Exercises Lecture 10 – Networking and Integrating the Web

Aim: This exercise will help you learn how to use networking classes and will take you through the process of using Qt WebKit to create an Web-enabled application.

Duration: 1h

Creating a simple web browser

This exercise come with a source code package. Extract that package into a working directory. The content is a number of projects that serve as starting points for each step of this exercise.

In this first step, you will implement a basic web browser application. Start from the *webbrowser* project. The project contains a basic implementation that shows a window with some widgets.

First task is to add a `QWebView` widget to the design. There is a member variable, `browser`, for holding it, so instantiate a `QWebView` widget and hold the pointer in the designated variable. Also, add the web view to the `mainLayout`.

Next, you will take care of the back, forward and reload buttons. You do this by retrieving the corresponding `QAction` objects from the `QWebView`. You can retrieve the actions `Back`, `Forward` and `Reload` using the `QWebView::pageAction()` method. Assign the actions to the buttons using the setDefaultAction method. This should give the buttons an icon, some text and functionality.

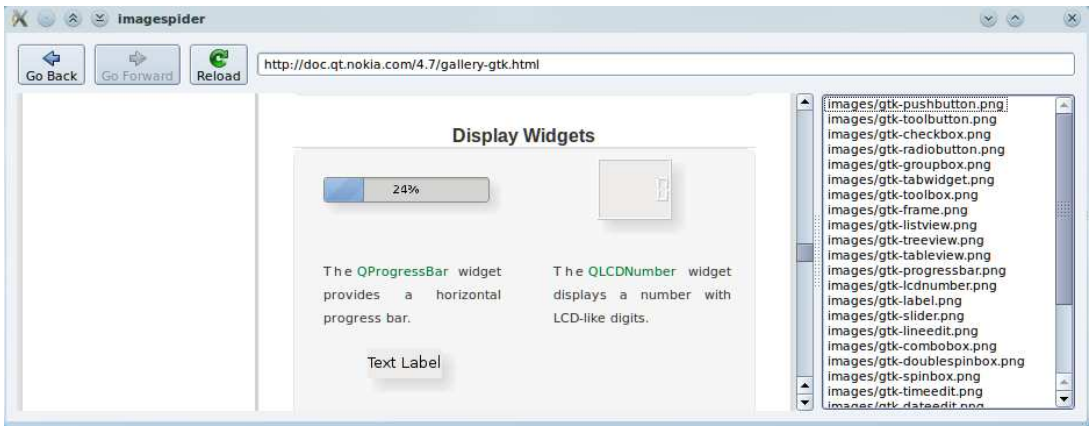
Now make your browser load the URL provided in the address bar whenever the `loadPage()` slot is called. The method `QUrl::fromUserInput()` can transform the string given by the user into a proper URL. Then ask the `QWebView` to `load` the URL. To make the slot work, connect the `returnPressed` signal from the `addressBar` to the `loadPage` slot. You should now be able to use the browser for basic browsing.

Letting the user set the URL only solves half the problem. Web pages sometimes redirect the browser. To handle this, connect the `urlChanged(QUrl)` signal from the `browser` to the `updateAddressBar(QUrl)` slot.

In the `updateAddressBar` slot, set the text of the address bar to the given URL. You can convert a `QUrl` object to a `QString` using the `toString` method.

Harvesting the Web

Providing users with web browsing abilities is good, however, it is also interesting to be able to traverse the web programatically. The *imagespider* project is a starting point for an application that acts like a web browser, but also harvests images from the web pages visited.



The idea is that the `scanPageForImages` slot is called as soon as a page has been loaded. It extracts all `img` tags and extracts the `src` attributes from them, populating the list of images to the right.

First, connect the `scanPageForImages` slot to the `loadFinished(bool)` signal of the `QWebView`.

In the `scanPageForImages()` slot, the plan is to retrieve all `img` tags and add their `src` attributes to the `images` string list. To be able to access the img elements, you first need access to the main frame of the web page. You get it through the web view as shown below.

```
    QWebFrame *frame = browser->page()->mainFrame();
```

From the frame a `QWebElementList` can be retrieved using the `findAllElements("img")` method. You can iterate over every `QWebElement` using the `foreach` macro.

For each element, use the `attribute` method to retrieve the value of the `src` attribute. Append this value to the string list.

When visiting a web page, wait for the page to fully load. You can see that this has happened when the reload button next to the address bar is enabled. When that happens, the list to the right should fill with image paths. You have harvested the page of its images.

UDP Chat

Until now, we have relied on the HTTP protocol, which is based on TCP/IP. For this next step, you will use UPD datagrams over IP directly. Use the *broadcastchat* project as the starting point.

The application provides a chat program user interface and the methods needed to implement a simple network chat system. The goal for this step is to add the UDP networking functions needed to transmit and receive messages.

Start by implementing the `startNetworking()` method. In it create a UDP socket and bind it to the given port and the broadcast host address. Store the pointer to the socket in the `socket` member variable. Make sure to pass the `QudpSocket::ShareAddress` bind mode flag when binding the socket, so that you can run another copy of your program on the same machine for testing purposes.

Having created the socket, connect the `readyRead` signal of the socket to the `processPendingDatagrams` slot to receive notifications whenever a new datagram arrives.

In the `processPendingDatagrams` slot, use the following code to read the next datagram into a buffer.

```
    QByteArray buffer;
    buffer.resize(socket->pendingDatagramSize());
    socket->readDatagram(buffer.data(), buffer.size());
```

Now use `decodeMessage` to convert the buffer contents into a pair of strings that carry the nickname of the sender and the message body. Pass these onto the `showMessage` with appropriate parameters as shown below to have the message displayed in the window.

```
    showMessage(pair.first, pair.second);
```

Finally implement the `sendMessage()` slot. Use the `encodeMessage` method to convert the (nick, message) pair into an opaque byte array and broadcast the message through the socket using the `writeDatagram` method. Broadcast to all on the network by providing the host address `QHostAddress::Broadcast`. Also, use the `socket->localPort()` method to retrieve the port to use.

Solution Tips

Step 2

To get all image tags from the page, use the `QWebFrame::findAllElements()` call and pass it the “img” search string.

```
    QWebElementCollection imageTags = frame->findAllElements("img");
```

Then loop over the result and use QWebElement::attribute() to ask for the “src” attribute. Finally add its value to the list.

```
    foreach(QWebElement element, imageTags) {
        QString src = element.attribute("src");
        if(!src.isEmpty())
            images.append(src);
    }
```

Step 3

To bind the UDP socket, use the following call:

```
socket->bind(QHostAddress::Broadcast,
    port, QUdpSocket::ShareAddress);
```

To broadcast to all stations in the local network, use `QHostAddress::Broadcast` as the destination address for `QUdpSocket::writeDatagram()`. Be sure to use the same port as for receiving messages.