# Lab 5 – Networking and Integrating the Web

**Aim:** The lab focuses combining regular desktop application features with the technology of the Web, building a hybrid application. This involves both mixing the web with application contents, as well as gathering data from the web programatically.

**Duration:** 2 h

## *Introduction*

In this lab you will implement an application that shows a report of activities for a given period of time. The report will be delivered to the user in form of a web page and shown inside the application. The program will extend the page with the ability to control what is shown and how it is shown.

Web technology will be mixed with application technology, forming a hybrid application.

It is assumed you already feel comfortable with using Qt widgets and handling signals and slots. Instructions for this lab will only discuss the task at hand, not every individual step that you need to take to solve it.

## *Preparing the web page*

For this project, you will use a simple static HTML file as the report. In a real world setting, you would most likely get the report from a web server.

Create a simple web page using a text editor of your choice. You can tweak backgrounds, styles and decorations and add other content if you like. You need to make sure to place a set of  tags with a format described below so that your application can detect them and work on their content.

- Each tag contains a mandatory `id` attribute with a value in format `report_day_month_year` where the “day”, “month” and “year” parts are substituted with two digit numbers specifying the day of the month, the month of the year and the two last digits of the year;
- Each tag contains a mandatory `class` attribute with the value of `report` for easy detection of reports;
- Each tag contains a mandatory `div` child container with a mandatory `class` attribute set to `reportcontent`; the web page may contain styling information for the `reportcontent` class such as background, border, etc.;
- Each of the child `div` tags can (optionally) contain any textual description of activities performed on this particular day;
- Each of the child `div` tag can contain `object` tags for describing the custom object that is to be embedded in the web page using a browser plugin; the tag has the format shown below where `width` and `height` attributes specify the size of the chart and `title` and `value` parameters specify the title of the chart and percentage of progress to be shown in the chart.

```
<object type="application/x-chart" width="500" height="400">
    <param name="title" value="Title of the activity"/>
    <param name="value" value="100"/>
</object>
```

An example of a simple web page that satisfies the given description is shown below.

```
<html>
    <body>
        <div class="report" id="report_18_12_10">
            <h2>Report for 18-Dec-2010</h2>
            <div class="reportcontent">
                <p>Doing exercises in reading and listening.</p>
                <table><tr>
                    <td><object type="application/x-chart"
                               width="300" height="300">
                            <param name="title" value="Reading"/>
                            <param name="value" value="80"/>
                        </object>
                    </td>
                    <td><object type="application/x-chart"
                               width="300" height="300">
                            <param name="title" value="Writing"/>
                            <param name="value" value="100"/>
                        </object>
                    </td>
                </tr></table>
            </div>
        </div>
```

```
        <div class="report" id="report_19_12_10">
            <h2>Report for 19-Dec-2010</h2>
            <div class="reportcontent">
                <p>Doing the dishes.</p>
                <object type="application/x-chart"
                        width="300" height="300">
                     <param name="title" value="Dishes"/>
                     <param name="value" value="20"/>
                </object>
            </div>
        </div>
    </body>
</html>
```

### *The application window*

Start a by creating a new Qt application project. Create a user interface derived from `QMainWindow` with a `QCalendarWidget` and a `QWebView` in a horizontal layout. Enable WebKit support for your project by adding the webkit module to the list of active modules in the project file.

In the main window, create an action for loading the report file, add it to a toolbar and menu. Connect its `triggered()` signal to a custom slot. In that slot, use `QWebView::load()` to load the web page, in this case the HTML file that you prepared earlier. Either use `QUrl::fromLocalFile()` to create a proper url from a file path or embed the web page in a resource file and use an url pointing there (it should have a `qrc:/` prefix).

Try running the application to verify that the web page loads into the view. In case of problems make sure the path you use is correct and the file is readable for the user.

> It is good practice to disable all the components the user should not interact with at a given state of the application. In this case it is a good idea to set the "enabled" property of the calendar widget to false when the application is started and only enable the widget when it becomes useful to the user. This will happen when the page is loaded into the browser window.

## Extracting data and showing reports

Having loaded and shown the page in question, the next step is to extract the relevant data from the webpage. Connect to the `loadFinished()` signal of the `QWebView` object to a custom slot. This will let you perform tasks when the page is fully loaded. What the slot needs to do, is to access the main frame of the page displayed in the view and find all elements representing reports.

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

Once you get hold of the frame, Use `QWebFrame::findAllElements()` to get a collection of all `div` tags with class `report`. The proper CSS selector for that is `div.report`. Then iterate over the collection and match values of `id` attribute against a regular expression detecting the dates associated with reports.

```
foreach(QWebElement report, reports) {
        QString id = report.attribute("id");
        QRegExp rx("report_(\\d+)_(\\d+)_(\\d+)");
        if(!rx.exactMatch(id)) continue;
        QDate date(2000+rx.cap(3).toInt(),
                   rx.cap(2).toInt(),
                   rx.cap(1).toInt());
```

Store the dates and corresponding `QWebElement` objects in a map (make the map a private class member) where the date is the key and the web element is the value of the entry. This will let you have quick access by date to all available reports.

While you process the elements you should do two more things. First access the report element's child tag `div` with class `reportcontainer` and use `QWebElement::setStyleProperty()` to set the `display` style attribute to `none`. This effectively hides the elements that have been processed from the web page. The other thing to do is to provide a visual hint on the calendar widget that a report is available for a particular date. Use the `QCalendarWidget::setDateTextFormat()` method to do so. You

can change the background, foreground color or even the font for any given date.

> Do not forget to always start with a clean state of all objects you work with before you start filling them with data. Clear the map of date-report associations and revert all custom formatting of the calendar widget so that your code still works properly if it gets called again and again (e.g. when the user requests a page refresh).
>
> If you disabled the calendar widget during the loading of the page, this is a good moment to enable it again.

If you run the application now, you should get a list of all report titles, but all the report contents should be hidden.

The missing feature is to show the contents of a given report when the corresponding date is selected in the calendar widget. The calendar widget's `selectionChanged()` signal tells you when a new date has been picked. Connect it to a custom slot that you create.

In the slot, retrieve the report element from the map based on the date selected in the calendar widget and set the `display` style property of the report's container to `block`.

You should also collapse the previously opened report. To do that, you need to keep track of the currently shown report. You could scan the collection of available reports and determine which one is visible, but it is much easier to store the last opened report element (or its date) in a member variable. This way you can immediately perform the operation of hiding the old report when you open the new one. Remember to update the state of the variable you just created when reloading the page or opening a new report. At this point reports should be shown and hidden as dates in the calendar widget are clicked.

## Exposing the calendar

To complete the user experience, it should also be possible to expand each report by interacting with the web page. It is possible to change the display style attribute from a script run in the web page, but this would put the calendar widget out of sync with the shown contents. A better solution is to make the calendar API available to the script. All that is need to make that happen is to add it as a global object in the JavaScript scope. You do that using the `addToJavaScriptWindowObject` call:

```
frame->addToJavaScriptWindowObject("calendar", ui->calendarWidget,
                                   QScriptEngine::QtOwnership);
```

The `frame` is the object retrieved during the data extraction phase (which is where you can place the above code as well). When adding an object to the JavaScript scope, all the properties and slots of the object in question become available from within the web page. This makes it possible to add code to the web page that interacts with QObjects. For instance, the snippets shown below changes the date of the calendar widget.

```
<input type="button" value="Open" onClick="calendar.selectedDate = '2010-12-12';"/>
```

As the date is changed, the `selectionChanged()` signal is emitted. This in turn triggers the same code path that is used when the user interacts directly with the widget. Now add buttons to your web page that allow changing the current show report.

In real life the web page would probably contain a library of scripts that would be enabled or disabled depending on whether the web page was accessed from within a 'real' web browser (so that all the basic functionality of the web was present) or from a hybrid application like here.

## Self check

- Ensure that the web page loads into your application; check the file path or URL if downloading from a remote server.
- Check that every report listed on the page also is marked in the calendar.
- Verify that clicking dates in the calendar causes reports to collapse and expand.
- Verify that clicking the “show report” button in the web page gives the same effect as when interacting directly with the calendar.

### *Browser plugin*

The last integration between the web page and the application is to embed a Qt widget in the web page. This is achieved by implementing a plugin that handles the mime-type `application/x-chart`. To embed a widget in a web page, the following is required:

- Plugin loading needs to be enabled for the web page
- The chart widget needs to be implemented
- The chart widget needs to be exported as a browser plugin handling `application/x-chart` data type.

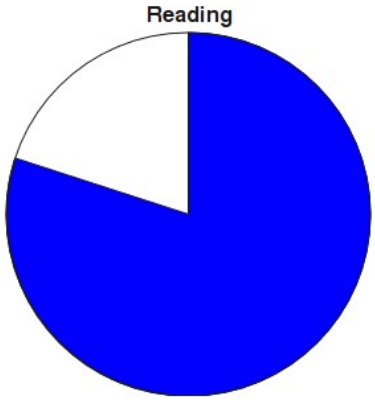
## Enabling plugins

Enabling plugins is the easiest part. Each web page in QtWebKit has a settings object of the class `QWebSettings` associated with it. You can manipulate this object to control the behavior of the page. One of the attributes that can be set is `PluginsEnabled`. Set it to true to allow WebKit to use plugins.

> This manipulates settings for a particular `QWebPage` object only. You can also change settings for all web pages by accessing the `QWebSettings::globalSettings()` static method that gives you access to the global settings.

## The plugin widget

The widget that you will plug in shows a pie chart. It is used to show the percentage of the work done along with the title of the task. An example of the widget is shown below.




To implement the widget, create the properties `QString title` and `int value` and provide proper setters and getters for them. Make the title font bold and a bit larger than normal. You can then use `QFontMetrics` to calculate the space taken by the title so that the pie does not intersect with the text.

## The browser plugin

To export a plugin to webkit, you have to provide an implementation of the `QWebPluginFactory` interface. This couples the widget class with a mime-type. By referring to the mime-type in the web page, you will tell webkit to use your widget.

Start by subclassing `QWebPluginFactory`. Implement the `plugins()` method to return a list of `Plugin` objects, describing the plugins exported by the factory. In this case, the list will contain only one item that describes the chart plugin.

As each plugin can handle more than one mime-type, the description object contains a field of `MimeType` objects. You only handle `application/x-chart` so this list will also contain only one item. You only need to fill the `name` of the mime-type. It is good practice to provide a name and a short description for the plugin object.

> There are many standard mime-types out there like `application/pdf`, `image/jpeg` or `text/plain`. There is also a need for custom mime-types. To distinguish between standard mime-types and custom ones, there is a convention that all custom types contain a 'x-' prefix. Unless there is a strong reasoning behind doing otherwise they should belong to the group 'application'. This prevents potential name clashes with any standard types that might be introduced in the future.

The second part of the plugin factory interface that needs to be implemented is the `create()` method. In it, you should instantiate the widget based on the mime-type given. Also initialize it using the provided url and a set of arguments described by the method parameters.

Since the widget in this case does not work on any external data, the url can be ignored. Argument names and values correspond to `param` child tags of the html object tag requesting the plugin. It is important that you only create the widget if the mime-type passed as an argument to `create` matches the type you really support. In all other cases you should return 0.

If the type is supported, create an instance of the chart widget and scan the list of argument names for the ones you support (i.e. 'title' and 'value'). Use the widget's API to set these values on the widget. Note that all the values are given as strings so you might have to convert some of them to proper types. When the widget has been setup,  return it so that webkit can place it in the web page.

The last step is to inform your application that your plugin factory exists. You do that by calling the setPluginFactory method once. You can do that in your main window constructor.

```
ui->webView->page()->setPluginFactory(
                      new ChartPluginFactory(ui->webView->page()) );
```

Now, run the application to see the final result.

## *Conclusions*

There are many ways to integrate the desktop application with the web. Integrating widgets in the web pages, exposing C++ objects to JavaScripts, as well as extracting information from web pages.

The application in this lab can be extended in many ways. The interested student can start with simple things such as adding functionality to the browser plugin or coupling the plugin more tightly with the page content. Exposing more C++ application objects to scripts is also an option, as is extracting additional data from the web page.

There are also more complex features that can be added. Consider providing editing support for a report inside the application. Data from reports can be extracted and put into widgets. When the widgets change, AJAX requests post the changes to the web server using a `QNetworkAccessManager` manager. If you have time left, try adding some features to the project on your own.