Porting WRT widgets to Qt applications

Introduction

28 Mar 2010

This article provides Web developers with information about porting Web Runtime (WRT) widgets to Qt applications. The IDE used here is Qt Creator, but basic principles should apply to other IDEs, such as NetBeans or Eclipse, as well.

Creating a Qt Application with a QWebView Component

1. Start the Qt Creator IDE.
2. Select **File** > **New File or Project...** > **Projects** > **Qt4 Gui Application**.
3. Click OK.
4. Give the project a name and set its location.
5. Click Next.
6. In the following dialog, you can select the modules that you want to include in your project. Check the QtWebKit module and click Next.
7. In the following dialog, name your main window class as `WRTWidgetWindow` (the `Class name` field) and uncheck the `Generate form` box (in this project, the UI components are added by hand).
8. Click Next.
9. Review that the proposed changes are correct and click Finish. The project is opened and there are two source files, `main.cpp` and `wrtwidgetwindow.cpp`, which can be found in the **Sources** folder in the **Projects** view.

Now you are ready to write some code! Start by opening the `main.cpp` file and modify it to look like this:

```
#include <QtGui/QApplication>

#include "wrtwidgetwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    WRTWidgetWindow window;
    window.show();

    return app.exec();
}
```

Next, open the header file of the main window, `wrtwidgetwindow.h` (in the **Headers** folder in the **Projects** view), and modify it to look like this:

```
#ifndef WRTWIDGETWINDOW_H
#define WRTWIDGETWINDOW_H

#include <QtCore/QPointer>
#include <QtGui/QMainWindow>

class QWebView;

class WRTWidgetWindow : public QMainWindow
{
    Q_OBJECT

public:
    WRTWidgetWindow(QWidget *parent = 0);
    ~WRTWidgetWindow();

private:
    void setupUI();
    QWebView* createWebView();

private:
    QPointer<QWebView> webView;
};

#endif // WRTWIDGETWINDOW_H
```

The class, which was declared above, represents the main window of the application. It contains one Qt widget, QWebView (`webView`), which is used for showing the web content (HTML, CSS, JavaScript) on Qt. Next, open the implementation for the class (`wrtwidgetwindow.cpp`) and modify it so that it looks like this:

```
#include "wrtwidgetwindow.h"

#include <QtGui/QFrame>
#include <QtGui/QVBoxLayout>
#include <QtWebKit/QWebView>

WRTWidgetWindow::WRTWidgetWindow(QWidget *parent)
    : QMainWindow(parent)
{
    setupUI();
```

```
}

WRTWidgetWindow::~WRTWidgetWindow()
{
    webView->deleteLater();
}

void WRTWidgetWindow::setupUI()
{
    QFrame* cw = new QFrame(this);
    setCentralWidget(cw);

    QVBoxLayout* layout = new QVBoxLayout(cw);
    cw->setLayout(layout);

    webView = createWebView();
    layout->addWidget(webView);
}

QWebView* WRTWidgetWindow::createWebView()
{
    QWebView* view = new QWebView(this);
    return view;
}
```

After this, the application can be compiled and run. You can try this, even though the application doesn't do much yet; there's only an empty QWebView component on the screen.

Showing the web content from Qt

Let's add your existing WRT widget to the project (if you don't have one, grab one from here, for example: Web Runtime Stub). Copy the HTML, CSS, JavaScript, and resource files (such as graphic files) into your Qt project. It is most easily done by using the file manager and copying the respective folders into your project's root directory as a whole (you have the web content in separate folders, don't you?). The directory structure of your widget could look like this, for example:

```
html/   (HTML files)
style/  (CSS files)
script/ (JavaScript files)
gfx/    (graphics)
```

In order to use the web content from the Qt application, you must create a resource file in your project and add the content into it. To create the resource file, select **File** > **New File or Project...** > **Qt** > **Qt Resource file** and click OK. Give the resource file a name and set its location. Click Next and make sure that the resource file is added to your project (the **Add to Project** checkbox and the **Project** drop-down list). Click Finish. The resource file is created and will be automatically added as a resource into the `.pro` file of the project. Open the resource file in Qt Creator's plain text editor by right-clicking on it in the **Projects** view and selecting **Open With** > **Plain Text Editor**. Add the web content to the resource file as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<RCC version="1.0">
    <qresource>
        <!-- HTML files -->
        <file>html/index.html</file>

        <!-- CSS files -->
        <file>style/general.css</file>

        <!-- JavaScript files -->
        <file>script/common.js</file>
        <file>script/main.js</file>

        <!-- Graphics -->
        <file>gfx/icon.png</file>
    </qresource>
</RCC>
```

The resources are now compiled into the Qt application, but they are not used for anything yet. To access the resources from within the HTML file, you must prepend the relative paths to them with `"qrc:/"`. For example, if the main HTML file of the WRT widget contained these lines:

```
<link rel="stylesheet" href="style/general.css" type="text/css" />
<script type="text/javascript" src="script/common.js" charset="utf-8"></script>
<script type="text/javascript" src="script/main.js" charset="utf-8"></script>
```

then the main HTML file in the Qt application would contain these lines:

```
<link rel="stylesheet" href="qrc:/style/general.css" type="text/css" />
<script type="text/javascript" src="qrc:/script/common.js" charset="utf-8"></script>
<script type="text/javascript" src="qrc:/script/main.js" charset="utf-8"></script>
```

Change all the paths in the HTML files (this doesn't concern the CSS files, they work like before) as instructed above, and make this final change before you can try out your widget: Open the `wrtwidgetwindow.cpp` file again and load the main HTML file into the QWebView component with the following code:

```
QWebView* WRTWidgetWindow::createWebView()
{
    QWebView* view = new QWebView(this);
    view->load(QUrl("qrc:/html/index.html"));
    return view;
}
```

Voilà! Go ahead and clean and build your Qt application now (**Build** > **Rebuild Project**). Along with other files, the resources are compiled into the application. Then, run the application (**Build** > **Run** or `Ctrl+R`). The HTML file that you loaded above appears in the QWebKit component.

Porting Considerations

It was assumed above that the HTML file of the widget is relatively simple (such as the one in the WRT Stub application). More often than not, this is not the case. This chapter presents several issues to consider when porting WRT widgets to Qt applications.

Web Runtime API

In addition to the standard JavaScript objects, there are extension APIs in WRT that allow widgets to access system properties, for example. For a list of these objects, see Web Runtime API reference in Web Developer's Library. Because these objects are WRT-specific, they won't be understood by Qt. For example, the code that hides the softkeys looks like this:

```
window.menu.hideSoftkeys();
```

It must be removed from the Qt application in order to execute the JavaScript, and another way to accomplish the same result must be implemented, if needed. For example, to hide the softkeys you can show the application in fullscreen:

```
WRTWidgetWindow widget;
widget.showFullScreen();
```

S60 Platform Services

WRT provides several JavaScript service APIs for accessing S60 Platform Services. As the name suggests, these APIs are S60-specific (and WRT-specific), so they won't be understood by Qt. In a QtWebKit application, there will be two alternatives for Platform services. For C++ developers, Qt Mobility APIs are a good choice. They are being developed and will eventually offer the same kind of functionality. Also, the Qt hybrid application framework that Nokia is developing will provide that kind of functionality. The benefit of the latter one is that the underlying services can be accessed using pure web technologies.

Dynamic Scalability

Handling orientation changes dynamically is important when designing usable mobile applications. The technique presented in the snippets "Detecting orientation changes in Symbian Web Runtime" and "Reacting to the changes in screen size in Symbian Web Runtime" works fine in Qt, because the JavaScript event `onresize` can be used also from within the QWebKit component to take care of the scalability. The only thing that you may want to take into account is to set the resolution to a fixed value on platforms that don't support orientation changes (for example, desktop Windows and Linux). Here is the code that accomplishes this:

```
WRTWidgetWindow widget;
#if defined Q_OS_SYMBIAN || defined Q_WS_HILDON
    // On Symbian and Maemo, show the widget in full screen
    widget.showFullScreen();
#elif defined Q_OS_WIN32 || defined Q_OS_LINUX
    // On desktop Windows and Linux, show the widget in WVGA resolution
    widget.setFixedSize(800, 480);
    widget.show();
#endif
```

Calling Qt Code from JavaScript

At some point, there may be a need to call Qt code from JavaScript. One such case is closing the application. The JavaScript code "`window.close()`", which would normally close the window, cannot be used to close a window in Qt. For that, you will need to write some Qt code and call it from JavaScript. Here's the Qt function that exits the application:

```
#include <QtGui/QApplication>

void WRTWidgetWindow::close()
{
    QApplication::exit();
}
```

In order to call this function from JavaScript, you will need to make the Qt class available to 'the JavaScript side' (or in more technical terms, from within QWebFrame's JavaScript context). Add the following function to the class:

```
#include <QtWebKit/QWebFrame>

void WRTWidgetWindow::addJavaScriptObject()
{
    // Make the WRTWidgetWindow available from within the QWebFrame's
    // JavaScript context (as variable "clientApp")
    this->webView->page()->mainFrame()->addToJavaScriptWindowObject("clientApp",
        this);
}
```

The above function should be called when the QWebView component is created:

```
QWebView* WRTWidgetWindow::createWebView()
{
    QWebView* view = new QWebView(this);
    // Call the addJavaScriptObject slot when the javaScriptWindowObjectCleared
    // signal is emitted, i.e. before the loading of the new window object
```

```
    // starts
    connect(view->page()->mainFrame(), SIGNAL(javaScriptWindowObjectCleared()),
        this, SLOT(addJavaScriptObject()));
    view->load(QUrl("qrc:/html/index.html"));
    return view;
}
```

Also add the necessary function declarations into the header file:

```
public:
    void close();

private slots:
    void addJavaScriptObject();
```

After these steps, you can write a JavaScript function which calls WRTWidgetWindow's `close()` function .

```
// Exits the application
function exit() {
    clientApp.close();
}
```

Finally, call the JavaScript function. From HTML, for example:

```
<input id="btnExit" type="button" onclick="exit();" value="Exit" />
```

Example project

The original BetaLabs widget can be downloaded at http://www.developer.nokia.com/info/sw.nokia.com/id/60bbf194-224a-44eb-b352-da5ad53069fe/Web_Run-Time_BetaLabsWidget_Example.html

The first version of the ported application is available at File:BetaLabsClientQtWebKit.zip.

Related documentation

For more information, refer to the following code snippets:

- Exposing QObjects to Qt Webkit
- Calling an exposed QObject slot from Qt WebKit with JavaScript
- Connecting to a QObjects signal with JavaScript slot in Qt WebKit

QtWebKitStub

There is also a template application available to make it simpler to start developing your own QtWebKit applications. To download it, go to QtWebKitStub.