Table of content

Updating QML content from Python threads 1
WorkingOnIt.py 1
Importing required modules 1
The Downloader object 1
Creating a new Downloader instance 2
QApplication, QDeclarativeView and context properties 2
WorkingOnIt.qml 2
How the example app looks like 4

Updating QML content from Python threads

This PySide tutorial shows you how to use native Python threads (non-QThread, i.e. threading.Thread) to carry out work in the background (e.g. downloading files). In the special case of downloading, you might want to use QNetworkAccessManager and friends, but in this case, we assume that you can't use it for whatever reason (e.g. because you want to use Twisted or because you already have your special download code that you want to reuse).

WorkingOnIt.py

Importing required modules

We will be using standard Python modules for threading (threading) and for downloading (urllib). From PySide, we need the standard module QtCore, QtGui and QtDeclarative:

```
import os
import sys
import threading
import urllib

from PySide import QtCore, QtGui, QtDeclarative
```

The Downloader object

We now subclass QObject (so we can have Signals, Slots and Properties in our downloader) and implement all the properties that we need for downloading the file and also for displaying the current status in the UI:

```
class Downloader(QtCore.QObject):
    def __init__(self, url, filename=None):
        QtCore.QObject.__init__(self)
        self._url = url
        if filename is None:
            filename = os.path.basename(self._url)
        self._filename = filename
        self._progress = 0.
        self._running = False
        self._size = -1

    def _download(self):
        def reporthook(pos, block, total):
            if self.size != total:
                self._size = total
                self.on_size.emit()
            self.progress = float(pos*block)/float(total)
        urllib.urlretrieve(self._url, self._filename, reporthook)
        self.running = False

    @QtCore.Slot()
    def start_download(self):
        if not self.running:
            self.running = True
            thread = threading.Thread(target=self._download)
            thread.start()

    def _get_progress(self):
```

```
        return self._progress

    def _set_progress(self, progress):
        self._progress = progress
        self.on_progress.emit()

    def _get_running(self):
        return self._running

    def _set_running(self, running):
        self._running = running
        self.on_running.emit()

    def _get_filename(self):
        return self._filename

    def _get_size(self):
        return self._size

    on_progress = QtCore.Signal()
    on_running = QtCore.Signal()
    on_filename = QtCore.Signal()
    on_size = QtCore.Signal()

    progress = QtCore.Property(float, _get_progress, _set_progress,
            notify=on_progress)
    running = QtCore.Property(bool, _get_running, _set_running,
            notify=on_running)
    filename = QtCore.Property(str, _get_filename, notify=on_filename)
    size = QtCore.Property(int, _get_size, notify=on_size)
```

Creating a new Downloader instance

As an example, we create a new Downloader here that downloads a kernel image for the N900 from the MeeGo repository:

```
downloader = Downloader('http://repo.meego.com/MeeGo/builds/trunk/1.1.80.8.20101130.1/
handset/images/meego-handset-armv7l-n900/meego-handset-armv7l-n900-1.1.80.8.20101130.1-
vmlinuz-2.6.35.3-13.6-n900')
```

QApplication, QDeclarativeView and context properties

As usual, we simply instantiate a new QApplication and a QDeclarativeView. Our Downloader is exposed to the QML context by setting it as context property downloader on the rootContext of our view. We then simply load the QML file via setSource, show the view and execute the application:

```
app = QtGui.QApplication(sys.argv)
view = QtDeclarative.QDeclarativeView()
view.rootContext().setContextProperty('downloader', downloader)
view.setSource(__file__.replace('.py', '.qml'))
view.show()
app.exec_()
```

WorkingOnIt.qml

This is the QML UI for our downloader example. The interesting parts here are:

* When the button is clicked, downloader.start_download() (a PySide Slot) is called, which starts the thread

* The UI elements use properties of the downloader to determine visiblity and content – they are automatically updated when the properties are notified to be updated

```
import Qt 4.7

Rectangle {
    width: 200; height: 160

    function formatProgress(size, progress) {
        return "" + parseInt(progress*size/1024) +
            " KiB ("+parseInt(progress*100.) + "%)";
    }

    Text {
        x: progressBar.x; y: 20
        width: progressBar.width
        font.pixelSize: 8
        text: downloader.filename
        elide: Text.ElideRight
    }

    Rectangle {
        id: progressBar
        color: "#aaa"

        x: 20; y: 60
        width: parent.width-40
        height: 20

        Rectangle {
            color: downloader.progress<1?"#ee8":"#8e8"
            clip: true

            anchors {
                top: parent.top
                bottom: parent.bottom
                left: parent.left
            }

            width: parent.width*downloader.progress

            Text {
                anchors {
                    fill: parent
                    rightMargin: 5
                }
                color: "black"
                text: formatProgress(downloader.size, downloader.progress)
                verticalAlignment: Text.AlignVCenter
                horizontalAlignment: Text.AlignRight
            }
        }
    }

    Rectangle {
        anchors.left: progressBar.left
        anchors.right: progressBar.right

        color: "#aad"
        y: progressBar.y + progressBar.height + 20
        height: 40

        Text {
```

```
            anchors.fill: parent
            color: "#003"
            text: downloader.running?"Please wait...":"Start download"

            verticalAlignment: Text.AlignVCenter
            horizontalAlignment: Text.AlignHCenter
        }

        MouseArea {
            anchors.fill: parent
            onClicked: { downloader.start_download() }
        }
    }
}
```

How the example app looks like

Save the files WorkingOnIt.py and WorkingOnIt.qml in the same folder and start the app using python WorkingOnIt.py: