Qt Features for Hybrid Web/Native Application Development

Today's computer users live and work in an interconnected world. They always have a network at hand and expect their data to be available at all times wherever they are. The boundary between desktop applications and the web is closing and hybrid applications, mixing the best of both worlds, are starting to emerge.

With the Qt WebKit Integration and the QtNetwork module, you have all the tools that you need to create your own hybrid applications by mixing JavaScript, style sheets, web content and Qt components freely. Combined with the QtXmlPatterns module, you can download, transform and query XML and present it in a hybrid web environment. This paper presents the features provided in the Qt WebKit Integration, and discusses how you can use them to develop hybrid applications.

NOKIA

Qt Features for Hybrid Web/Native Application Development

Today's computer users live and work in an interconnected world. They always have a network at hand and expect their data to be available at all times wherever they are. The boundary between desktop applications and the web is closing and hybrid applications, mixing the best of both worlds, are starting to emerge.

With the Qt WebKit Integration and the QtNetwork module, you have all the tools that you need to create your own hybrid applications by mixing JavaScript, style sheets, web content and Qt components freely. Combined with the QtXmlPatterns module, you can download, transform and query XML and present it in a hybrid web environment.

Features of the Qt WebKit Integration

One of the key components when mixing web and native applications is the web contents rendering engine – QtWebKit. This module makes it easy to integrate the wide-spread WebKit engine into your applications. The integration includes triggering JavaScript from C++, as well as integrating C++ objects into a web page and interacting with those objects through JavaScript.

The WebKit engine can be seen as a fully fledged web browser engine. The highlights of this engine are its modern rendering features:

- ACID3 compliance gives you proper standards-compliant rendering.
- CSS-based transformations makes it possible to scale, rotate, skew and translate page elements through CSS.
- CSS-based animations let you make smooth transitions between different states. For instance, elements can be made to fade in and out as the mouse cursor hovers over them.
- Support for the video-tag allows for embedding video contents easily. The video player uses codecs available from the Phonon Multimedia framework (also a part of Qt).
- Full page zooming makes it possible to scale not only the font size, but the entire page including images.
- NPAPI support to embed standard web browser plugins enabling third party media and more.
- The SquirrelFish JavaScript engine offers one of the fastest JavaScript experiences available.

A modern, capable HTML rendering engine is only one half of a hybrid application. The other half is the interaction between the native application and the rendered contents. The Qt WebKit integration offers all the pieces of such a puzzle.

- Embed Qt widgets into a web page using the object-tag. This lets your web page contain Qt widgets along with native C++ code as part of the visual appearance of the page.
- Access Qt objects from the JavaScript. This allows you to insert C++ objects into the JavaScript context, letting the web page's script interact directly with your data structures.

- Trigger JavaScript from Qt. You can call JavaScript functions in their web page context from your C++ code and trigger events in the web page.
- Shared client side storage. This gives you access to a database accessible both from JavaScript and C++, making it possible to share large amounts of data easily even when offline.

The Qt WebKit integration forms the base of all hybrid applications, but the key factor is getting the native code and web contents to interact. Qt offers a number of techniques to interconnect the two worlds, sharing events and data while sharing the screen as one homogeneous interface to the user.

Interacting With Embedded Qt Objects

There are two ways to embed Qt objects into a web page illustrated using the `QWebView` widget. You can either add your objects to the JavaScript context of a page, or you can create a plugin that makes it possible to place Qt widgets inside a web page using the object tag.

The latter is an easy way to start creating a hybrid application: simply put your widgets inside a web page. When the widgets are in the page, their public slots will be exposed to JavaScript functions of the page as ordinary functions.

To add a widget to a page, you need to tell your `QWebPage` object that the widget is available. This is done by subclassing the `QWebPluginFactory` class and reimplementing the methods plugins and create. The plugins method informs the web page of which plugins it makes available, while the create method creates widgets on demand.

From the HTML making up the web page in question, the widgets are created using the object tag. For instance, the following tag will attempt to create an instance of an *application/x-qt-colorlabel* widget.

```
<object type="application/x-qt-colorlabel" width="50px" height="20px" id="label"
/>
```

To facilitate the creation of such a widget, we must enable plugins and tell `QWebPage` about the plugin factory class. In the code below, the `ColorLabelFactory` creates instances of `ColorLabel` widgets in in response to *application/x-qt-colorlabel* requests.

```
{
      ...
      QWebSettings::globalSettings()->
         setAttribute( QWebSettings::PluginsEnabled, true );
      webView->page()->setPluginFactory( new ColorLabelFactory( this ) );
      ...
}
```

The `ColorLabel` widget exposes a public slot called `changeColor()`. This is automatically made available to JavaScript in the web page. Adding a link to the HTML referring to the element makes it possible to activate C++ functions in a simple way.

```
<a href='javascript:document.getElementById("label").changeColor();'>Change color!</a>
```

To make it possible to push events in the other direction, you need to make your objects available through the JavaScript document contexts. Using the `addToJavaScriptWindowObject` method available from each `QWebFrame` of your `QWebPage`. This method allows you to add an object to the JavaScript context with a name:

```
webView->page()->mainFrame()->
   addToJavaScriptWindowObject( "eventSource", new eventSource( this ) );
```

To connect a signal from the object you have just added, you need to inject a piece of JavaScript into the web page context. This is done using the evaluateJavaScript method. The code below connects the signal `signalName` from the `eventSource` object to the JavaScript function `destFunction`. You can even pass arguments from the signal to the connected JavaScript function.

```
webView->page()->mainFrame()->
   evaluateJavaScript( "eventSource.signalName.connect(destFunction);" );
```

If you want to add a Qt object to the JavaScript contents of pages viewed through a standard browser, there is a signal to be aware of. Each time the JavaScript context is cleared, the frame emits the `javaScriptWindowObjectCleared` signal. To ensure that your Qt object is available at all times, you need to connect to this signal and make your `addToJavaScriptWindowObject` calls there.

The Qt WebKit integration lets you create applications where the native C++ interacts with the JavaScript context in a seamless manner. This is the foundation of powerful hybrid applications, and Qt makes it easy.

Using the Client Side Storage to Share Data

With HTML 5, the web standard moves closer to the desktop in the same way as the desktop has started to integrate the web. One of the bigger changes in this direction is the introduction of client side storage. This gives each origin (i.e. each web page context) a SQL capable database engine on the client machine. This can be used to cache data locally, to reduce traffic and make the page available off-line. It can also be used to store large quantities of data in a structured, searchable way.

The client side storage is available from JavaScript. The idea is to search the database from your JavaScript code and generate the HTML from the search result. This is done using the `openDatabase` and `transaction` functions. Assuming the database is present and filled, the following snippet shows the idea.

```
db = openDatabase("TestDb ", "1.0", "Client side storage test", 200000);
db.transaction(function(tx) {
  tx.executeSql("SELECT id, text FROM Texts",
    [], function(tx, result) {
      for (var i = 0; i < result.rows.length; ++i) {
        var row = result.rows.item(i);
        processText( row['id'], row['text'] );
      }
    }, function(tx, error) {
      alert('Failed to retrieve texts from the database - ' + error.message);
      return;
    });
  });
```

Using the Qt WebKit integration, you can access the same databases through Qt's SQL module. This can be a very useful feature in a hybrid application. For example, the web page half of you application can use exactly the same mechanism for keeping data as when sharing data with your native application half.

To avoid security breaches, the client side databases are only accessible through JavaScript from the right security origin. From your native C++ code you can access all security origins through the static `QWebSecurityOrigin::allOrigins` method, or via the `QWebFrame::securityOrigin` method.

Given an origin object, you can get access to a list of `QWebDatabase` objects through the `databases` method. Each web database object has a `fileName` property that can be used to open the database for access from the native code.

```
QWebDatabase webdb = mySecurityOrigin.databases()[index];
QSqlDatabase sqldb = QSqlDatabase::addDatabase("QSQLITE", "webconnection");
sqldb.setDatabaseName(webdb.fileName());
if (sqldb.open()) {
  QStringList tables = sqldb.tables();
  ...
}
```

Combining this data sharing mechanism with the ability to connect events between the web and your native application, makes it easy to blur the border between web and desktop.

Transforming the Web

Much of the data available through the web is not suitable for direct presentation. Examples are news feeds, geo data and other application specific data formats. Qt's networking module makes it possible to download such data in an easy manner. Parsing the data and converting it into a suitable format can then be handled by your custom code or, for instance, by the QtXmlPatterns module. The latter is handy when the output format is expected to be XML, or if you want to display it in a web page, XHTML.

In this section we will walk through the interesting parts of a small example downloading a news feed, converting it from XML to XHTML using an XSL transform, before presenting it through a QWebView as shown in figure 1.

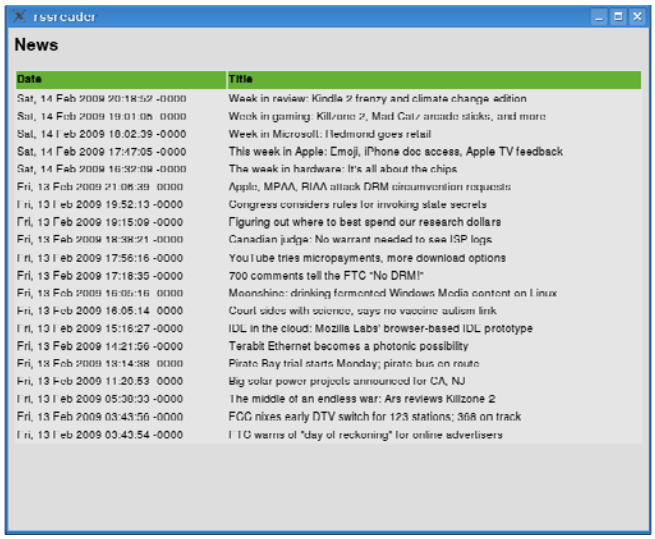

Figure 1: A news feed from ArsTechnica shown as XHTML using a QWebView.

The QNetworkAccessManager class allows you to easily handle the interaction between a client and a web server. It helps you handle details such as proxy and cache settings, but also cookies and SSL/TSL sessions. All in all, it makes it easy for the average case – to upload or download data -- but allows you to handle complex sessions with logins, certificates, etc..

To download the news feed for our example, all we need to do is create a `QNetworkAccessManager` and call its `get` method. The result is returned through the `finished(QNetworkReply*)` signal.

However, as you can tell from the source code below, the reply object is available from the moment you make the request. This allows us to monitor the progress of the request by listening to the `downloadProgress(qint64,qint64)` signal.

```
{
      ...

      QNetworkAccessManager *manager = new QNetworkAccessManager( this );

      connect( manager, SIGNAL(finished(QNetworkReply*)),
               this,    SLOT(handleReply(QNetworkReply*)) );
      connect( manager, SIGNAL(finished(QNetworkReply*)),
               m_progressBar, SLOT(hide()) );

      QNetworkReply *reply = manager->get( QNetworkRequest( QUrl( feedUrl ) ) );

      connect( reply, SIGNAL(downloadProgress(qint64, qint64)),
               this,  SLOT(updateProgress(qint64,qint64)) );
}
```

When the request is finished, the `QNetworkReply` object holds the entire reply. This reply needs further processing before it can be presented to users. In the case of an XML news feed such as RSS or ATOM, the processing can be done using an XSL transformation. In the case of this transformation, we use convert each item-node into a XHTML table row, populating the columns with the contents of the tags pubDate and title. This XSL template is part of a larger file containing more rules and the rest of the XHTML tags surrounding the table data.

```
<xsl:template match="item">
<tr>
      <td><xsl:value-of select="pubDate"/></td>
      <td><xsl:value-of select="title"/></td>
</tr>
</xsl:template>
```

The XSL query is combined with the reply given from the `QNetworkReply` object using a `QXmlQuery` object. This objects lets us perform XQueries on our XML data, transforming it from the news feed into XHTML. In the code below, the `replyBuffer` contains the reply from the `QNetworkReply`. The `queryFile` is a `QFile` object with the XSL query and the `resultBuffer` is a `QIODevice` wrapping the `QByteArray result` into an interface suitable for `QXmlQuery`. The result is then displayed through the `QWebView m_webView` using the `setHtml` method.

```
{
      ...

      QXmlQuery qry( QXmlQuery::XSLT20 );
      qry.setFocus( &replyBuffer );
      qry.setQuery( &queryFile );
      qry.evaluateTo( &resultBuffer );

      m_webView->setHtml( QString(result) );
}
```

As you can see, Qt gives you all the tools that you need to easily download, process and display data from the Web. The combination of QNetworkAccessManager, QXmlQuery and QWebView forms a flexible flow from unformatted XML to data displayed on the screen. If your data is not XML, you can still use the QXmlQuery by providing a QAbstractXmlNodeModel, modeling non-XML data into XML nodes.

Summary

By providing the means for seamless integration between native C++ objects and JavaScript, Qt is the ideal platform for hybrid application development. The interaction between these two worlds is not limited to sharing data. Just like data, events can be passed both ways too.

Hybrid applications have a number of attractive features. For instance, the user knows the web metaphor and is used to working with it. As parts of the application is deployed over the web, it can be updated without having to install anything on each user's machine. Another benefit is the separation of style from content using style sheets. Creating hybrid applications using Qt takes this one step further – parts of your application are scripted, other parts consist of native code. The user interface can comprise of code, generated widgets, HTML or CSS.

The combination of the Qt WebKit integration, QtNetwork and QtXmlPatterns lets you harness the power of native code and the flexibility of HTML, CSS and JavaScript. These are the tools for building the next generation of applications.

For more information on the Qt WebKit Integration, visit http://qt.nokia.com./products or browse the Qt technical documentation at http://doc.trolltech.com/qtwebkit.html.

About Qt:

Qt is a cross-platform application framework. Using Qt, you can develop applications and user interfaces once, and deploy them across many desktop and embedded operating systems without rewriting the source code. Qt Development Frameworks, formerly Trolltech, was acquired by Nokia in June 2008. For more details about Qt please visit http://qt.nokia.com.

About Nokia

Nokia is the world leader in mobility, driving the transformation and growth of the converging Internet and communications industries. We make a wide range of mobile devices with services and software that enable people to experience music, navigation, video, television, imaging, games, business mobility and more. Developing and growing our offering of consumer Internet services, as well as our enterprise solutions and software, is a key area of focus. We also provide equipment, solutions and services for communications networks through Nokia Siemens Networks.

Code less.
Create more.
Deploy everywhere.

NOKIA

Oslo, Norway +47 21 60 48 00 ■ Redwood City, CA, USA +1 650-551-1676 ■ Beijing, China +86 10 6588 8836 ■ qt-info@nokia.com www.qt.nokia.com