemente care ar trebui să fie utilizate pentru a comunica cu utilizatorul, în scopul menținerii coerenței interfaței cu utilizatorul.

12.1 Showing messages. The `QgsMessageBar` class

Folosirea casetelor de mesaje poate fi o idee rea, din punctul de vedere al experienței utilizatorului. Pentru a arăta o mică linie de informații sau un mesaj de avertizare/eroare, bara QGIS de mesaje este, de obicei, o opțiune mai bună.

Folosind referința către obiectul interfeței QGIS, puteți afișa un text în bara de mesaje, cu ajutorul următorului cod

```
from qgis.gui import QgsMessageBar
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't do that", level=QgsMe
```

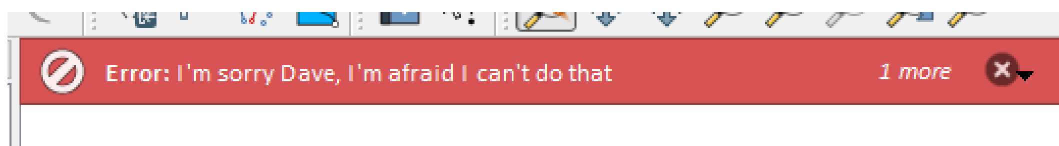


Figure 12.1: Bara de mesaje a QGIS

Puteți seta o durată, pentru afișarea pentru o perioadă limitată de timp

```
iface.messageBar().pushMessage("Error", "Ooops, the plugin is not working as it should", level=Qg
```

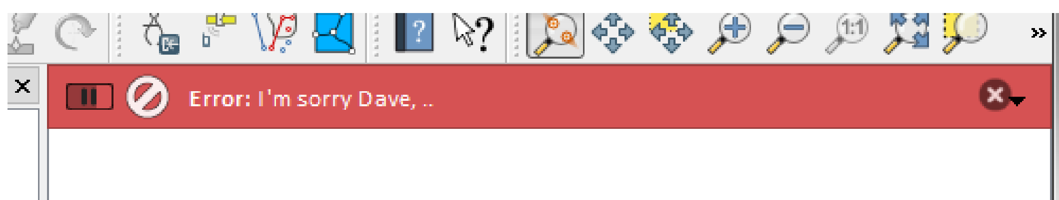


Figure 12.2: Bara de mesaje a QGIS, cu cronometru

The examples above show an error bar, but the `level` parameter can be used to creating warning messages or info messages, using the `QgsMessageBar.WARNING` and `QgsMessageBar.INFO` constants respectively.

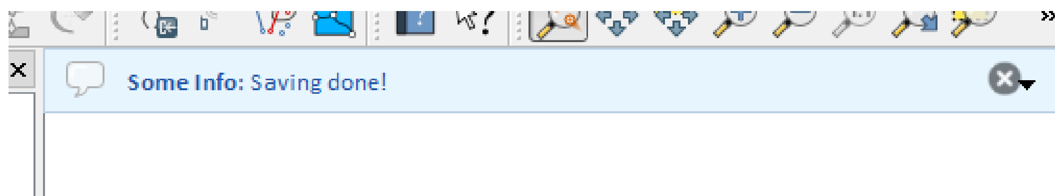


Figure 12.3: Bara de mesaje a QGIS (info)

Widget-urile pot fi adăugate la bara de mesaje, cum ar fi, de exemplu, un buton pentru afișarea mai multor informații

```
def showError():
    pass

widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
button = QPushButton(widget)
button.setText("Show Me")
button.pressed.connect(showError)
widget.layout().addWidget(button)
iface.messageBar().pushWidget(widget, QgsMessageBar.WARNING)
```

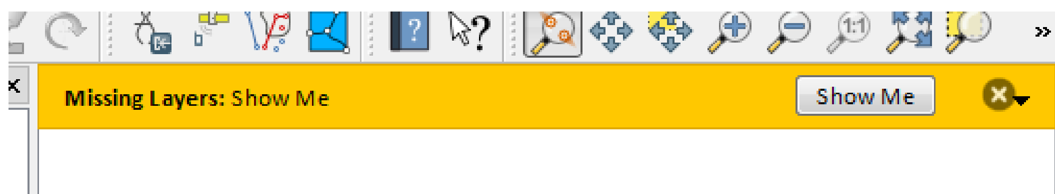


Figure 12.4: Bara de mesaje a QGIS, cu un buton

Puteți utiliza o bară de mesaje chiar și în propria fereastră de dialog, în loc să apelați la o casetă de text, sau să arătați mesajul în fereastra principală a QGIS

```
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0, 0, 0, 0)
        self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
        self.buttonbox.accepted.connect(self.run)
        self.layout().addWidget(self.buttonbox, 0, 0, 2, 1)
        self.layout().addWidget(self.bar, 0, 0, 1, 1)

    def run(self):
        self.bar.pushMessage("Hello", "World", level=QgsMessageBar.INFO)
```

12.2 Afișarea progresului

Barele de progres pot fi, de asemenea, incluse în bara de mesaje QGIS, din moment ce, așa cum am văzut, aceasta acceptă widget-uri. Iată un exemplu pe care îl puteți încerca în consolă.

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
```

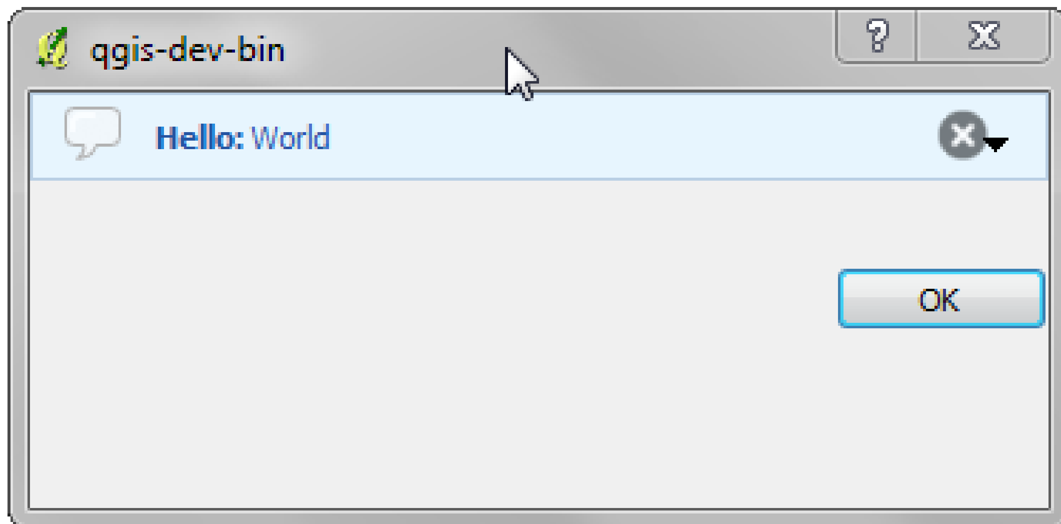


Figure 12.5: Bara de mesaje a QGIS, într-o fereastră de dialog

```
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
    time.sleep(1)
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

Also, you can use the built-in status bar to report progress, as in the next example

```
count = layers.featureCount()
for i, feature in enumerate(features):
    #do something time-consuming here
    ...
    percent = i / float(count) * 100
    iface.mainWindow().statusBar().showMessage("Processed {} %".format(int(percent)))
iface.mainWindow().statusBar().clearMessage()
```

12.3 Jurnalizare

Puteți utiliza sistemul de jurnalizare al QGIS, pentru a salva toate informațiile pe care doriți să le înregistrați, cu privire la execuția codului dvs.

```
# You can optionally pass a 'tag' and a 'level' parameters
QgsMessageLog.logMessage("Your plugin code has been executed correctly", 'MyPlugin', QgsMessageLo
QgsMessageLog.logMessage("Your plugin code might have some problems", level=QgsMessageLog.WARNING
QgsMessageLog.logMessage("Your plugin code has crashed!", level=QgsMessageLog.CRITICAL)
```

CHAPTER 13

Dezvoltarea plugin-urilor Python

- Scrierea unui plugin
  - Fișierele Plugin-ului
- Conținutul Plugin-ului
  - |Metadatele plugin-ului
  - __init__.py
  - mainPlugin.py
  - Fișier de resurse
- Documentație
- Traducerea
  - Cerințe software
  - Fișierele și directorul
    - fişier .pro
    - fişier .ts
    - fişier .qm
  - Translate using Makefile
  - Încărcarea plugin-ului

Este posibil să se creeze plugin-uri în limbajul de programare Python. În comparație cu plugin-urile clasice scrise în C++ acestea ar trebui să fie mai ușor de scris, de înțeles, de menținut și de distribuit, din cauza naturii dinamice a limbajului Python.

Plugin-urile Python sunt listate, împreună cu plugin-urile C++, în managerul de plugin-uri QGIS. Ele sunt căutate în aceste căi:

- UNIX/Mac: `~/.qgis2/python/plugins` și `(qgis_prefix)/share/qgis/python/plugins`
- Windows: `~/.qgis2/python/plugins` și `(qgis_prefix)/python/plugins`

Directorul de casă (notat `~`) din Windows este, de obicei, ceva de genul `C:\Documents and Settings\(user)` (în Windows XP sau versiunile anterioare) sau `C:\Users\(user)`. Deoarece QGIS utilizează Python 2.7, subdirectoarele acestor căi trebuie să conțină un fișier __init__.py, pentru a fi considerate pachete Python care pot fi importate ca plugin-uri.

Note: Prin atașarea *QGIS_PLUGINPATH* căii unui director existent, puteți vedea această cale în lista căilor de căutare pentru plugin-uri.

Pași:

1. *Ideea*: Conturați-vă o idee despre ceea ce vreți să faceți cu noul plugin QGIS. De ce-l faceți? Ce problemă doriți să rezolve? Există deja un alt plugin pentru această problemă?
2. *Creare fișiere*: Se creează fișierele descrise în continuare. Un punct de plecare (`__init.py__`). Completați *|Metadatele plugin-ului* (`metadata.txt`). Un corp python principal al plugin-ului (`mainplugin.py`). Un formular în QT-Designer (`form.ui`), cu al său `resources.qrc`.

3. *Scrierea codului*: Scrieți codul în interiorul `mainplugin.py`
4. *Testul*: Închideți și re-deschideți QGIS, apoi importați-l din nou. Verificați dacă totul este în regulă.
5. *Publicare*: Se publică plugin-ul în depozitul QGIS sau vă faceți propriul depozit ca un "arsenal" de "arme GIS" personale

13.1 Scrierea unui plugin

De la introducerea plugin-urilor Python în QGIS, a apărut un număr de plugin-uri - pe pagina wiki a Depozitelor de Plugin-uri puteți găsi unele dintre ele, le puteți utiliza sursa pentru a afla mai multe despre programarea în PyQGIS sau să aflați dacă nu cumva duplicați efortul de dezvoltare. Echipa QGIS menține, de asemenea, un *Depozitul oficial al plugin-urilor python*. Sunteți gata de a crea un plugin, dar nu aveți nici o idee despre cum ați putea începe? În pagina wiki cu idei de plugin-uri Python sunt enumerate doleanțele comunității!

13.1.1 Fișierele Plugin-ului

Iată structura de directoare a exemplului nostru de plugin

```
PYTHON_PLUGINS_PATH/
  MyPlugin/
    __init__.py    --> *required*
    mainPlugin.py  --> *required*
    metadata.txt   --> *required*
    resources.qrc  --> *likely useful*
    resources.py   --> *compiled version, likely useful*
    form.ui        --> *likely useful*
    form.py        --> *compiled version, likely useful*
```

Care este semnificația fișierelor:

- `__init__.py` = Punctul de plecare al plugin-ului. Acesta trebuie să aibă metoda `classFactory()` și poate avea orice alt cod de inițializare.
- `mainPlugin.py` = Principalul cod lucrativ al plugin-ului. Conține toate informațiile cu privire la acțiunile plugin-ului și ale codului principal.
- `resources.qrc` = Documentul .xml creat de Qt Designer. Conține căi relative la resursele formelor.
- `resources.py` = Traducerea în Python a fișierului .qrc, descris mai sus.
- `form.ui` = GUI-ul creat de Qt Designer.
- `form.py` = Traducerea în Python a fișierului form.ui, descris mai sus.
- `metadata.txt` = Necesar pentru QGIS >= 1.8.0. Conține informații generale, versiunea, numele și alte metadate utilizate de către site-ul de plugin-uri și de către infrastructura plugin-ului. Începând cu QGIS 2.0 metadatele din `__init__.py` nu mai sunt acceptate, iar `metadata.txt` este necesar.

Aici este o modalitate on-line, automată, de creare a fișierelor de bază (carcase) pentru un plugin tipic QGIS Python.

De asemenea, există un plugin QGIS numit Plugin Builder care creează un șablon de plugin QGIS și nu are nevoie de conexiune la internet. Aceasta este opțiunea recomandată, atât timp cât produce surse compatibile 2.0.

> **Warning:** Dacă aveți de gând să încărcați plugin-ul la *Depozitul oficial al plugin-urilor python* trebuie să verificați dacă plugin-ul urmează anumite reguli suplimentare, necesare pentru plugin-ul *Validare*

13.2 Conținutul Plugin-ului

Aici puteți găsi informații și exemple despre ceea ce să adăugați în fiecare dintre fișierele din structura de fișiere descrisă mai sus.

13.2.1 Metadatele plugin-ului

În primul rând, managerul de plugin-uri are nevoie de preluarea câtorva informații de bază despre plugin, cum ar fi numele, descrierea etc. Fișierul `metadata.txt` este locul potrivit pentru a reține această informație.

Important: Toate metadatele trebuie să fie în codificarea UTF-8.

Numele metadatei	Obligatoriu	Note
nume	True	un șir scurt conținând numele pluginului
qgisMinimumVersion	True	notație cu punct a versiunii minime QGIS
qgisMaximumVersion	False	notație cu punct a versiunii maxime QGIS
descriere	True	scurt text care descrie plugin-ul, HTML nefiind permis
despre	True	text mai lung care descrie plugin-ul în detalii, HTML nefiind permis
versiune	True	scurt șir cu versiunea notată cu punct
autor	True	nume autor
email	True	e-mail-ul autorului, care nu este afișat în managerul de plugin-uri QGIS sau pe site-ul web, fiind vizibile doar pentru utilizatorii autentificați, cum ar fi alți autori de plugin-uri și administratorii site-ului
jurnalul schimbărilor	False	șir, poate fi pe mai multe linii, HTML nefiind permis
experimental	False	semnalizator boolean, *True* sau *False*
învechit	False	semnalizator boolean, *True* sau *False*, se aplică întregului plugin și nu doar la versiunea încărcată
etichete	False	o listă de valori separate prin virgulă, spațiile fiind permise în interiorul etichetelor individuale
pagina de casă	False	o adresă URL validă indicând spre pagina plugin-ului dvs.
depozit	True	o adresă URL validă pentru depozitul de cod sursă
tracker	False	o adresă validă pentru bilete și rapoartare de erori
pictogramă	False	un nume de fișier sau o cale relativă (relativă la directorul de bază al pachetului comprimat al plugin-ului) pentru o imagine adecvată pentru web (PNG, JPEG)
categorie	False	una din valorile *Raster*, *Vector*, *Bază de date* și *Web*

În mod implicit, plugin-urile sunt plasate în meniul *Plugin-uri* (vom vedea în secțiunea următoare cum se poate adăuga o intrare de meniu pentru plugin-ul dvs.), dar ele pot fi, de asemenea, plasate și în meniurile *Raster*, *Vector*, *Database* și *Web*.

Există o "categorie" de intrare de metadate corespunzătoare pentru a preciza că, astfel, plugin-ul poate fi clasificat în consecință. Această intrare de metadate este folosită ca indiciu pentru utilizatori și le spune unde (în care meniu) se poate găsi plugin-ul. Valorile permise pentru "categorie" sunt: Vector, Raster, Baza de date sau Web. De exemplu, dacă plugin-ul va fi disponibil din meniul *Raster*, atunci adăugați-l în `metadata.txt`

```
category=Raster
```

Note: În cazul în care valoarea *qgisMaximumVersion* este vidă, ea va fi setată automat la versiunea majoră plus *.99* încărcată în depozitul *Depozitul oficial al plugin-urilor python*.

Un exemplu pentru acest metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
; end of mandatory metadata

; start of optional metadata
category=Raster
changelog=The changelog lists the plugin versions
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

; Tags are in comma separated value format, spaces are allowed within the
; tag name.
; Tags should be in English language. Please also check for existing tags and
; synonyms before creating a new one.
tags=wkt,raster,hello world

; these metadata can be empty, they will eventually become mandatory.
homepage=http://www.itopen.it
icon=icon.png

; experimental flag (applies to the single version)
experimental=True

; deprecated flag (applies to the whole plugin and not only to the uploaded version)
deprecated=False

; if empty, it will be automatically set to major version + .99
qgisMaximumVersion=2.0
```

13.2.2 __init__.py

Acest fișier este necesar pentru sistemul de import al Python. De asemenea, QGIS necesită ca acest fișier să conțină o funcție `classFactory()`, care este apelată atunci când plugin-ul se încarcă în QGIS. Acesta primește referirea la o instanță a `QgisInterface` și trebuie să returneze instanța clasei plugin-ului din `mainplugin.py` — în cazul nostru numindu-se `TestPlugin` (a se vedea mai jos). Acesta este modul în care `__init__.py` ar trebui să arate

```
def classFactory(iface):
  from mainPlugin import TestPlugin
```

```
  return TestPlugin(iface)

## any other initialisation needed
```

13.2.3 mainPlugin.py

Aici este locul în care se întâmplă magia, și iată rezultatul acesteia: (de exemplu `mainPlugin.py`)

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

  def __init__(self, iface):
    # save reference to the QGIS interface
    self.iface = iface

  def initGui(self):
    # create action that will start plugin configuration
    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWin
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

    # add toolbar button and menu item
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

    # connect to signal renderComplete which is emitted when canvas
    # rendering is done
    QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest

  def unload(self):
    # remove the plugin menu item and icon
    self.iface.removePluginMenu("&Test plugins", self.action)
    self.iface.removeToolBarIcon(self.action)

    # disconnect form signal of the canvas
    QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderT

  def run(self):
    # create and show a configuration dialog or something similar
    print "TestPlugin: run called!"

  def renderTest(self, painter):
    # use painter for drawing to map canvas
    print "TestPlugin: renderTest called!"
```

Singurele funcții care trebuie să existe în fișierul sursă al plugin-ului principal (de exemplu `mainPlugin.py`) sunt:

- `__init__` –> care oferă acces la interfața QGIS
- `initGui()` –> apelat atunci când plugin-ul este încărcat
- `unload()` –> apelat atunci când plugin-ul este descărcat

Puteți vedea că în exemplul de mai sus se folosește `addPluginToMenu()`. Aceasta va adăuga acțiunea meniului corespunzător la meniul *Plugins*. Există metode alternative pentru a adăuga acțiunea la un alt meniu. Iată o listă a acestor metode:

- `addPluginToRasterMenu()`
- `addPluginToVectorMenu()`
- `addPluginToDatabaseMenu()`
- `addPluginToWebMenu()`

Toate acestea au aceeași sintaxă ca metoda `addPluginToMenu()`.

Adăugarea unui meniu la plugin-ul dvs. printr-una din metoldele predefinite este recomandată pentru a păstra coerența în stilul de organizare a plugin-urilor. Cu toate acestea, puteți adăuga grupul dvs. de meniuri personalizate direct în bara de meniu, așa cum demonstrează următorul exemplu :

```
def initGui(self):
    self.menu = QMenu(self.iface.mainWindow())
    self.menu.setObjectName("testMenu")
    self.menu.setTitle("MyMenu")

    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWin
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)
    self.menu.addAction(self.action)

    menuBar = self.iface.mainWindow().menuBar()
    menuBar.insertMenu(self.iface.firstRightStandardMenu().menuAction(), self.menu)

def unload(self):
    self.menu.deleteLater()
```

Nu uitați să stabiliți pentru `QAction` și `QMenu objectName` un nume specific plugin-ului dvs, astfel încât acesta să poată fi personalizat.

13.2.4 Fișier de resurse

Puteți vedea că în `initGui()` am folosit o pictogramă din fișierul de resurse (denumit `resources.qrc`, în cazul nostru)

```
<RCC>
  <qresource prefix="/plugins/testplug" >
     <file>icon.png</file>
  </qresource>
</RCC>
```

Este bine să folosiți un prefix pentru a evita coliziunile cu alte plugin-uri sau cu oricare alte părți ale QGIS, în caz contrar, ați putea obține rezultate nedorite. Trebuie doar să generați un fișier Python care va conține resursele. Acest lucru se face cu comanda: **:comanda:‘pyrcc4‘**

```
pyrcc4 -o resources.py resources.qrc
```

Note: În mediile Windows, încercând să rulați **pyrcc4** din Linia de Comandă sau din Powershell va rezulta, probabil, eroarea "Windows cannot access the specified device, path, or file [...]". Cea mai simplă soluție constă, probabil, în utilizarea aplicației OSGeo4W, dar dacă puteți modifica variabila de mediu PATH, sau să specificați calea către executabilul explicit, ar trebui să-l găsiți în `<Your QGIS Install Directory>\bin\pyrcc4.exe`.

Și asta e tot ... nimic complicat :)

Dacă ați făcut totul corect, ar trebui să găsiți și să încărcați plugin-ul în managerul de plugin-uri și să vedeți un mesaj în consolă, atunci când este selectat meniul adecvat sau pictograma din bara de instrumente.

Când lucrați la un plug-in real, este înțelept să stocați plugin-ul într-un alt director (de lucru), și să creați un fișier make care va genera UI + fișierele de resurse și să instalați plugin-ul în instalarea QGIS.

13.3 Documentație

Documentația pentru plugin poate fi scrisă ca fișiere HTML. Modulul `qgis.utils` oferă o funcție, `showPluginHelp()`, care se va deschide navigatorul de fișiere, în același mod ca și altă fereastră de ajutor QGIS.

Funcția `showPluginHelp()` caută fișierele de ajutor în același director ca și modulul care îl apelează. Acesta va căuta, la rândul său, în `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` și `index.html`, afișând ceea ce găsește mai întâi. Aici, `ll_cc` reprezintă limba în care se afișează QGIS. Acest lucru permite multiplelor traduceri ale documentelor să fie incluse în plugin.

Funcția `showPluginHelp()` poate lua, de asemenea, parametrii packageName, care identifică plugin-ul specific pentru care va fi afișat ajutorul, numele de fișier, care poate înlocui 'index' în numele fișierelor în care se caută, și secțiunea, care este numele unei ancore HTML în documentul în care se va poziționa browser-ul.

13.4 Traducerea

Cu câțiva pași puteți configura mediul pentru localizarea plugin-ului, astfel încât, în funcție de setările regionale ale computerului, plugin-ul va fi încărcat în diferite limbi.

13.4.1 Cerințe software

The easiest way to create and manage all the translation files is to install Qt Linguist. In a Debian-based GNU/Linux environment you can install it typing:

```
sudo apt-get install qt4-dev-tools
```

13.4.2 Fișierele și directorul

Atunci când creați plugin-ul, veți găsi folderul `i18n` în directorul principal al plugin-urilor.

Toate fișierele de traducere trebuie să se afle în acest director.

fişier .pro

First you should create a `.pro` file, that is a *project* file that can be managed by **Qt Linguist**.

In this `.pro` file you have to specify all the files and forms you want to translate. This file is used to set up the localization files and variables. A possible project file, matching the structure of our *example plugin*:

```
FORMS = ../form.ui
SOURCES = ../your_plugin.py
TRANSLATIONS = your_plugin_it.ts
```

Your plugin might follow a more complex structure, and it might be distributed across several files. If this is the case, keep in mind that `pylupdate4`, the program we use to read the `.pro` file and update the translatable string, does not expand wild card characters, so you need to place every file explicitly in the `.pro` file. Your project file might then look like something like this:

```
FORMS = ../ui/about.ui ../ui/feedback.ui \
        ../ui/main_dialog.ui
SOURCES = ../your_plugin.py ../computation.py \
          ../utils.py
```

Mai mult, fișierul dvs. `your_plugin.  fişier py` este cel care *apelează* toate meniurile și sub-meniurile plugin-ului dvs. din bara de instrumente QGIS, iar dumneavoastră doriți să le traduceți pe toate.

În cele din urmă, cu ajutorul variabilei *TRANSLATIONS* puteți specifica limbile de traducere pe care le doriți.

> **Warning:** Asigurați-vă că denumirea fișierului `ts` este ceva de genul `pluginul_dvs` + `limba` + `.ts`, în caz contrar încărcarea limbii va eșua! Utilizați o prescurtare pe 2 litere a limbii (**it** pentru italiană, **de** pentru germană etc...)

fişier .ts

O dată ce ați creat un `.  pro` sunteți gata să generați fișier(ele) `.ts`, ale limb(ilor) plugin-ului dvs.

Deschideți o fereastră terminal, mergeți în directorul `your_plugin/i18n` și introduceți:

```
pylupdate4 your_plugin.pro
```

ar trebui să vedeți fișier(ele) `your_plugin_language.ts`.

Deschideți cu **Qt Linguist** fișierul `.ts`, apoi începeți să-l traduceți,

fişier .qm

Când ați terminat de tradus plugin-ul (în cazul în care unele șiruri de caractere nu sunt completate, pentru acestea va fi utilizată limba sursă) trebuie să creați un fișier `.qm` (fișierul `.ts`, compilat, care va fi utilizat de către QGIS).

Este suficient să deschideți o fereastră terminal în directorul `your_plugin/i18n`, apoi tastați:

```
lrelease your_plugin.ts
```

acum, in directorul `i18n` veți vedea fișier(ele) `pluginului_dvs.qm`.

13.4.3 Translate using Makefile

Alternatively you can use the makefile to extract messages from python code and Qt dialogs, if you created your plugin with Plugin Builder. At the beginning of the Makefile there is a LOCALES variable:

```
LOCALES = en
```

Add the abbreviation of the language to this variable, for example for Hungarian language:

```
LOCALES = en hu
```

Now you can generate or update the `hu.ts` file (and the `en.ts` too) from the sources by:

```
make transup
```

After this, you have updated `.ts` file for all languages set in the LOCALES variable. Use **Qt4 Linguist** to translate the program messages. Finishing the translation the `.qm` files can be created by the transcompile:

```
make transcompile
```

You have to distribute `.ts` files with your plugin.

13.4.4 Încărcarea plugin-ului

Pentru a putea vedea traducerea plugin-ului dvs., este suficient să deschideți QGIS, schimbați limba (*Setări* → *Opțiuni* → *Limba*) și restartați QGIS.

Ar trebui să vedeți plugin-ul în limba corectă.

> **Warning:** Dacă schimbați ceva în plugin (noi interfețe, un meniu nou etc.), va trebui să **generați din nou** versiunea de actualizare a fișierelor `.ts` și `.qm`, de aceea, executați iarăși comanda de mai sus.

Infrastructura de autentificare

- Introducere
- Glosar
- QgsAuthManager the entry point
  - Init the manager and set the master password
  - Populate authdb with a new Authentication Configuration entry
    - Available Authentication methods
    - Populate Authorities
    - Manage PKI bundles with QgsPkiBundle
  - Remove entry from authdb
  - Leave authcfg expansion to QgsAuthManager
    - PKI examples with other data providers
- Adapt plugins to use Authentication infrastructure
- Authentication GUIs
  - GUI to select credentials
  - Authentication Editor GUI
  - Authorities Editor GUI

14.1 Introducere

Referința utilizatorului pentru infrastructura de autentificare se găsește în Manualul Utilizatorului, în paragraful *authentication_overview*.

Acest capitol descrie cele mai bune practici de utilizare, din perspectiva dezvoltatorului, a Sistemului de Autentificare.

Warning: Authentication system API is more than the classes and methods exposed here, but it's strongly suggested to use the ones described here and exposed in the following snippets for two main reasons

1. Authentication API will change during the move to QGIS3
2. Python bindings will be restricted to the `QgsAuthManager` class use.

Most of the following snippets are derived from the code of Geoserver Explorer plugin and its tests. This is the first plugin that used Authentication infrastructure. The plugin code and its tests can be found at this link. Other good code reference can be read from the authentication infrastructure tests code

14.2 Glosar

Here are some definition of the most common objects treated in this chapter.

Parola Master Password to allow access and decrypt credential stored in the QGIS Authentication DB

Baza de Date de Autentificare A *Master Password* crypted sqlite db `<user home>/.qgis2/qgis-auth.db` where *Authentication Configuration* are stored. e.g user/password, personal certificates and keys, Certificate Authorities

Authentication DB *Authentication Database*

Authentication Configuration A set of authentication data depending on *Authentication Method*. e.g Basic authentication method stores the couple of user/password.

Authentication config *Authentication Configuration*

Authentication Method A specific method used to get authenticated. Each method has its own protocol used to gain the authenticated level. Each method is implemented as shared library loaded dynamically during QGIS authentication infrastructure init.

14.3 QgsAuthManager the entry point

The QgsAuthManager singleton is the entry point to use the credentials stored in the QGIS encrypted *Authentication DB*:

`<user home>/.qgis2/qgis-auth.db`

This class takes care of the user interaction: by asking to set master password or by transparently using it to access crypted stored info.

14.3.1 Init the manager and set the master password

The following snippet gives an example to set master password to open the access to the authentication settings. Code comments are important to understand the snippet.

```
authMgr = QgsAuthManager.instance()
# check if QgsAuthManager has been already initialized... a side effect
# of the QgsAuthManager.init() is that AuthDbPath is set.
# QgsAuthManager.init() is executed during QGis application init and hence
# you do not normally need to call it directly.
if authMgr.authenticationDbPath():
    # already initilised => we are inside a QGIS app.
    if authMgr.masterPasswordIsSet():
        msg = 'Authentication master password not recognized'
        assert authMgr.masterPasswordSame( "your master password" ), msg
    else:
        msg = 'Master password could not be set'
        # The verify parameter check if the hash of the password was
        # already saved in the authentication db
        assert authMgr.setMasterPassword( "your master password",
                                          verify=True), msg
else:
    # outside qgis, e.g. in a testing environment => setup env var before
    # db init
    os.environ['QGIS_AUTH_DB_DIR_PATH'] = "/path/where/located/qgis-auth.db"
    msg = 'Master password could not be set'
    assert authMgr.setMasterPassword("your master password", True), msg
    authMgr.init( "/path/where/located/qgis-auth.db" )
```

14.3.2 Populate authdb with a new Authentication Configuration entry

Any stored credential is a *Authentication Configuration* instance of the QgsAuthMethodConfig class accessed using a unique string like the following one:

```
authcfg = 'fm1s770'
```

that string is generated automatically when creating an entry using QGIS API or GUI.

QgsAuthMethodConfig is the base class for any *Authentication Method*. Any Authentication Method sets a configuration hash map where authentication informations will be stored. Hereafter an useful snippet to store PKI-path credentials for an hypothetic alice user:

```
authMgr = QgsAuthManager.instance()
# set alice PKI data
p_config = QgsAuthMethodConfig()
p_config.setName("alice")
p_config.setMethod("PKI-Paths")
p_config.setUri("http://example.com")
p_config.setConfig("certpath", "path/to/alice-cert.pem" ))
p_config.setConfig("keypath", "path/to/alice-key.pem" ))
# check if method parameters are correctly set
assert p_config.isValid()

# register alice data in authdb returning the ``authcfg`` of the stored
# configuration
authMgr.storeAuthenticationConfig(p_config)
newAuthCfgId = p_config.id()
assert (newAuthCfgId)
```

Available Authentication methods

*Authentication Method*s are loaded dynamically during authentication manager init. The list of Authentication method can vary with QGIS evolution, but the original list of available methods is:

1. `Basic` User and password authentication
2. `Identity-Cert` Identity certificate authentication
3. `PKI-Paths` PKI paths authentication
4. Autentificare `PKI-PKCS#12` PKI PKCS#12

The above strings are that identify authentication methods in the QGIS authentication system. In Development section is described how to create a new c++ *Authentication Method*.

Populate Authorities

```
authMgr = QgsAuthManager.instance()
# add authorities
cacerts = QSslCertificate.fromPath( "/path/to/ca_chains.pem" )
assert cacerts is not None
# store CA
authMgr.storeCertAuthorities(cacerts)
# and rebuild CA caches
authMgr.rebuildCaCertsCache()
authMgr.rebuildTrustedCaCertsCache()
```

Warning: Due to QT4/OpenSSL interface limitation, updated cached CA are exposed to OpenSsl only almost a minute later. Hope this will be solved in QT5 authentication infrastructure.

Manage PKI bundles with QgsPkiBundle

A convenience class to pack PKI bundles composed on SslCert, SslKey and CA chain is the QgsPkiBundle class. Hereafter a snippet to get password protected:

```
# add alice cert in case of key with pwd
boundle = QgsPkiBundle.fromPemPaths( "/path/to/alice-cert.pem",
                                     "/path/to/alice-key_w-pass.pem",
                                     "unlock_pwd",
                                     "list_of_CAs_to_bundle" )
assert boundle is not None
assert boundle.isValid()
```

Refer to QgsPkiBundle class documentation to extract cert/key/CAs from the bundle.

14.3.3 Remove entry from authdb

We can remove an entry from *Authentication Database* using it's `authcfg` identifier with the following snippet:

```
authMgr = QgsAuthManager.instance()
authMgr.removeAuthenticationConfig( "authCfg_Id_to_remove" )
```

14.3.4 Leave authcfg expansion to QgsAuthManager

The best way to use an *Authentication Config* stored in the *Authentication DB* is referring it with the unique identifier `authcfg`. Expanding, means convert it from an identifier to a complete set of credentials. The best practice to use stored *Authentication Configs*, is to leave it managed automatically by the Authentication manager. The common use of a stored configuration is to connect to an authentication enabled service like a WMS or WFS or to a DB connection.

Note: Take into account that not all QGIS data providers are integrated with the Authentication infrastructure. Each authentication method, derived from the base class QgsAuthMethod and support a different set of Providers. For example `Identity-Cert` method supports the following list of providers:

```
In [19]: authM = QgsAuthManager.instance()
In [20]: authM.authMethod("Identity-Cert").supportedDataProviders()
Out[20]: [u'ows', u'wfs', u'wcs', u'wms', u'postgres']
```

For example, to access a WMS service using stored credentials identified with `authcfg = 'fm1s770'`, we just have to use the `authcfg` in the data source URL like in the following snippet:

```
authCfg = 'fm1s770'
quri = QgsDataSourceURI()
quri.setParam("layers", 'usa:states')
quri.setParam("styles", '')
quri.setParam("format", 'image/png')
quri.setParam("crs", 'EPSG:4326')
quri.setParam("dpiMode", '7')
quri.setParam("featureCount", '10')
quri.setParam("authcfg", authCfg)   # <---- here my authCfg url parameter
quri.setParam("contextualWMSLegend", '0')
quri.setParam("url", 'https://my_auth_enabled_server_ip/wms')
rlayer = QgsRasterLayer(quri.encodedUri(), 'states', 'wms')
```

In the upper case, the `wms` provider will take care to expand `authcfg` URI parameter with credential just before setting the HTTP connection.

Warning: Developer would have to leave `authcfg` expansion to the QgsAuthManager, in this way he will be sure that expansion is not done too early.

Usually an URI string, build using `QgsDataSourceURI` class, is used to set QGIS data source in the following way:

```
rlayer = QgsRasterLayer( quri.uri(False), 'states', 'wms')
```

Note: The `False` parameter is important to avoid URI complete expansion of the `authcfg` id present in the URI.

PKI examples with other data providers

Other example can be read directly in the QGIS tests upstream as in test_authmanager_pki_ows or test_authmanager_pki_postgres.

14.4 Adapt plugins to use Authentication infrastructure

Many third party plugins are using httplib2 to create HTTP connections instead of integrating with `QgsNetworkAccessManager` and its related Authentication Infrastructure integration. To facilitate this integration an helper python function has been created called `NetworkAccessManager`. Its code can be found here.

This helper class can be used as in the following snippet:

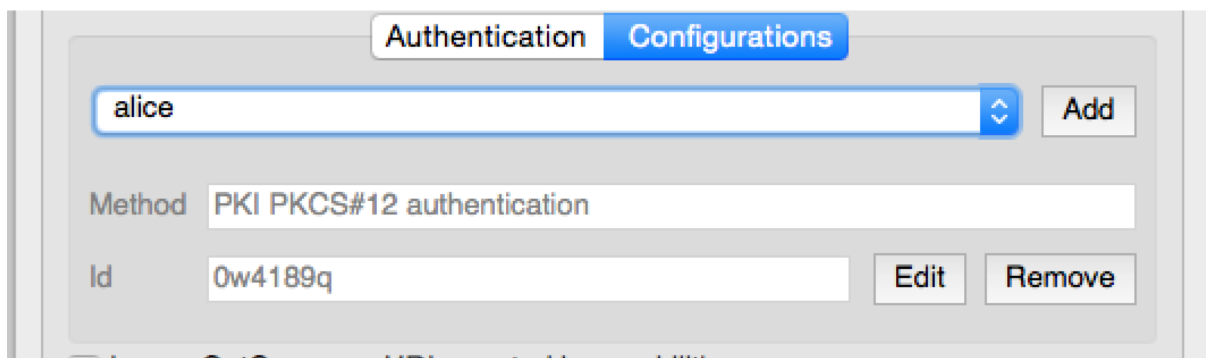
```
http = NetworkAccessManager(authid="my_authCfg", exception_class=My_FailedRequestError)
try:
  response, content = http.request( "my_rest_url" )
except My_FailedRequestError, e:
  # Handle exception
  pass
```

14.5 Authentication GUIs

In this paragraph are listed the available GUIs useful to integrate authentication infrastructure in custom interfaces.

14.5.1 GUI to select credentials

If it's necessary to select a *Authentication Configuration* from the set stored in the *Authentication DB* it is available in the GUI class QgsAuthConfigSelect



and can be used as in the following snippet:

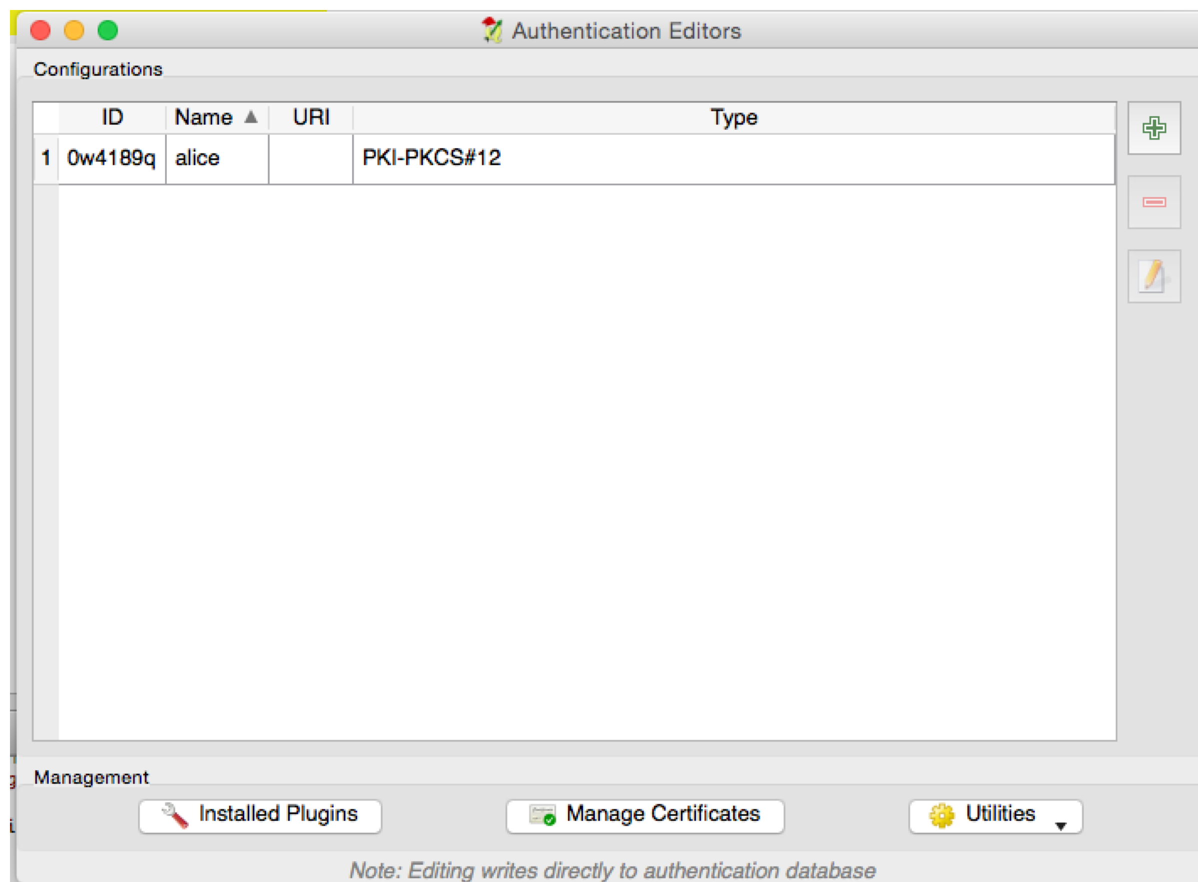
```
# create the instance of the QgsAuthConfigSelect GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent, "postgres" )
# add the above created gui in a new tab of the interface where the
```

```
# GUI has to be integrated
tabGui.insertTab( 1, gui, "Configurations" )
```

The above example is get from the QGIS source code The second parameter of the GUI constructor refers to data provider type. The parameter is used to restrict the compatible *Authentication Method*s with the specified provider.

14.5.2 Authentication Editor GUI

The complete GUI used to manage credentials, authorities and to access to Authentication utilities is managed by the class QgsAuthEditorWidgets



and can be used as in the following snippet:

```
# create the instance of the QgsAuthEditorWidgets GUI hierarchically linked to
# the widget referred with 'parent'
gui = QgsAuthConfigSelect( parent )
gui.show()
```

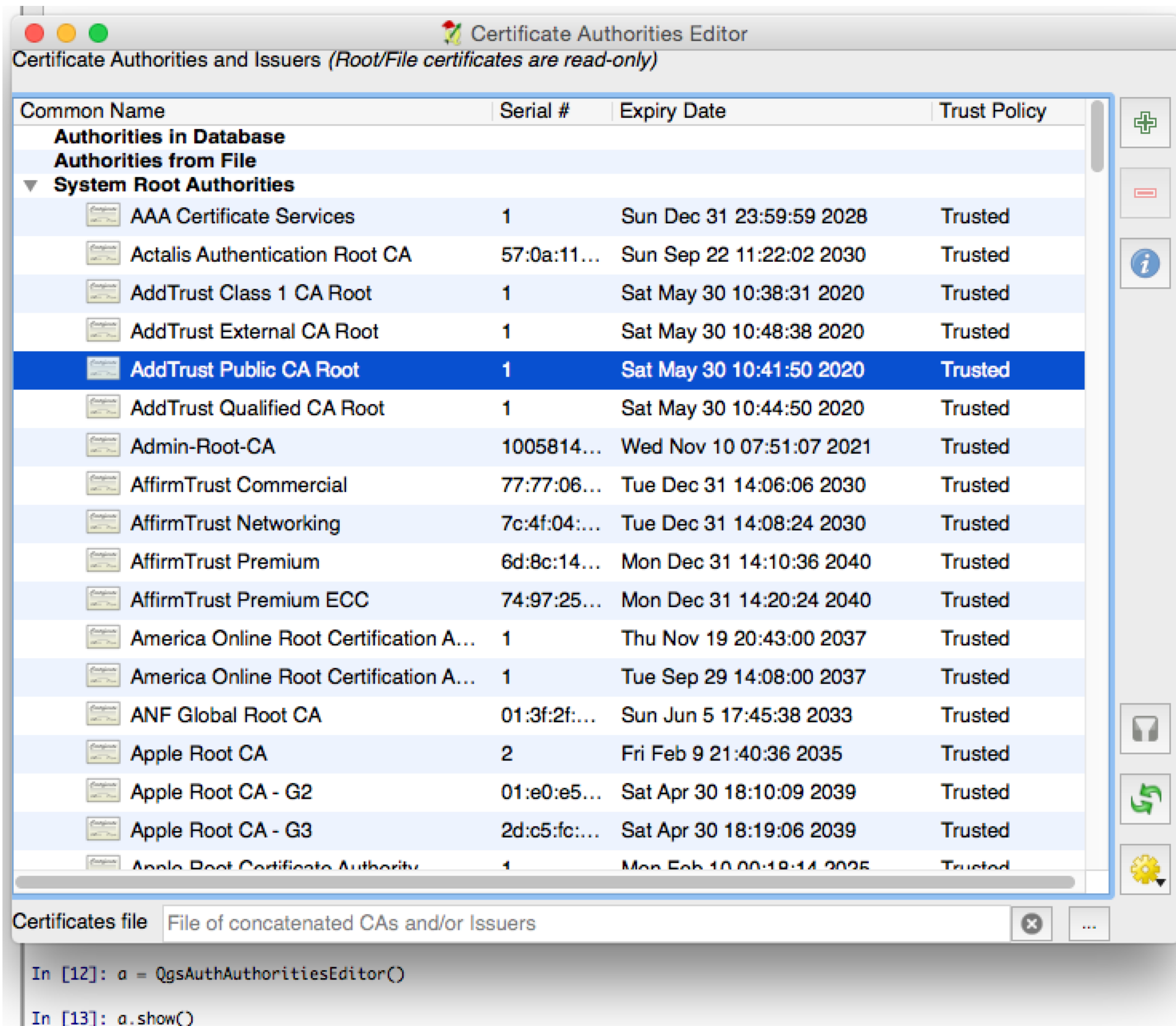
an integrated example can be found in the related test

14.5.3 Authorities Editor GUI

A GUI used to manage only authorities is managed by the class QgsAuthAuthoritiesEditor

and can be used as in the following snippet:

```
# create the instance of the QgsAuthAuthoritiesEditor GUI hierarchically
#  linked to the widget referred with 'parent'
```


Certificate Authorities Editor
Certificate Authorities and Issuers (Root/File certificates are read-only)
Common Name
Serial #
Expiry Date
Trust Policy
Authorities in Database
Authorities from File
System Root Authorities
AAA Certificate Services | 1 | Sun Dec 31 23:59:59 2028 | Trusted
Actalis Authentication Root CA | 57:0a:11… | Sun Sep 22 11:22:02 2030 | Trusted
AddTrust Class 1 CA Root | 1 | Sat May 30 10:38:31 2020 | Trusted
AddTrust External CA Root | 1 | Sat May 30 10:48:38 2020 | Trusted
AddTrust Public CA Root | 1 | Sat May 30 10:41:50 2020 | Trusted
AddTrust Qualified CA Root | 1 | Sat May 30 10:44:50 2020 | Trusted
Admin-Root-CA | 1005814… | Wed Nov 10 07:51:07 2021 | Trusted
AffirmTrust Commercial | 77:77:06… | Tue Dec 31 14:06:06 2030 | Trusted
AffirmTrust Networking | 7c:4f:04:… | Tue Dec 31 14:08:24 2030 | Trusted
AffirmTrust Premium | 6d:8c:14… | Mon Dec 31 14:10:36 2040 | Trusted
AffirmTrust Premium ECC | 74:97:25… | Mon Dec 31 14:20:24 2040 | Trusted
America Online Root Certification A… | 1 | Thu Nov 19 20:43:00 2037 | Trusted
America Online Root Certification A… | 1 | Tue Sep 29 14:08:00 2037 | Trusted
ANF Global Root CA | 01:3f:2f:… | Sun Jun 5 17:45:38 2033 | Trusted
Apple Root CA | 2 | Fri Feb 9 21:40:36 2035 | Trusted
Apple Root CA - G2 | 01:e0:e5… | Sat Apr 30 18:10:09 2039 | Trusted
Apple Root CA - G3 | 2d:c5:fc:… | Sat Apr 30 18:19:06 2039 | Trusted
Certificates file
File of concatenated CAs and/or Issuers
In [12]: a = QgsAuthAuthoritiesEditor()
In [13]: a.show()

```
gui = QgsAuthAuthoritiesEditor( parent )
gui.show()
```

CHAPTER 15

Setările IDE pentru scrierea și depanarea de plugin-uri

- O notă privind configurarea IDE-ului în Windows
- Depanare cu ajutorul Eclipse și PyDev
  - Instalare
  - Pregătirea QGIS
  - Configurarea Eclipse
  - Configurarea depanatorului
  - Configurați Eclipse pentru a înțelege API-ul
- Depanarea cu ajutorul PDB

Deși fiecare programator preferă un anumit editor IDE/Text, iată câteva recomandări de setare a unor IDE-uri populare, pentru scrierea și depanarea plugin-urilor Python specifice QGIS.

15.1 O notă privind configurarea IDE-ului în Windows

În Linux nu este necesară nici o configurare suplimentară pentru dezvoltarea plugin-urilor. În Windows însă, trebuie să vă asigurați că aveți aceleași setări de mediu și folosiți aceleași bibliotecile și interpretor ca și QGIS. Cel mai rapid mod de a face acest lucru, este de a modifica fișierul batch de pornire a QGIS.

Dacă ați folosit programul de instalare OSGeo4W, îl puteți găsi în folderul `bin` al propriei instalări OSGeo4W. Căutați ceva de genul `C:\OSGeo4W\bin\qgis-unstable.bat`.

Pentru utilizarea IDE-ului Pyscripter, iată ce aveți de făcut:

- Faceți o copie a fișierului `qgis-unstable.bat` și redenumiți-o `pyscripter.bat`.
- Deschideți-o într-un editor. Apoi eliminați ultima linie, cea care startează QGIS.
- Adăugați o linie care să indice calea către executabilul Pyscripter, apoi adăugați argumentul care stabilește versiunea de Python ce urmează a fi utilizată (2.7 în cazul QGIS >= 2.0)
- De asemenea, adăugați și un argument care să indice folderul unde poate găsi Pyscripter dll-ul Python folosit de către QGIS, acesta aflându-se în folderul bin al instalării OSGeoW

```
@echo off
SET OSGEO4W_ROOT=C:\OSGeo4W
call "%OSGEO4W_ROOT%"\bin\o4w_env.bat
call "%OSGEO4W_ROOT%"\bin\gdal16.bat
@echo off
path %PATH%;%GISBASE%\bin
Start C:\pyscripter\pyscripter.exe --python25 --pythondllpath=C:\OSGeo4W\bin
```

Acum, când veți efectua dublu clic pe acest fișier batch, el va starta Pyscripter, având calea corectă.

Mult mai popular decât Pyscripter, Eclipse este o alegere comună în rândul dezvoltatorilor. În următoarele secțiuni, vă vom explica cum să-l configurați pentru dezvoltarea și testarea plugin-urilor. Pentru utilizarea în Windows, ar trebui să creați, de asemenea, un fișier batch pe care să-l utilizați la pornirea Eclipse.

Pentru a crea fișierul batch, urmați acești pași:

- Localizați folderul în care rezidă `qgis_core.dll`. În mod normal, el se găsește în `C:OSGeo4Wappsqgisbin`, dar dacă ați compilat propria aplicație QGIS, atunci el va fi în folderul `output/bin/RelWithDebInfo`
- Localizați executabilul `eclipse.exe`.
- Creați următorul script și folosiți-l pentru a starta Eclipse, atunci când dezvoltați plugin-uri QGIS.

```
call "C:\OSGeo4W\bin\o4w_env.bat"
set PATH=%PATH%;C:\path\to\your\qgis_core.dll\parent\folder
C:\path\to\your\eclipse.exe
```

15.2 Depanare cu ajutorul Eclipse și PyDev

15.2.1 Instalare

Pentru a utiliza Eclipse, asigurați-vă că ați instalat următoarele

- Eclipse
- Plugin-ul Eclipse Aptana sau PyDev
- QGIS 2.x

15.2.2 Pregătirea QGIS

Chiar și în QGIS, este necesară efectuarea anumitor acțiuni pregătitoare. Două plugin-uri sunt de interes: *Remote Debug* și *Plugin Reloader*.

- Mergeți la *Plugin-uri* → *Gestiune și Instalare plugin-uri...*
- Căutați *Remote Debug* (la această dată este încă experimental, deci, în cazul în care nu-l observați, va trebui să activați plugin-urile experimentale, din fila *Opțiunilor*). Instalați-l.
- De asemenea, căutați *Plugin Reloader* și instalați-l. Acest lucru vă va permite să reîncărcați un plug-in, fără a fi necesare închiderea și repornirea QGIS.

15.2.3 Configurarea Eclipse

În Eclipse, creați un nou proiect. Puteți să selectați *General Project* și să legați ulterior sursele dvs. reale, așa că nu prea contează unde plasați acest proiect.

Acum, faceți clic dreapta pe noul proiect și alegeți *New* → *Folder*.

Faceți clic pe **[Avansat]** și alegeți *Legătură către o locație alternativă (Folder Legat)*. În cazul în care deja aveți sursele pe care doriți să le depanați, le puteți alege pe acestea. În caz contrar, creați un folder, așa cum s-a explicat anterior.

Acum, în fereastra *Project Explorer*, va apărea arborele sursă și veți putea începe să lucrați la cod. Aveți disponibile deja evidențierea sintaxei și toate celelalte instrumente puternice din IDE.

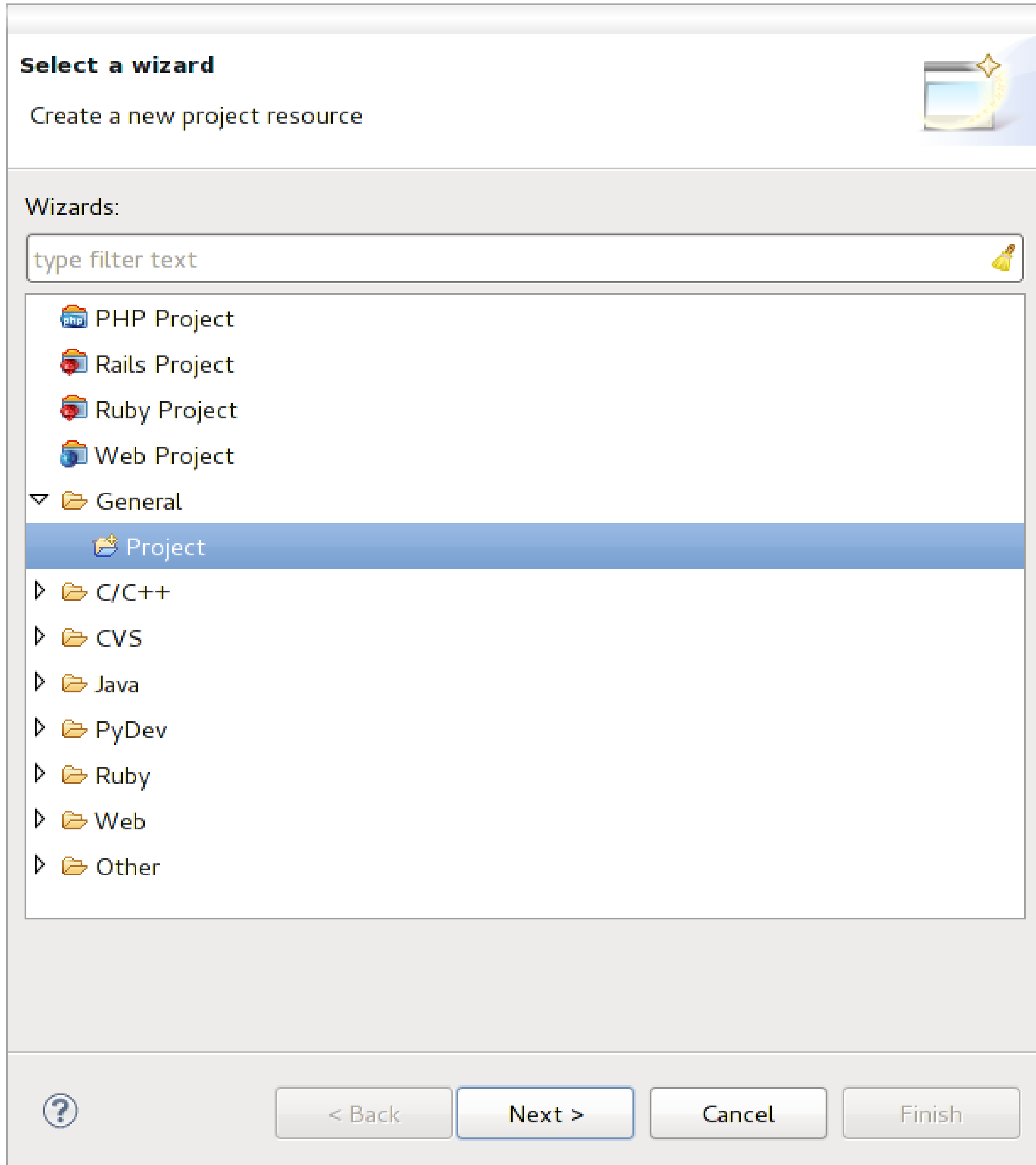


Figure 15.1: Proiectul Eclipse

15.2.4 Configurarea depanatorului

Pentru a vedea depanatorul la lucru, comutați în perspectiva Depanare din Eclipse (*Window* → *Open Perspective* → *Other* → *Debug*).

Acum, porniți serverul de depanare PyDev, alegând *PyDev* → *Start Debug Server*.

În acest moment, Eclipse așteaptă o conexiune de la QGIS către serverul de depanare, iar când QGIS se va conecta la serverul de depanare va fi permis controlul scripturilor Python. Exact pentru acest lucru am instalat plugin-ul *Remote Debug*. Deci, startați QGIS, în cazul în care nu ați făcut-o deja și efectuați clic pe simbolul insectei.

Acum puteți seta un punct de întrerupere, și de îndată ce codul îl va atinge, execuția se va opri, după care veți putea inspecta starea actuală a plugin-ului. (Punctul de întrerupere este reprezentat de punctul verde din imaginea de mai jos, și se poate introduce printr-un dublu clic în spațiul alb din stânga liniei în care doriți un punct de întrerupere).

```
 88
 89    def printProfile(self):
 90        printer = QPrinter( QPrinter.HighResolution )
 91        printer.setOutputFormat( QPrinter.PdfFormat )
 92        printer.setPaperSize( QPrinter.A4 )
 93        printer.setOrientation( QPrinter.Landscape )
 94
 95        printPreviewDlg = QPrintPreviewDialog( )
 96        printPreviewDlg.paintRequested.connect( self.printRequested )
 97
 98        printPreviewDlg.exec_()
 99
100    @pyqtSlot( QPrinter )
101    def printRequested( self, printer ):
102        self.webView.print_( printer )
```

Figure 15.2: Punct de întrerupere

Un aspect foarte interesant este faptul că puteți utiliza consola de depanare. Asigurați-vă că execuția este, în mod curent, stopată, înainte de a continua.

Deschideți fereastra consolei (*Window* → *Show view*). Se va afișa consola *Debug Server*, ceea ce nu este prea interesant. În schimb, butonul **[Open Console]** permite trecerea la mult mai interesanta consolă de depanare PyDev. Faceți clic pe săgeata de lângă butonul **[Open Console]** și alegeți *PyDev Console*. Se deschide o fereastră care vă va întreba ce consolă doriți să deschideți. Alegeți *PyDev Debug Console*. În cazul când aceasta este gri, vă cere să porniți depanatorul și să selectați un cadru valid, asigurați-vă că ați atașat depanatorul la distanță, iar în prezent sunteți pe un punct de întrerupere.

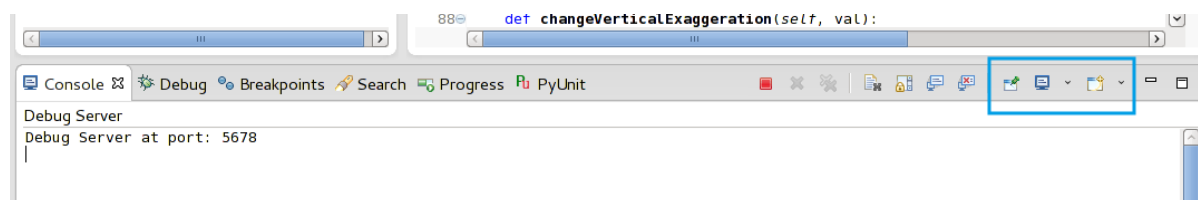


Figure 15.3: Consola de depanare PyDev

Acum aveți o consolă interactivă care vă permite să testați orice comenzi din interior, în contextul actual. Puteți manipula variabile, să efectuați apeluri API sau orice altceva.

Un pic enervant este faptul că, de fiecare dată când introduceți o comandă, consola comută înapoi la serverul de depanare. Pentru a opri acest comportament, aveți posibilitatea să faceți clic pe butonul *Pin Console* din pagina serverului de depanare, pentru reținerea acestei decizii, cel puțin pentru sesiunea de depanare curentă.

15.2.5 Configurați Eclipse pentru a înțelege API-ul

O caracteristică facilă este de a pregăti Eclipse pentru API-ul QGIS. Aceasta va permite verificarea eventualelor greșeli de ortografie din cadrul codului. Dar nu doar atât, va permite ca Eclipse să autocompleteze din importurile către apelurile API.

Pentru a face acest lucru, Eclipse analizează fișierele bibliotecii QGIS și primește toate informațiile de acolo. Singurul lucru pe care trebuie să-l faceți este de a-i indica lui Eclipse unde să găsească bibliotecile.

Faceți clic pe *Window* → *Preferences* → *PyDev* → *Interpreter* → *Python*.

Veți vedea interpretorul de python (pe moment versiunea 2.7) configurat, în partea de sus a ferestrei și unele file în partea de jos. Filele interesante pentru noi sunt *Libraries* și *Forced Builtins*.

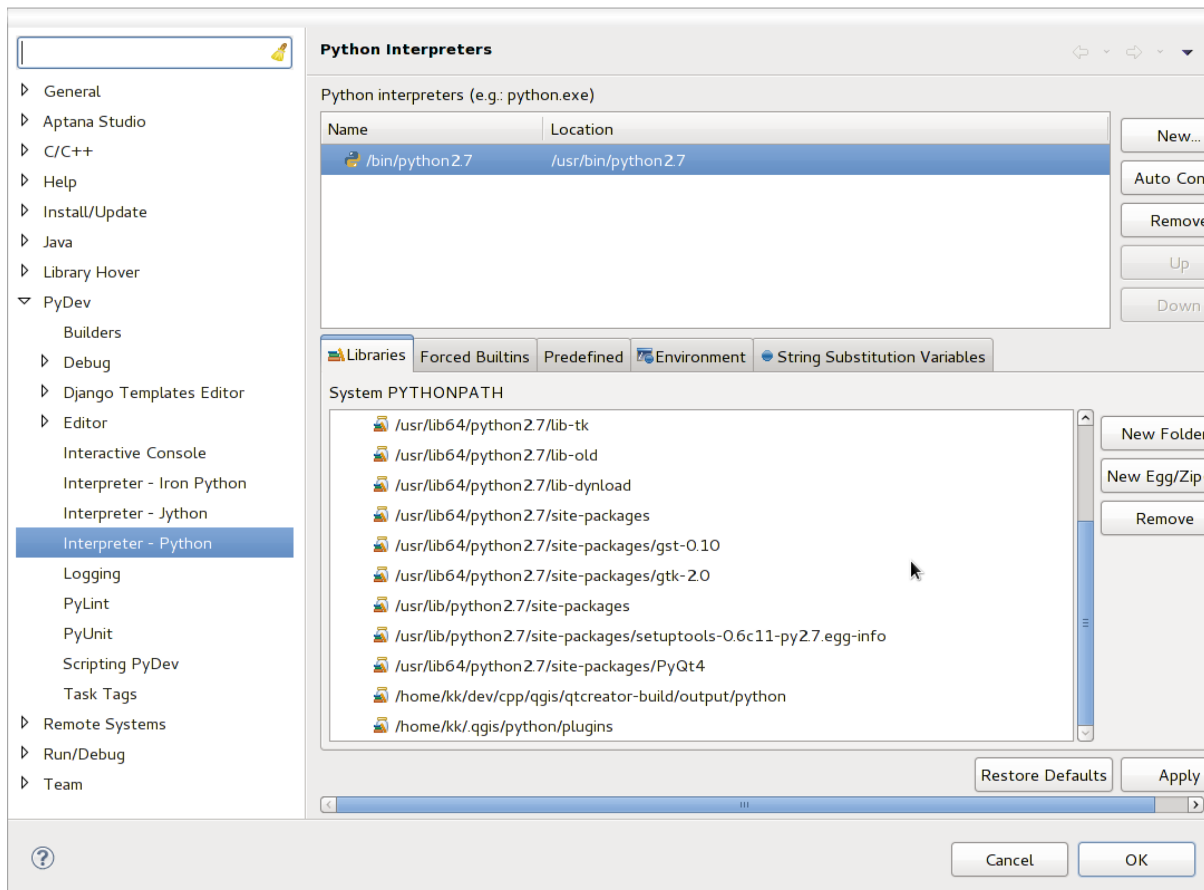


Figure 15.4: Consola de depanare PyDev

În primul rând deschideți fila Libraries. Adăugați un folder nou și selectați folderul python al aplicației QGIS instalate. Dacă nu știți unde se află acest director (acesta nu este folderul plugin-urilor) deschideți QGIS, startați o consolă python și pur și simplu introduceți `qgis`, apoi apăsați Enter. Acest lucru vă va arăta care modul QGIS este folosit, precum și calea sa. Ștergeți `/qgis/__init__.pyc` și veți obține calea pe care o căutați.

Ar trebui să adăugați, de asemenea, propriul director de plugin-uri aici (în Linux este `~/.qgis2/python/plugins`).

Apoi deschideți tab-ul *Includeri Forțate*, faceți clic pe *Nou...* și introduceți `qgis`. Acest lucru va face ca Eclipse să analizeze API-ul QGIS. Probabil doriți, de asemenea, ca Eclipse să știe și despre API-ul PyQt4. Prin urmare, adăugați și PyQt4 ca includere forțată. Probabil că ar trebui să se afle deja în fila bibliotecilor.

Faceți clic pe *OK* și ați terminat.

Note: Notă: la orice modificare a API-ului QGIS (de exemplu, în urma compilării QGIS master și a modificării

fișierului SIP), ar trebui să mergeți înapoi la această pagină și pur și simplu să faceți clic pe *Aplicare*. Acest lucru va face ca Eclipse să analizeze toate bibliotecile din nou.

15.3 Depanarea cu ajutorul PDB

Dacă nu folosiți un IDE, cum ar fi Eclipse, puteți depana folosind PDB, urmând acești pași.

Mai întâi, adăugați acest cod în locul în care doriți depanarea

```
# Use pdb for debugging
import pdb
# These lines allow you to set a breakpoint in the app
pyqtRemoveInputHook()
pdb.set_trace()
```

Apoi executați QGIS din linia de comandă.

On Linux do:

```
$ ./Qgis
```

On macOS do:

```
$ /Applications/Qgis.app/Contents/MacOS/Qgis
```

Iar când aplicația atinge punctul de întrerupere aveți posibilitatea de a tasta în consolă!

DE EFECTUAT: Adăugați informații pentru testare

CHAPTER 16

Utilizarea straturilor plugin-ului

Dacă plugin-ul dvs. folosește propriile metode de a randa un strat de hartă, scrierea propriului tip de strat, bazat pe QgsPluginLayer, ar putea fi cea mai indicată.

DE EFECTUAT: Verificați corectitudinea și elaborați cazuri de utilizare corectă pentru QgsPluginLayer, ...

16.1 Subclasarea QgsPluginLayer

Mai jos este un exemplu minimal de implementare pentru QgsPluginLayer. Acesta este un extras din Exemplu de plugin filigran

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

  def __init__(self):
    QgsPluginLayer.__init__(self, WatermarkPluginLayer.LAYER_TYPE, "Watermark plugin layer")
    self.setValid(True)

  def draw(self, rendererContext):
    image = QImage("myimage.png")
    painter = rendererContext.painter()
    painter.save()
    painter.drawImage(10, 10, image)
    painter.restore()
    return True
```

De asemenea, pot fi adăugate metode pentru citirea și scrierea de informații specifice, în fișierul proiectului

```
def readXml(self, node):
  pass

def writeXml(self, node, doc):
  pass
```

Când încărcați un proiect care conține un astfel de strat, este nevoie de o fabrică de clase

```
class WatermarkPluginLayerType(QgsPluginLayerType):

  def __init__(self):
    QgsPluginLayerType.__init__(self, WatermarkPluginLayer.LAYER_TYPE)

  def createLayer(self):
    return WatermarkPluginLayer()
```

Puteți adăuga, de asemenea, codul pentru afișarea de informații personalizate în proprietățile stratului

```
def showLayerProperties(self, layer):
  pass
```

CHAPTER 17

Compatibilitatea cu versiunile QGIS anterioare

17.1 Meniul plugin-ului

Dacă plasați intrările de meniu ale plugin-ului într-unul dintre noile meniuri (*Raster*, *Vector*, *Database* sau *Web*), va trebui să modificați codul funcțiilor `initGui()` și `unload()`. Din moment ce aceste noi meniuri sunt disponibile începând de la QGIS 2.0, primul pas este de a verifica faptul că versiunea de QGIS are toate funcțiunile necesare. Dacă noile meniuri sunt disponibile, vom plasa pluginul nostru în cadrul acestui meniu, altfel vom folosi vechiul meniu *Plugins*. Iată un exemplu pentru meniul *Raster*

```
def initGui(self):
  # create action that will start plugin configuration
  self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWindo
  self.action.setWhatsThis("Configuration for test plugin")
  self.action.setStatusTip("This is status tip")
  QObject.connect(self.action, SIGNAL("triggered()"), self.run)

  # check if Raster menu available
  if hasattr(self.iface, "addPluginToRasterMenu"):
    # Raster menu and toolbar available
    self.iface.addRasterToolBarIcon(self.action)
    self.iface.addPluginToRasterMenu("&Test plugins", self.action)
  else:
    # there is no Raster menu, place plugin under Plugins menu as usual
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

  # connect to signal renderComplete which is emitted when canvas rendering is done
  QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

def unload(self):
  # check if Raster menu available and remove our buttons from appropriate
  # menu and toolbar
  if hasattr(self.iface, "addPluginToRasterMenu"):
    self.iface.removePluginRasterMenu("&Test plugins", self.action)
    self.iface.removeRasterToolBarIcon(self.action)
  else:
    self.iface.removePluginMenu("&Test plugins", self.action)
    self.iface.removeToolBarIcon(self.action)

  # disconnect from signal of the canvas
  QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest
```

CHAPTER 18

Lansarea plugin-ului dvs.

- Metadate și nume
- Codul și ajutorul
- Depozitul oficial al plugin-urilor python
  - Permisiuni
  - Managementul încrederii
  - Validare
  - Structura plugin-ului

O dată ce plugin-ul este gata și credeți că el ar putea fi de ajutor pentru unii utilizatori, nu ezitați să-l încărcați la *Depozitul oficial al plugin-urilor python*. Pe acea pagină puteți găsi instrucțiuni de împachetare și de pregătire a plugin-ului, pentru a lucra bine cu programul de instalare. Sau, în cazul în care ați dori să înființați un depozit propriu pentru plugin-uri, creați un simplu fișier XML, care va lista plugin-urile și metadatele lor; de exemplu, conssultați depozitele pentru plugin-uri.

Vă rugăm să acordați o grijă deosebită următoarelor recomandări:

18.1 Metadate și nume

- evitați folosirea unui nume prea asemănător cu cel al plugin-urilor existente
- dacă plugin-ul are o funcționalitate similară cu cea a unui plugin existent, vă rugăm să explicați diferențele în câmpul Despre, astfel încât utilizatorul va ști pe care să-l folosească, fără a fi nevoie de instalare și testare
- evitați repetarea cuvântului "plugin", în denumirea unui plugin
- utilizați câmpul descriere din metadate pentru o descriere de 1 linie, și câmpul Despre pentru instrucțiuni mai detaliate
- includeți un depozit de cod, un monitor de erori, și o pagină de start; astfel, va spori considerabil posibilitatea de colaborare, aceasta făcându-se foarte ușor cu ajutorul infrastructurilor web disponibile (GitHub, GitLab, BitBucket, etc.)
- alegeți etichetele cu grijă: evitați-le pe cele neinformative (ex: vector), preferându-le pe cele deja folosite de către alții (a se vedea site-ul plugin-urilor)
- adăugați o pictogramă adecvată, și nu o lăsați pe cea implicită; vedeți interfața QGIS pentru o sugestie despe stilul de utilizat

18.2 Codul și ajutorul

- nu includeți fișierul generat (ui_*.py, resources_rc.py, fișiere de ajutor generate. . .) și chestii inutile (ex: .gitignore) în depozit

- adăugați pluginul în meniul corespunzător (Vector, Raster, Web, Bază de date)
- atunci când este cazul (plugin-uri efectuând analize), luați în considerare adăugarea plugin-ului ca subplugin al cadrului de Procesare: acest lucru va permite utilizatorilor să-l rulați în lot, să-l integrați în fluxurile de lucru mai complexe, eliberându-vă de povara proiectării unei interfețe
- includeți cel puțin documentația minimă și, dacă este util pentru testare și înțelegere, datele eșantion.

18.3 Depozitul oficial al plugin-urilor python

Puteți găsi depozitul *oficial* al plugin-urilor python la http://plugins.qgis.org/.

Pentru a folosi depozitul oficial, trebuie să obțineți un ID OSGEO din portalul web OSGEO.

O dată ce ați încărcat plugin-ul, acesta va fi aprobat de către un membru al personalului și veți primi o notificare.

DE EFECTUAT: Introduceți un link către documentul guvernanței

18.3.1 Permisiuni

Aceste reguli au fost implementate în depozitul oficial al plugin-urilor:

- fiecare utilizator inregistrat poate adăuga un nou plugin
- membrii *staff-ului* pot aproba sau dezaproba toate versiunile plugin-ului
- utilizatorii care au permisiunea specială *plugins.can_approve* au versiunile pe care le încarcă aprobate în mod automat
- utilizatorii care au permisiunea specială *plugins.can_approve* pot aproba versiunile încărcate de către alții, atât timp cât aceștia sunt prezenți în lista *proprietarilor* de plugin-uri
- un anumit plug-in pot fi șters și editat doar de utilizatorii *staff-ului* și de către *proprietarii* plugin-uri
- în cazul în care un utilizator fără permisiunea *plugins.can_approve* încarcă o nouă versiune, versiunea plug-inului nu va fi aprobată, din start.

18.3.2 Managementul încrederii

Membrii personalului pot acorda *încredere* creatorilor de plugin-uri, bifând permisiunea *plugins.can_approve* în cadrul front-end-ului.

Detaliile despre plugin oferă legături directe pentru a crește încrederea în creatorul sau *proprietarul*.plugin-ului.

18.3.3 Validare

Metadatele plugin-ului sunt importate automat din pachetul arhivat și sunt validate, la încărcarea plugin-ului.

Iată câteva reguli de validare pe care ar trebui să le cunoașteți atunci când doriți să încărcați un plugin în depozitul oficial:

1. numele folderului principal, care include plugin-ul, trebuie să conțină numai caracterele ASCII (A-Z și a-z), cifre, caractere de subliniere (_), minus (-) și, de asemenea, nu poate începe cu o cifră
2. `metadata.txt` este obligatoriu
3. toate metadatele necesare, menționate în *metadata table* trebuie să fie prezente
4. the *version* metadata field must be unique

18.3.4 Structura plugin-ului

Conform regulilor de validare, pachetul compresat (.zip) al plugin-ului trebuie să aibă o structură specifică, pentru a fi validat ca plugin funcțional. Deorece plugin-ul va fi dezarhivat în interiorul directorului de plugin-uri ale utilizatorului, el trebuie să aibă propriul director în interiorul fișierului zip, pentru a nu interfera cu alte plugin-uri. Fișierele obligatorii sunt: `metadata.txt` și `__init__.py`. Totuși, ar fi frumos să existe un `README` și, desigur, o pictogramă care să reprezinte pluginul (`resources.qrc`). Iată un exemplu despre modul în care ar trebui să arate un plugin.zip.

```
plugin.zip
  pluginfolder/
  |-- i18n
  |   |-- translation_file_de.ts
  |-- img
  |   |-- icon.png
  |   `-- iconsource.svg
  |-- __init__.py
  |-- Makefile
  |-- metadata.txt
  |-- more_code.py
  |-- main_code.py
  |-- README
  |-- resources.qrc
  |-- resources_rc.py
  `-- ui_Qt_user_interface_file.ui
```

CHAPTER 19

Fragmente de cod

- Cum să apelăm o metodă printr-o combinație rapidă de taste
- Inversarea Stării Straturilor
- Cum să accesați tabelul de atribute al entităților selectate

Această secțiune conține fragmente de cod, menite să faciliteze dezvoltarea plugin-urilor.

19.1 Cum să apelăm o metodă printr-o combinație rapidă de taste

În plug-in adăugați în `initGui()`

```
self.keyAction = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.keyAction, "F7") # action1 triggered by F7 key
self.iface.addPluginToMenu("&Test plugins", self.keyAction)
QObject.connect(self.keyAction, SIGNAL("triggered()"),self.keyActionF7)
```

În `unload()` add

```
self.iface.unregisterMainWindowAction(self.keyAction)
```

Metoda care este chemată atunci când se apasă tasta F7

```
def keyActionF7(self):
  QMessageBox.information(self.iface.mainWindow(),"Ok", "You pressed F7")
```

19.2 Inversarea Stării Straturilor

Începând de la QGIS 2.4, un nou API permite accesul direct la arborele straturilor din legendă. Exemplul următor prezintă modul în care se poate inversa vizibilitatea stratului activ

```
root = QgsProject.instance().layerTreeRoot()
node = root.findLayer(iface.activeLayer().id())
new_state = Qt.Checked if node.isVisible() == Qt.Unchecked else Qt.Unchecked
node.setVisible(new_state)
```

19.3 Cum să accesați tabelul de atribute al entităților selectate

```
def changeValue(self, value):
  layer = self.iface.activeLayer()
  if(layer):
    nF = layer.selectedFeatureCount()
    if (nF > 0):
      layer.startEditing()
      ob = layer.selectedFeaturesIds()
      b = QVariant(value)
      if (nF > 1):
        for i in ob:
        layer.changeAttributeValue(int(i), 1, b) # 1 being the second column
      else:
        layer.changeAttributeValue(int(ob[0]), 1, b) # 1 being the second column
      layer.commitChanges()
    else:
      QMessageBox.critical(self.iface.mainWindow(), "Error",
        "Please select at least one feature from current layer")
  else:
    QMessageBox.critical(self.iface.mainWindow(), "Error", "Please select a layer")
```

Metoda necesită un parametru (noua valoare pentru câmpul atribut al entităţi(lor) selectate) şi poate fi apelat de către

```
self.changeValue(50)
```

CHAPTER 20

Scrierea unui plugin Processing

- Creating a plugin that adds an algorithm provider
- Creating a plugin that contains a set of processing scripts

În funcție de tipul de plugin avut în vedere, adăugarea funcționalității acestuia ca algoritm (sau ca set) pentru Processing, ar putea fi o opțiune mai bună. Acest lucru ar consta într-o mai bună integrare în cadrul QGIS, o funcționalitate suplimentară (din moment ce poate fi rulat din cadrul componentelor Processing, cum ar fi modelatorul sau interfața de prelucrare în serie), precum și un timp de dezvoltare mai rapid (atât timp cât Processing vă va scuti de o mare parte din muncă).

This document describes how to create a new plugin that adds its functionality as Processing algorithms.

There are two main mechanisms for doing that:

- Creating a plugin that adds an algorithm provider: This options is more complex, but provides more flexibility
- Creating a plugin that contains a set of processing scripts: The simplest solution, you just need a set of Processing script files.

20.1 Creating a plugin that adds an algorithm provider

To create an algorithm provider, follow these steps:

- Instalarea Plugin Builder
- Creați un plugin nou, utilizând Plugin Builder. În cazul în care Plugin Builder vă cere șablonul de utilizat, selectați "Furnizor Processing".
- Plugin-ul creat conține un furnizor cu un singur algoritm. Atât fișierul furnizorului cât și cel al algoritmului sunt complet comentate și conțin informații cu privire la modul de modificare a furnizorului și de adăugare a algoritmilor suplimentari.

20.2 Creating a plugin that contains a set of processing scripts

To create a set of processing scripts, follow these steps:

- Create your scripts as described in the PyQGIS cookbook. All the scripts that you want to add, you should have them available in the Processing toolbox.
- In the *Scripts/Tools* group in the Processing toolbox, double-click on the *Create script collection plugin* item. You will see a window where you should select the scripts to add to the plugin (from the set of available ones in the toolbox), and some additional information needed for the plugin metadata.
- Click on OK and the plugin will be created.

- You can add additional scripts to the plugin by adding scripts python files to the *scripts* folder in the resulting plugin folder.

CHAPTER 21

Biblioteca de analiză a rețelelor

- Informații generale
- Construirea unui graf
- Analiza grafului
  - Găsirea celor mai scurte căi
  - Ariile de disponibilitate

Începând cu revizia ee19294562 (QGIS >= 1.8) noua bibliotecă de analiză de rețea a fost adaugată la biblioteca de analize de bază din QGIS. Biblioteca:

- creează graful matematic din datele geografice (straturi vectoriale de tip polilinie)
- implementează metode de bază din teoria grafurilor (în prezent, doar algoritmul lui Dijkstra)

Biblioteca analizelor de rețea a fost creată prin exportarea funcțiilor de bază ale plugin-ului RoadGraph, iar acum aveți posibilitatea să-i utilizați metodele în plugin-uri sau direct în consola Python.

21.1 Informații generale

Pe scurt, un caz tipic de utilizare poate fi descris astfel:

1. crearea grafului din geodate (de obicei un strat vectorial de tip polilinie)
2. rularea analizei grafului
3. folosirea rezultatelor analizei (de exemplu, vizualizarea lor)

21.2 Construirea unui graf

Primul lucru pe care trebuie să-l faceți — este de a pregăti datele de intrare, ceea ce înseamnă conversia stratului vectorial într-un graf. Toate acțiunile viitoare vor folosi acest graf, și nu stratul.

Ca și sursă putem folosi orice strat vectorial de tip polilinie. Nodurile poliliniilor devin noduri ale grafului, segmentele poliliniilor reprezentând marginile grafului. În cazul în care mai multe noduri au aceleași coordonate, atunci ele sunt în același nod al grafului. Astfel, două linii care au un nod comun devin conectate între ele.

În plus, în timpul creării grafului este posibilă "fixarea" ('legarea") de stratul vectorial de intrare a oricărui număr de puncte suplimentare. Pentru fiecare punct suplimentar va fi găsită o potrivire — cel mai apropiat nod sau cea mai apropiată muchie a grafului. În ultimul caz muchia va fi divizată iar noul nod va fi adăugat.

Atributele stratului vectorial și lungimea unei muchii pot fi folosite ca proprietăți ale marginii.

Converting from a vector layer to the graph is done using the Builder programming pattern. A graph is constructed using a so-called Director. There is only one Director for now: QgsLineVectorLayerDirector. The director sets the basic settings that will be used to construct a graph from a line vector layer, used by the builder to create the

graph. Currently, as in the case with the director, only one builder exists: QgsGraphBuilder, that creates QgsGraph objects. You may want to implement your own builders that will build a graphs compatible with such libraries as BGL or NetworkX.

To calculate edge properties the programming pattern strategy is used. For now only QgsDistanceArcProperter strategy is available, that takes into account the length of the route. You can implement your own strategy that will use all necessary parameters. For example, RoadGraph plugin uses a strategy that computes travel time using edge length and speed value from attributes.

Este timpul de a aprofunda acest proces.

Înainte de toate, pentru a utiliza această bibliotecă ar trebui să importăm modulul networkanalysis

```
from qgis.networkanalysis import *
```

Apoi, câteva exemple pentru crearea unui director

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsLineVectorLayerDirector(vLayer, -1, '', '', '', 3)

# use field with index 5 as source of information about road direction.
# one-way roads with direct direction have attribute value "yes",
# one-way roads with reverse direction have the value "1", and accordingly
# bidirectional roads have "no". By default roads are treated as two-way.
# This scheme can be used with OpenStreetMap data
director = QgsLineVectorLayerDirector(vLayer, 5, 'yes', '1', 'no', 3)
```

Pentru a construi un director ar trebui să transmitem stratul vectorial, care va fi folosit ca sursă pentru structura grafului și informațiile despre mișcările permise pe fiecare segment de drum (circulație unilaterală sau bilaterală, sens direct sau invers). Acest apel arată în felul următor

```
director = QgsLineVectorLayerDirector(vl, directionFieldId,
                                      directDirectionValue,
                                      reverseDirectionValue,
                                      bothDirectionValue,
                                      defaultDirection)
```

Iată lista completă a ceea ce înseamnă acești parametri:

- `vl` — stratul vectorial utilizat pentru a construi graf
- `directionFieldId` — indexul câmpului din tabelul de atribute, în care sunt stocate informații despre direcțiile drumurilor. Dacă este `-1`, atunci aceste informații nu se folosesc deloc. Număr întreg.
- `directDirectionValue` — valoarea câmpului pentru drumurile cu sens direct (trecere de la primul punct de linie la ultimul). Șir de caractere.
- `reverseDirectionValue` — valoarea câmpului pentru drumurile cu sens invers (în mișcare de la ultimul punct al liniei până la primul). Șir de caractere.
- `bothDirectionValue` — valoarea câmpului pentru drumurile bilaterale (pentru astfel de drumuri putem trece de la primul la ultimul punct și de la ultimul la primul). Șir de caractere.
- `defaultDirection` — direcția implicită a drumului. Această valoare va fi folosită pentru acele drumuri în care câmpul `directionFieldId` nu este setat sau are o valoare diferită de oricare din cele trei valori specificate mai sus. Număr întreg. `1` indică sensul direct, `2` indică sensul inversă, iar `3` indică ambele sensuri.

Este necesară, apoi, crearea unei strategii pentru calcularea proprietăților marginii

```
properter = QgsDistanceArcProperter()
```

Apoi spuneți directorului despre această strategie

```
director.addProperter(properter)
```

Acum putem crea constructorul, care va crea graful. Constructorul clasei QgsGraphBuilder ia mai multe argumente:

- crs — sistemul de coordonate de referință de utilizat. Argument obligatoriu.
- otfEnabled — utilizați sau nu reproiectarea "din zbor". În mod implicit const:*True* (folosiți OTF).
- topologyTolerance — toleranța topologică. Valoarea implicită este 0.
- ellipsoidID — elipsoidul de utilizat. În mod implicit "WGS84".

```
# only CRS is set, all other values are defaults
builder = QgsGraphBuilder(myCRS)
```

De asemenea, putem defini mai multe puncte, care vor fi utilizate în analiză. De exemplu

```
startPoint = QgsPoint(82.7112, 55.1672)
endPoint = QgsPoint(83.1879, 54.7079)
```

Acum că totul este la locul lui, putem să construim graful și să "legăm" aceste puncte la el

```
tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
```

Construirea unui graf poate dura ceva timp (depinzând de numărul de entități dintr-un strat și de dimensiunea stratului). `tiedPoints` reprezintă o listă cu coordonatele punctelor "asociate". Când s-a terminat operațiunea de construire putem obține graful și să-l utilizăm pentru analiză

```
graph = builder.graph()
```

Cu următorul cod putem obține indecșii punctelor noastre

```
startId = graph.findVertex(tiedPoints[0])
endId = graph.findVertex(tiedPoints[1])
```

21.3 Analiza grafului

Analiza de rețea este utilizată pentru a găsi răspunsuri la două întrebări: care noduri sunt conectate și identificarea celei mai scurte căi. Pentru a rezolva această problemă, biblioteca de analiză de rețea oferă algoritmul lui Dijkstra.

Algoritmul lui Dijkstra găsește cea mai bună cale între unul dintre vârfurile grafului și toate celelalte, precum și valorile parametrilor de optimizare. Rezultatele pot fi reprezentate ca cel mai scurt arbore.

Arborele drumurilor cele mai scurte reprezintă un graf (sau mai precis — arbore) orientat, ponderat, cu următoarele proprietăți:

- doar un singur nod nu are muchii de intrare — rădăcina arborelui
- toate celelalte noduri au numai o margine de intrare
- dacă nodul B este accesibil din nodul A, apoi calea de la A la B este singura disponibilă și este optimă (cea mai scurtă) în acest graf

To get the shortest path tree use the methods `shortestTree()` and `dijkstra()` of QgsGraphAnalyzer class. It is recommended to use method `dijkstra()` because it works faster and uses memory more efficiently.

Metoda `shortestTree()` este utilă atunci când doriți să vă plimbați de-a lungul celei mai scurte căi. Aceasta creează mereu un nou obiect de tip graf (QgsGraph) care acceptă trei variabile:

- source — graf de intrare
- startVertexIdx — Indexul punctului de pe arbore (rădăcina arborelui)
- criterionNum — numărul de proprietății marginii de folosit (începând de la 0).

```
tree = QgsGraphAnalyzer.shortestTree(graph, startId, 0)
```

Metoda `dijkstra()` are aceleași argumente, dar întoarce două matrici. În prima matrice, elementul i conține indicele marginii de intrare, sau -1 în cazul în care nu există margini de intrare. În a doua matrice, elementul i conține distanța de la rădăcina arborelui la nodul i, sau DOUBLE_MAX dacă vertexul nu poate fi accesat pornind de la rădăcină.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, startId, 0)
```

Here is some very simple code to display the shortest path tree using the graph created with the `shortestTree()` method (select linestring layer in TOC and replace coordinates with your own). **Warning**: use this code only as an example, it creates a lots of QgsRubberBand objects and may be slow on large data-sets.

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.743804, 0.22954)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)

i = 0;
while (i < tree.arcCount()):
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (tree.vertex(tree.arc(i).inVertex()).point())
  rb.addPoint (tree.vertex(tree.arc(i).outVertex()).point())
  i = i + 1
```

Același lucru, dar cu ajutorul metodei `dijkstra()`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-1.37144, 0.543836)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]
```

```
graph = builder.graph()

idStart = graph.findVertex(pStart)

(tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.arc(edgeId).inVertex()).point())
  rb.addPoint (graph.vertex(graph.arc(edgeId).outVertex()).point())
```

21.3.1 Găsirea celor mai scurte căi

Pentru a găsi calea optimă între două puncte este utilizată următoarea abordare. Ambele puncte (se începe cu A și se termină cu B) sunt "legate" de un graf, atunci când se construiește. Apoi folosind metodele `shortestTree()` sau `dijkstra()` vom construi cel mai scurt arbore cu rădăcina în punctul de pornire A. În același arbore am găsit, de asemenea, punctul de final B și începem parcurgerea arborelui de la punctul B la punctul A. Întregul algoritm poate fi scris ca

```
assign  = B
while  != A
    add point  to path
    get incoming edge for point
    look for point , that is start point of this edge
    assign  =
add point  to path
```

În acest moment avem calea, sub formă de listă inversată de noduri (nodurile sunt listate în ordine inversă, de la punctul de final către cel de start), ele fiind vizitate în timpul parcurgerii căii.

Aici este codul de test pentru consola Python a QGIS (va trebui să selectați stratul linie în TOC și să înlocuiți coordonate din cod cu ale dvs.), care folosește metoda `shortestTree()`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.835953, 0.15679)
pStop = QgsPoint(-1.1027, 0.699986)

tiedPoints = director.makeGraph(builder, [pStart, pStop])
graph = builder.graph()

tStart = tiedPoints[0]
tStop = tiedPoints[1]

idStart = graph.findVertex(tStart)
tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)
```

```
idStart = tree.findVertex(tStart)
idStop = tree.findVertex(tStop)

if idStop == -1:
  print "Path not found"
else:
  p = []
  while (idStart != idStop):
    l = tree.vertex(idStop).inArc()
    if len(l) == 0:
      break
    e = tree.arc(l[0])
    p.insert(0, tree.vertex(e.inVertex()).point())
    idStop = e.outVertex()

  p.insert(0, tStart)
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor(Qt.red)

  for pnt in p:
    rb.addPoint(pnt)
```

Iar aici este același exemplu, dar folosind metoda `dijkstra()`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.835953, 0.15679)
pStop = QgsPoint(-1.1027, 0.699986)

tiedPoints = director.makeGraph(builder, [pStart, pStop])
graph = builder.graph()

tStart = tiedPoints[0]
tStop = tiedPoints[1]

idStart = graph.findVertex(tStart)
idStop = graph.findVertex(tStop)

(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

if tree[idStop] == -1:
  print "Path not found"
else:
  p = []
  curPos = idStop
  while curPos != idStart:
    p.append(graph.vertex(graph.arc(tree[curPos]).inVertex()).point())
    curPos = graph.arc(tree[curPos]).outVertex();

  p.append(tStart)
```

```
rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
rb.setColor(Qt.red)

for pnt in p:
  rb.addPoint(pnt)
```

21.3.2 Ariile de disponibilitate

Aria de disponibilitate a nodului A este un subset de noduri ale graf-ului, care sunt accesibile din nodul A iar costurile căii de la A la aceste noduri nu sunt mai mari decât o anumită valoare.

Mai clar, acest lucru poate fi dovedit cu următorul exemplu: "Există o echipă de intervenție în caz de incendiu. Ce zone ale orașului acoperă această echipă în 5 minute? Dar în 10 minute? Dar în 15 minute?". Răspunsul la aceste întrebări îl reprezintă zonele de disponibilitate ale echipei de intervenție.

Pentru a găsi zonele de disponibilitate putem folosi metoda `dijksta()` a clasei `QgsGraphAnalyzer`. Este suficientă compararea elementelor matricei de costuri cu valoarea predefinită. În cazul în care costul[i] este mai mic sau egal decât o valoare predefinită, atunci nodul i se află în zona de disponibilitate, în caz contrar este în afară.

Mai dificilă este obținerea granițelor zonei de disponibilitate. Marginea de jos reprezintă un set de noduri care încă sunt accesibile, iar marginea de sus un set de noduri inaccesibile. De fapt, acest lucru este simplu: marginea disponibilă a atins aceste margini parcurgând arborele cel mai scurt, pentru care nodul de start este accesibil, spre deosebire de celelalt capăt, care nu este accesibil.

Iată un exemplu

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(65.5462, 57.1509)
delta = qgis.utils.iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
rb.setColor(Qt.green)
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() + delta))

tiedPoints = director.makeGraph(builder, [pStart])
graph = builder.graph()
tStart = tiedPoints[0]

idStart = graph.findVertex(tStart)

(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

upperBound = []
r = 2000.0
i = 0
```

```
while i < len(cost):
  if cost[i] > r and tree[i] != -1:
    outVertexId = graph.arc(tree [i]).outVertex()
    if cost[outVertexId] < r:
      upperBound.append(i)
  i = i + 1

for i in upperBound:
  centerPoint = graph.vertex(i).point()
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
  rb.setColor(Qt.red)
  rb.addPoint(QgsPoint(centerPoint.x() - delta, centerPoint.y() - delta))
  rb.addPoint(QgsPoint(centerPoint.x() + delta, centerPoint.y() - delta))
  rb.addPoint(QgsPoint(centerPoint.x() + delta, centerPoint.y() + delta))
  rb.addPoint(QgsPoint(centerPoint.x() - delta, centerPoint.y() + delta))
```

CHAPTER 22

Plugin-uri Python pentru Serverul QGIS

- Arhitectura Plugin-urilor de Filtrare de pe Server
  - requestReady
  - sendResponse
  - responseComplete
- Tratarea excepțiilor provenite de la un plugin
- Scrierea unui plugin pentru server
  - Fișierele Plugin-ului
  - __init__.py
  - HelloServer.py
  - Modificarea intrării
  - Modificarea sau înlocuirea rezultatului
- Plugin-ul de control al accesului
  - Fișierele Plugin-ului
  - __init__.py
  - AccessControl.py
  - layerFilterExpression
  - layerFilterSubsetString
  - layerPermissions
  - authorizedLayerAttributes
  - allowToEdit
  - cacheKey

Python plugins can also run on QGIS Server (see *label_qgisserver*): by using the *server interface* (`QgsServerInterface`) a Python plugin running on the server can alter the behavior of existing core services (**WMS**, **WFS** etc.).

With the *server filter interface* (`QgsServerFilter`) we can change the input parameters, change the generated output or even by providing new services.

With the *access control interface* (`QgsAccessControlFilter`) we can apply some access restriction per requests.

22.1 Arhitectura Plugin-urilor de Filtrare de pe Server

Server python plugins are loaded once when the FCGI application starts. They register one or more `QgsServerFilter` (from this point, you might find useful a quick look to the server plugins API docs). Each filter should implement at least one of three callbacks:

- `requestReady()`
- `responseComplete()`
- `sendResponse()`

Toate filtrele au acces la obiectul cerere/răspuns (`QgsRequestHandler`), putându-i manipula toate proprietățile (de intrare/ieșire) și tratându-i excepțiile (deși într-un mod cu totul particular, după cum vom vedea mai jos).

Mai jos se află un pseudocod care prezintă o sesiune tipică de server și reapelarea filtrelor:

- **se obține cererea de intrare**
    - se creează o rutină de tratare a cererilor GET/POST/SOAP
    - se transmite cererea către o instanță a clasei `QgsServerInterface`
    - se apelează filtrele `requestReady()` ale plugin-urillor
    - **în cazul în care nu există un răspuns**
        - **dacă SERVICE este de tipul WMS/WFS/WCS**
            - **se creează serverul WMS/WFS/WCS**

              call server's `executeRequest()` and possibly call `sendResponse()` plugin filters when streaming output or store the byte stream output and content type in the request handler
        - se apelează filtrele `responseComplete()` ale plugin-urillor
    - se apelează filtrele `sendResponse()` ale plugin-urillor
    - request handler output the response

Următoarele paragrafe descriu, în detaliu, funcțiile de reapelare disponibile.

22.1.1 requestReady

Este apelată atunci când cererea este pregătită: adresa și datele primite au fost analizate și, înainte de a intra în comutatorul serviciilor de bază (WMS, WFS, etc), acesta este punctul în care se poate interveni asupra datelor de intrare, putându-se efectua acțiuni de genul:

- autentificare/autorizare
- redirectări
- adăugarea/eliminarea anumitor parametri (denumirile tipurilor, de exemplu)
- tratarea excepțiilor

Ați putea chiar să substituiți în întregime un serviciu de bază, prin schimbarea parametrului **SERVICE**, astfel, ocolindu-se complet serviciul de bază (deși, acest lucru nu ar avea prea mult sens).

22.1.2 sendResponse

This is called whenever output is sent to **FCGI** `stdout` (and from there, to the client), this is normally done after core services have finished their process and after responseComplete hook was called, but in a few cases XML can become so huge that a streaming XML implementation was needed (WFS GetFeature is one of them), in this case, `sendResponse()` is called multiple times before the response is complete (and before `responseComplete()` is called). The obvious consequence is that `sendResponse()` is normally called once but might be exceptionally called multiple times and in that case (and only in that case) it is also called before `responseComplete()`.

`sendResponse()` reprezintă cel mai bun loc pentru manipularea directă a rezultatului serviciilor de bază și, în timp ce `responseComplete()` constă, de asemenea, într-o opțiune, `sendResponse()` este singura opțiune viabilă în cazul serviciilor de redare continuă a fluxului.

22.1.3 responseComplete

This is called once when core services (if hit) finish their process and the request is ready to be sent to the client. As discussed above, this is normally called before `sendResponse()` except for streaming services (or other plugin filters) that might have called `sendResponse()` earlier.

`responseComplete()` reprezintă locul ideal pentru implementarea unor noi servicii (WPS sau servicii personalizate), precum și pentru a efectua intervenții directe asupra rezultatului care provine de la serviciile de bază (de exemplu, pentru a adăuga un filigran pe o imagine WMS).

22.2 Tratarea excepțiilor provenite de la un plugin

Mai sunt ceva acțiuni de efectuat în acest capitol: implementarea curentă poate distinge între excepțiile tratate și cele netratate, prin setarea proprietății `QgsRequestHandler` pentru o instanță a clasei `QgsMapServiceException`; în acest fel, codul C++ principal poate să intercepteze excepțiile Python tratate și să le ignore pe cele netratate (sau mai bine: să le jurnalizeze).

Această abordare funcționează în principiu, dar nu este în spiritul limbajului "python": o abordare mai bună ar fi de a face vizibile excepțiile din codul python la nivelul buclei C++, pentru a fi manipulată acolo.

22.3 Scrierea unui plugin pentru server

A server plugins is just a standard QGIS Python plugin as described in *Dezvoltarea plugin-urilor Python*, that just provides an additional (or alternative) interface: a typical QGIS desktop plugin has access to QGIS application through the `QgisInterface` instance, a server plugin has also access to a `QgsServerInterface`.

Pentru a spune Serverului QGIS că un plugin are o interfață de server, este necesară o intrare de metadate specială (în *metadata.txt*)

```
server=True
```

The example plugin discussed here (with many more example filters) is available on github: QGIS HelloServer Example Plugin

22.3.1 Fișierele Plugin-ului

Iată structura de directoare a exemplului nostru de plugin pentru server

```
PYTHON_PLUGINS_PATH/
  HelloServer/
    __init__.py    --> *required*
    HelloServer.py  --> *required*
    metadata.txt   --> *required*
```

22.3.2 __init__.py

This file is required by Python's import system. Also, QGIS Server requires that this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin's class. This is how the example plugin *__init__.py* looks like:

```
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from HelloServer import HelloServerServer
    return HelloServerServer(serverIface)
```

22.3.3 HelloServer.py

Aici este locul în care se întâmplă magia, și iată rezultatul acesteia: (de exemplu `HelloServer.py`)

Un plug-in de server este format, de obicei, dintr-una sau mai multe funcții Callback, ambalate în obiecte denumite QgsServerFilter.

Fiecare `QgsServerFilter` implementează una sau mai multe dintre următoarele funcții callback:

- `requestReady()`
- `responseComplete()`
- `sendResponse()`

Exemplul următor implementează un filtru minimal, care generează textul *HelloServer!* atunci când parametrul **SERVICE** este egal cu "HELLO":

```
from qgis.server import *
from qgis.core import *

class HelloFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(HelloFilter, self).__init__(serverIface)

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap()
        if params.get('SERVICE', '').upper() == 'HELLO':
            request.clearHeaders()
            request.setHeader('Content-type', 'text/plain')
            request.clearBody()
            request.appendBody('HelloServer!')
```

Filtrele trebuie să fie înregistrate în **serverIface** ca în exemplul următor:

```
class HelloServerServer:
    def __init__(self, serverIface):
        # Save reference to the QGIS server interface
        self.serverIface = serverIface
        serverIface.registerFilter( HelloFilter, 100 )
```

Al doilea parametru al funcției `registerFilter()` permite stabilirea unei priorități, care definește ordinea pentru funcțiile callback cu același nume (prioritatea inferioară este invocată mai întâi).

Prin utilizarea celor trei funcții callback, plugin-urile pot manipula intrarea și/sau ieșirea serverului în mai multe moduri diferite. În orice moment, instanța plugin-ului are acces la `QgsRequestHandler` prin intermediul clasei `QgsServerInterface`. Clasa `QgsRequestHandler` dispune de o mulțime de metode care pot fi utilizate pentru modificarea parametrilor de intrare, înainte de a intra în nucleul de prelucrare al serverului (prin utilizarea `requestReady()`) sau în urma procesării cererii de către serviciile de bază (prin utilizarea `sendResponse()`).

Următorul exemplu demonstrează câteva cazuri de utilizare obișnuită:

22.3.4 Modificarea intrării

The example plugin contains a test example that changes input parameters coming from the query string, in this example a new parameter is injected into the (already parsed) `parameterMap`, this parameter is then visible by core services (WMS etc.), at the end of core services processing we check that the parameter is still there:

```
from qgis.server import *
from qgis.core import *

class ParamsFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(ParamsFilter, self).__init__(serverIface)

    def requestReady(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        request.setParameter('TEST_NEW_PARAM', 'ParamsFilter')

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        if params.get('TEST_NEW_PARAM') == 'ParamsFilter':
            QgsMessageLog.logMessage("SUCCESS - ParamsFilter.responseComplete", 'plugin', QgsMess
        else:
            QgsMessageLog.logMessage("FAIL    - ParamsFilter.responseComplete", 'plugin', QgsMess
```

This is an extract of what you see in the log file:

```
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloServerServ
src/core/qgsmessagelog.cpp: 45: (logMessage) [1ms] 2014-12-12T12:39:29 Server[0] Server plugin H
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 Server[0] Server python p
src/mapserver/qgsgetrequesthandler.cpp: 35: (parseInput) [0ms] query string is: SERVICE=HELLO&re
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [1ms] inserting pair
src/mapserver/qgshttprequesthandler.cpp: 547: (requestStringToParameterMap) [0ms] inserting pair
src/mapserver/qgsserverfilter.cpp: 42: (requestReady) [0ms] QgsServerFilter plugin default reque
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.req
src/mapserver/qgis_map_serv.cpp: 235: (configPath) [0ms] Using default configuration file path: 
src/mapserver/qgshttprequesthandler.cpp: 49: (setHttpResponse) [0ms] Checking byte array is ok t
src/mapserver/qgshttprequesthandler.cpp: 59: (setHttpResponse) [0ms] Byte array looks good, sett
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.res
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] SUCCESS - Param
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] RemoteConsoleFi
src/mapserver/qgshttprequesthandler.cpp: 158: (sendResponse) [0ms] Sending HTTP response
src/core/qgsmessagelog.cpp: 45: (logMessage) [0ms] 2014-12-12T12:39:29 plugin[0] HelloFilter.sen
```

On the highlighted line the "SUCCESS" string indicates that the plugin passed the test.

Aceeași tehnică poate fi exploatată pentru a utiliza un serviciu personalizat in locul unuia de bază: de exemplu, ați putea sări peste o cerere **WFS SERVICE**, sau peste oricare altă cerere de bază, doar prin schimbarea parametrului **SERVICE** în ceva diferit, iar serviciul de bază va fi omis; în acel caz, veți puteți injecta datele dvs. în interiorul rezultatului, trimițându-le clientului (acest lucru este explicat în continuare).

22.3.5 Modificarea sau înlocuirea rezultatului

The watermark filter example shows how to replace the WMS output with a new image obtained by adding a watermark image on the top of the WMS image generated by the WMS core service:

```
import os

from qgis.server import *
```

```
from qgis.core import *
from PyQt4.QtCore import *
from PyQt4.QtGui import *


class WatermarkFilter(QgsServerFilter):

    def __init__(self, serverIface):
        super(WatermarkFilter, self).__init__(serverIface)

    def responseComplete(self):
        request = self.serverInterface().requestHandler()
        params = request.parameterMap( )
        # Do some checks
        if (request.parameter('SERVICE').upper() == 'WMS' \
                and request.parameter('REQUEST').upper() == 'GETMAP' \
                and not request.exceptionRaised() ):
            QgsMessageLog.logMessage("WatermarkFilter.responseComplete: image ready %s" % request
            # Get the image
            img = QImage()
            img.loadFromData(request.body())
            # Adds the watermark
            watermark = QImage(os.path.join(os.path.dirname(__file__), 'media/watermark.png'))
            p = QPainter(img)
            p.drawImage(QRect( 20, 20, 40, 40), watermark)
            p.end()
            ba = QByteArray()
            buffer = QBuffer(ba)
            buffer.open(QIODevice.WriteOnly)
            img.save(buffer, "PNG")
            # Set the body
            request.clearBody()
            request.appendBody(ba)
```

În cadrul acestui exemplu, este verificată valoarea parametrului **SERVICE**, iar în cazul în care cererea de intrare este un **WMS GETMAP** și nici un fel de excepții nu au fost stabilite de către un plugin executat anterior, sau de către serviciul de bază (WMS în acest caz), imaginea generată de către WMS este preluată din zona tampon de ieșire, adăugându-i-se imaginea filigran. Pasul final este de a goli tamponul de ieșire și de-l înlocui cu imaginea nou generată. Rețineți că într-o situație reală, ar trebui, de asemenea, să verificați tipul imaginii solicitate în loc de a returna, în toate cazurile, PNG-ul.

22.4 Plugin-ul de control al accesului

22.4.1 Fișierele Plugin-ului

Iată structura de directoare a exemplului nostru de plugin pentru server:

```
PYTHON_PLUGINS_PATH/
  MyAccessControl/
    __init__.py      --> *required*
    AccessControl.py   --> *required*
    metadata.txt     --> *required*
```

22.4.2 __init__.py

This file is required by Python's import system. As for all QGIS server plugins, this file contains a `serverClassFactory()` function, which is called when the plugin gets loaded into QGIS Server when the

server starts. It receives reference to instance of `QgsServerInterface` and must return instance of your plugin's class. This is how the example plugin *__init__.py* looks like:

```
# -*- coding: utf-8 -*-

def serverClassFactory(serverIface):
    from MyAccessControl.AccessControl import AccessControl
    return AccessControl(serverIface)
```

22.4.3 AccessControl.py

```
class AccessControl(QgsAccessControlFilter):

    def __init__(self, server_iface):
        super(QgsAccessControlFilter, self).__init__(server_iface)

    def layerFilterExpression(self, layer):
        """ Return an additional expression filter """
        return super(QgsAccessControlFilter, self).layerFilterExpression(layer)

    def layerFilterSubsetString(self, layer):
        """ Return an additional subset string (typically SQL) filter """
        return super(QgsAccessControlFilter, self).layerFilterSubsetString(layer)

    def layerPermissions(self, layer):
        """ Return the layer rights """
        return super(QgsAccessControlFilter, self).layerPermissions(layer)

    def authorizedLayerAttributes(self, layer, attributes):
        """ Return the authorised layer attributes """
        return super(QgsAccessControlFilter, self).authorizedLayerAttributes(layer, attributes)

    def allowToEdit(self, layer, feature):
        """ Are we authorise to modify the following geometry """
        return super(QgsAccessControlFilter, self).allowToEdit(layer, feature)

    def cacheKey(self):
        return super(QgsAccessControlFilter, self).cacheKey()
```

Acest exemplu oferă un acces deplin pentru oricine.

Este de datoria plugin-ului să știe cine este conectat.

Toate aceste metode au ca argument stratul, pentru a putea personaliza restricțiile pentru fiecare strat.

22.4.4 layerFilterExpression

Se folosește pentru a adăuga o Expresie de limitare a rezultatelor, ex.:

```
def layerFilterExpression(self, layer):
    return "$role = 'user'"
```

Pentru restrângerea la entitățile pentru care atributul "rol" are valoarea "user".

22.4.5 layerFilterSubsetString

La fel ca și precedenta, dar folosește `SubsetString` (executată în baza de date)

```
def layerFilterSubsetString(self, layer):
    return "role = 'user'"
```

Pentru restrângerea la entitățile pentru care atributul "rol" are valoarea "user".

22.4.6 layerPermissions

Limitează accesul la strat.

Returnează un obiect de tip `QgsAccessControlFilter.LayerPermissions`, care are proprietățile:

- `canRead` pentru a-l vedea în `GetCapabilities` și pentru a avea acces de citire.
- `canInsert` pentru a putea insera o nouă entitate.
- `canUpdate` pentru a putea actualiza o entitate.
- `candelete` pentru a fi putea șterge o entitate.

Exemplu:

```
def layerPermissions(self, layer):
    rights = QgsAccessControlFilter.LayerPermissions()
    rights.canRead = True
    rights.canRead = rights.canInsert = rights.canUpdate = rights.canDelete = False
    return rights
```

Pentru a permite tuturor accesul numai pentru citire.

22.4.7 authorizedLayerAttributes

Folosit pentru a reduce vizibilitatea unui subset specific de atribute.

Atributul argument returnează setul actual de atribute vizibile.

Exemplu:

```
def authorizedLayerAttributes(self, layer, attributes):
    return [a for a in attributes if a != "role"]
```

Pentru a ascunde atributul 'role'.

22.4.8 allowToEdit

Se folosește pentru a limita editarea unui subset specific de entități.

Este folosit în protocolul `WFS-Transaction`.

Exemplu:

```
def allowToEdit(self, layer, feature):
    return feature.attribute('role') == 'user'
```

Pentru a putea modifica numai entitatea pentru care atributul "rol" are valoarea de "utilizator".

22.4.9 cacheKey

Serverul QGIS menține o memorie tampon a capabilităților, de aceea, pentru a avea o memorie cache pentru fiecare rol, puteți specifica rolul cu ajutorul acestei metode. Sau puteți seta valoarea `None`, pentru a dezactiva complet memoria tampon.

Index

API, 1
Authentication config, **72**
Authentication Configuration, **72**
Authentication DB, **72**
Authentication Method, **72**

Baza de Date de Autentificare, **72**

Calculating values, 50
Categorized symbology renderer, 27
Console
    Python, 2
Custom
    Renderer, 31
Custom applications
    Python, 3
    Running, 4

Delimited text files
    Loading, 10

Environment
    PYQGIS_STARTUP, 1
Expressions, 50
    Evaluating, 52
    Parsing, 52

Filtering, 50

Geometry
    Access to, 36
    Construction, 35
    Handling, 33
    Predicates and operations, 36
GPX files
    Loading, 10
Graduated symbol renderer, 28

Iterating features, 18

Loading
    Delimited text files, 10
    GPX files, 10
    MySQL geometries, 10
    OGR layers, 9
    PostGIS layers, 9
    Projects, 7
    Raster layers, 11
    SpatiaLite layers, 10
    Vector layers, 9
    WFS vector, 10
    WMS raster, 11

Map canvas, 40
    Custom canvas items, 45
    Custom map tools, 44
    Embedding, 41
    Map tools, 42
    Rubber bands, 43
    Vertex markers, 43
map canvas
    architecture, 41
Map layer registry, 11
    Adding a layer, 11
Map printing, 46
Map rendering, 46
    Simple, 47
Memory layer, 24
Metadata, 107
metadata, 107
metadata.txt, 63, 107
MySQL geometries
    Loading, 10

OGR layers
    Loading, 9
Output
    PDF, 50
    Raster image, 50
    Using Map Composer, 48

Parola Master, **71**
Plugin layers, 84
    Subclasarea QgsPluginLayer, 85
Plugins
    Access attributes of selected features, 93
    Adding shortcut, 93
    Code snippets, 67
    Debugging, 78
    Depozitul oficial al plugin-urilor python, 90
    Developing, 59, 69

Documentație, 67
Implementing help, 67
Initialisation, 64
Metadata, 63
metadata.txt, 107
Processing algorithm, 94
Releasing, 87
Resource file, 66
Toggle layers, 93
Traducerea, 67
User interaction, 56
Writing, 62
Writing code, 62

plugins
testing, 84

PostGIS layers
Loading, 9

Projections, 40

Projects
Loading, 7

PyQGIS
Vector layers, 17

PYQGIS_STARTUP
Environment, 1

Python
Console, 2
Custom applications, 3
Developing plugins, 59
Developing server plugins, 104
Infrastructura de autentificare, 69
Plugins, 2
Standalone scripts, 3
startup, 1
startup.py, 2

Querying
Raster layers, 15

Raster
Raster layers, 12

Raster layers
Details, 13
Loading, 11
Multi band, 14
Querying, 15
Raster, 12
Refreshing, 15
Render, 13
Single band, 14

Refreshing
Raster layers, 15

Renderer
Custom, 31

resources.qrc, 66

Running
Custom applications, 4

Selectarea entităților, 17

Server plugins
Developing, 104

server plugins
metadata.txt, 107

Settings
Global, 55
Map layer, 56
Project, 55
Reading, 53
Storing, 53

Single symbol renderer, 26

Sisteme de coordonate de referință, 39

Spatial index, 23

SpatiaLite layers
Loading, 10

Standalone scripts
Python, 3

startup
Python, 1

Symbol layers
Creating custom types, 29
Working with, 29

Symbology
Categorized symbol renderer, 27
Graduated symbol renderer, 28
Single symbol renderer, 26

Symbols
Working with, 28

Vector layers
Creating, 23
Editing, 20
Loading, 9
Symbology, 25

WFS vector
Loading, 10

WMS raster
Loading, 11