# Developing MeeGo apps with Python and QML

by Thomas Perl <m@thp.io>
http://thp.io/2010/meego-python/

## Table of Contents

Introduction....3
- Why should I use Python to develop MeeGo apps?....3
- Links....3

Setting up the environment....3
- Getting the build scripts....4
- Building PySide and installing into $HOME....4
- Setting up environment variables....5
- Testing the installations....5

Basic QML tutorial examples....6
- Hello World....6
  - The Python source (HelloMeeGo.py)....6
  - The QML UI file (HelloMeeGo.qml)....8
- Displaying HTML content in a QML WebView....9
  - The Python source code (WebKitView.py)....9
  - The QML content (WebKitView.qml)....11
  - The HTML page (WebKitView.html)....11

Writing a new QML UI for existing apps for MeeGo....13
- A QML UI for gPodder....13
  - The glue layer in Python (gpodder-qml.py)....14
  - The QML main UI file (gpodder-qml.qml)....18
  - The QML file for displaying a list of podcasts (PodcastList.qml)....21
  - The QML file for displaying a list of episodes (EpisodeList.qml)....24

Packaging a Python application for MeeGo....25
- Installing the required dependencies....25
- Creating the required metadata files....25
  - The desktop entry: gpodder-qml.desktop....25
  - The icon for the app menu: gpodder-qml.png....25
  - The Spectacle YAML file: gpodder-qml.yaml....26
  - The Python distutils file: setup.py....26
- Changes in the gpodder-qml.py script....26
- Creating the RPM sources....27
- Building the RPM package from source....27
- Installing and testing the RPM package....27

## Introduction

This tutorial will guide you through how to set up a PySide environment on your MeeGo Netbook and then show you some basics through examples, and finally we'll create a QML UI for an existing application (gPodder) that can be used on MeeGo Netbook and MeeGo Handset.

### Why should I use Python to develop MeeGo apps?

- Low barrier to entry: Python is a very easy to learn language, so you can get quickly up to speed – independent of whether or not you are already familiar with other languages
- Garbage collection: You don't have to manually manage the memory of the objects you create – the Python garbage collector takes care of removing no longer needed objects
- No compiling: Python is an interpreted language, so you can run your application right after saving the source in an editor. No need to wait for code to compile. This is especially important on low-powered netbooks.
- Full access to the Qt libraries: PySide bindings allow access to all modules of Qt. And because it uses the native Qt libraries, library functions run at native (compiled) speed.
- Shorter code: In my experience, C++ applications using Qt have about 3 times as much lines of code as the equivalent Python applications – using the same libraries and Qt classes!
- Prototyping: Even if you plan on writing a C++ Qt application, Python and PySide are a great combination for quickly prototyping your ideas. This could be useful to prototype a C++ backend of your QML application. Later on, you just exchange the Python backend with a C++ backend and can reuse the QML files from the Python app.
- Development on the go: As Python is an interpreted language, the runtime already contains all the necessary tools to develop applications, so you don't have to install a compiler, development libraries and header files just to create apps – it's instant, and some people even develop Python GUI apps directly on handset devices like the N900.

### Links

- PySide Homepage: http://www.pyside.org/
- MeeGo Homepage: http://www.meego.com/
- PySide Wiki: http://developer.qt.nokia.com/wiki/PySide
- Tutorial homepage: http://thp.io/2010/meego-python/

## Setting up the environment

First, make sure to have MeeGo Netbook installed – either directly on your netbook or in a virtual machine. How to do this is out of scope for this document, so please refer to the MeeGo Wiki.

As you probably want to start developing with PySide right now, and given the agile state of PySide at this moment (lots of bugfixes being integrated before the 1.0 release), it's a good idea to build PySide from source. This will not be necessary in the future (when PySide packages are hopefully integrated into MeeGo), but you might nevertheless use an up-to-date version for development.

## Getting the build scripts

I've created a set of build scripts that will automate the building of PySide from the Git repositories, including installing required build dependencies, so you can get up to speed quickly.

1. Click on the red “Applications” icon on MeeGo
2. Search for “Terminal” and start the application
3. Install Git via “sudo zypper install git”
4. Create a “pyside” folder in your $HOME with “mkdir pyside”
5. Change into the “pyside” folder: “cd pyside”
6. Check out the source: “git clone git://gitorious.org/pyside/buildscripts.git”
7. Change into the “buildscripts” folder: “cd buildscripts”
8. Fetch the sources: “git submodule init” and “git submodule update”
9. Fetching the sources will take a while – grab a coffee or a tea :)

After the buildscripts have been downloaded, you can use them to build PySide for MeeGo Netbook.

## Building PySide and installing into $HOME

Now that you have gotten all the sources, you can build PySide for your netbook and install it into your home directory so that it does not conflict with any system-wide installations you might have in place. That's a safe way to install PySide from Git, as you can always just remove it from $HOME or reinstall it as often as you wish without cluttering up your system directories.

1. Open another Terminal (or use the one you still have open from the previous step)
2. Go into the PySide buildscripts folder: “cd ~/pyside/buildscripts”
3. Install the build dependencies: “sudo ./dependencies.meego.sh”
4. Before the next step, make sure to close down any other apps you might have running – building PySide takes a lot of RAM. If you have problems building PySide, consider adding some RAM or adding a swapfile
5. (started 14;28) Build and install everything using “./build_and_install”
6. This will take about 2.5 hours (on an Atom single-core netbook) – yes, that's a lot, but the upside of using Python for development on your MeeGo netbook is that your applications do not have to be compiled, so instead of a develop-compile-run cycle, your have a develop-run cycle. This is especially nice on low-powered netbooks and saves you time and battery power on the go.

## Setting up environment variables

Because we installed PySide into your $HOME, you have to set up some environment variables for that PySide version to become the active one for Python to use. Luckily, the “environment.sh” script in the PySide buildscripts takes care of that. You can either use

```
source ~/pyside/buildscripts/environment.sh
```

every time you want to do PySide development, or add this command to your ~/.bashrc file, so it gets executed every time you open a shell or terminal window (that's the recommended way to do it, so you don't forget to source the “environment.sh” file when trying out the examples).

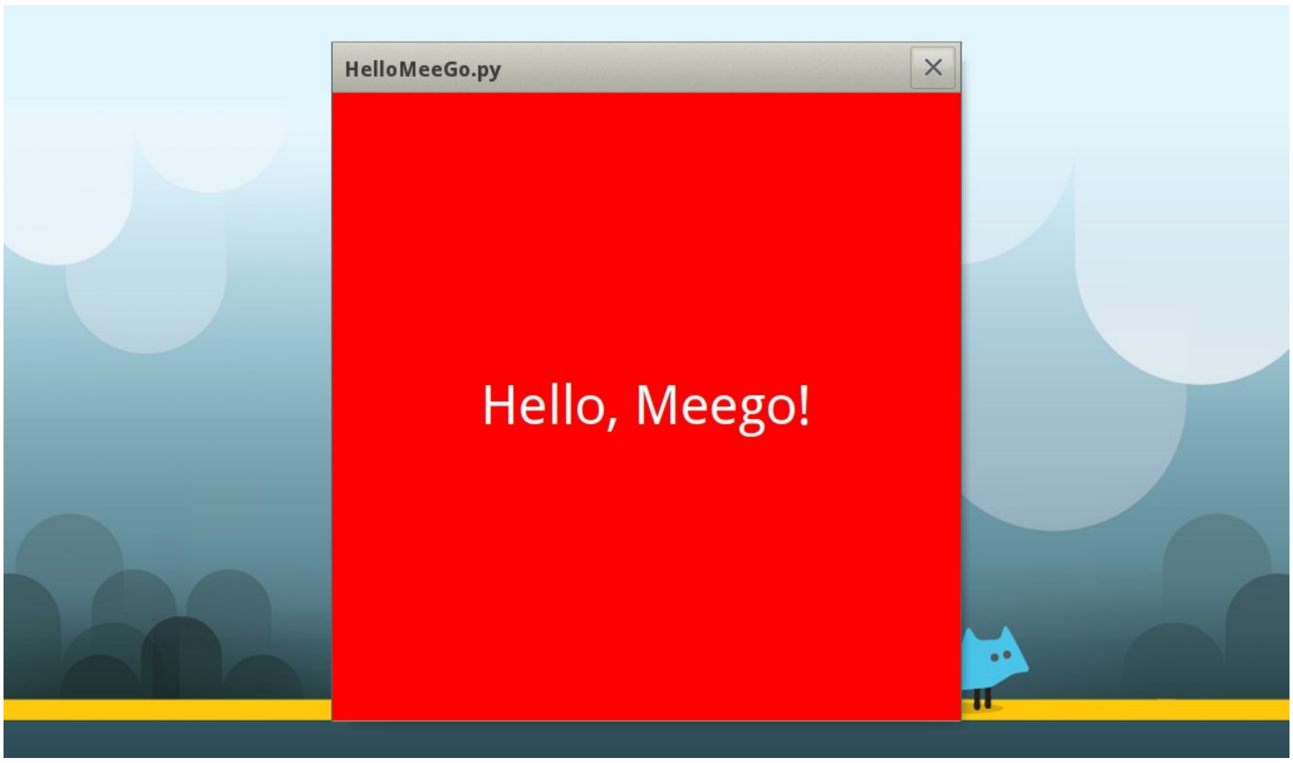
## Testing the installations

Before we can start to write great PySide apps for MeeGo, let's see if everything was installed correctly. Fire up another terminal window and enter “python” to start the interactive Python shell. Now, enter “from PySide import QtGui” to check if the QtGui module was correctly installed. After that, try “from PySide import QtDeclarative” to check if the QtDeclarative module is working. We need the QtGui for basic UI classes (non-QML), and QtDeclarative for loading and displaying QML content.

## Basic QML tutorial examples

This section should give you a short overview with code examples on how to do simple things with PySide and QML. More examples can be found on the homepage of this tutorial on http://thp.io/2010/meego-python/

### Hello World



This example shows you how to create a minimalistic Hello world QML application with Python. We already subclass QObject here and provide a simple property ('greeting') that we can then access from QML. Let's dive right into the source code.

#### The Python source (HelloMeeGo.py)

For our first hello world program, we want to generate a greeting in Python and show it in a QML UI. We do this by subclassing QObject, giving our subclass a property called “greeting” and exposing an instance of that object to the QML root context, where we can access it from the QML file.

```
# -*- coding: utf-8 -*-

import sys

from PySide.QtCore import *
from PySide.QtGui import *
from PySide.QtDeclarative import *
```

We need the “sys” module (a Python standard module) to access the command line arguments (we need to pass them to the constructor of QApplication). From PySide, we need QtCore (which contains core objects, like QObject in our example) and QtGui (which contains Qapplication, which we need for the Qt main loop). The QML view is provided by the QtDeclarative module – it provides QDeclarativeView.

```
class Hello(QObject):
    def get_greeting(self):
        return u'Hello, Meego!'

    greeting = Property(unicode, get_greeting)
```

This is the definition of our “Hello” class – we provide a getter method for the greeting, and return a unicode string “Hello, MeeGo!” in it – if you want, you can also customize this greeting, i.e. add the time and date using the “datetime” Python module. In order for QML to be able to access this property, we have to declare it as such by using “Property” (which is in QtCore). The first parameter of Property is the type (unicode means unicode string) and the second one is the getter method – if you want the property to be modifyable, you need to add a setter method, and if you want it to be dynamically updated, you also need a notifyable property.

```
app = QApplication(sys.argv)

hello = Hello()

view = QDeclarativeView()
```

Here we create instances of the classes we need:

- QApplication is needed by every Qt application and handles command-line arguments, sets up the graphics system and does other initializations. It also provides the main loop through the “exec_” method.
- Hello is our class defined above. We create an instance of it that we later pass to QML using context properties.
- QDeclarativeView is the window / view in which we can load QML content and display it on the screen.

```
context = view.rootContext()
context.setContextProperty('hello', hello)
```

For QML to be able to access our “hello” object, we need to expose it as a context property to the view's root context – this is done using setContextProperty. As property name we use “hello”, so we can access the greeting of that object later using “hello.greeting” in QML.

```
view.setSource(__file__.replace('.py', '.qml'))
```

Here the QML file is loaded and displayed in the view. As the QML file has the same name as our Python script (just a different file extension), we can use __file__.replace('.py', '.qml') to always get the correct file name – this also allows you to easily rename both files and still have them work together as expected (for example, if you want to try to create different variants of this example).

```
view.show()
app.exec_()
```

And finally, here our application starts: We always have to call the show() method on the view, or otherwise it won't be shown on the screen. If you want to show the window in fullscreen mode, use “showFullScreen” here – try it out!

In order to start the Qt main loop and process events, we have to call app.exec_(), so the application does not quit. This is very important.

### The QML UI file (HelloMeeGo.qml)

The UI definition is placed in “.qml” files – these have JavaScript-like syntax, and describe the appearance of your application. In our case, we simply want to have a red rectangle in which we place the greeting in a white font.

```
import Qt 4.7
```

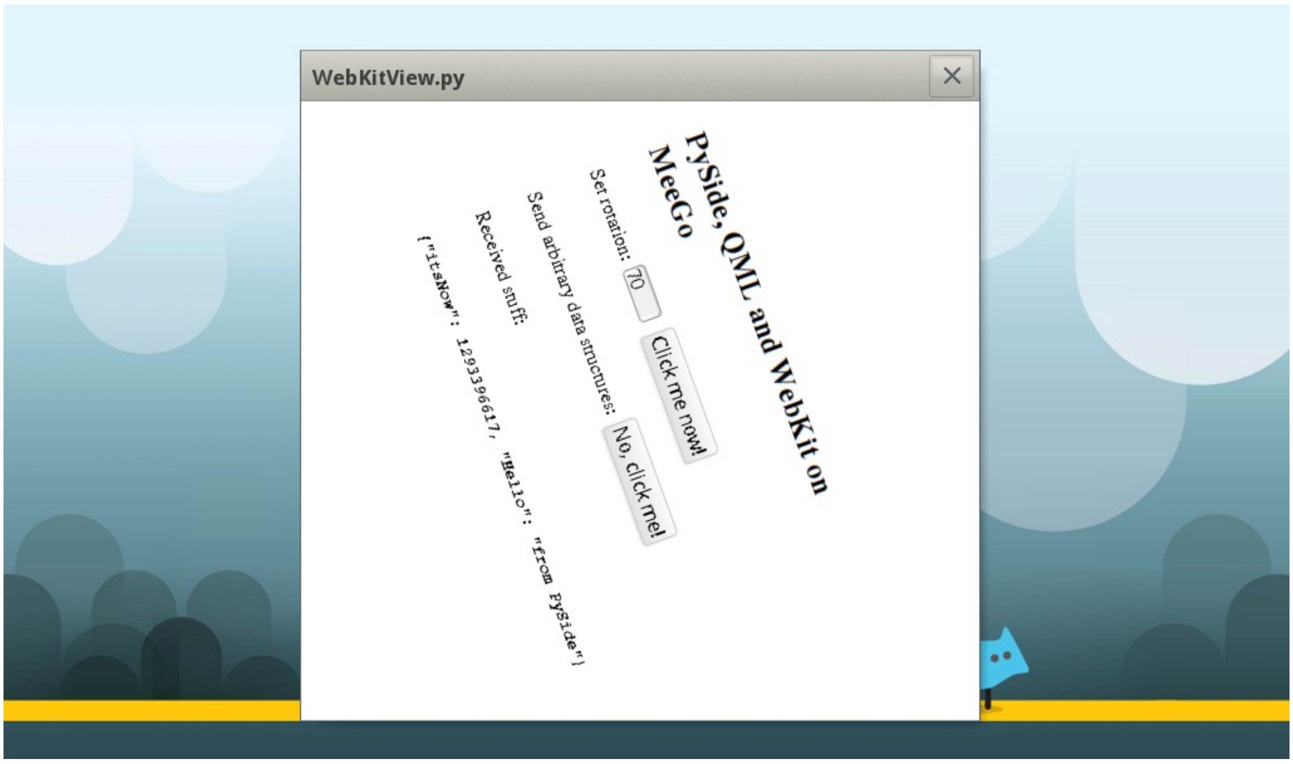
In order to use the built-in QML components like “Rectangle” and “Text”, we need to import the Qt 4.7 module into our QML file. In future versions of Qt (4.7.1 and newer), this will be called QtQuick 1.0, but for now, it's called Qt 4.7, and this is what you have to use.

```
Rectangle {
    color: "red"
    width: 500
    height: 500
    Text {
        anchors.centerIn: parent
        font.pointSize: 32
        color: "white"
        text: hello.greeting
    }
}
```

The root object is a 400x400 pixel, red rectangle, which contains a child Text component that is centered into its parent (i.e. the red rectangle). The text is 32pt in size and has a white color. Its text is taken from the “hello” object (our Python object instance) as the “greeting” property (which we have defined in the Python source code).

Save these two files and start the example using  “python HelloMeeGo.py”. You should see a window like the one in the screenshot above. Try it out, and experiment with changing some properties.

## Displaying HTML content in a QML WebView



```
{"itsNow": 1293396617, "Hello": "from PySide"}
```

This short tutorial shows you how to combine the powers of Python, QML, HTML and JavaScript to create good-looking, rich web applications or netbook/handset applications that can display web content. This application consists of three files: The Python source, the HTML content and the QML view (but as most of the interaction takes place in Python and HTML, the QML is really short). This example has been ported from my other PySide/QML examples to MeeGo, and the transition was very easy, as PySide code is very portable across devices.

### The Python source code (WebKitView.py)

What we need in the Python world is a way to send data to the HTML view, and also a way to receive the data – this is done by evaluating JavaScript code inside the WebView and by listening to "alert()" calls from the web view. Communication happens by using JSON to encode data structures, as alert() does not allow to send arbitrary data.

```python
# -*- coding: utf-8 -*-

import sys
import time

try:
    import simplejson as json
except ImportError:
    import json

from PySide import QtCore, QtGui, QtDeclarative
```

We need the standard Python modules “sys” and “time”, and the Python json module (included in the Python version shipped with MeeGo Netbook 1.1) to encode and decode JSON data in Python. From PySide, we need our “usual suspect” modules – QtCore, QtGui and QtDeclarative. These three modules are always needed when you want to do something with PySide and QML.

```
def sendData(data):
    global rootObject
    print 'Sending data:', data
    json_str = json.dumps(data).replace('"', '\\"')
    rootObject.evaluateJavaScript('receiveJSON("%s")' % json_str)
```

In order to send data to the WebView, we need to get a reference to the root object of our QML (the root object is the web view) and then evaluate some javascript inside it. The assumption here is that our HTML file has a “receiveJSON” function declared in its JavaScript code which receives the data and handles it. See below for what the receiveJSON function does in the HTML code.

```
def receiveData(json_str):
    global rootObject

    data = json.loads(json_str)
    print 'Received data:', data

    if len(data) == 2 and data[0] == 'setRotation':
        animation = QtCore.QPropertyAnimation(rootObject, 'rotation', rootObject)
        animation.setDuration(3000)
        animation.setEasingCurve(QtCore.QEasingCurve.InOutElastic)
        animation.setEndValue(data[1])
        animation.start(QtCore.QAbstractAnimation.DeleteWhenStopped)
    else:
        sendData({'Hello': 'from PySide', 'itsNow': int(time.time())})
```

The receiving of data from HTML is done by the receiveData function – it will be connected to the “alert” signal of the WebView, which gets sent when “alert()” is called from somewhere inside the WebView. What this does is decode the received string from JSON to a Python data structure and then check the contents of the received data – if it's a two-item list, and the first item is a string “setRotation”, we interpret the second value as the target rotation value, and use a QPropertyAnimation on the root object to rotate the QML component inside the QML view – because all QML components have standard Qt properties that can be animated! If it's not a request for rotation, we simply send an arbitrary data structure to the HTML view as a “reply”.

```
app = QtGui.QApplication(sys.argv)

view = QtDeclarative.QDeclarativeView()
view.setRenderHints(QtGui.QPainter.SmoothPixmapTransform)
view.setSource(__file__.replace('.py', '.qml'))
rootObject = view.rootObject()
rootObject.setProperty('url', __file__.replace('.py', '.html'))
rootObject.alert.connect(receiveData)
view.show()

app.exec_()
```

This code creates instances of the QApplication and the QDeclarativeView. The SmoothPixmapTransform render hint makes the rotated web view look smoother and not so pixellated. We then load the QML file and set the “url” property of our root object (a WebView) to point to the HTML file in the same directory. We also need to hook up the “alert” signal of the root object to our custom callback “receiveData” to handle data sent from the HTML.

As always, we need to show the view (so it gets displayed on the screen) and finally call “exec_” on the QApplication instance in order to start the Qt main loop.

**The QML content (WebKitView.qml)**

This is very, very short and shows just how easy it is to create a WebView in QML – first, you need to import QtWebKit 1.0, as the WebView component is not included in the default Qt QML module. Then, we create a WebView component as our root object, enable javascript for it (otherwise we won't be able to run any JavaScript inside it – it's disabled by default) and resize it to 400x280. We do not yet set the “url” property of it to load the HTML, but instead we do that directly from our Python code (see above) to show you how to modify properties in QML components directly from Python code.

```
import QtWebKit 1.0

WebView { settings.javascriptEnabled: true; width: 400; height: 280 }
```

**The HTML page (WebKitView.html)**

This is the HTML content that gets rendered in the WebView QML component. We need to have some JavaScript in its head (which is the most interesting part here) that is capable of sending and receiving data to and from our Python code, so we can connect both worlds and send commands and data back and forth.

```
<html>
    <head>
        <script type="text/javascript">
            function sendJSON(data) {
                alert(JSON.stringify(data));
            }
```

The sendJSON function accepts arbitrary JavaScript data structures (e.g. arrays) and encodes them with JSON and then uses “alert()” to send it to Python.

```
            function receiveJSON(data) {
                element = document.getElementById('received');
                element.innerHTML += "\n" + data;
            }
```

The receiveJSON function gets called from the Python code directly and receives one string (already in JavaScript data format, not encoded in JSON) and adds it to our HTML document. We could do more sophisticated processing of the data here if need be.

```
            function setRotation() {
                element = document.getElementById('rotate');
                angle = parseInt(element.value);
                message = ['setRotation', angle];
                sendJSON(message);
            }
```

This is a convenience function used by the HTML below to create a “set rotation” command message to be sent to the Python code. It's a simple two-element list with “setRotation” as first element and the angle as second element. The angle is taken from the input element on the web page.

```
            function sendStuff() {
                sendJSON([42, 'PySide', 1.23, true, {'a':1,'b':2}]);
            }
```

This is similar to the setRotation function ,and simply sends some arbitrary data to the Python side (where it gets printed and answered to with a reply message, which is received using receiveJSON above).

```
        </script>
    </head>
    <body style="background-color: white;">
        <h2>PySide, QML and WebKit on MeeGo</h2>
        <p>
        Set rotation:
        <input type="text" size="5" id="rotate" value="10"/>
        <button onclick="setRotation();">Click me now!</button>
        </p>
        <p>
        Send arbitrary data structures:
        <button onclick="sendStuff();">No, click me!</button>
        </p>
        <p>Received stuff:</p>
        <pre id="received"></pre>
    </body>
</html>
```

And finally, here is the HTML content of the web page (the part that gets displayed on the screen – we have a heading, two paragraphs and some form elements like a text entry box and buttons, which carry out the requested actions when clicked. These fields make use of the JavaScript functions defined above.

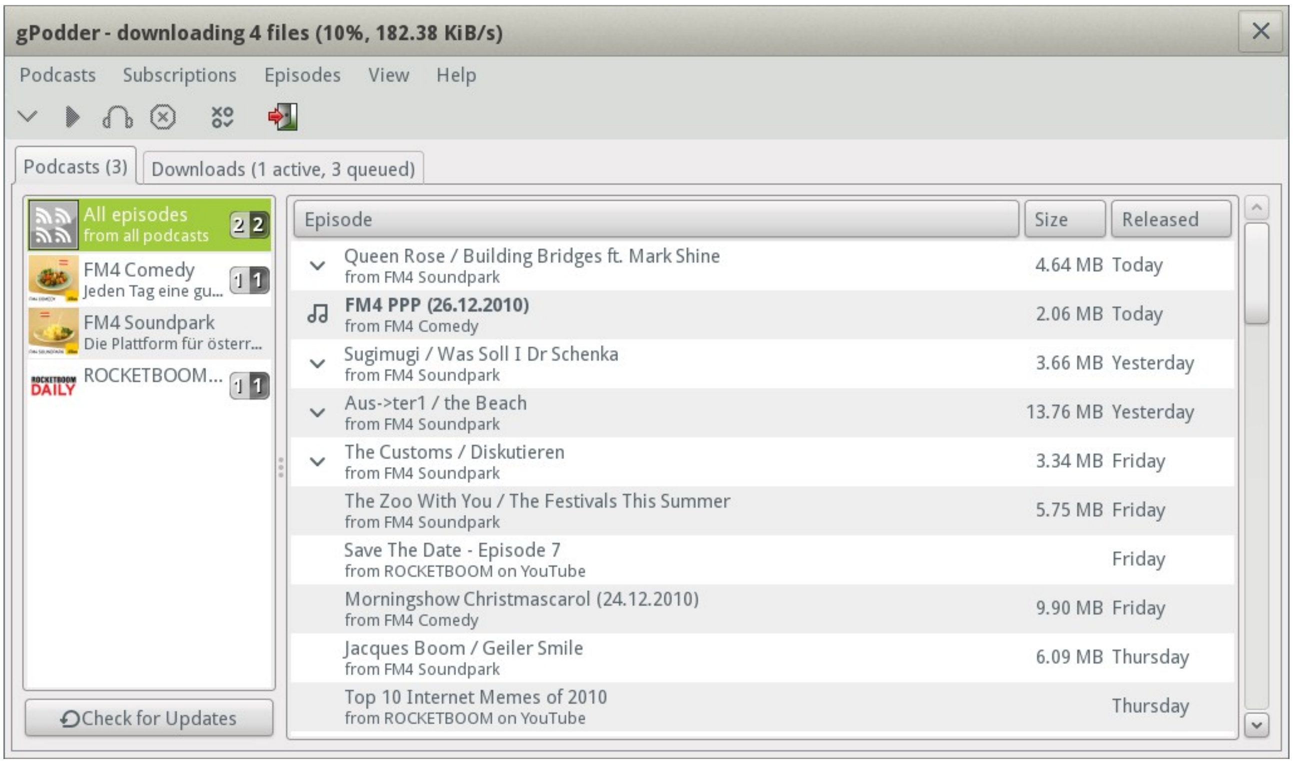
## Writing a new QML UI for existing apps for MeeGo

This section deals with a real-world application example and how to use our knowledge gained in the previous section to create a great mobile UX for your application to be used on MeeGo netbook and handset. As the dependencies are in MeeGo Core, the same application obviously works on all other MeeGo devices (IVI, TV, …) as well, but you might have to tailor your QML UIs to be usable on these devices, as the usage situation is different. Again – you have to create a special UI, but the technologies to create the UI are the same.

### A QML UI for gPodder

gPodder is a podcast client with which you can download video and audio content from the web to your device to play it on the go and manage subscriptions to radio shows. The current gPodder UI is written using PyGTK, which is still available on MeeGo Netbook, but the preferred UI toolkit is Qt with QML, and MeeGo Handset does not support PyGTK very well. Our goal here is not to show how to do a complete port of gPodder to Qt/QML, but to show how to start and how to integrate existing code with QML UIs through the use of code examples.

The normal gPodder Desktop UI does not use QML yet, and it's more tailored towards Desktop use and is not very easy to use with touchscreen devices. We now want to create a touch-friendly podcast and episode list UI. The old UI looks like this on MeeGo Netbook:


gPodder - downloading 4 files (10%, 182.38 KiB/s)

Podcasts (3) | Downloads (1 active, 3 queued)

| Podcast | New |
|---|---|
| All episodes<br>from all podcasts | 2 2 |
| FM4 Comedy<br>Jeden Tag eine gu... | 1 1 |
| FM4 Soundpark<br>Die Plattform für österr... | |
| ROCKETBOOM... | 1 1 |

Check for Updates

| Episode | Size | Released |
|---|---|---|
| Queen Rose / Building Bridges ft. Mark Shine<br>from FM4 Soundpark | 4.64 MB | Today |
| **FM4 PPP (26.12.2010)**<br>from FM4 Comedy | 2.06 MB | Today |
| Sugimugi / Was Soll I Dr Schenka<br>from FM4 Soundpark | 3.66 MB | Yesterday |
| Aus->ter1 / the Beach<br>from FM4 Soundpark | 13.76 MB | Yesterday |
| The Customs / Diskutieren<br>from FM4 Soundpark | 3.34 MB | Friday |
| The Zoo With You / The Festivals This Summer<br>from FM4 Soundpark | 5.75 MB | Friday |
| Save The Date - Episode 7<br>from ROCKETBOOM on YouTube | | Friday |
| Morningshow Christmascarol (24.12.2010)<br>from FM4 Comedy | 9.90 MB | Friday |
| Jacques Boom / Geiler Smile<br>from FM4 Soundpark | 6.09 MB | Thursday |
| Top 10 Internet Memes of 2010<br>from ROCKETBOOM on YouTube | | Thursday |

**The glue layer in Python (gpodder-qml.py)**

This file uses the existing gPodder codebase (which thankfully exports an easy-to-use API to our data structures via the “gpodder.api” module) and implements the classes and structures necessary to expose the gPodder data to our QML UI and also enables us to interact with the data by using a Controller that is also exposed to the QML UI and can be accessed directly from QML.

```
# -*- coding: utf-8 -*-

import os
import sys

from PySide import QtCore
from PySide import QtGui
from PySide import QtDeclarative
from PySide import QtOpenGL

from gpodder import api
```

These statements import the required modules. If you don't want or need OpenGL-accelerated QML, you can skip the import of the QtOpenGL module. This example also assumes that you have gPodder already installed system-wide. It is also advisable to use the legacy gPodder application to subscribe to some podcasts so that you can see some contents in the QML UI.

```
class EpisodeWrapper(QtCore.QObject):
    def __init__(self, episode):
        QtCore.QObject.__init__(self)
        self._episode = episode

    def _title(self):
        return self._episode.title

    def _description(self):
        return unicode(self._episode.one_line_description())

    def _downloaded(self):
        return self._episode.was_downloaded(and_exists=True)

    @QtCore.Signal
    def changed(self): pass

    title = QtCore.Property(unicode, _title, notify=changed)
    description = QtCore.Property(unicode, _description, notify=changed)
    downloaded = QtCore.Property(bool, _downloaded, notify=changed)
```

The “EpisodeWrapper” class is a subclass of QObject (because we need to access it from QML) and exposes some information about the episode (i.e. its title and description and whether or not it has already been downloaded) to the QML UI. It's important here that you make the properties notifyable (by defining a Signal “changed” and specifying it as notification signal for the properties by using “notify=changed” when defining the properties), so that the QML UI can be notified when it has to update its fields (i.e. when the downloaded state of an episode changes, the QML UI should update itself to reflect that change).

```
class PodcastWrapper(QtCore.QObject):
    def __init__(self, podcast):
        QtCore.QObject.__init__(self)
        self._podcast = podcast

    def _url(self):
        return self._podcast.url

    def _title(self):
        return self._podcast.title

    def _description(self):
        return unicode(self._podcast._podcast.description)

    def _cover_file(self):
        f = self._podcast._podcast.cover_file
        if os.path.exists(f):
            return f
        else:
            return '/usr/share/gpodder/podcast-0.png'

    def _count(self):
        total, deleted, new, downloaded, unplayed = self._podcast._podcast.get_statistics()
        return downloaded

    @QtCore.Signal
    def changed(self): pass

    url = QtCore.Property(unicode, _url, notify=changed)
    title = QtCore.Property(unicode, _title, notify=changed)
    description = QtCore.Property(unicode, _description, notify=changed)
    cover_file = QtCore.Property(unicode, _cover_file, notify=changed)
    count = QtCore.Property(int, _count, notify=changed)
```

As we display not only episodes, but also podcasts (a podcast is a collection of several episodes), we have to wrap the gPodder-internal podcast objects as well, and do the same thing as for the episode objects. We do the same as for the EpisodeWrapper, but expose different properties. If one of the properties were to change, we would call “self.changed.emit()” from the PodcastWrapper instance to update its representation in the UI.

```
class PodcastListModel(QtCore.QAbstractListModel):
    COLUMNS = ('podcast',)

    def __init__(self):
        QtCore.QAbstractListModel.__init__(self)
        self._client = api.PodcastClient()
        self._podcasts = [PodcastWrapper(x) for x in self._client.get_podcasts()]
        self.setRoleNames(dict(enumerate(PodcastListModel.COLUMNS)))

    def rowCount(self, parent=QtCore.QModelIndex()):
        return len(self._podcasts)

    def data(self, index, role):
        if index.isValid() and role == PodcastListModel.COLUMNS.index('podcast'):
            return self._podcasts[index.row()]
        return None
```

QML can display lists of items and automatically provide an easy way to display lists using the ListView component. In order to display any data, it has to be put into a list model (which is a collection of rows that need to be displayed). We create such a list model for podcasts here. You could define multiple columns, but in order to make use of QObject properties, we only have one column which is a QObject. The podcast objects are directly loaded from the gPodder API.

The internal representation of the data is a normal Python list (“self._podcasts”), whereas the QML ListView has some expectations on how to get the data – namely the rowCount method (which returns the number of rows in the model) and the data method, which should return the data for a given row and column (in our case, we only have one column – the “podcasts” column that contains a PodcastWrapper for every episode).

```python
class EpisodeListModel(QtCore.QAbstractListModel):
    COLUMNS = ('episode',)

    def __init__(self, episodes):
        QtCore.QAbstractListModel.__init__(self)
        self._episodes = [EpisodeWrapper(x) for x in episodes]
        self.setRoleNames(dict(enumerate(EpisodeListModel.COLUMNS)))

    def rowCount(self, parent=QtCore.QModelIndex()):
        return len(self._episodes)

    def data(self, index, role):
        if index.isValid() and role == EpisodeListModel.COLUMNS.index('episode'):
            return self._episodes[index.row()]
        return None
```

Just as with the PodcastListModel, in order to display a list of episodes, we need an EpisodeListModel. The problem here is that the list of episodes is not static, as it depends on which podcast was selected in the UI (this will become clear in a bit when you see how the UI is structured). Therefore, we don't grab the list of episodes directly in the constructor, but receive them as a parameter to the constructor.

We then take all “native” gPodder episode objects, wrap them in our EpisodeWrapper (to be accessible from QML) and implement rowCount and data. This is basically the same as for the PodcastListModel, but it shows how you can have parameters in your model to provide the data.

```python
class Controller(QtCore.QObject):
    @QtCore.Slot(QtCore.QObject)
    def podcastSelected(self, wrapper):
        global view, episodeList
        view.rootObject().setProperty("state", "Episodes")

        episodeList = EpisodeListModel(wrapper._podcast._podcast.get_all_episodes())
        view.rootObject().setEpisodeModel(episodeList)

        print wrapper._podcast._podcast.__dict__

    @QtCore.Slot(QtCore.QObject)
    def episodeSelected(self, wrapper):
        global view
        view.rootObject().setProperty("state", "Podcasts")
```

The Controller is the “director” of our QML show – it exposes some helpful functions for our QML UI to use, and knows about the underlying Python objects, and makes sure that the view receives updated data when something is to be shown. In order to do so, we again need to subclass it from QObject (all objects that you want to access from QML have to be QObject subclasses, as QML does not know about Python objects).

Methods on that objects that need to be accessible (callable) from QML need to be decorated with the “QtCore.Slot” decorator – the parameters of the decorator describe the count and data type of the parameters that the functions expect – in that case, it's a single parameter of type QObject. We expose two simple functions – podcastSelected and episodeSelected that will be called from QML when the user clicks on (or touches) an item in one of the list views.

When a podcast is selected, we set the state of the UI to “Episodes” (which automatically starts the transition in QML – which we will see later) and we populate the list of episodes from the podcast, and when an episode is selected, we simply go back to the podcasts list (for this example, this is enough – a full-fledged application might want to show a “episode details” view, play the episode or start the download).

```
app = QtGui.QApplication(sys.argv)

view = QtDeclarative.QDeclarativeView()
glw = QtOpenGL.QGLWidget()
view.setViewport(glw)
view.setResizeMode(QtDeclarative.QDeclarativeView.SizeRootObjectToView)
```

This code sets up a new Qt UI application and creates a window that can be used for displaying QML content (QDeclarativeView). The two lines with “glw” are optional and enable OpenGL rendering of QML content (which – depending on your device – might be faster than normal rendering). If your MeeGo device does not support this, or if the performance is worse, simply comment out these two lines.

The “setResizeMode” function on QDeclarativeView defines how the resizing of the window is handled – in our case, we want the root object in our QML to be automatically resized to fill the window. This is what you usually want if you don't want to hardcode the QML to a specific size.

```
controller = Controller()
podcastList = PodcastListModel()
episodeList = EpisodeListModel([])
```

These three lines create instances of our classes defined above – a Controller used as access point for QML, the podcast list (already populated with the user's podcast subscriptions) and the episode list (which is empty until the user clicks on a podcast).

```
rc = view.rootContext()

rc.setContextProperty('controller', controller)
rc.setContextProperty('podcastList', podcastList)
rc.setContextProperty('episodeList', episodeList)
```

The QML root context can have properties that can be accessed by name from QML code. We have to expose our three objects (the controller, the podcast list an the episode list) and give them property names so that we can access them directly from QML.

```
view.setSource(__file__.replace('.py', '.qml'))

view.show()

app.exec_()
```

Now that we have everything set up, we simply have to load our QML UI into the view (by using setSource on the view). The “__file__.replace('.py', '.qml')” means that the QML file has the same name as our Python script, but with the extension “.qml” instead of “.py”.

Now all we need to do is create the QML UI code for our little app.

**The QML main UI file (gpodder-qml.qml)**

Now that we have our backend code (written in Python) set up, we just need to create a nice QML UI on top of it. The “interface” to the backend is already defined by the context properties that we have defined – there's no other “route” to call Python code from QML. In our special case, that is the Controller object on which we can call methods and the two models, which will be used by the ListView components to display lists of podcasts and episodes.

In order to demonstrate good modularity of QML apps, we also split out the podcast list and episode list as separate components, so that the look and feel (and behaviour) of the lists can be changed without needing to edit the main UI file (it also makes the main UI file smaller and easier to understand).

```
import Qt 4.7
```

This imports all basic QML elements for use into this QML file. In Qt 4.7.1 and newer, you should use “import QtQuick 1.0” instead, but with Qt 4.7.0 (as is the case on MeeGo Netbook), you have to use “import Qt 4.7”.

```
Rectangle {
    id: rectangle1
    width: 400
    height: 400
    opacity: 1
    state: "Podcasts"
```

Here, we define our outermost “root” object – a simple Rectangle with the id “rectangle1” that has a width and height of 400 pixels (due to the setResizeMode call in our Python code, this object will get resized when the window size changes). The default state of this object is “Podcasts” (see below), which means that by default, the podcast list will be shown when the QML UI is first loaded.

```
    function setEpisodeModel(mod) {
        episodelist.model = mod
    }
```

This function is used to set a new model on the episode list view. It is a method of our root object and can therefore be called from Python – see the Controller class on how this function is used to load a list of episodes into the view.

```
    PodcastList {
        id: podcastlist
        model: podcastList
        contr: controller
    }
```

The PodcastList component isn't defined in Qt – it's a component we will define by ourself as “PodcastList.qml” in the same directory as this QML file. We give it an ID to be able to reference it from other parts of the file, and set properties “model” (the data model to use – in our case the list of podcasts) and “contr” (a custom property that we use to give a reference to the controller to the list view, so that we can access the controller directly from the list view).

```
    EpisodeList {
        id: episodelist
        model: episodeList
        contr: controller
    }
```

This is equivalent to the PodcastList component usage above – the component will be defined by us later in the file “EpisodeList.qml”, and will be accessible through the ID “episodelist” in this QML file (i.e. it's already referenced by setEpisodeModel above).

```
    states: [
    State {
        name: "Podcasts"

        PropertyChanges {
            target: podcastlist
            opacity: 1
            visible: true
        }

        PropertyChanges {
            target: episodelist
            scale: 0
            opacity: 0
            rotation: 180
        }
    },
```

Here, we define the “states” in which this QML component (our root object) can be in – in our case, there are two states: “Podcasts” (show a list of podcasts) and “Episodes” (show a list of episodes). When the component is in this state, the podcastlist object will be shown and the episode list will be hidden (by scaling it to a factor of zero, setting its opacity to zero and rotating it by 180 degrees – for a nice effect that we will define later by the use of transitions).

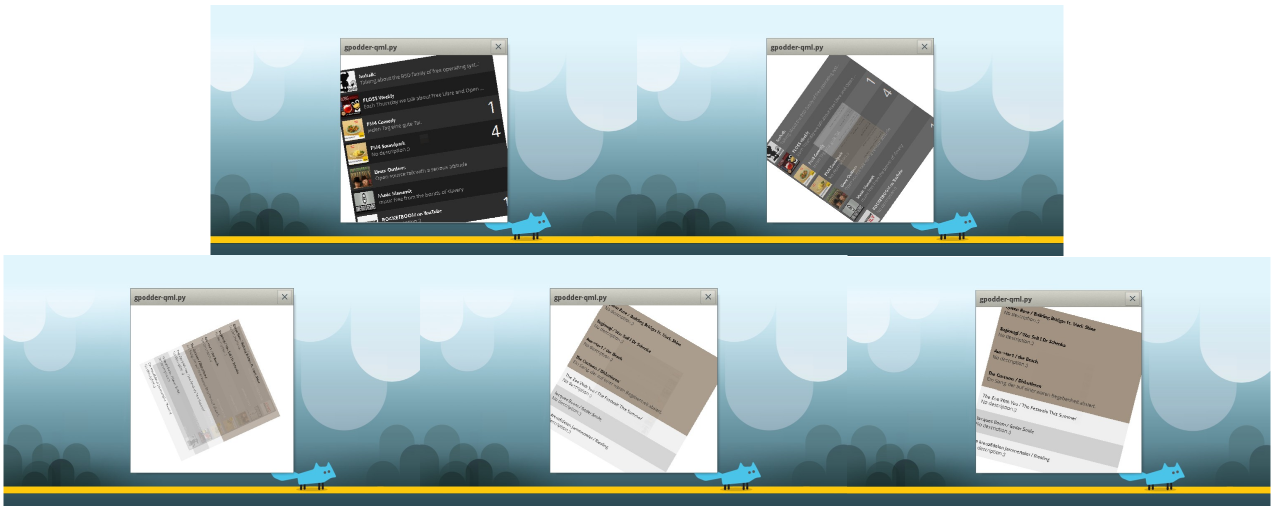
```
    State {
        name: "Episodes"

        PropertyChanges {
            target: podcastlist
            z: 0
            rotation: -180
            scale: 0.3
            visible: true
            opacity: 0
        }

        PropertyChanges {
            target: episodelist
            scale: 1
            opacity: 1
        }
    }
    ]
```

The other state that our root object can be in is “Episodes”. In this state, we hide the podcast list (and rotate and scale it and then set its opacity to zero to hide it) and instead show the episode list. When this state is entered, the target components will automatically get these properties assigned.

```
    transitions: [
        Transition {
            PropertyAnimation {
                properties: "scale,opacity,rotation"
                duration: 500
            }
        }
    ]
}
```

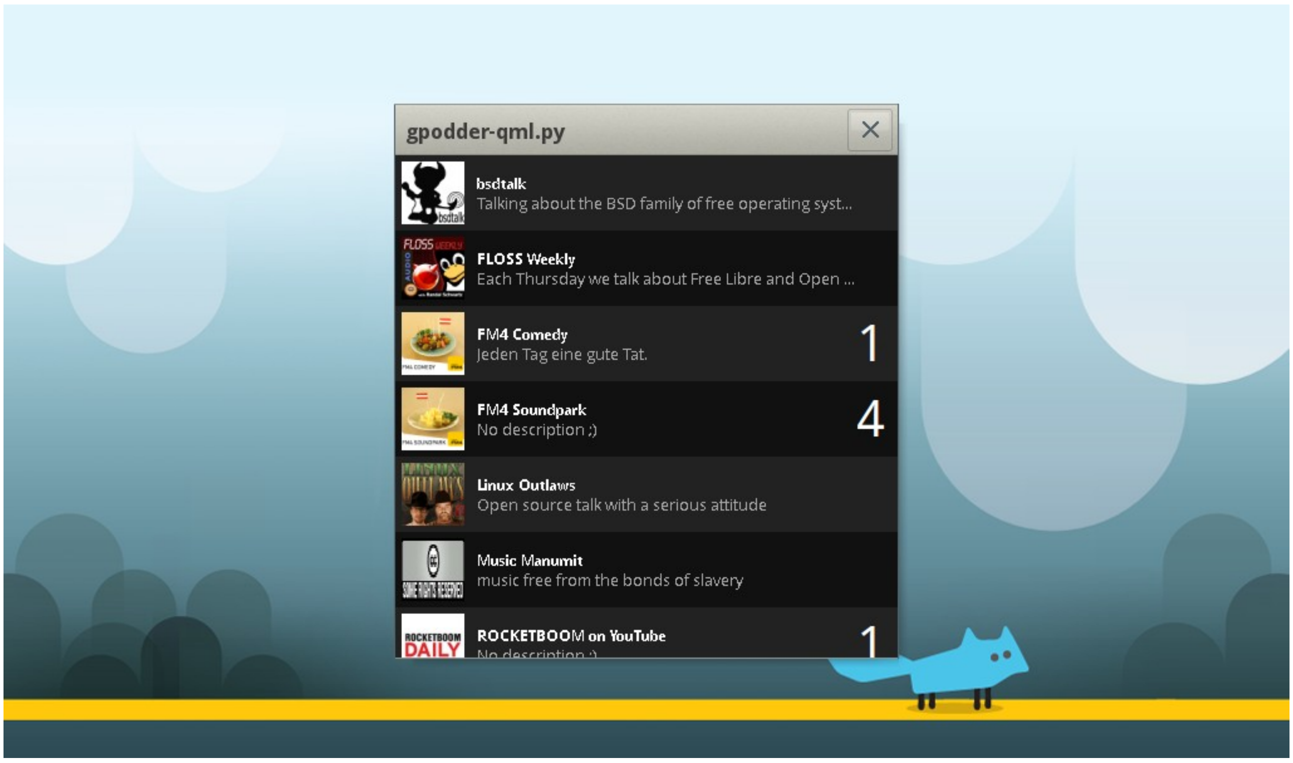
I mentioned transitions. Just hiding and showing elements is boring. Therefore, we simply define some transitions on our root object that will be used to animate its children when the root object state changes – in our case, we want to animate the “scale”, “opacity” and “rotation” properties, and the animation should take exactly 500 milliseconds. With this definition, changing the properties will not have an immediate effect, but the properties will be “animated” to reach the end value in 500 milliseconds from the time the property has been set.

Here's how the transition looks like:



This is everything we need for our little QML app – the specific appearance of each component is then defined in the PodcastList.qml and EpisodeList.qml files. The transitions and state changes between these objects are taken care of by the root view and our controller written in Python.

**The QML file for displaying a list of podcasts (PodcastList.qml)**



gpodder-qml.py

bsdtalk
Talking about the BSD family of free operating syst...

FLOSS Weekly
Each Thursday we talk about Free Libre and Open ...

FM4 Comedy
Jeden Tag eine gute Tat. 1

FM4 Soundpark
No description ;) 4

Linux Outlaws
Open source talk with a serious attitude

Music Manumit
music free from the bonds of slavery

ROCKETBOOM on YouTube 1
No description :)

What's left to do now is to specify the appearance of the podcast and episode list. Let's start with the podcast list, as this is th-e one that is shown first:

```
import Qt 4.7
```

We again need the default QML components shipped with Qt, so we import them here.

```
ListView {
    id: podcastListView

    property variant contr
```

The component is based on the QML ListView, but has a new property (of type “variant”, so we can place any arbitrary QObject in there) and has the name of “contr”. This is used by our QML application to give the Python Controller object to this listview. We also need to give this component a component-wide ID, so that we can access the controller using “podcastListView.contr” in other parts of this file.

```
    anchors.fill: parent
```

This component should automatically fill all the available space of the parent component – in practice, this means that the podcast list will always fill the whole visible area of our window – even when its size changes.

```
    delegate: Component {
        Rectangle {
            width: podcastListView.width
            height: 60
            color: ((index % 2 == 0)?"#222":"#111")
```

A delegate is used as “template” for rendering a single row in the list view.

Delegates get created and destroyed automatically as needed by QML. Our delegate is a simple Rectangle that has the same width as our view (because we want the list items to fill the whole width of the list). The background color of the list is defined by the "color" property of the Rectangle.

The "index" value inside a delegate gives us the (zero-based) row index of the current row that is to be rendered. We can utilize it to shade alternating rows with different background colors – when "index % 2 == 0" (the first, third, fifth, … row), the background color will be "#222" and when it is not (the second, fourth, sixth, … row), the background color will be "#333".

```
            Image {
                id: cover
                source: model.podcast.cover_file
                sourceSize {
                    width: height
                    height: height
                }
                width: 50
                height: 50
                anchors.left: parent.left
                anchors.top: parent.top
                anchors.leftMargin: (parent.height - width)/2
                anchors.topMargin: (parent.height - height)/2
            }
```

We want to display the cover art of a podcast – it should be 50x50 pixels in size, and its filename is taken from "model.podcast.cover_file". The "model" property accesses the underlying model (current row), and the "podcast" accesses the role name of "podcast" from the model – in our case, it is the first column (column index zero) of the model – a PodcastWrapper instance. The PodcastWrapper instance for the given row has a "cover_file" property that points to the file that should be displayed as cover art.

The rest of the definitions (anchors) is used for layouting, and is out of scope for this tutorial – you can read about this in the QML documentation.

```
            Text {
                id: title
                elide: Text.ElideRight
                text: model.podcast.title
                color: "white"
                font.bold: true
                anchors.top: parent.top
                anchors.left: cover.right
                anchors.right: count.left
                anchors.bottom: parent.verticalCenter
                anchors.leftMargin: 10
                verticalAlignment: Text.AlignBottom
            }
            Text {
                id: subtitle
                elide: Text.ElideRight
                color: "#aaa"
                text: model.podcast.description || "No description ;)"
                font.pointSize: 10
                anchors.top: title.bottom
                anchors.left: cover.right
                anchors.right: count.left
                anchors.leftMargin: 10
                verticalAlignment: Text.AlignTop
            }
```

Here we simply show two different lines of text – one being the title of the podcast, taken from "model.podcast.title", and the other one the description of the podcast, taken from "model.podcast.description". If the PodcastWrapper for a given line does not give a description, we use the “No description” text as default description.

```
            Text {
                id: count
                color: "white"
                font.pointSize: 30
                visible: model.podcast.count > 0
                text: model.podcast.count
                anchors.verticalCenter: parent.verticalCenter
                anchors.right: parent.right
                anchors.rightMargin: 10
            }
```

This is the amount of podcasts – the “count” property of PodcastWrapper. This component is only visible when the count of (downloaded) episodes is greater than zero. You can use as many properties as you want in a single expression, and can also use properties for different kinds of things – in this case we use it for both the visibility of the text and the contents of the text itself.

```
            MouseArea {
                anchors.fill: parent
                onClicked: { contr.podcastSelected(model.podcast) }
            }
```

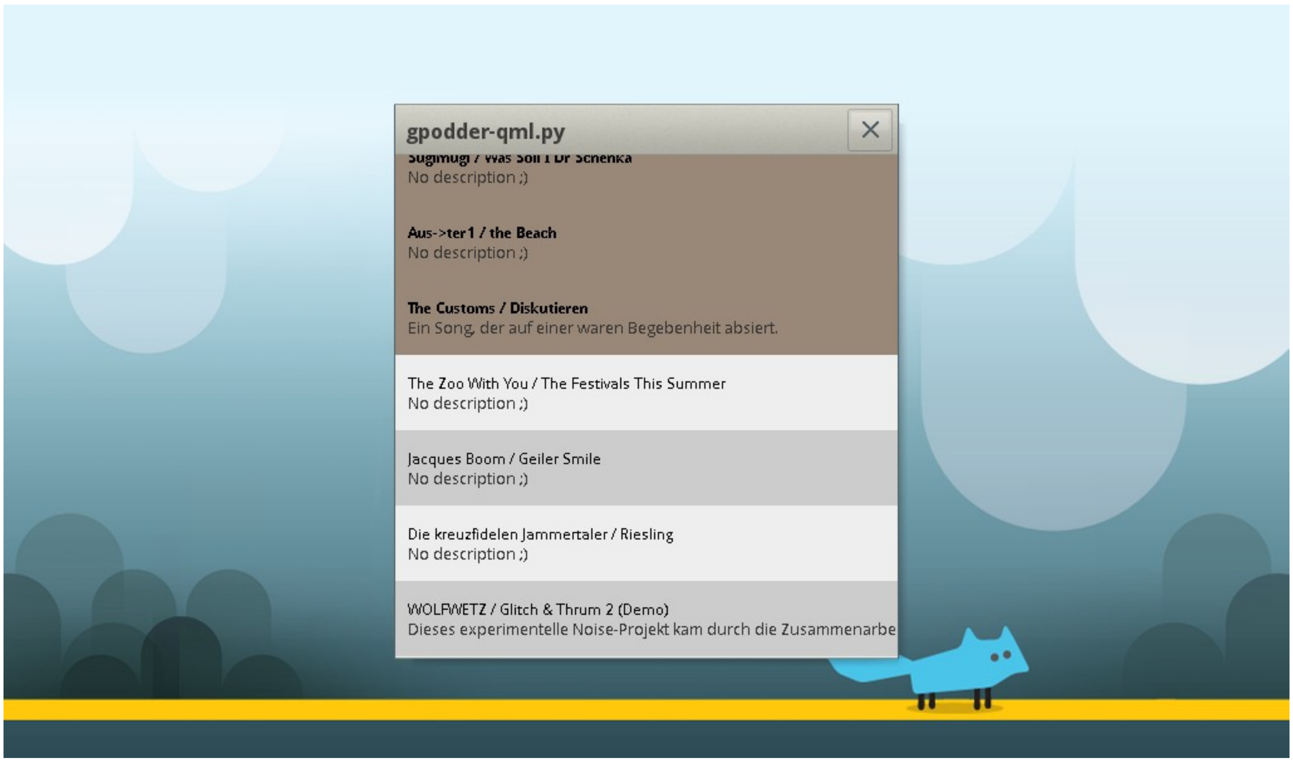
Up to now, we only have displaying of data. What we also want is to be able to click on a row and have some action performed. We can do this via a MouseArea – it should fill the parent component (otherwise it would not “catch” any clicks) and when it is clicked, the “podcastSelected” slot of our “contr” property should be called with “model.podcast” (a PodcastWrapper instance) as parameter. This is the “magic” in this case, as it will pass the object to the controller, which in turn will take care of switching the state of the main application and populating the list of episodes in the view.

```
        }
    }
}
```

Always make sure to close the brackets that you open in QML, or you will get a syntax error :)

## The QML file for displaying a list of episodes (EpisodeList.qml)

-


gpodder-qml.py

Sugimugi / Was Soll I Dr Schenka
No description ;)

Aus->ter1 / the Beach
No description ;)

The Customs / Diskutieren
Ein Song, der auf einer waren Begebenheit absiert.

The Zoo With You / The Festivals This Summer
No description ;)

Jacques Boom / Geiler Smile
No description ;)

Die kreuzfidelen Jammertaler / Riesling
No description ;)

WOLFWETZ / Glitch & Thrum 2 (Demo)
Dieses experimentelle Noise-Projekt kam durch die Zusammenarbe

The last part of our little project is the list of episodes. This is similar to the list of podcasts, so the listing of the code should be enough in this case:

```
import Qt 4.7

ListView {
    id: episodeListView

    property variant contr

    anchors.fill: parent

    delegate: Component {
        Rectangle {
            width: episodeListView.width
            height: 60
            color: (model.episode.downloaded?("#987"):((index % 2 == 0)?"#eee":"#ccc"))
            Text {
                id: title
                text: model.episode.title
                color: "black"
                font.bold: model.episode.downloaded
                anchors.top: parent.top
                anchors.left: parent.left
                anchors.right: parent.right
                anchors.bottom: parent.verticalCenter
                anchors.leftMargin: 10
                verticalAlignment: Text.AlignBottom
            }
            Text {
                id: subtitle
                color: "#333"
                text: model.episode.description || "No description ;)"
                font.pointSize: 10
                anchors.top: title.bottom
                anchors.left: parent.left
                anchors.right: parent.right
                anchors.leftMargin: 10
                verticalAlignment: Text.AlignTop
            }
            MouseArea {
                anchors.fill: parent
                onClicked: { contr.episodeSelected(model.episode) }
            }
        }
    }
}
```

We now have a fully-functional QML application written using Python. You can start the app using “python gpodder-qml.py”. What we need to do now is package it up for MeeGo to be installable as package.

## Packaging a Python application for MeeGo

This section explains how to create installable RPM packages from Python applications. We will continue to use our gpodder-qml example for this.

### Installing the required dependencies

In order to create RPM packages,  you need to install the following tools on your MeeGo Netbook (with “`sudo zypper install <packagename>`”):

- python-setuptools
- rpmdevtools
- rpm-build
- meego-rpm-config
- spectacle

### Creating the required metadata files

You now need to create some more files that are needed for packaging and for displaying your application in the MeeGo “Applications” page.

#### The desktop entry: gpodder-qml.desktop

This file describes the icon that will appear in the Applications menu:

```
[Desktop Entry]
Name=gPodder-QML
Exec=gpodder-qml.py
Icon=gpodder-qml
Terminal=false
Type=Application
Categories=AudioVideo;Audio;Network;FileTransfer;News;
```

#### The icon for the app menu: gpodder-qml.png

This file is the icon that will appear in the Applications menu. It has to have the same name (without the “.png” extension) as the “Icon=” key in gpodder-qm.desktop. You can download the icon from http://thp.io/2010/meego-python/

### The Spectacle YAML file: gpodder-qml.yaml

MeeGo uses YAML to describe the packaging information. Based on this YAML file, the RPM ".spec" will be generated. You can find out more about Spectacle on http://wiki.meego.com/Spectacle

```
Name: gpodder-qml
Summary: gPodder QML
Version: 0.1
Release: 1
Group: Network
License: BSD
URL: http://thp.io/2010/meego-python/
Sources:
    - "%{name}.tar.gz"
Description: A QML UI for gPodder

Builder: python
BuildArch: noarch
Files:
    - "%{_bindir}/%{name}.py"
    - "%{_datadir}/gpodder-qml/*.qml"
    - "%{_prefix}/lib/python2.6/site-packages/*.egg-info"
    - "%{_datadir}/applications/%{name}.desktop"
    - "%{_datadir}/icons/%{name}.png"
```

### The Python distutils file: setup.py

This file is used by the setup process to copy the required files to the corresponding locations in the filesystem when installed.

```
from distutils.core import setup

import glob

APP_NAME = 'gpodder-qml'

SCRIPTS = [APP_NAME+'.py']

DATA_FILES = [
    ('/usr/share/'+APP_NAME, glob.glob('*.qml')),
    ('/usr/share/applications', glob.glob('*.desktop')),
    ('/usr/share/icons', glob.glob('*.png')),
]

setup(name=APP_NAME,
      version='0.1',
      description='A QML UI for gPodder',
      author='Thomas Perl',
      author_email='m@thp.io',
      url='http://thp.io/2010/meego-python/',
      scripts=SCRIPTS,
      data_files=DATA_FILES)
```

## Changes in the gpodder-qml.py script

You need to do some changes to the gpodder-qml.py script so that it is executable as application. First, use "chmod +x gpodder-qml.py" to make it executable. After that, open it in an editor and add the text "#!/usr/bin/python" (without the quotes) as first line – this makes sure that Python is used to interpret this script when started (by default, the shell is assumed as interpreter).

## Creating the RPM sources

Create a .tar.gz archive of your source folder that contains all the files (the source folder should be named “gpodder-qml” and should contain all the files we've created above):

- `tar czvf ~/rpmbuild/SOURCES/gpodder-qml.tar.gz /path/to/gpodder-qml/`

You can now go ahead and create the directory structure used by rpmbuild:

- `mkdir -p ~/rpmbuild/SOURCES`
- `mkdir -p ~/rpmbuild/SPECS`

The “SOURCES” folder will host your YAML file and a source tarball of your application. The “SPECS” folder will host the generated .spec file. You can now create the spec file using spectacle's “specify” tool:

- `cd ~/rpmbuild/SOURCES`
- `specify gpodder-qml.yaml`
- `mv gpodder-qml.spec ../SPECS/`

You can ignore the warning about the missing “Makefile” - for our application, a Makefile is not needed, as the Python build uses setup.py only.

## Building the RPM package from source

Now that everything is set up, you can easily create the architecture-independent (“noarch”) RPM package with rpmbuild like this:

- `cd ~/rpmbuild/SPECS`
- `rpmbuild -ba gpodder-qml.spec`

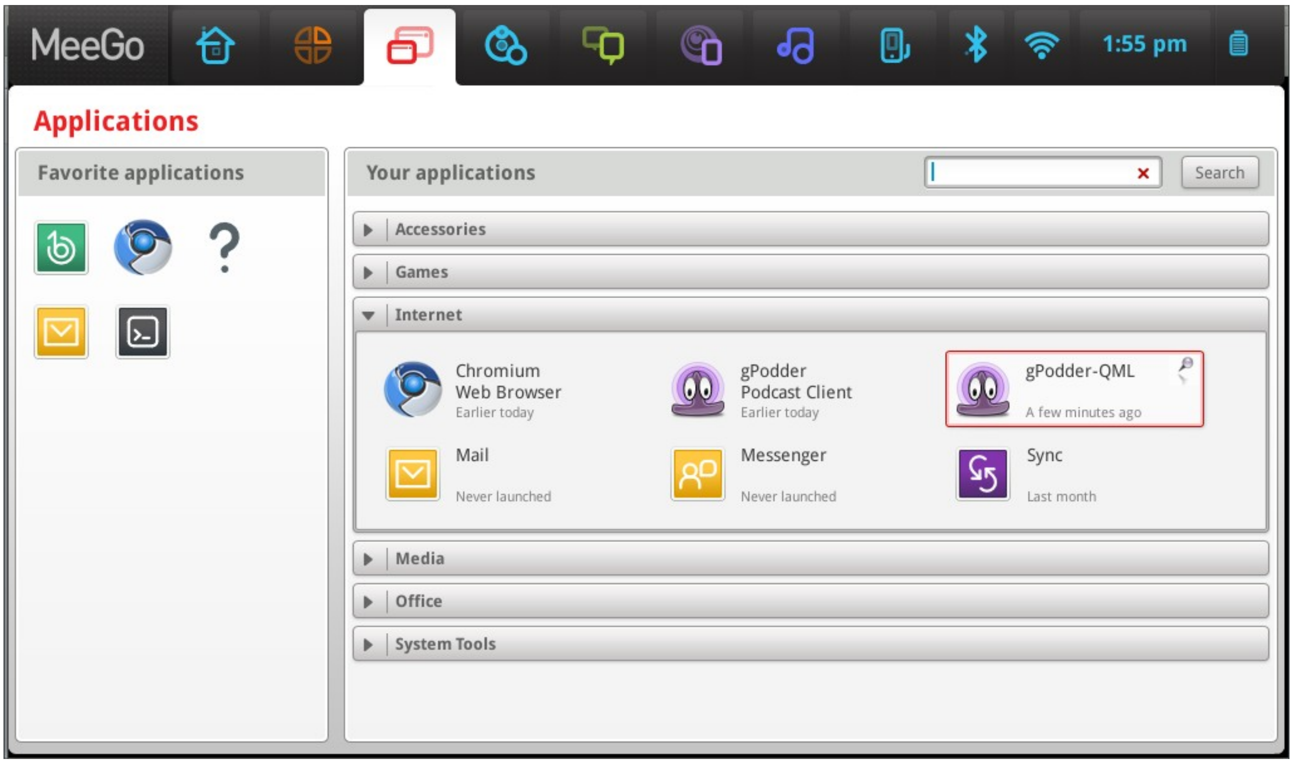
This will generate the package and save it in ~/rpmbuild/RPMS/noarch/. You should have a file named “gpodder-qml-0.1-1.noarch.rpm”.

## Installing and testing the RPM package

In order to be able to test the package, you can now install it using zypper:

- `cd ~/rpmbuild/RPMS/noarch`
- `sudo zypper install gpodder-qml-0.1-1.noarch.rpm`

This will install the package, and you should see it in your application menu:



Congratulations, you've successfully created your first MeeGo Python package!

That's it. Thank you for reading this tutorial. I hope to have given you some insights into developing QML applications for MeeGo in Python. Any feedback to this article is greatly appreciated! See http://thp.io/2010/meego-python/ for downloads of the code examples and more information as well as contact information for feedback.