CHALMERS

Using WebKit as a cross-platform graphical user interface renderer for the Spotify client

Master of Science Thesis in the Programme Software Engineering and Technology

Charles Dinsdale
Alexander Pekkari

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2009

Using WebKit as a cross-platform graphical user interface renderer for the Spotify client

CHARLES DINSDALE
ALEXANDER PEKKARI

Examiner: CHRISTER CARLSSON

Department of Computer Science and Engineering
University of Gothenburg
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Work done in cooperation with:
Spotify AB (R&D)
Humlegårdsgatan 20, 4 tr
114 46 Stockholm
Sweden

Department of Computer Science and Engineering
Göteborg, Sweden October 2009

Cover: The Spotify Logo (copyright of Spotify AB) and the WebKit logo (copyright of Apple Inc.).

ABSTRACT

Spotify provides a streaming music service used by thousands of users on a daily basis. The Spotify client consists of a simple yet effective graphical user interface. The interface is defined programmatically, in C++ and Objective-C, leaving little space for modifications post-compile-time.

WebKit is an open source web renderer, providing a framework that performs rendering of web pages. WebKit is used in Safari, Apple's web browser flagship.

HTML, CSS and JavaScript are versatile, and widely used, tools that are used to construct graphical user interfaces for the web.

This thesis investigates and lists the benefits of integrating WebKit into the Spotify client in order to utilize the flexibility of WebKit and the surrounding tools and languages that WebKit make available. Some of the overwhelming number of drawbacks and obstacles spawned by the integration process are explained in detail and reflected upon, explaining why this solution may, for most purposes, be a sub-optimal one.

Keywords:

WebKit, Spotify, HTML, JavaScript, CSS, TrollTech Qt, Cocoa, DOM, GUI, Integration

SAMMANFATTNING

Spotify erbjuder en streamad musiktjänst som används av tusentals användare dagligen. Spotifyklienten tillhandahåller ett simpelt och effektivt användargränssnitt som är programmatiskt definierat, i C++ och Objective-C, vilket lämnar lite utrymme för modifikationer utan att behöva kompilera om applikationen.

WebKit är en öppen mjukvara vars syfte är att rendera webbinnehåll. För att uppfylla detta syfte kräver WebKit stödbibliotek. På Mac OS X används WebKit för att driva Safari, Apples webbläsarflaggskepp.

HTML, CSS och JavaScript är flexibla verktyg som är vitt använda i webbutveckling. Dessa verktyg används för att konstruera webbinnehåll, vilket användargränssnittet kommer bestå av i denna tes.

Denna tes undersöker och listar fördelarna med att integrera WebKit i Spotifyklienten för att utnyttja flexibiliteten som tillhandahålls i verktyg som görs tillgängliga tack vare WebKit. Tesen beskriver några av det överväldigande antal problem som stötts på under utvecklingens gång i detalj och reflekterar över eventuella metoder för att kringgå dem samt förklarar varför denna lösning, i många fall, kan ses som suboptimal.

PREFACE

This thesis was initiated by Spotify and conducted at the Chalmers Institute of Technology at the Department of Computer Science and the Division of Software Engineering and Technology. The thesis was conducted during April ‘09 to October ’09.

This thesis, except the implementation of the Cocoa, Awesomium and Qt version, is the result of a joint effort. Alexander was responsible for the implementation of the Qt version and Charles for the Cocoa and Awesomium versions.

We would like to give acknowledgement and thanks to our supervisor, Ludvig Strigeus, your help proved to be invaluable.

Table of Contents

1 INTRODUCTION

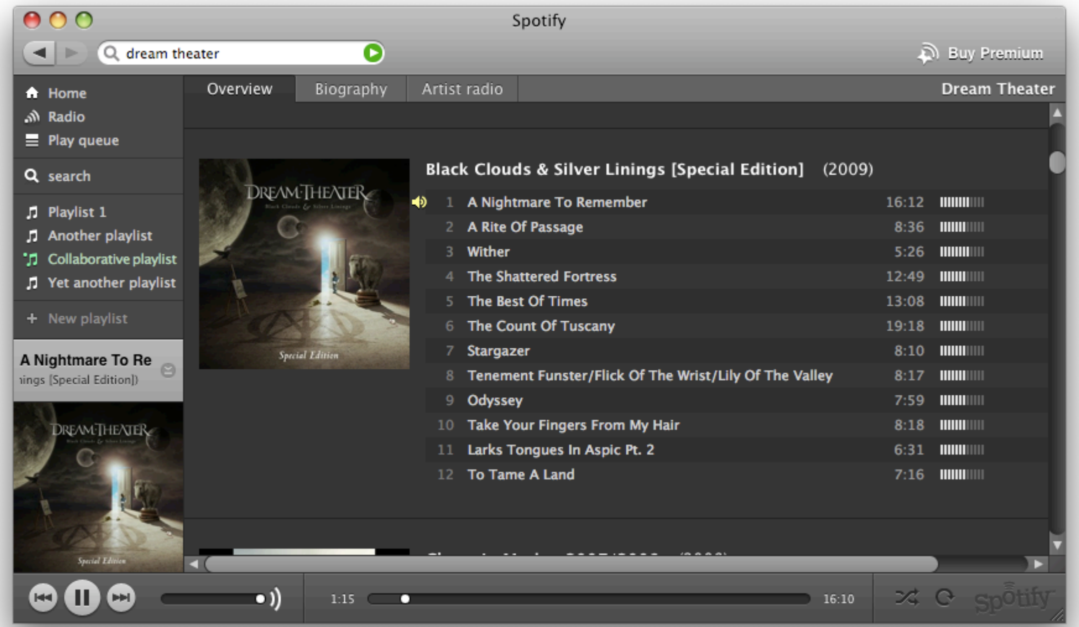
This chapter contains an overview of the thesis. It will introduce the background of the company and the reason for this thesis' existence, the Spotify client as well as the frameworks used and the thesis' purpose and delimitations.

1.1 Background

Spotify was founded in 2006 by Daniel Ek and Martin Lorentzon [1]. Spotify provides an online service that offers streaming music for free, financed by advertisements, or paid by a monthly subscription. The Spotify headquarters are located in London and the Research and Development department is situated in Stockholm.

1.1.1 The Client

The Spotify client is a cross-platform application providing the user with a graphical user interface (gui), see Figure 1-1, allowing the user to search the extensive library of music and create playlists. A radio that plays music from genres that the user chooses is also provided. The free version will occasionally play ads between songs or display banner ads.



Black Clouds & Silver Linings [Special Edition] (2009)

1	A Nightmare To Remember	16:12
2	A Rite Of Passage	8:36
3	Wither	5:26
4	The Shattered Fortress	12:49
5	The Best Of Times	13:08
6	The Count Of Tuscany	19:18
7	Stargazer	8:10
8	Tenement Funster/Flick Of The Wrist/Lily Of The Valley	8:17
9	Odyssey	7:59
10	Take Your Fingers From My Hair	8:18
11	Larks Tongues In Aspic Pt. 2	6:31
12	To Tame A Land	7:16

Figure 1-1: The Spotify client

1.1.2 Safari/WebKit

WebKit [2] is an open-source framework that provides a foundation on which to build a web browser. WebKit originated as a fork of KHTML, an open source HTML renderer for KDE, made by Apple in order to create a port for Mac. It is used as an HTML rendering engine for Safari on Windows, Mac OS X and iPhone OS. Some components of the framework are proprietary and will only be available if Safari is installed. The components that are licensed under LGPL are the ones of interest to this thesis. WebKit has been adopted by a number of projects, one of the most relevant to this thesis being Google Chrome.

1.2 The Task

The goal of this thesis is to redesign the Spotify client to use WebKit to render one or more of its views, for Windows and Mac OS X. This task presents challenges, many pertaining to the need for Safari to be present on a system for WebKit to work.

On Mac OS X this problem is non-existent since Safari is shipped with Mac OS X. Due to Safari libraries being proprietary [3], and Safari not being shipped with Windows, one needs to either implement own libraries or use an open library. This thesis will deal with two frameworks that incorporate WebKit on Windows that do not require the Apple libraries. These frameworks are Awesomium and Qt.

1.3 Purpose

The graphical views of the Spotify client are currently programmatically defined. If a view requires alteration Spotify needs to build and distribute a new version of their client.

With the help of HTML, CSS and JavaScript, this issue could be mitigated. The purpose of this thesis is to determine the viability of using WebKit as a graphical user interface renderer and whether it will be able to match the behavior of the Spotify client or not.

1.4 Delimitations

The goal of this thesis is to provide a proof of concept, or argue for why the task cannot be fulfilled.

2 Analysis and Method Description

This section describes some of the prerequisites that apply to this thesis and important issues that will be addressed.

2.1 Program requirements

Several issues are to be considered during the development of this thesis, among them are:

License Issues – What impact the license of the framework used will have on the structure of the program; will there be a need to extend the current licenses? Does the license of the framework enforce separate installation?

Size – The current application size is approximately 2.5 Megabytes, a property Spotify treasures and wishes to maintain. The low memory and processor consumption are also features that are important to conserve.

Uniformity of the two versions – Two applications are harder to manage than one. The system-specific part of each implementation should be kept as small as possible.

Dependant on external code – The more external code needed to run the application the more space will be required both during execution and on the hard disk drive.

Client/server responsibilities – The current implementation is client-heavy, leaving the server to perform lookups and manage data transfers. Spotify's wish is to maintain this separation of concerns. What effect will the inherent behavior of a web browser have on the client/server separation of workload?

Flash – The Spotify client occasionally displays Flash ads. To display Flash animations in a browser you need to install extra software. Can this be avoided? At what cost?

2.2 Method description

Since WebKit is included with Mac OS X the first step is to implement a small browser, in Mac OS X, to gain a basic understanding of WebKit and how one can use it.

2.2.1 Code Analysis

After exploring WebKit the thesis will move on to examine the implementation of the Spotify client in hopes of gaining a basic understanding of the application and determine the best “entry-point” for the WebKit component.

2.2.2 Mac implementation

The initial goal is to display a webpage within the Spotify client, using WebKit. This phase will hopefully reveal potential obstacles that can occur when embedding WebKit in a large application. If and when the initial goal is reached the next step will be to customize the webpage to mimic a user interface. This will most likely require venturing into the world of HTML, CSS and JavaScript.

2.2.3 Windows Implementation

WebKit is not bundled with Windows. Therefore this phase will require some research into what frameworks are available. If several viable alternatives are found they will be evaluated to such a point that one framework is declared the most appropriate. The intention is to make the implementation as similar as possible, yet complying with the specification, to the Mac version and vice versa.

2.2.4 Comparison

The results will finally be compared to determine potential benefits and gains of each version, such as differences in performance, similarity between the two versions, interface response or the amount of overhead added by using WebKit.

3 SOFTWARE AND FRAMEWORKS

This chapter describes the frameworks that were considered.

3.1 LGPL and program structure

The Gnu Lesser General Public License, LGPL [4], is a free software license published by the Free Software Foundation. It is, as the name suggests, an extension of the General Public License (GPL) [5].

LGPL defines a few important terms:

- Library – “a covered work governed by this License, other than an Application or a Combined Work”

- Application – “Any work that makes use of an interface provided by the library, but which is not otherwise based on the Library.”

- Combined Work – “a work produced by combining or linking an application with the Library. The particular version of the Library with which the Combined Work was made is also called the ‘Linked Version’.”

- Minimal corresponding source – “the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the linked version”. Simply put this means all code that interacts with the library.

- Corresponding Application Code – “the object code and/or source code for the application, including any data and utility programs needed for reproducing the combined work from the application, but excluding the system libraries of the combined work. ”

An important difference between LGPL and GPL is the restrictions they impose. A GPL licensed library may only be used in free software, under the terms of the GPL license, while an LGPL licensed library may be used in proprietary software.

The LGPL allows you to modify a copy of a library, provided that you do not break the functionality of the library, but you are bound to make that version available under either the LGPL or the GPL.

If you want to use an LGPL library in your application and thereby produce a Combined Work, regardless of what license your application is using, you must fulfill several requirements [4]:

- Give prominent notice with every copy of the combined work that the library is used in it and that the library and its use are covered by the LGPL.
- Accompany the combined work with a GPL and LGPL license.
- Display the LGPL copyright notice with notices displayed by the application during runtime.
    - Provide the Minimal Corresponding Source under the LGPL and the Corresponding Application Code in a way that a user can recombine or relink the application with a modified version of the Linked Version to produce a modified Combined Work.
      OR
    - Use a shared library mechanism for linking with the Library, i.e. loading the Library at run-time.
- Provide installation information depending on how you use the library in your application.

In this thesis the use of a shared library is of great interest. Little or no time will be spent investigating whether or not linking libraries statically is functional.

3.2 Windows Programming

This chapter provides basic information about details surrounding the development of the client on Windows.

3.2.1 Shared Libraries and DLL:s

There are two types of shared Libraries, static link libraries and dynamic link libraries [6]. Static link libraries are libraries where program and data addresses are bound to executables at link time, whilst dynamic link libraries are loaded at run time.

Dynamic linking is more flexible but a lot slower than static linking since the work that otherwise would have been performed at compile time is now performed at run time. If a system supports both static and dynamic linking of libraries, provided that programs do not need the extra flexibility of dynamic linking, the system will be faster and smaller with statically linked libraries.

There are several cons and pros with the two library versions. Static libraries are not easily editable and potential updates need to be made without moving any addresses in the library that the program depends upon. This permits minor updates, typically for small fixes.

A dynamic link library can be modified, recompiled and replaced as long as it fulfills the interface the application requires. This aspect of dynamic link libraries is a key feature to fulfilling the LGPL, which requires that the user should be able to replace any LGPL library in any application with another version that fulfills the same API and the application will still work.

Dynamic Link Libraries, DLL:s [7], are Microsoft's implementation of the shared library mechanism. A DLL is a module that contains functions and data that can be used by applications or libraries. The DLL contains executable code that is compiled, linked and stored separately from the processes that use them. Multiple processes can access the same representation of a DLL in the memory at the same time, reducing memory overhead, swapping and saving disk space.

3.2.2 The Windows API and Windows

The Windows API [8] offers the ability to develop applications that work on all versions of Windows. This API is also referred to as Win32, but the name "Windows API" is more appropriate since the API supports 64-bit systems as well as 16-bit.

In the Windows API a HWND [9], or a Window handle, represents a window, which is a pointer to the actual window. The window contains information such as the current size, instructions on how to repaint or position the window on the desktop.

All windows have a processer, which handles messages sent to the window. Examples of such messages are resize or hide commands.

3.3 WebKit structure

WebKit consists of several components. The modules exposed in the API are the WebView, WebPage and WebFrame. In some frameworks, such as Cocoa, the responsibilities of the WebPage and the WebView are combined in the WebView object. In the following chapter 'WebView' refers to the WebView as a separate entity [10].

3.3.1 WebView, WebPage and WebFrames

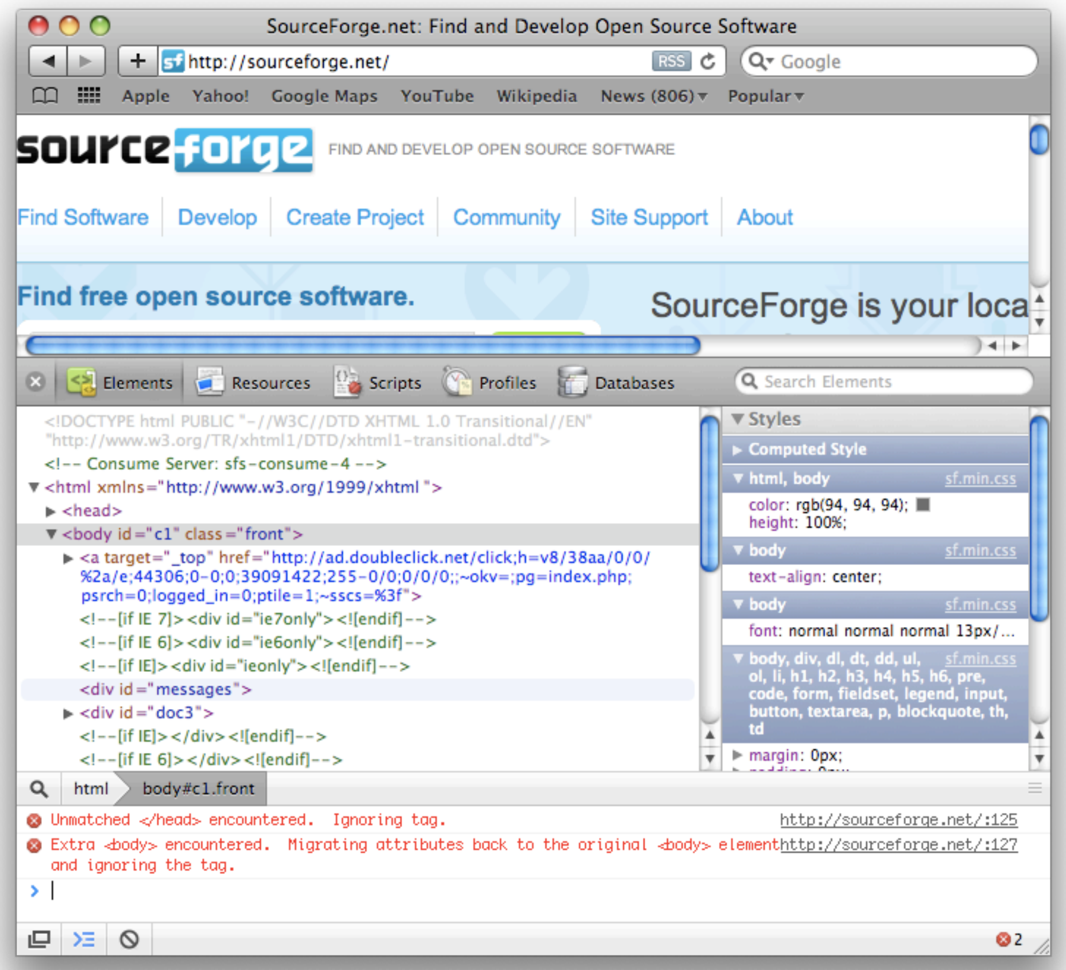
Each WebView has exactly one WebPage and each WebPage may have one or more WebFrames. The WebFrame corresponds to the frames defined in HTML. They can have children and a parent, so it is essentially a tree structure of frames. The root frame of this tree is called the main frame [11].

The WebPage represents a page and the actions one can perform on it such as opening a URL, executing a JavaScript or moving through the page history. The WebPage object has a reference to the mainframe object.

The WebView is responsible for a graphical interface of the WebPage. In some cases it may not be needed at all to use the WebView, for instance if one implements a tool to make thumbnails from webpages.

3.3.2 Web Inspector

The Web Inspector is a tool that lets you view page source, Document Object Model (DOM) hierarchy and debug scripts [12]. A resource analyzer is available, which monitors execution time spent on loading the web page and its resources. The Web Inspector is available in any WebKit-based client.



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- Consume Server: sfs-consume-4 -->
▼<html xmlns="http://www.w3.org/1999/xhtml">
  ▶<head>
  ▼<body id="c1" class="front">
    ▶<a target="_top" href="http://ad.doubleclick.net/click;h=v8/38aa/0/0/
     %2a/e;44306;0-0;0;39091422;255-0/0;0/0/0;;~okv=;pg=index.php;
     psrch=0;logged_in=0;ptile=1;~sscs=%3f">
     <!--[if IE 7]><div id="ie7only"><![endif]-->
     <!--[if IE 6]><div id="ie6only"><![endif]-->
     <!--[if IE]><div id="ieonly"><![endif]-->
     <div id="messages">
    ▶<div id="doc3">
     <!--[if IE]></div><![endif]-->
     <!--[if IE 6]></div><![endif]-->
```

```
html, body                                sf.min.css
  color: rgb(94, 94, 94);
  height: 100%;
body                                      sf.min.css
  text-align: center;
body                                      sf.min.css
  font: normal normal normal 13px/…
body, div, dl, dt, dd, ul,                sf.min.css
ol, li, h1, h2, h3, h4, h5, h6, pre,
code, form, fieldset, legend, input,
button, textarea, p, blockquote, th,
td
  ▶ margin: 0px;
```

```
Unmatched </head> encountered.  Ignoring tag.                         http://sourceforge.net/:125
Extra <body> encountered.  Migrating attributes back to the original <body> elementhttp://sourceforge.net/:127
and ignoring the tag.
> |
```

Figure 3-1 The Web Inspector in action.

3.4 The Spotify client interface

The Spotify client has its own GUI toolkit where each graphical component is defined as a C++ class. In order to add new graphical components one has to implement one such class.

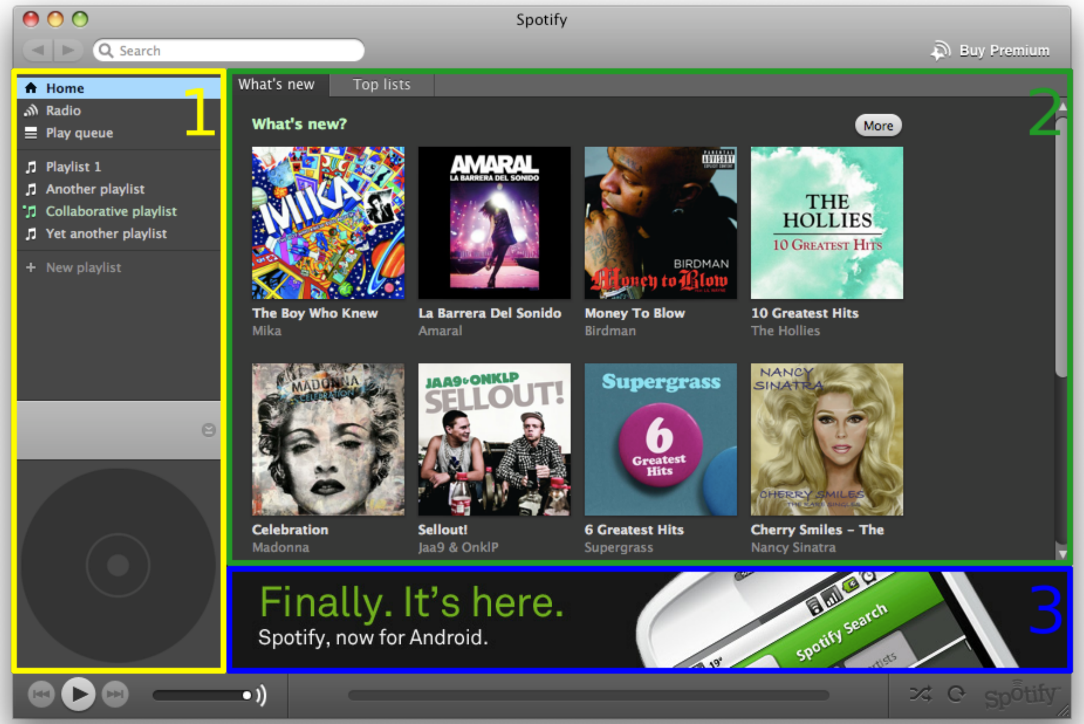


Figure 3-2: The Spotify client home view

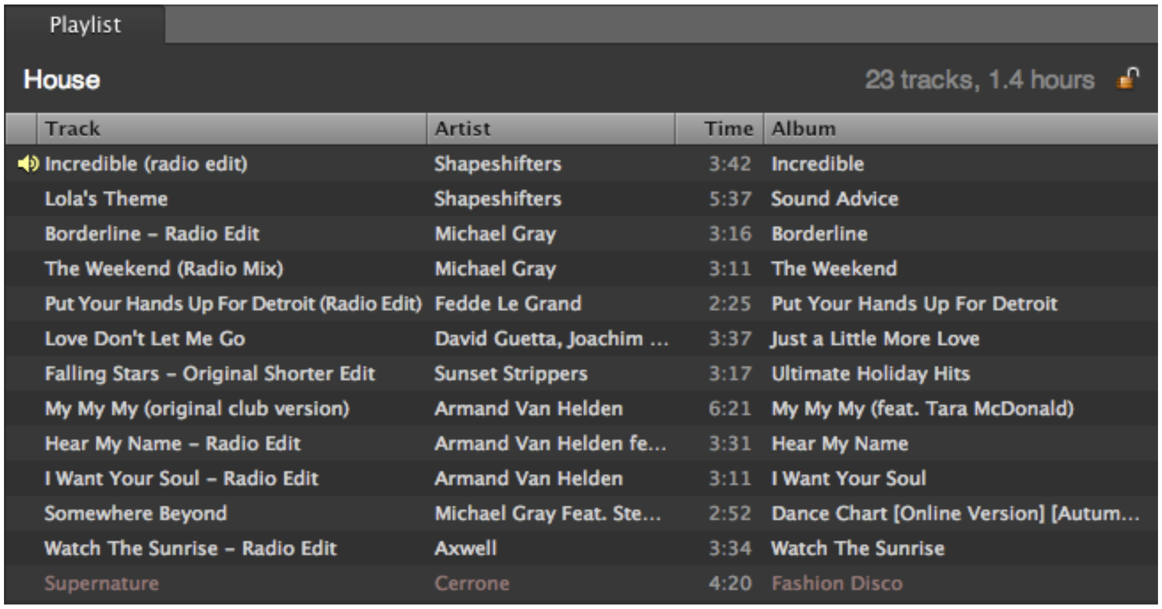
Three important areas are outlined in Figure 3-2:

1. The Side bar. Contains playlist links, a play queue link, a radio view link and a home view link. The bottom of the sidebar displays the currently played track and the album image of the album that the track belongs to.
2. The main view. Can display playlists, the play queue, the album view, the home view, the artist view and the radio view.
3. Banner area. Displays commercials and Spotify announcements. Is rendered by Internet Explorer on the Windows client and by WebKit on Mac.

These areas will be described in greater detail in upcoming chapters, and will by referred to by these terms in the thesis.

3.4.1 Playlists and play queues

The playlist view, depicted in Figure 3-3, contains a list of tracks belonging to the playlist currently browsed. If the playlist is being played a loudspeaker is displayed next to the song currently played.


Playlist

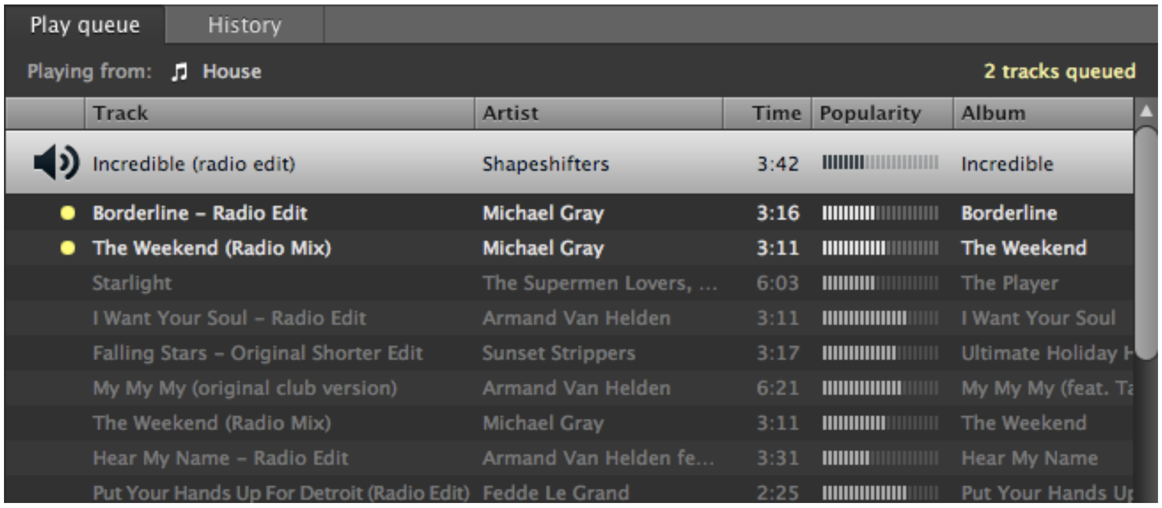
House

23 tracks, 1.4 hours

	Track	Artist	Time	Album
🔊	Incredible (radio edit)	Shapeshifters	3:42	Incredible
	Lola's Theme	Shapeshifters	5:37	Sound Advice
	Borderline – Radio Edit	Michael Gray	3:16	Borderline
	The Weekend (Radio Mix)	Michael Gray	3:11	The Weekend
	Put Your Hands Up For Detroit (Radio Edit)	Fedde Le Grand	2:25	Put Your Hands Up For Detroit
	Love Don't Let Me Go	David Guetta, Joachim …	3:37	Just a Little More Love
	Falling Stars – Original Shorter Edit	Sunset Strippers	3:17	Ultimate Holiday Hits
	My My My (original club version)	Armand Van Helden	6:21	My My My (feat. Tara McDonald)
	Hear My Name – Radio Edit	Armand Van Helden fe…	3:31	Hear My Name
	I Want Your Soul – Radio Edit	Armand Van Helden	3:11	I Want Your Soul
	Somewhere Beyond	Michael Gray Feat. Ste…	2:52	Dance Chart [Online Version] [Autum…
	Watch The Sunrise – Radio Edit	Axwell	3:34	Watch The Sunrise
	Supernature	Cerrone	4:20	Fashion Disco

Figure 3-3: The playlist view

The play queue lists the tracks that will be played. These tracks may be queued tracks, indicated by a yellow dot on the left, as shown in Figure 3-4. The tracks that do not have a yellow dot on their left are not queued but are due to be played from the relevant context. The context may be an album view, artist view, radio or a playlist.


Play queue | History

Playing from: ♫ House

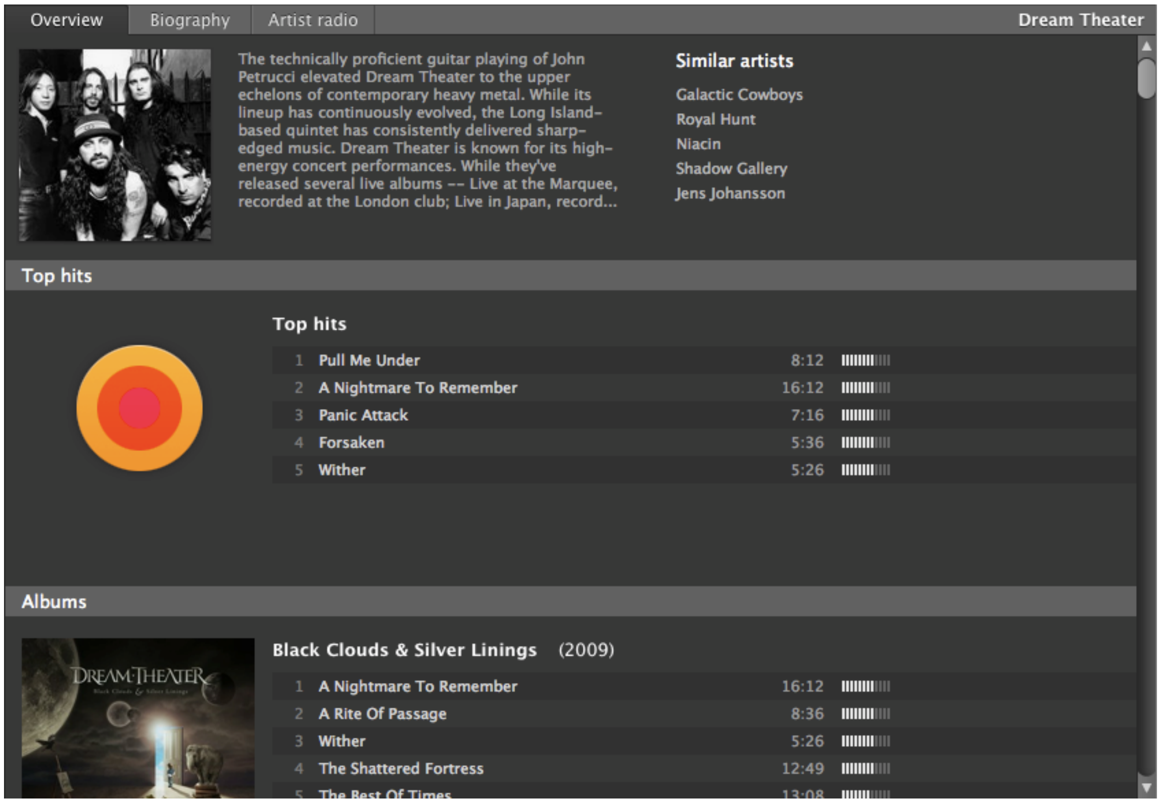
2 tracks queued

	Track	Artist	Time	Popularity	Album
🔊	Incredible (radio edit)	Shapeshifters	3:42		Incredible
●	Borderline – Radio Edit	Michael Gray	3:16		Borderline
●	The Weekend (Radio Mix)	Michael Gray	3:11		The Weekend
	Starlight	The Supermen Lovers, …	6:03		The Player
	I Want Your Soul – Radio Edit	Armand Van Helden	3:11		I Want Your Soul
	Falling Stars – Original Shorter Edit	Sunset Strippers	3:17		Ultimate Holiday H
	My My My (original club version)	Armand Van Helden	6:21		My My My (feat. Ta
	The Weekend (Radio Mix)	Michael Gray	3:11		The Weekend
	Hear My Name – Radio Edit	Armand Van Helden fe…	3:31		Hear My Name
	Put Your Hands Up For Detroit (Radio Edit)	Fedde Le Grand	2:25		Put Your Hands Up

Figure 3-4: The play queue

3.4.2 Artist and album views

The artist view, depicted in Figure 3-5, contains information about the artist, a list of similar artists and a top-five list of the artists most played songs on Spotify. It also lists all their albums with the cover art and track listing.



The technically proficient guitar playing of John Petrucci elevated Dream Theater to the upper echelons of contemporary heavy metal. While its lineup has continuously evolved, the Long Island-based quintet has consistently delivered sharp-edged music. Dream Theater is known for its high-energy concert performances. While they've released several live albums -- Live at the Marquee, recorded at the London club; Live in Japan, record...

Similar artists

Galactic Cowboys
Royal Hunt
Niacin
Shadow Gallery
Jens Johansson

Top hits

Top hits

1	Pull Me Under	8:12
2	A Nightmare To Remember	16:12
3	Panic Attack	7:16
4	Forsaken	5:36
5	Wither	5:26

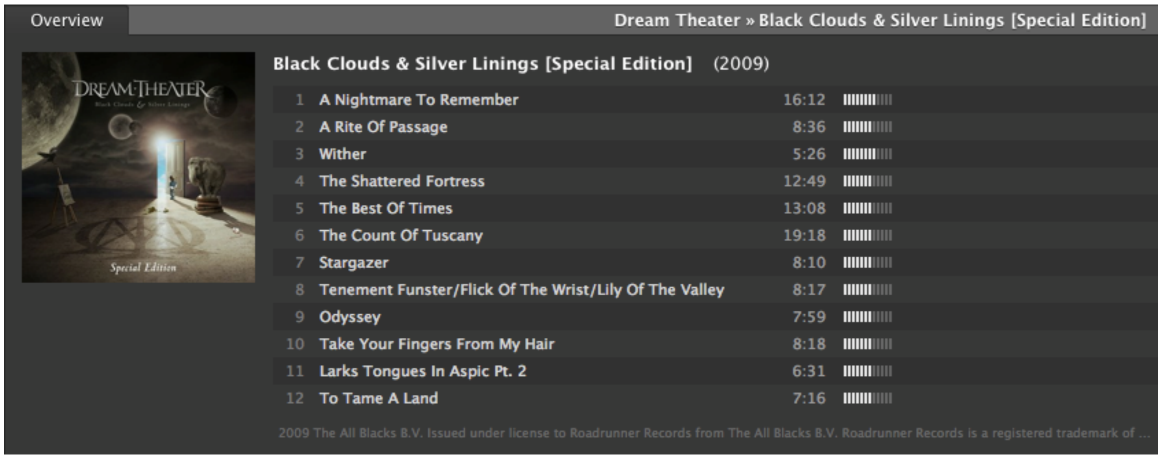
Albums

Black Clouds & Silver Linings (2009)

1	A Nightmare To Remember	16:12
2	A Rite Of Passage	8:36
3	Wither	5:26
4	The Shattered Fortress	12:49
5	The Best Of Times	13:08

Figure 3-5: The artist view

The album view, Figure 3-6, contains the cover art of the album and a track listing along with copyrights. It may contain a third-party review of the album if one is available.



Overview

Dream Theater » Black Clouds & Silver Linings [Special Edition]

Black Clouds & Silver Linings [Special Edition] (2009)

1	A Nightmare To Remember	16:12
2	A Rite Of Passage	8:36
3	Wither	5:26
4	The Shattered Fortress	12:49
5	The Best Of Times	13:08
6	The Count Of Tuscany	19:18
7	Stargazer	8:10
8	Tenement Funster/Flick Of The Wrist/Lily Of The Valley	8:17
9	Odyssey	7:59
10	Take Your Fingers From My Hair	8:18
11	Larks Tongues In Aspic Pt. 2	6:31
12	To Tame A Land	7:16

2009 The All Blacks B.V. Issued under license to Roadrunner Records from The All Blacks B.V. Roadrunner Records is a registered trademark of ...

Figure 3-6: The album view

3.4.3 Home and radio views

The home view, depicted in Figure 3-2, lists new albums and artists recommended for users. The selection of recommended artists is based on the listening history of the user.

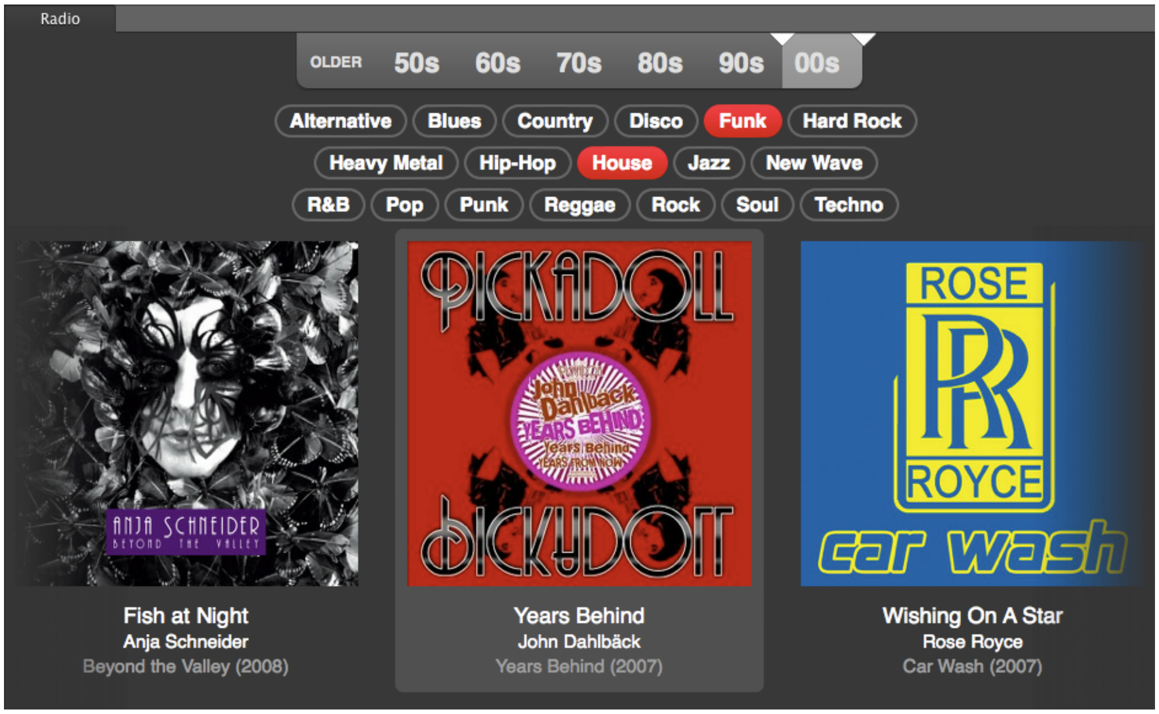


Figure 3-7: The radio view

The radio view, Figure 3-7, lets the user select genres and a span of years and will play tracks from the selected genres and era, scrolling the album images as it switches songs.

3.4.4 Color Correction

The International Color Consortium, the ICC, is an organization established in 1993 by eight major industry vendors with the purpose of creating standards for a cross-platform color management system. The ICC profile specification is the result of their efforts.

The ICC profile specification is accepted by several operating systems, giving end users the ability to transparently move profiles and images between operating systems. This assures that an image will retain color fidelity when transferred between operating systems and applications.

A profile contains information that is used to display the image in its intended form. The software displaying the image will give input, which depends on what device the image is to be displayed on, that is used to adjust the colors of the image to the desired levels [13]. sRGB, a format that supports the ICC profiling system, is used to represent the images in the Spotify client [14]. Figure 3-8 depicts a comparison of a color-profiled image and a non-color profiled image.

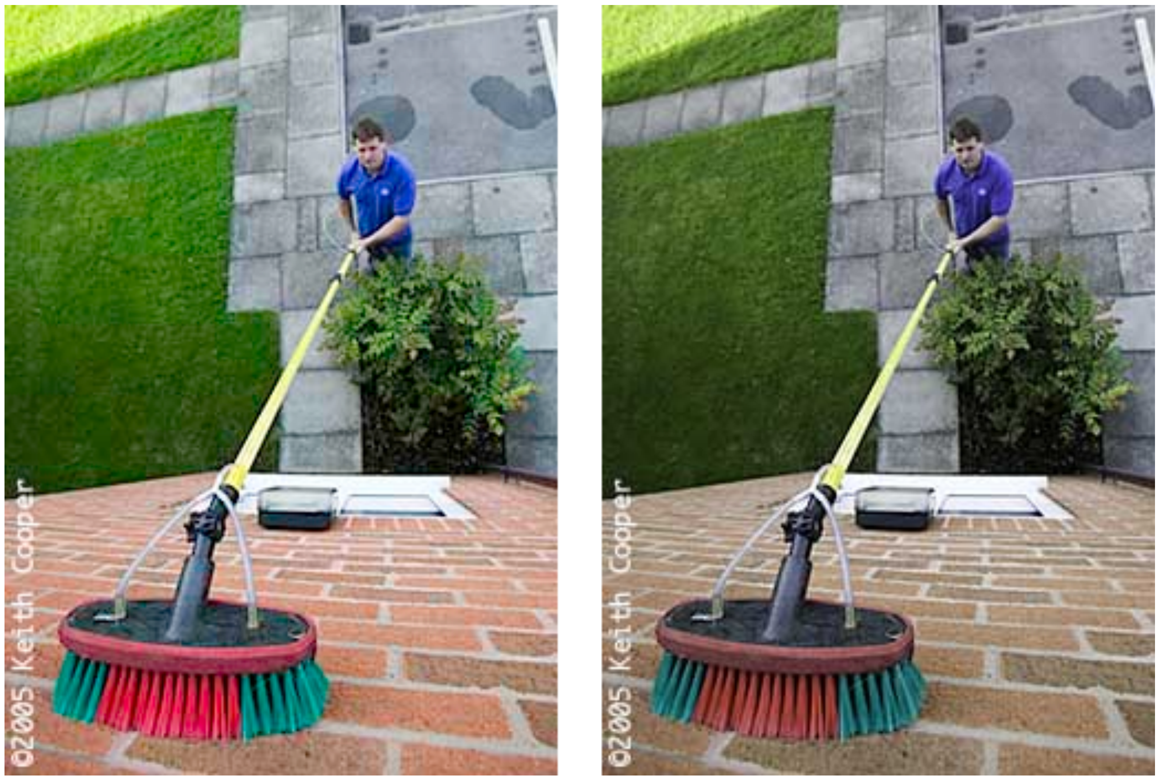


Figure 3-8: An image with color profiling compared to an image without color profiling images © Keith Cooper of Northlight Images

3.4.5 Client communication

The Spotify client uses a URI, called a Spotify link, for identifying specific resources in the system, for instance; ‘spotify:album:7khFXQNBzcfSfgGjPKerdE’ for a specific album, ‘spotify:track:6JEK0CvvjDjjMUBFoXShNZ’ for a specific track or ‘spotify:artist:2aaLAng2L2aWD2FClzwiep’ for a specific artist.

The client communicates with the Spotify servers through a secure connection. It is through this connection that all data is transmitted. This behavior should be maintained in any added functionality. The client to server communication can be defined as a set of requests. The requests relevant to this thesis are:

- Image – Request raw image data
- Album – Request information about an album
- Artist – Request information about an artist
- Top list – Request information about a user’s or region’s top played tracks or albums
- Search – Request information based on a query

3.5 Data URIs and base64 conversion

On web pages, external data is referenced with Uniform Resource Identifiers (URIs) which has the general form ‘scheme:locator’ where ‘scheme’ identifies which protocol should be used and what the ‘locator’ part will contain.

The ‘data’ scheme allows for resources to be referenced inline. It has the form ‘data:[<mediatype>][;base64],<data>‘ where ‘mediatype’ specifies a mime type of the contents. Base64 is specified if the contents are base 64 encoded [15].

A base 64 encoding transforms raw data into bytes that are in the printable spectrum of characters. In this format A-Z represents numbers 0-25, a-z represents 26-51, 0-9 represents 52-61, + and / represents 62 and 63 respectively. Encoding is done by taking groups of 3 bytes and converting 6 bits at a time to the base64 alphabet. This means that base64-encoded data has a 4/3-size ratio over raw data [16].

3.6 Nokia/Trolltech Qt

One of the frameworks considered is Qt, an open source framework for building gui- and command-line applications. Qt is available under an LGPL license as well as sold under a commercial license. The C++ version of Qt has been under development since 1995 [17] and supports several platforms, including Linux/Windows/Mac OS X. This chapter will explain parts of Qt that are essential to this thesis.

3.6.1 The QApplication, Widgets

All applications using any gui-component of Qt needs a QApplication, QApp, running [18]. Only one instance of the QApp exists at any given time. It manages the event loop and the application's control flow as well as several settings. All events from the window system and other sources are processed and dispatched by the QApp.

The QApp blocks the thread it is running in, releasing only when the QApp is terminated. The effect of this is noticeable especially if you want to embed Qt into an already existing application, where you will need to add a separate thread for all parts of Qt that needs to interact with the QApp.

Widgets are Qt's representation of graphical components such as buttons, menus or application windows [19].

3.6.2 Signals and Slots

Signals and slots are a form of event handling [20]. A QObject, or a class subclassing a QObject, emits a signal when an event occurs. What signal is to be emitted upon an event is based on the implementation. Slots are normal methods with the "slot-quality" assigned to them, which makes it possible to connect them to a signal, and being ordinary methods they can be called by anyone with the proper access.

When a signal is emitted all slots (or methods) connected to it will be executed. A slot can be connected to several signals and a signal can be connected to several slots. Figure 3-9 illustrates a sample execution case where Object A emits Signal A, which has Slot B connected to it, and Signal B has slot C connected to it. When Signal A is emitted both Slot B and C will be executed.

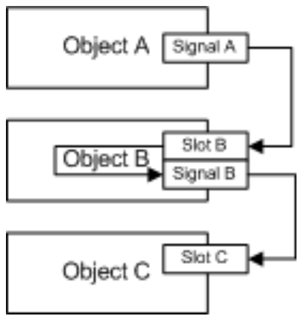


Figure 3-9: Illustration of a possible slot-execution case

3.6.3 Hit result resolution

Presuming that a click has been performed inside the WebView, one can obtain certain information to determine the proper response. The information that can be retrieved is the exact position that was clicked, the 'title' attribute of the HTML element clicked and the URL of the HTML link clicked [21].

3.6.4 Event handling

In Qt, events are delivered to QObjects so that they can respond to them. Programmers have the option to monitor and filter all events at the application and object-level [22]. It is also possible to re-implement the handling of events to obtain a specific behavior when an event occurs. One can also install event filters to monitor events before they reach their intended destination.

Most events are generated by the window system and inform the program of user-actions such as key presses or mouse clicks. The events contain information relevant to the event such as the exact position clicked.

3.6.5 Meta Object System

The main reason for using the meta-object system in Qt is the signals and slots mechanism. The meta-object system [23] also provides other features such as dynamic casting and access to inheritance information. The system is based on three foundations:

- The QObject class provides a base class for objects that can use the meta-object system.
- The Q_OBJECT macro is used to enable meta-object features such as signals and slots.
- The Meta-Object Compiler (moc) supplies QObject subclasses with code to implement meta-object features.

Qt has a moc tool that reads a C++ source file and searches for the Q_OBJECT macro in each class. If a class contains the macro, another C++ source file, which contains the meta-object code for that class, will be generated. The generated source is either included into the class or compiled and linked with the implementation.

3.6.6 Qt Migration Framework

The Qt/MFC Migration Framework [24] (Qt MF) provides classes that make it possible to use Qt and Microsoft Foundation Class Library (MFC), or the Windows API (Win32), in the same application. It allows for Win32/MFC interface items to be embedded in Qt applications and vice versa. The Qt MF is a separate framework released under a commercial license and under LGPL.

The essential component, for this thesis, of the framework is the QWinWidget [25]. The QWinWidget is a class that encapsulates a Qt Widget in a Window (in the Windows API sense) and alleviates the integration of Qt into a Win32 application. If the QWinWidget was not used, one would have to write a few hundred lines of code, which the writing of would require a deep understanding of the Windows and Qt API.

3.6.7 Qt WebKit

The WebKit module of Qt [27] uses the same Open-Source version of WebKit as is used on Mac. This module provides an acid3 [26] compliant web-browser engine that facilitates embedding web-content into a Qt application. It supports HTML, XHTML, Scalable Vector Graphics documents, Cascading style sheets and scripting with JavaScript. Qt WebKit also includes the Web Inspector.

3.7 Google Chromium

Chromium [28] is a Google initiated open source project distributed under a multitude of licenses. Its aim is to create a lightweight tabbed browser with a separate process for each tab. The project is built upon the WebKit framework. Google bases their Chrome browser on Chromium.

3.7.1 License issues

The WebKit components of Chromium are licensed under LGPL. Other elements of the browser however, are licensed under several other licenses such as the BSD or MIT license [29]. The graphic libraries and WebKit complementary libraries, such as Skia, the Google 2D graphics library, are licensed under the Apache 2.0 license.

3.8 Awesomium

Awesomium is an open source project based on Google Chromium by Adam J. Simmons, also known as the prince of code [30]. The project extracts the rendering components from Chromium. Awesomium is an HTML renderer that can be embedded into any application without the need of any proprietary libraries.

4 Mac Implementation

This chapter describes the Mac client. It will begin by describing the user interface, followed by a more programmatically focused description.

WebKit is available in Cocoa, a framework for Mac, through an Objective-C interface. Any installation of Mac OS X has the Cocoa framework by default. WebKit supports the file and HTTP protocol, neither of which will contribute to fulfilling the requirement that data shall travel through the Spotify client and its connections to the Spotify servers. Luckily, the Cocoa framework allows for definition of custom protocols.

4.1 Look and feel

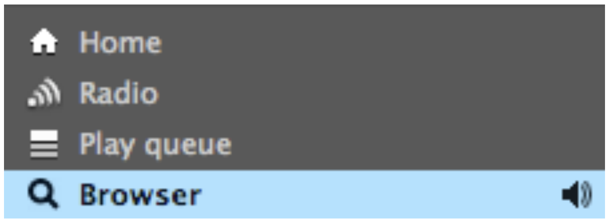
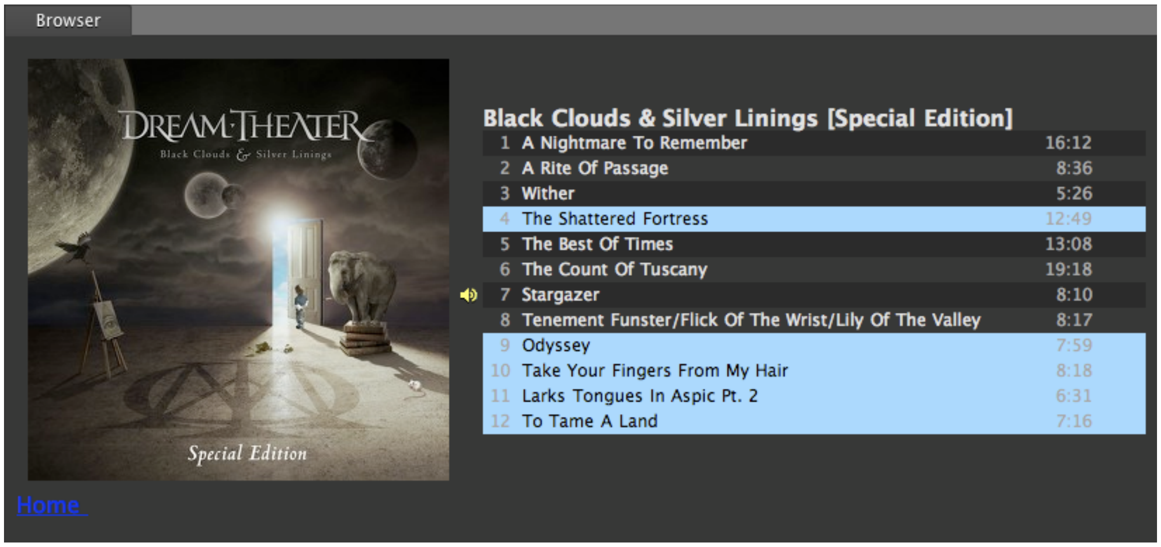


Figure 4-1: The added option 'Browser' on the sidebar

Figure 4-1 shows an image of the 'Browser' option added on the side bar. Selecting this option displays the WebView, containing an album view per default, in the main view. From the album view one can navigate to a WebKit rendered implementation of the home view.

Both the album and home view are constructed with HTML, CSS and JavaScript. It is required that these three components are all specified inline on the same page. External links are not supported unless over HTTP.

4.1.1 Album view



Black Clouds & Silver Linings [Special Edition]

1	A Nightmare To Remember	16:12
2	A Rite Of Passage	8:36
3	Wither	5:26
4	The Shattered Fortress	12:49
5	The Best Of Times	13:08
6	The Count Of Tuscany	19:18
7	Stargazer	8:10
8	Tenement Funster/Flick Of The Wrist/Lily Of The Valley	8:17
9	Odyssey	7:59
10	Take Your Fingers From My Hair	8:18
11	Larks Tongues In Aspic Pt. 2	6:31
12	To Tame A Land	7:16

Figure 4-2: The Spotify client on Mac using WebKit to render the album view

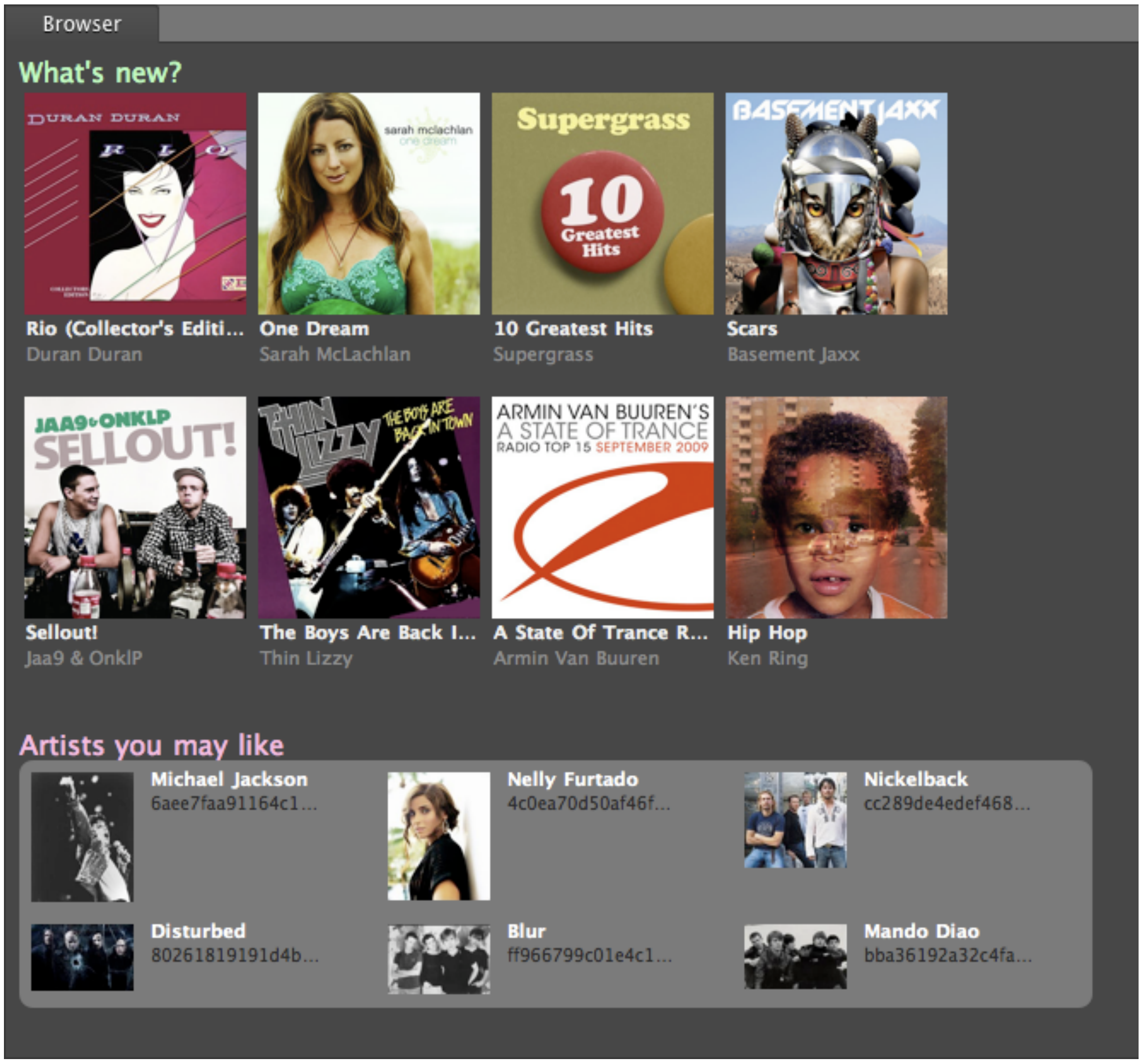
The Album view (shown in Figure 4-2) shows the album cover, if one exists and a listing of all the tracks of the album. The rows in the track listing may be selected, indicated by blue background, either by clicking on them or using the arrow buttons.

As in the Spotify client the tracks in the track listing as well as the cover image are draggable if they are clicked and dragged. It is possible to drag them anywhere inside of Spotify or even outside of it.

The same context menu as is used in the Spotify client is displayed upon a right click event.

The small loudspeaker next to the name of a track in the track list identifies that track as being currently played.

4.1.2 Home View



What's new?

Rio (Collector's Editi...	Duran Duran
One Dream	Sarah McLachlan
10 Greatest Hits	Supergrass
Scars	Basement Jaxx
Sellout!	Jaa9 & OnklP
The Boys Are Back I...	Thin Lizzy
A State Of Trance R...	Armin Van Buuren
Hip Hop	Ken Ring

Artists you may like

Michael Jackson	6aee7faa91164c1...
Nelly Furtado	4c0ea70d50af46f...
Nickelback	cc289de4edef468...
Disturbed	80261819191d4b...
Blur	ff966799c01e4c1...
Mando Diao	bba36192a32c4fa...

Figure 4-3: The home view as rendered by WebKit on Mac

The home view, Figure 4-3, is a representation of the home view in the Spotify client. It consists of 8 cover arts at the top. Each of these cover arts are a link to the album to which the cover art belongs, and are accompanied by a text-represented link to the same album. Pressing any of the links will show the album view of the album clicked. Below the albums there are 6 recommended artists. It is possible to perform drag-and-drop and open a context menu with the album links and album covers as sources/targets.

4.2 Implementation

This chapter explains, in detail, how the Look and feel features are implemented.

4.2.1 Image loading

The images are loaded in one of two ways, decided at compile time. The first way is by an implementation of a new protocol scheme, called ‘spot’. Whenever WebKit encounters a resource with the ‘spot’ scheme, WebKit will perform actions as they are defined in the protocol scheme implementation, querying the Spotify servers and replying with data once it gets a reply from the Spotify servers.

The second way is by a JavaScript that runs once when a new page is loaded. This script ensures that all image elements have a unique id, provided that the image does not already have an id. If the element’s source tag has the spot scheme a Spotify request for that id is performed and the image is set to the default image for that image type, ensuring that all images will at least have the default image set if there is no image for a certain album/artist on the server. This process requires that all images have a type associated with them and is illustrated in Figure 4-4.

The locator to specify in the ‘src’ attribute of the ‘img’ tag should have the format ‘spot:[hex string of the image id]’ e.g. spot:3e6a44ac2f9ace0f98505a86303d723312bacd46.

When the client receives an image result the image is converted to base64 and inserted into a data URI. The URI is then inserted into the ‘src’ attribute of the corresponding image.

WebKit automatically resizes the images based on their HTML and CSS properties. This resize is done smoothly with no jagged edges. Figure 4-4 shows how an incorrectly resized image would look and how WebKit resizes that same image.

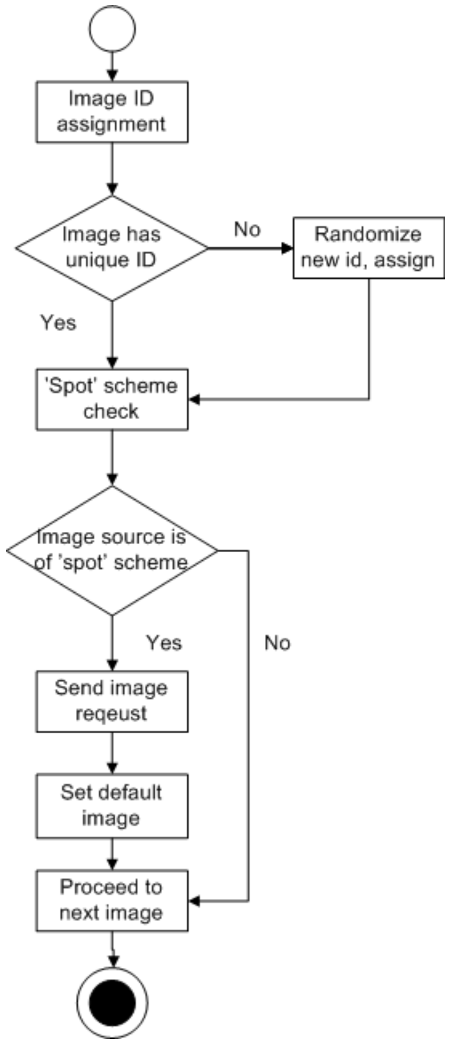


Figure 4-4: Flowchart describing the imageHandling JavaScript

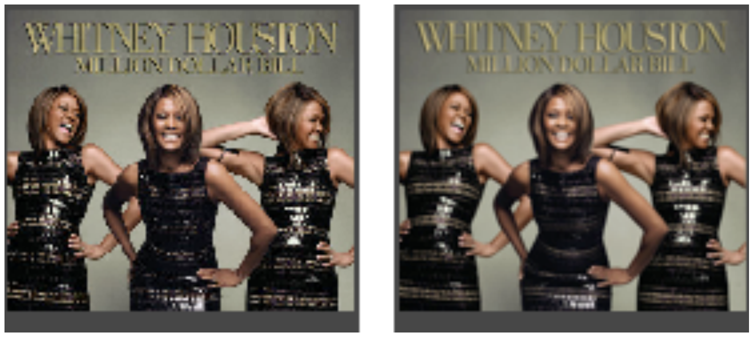
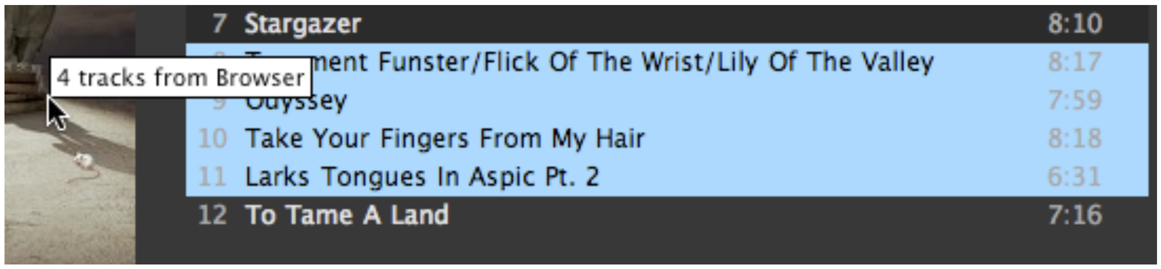


Figure 4-4: Naive image resize versus WebKit's resize

4.2.2 Context menus and drag-and-drop

Any HTML element that has the tag attribute ‘title’ set to a Spotify link is considered a valid target for context menus. If the title is a link to a track then it may also be selected. If a selected track should be the target of a context menu, then the context menu will use all selected tracks as input for any action that takes tracks as input.



7	Stargazer	8:10
8	Tenement Funster/Flick Of The Wrist/Lily Of The Valley	8:17
9	Odyssey	7:59
10	Take Your Fingers From My Hair	8:18
11	Larks Tongues In Aspic Pt. 2	6:31
12	To Tame A Land	7:16

Figure 4-5: Drag and drop in the Mac version

Drag and drop works in the same manner as the context menu. Any element with a Spotify link is a valid drag and drop object. The label that is shown is handled by the Spotify client and gets its text from the dragged objects, illustrated in Figure 4-5.

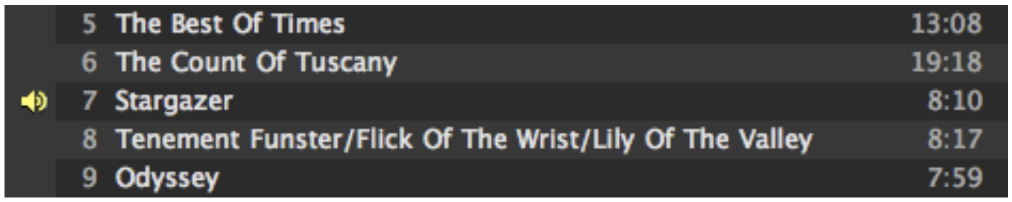
4.2.3 Selection and currently played track

Three CSS classes are defined to handle the graphical alterations when the user selects a track, ‘selected’ and ‘row0’ ‘row1’. ‘row0’ and ‘row1’ handle the standard color when a track is not marked while ‘selected’ is the blue color that Spotify uses for selected tracks.

Changing the CSS class of the selected element to ‘selected’ when it is selected and then reverting it to its old class once it is deselected performs graphical selection.

The keyboard up and down buttons can be used to select tracks in the track list. If the shift key is pressed a span of tracks will be selected. The Cocoa framework seems to lack support for listening to all key presses in the WebView. The workaround used is adding a JavaScript keyboard event listener to the body tag that invokes an Objective-C method.

The mouse events are similar; by default the only two events you can intercept are when the left or right mouse button is released. So a similar workaround as for the keyboard is required if more advanced functionality is desired.



	5	The Best Of Times	13:08
	6	The Count Of Tuscany	19:18
🔊	7	Stargazer	8:10
	8	Tenement Funster/Flick Of The Wrist/Lily Of The Valley	8:17
	9	Odyssey	7:59

Figure 4-6: Currently playing track represented by loudspeaker image

The client shows which track is currently playing by calling a JavaScript function on the page whenever a change of track or play status occurs. The page itself carries the actual responsibility of displaying and hiding the graphical representation of the currently played track. In the album view the graphical representation is a small loudspeaker, the same as in the Spotify client, next to the row of the song that is currently playing, see Figure 4-6. This loudspeaker is stored on the page as a JavaScript string that contains the image data for the loudspeaker as a base64 converted URI.

4.2.4 Web Inspector

If the client is run in debug mode and alt + right click is pressed the original WebKit context menu is shown. On this menu there is an option called inspect element that will display the Web Inspector.

5 WINDOWS IMPLEMENTATION

This chapter deals with the different versions of the Windows client implemented in this thesis. It will begin by evaluating Chromium, followed by the result of the Awesomium version and describe the result of the Qt version.

5.1 Chromium

The initial goal of this thesis was to extract the WebKit module along with the required libraries from Google Chromium into a separate project that could be embedded into the Spotify client.

The components of Chromium were very tightly coupled, and in order for an extraction to be possible new versions of many of the components would have to be composed.

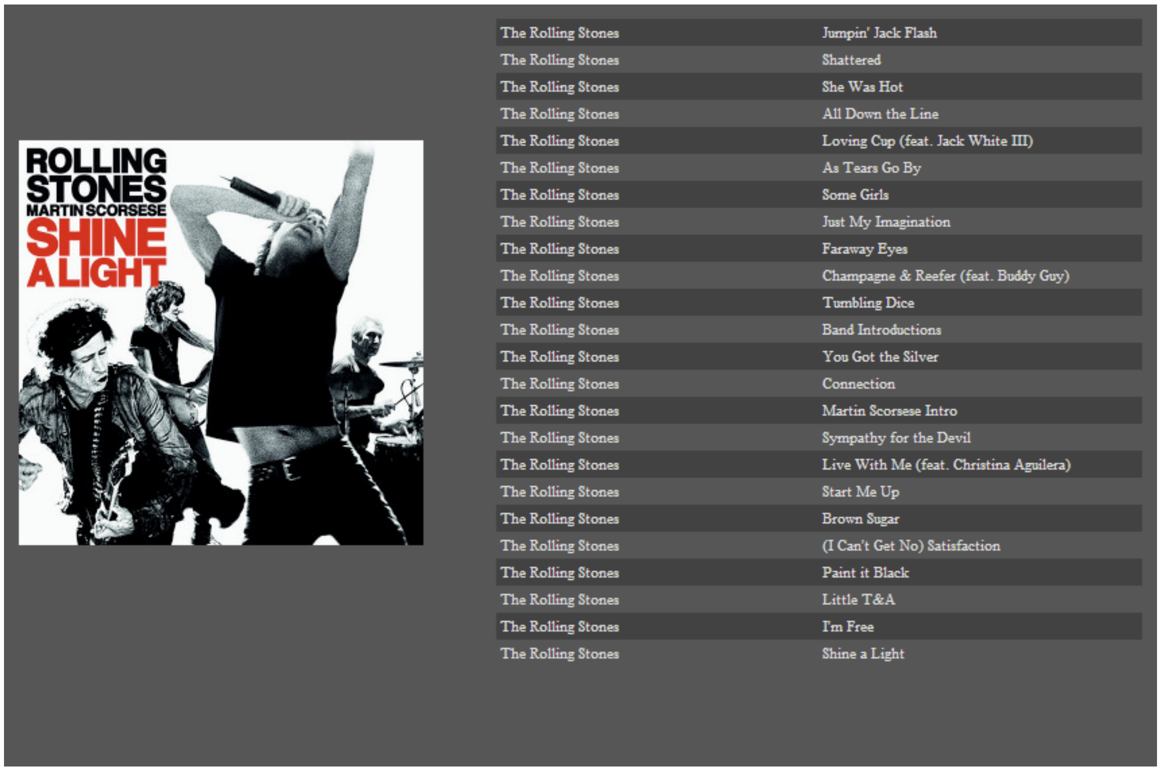
Due to the seemingly massive amount of work that was in store for anyone attempting this approach, the Chromium aspect of this thesis was put on hold indefinitely.

However, the concept was still viable, especially if the arduous and complicated process of extraction from Chromium was performed and delivered as a framework. Enter Awesomium.

5.2 Awesomium

Awesomium is a package that has extracted essential components from Chromium in order to create a framework [30], using WebKit, to provide means to create your own HTML renderer. This chapter describes the state of the Awesomium version.

5.2.1 Look and Feel



The Rolling Stones	Jumpin' Jack Flash
The Rolling Stones	Shattered
The Rolling Stones	She Was Hot
The Rolling Stones	All Down the Line
The Rolling Stones	Loving Cup (feat. Jack White III)
The Rolling Stones	As Tears Go By
The Rolling Stones	Some Girls
The Rolling Stones	Just My Imagination
The Rolling Stones	Faraway Eyes
The Rolling Stones	Champagne & Reefer (feat. Buddy Guy)
The Rolling Stones	Tumbling Dice
The Rolling Stones	Band Introductions
The Rolling Stones	You Got the Silver
The Rolling Stones	Connection
The Rolling Stones	Martin Scorsese Intro
The Rolling Stones	Sympathy for the Devil
The Rolling Stones	Live With Me (feat. Christina Aguilera)
The Rolling Stones	Start Me Up
The Rolling Stones	Brown Sugar
The Rolling Stones	(I Can't Get No) Satisfaction
The Rolling Stones	Paint it Black
The Rolling Stones	Little T&A
The Rolling Stones	I'm Free
The Rolling Stones	Shine a Light

Figure 5-1: The Album view as rendered by Awesomium

Figure 5-1 shows the result of the Awesomium implementation. Similar to the Mac version, selecting the Browser option in the side bar displays the Awesomium view. The only view implemented with Awesomium is the album view.

To the left is the album cover and to the right is a track list. Drag-and-drop and context menus are not implemented. Neither is selection or currently playing.

5.2.2 Implementation

This chapter will describe aspects of the implementation of the Awesomium version of the Spotify client.

5.2.2.1 WebCore and initialization

When an Awesomium browser view is created it starts by making sure that the WebCore is initialized. This initialization also includes starting a timer that runs the update method on the WebCore object every 200 milliseconds. This method must be called in order for WebViewListeners and JavaScript callbacks to work [31].

Every Awesomium browser instance has a window in which it draws its rendered data. When this window receives messages from the main message loop it injects the keyboard and mouse messages into the Awesomium WebView and lets it handle them.

5.2.2.2 Image loading

Images are loaded using a JavaScript that first makes sure that all image elements has an id attribute, it then requests a raw image through the Spotify connection. When that image arrives it is converted to a base64 string, which is then set to the image source attribute using a data URI.

5.3 Trolltech Qt

This section describes the setup, use and result of using the Qt and Qt WinMigrate Frameworks.

5.3.1 Compiling and Visual Studio Setup

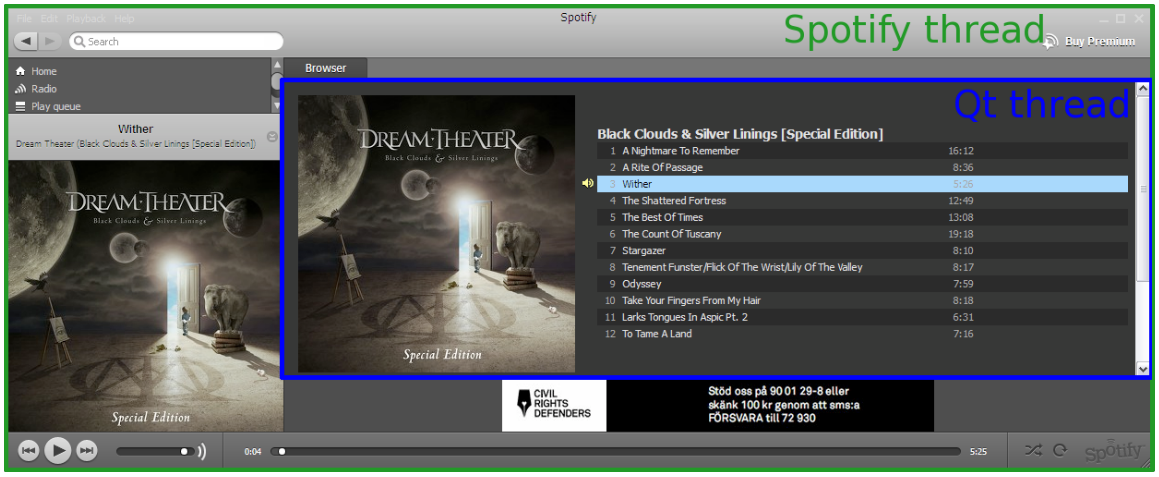
A Makefile is a set of instructions that tells make-equivalent software how to compile and link software [32]. An example of make software is GNU make. It operates by reading a Makefile and processes all rules/commands in it, compiling all files that are necessary. Makefiles can also contain instructions that affect the file system such as file removal or copying.

Qmake is a tool provided by Trolltech that facilitates the project build process across platforms [33]. Qmake automates the generation of Makefiles, minimizing the information needed to create a Makefile. Qmake generates a Makefile based on a project file and can generate projects for Microsoft Visual Studio (Visual Studio) without changing the project file. For this thesis, Qt was compiled with the aid of Qmake using Visual Studio and project-mode flags.

Trolltech provides a Visual Studio extension that allows users to open Qt projects in Visual Studio. The extension is not a must have, since one can recompile the Qt source with the help of Qmake-generated Makefiles.

5.3.2 Threading and Event-passing

All graphical user interface applications based on Qt require an instance of the QApplication object to be running in order to be able to use signals and slots [18], which the Qt graphical interface is heavily dependant on. Unfortunately, a QApplication will lock its thread the moment it executes, not releasing the thread until it exits. Since the integration requires the use of a QApplication an extra thread is used to encapsulate all elements of Qt that use signals and slots, illustrated in Figure 5-2.


Spotify
Spotify thread
Buy Premium
Search
Home
Radio
Play queue
Wither
Dream Theater (Black Clouds & Silver Linings [Special Edition])
Browser
Qt thread
Black Clouds & Silver Linings [Special Edition]
CIVIL RIGHTS DEFENDERS
0:04
5:25

1	A Nightmare To Remember	16:12
2	A Rite Of Passage	8:36
3	Wither	5:26
4	The Shattered Fortress	12:49
5	The Best Of Times	13:08
6	The Count Of Tuscany	19:18
7	Stargazer	8:10
8	Tenement Funster/Flick Of The Wrist/Lily Of The Valley	8:17
9	Odyssey	7:59
10	Take Your Fingers From My Hair	8:18
11	Larks Tongues In Aspic Pt. 2	6:31
12	To Tame A Land	7:16

Stöd oss på 90 01 29-8 eller
skänk 100 kr genom att sms:a
FÖRSVARA till 72 930

Figure 5-2: Visual representation of graphical element separation in threads

Adding an extra thread imposes new restrictions. The first issue tackled was the communication between the two threads, illustrated in Figure 5-3. Communication from the Win32-thread to the Qt-thread is performed with Qt-events [34] that are sent to, and dispatched by, the QApplication. Communication from the Qt-thread to the Win32-thread is performed with Win32 message posting [35].

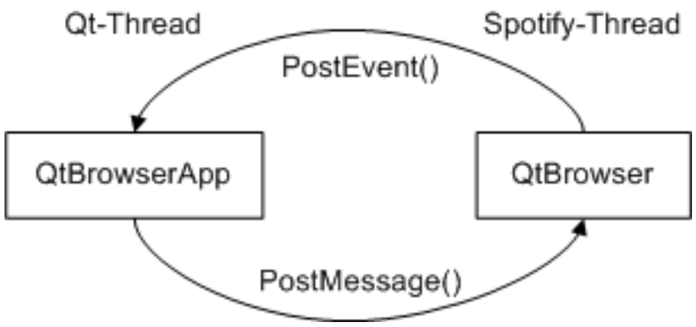


Figure 5-3: Cross-thread communication

Due to the need for a plethora of events and message identifiers, a namespace was created containing all constants and datastructures related to thread-communication.

5.3.3 Window and Widget structure

The QWinWidget is the child of a window created solely for the purpose of containing the QWinWidget. That window is the child of the window the Spotify client uses to display the playlist or radio view. The QWinWidget resides in the Qt-thread together with the QWebView, which is a child of the QWinWidget, while the parent window of the WinWidget resides in the Spotify-thread, as illustrated in Figure 5-4. The QWinWidget also maintains a Window of its own.

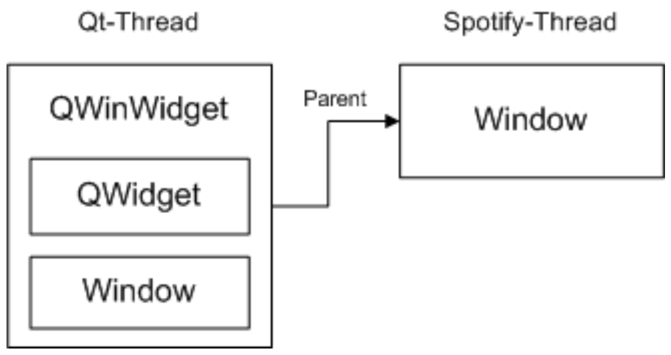


Figure 5-4: Window structure across threads

This composition brings some restrictions with it. The window processer of the WinWidgets parent is in the Spotify-thread and therefore has to send Qt-events, across the threads, when it receives a message that affects its children.

5.3.4 Hit target resolution

The Qt-version uses the functionality of QWebHitTestResult, which allows for retrieval of two values from the clicked DOM-node; the link URL (if one exists) and the title attribute (if one exists) [21]. The title contains a Spotify link, which identifies whether an album or a track is being pressed and appropriate action is taken based on that information.

5.3.5 Overridden event handlers

Several events are propagated to the WebView, containing essential information on how the interface is to be updated or what actions the user is trying to perform. In order to customize the behavior, several event handlers of QWebView’s inherited QWidget [36] are overridden with implementations that have the desired behavior.

5.3.5.1 Mouse clicks and movement

The mouse interaction behavior of the Spotify client differs greatly from that of a web browser. To mimic the Spotify clients’ behavior a number of low-level-event handlers are overridden:

- mousePressEvent – triggered when a mouse button is pressed
- mouseDoubleClickEvent – triggered upon a double click
- mouseReleaseEvent – triggered when a button is released
- mouseMoveEvent – triggered upon mouse movement

The mousePressEvent has a long checklist of possible responses to input. Its purpose is to remove much of the standard browser behavior such as the ability to mark text or drag images, while still allowing clicks on certain elements of the interface. It is also responsible for retrieving and forwarding information upon which the marking of tracks is based.

MousePressEvent globally stores necessary information in preparation for potential drag and drops originating from the QWebView, which always start by a mousePressEvent. Since the slot handles all button clicks it houses several special cases for events such as pressing a scrollbar. Filtering is performed in order to determine when the event should be propagated to the WebView or not, instead of altering the state of the WebView manually.

MouseDoubleClickEvent has only one responsibility, to start playing the target, if the target was a track. The event is never forwarded to the WebView.

MouseReleaseEvent is responsible for updating the internal representation of the interface to maintain correctness in the interface’s response to actions. It also unmarks all tracks except the one pressed, if a track was clicked follow by a mouse release and no modifier keys (alt, ctrl) were used. The event is always forwarded to the WebView.

MouseMovedEvent uses the stored information to decide whether to initiate a drag and drop or not. The decision is based on information about the position moved to and whether the mouse is pressed or not. If the mouse is pressed and has moved since it was pressed, a drag is initiated if the point for the press is on some relevant and draggable information such as an album cover, track or album name. The event is not forwarded if a drag is in progress.

5.3.5.2 Drag and drop

Several drag event handlers are overridden:

- dragMoveEvent – the event handler that commences the drag and drop
- dragEnterEvent – triggered when something is dragged into the Qt area, from an outside view or application
- dragLeaveEvent – triggered when the drag leaves the Qt area
- dropEvent – triggered when the drag ends

The purpose of overriding these handlers is to keep the drag label in a proper state during user interaction. The drag label is a QLabel (see Figure 5-5), which contains the title of the item being dragged. That label needs to be hidden when the drag leaves the WebView, which is the responsibility of dragLeaveEvent, and showed when a drag enters, which is the responsibility of dragEnterEvent.

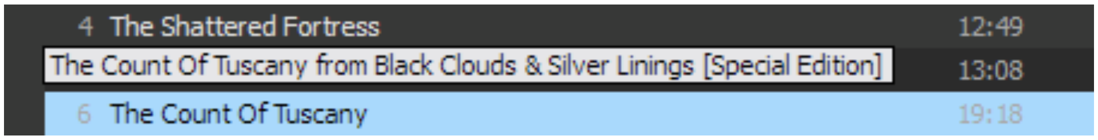

4 The Shattered Fortress
12:49
The Count Of Tuscany from Black Clouds & Silver Linings [Special Edition]
13:08
6 The Count Of Tuscany
19:18

Figure 5-5: A QLabel displaying the title of the currently dragged item

DragMoveEvent is performed every time a drag is in progress and is moved. This is also the entry point of drags originating from the WebView.

5.3.5.3 Key Events

All key presses are intercepted, and forwarded. Certain keys, namely the up and down arrow keys, are used to modify the currently selected tracks on the page.

5.3.6 Context menus and labels

Spotifys native context menus are used, and will offer varying options depending on what type of item was clicked, and how many of that item was selected. For instance, if a set of marked tracks is right clicked the action will apply to all marked tracks.

5.3.7 Image Handling

There are two types of image transformations in Qt, SmoothTransformation (Smooth) and FastTransformation (Fast) [37]. As the name implies, Smooth provides a far smoother transformation while Fast is a brutal and destructive transformation, which is significantly faster, see Figure 5-6 for a comparison. Unless specified otherwise Fast is used, and is used in the current Windows version.

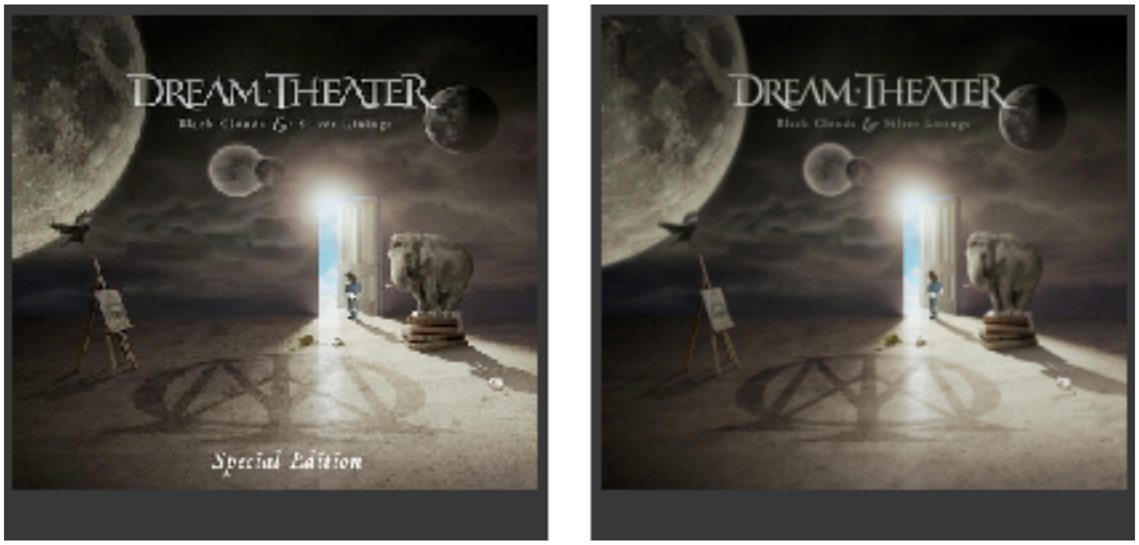


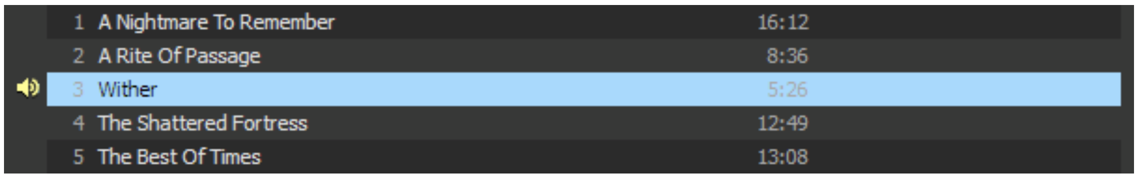
Figure 5-6: Qt Fast and Smooth Transformation

5.3.8 Image loading

The Qt-version loads images in essentially the same way as the Mac client does, see chapter 4.2.1 for a flowchart of the operation. Once the page has finished loading a script is run that ensures that each image has a unique ‘id’ attribute. Each image that has the scheme ‘spot’ in its source attribute will be set to the default image for that image type followed by a request for that image from the Spotify servers. If a reply is received that image is overwritten by the server’s reply.

5.3.9 Selection and currently played track

Selection in the Qt-version works in the same way as the Cocoa version. The keyboard can be used to move the selection up or down, holding the shift key extends the selection. A mouse click will select only the clicked row, while a click with the control key or shift key will toggle selection or select a span respectively.



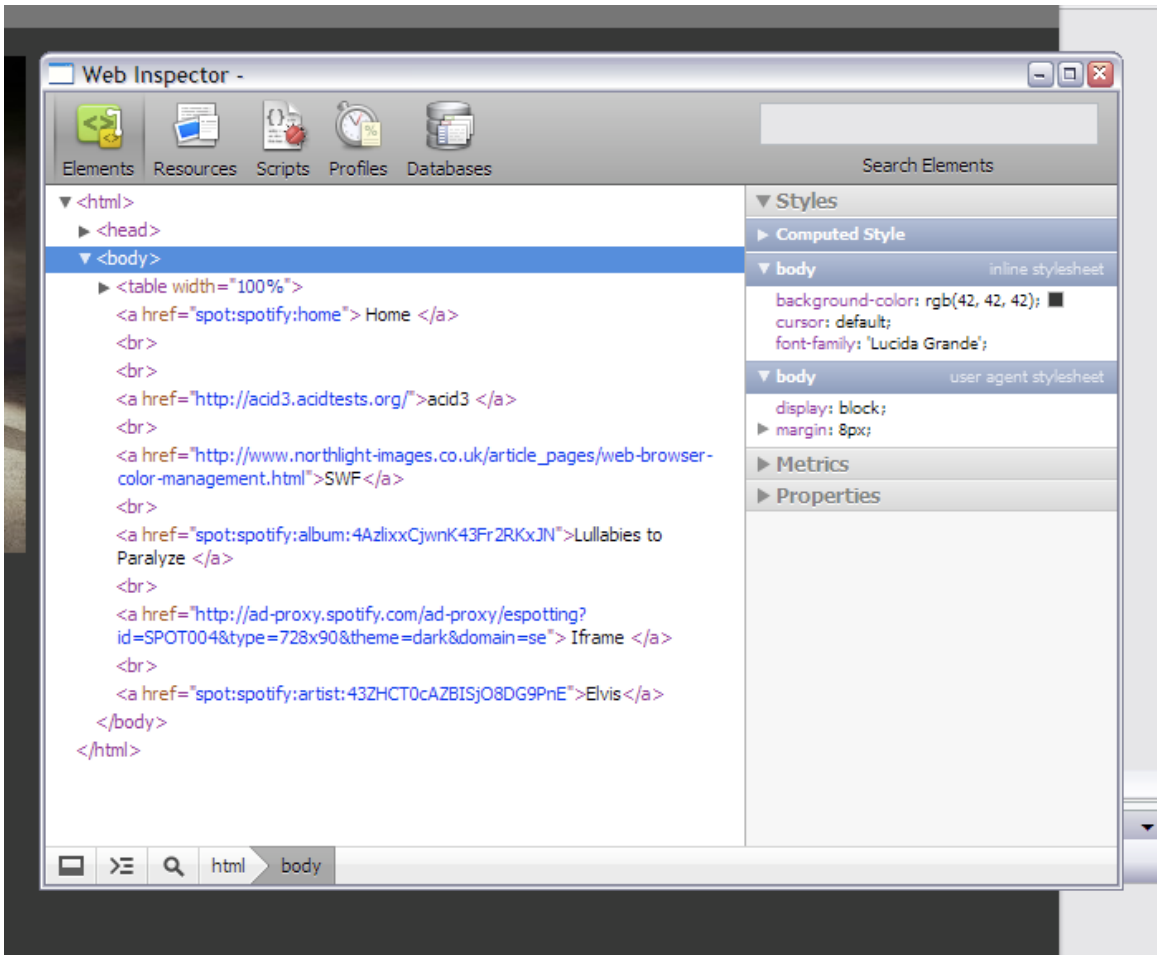
	1	A Nightmare To Remember	16:12
	2	A Rite Of Passage	8:36
🔊	3	Wither	5:26
	4	The Shattered Fortress	12:49
	5	The Best Of Times	13:08

Figure 5-7: Loudspeaker image indicating the track currently playing

When the currently played track is changed the view reflects this fact by showing a small loudspeaker, represented by a data URI containing base64 data, next to the track that is currently playing, depicted in Figure 5-7. The responsibility for representing the currently played track lies on the page. The page has to implement certain JavaScript functions in order so allow for this functionality.

5.3.10 Web Inspector

In order to start the Web Inspector one has to press alt and then double-click somewhere inside the WebView. This will open the Web Inspector in a new free-floating window as can be seen in Figure 5-8 as opposed to Cocoa which opens the Web Inspector in the same frame as the page.



```
▼ <html>
  ▶ <head>
  ▼ <body>
    ▶ <table width="100%">
      <a href="spot:spotify:home"> Home </a>
      <br>
      <br>
      <a href="http://acid3.acidtests.org/">acid3 </a>
      <br>
      <a href="http://www.northlight-images.co.uk/article_pages/web-browser-
      color-management.html">SWF</a>
      <br>
      <a href="spot:spotify:album:4AzlixxCjwnK43Fr2RKxJN">Lullabies to
      Paralyze </a>
      <br>
      <a href="http://ad-proxy.spotify.com/ad-proxy/espotting?
      id=SPOT004&type=728x90&theme=dark&domain=se"> Iframe </a>
      <br>
      <a href="spot:spotify:artist:43ZHCT0cAZBISjO8DG9PnE">Elvis</a>
    </body>
</html>
```

```
▼ body                                inline stylesheet
  background-color: rgb(42, 42, 42);
  cursor: default;
  font-family: 'Lucida Grande';
▼ body                                user agent stylesheet
  display: block;
▶ margin: 8px;
```

Figure 5-8: Web Inspector opens in new window on Qt

6 DISCUSSION

This chapter discusses the results of the Mac and Windows implementation followed by a comparison of important differences between the two.

6.1 Cocoa (Mac)

The Mac version became the version from which we set the standards on how the other implementations should behave. It soon became apparent that the Mac version was not without flaws. These flaws will be discussed in this chapter.

6.1.1 Input Issues

One cannot override the keyboard actions in Cocoa in any straightforward way. The mouse events suffer from a similar problem; the only two events you can intercept are left and right mouse button releases. Even though the WebView inherits its parents' classes (in the graphical hierarchical sense) it cannot override the event methods, leaving little room for customization.

The only way to manage events is to connect JavaScript events to Objective-C objects. This needs to be done in a way as to not steal events from a potential event handler already defined on the web page.

The technique used in this implementation is a script, that runs once a page is loaded, that injects a key press listener into the HTML-body while still maintaining the original listener, if one exists, see Figure 6-1 for an example. This technique could be abstracted into adding any event listener to any event.

```
var x = document.body.onkeypress;
document.body.onkeypress = function(event) {
Client.handleKeyPress_(event); return x(event) };
```

Figure 6-1: Key press listener code

6.1.2 Image Loading

The Mac client features two methods for loading images: the JavaScript-method and the custom protocol method. These two implementations have slightly different implications for the behavior of the program. The more useful of these two is the JavaScript method; it allows for default images while the actual image of an artist or their record is loading. When loading the images each image loads with a one-second delay, approximately. This is really noticeable on a page with several images such as the home view. What causes this behavior is at the moment undetermined.

The custom protocol method does not allow functionality for loading default images. While images are loaded there are no placeholder images, which will lead to the behavior expected of a web page. This behavior does not match the behavior of the Spotify client and a user familiar with browsers will most likely discover the similarities. Interestingly enough the same problem with image loading sometimes occurs with the protocol implementation as with the JavaScript implementation, but less severe. Due to WebKit automatically caching images loaded through protocols, the image loading problem will only occur on images loaded for the first time.

6.1.3 Flash

Since WebKit implements the Netscape Plugin API (NPAPI) it supports Flash. As Safari uses NPAPI for Flash we can assume that the user has the required Flash version installed.

6.2 Awesomium

A number of problems occurred during the development of the Awesomium version of the client leading to the abandonment of this version. This section will explain and discuss these problems.

6.2.1 Stability and reliability

Most problems encountered during the use of Awesomium stem from its lack of reliability. Sporadic crashes can appear deep inside the library for no apparent reason. Since Awesomium is still being actively developed and updated, and the version used in this thesis is an early release, the crashes could be bugs. While this may be resolved in upcoming versions, it unfortunately hinders us from being able to use it for the intended purpose at the moment.

Retrieving the result from a JavaScript execution fairly often results in deadlocks that freeze the whole program. These deadlocks cannot be circumvented and are the main reason for abandoning the Awesomium solution.

6.2.2 Responsiveness

Awesomium renders its content to an off screen buffer. This buffer is then transformed into a bitmap, which can in turn be blotted onto the screen. This is not optimal, the bitmap transformation requires quite a bit of overhead that makes repainting slower. This extra overhead is noticeable, even for the novice user, especially in the case of resizing where the CPU usage would spike. Had there been some functionality for supplying the render method with a device context into which it should render there would most likely have been a performance gain.

It is also not possible to enable automatic re-rendering, while this is feasible for normal operation where any change in graphics is triggered by an event, it is not acceptable in the case of animation. When using Flash, for instance, Flash animations does not update unless triggered manually, since the only way to know if the WebView needs repainting is to explicitly query it. The only way to render Flash correctly is to continually call the render method, which would consume a too great amount of resources.

6.2.3 Customizability

Since the web browser is used for some ads in the Spotify client and the links in those ads should not be opened in that browser but rather in the default browser for the system. Awesomium does not however have any functionality for link delegation.

The banner ads in the Spotify client should not have scrollbars even if the banner size is larger than the area in which it is shown. There is no way in Awesomium to specify the behavior of the scrollbars and thus no way to disable them.

While WebKit has the Web Inspector built in, the Awesomium API does not offer any functionality for accessing it. This makes developing JavaScript more difficult since you cannot debug your code.

Clicking and dragging selects text on the page by default in WebKit, however this behavior is unwanted when using WebKit as an interface renderer. There is no way in Awesomium to disallow selection. Since it is not possible to get information about a specific element given a location it is impossible to disallow selection by filtering which messages to inject.

The lack of the hit test functionality prohibits native context menus as well as drag-and-drop functionality. It is of course possible to manually specify these events using JavaScript on each page. However requiring every page to have similar, if not identical, JavaScript functions is redundant. These scripts could of course be a server side import.

There is a risk that separate pages need different implementation of these functions. Developing these JavaScripts would be difficult due to the lack of a Web Inspector. While it is possible to develop them using the Mac version of the Web Inspector, one still has to get the integration with the Awesomium version working. With no Web Inspector this will be harder.

6.3 Trolltech Qt

The Qt version has quite a few issues and limitations, which makes it harder to implement the required functionality. In this chapter these issues will be discussed.

6.3.1 Flash

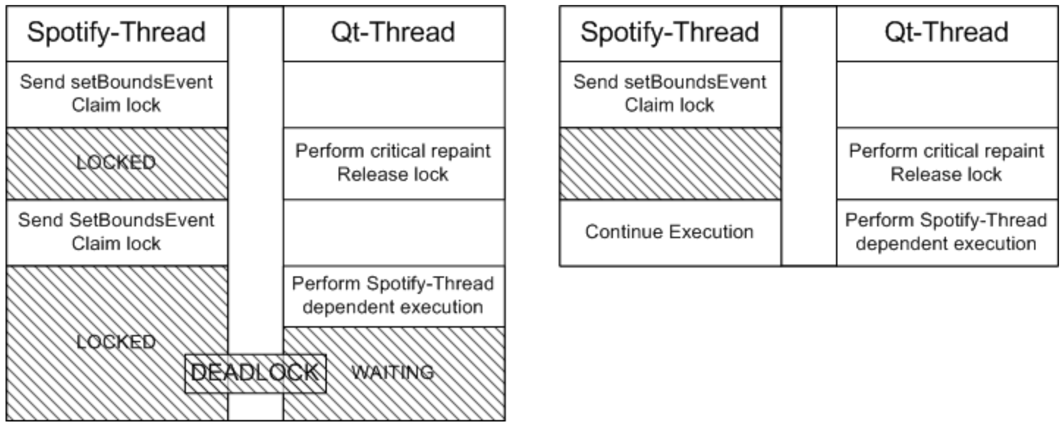
The NPAPI Flash package is not installed on Windows by default. Since most [38] users of Windows use the Internet Explorer (IE) browser a Windows user will most likely have the IE Flash plugin version installed. This adds the need for NPAPI Flash plugin to be installed for Flash to work in Qt. It might not seem very appealing for a Windows user to be asked to install extra software so that ads can be displayed for her.

6.3.2 Threading issues

Since the Qt gui is in one thread while the Spotify client gui is in another an interesting situation occurred during implementation. When a resize occurs an event will be dispatched to the Qt-thread, telling it to resize its windows. In the first version this situation resulted in starvation of the Qt-thread, leading to extremely slow repainting of the WebView contents upon resizing.

Since we were dealing with threads our first attempt at solving this was by the use of a semaphore. The intention was to lock the Spotify thread momentarily, allowing the Qt-thread to finish its repaint without being starved. The repaint was performed as a response to a SetBoundsEvent, but the response consisted of several function calls. Some of these function calls are dependent on components existing, and being available, in the Spotify thread. Primarily functions called by the WinWidget. If the resource required by these functions is not available they will wait for the resources to become available.

It would become apparent that such functionality does not mix well with semaphores. As illustrated in Figure 6-2, our intention was to lock the Spotify thread as soon as it had dispatched an event, performed by the Spotify thread itself by claiming a resource from the semaphore. The Qt-thread would upon completion of its tasks release a resource, effectively unlocking the Spotify thread. After the Spotify thread is unlocked, the rest of the SetBoundsEvent handling would take place in the Qt thread, which is now possible since the Spotify thread is running again.



Spotify-Thread
Qt-Thread
Send setBoundsEvent Claim lock
LOCKED
Perform critical repaint Release lock
Send SetBoundsEvent Claim lock
Perform Spotify-Thread dependent execution
LOCKED
DEADLOCK
WAITING
Spotify-Thread
Qt-Thread
Send setBoundsEvent Claim lock
Perform critical repaint Release lock
Continue Execution
Perform Spotify-Thread dependent execution

Figure 6-2: The deadlock and normal execution case

However, this allowed for a deadlock to occur. If a high number of resizes occurred during a short period of time the Spotify thread would have a new event to process as soon as it left its first lock. With a bit of bad luck with the scheduler the Spotify thread could execute faster than the Qt thread, allowing the Spotify thread to dispatch the next event and lock itself, waiting for the Qt thread to unlock it, before the Qt thread had processed all the function calls that are dependent on the Spotify thread being available, leading to a deadlock, as illustrated in Figure 6-2.

We could not find a way to solve this using semaphores or locks. After some careful optimization of the handling of a SetBoundsEvent, the effect of possible starvation was alleviated to such a level that it was hardly visually detectable on a multi-cored system.

6.3.3 Event interception

A large number of event-handling methods are overridden to provide us with the control needed to adapt the possible interactions as we please. The consequence of this is that we need to create a special case for each behavior we want to alter in any way.

The abstraction level of Qt-events is quite low, focusing on basic input actions such as mouse clicks or key presses. This leaves it up to us to determine what was clicked, where it was clicked and if any potential modifiers were used, such as alt or ctrl-keys.

Since the behavior of the application we are trying to mimic is very specific, and differs greatly from that of a web browser, the overridden methods are intricate and contain many special cases that depend on the current state of the application. If the problem was local, as in affecting only one method, this issue would probably not be severe but only a slight hinder. However, due to the level of detail, the methods need to store information about the state as several instance variables. This makes the methods highly interdependent and fragile to changes.

If one for some reason wants to add or alter a behavior it is easy to unintentionally alter the behavior of other cases, much due to the interdependency, something that happened a few times during our implementation. If it was possible to alter the behavior at a higher level of abstraction this issue could be alleviated.

6.3.4 DOM tree accessibility and hit target resolution

When something in the interface is clicked it is necessary to resolve what has been clicked in order to perform the required actions. A handy way to do this would be to extract which object in the DOM tree has been clicked. One could then use the information of the DOM node and its parents to determine the proper action to take.

Unfortunately, Qt (version 4.5) does not offer access to the DOM tree. The current solution (using the title to store information about the DOM node) is not feasible on a large scale and is used as a ‘quick fix’ at the moment. The easiest way to resolve the issue is probably to wait for the upcoming Qt version, or using a recent nightly build.

6.3.5 Image Handling

The FastTransformation, as described in chapter 5.3.7, is the standard setting. This results in significantly faster execution time when images are to be resized in the home view but also a noticeable lack of smoothness in the images.

There is a neat way to speed up the smooth transformation, referred to as cheat scaling, which uses half-scaling to significantly reduce runtime of the resize [39].

Using this method will bring the execution time for resizing an image, using SmoothTransformation, from approximately 100 times the execution time of FastTransformation to being on par with FastTransformation.

There are minor drawbacks that come with the Cheat Scaling. Since the image will be scaled with the destructive FastTransformation first, some data will be irreparably lost. The visual difference between FastTransformation and Cheat Scaling will be obvious, see Figure 6-3, but the difference between SmoothTransformation and Cheat Scaling is not easy to detect with the human eye.

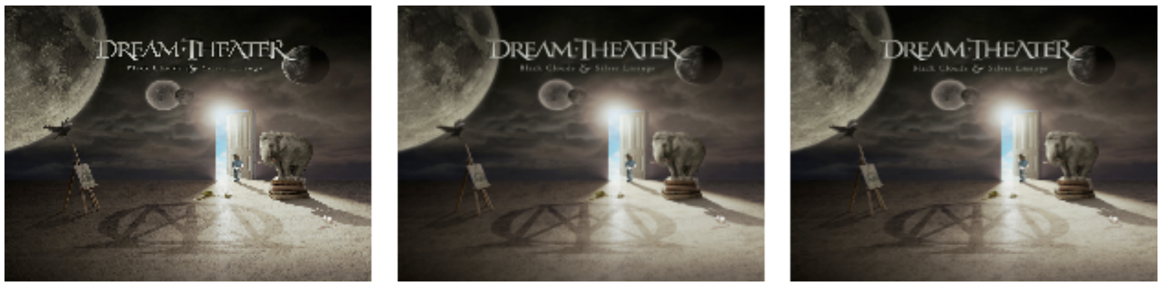


Figure 6-3 Fast, Smooth and Cheat transformations

Unfortunately the actions required to be able to perform the image transformations yourself are arduous and inefficient, in a setting such as ours. The image data needs to undergo several steps before it is in the required state to perform the scaling, of which a few steps involve reading and writing from and to buffers.

Even if those steps were trivial, both computationally and programmatically, the issue of knowing which size an image should be scaled to remains since that information is, in all likelihood, stored in the CSS of the web page. And even if that information was trivial to acquire one would need to store it in such a way that it doesn't order WebKit to alter the size of the image.

Using this method of scaling would most likely create pitfalls for web designers since it would restrict the designers from setting the size of an image in any other way than the non-size-altering manner mentioned above. If the web page for some reason wants to resize the image, e.g. due a table containing the image shrinking below the image's size, the FastTransformation would be used, undoing all cheat/smoothTransformation.

6.3.6 License issues

The LGPL license of Qt will require us to include the copyright notice of the library used and add a link to the LGPL and GPL licenses wherever there is a Spotify license. Furthermore, we would need to redistribute the version of Qt that we used to compile the DLL:s and the WebKit version used. Even if we used the commercial version of Qt, redistribution of the WebKit version would be necessary since its strictly licensed under LGPL/GPL.

6.3.7 Size

The Qt DLL:s require approximately 20 megabytes of disk space. Since they need to be loaded during runtime to be used this will add to the memory consumption of the program. It will affect the disk size of the Spotify client more than the memory consumption, since the Spotify client only requires approximately 2.5 megabytes of disk space without the added DLL:s, which when added will roughly multiply the space required on disk by a factor of eight.

This is a non-desirable effect, since Spotify prides itself upon low memory consumption and little disk space required.

6.4 Summary

This chapter discusses the combined results of the Mac and Windows versions, their similarities and differences. It will also discuss potential difficulties in implementing other views from the Spotify client and which problems one might encounter.

6.4.1 Explicit data sharing

Cocoa, Qt and Awesomium share the same general approach for connecting JavaScript with the programming language (C++/Obj-C) [40] [41], Calling functions for a specific object in JavaScript will call a function in the programming language. However, in the three frameworks one had to specifically state which methods JavaScript should have access to, and it is not possible to make arbitrary objects available. If new views or new functionality needs to be added it is likely that one needs to expose more of the system to JavaScript. Ideally one would like to expose the central components of the Spotify client fully to JavaScript so that one can access the same functions from the script as from the programming language.

6.4.2 Look n' feel and Responsiveness

The same customization options are available in both clients, restricted primarily by the limits of JavaScript, CSS and HTML. However, the same settings might render differently on the two versions. A good example of this is fonts. While the same CSS-settings are used for the Mac and Windows client, they render the fonts quite differently. Figure 6-4 depicts that difference.

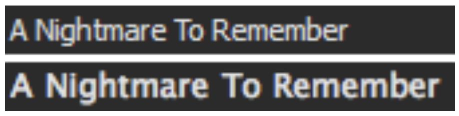


Figure 6-4: Windows and Mac rendering the same font differently, Windows on top.

This behavior requires the design of a view to be adapted to both the Windows and Mac version, if the desire is to have only one design, a process during which a multitude of issues can be encountered. In our current version we are experiencing an issue with the loudspeaker icon not being represented graphically in the same way on each operating system, a typical problem that will need delicate adjustment to look similar on both operating systems.

If we were to use more than one design one of the potential gains of using the same renderer on both operating systems is lost – the ability to design once and use several times.

The responsiveness of the two versions differs slightly. The most prominent difference is the resizing of the WebView. The Mac version has no issues, while the Windows version can, as described in the Windows discussion, lag behind in its repainting during resizing. The occurrence of this issue depends heavily on the performance of the system running the application. A multi-core system will rarely encounter this issue while a single-core system will probably see it more often.

6.4.3 DomTree Manipulation

In both versions, JavaScript is used to manipulate the DOM Tree of the pages. In Qt, which offers no access to it, it is necessary. In Cocoa it is fully possible to do most, if not all, of the DOM manipulation directly through the DOM interface. We decided to strive for maximal similarity between the two versions, therefore the JavaScript technique was utilized in both versions. With the upcoming version of Qt that offers full access to the DOM Tree one would do well to replace the JavaScript code with direct manipulation of the DOM Tree. There would many benefits from such a change.

Developing the JavaScript code was tedious work since you get no help from the compiler if you mistype or forget some minor detail, the JavaScript simply will fail to run. If one could manipulate the DOM Tree directly the compiler would detect these problems at compile time.

A relevant problem is that even if a piece of code is syntactically correct it still might be wrong. Usually one would use a debugger to catch these errors but that is not possible when executing the JavaScript code. Were it possible to manipulate the tree directly one would be able to step through the instructions one at a time and identify the problem.

6.4.4 Image handling

The Spotify client is designed with usability and aesthetics in mind. As mentioned in the Qt discussion chapter the Qt client has a problem with image resizing, leaving the images grainy and pixelated. While it may be possible to circumvent these problems by the methods discussed, it imposes a lot of responsibility upon the designer.

There are quite a few web browsers available that has that same problem with image resizing which has lead web developers to apply a different approach to image resizing, thumbnails.

By creating a resized copy of each image on the server it is possible to achieve an identical image look on both versions. This would probably reduce the amount of data transmitted while browsing the Spotify catalogue due to smaller file sizes. Although we have no statistical data on patterns in common user-browsing of the client views, one could argue that it might increase the load since many views have their own image size, which requires the same image to be transmitted several times in different sizes.

This moves responsibility from the client to the server, which is not desirable. Additionally, this requires the server to keep thumbnail copies of every image and should a new image size be required a new copy of every image needs to be created and stored. A solution to this could be to keep the current system but let the client shrink the image prior to delivering it to WebKit. While this functionality might seem similar to the solution offered in the Qt discussion, it differs by applying the solution to both versions. Neither the Qt nor the Cocoa version are allowed to let WebKit handle the image resizing, instead both of them resize the images themselves. This solution will of course forbid the web designer from using relative resizing, i.e. percentages, for any HTML element.

6.4.5 Extendibility

Qt and Cocoa differ slightly in their interface to WebKit, the Qt interface is in a sense more raw than Cocoa's. The lower abstraction of mouse input is a good example. Where Qt simply offers the possibility to intercept mouse inputs and perform low-level actions yourself or simply pass the event on, Cocoa offers you the option to allow or disallow specific mouse-triggered actions, such as drag and drop or selecting text. Differences such as these can limit the customizability of the different versions.

If WebKit were to be used as the UI renderer the views that are not implemented in this thesis needs to be implemented. The artist view does not differ much from the album view in terms of functionality, which makes this view simple to include. The radio view on the other hand would require some modifications to the system handling input due to the genre and epoch selection buttons. In both the current Cocoa and Qt versions clicks are only allowed if the target has a title. Simply adding a title to these elements would solve this problem, but would also introduce another. Drag and drop is possible on anything with a title, to solve this problem one would have to do string comparisons in order to decide which element was clicked. Essentially this stems from Qt's lack of access to the DOM tree on input events. If Qt had had this functionality it would be possible to set specific attributes which would effectively allow or disallow certain actions.

The remaining functionality of the radio view would probably be less of a challenge to implement. Features like the sliding animation performed when a track changes might require making the page aware of when the track changes.

Should a new view, that is not in the Spotify client at the moment, need to be added it might be necessary to alter the input handling once again to accommodate for that page. It might be a better idea to transfer the responsibility of managing input from the client software to the page, i.e. let each page have JavaScript code to deal with input on specific elements. This would require that much of the Spotify client graphical user interface was made available to the pages.

In the Spotify client there is a navigation system for accessing the previous or next visited view. This behavior not present in this thesis, where the Browser view is considered to be a single entry in the previous-next-list even if one browses through several pages inside the Browser view. There are essentially two ways of rectifying this, either creating new Browser views whenever navigation is necessary or in some way link the previous and next buttons to the browser's internal history. There are problems with both solutions.

Keeping several instances of the Browser view can be memory consuming since each page will keep a separate copy of its resources, i.e. images and HTML, in memory. On the other hand, cycling through the views will be quick since each page will be pre-rendered.

Linking the navigation buttons to the Browser view would require keeping track of which view has been visited recently, presuming that there are views not rendered by WebKit, to be able to determine whether the browser should go back or if another view should be shown. Additionally, whenever the Browser goes back or forwards the page will have to be re-rendered.

7 CONCLUSIONS

The purpose of this thesis was to evaluate whether WebKit is a viable option for interface rendering or not. The Spotify client has several properties that were desirable to maintain after integrating WebKit; low memory consumption, little disk space required and a clean separation of concerns between the server and client.

Using WebKit on Mac OS X was an easy task since it, and required support-libraries, are bundled with the operating system. Windows is a different story, where neither support-libraries nor WebKit itself is included. There are two paths to be able to use WebKit on Windows, using WebKit and implementing support-libraries yourself or using a framework that supplies support-libraries.

TrollTech Qt was selected as the alternative most likely to produce a working version for Windows. Two implementations were produced, one for Windows and one for Mac OS X, which showcase advantages and disadvantages of the respective frameworks.

Several issues concerning input were encountered, most of them stemming from the need to remove functionality from the two versions, basic functionality such as marking text or performing drag and drops. In Qt, this process proved to be taxing due to the high amount of use cases, ultimately resulting in a re-implementation of user-input handling. Cocoa provided easy access to redefining certain high-level behaviors such as marking text, but did not provide easy access to low-level events such as key presses.

The two versions handled images differently. While Cocoa uses a smooth scaling for images, Qt uses nearest-neighbor algorithm. Since images are a central part of the Spotify client such behavior is not acceptable. A smooth transformation is available in Qt but the use of it requires access to a representation of the image and sufficient data to perform the scaling, the extraction of which is complicated and inefficient.

Although both versions use the same HTML code to render the views they differ in several ways visually. If one wants to use the same HTML code to render both views it needs adaptation, which will most likely be iterative, to look and function in the same way on both versions.

Rendering the views with WebKit adds a negligible amount of data to the Spotify client, since the Spotify client already uses WebKit for banners, and will require little extra memory during runtime. Qt however, multiplies the size of the Spotify client by a factor of 8, not including the space required to install NPAPI Flash, and requires significantly more memory during runtime.

In both implementations, some of the clients' responsibilities have been transferred to the server, namely the storage and/or generation of the structure of the interface. In the current Spotify client the server only sends data, while the client constructs and renders the interface. With WebKit, the server would need to send a web page, thereby handling the construction of the interface and leaving only rendering to the client.

Both versions fulfill the most important goal; the ability to alter the interface without rebuilding. WebKit makes it possible to use ample tools - HTML, CSS and JavaScript, for performing the changes.

Unfortunately, the goal is fulfilled at a steep price. Both versions expose many unwanted behaviors that often require less than optimal fixes to perform as desired. Some of these fixes ultimately lead to what can be considered a reimplementation of features that constitute the reason for choosing WebKit, effectively nullifying some of the most desired features.

In closing, we would like to remind you that this thesis describes the results of attempting to integrate WebKit, a product designed solely for web-content [42], into a foreign environment, performing a task that WebKit is not designed to perform. We recommend you to not draw conclusions about WebKit's potency in its natural environment based on our findings.

Dictionary

API – Application Programming Interface, an interface software implements.

Awesomium – A library using components extracted from Chromium to power a web renderer.

Chrome – A web browser created by Google, which uses the WebKit engine.

Chromium – Google's open source web browser upon which Google Chrome is based.

Cocoa – A native object-oriented programming environment for the MAC OS X operating system.

CSS – Cascading Style Sheets, a language used to describe look and formatting of a document written in a markup language.

Deadlock – A situation in which two or more processes are waiting for the other to finish, resulting in neither of them finishing.

DLL – Dynamic Link Library. Microsoft's implementation of the shared library concept.

DOM – Document Object Model, a convention for representing and interacting with objects in HTML, XHTML and XML documents.

GUI – Graphical User Interface

HTML – Hyper Text Markup Language, a language used to construct web pages.

HWND – A window handle, referring to a Windows-window.

JavaScript – Also known as ECMAScript, an object-oriented scripting language.

NPAPI – a cross-platform plugin architecture used by web browsers such as Firefox.

Qt – A cross-platform application and UI framework.

URI – Uniform Resource Identifier, a string of characters used to identify or name a resource.

References

[1] Background information – Press – Spotify, Spotify
http://www.spotify.com/en/about/press/background-info/ - October 2009

[2] The WebKit Open Source Project, Mac OS Forge
http://webkit.org - October 2009
Companies and Organizations that have contributed to WebKit, Apple
http://trac.webkit.org/wiki/Companies%20and%20Organizations%20that%20have%20contributed%20to%20WebKit - October 2009

[3] WebKit Support Library Agreement, Apple Inc.
http://developer.apple.com/opensource/internet/webkit_sptlib_agree.html - October 2009

[4] GNU Lesser General Public License, Free Software Foundation
http://www.gnu.org/licenses/lgpl.html - October 2009

[5] A Quick Guide to GPLv3, Brett Smith
http://www.gnu.org/licenses/quick-guide-gplv3.html - October 2009

[6] Linkers and Loaders – Shared Libraries, John R. Levine
http://www.iecc.com/linker/linker09.html - October 2009

[7] MSDN - Dynamic-Link Libraries, Microsoft Corporation
http://msdn.microsoft.com/en-us/library/ms682589(VS.85).aspx - October 2009

[8] MSDN - Windows API, Microsoft Corporation
http://msdn.microsoft.com/en-us/library/aa383750(VS.85).aspx - October 2009

[9] MSDN - About Windows – Window Handle, Microsoft Corporation
http://msdn.microsoft.com/en-us/library/ms632597%28VS.85%29.aspx#window_handle - October 2009

[10] Mac Dev Center: WebView Class Reference – Overview, Apple Inc.
http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Reference/WebKit/Classes/WebView_Class/Reference/Reference.html#//apple_ref/doc/uid/20001903-30112 - October 2009

[11] Qt 4.5: QWebView Class Reference - Elements of QWebView, Nokia Corporation
http://doc.trolltech.com/4.5/qwebview.html#elements-of-qwebview - October 2009

[12] WebKit trac: Web Inspector
http://trac.webkit.org/wiki/Web%20Inspector - October 2009

[13] The role of ICC profiles in a colour reproduction system – Chapter 2.0
http://www.color.org/ICC_white_paper_7_role_of_ICC_profiles.pdf - October 2009

[14] A Standard Default Color Space for the Internet – sRGB
http://www.w3.org/Graphics/Color/sRGB - October 2009

[15] IETF: The "data" URL scheme
http://www.ietf.org/rfc/rfc2397.txt - October 2009

[16] IETF: The Base16, Base32, and Base64 Data Encodings
http://tools.ietf.org/html/rfc3548 - October 2009

[17] Qt 4.4 whitepaper – chapter 1.1, Nokia Corporation
http://qt.nokia.com/files/pdf/qt-4.4-whitepaper - October 2009

[18] Qt 4.5: QApplication Class Reference - Detailed Description, Nokia Corporation
http://qt.nokia.com/doc/4.5/qapplication.html#details - October 2009

[19] Qt 4.5: QWidget Class Reference - Detailed Description , Nokia Corporation
http://qt.nokia.com/doc/4.5/qwidget.html#details - October 2009

[20] Qt 4.5: Signals and Slots, Nokia Corporation
http://doc.trolltech.com/4.5/signalsandslots.html - October 2009

[21] Qt 4.5: QWebHitTestResult Class Reference, Nokia Corporation
http://doc.trolltech.com/4.5/qwebhittestresult.html October 2009

[22] Qt 4.5: Events and Filters, Nokia Corporation
http://doc.trolltech.com/4.5/eventsandfilters.html - October 2009

[23] Qt 4.5 Meta Objects, Nokia Corporation
http://doc.trolltech.com/4.5/metaobjects.html - October 2009

[24] Qt/MFC Migration Framework, Nokia Corporation
http://doc.trolltech.com/solutions/4/qtwinmigrate/ - October 2009

[25] QWinWidget Class Reference, Nokia Corporation
http://doc.trolltech.com/solutions/qtwinmigrate/qwinwidget.html - October 2009

[26] The Web Standards Project - Acid Tests
http://acidtests.org/ - October 2009

[27] Qt 4.5: QtWebKit Module, Nokia Corporation
http://doc.trolltech.com/4.5/qtwebkit.html - October 2009

[28] Google Code: Chromium, Google
http://code.google.com/chromium/ - October 2009

[29] Google Code: Chromium – Terms and Conditions, Google
http://code.google.com/chromium/terms.html - October 2009

[30] Prince of Code – Awesomium, KHRONA LLC
http://princeofcode.com/awesomium.php - October 2009

[31] Awesomium: Awesomium::WebCore Class Reference, KHRONA LLC
http://princeofcode.com/software/awesomium/docs/class_awesomium_1_1_web_core.html#658ab47066960b2b8a557b3ab7b98048 - October 2009

[32] GNU 'make' - An Introduction to Makefiles, Free Software Foundation, Inc.
http://www.gnu.org/software/make/manual/make.html#Introduction - October 2009

[33] Qt 4.5: qmake Manual, Nokia Corporation
http://doc.trolltech.com/4.5/qmake-manual.html - October 2009

[34] Qt 4.5: QEvent Class Reference, Nokia Corporation
http://qt.nokia.com/doc/4.5/qevent.html - October 2009

[35] MSDN - PostMessage Function, Microsoft Corporation
http://msdn.microsoft.com/en-us/library/ms644944%28VS.85%29.aspx - October 2009

[36] Qt 4.5: QWidget Class Reference – Events, Nokia Corporation
http://qt.nokia.com/doc/4.5/qwidget.html#event – October 2009

[37] Qt 4.5: Qt Namespace Reference - enum Qt::TransformationMode, Nokia Corporation
http://doc.trolltech.com/4.5/qt.html#TransformationMode-enum - October 2009

[38] AT Internet Institute - Browsers barometer - 05/13/2009, AT Internet/XiTi.com
http://atinternet-institute.com/en-us/browsers-barometer/browser-barometer-april-2009/index-1-2-3-169.html - October 2009

[39] Qt Labs Blogs - Creating thumbnail preview, Ariya Hidayat
http://labs.trolltech.com/blogs/2009/01/26/creating-thumbnail-preview/ - October 2009

[40] Mac Dev Center: WebScripting Protocol Reference, Apple Inc.
http://developer.apple.com/mac/library/documentation/Cocoa/Reference/WebKit/Protocols/WebScripting_Protocol/Reference/Reference.html - October 2009

[41] Qt 4.5: QWebFrame Class Reference - addToJavaScriptWindowObject method, Nokia Corporation
http://doc.trolltech.com/4.5/qwebframe.html#addToJavaScriptWindowObject - October 2009

[42] The WebKit Open Source Project - WebKit Project Goals – Non-Goals, Mac OS Forge
http://webkit.org/projects/goals.html - October 2009