Fundamental use cases for porting iPhone and Android applications to Qt

This article introduces some fundamental use cases that developers porting applications from iPhone and Android will need to implement, and discusses sample solutions. In general, Qt provides all modern classes and functionalities familiar from iPhone and Android. This makes it easy to keep the application logic close to the original when porting (see Snapshot of Qt classes).

Using Internet Services

Mobile applications are extremely likely to use the Internet — whether loading news stories from a server, using a remote service API such as Facebook or Twitter, or uploading photos to a photo-sharing service.

While protocols and service APIs vary considerably, it is very typical that the traffic happens on top of the HTTP protocol and is described as XML or JSON content.

The QNetworkAccessManager class makes it easy to interact with remote services. The class handles common configuration and settings for the requests it sends, such as settings for proxies and caching. The API is asynchronous and uses signals to notify clients about events such as finished requests. QNetworkAccessManager supports HTTP(S) and FTP.

A simple example of retrieving data using an URL asynchronously: (*replyFinished()* slot is called upon completion)

```
    ...
    QUrl url("http://www.example.com/example.xml");
    QNetworkAccessManager *manager = new QNetworkAccessManager(this);
    connect(manager, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(replyFinished(QNetworkReply*)));
    manager->get(QNetworkRequest(url);
    ...

myClass::replyFinished(QNetworkReply *reply) {
    QByteArray replyData(reply->readAll());
    // Process the reply data
}
```

See more info at http://doc.trolltech.com/main-snapshot/qnetworkaccessmanager.html

Web View

Qt has a WebView[1] QML element for displaying web content inside the application, like iPhone and Android. Thus, your existing server-side web content is directly reusable in Qt platforms.

Here is a short example of how to use a WebView element. A website can be loaded into the element by setting its url[2] property. Note that the QtWebKit import is also required.

```
import QtQuick 1.0
import QtWebKit 1.0

WebView {
    height: 640
    width: 480
    url: "http://www.developer.nokia.com/"
}
```

To get a scrolling webview, put the WebView element inside a Flickable[3] element:

```
import QtQuick 1.0
import QtWebKit 1.0

Flickable {
    id: flick
    width: 640
    height: 480
    contentWidth: web.width
    contentHeight: web.height
    WebView {
        id: web
        anchors.top: parent.top
        anchors.left: parent.left
        preferredWidth: flick.width
        preferredHeight: flick.height
        url: "http://www.developer.nokia.com"
    }
}
```

There are a few notable things about the above example: first, the WebView gets its preferredWidth and preferredHeight properties set to the size of the Flickable, which helps the WebView to try to resize the page in an intelligent way. Second, the Flickable needs to know the size of its contents to enable flicking: this is done using the contentWidth and contentHeight properties. Third, as the actual size of the WebView changes according to the loaded page, the WebView is anchored in the Flickable only by its top and left edges.

OpenGL ES Graphics using Qt GameEnabler

(C) Copyright Nokia 2013. All rights reserved.

It is easy to integrate OpenGL ES graphics into Qt applications using Qt's OpenGL module. Qt GameEnabler takes the integration one step further by providing a simple starting point for game development, including not only graphics, but also audio playback and input event handling. By downloading a Qt GameEnabler release you get a project template that you can then start customizing and extending.

There are many projects using Qt GameEnabler hosted on the developer site that can be used for reference. One of them, Match'em Poker, also has a porting diary noting the steps taken to port it from iOS to Qt.

For a step-by-step introduction to Qt Game Enabler, see this guide.

Geographic Location and Orientation

Qt gives developers easy access to device orientation and location data through the Location and Sensors APIs. A good example of sensor usage is the Compass example app in projects. It uses orientation and compass sensors together with GPS positioning to implement a virtual compass and a map view.

To get a compass reading, the app just includes the sensors API in its project file:

```
CONFIG += mobility
MOBILITY += sensors ...
```

Then in "mainwindow.cpp", the app creates a QCompass instance, adds a filter to it and registers to receive compass reading updates:

```
m_compass = new QCompass(this);
m_compassFilter = new CompassFilter(this);
m_compass->addFilter(m_compassFilter);
...
bool compassStarted = m_compass->start();
...
connect(m_compassFilter,
        SIGNAL(azimuthChanged(const QVariant&, const QVariant&)),
        rootObject, SLOT(handleAzimuth(const QVariant&, const QVariant&)));
```

In this case the azimuth changes are delivered straight to the root object of the QML UI, but of course the final signal handler can be something else entirely.

Media Usage

Many application ideas and concepts can be realised thanks to the ease and convenience of the audio/video capabilities of Qt.

Qt provides a quick and simple way to display images and simple animations using the Image and AnimatedImage class:

```
AnimatedImage {
    source: "animations/fire.gif"
}
```

Also, the common task of playing an audio file can be easily accomplished:

```
QSound::play("mysounds/bells.wav");
```

For more complex cases, Qt provides the [http://doc.trolltech.com/4.7-snapshot/phonon.html Phonon multimedia framework]. Phonon provides functionality for playback of the most common multimedia formats. The media can be read from files or streamed over a network, using a QURL to a file.

Music file playback can be constructed as follows:

```
Phonon::MediaObject *music =
Phonon::createPlayer(Phonon::MusicCategory,
Phonon::MediaSource("/path/song.mp3"));
music->play();
```

For video playback, the QtMultimediaKit module[4] provides the Video QML element[5].

```
import QtQuick 1.0
import QtMultimediaKit 1.1

Rectangle {
    width: 360
    height: 360
    Video {
        source: "movie.avi"
        anchors.fill: parent
    }
}
```

Application Data Storage

SQLite has changed the way you store and read data easily in mobile applications, and there's no reason to change it. Qt allows the application to use either the device's own SQLite database or include one inside Qt project.

In addition to features that are familiar from iPhone and Android, Qt also provides an offline storage API. With the offline storage API, you can use SQLite directly from QML code with an intuitive JavaScript interface.

 (C) Copyright Nokia 2013. All rights reserved.

Read more

Qt provides several different methods for storing data.

Simple data, such as application settings, is easily handled with QSettings [6]. All details are handled automatically:

```
QSettings settings("CompanyName", "ApplicationName");
settings.setValue("store/price", 13);
...
int price = settings.value("store/price").toInt();
```

More complex data can be serialized into a binary form using QDataStream [7]. Serialization of C++'s basic data types, such as char, short, int and char*, is automatic. Serialization of more complex data is accomplished by breaking up the data into these primitive units. An XML format for data storage is available with QtXml [8], which provides a stream reader and writer for XML documents, and C++ implementations of SAX and DOM.

The most powerful data storage method is QtSql [9], which provides full SQL database functionalities. The following snippet creates a to-do list into a local database, inserts some data there, and reads it back selectively.

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName("myDataBase");

QSqlQuery query;
query.exec("CREATE TABLE items (id int primary key, item TEXT, description TEXT)");
query.exec("INSERT INTO items values(101, 'Shopping list', 'Milk')");
query.exec("INSERT INTO items values(102, 'Shopping list', 'Beer')");

query.exec("SELECT * FROM items WHERE description='Beer'");

while (query.next()) {
    QString list = query.value(1).toString();
    QString item = query.value(2).toString();
    qDebug() << list;
    qDebug() << item;
}
```

A temporary memory based database can be used by setting ':memory:' as the database name.

 (C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.