The QtWebKit Bridge

Overview

The technology

The QtWebKit bridge is a mechanism that extends WebKit's JavaScript environment to access native objects that are represented as QObjects. It takes advantage of the QObject introspection, a part of the Object Model, which makes it easy to integrate with the dynamic JavaScript environment, for example QObject properties map directly to JavaScript properties.

For example, both JavaScript and QObjects have properties: a construct that represent a getter/setter pair under one name.

Use Cases

There are two main use cases for the QtWebKit bridge. Web content in a native application, and Thin Clients.

Web Content in a Native Application

This is a common use case in classic Qt application, and a design pattern used by several modern applications. For example, an application that contains a media-player, playlist manager, and music store. The playlist manager is usually best authored as a classic desktop application, with the native-looking robust QWidgets helping with producing that application. The media-player control, which usually looks custom, can be written using the Graphics View framework or with in a declarative way with QtDeclarative. The music store, which shows dynamic content from the internet and gets modified rapidly, is best authored in HTML and maintained on the server.

With the QtWebKit bridge, that music store component can interact with native parts of the application, for example, if a file needs to be saved to a specific location.

Thin Client

Another use case is using Qt as a native backend to a full web application, referred to here as a thin client. In this use-case, the entire UI is driven by HTML, JavaScript and CSS, and native Qt-based components are used to allow that application access to native features not usually exposed to the web, or to enable helper components that are best written with C++.

An example for such a client is a UI for a video-on-demand service on a TV. The entire content and UI can be kept on the server, served dynamically through HTTP and rendered with WebKit, with additional native components for accessing hardware-specific features like extracting the list of images out of the video.

Difference from Other Bridge Technologies

Of course QtWebKit is not the only bridge technology out there. NPAPI, for example, is a long-time standard or web-native bridging. Due to Qt's meta-object system, full applications built partially with web-technologies are much easier to develop. NPAPI, however, is more geared towards cross-browser plugins, due to it being an accepted standard.

When developing a plugin for a browser, NPAPI is recommended. When developing a full application that utilizes HTML-rendering, the QtWebKit bridge is recommended.

Relationship with QtScript

The QtWebKit bridge is similar to QtScript, especially to some of the features described in the Making Applications Scriptable page. However, as of Qt 4.7, full QtScript API is not supported for web applications. That is planned as an enhancement for future versions. You might notice that some of the features described here are an exact copy of the ones described in the Making Applications Scriptable page. That is because the QtWebKit bridge is a subset of that functionality, and this page tries to capture the full capabilities available through the QtWebKit bridge specifically.

Accessing QObjects

Creating the link via QWebFrame

By default, no QObjects are accessible through the web environment, for security reasons. To enable web content access for a native QObject, the application must explicitly grant it access, using the following call:

```
// ...
QWebFrame *frame = myWebPage->mainFrame();
frame->addToJavaScriptWindowObject("someNameForMyObject", myObject);
// ...
```

See QWebFrame::addToJavaScriptWindowObject() for more information.

Using Signals and Slots

The QtWebKit bridge adapts Qt's central Signals and Slots feature for scripting. There are three principal ways to use signals and slots with the QtWebKit bridge:

- **Hybrid C++/script**: C++ application code connects a signal to a script function. For example, the script function can be a function that the user has typed in, or one that you have read from a file. This approach is useful if you have a QObject but don't want to expose the object itself to the scripting environment; you just want a script to be able to define how a signal should be reacted to, and leave it up to the C++ side of your application to establish the connection.
- **Hybrid script/C++**: A script can connect signals and slots to establish connections between pre-defined objects that the application exposes to the scripting environment. In this scenario, the slots themselves are still written in C++, but the definition of the connections is fully dynamic (script-defined).
- **Purely script-defined**: A script can both define signal handler functions (effectively "slots written in JavaScript"), *and* set up the connections that utilize those handlers. For example, a script can define a function that will handle the QLineEdit::returnPressed() signal, and then connect that signal to the script function.

Note that QtScript functions such as qScriptConnect are unavilable in the web environment.

Signal to Function Connections

```
connect(function);
```

In this form of connection, the argument to `connect()` is the function to connect to the signal.

```
function myInterestingScriptFunction() { ... }
...
myQObject.somethingChanged.connect(myInterestingScriptFunction);
```

The argument can be a JavaScript function, as in the above example, or it can be a QObject slot, as in the following example:

```
myQObject.somethingChanged.connect(myOtherQObject.doSomething);
```

When the argument is a QObject slot, the argument types of the signal and slot do not necessarily have to be compatible; If necessary, the QtWebKit bridge will, perform conversion of the signal arguments to match the argument types of the slot.

To disconnect from a signal, you invoke the signal's `disconnect()` function, passing the function to disconnect as argument:

```
myQObject.somethingChanged.disconnect(myInterestingFunction);
myQObject.somethingChanged.disconnect(myOtherQObject.doSomething);
```

When a script function is invoked in response to a signal, the `this` object will be the Global Object.

Signal to Member Function Connections

```
connect(thisObject, function)
```

In this form of the `connect()` function, the first argument is the object that will be bound to the variable, `this`, when the function specified using the second argument is invoked.

If you have a push button in a form, you typically want to do something involving the form in response to the button's `clicked` signal; passing the form as the `this` object makes sense in such a case.

```
var obj = { x: 123 };
var fun = function() { print(this.x); };
myQObject.somethingChanged.connect(obj, fun);
```

To disconnect from the signal, pass the same arguments to `disconnect()`:

```
myQObject.somethingChanged.disconnect(obj, fun);
```

Signal to Named Member Function Connections

```
connect(thisObject, functionName)
```

This form of the `connect()` function requires that the first argument is the object that will be bound to the variable `this` when a function is invoked in response to the signal. The second argument specifies the name of a function that is connected to the signal, and this refers to a member function of the object passed as the first argument (thisObject in the above scheme).

Note that the function is resolved when the connection is made, not when the signal is emitted.

```
var obj = { x: 123, fun: function() { print(this.x); } };
myQObject.somethingChanged.connect(obj, "fun");
```

To disconnect from the signal, pass the same arguments to `disconnect()`:

```
myQObject.somethingChanged.disconnect(obj, "fun");
```

Error Handling

When `connect()` or `disconnect()` succeeds, the function will return `undefined`; otherwise, it will throw a script exception. You can obtain an error message from the resulting `Error` object. Example:

```
try {
    myQObject.somethingChanged.connect(myQObject, "slotThatDoesntExist");
} catch (e) {
    print(e);
}
```

Emitting Signals from Scripts

To emit a signal from script code, you simply invoke the signal function, passing the relevant arguments:

```
myQObject.somethingChanged("hello");
```

It is currently not possible to define a new signal in a script; i.e., all signals must be defined by C++ classes.

Overloaded Signals and Slots

When a signal or slot is overloaded, the QtWebKit bridge will attempt to pick the right overload based on the actual types of the QScriptValue arguments involved in the function invocation. For example, if your class has slots `myOverloadedSlot(int)` and `myOverloadedSlot(QString)`, the following script code will behave reasonably:

```
myQObject.myOverloadedSlot(10);   // will call the int overload
myQObject.myOverloadedSlot("10"); // will call the QString overload
```

You can specify a particular overload by using array-style property access with the normalized signature of the C++ function as the property name:

```
myQObject['myOverloadedSlot(int)']("10");   // call int overload; the argument is converted to
myQObject['myOverloadedSlot(QString)'](10); // call QString overload; the argument is converted
```

If the overloads have different number of arguments, the QtWebKit bridge will pick the overload with the argument count that best matches the actual number of arguments passed to the slot.

For overloaded signals, JavaScript will throw an error if you try to connect to the signal by name; you have to refer to the signal with the full normalized signature of the particular overload you want to connect to.

Invokable Methods

Both slots and signals are invokable from script by default. In addition, it's also possible to define a method that's invokable from script without it being a signal or a slot. This is especially useful for functions with return types, as slots normally do not return anything (it would be meaningless to return values from a slot, as the connected signals don't handle the returned data). To make a non-slot method invokable, simply add the Q_INVOKABLE macro before its definition:

```
class MyObject : public QObject
{
    Q_OBJECT

public:
    Q_INVOKABLE void thisMethodIsInvokableInQtScript();
    void thisMethodIsNotInvokableInQtScript();

    ...
};
```

Accessing Properties

The properties of the QObject are available as properties of the corresponding JavaScript object. When you manipulate a property in script code, the C++ get/set method for that property will automatically be invoked. For example, if your C++ class has a property declared as follows:

```
    Q_PROPERTY(bool enabled READ enabled WRITE setEnabled)
```

then script code can do things like the following:

```
myQObject.enabled = true;

...
```

```
myQObject.enabled = !myQObject.enabled;
```

Accessing Child QObjects

Every named child of the QObject (that is, for which QObject::objectName() is not an empty string) is by default available as a property of the JavaScript wrapper object. For example, if you have a QDialog with a child widget whose `objectName` property is `"okButton"`, you can access this object in script code through the expression

```
myDialog.okButton
```

Since `objectName` is itself a Q_PROPERTY, you can manipulate the name in script code to, for example, rename an object:

```
myDialog.okButton
myDialog.okButton.objectName = "cancelButton";
// from now on, myDialog.cancelButton references the button
```

Data types

When calling slots, receiving signals or accessing properties, usually some payload is involved. For example, a property "text" might return a QString parameter. The QtWebKit bridge does the job of converting between a given JavaScript data-type, and the expected or given Qt type. Each Qt type has a coresponding set of rules of how JavaScript treats it.

The data type conversions are also applicable for the data returned from non-void invokable methods.

Numbers

All Qt numeric data types are converted to or from a JavaScript number. These include int, short, float, double, and the porable Qt types (qreal, qint etc). A special case is QChar; If a slot expects a QChar, the QtWebKit bridge would use the unicode value in case of a number, or the first character in a string.

Note that non-standard (typedefed) number types are not automatically converted to or from a JavaScript number - it's advised to use standard number types for signal, slots and properties.

When a non-number is passed as an argument to a method or property that expects a number, the appropriate JavaScript conversion function (parseInt / parseFloat) would be used.

Strings

When JavaScript accesses methods or properties that expect a QString, the QtWebKit bridge will automatically convert the value to a string (if it is not already a string), using the built-in JavaScript toString method.

When a QString is passed to JavaScript from a signal or a property, The QtWebKit bridge will convert it into a JavaScript string.

Date & Time

Both QDate, QTime and QDateTime are automatically translated to or from the JavaScript Date object. If a number were passed as an argument to a method that expects one of the date/time types, the QtWebKit bridge would treat it as a timestamp. If a sting is passed, QtWebKit would try different Qt date parsing functions to find the right one.

Regular Expressions

The QtWebKit bridge would automatically convert JavaScript RegEx object to a QRegExp. If a string is passed to a method expecting a QRegExp, the string would be converted to that QRegExp.

Lists

The QtWebKit bridge treats several types of lists in a special way: QVariantList, QStringList, QObjectList and QList<int>. When a slot or property expects one of those list types, the QtWebKit bridge would try to convert a JavaScript array into that type, converting each of the array's elements to the single-element type of the list.

The most useful type of list is perhaps QVariantList, which can be converted to from any JavaScript array.

Compound (JSON) objects

JavaScript compound objects, also known as JSON objects, are variables that hold a list of key-value pairs, where all the keys are strings and the values can have any type. This translates very well to QVariantMap, which is nothing more than a QMap of QString to QVariant.

The seamless conversion between JSON objects and QVariantMap allows for a very convenient way of passing arbitrary structured data between C++ and the JavaScript environment. The native QObject has to make sure that compound values are converted to QVariantMaps and QVariantLists, and JavaScript is guaranteed to receive them in a meaningful way.

Note that types that are not supported by JSON, such as JavaScript functions and getters/setters, are not converted.

QVariants

When a slot or property accepts a QVariant, the QtWebKit bridge would create a QVariant that best matches the argument passed by JavaScript. A string, for example, would become a QVariant holding a QString, a normal JSON object would become a QVariantMap, and a JavaScript array would become a QVariantList.

Using QVariants generously in C++ in that way makes C++ programming feel a bit more like JavaScript programming, as it adds another level of indirection. Passing QVariants around like this q is very flexible, as the program can figure out the type of argument in runtime just like JavaScript would do, but it also takes away from the type-safety and robust nature of C++. It's recommended to use QVariants only for convenience high-level functions, and to keep most of your QObjects somewhat type-safe.

QObjects

A pointer to a QObject or a QWidget can be passed as payload in signals, slots and properties. That object can then be used like an object that's exposed directly; i.e. its slots can be invoked, its signals connected to etc. However, this functionality is fairly limited - the type used has to be QObject* or QWidget*. If the type specified is a pointer to a non-QWidget subclass of QObject, the QtWebKit bridge would not recognize it to be a QObject.

In general its advised to use care when passing QObjects as arguments, as those objects don't become owned by the JavaScript engine; That means that the application developer has to be extra careful not to try to access QObjects that have already been deleted by the native environment.

Pixmaps and Images

The QtWebKit bridge handles QPixmaps and QImages in a special way. Since QtWebKit stores QPixmaps to represent HTML images, QPixmaps coming from the native environment can be used directly inside WebKit. A QImage or a QPixmap coming from the Qt environment would convert to an intermediate JavaScript object, that can be represented like this:

```
    {
        width: ...,
        height: ...,
        toDataURL: function() { ... },
        assignToHTMLImageElement: function(element) { ... }
    }
```

The JavaScript environment can then use the pixmap it gets from Qt and display it inside the HTML environment, by assigning it to an existing <img /> element using assignToHTMLImageElement. It can also use the toDataURL() function, which allows using the pixmap as the src attribute of an image or as a background-image url. Note that the toDataURL() function is costly and should be used with caution.

Example code:

C++:

```
    class MyObject : QObject {
        Q_OBJECT
        Q_PROPERTY(QPixmap myPixmap READ getPixmap)

    public:
        QPixmap getPixmap() const;
    };

    /* ... */

    MyObject myObject;
    myWebPage.mainFrame()->addToJavaScriptWindowObject("myObject", &myObject);
```

HTML:

```
    <html>
        <head>
            <script>
                function loadImage()
                {
                    myObject.myPixmap.assignToHTMLImageElement(document.getElementById("imageElemen
                }
            </script>
        </head>
        <body onload="loadImage()">
            <img id="imageElement" width="300" height="200" />
        </body>
    </html>
```

When a Qt object expects a QImage or a QPixmap as input, and the argument passed is an HTML image element, the QtWebKit bridge would convert the pixmap assigned to that image element into a QPixmap or a QImage.

QWebElement

A signal, slot or property that expects or returns a QWebElement can work seamlessly with JavaScript references to DOM elements. The JavaScript environment can select DOM elements, keep them in variables, then pass them to Qt as a QWebElement, and receive them back. Example:

C++:

```
 class MyObject : QObject {
        Q_OBJECT

    public slots:
        void doSomethingWithWebElement(const QWebElement&);
    };

    /* ... */
```

```
MyObject myObject;
myWebPage.mainFrame()->addToJavaScriptWindowObject("myObject", &myObject);
```

HTML:

```
<html>
     <head>
         <script>
             function runExample() {
                myObject.doSomethingWithWebElement(document.getElementById("someElement"));
             }
         </script>
     </head>
     <body onload="runExample()">
         <span id="someElement">Text</span>
     </body>
 </html>
```

This is specifically useful to create custom renderers or extensions to the web environment. Instead of forcing Qt to select the element, the web environment already selects the element and then send the selected element directly to Qt.

Note that QWebElements are not thread safe - an object handling them has to live in the UI thread.

Architecture issues

Limiting the Scope of the Hybrid Layer

When using QtWebKit's hybrid features, it is a common pitfall to make the API exposed to JavaScript very rich and use all its features. This, however, leads to complexity and can create bugs that are hard to trace. Instead, it is advisable to keep the hybrid layer small and manageable: create a gate only when there's an actual need for it, i.e. there's a new native enabler that requires a direct interface to the application layer. Sometimes new functionality is better handled internally in the native layer or in the web layer; simplicity is your friend.

This usually becomes more apparent when the hybrid layer can create or destroy objects, or uses signals slots or properties with a QObject* argument. It is advised to be very careful and to treat an exposed QObject as a system - with careful attention to memory management and object ownership.

Internet Security

When exposing native object to an open web environment, it is importwhichant to understand the security implications. Think whether the exposed object enables the web environment access to things that shouldn't be open, and whether the web content loaded by that web page comes from a trusted. In general, when exposing native QObjects that give the web environment access to private information or to functionality that's potentially harmful to the client, such exposure should be balanced by limiting the web page's access to trusted URLs only with HTTPS, and by utilizing other measures as part of a security strategy.