Lively Mashups for Mobile Devices

Feetu Nyrhinen[1], Arto Salminen[1], Tommi Mikkonen[1], and Antero Taivalsaari[2]

[1] Tampere University of Technology, Korkeakoulunkatu 1, FI-33720 Tampere, Finland
{feetu.nyrhinen,arto.salminen,tommi.mikkonen}@tut.fi
[2] Sun Microsystems Laboratories, P.O. Box 553 (TUT), FI-33101 Tampere, Finland
antero.taivalsaari@sun.com

Abstract. The software industry is currently experiencing a paradigm shift towards web-based software and web-enabled mobile devices. With the Web as the ultimate information distribution platform, mashups that combine data, code and other content from numerous web sites are becoming popular. Unfortunately, there are various limitations when building mashups that run in a web browser. The problems are even more challenging when using those mashups on mobile devices. In this paper, we present our experiences in building mashups using *Qt*, a Nokia-owned cross-platform application framework that provides built-in support for web browsing and scripting. These experiences are part of a larger activity called *Lively for Qt*, an effort that has created a highly interactive, mobile web application and mashup development environment on top of the Qt framework.

Keywords: mobile web applications, mashup development, Qt, Lively for Qt.

1 Introduction

In the past few years, the Web has become a popular deployment environment for new software systems and applications such as word processors, spreadsheets, calendars and games. In the new era of web-based software, applications live on the Web as services. They consist of data, code and other resources that can be located anywhere in the world. Furthermore, they require no installation or manual upgrades. Ideally, applications should also support user collaboration, i.e. allow multiple users to interact and share the same applications and data over the Internet.

An important realization about web applications is that they do not have to live by the same constraints that characterized the evolution of conventional desktop software. The ability to instantly publish software worldwide, and the ability to dynamically combine data, code and other content from numerous web sites all over the world will open up entirely new possibilities for software development.

In web terminology, a web site that combines ("mashes up") content from more than one source is commonly referred to as a *mashup*. Mashups are content aggregates that leverage the power of the Web to support instant, worldwide sharing of content. Typical examples of mashups are web sites that combine photographs or maps taken from one site with other data (e.g., news, blog entries, weather or traffic information, or price comparison data) that are overlaid on top of the map or photo.

S. Ystad et al. (Eds.): MobiCASE 2009, LNICST 35, pp. 123–141, 2010.
© Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering 2010

Mashups usually run inside a web browser. However, because the web browser was originally designed to be a document viewing tool – not an environment for highly interactive applications – there are challenges when running web applications and mashups that behave in a highly interactive fashion. Support for user interface widgets can also be limited. Furthermore, poor performance of the web browser can be a major issue especially when running mashups in mobile devices. On mobile devices, usability issues cannot be ignored either [9].

In this paper, we present our experiences in developing practical, compelling web mashups, with a special emphasis on making those mashups work well on mobile devices. The work reported here is part of a larger activity called *Lively for Qt* (http://lively.cs.tut.fi/qt) – a project that has created a highly interactive, mobile web application and mashup development environment for the Qt cross-platform application framework (http://www.qtsoftware.com/). The Qt framework was recently acquired by Nokia, and versions of the framework have already been announced for Nokia's device platforms.

The rest of the paper is structured as follows. In Section 2 we summarize the existing environments and tools available for mashup development. In Section 3 we provide an overview of the Qt platform from the viewpoint of web application and mashup development. Section 4 contains a description of the most interesting mashups and other applications that we have written, including some source code of one of the applications. In Section 5, we discuss our experiences and lessons learned during the development of those mashups. Finally, Section 6 concludes the paper and outlines some future directions.

2 Existing Mashup Development Environments and Tools

The landscape of mashup development technologies is still rather diverse, reflecting the rapidly evolving state of the art in web development. Since mashups are usually built on top of existing content available on the Web, mashups can be composed manually using the classic DHTML technologies available in every commercial web browser: HTML, Cascading Style Sheets (CSS), JavaScript and the Document Object Model (DOM) [6]. However, since the actual representation of data, behavior and content can vary dramatically between different web sites, manual mashup construction can be extremely tedious, fragile and error-prone. For instance, since web sites do not generally present any well-defined interfaces that would clearly separate the public parts of the sites from their implementation details, there are usually few guarantees that the behavior and the data representations used by those web sites would remain the same over time.

To facilitate mashup development, a number of tools are available. In principle, mashups can be developed using general-purpose web application development platforms such as Adobe AIR [15], Google Web Toolkit [8], Microsoft Silverlight [12] and Sun Microsystems' JavaFX [1]. However, in practice the capabilities of these general-purpose web programming environments are still somewhat limited when it comes to the flexible extraction and combination of data from different web sites. The same comment applies also to general-purpose web content development tools including for example Adobe Creative Suite (http://www.adobe.com/products/creativesuite/) and Microsoft Expression (http://www.microsoft.com/expression/).

There are a number of existing tools that have been designed specifically for mashup development. Such tools include (in alphabetical order):

- Google Mashup Editor (http://code.google.com/gme/),
- IBM Mashup Center (http://www.ibm.com/software/info/mashup-center/),
- Intel Mash Maker (http://mashmaker.intel.com/),
- Microsoft Popfly (http://www.popfly.com/),
- Open Mashups Studio (http://www.open-mashups.org/),
- Yahoo Pipes (http://pipes.yahoo.com/).

We have reported our experiences in using these systems in an earlier paper [13]. In analyzing the systems, some common themes and trends have started to emerge. Such trends include:

- *Using the web browser not only to run applications/mashups but also to develop them*. For instance, Google Mashup Editor, Microsoft Popfly and Yahoo Pipes use the web browser to host the development environment and to provide seamless transition between the development and use of the mashups.
- *Using visual programming techniques to facilitate end-user development*. Visual "tile scripting" and "program by wire" environments are provided, e.g., by Microsoft Popfly and Yahoo Pipes.
- *Using the web server to host and share the created mashups*. Most of the mashup development tools mentioned above store the created mashups and applications on a web server that is hosted by the service provider.
- *Direct hook-ups to various existing web services*. Since the Web itself does not provide enough semantic information or well-defined interfaces to access information in web sites in a generalized fashion, most of the mashup development tools include custom-built hook-ups to existing web services such as Digg, Facebook, Flickr, Google Maps, Picasa, Twitter, Yahoo Traffic and various RSS newsfeeds.

So far, very little attention has been put on optimizing mashup development for mobile devices. It should also be mentioned that most of the above listed mashup development tools are still under development, e.g., in beta or some other pre-release stage, reflecting the rapidly evolving state of the art in mashup development. Nevertheless, many of the systems are already quite advanced and capable, and – perhaps most importantly – a lot of fun even for children to use.

3 Qt as a Mashup Platform

As part of the broader *Lively for Qt* (http://lively.cs.tut.fi/qt) activity mentioned earlier, we have created a dynamic, cross-platform mashup environment based on the Qt application framework. The broader *Lively for Qt* activity is reported in a separate paper [10]. In this section we provide an introduction to Qt, with a particular emphasis on its suitability for mashup development.

3.1 Introduction to Qt

Qt (http://www.qtsoftware.com/) is a mature, well-documented cross-platform application framework that has been under development since the early 1990s. Qt supports a rich

set of APIs, widgets and tools that run on most commercial software platforms, including Mac OS X, Linux and Windows. In addition, Qt is available for mobile devices based on Nokia's Maemo Linux platform (http://maemo.org/) and Series 60 Symbian platform (http://www.s60.com/). Qt has been used in various commercial applications before. Examples of desktop applications built with Qt include Adobe Photoshop Elements, Google Earth, Skype, and the KDE desktop environment for the Linux operating system. In addition, Qt has been used in various embedded devices and applications, including mobile phones, PDAs, GPS receivers and handheld media players.

Trolltech, the company developing Qt, was acquired by Nokia in 2008. Nokia is currently in the process of making Qt libraries available on their phone platforms. Nokia's market share will make Qt an extremely interesting target platform for mobile applications as well.

From the technical viewpoint, Qt is primarily a GUI framework that includes a rich set of widgets, graphics rendering APIs, layout and stylesheet mechanisms and associated tools that can be used for creating compelling user interfaces that run in a wide array of target platforms. Qt widgets range from simple objects such as push buttons and labels to advanced widgets such as full-fledged text editors, calendars, and objects that host a complete web browser. Dozens and dozens of widget types are supported.

The GUI features of Qt adapt to the native look-and-feel of the target platform. For instance, on Mac OS X, all the widgets look like native Macintosh widgets, while on Windows applications utilizing the same widgets will look like native Windows applications. An essential part in enabling cross-platform GUI behavior is flexible support for widget positioning using *layouts*. Qt's layout components can adapt to different sizes, styles and fonts used by the host operating system. In general, automated layouts give significant advantage when a program is translated to other platforms and languages. The program adapts automatically to changed text sizes and resizes widgets in an aesthetically pleasant way. Additionally, since Qt supports full *internationalization*, all the locale-specific components (such as a calendar widget) automatically adapt to the current regional settings of the target platform.

In addition to its GUI capabilities, Qt has classes for networking, file access, database access, text processing, XML parsing and many other useful tasks. A multimedia framework called *Phonon* is included to support audio and video playback. Qt networking libraries provide support for *asynchronous HTTP communication* familiar from Ajax [2]. Asynchronous networking support is critical in building web applications that do not block their user interface while networking requests are in progress.

3.2 Qt and Web Development

What makes Qt relevant from the viewpoint of web development is that Qt libraries include a complete web browser based on the *WebKit* (http://webkit.org/) browser engine. The necessary DOM and XML APIs are also included to parse, manipulate and generate new web content easily. In addition, Qt includes a fully functional ECMAScript [4] (JavaScript) engine called *QtScript*. The presence of a JavaScript engine is important, since JavaScript – along with XML – is the *lingua franca* of the Web that is used by popular web service APIs such as the Google Maps API [5].

The web browser integration in Qt works in a number of different ways. For instance, it is possible to instantiate any number of web browsers inside a Qt application

using the *QWebView* API. The *QWebView* class provides a widget that can be used to view and edit web documents inside applications. The data in web documents can be manipulated using the built-in DOM and XML APIs.

To support Qt applications in any web browser, a plugin called *QtBrowserPlugin* exists for embedding the Qt environment into any commercial web browser such as Mozilla Firefox or Apple Safari. The plugin makes it possible to run Qt applications inside a web browser, either as standalone Rich Internet Applications or alongside (or embedded in) conventional DHTML and Ajax web content.

JavaScript support in Qt is available both inside and outside the web browser. By default, the QtScript engine can only access those APIs that are part of the ECMAScript Specification [4]. However, by using a tool called *QtScriptGenerator* bindings to all the Qt APIs can be made visible to the JavaScript engine. This makes it possible to create JavaScript applications that combine classic DHTML behavior with widgets and other APIs offered by Qt.

3.3 Using Qt for Mashup Development

With Qt and its built-in web browser and JavaScript support, we have created a dynamic mashup environment that makes it possible to create mashups that can run inside the web browser as well as native desktop or mobile “phonetop” applications. The mashups are written in JavaScript, and they communicate with existing web services using asynchronous networking.

The mashups can leverage the rich Qt APIs for information visualization and processing. This is important since mashup development commonly relies on a plethora of data formats used by different web sites and services. In addition to binary image and video formats such as GIF, JPEG, PNG and MPEG-4, textual representations such as XML, CSV (Comma-Separated Value format), JSON (JavaScript Object Notation) and plain JavaScript source code play a central role in enabling the reuse of web content and scripts in new contexts. Qt provides excellent capabilities for processing such information, especially when combined with a dynamic language such as JavaScript that allows new object types to be constructed on the fly to accommodate the different data formats.

4 Sample Mashups

In this section we summarize the mashups that we have developed for our Lively for Qt system. First, we provide an example that includes source code as well. Then, we present two mashups that have been built on top of the Google Maps API [5]. Finally, we introduce some other types of mashups. All our mashups run on desktop computers (inside and outside the web browser), as well as in the Nokia N810 mobile device – a handheld WiFi webpad built around Nokia's Maemo Linux platform.

In the application descriptions below, some of the screen snapshots have been taken on the Nokia N810 device. For improved viewability, some of the snapshots have been taken on a PC. Further information on these applications is available on our website (http://lively.cs.tut.fi/qt).

4.1 QtFlickr: Animated Flickr Photo Viewer

In order to demonstrate mashup development with Qt, this section provides a simple example that includes source code. The application used here is called *QtFlickr* – a photo viewer application that fetches images from *Flickr* (http://www.flickr.com/) photo service based on keywords (photo tags) that are obtained automatically from the *Twitter* (http://www.twitter.com/) microblogging service, based on current Twitter trends (http://twitter.com/trends). Images are displayed using timer-based animation (rotation).

The general idea of this application is to automatically display images that reflect the most actively microblogged topics in the world. For instance, when the screenshot of the application shown in Figure 1 was taken, the most actively discussed topic in the world was the swine flu (H1N1).

Fig. 1. *QtFlickr* application running on a PC

When the *QtFlickr* application is started, it first obtains a list of the current microblogging trends from Twitter. Then, the application fetches images from Flickr using the trend names as photo tags. The actual loading of trends and images is performed asynchronously using *QNetworkRequest* and *QNetworkAccessManager* classes so that the user does not have to wait while data is being loaded. The image feed from Flickr is parsed using the *QXmlStreamReader* class. The image URLs contained within the feed are stored and the images to be shown are chosen randomly. Initially, each image is scaled according to the size of the application window. To simplify the implementation and to shorten the source code, only one image is displayed at a time.

Source code. The source code of the application's main class definition is shown in Listing 1. Note that this source code is ECMAScript (ECMA standard 262 [4]) code without any additional syntactic sugar. The *Lively for Qt* includes the option to also use the more class-oriented syntax defined by the *Prototype* JavaScript library (http://www.prototypejs.org/).

The main function of the application, *FlickrWidget*, defines the photo viewer class and its constructor. The class is defined as a subclass of Qt's class *QWidget*, allowing

the application to flexibly behave both as a standalone main application (main window) as well as a widget that can be embedded in other Qt components.

The *FlickrWidget* constructor sets up the UI components and connects the components to the required actions. Two layout components are created to arrange widgets within the application window. A *QHBoxLayout* instance is used for horizontally lining up the *QLabel* widgets shown at the top of the application window. A *QVBoxLayout* object then vertically arranges the *QHBoxLayout* object and the *QLabel* object holding the image (*QPixmap*) to be displayed. Two separate *QPixmap* objects are used for images: the first one holds the current image and the second one the image to be displayed next.

```
function FlickrWidget(parent) {

    // FlickrWidget is a subclass of QWidget
    QWidget.call(this, parent);

    // The image references
    this.flickrUrl = 'http://api.flickr.com/'
    +'services/feeds/'
      +'photos_public.gne?format=rss2';
    this.imageUrls = new Array();

    // The visible UI components
    this.currentTagLabel = new QLabel("", this);
    this.imageLabel = new QLabel(this);
    this.imageLabel.setSizePolicy(
    QSizePolicy.Ignored, QSizePolicy.Ignored);
    this.imageLabel.alignment = Qt.AlignCenter;

    this.imagePixmap = new QPixmap();
    this.nextPixmap = new QPixmap();

    // The timers for downloading and rotation
    this.changeTagsTimer = new QTimer(this);
    this.changeTagsTimer["timeout"].connect(this,
    this.changeTagsTimerTimeout);
    this.changeTagsTimer.start(30000); // 30 seconds

    this.fetchImageTimer = new QTimer(this);
    this.fetchImageTimer["timeout"].connect(this,
    this.fetchImageTimerTimeout);

    this.rotTimer = new QTimer(this);
    this.rotTimer["timeout"].connect(this,
    this.rotTimerTimeout);

    this.angle = 90;

    // The layout components
    var hBoxLayout = new QHBoxLayout();
    hBoxLayout.addWidget(new QLabel("Tags:"),0,0);
    hBoxLayout.addWidget(this.currentTagLabel,1,0);
    this.layout = new QVBoxLayout();
    this.layout.addLayout(hBoxLayout);
    this.layout.addWidget(this.imageLabel,1,0);

    this.resize(300,300);
    this.getTwitterTrends();
  }
```

Listing 1. The main function (JavaScript class) *FlickrWidget*

Three *QTimer* timer objects are utilized to execute functions in regular intervals. The first timer called *changeTagsTimer* handles the downloading of image tags from Twitter. The second *QTimer* called *fetchImageTimer* is used for downloading the next image from Flickr on the background after the current image has been displayed for five seconds. The third timer called *rotTimer* is used for rotating the current image. Qt's *connect* function is used for creating the connections between the timers and the callback functions that are invoked when the timers are triggered.

When the timer named *changeTagsTimer* timeouts, the function *getTwitterTrends*, shown in Listing 2, is called to downloads the current Twitter trends. At first a URL pointing to trend file (a JSON file available from Twitter's web site) is defined. The actual asynchronous HTTP GET request is sent using the class *QNetworkAccessManager*.

```
FlickrWidget.prototype.getTwitterTrends =
function() {
  var url = 'http://search.twitter.com/'
    + 'trends.json';
  var accessMgr = new QNetworkAccessManager(this);
  accessMgr["finished(QNetworkReply*)"].connect(
    this, twitterReplyFinished);
  accessMgr.get(new QNetworkRequest(
    new QUrl(url)));
}
```

Listing 2. Function *getTwitterTrends*

Parameter *twitterReplyFinished* defines the callback function that will be called when the asynchronous network request has been completed. The function *twitterReplyFinished*, shown in Listing 3, processes the JSON file that contains a list of Twitter trends. The JSON string is parsed and the tags in it are stored in an array. When the tags have been obtained, the function *loadFeed* is invoked to load images from Flickr.

```
twitterReplyFinished = function(reply) {
  var trendJSONString =
    reply.readAll().toString();
  var trendJSONObject =
    eval('(' + trendJSONString + ')');
  var tags = new Array();
  for(i=0;i<trendJSONObject.trends.length;i=i+1) {
    tags.push(trendJSONObject.trends[i].name);
     }
  this.loadFeed(tags);
}
```

Listing 3. Function *twitterReplyFinished*

The function *loadFeed*, presented in Listing 4, handles the loading of the Flickr XML feed containing image URLs. At first a URL for the HTTP GET request is constructed by adding the user's search terms (image tags) to it. This is the URL that is sent to the Flickr web service to obtain images. The network request and the callback functionality are created and handled in a manner that is analogous to the functions that were used for downloading Twitter data.

The function *flickrReplyFinished*, presented in Listing 5, reads the contents of the HTTP reply utilizing the *QXmlStreamReader* class. The image URLs found in the HTTP reply are parsed and stored in the *imageUrls* array. The actual images are then downloaded using a function called *showRandomImage.* Its behavior is analogous to the *loadFeed* function, so the code is not presented here.

```
FlickrWidget.prototype.loadFeed = function(tags) {
     this.imageUrls = [];
     var currentTag = tags[Math.floor(
       Math.random()*tags.length)];
     var url = this.flickrUrl +"&tags=" +currentTag;
     this.currentTagLabel.text = currentTag;
     var accessMgr = new
    QNetworkAccessManager(this);

     accessMgr["finished(QNetworkReply*)"].connect(
       this, flickrReplyFinished);
       accessMgr.get(
         new QNetworkRequest(new QUrl(url)));
   }
```

Listing 4. Function *loadFeed*

To support image animation (rotation), the *QTimer* object stored in the *rotTimer* variable invokes a function called *rotTimerTimeout* (shown in Listing 6) every 50 milliseconds. The function utilizes a *QTransform* object to rotate the currently displayed image. Qt's fast transformation mode is used instead of smooth transformation to improve animation performance on mobile devices at the cost of the quality of the displayed images. If the angle of rotation is 90 or 270 degrees, the image is projected sideways and is invisible to the user. At that point the image can be switched to the next one.

```
flickrReplyFinished = function(reply) {
  var xml = new QXmlStreamReader();
  xml.addData(reply.readAll());

  while (!xml.atEnd()) {
    xml.readNext();
    if (xml.isStartElement()) {
      if (xml.name() == "enclosure") {
      this.imageUrls.push(
        xml.attributes().value("url").toString());
      }
    }
  }
  /* fetch next image after 3 seconds */
  this.fetchImageTimer.start(3000);
}
```

Listing 5. Function *flickrReplyFinished*

Since image rotation is rather computation-intensive, it is not well suited to low-end mobile devices. We have used it in this application, because it gives a rather

realistic view of the limited processing power and the graphics capabilities of the mobile device and its software stack.

```
FlickrWidget.prototype.rotTimerTimeout=function(){

  // When current image is drawn sideways,
  // switch to the next image
  if (this.angle % 90  == 0 ||
      this.angle % 270 == 0) {

    // Switch to the next image
    this.imagePixmap =
      new QPixmap(this.nextPixmap);
  }

  // Perform image rotation
  var trans = new QTransform();
  trans.rotate(this.angle, Qt.YAxis);
  trans.rotate(this.angle, Qt.ZAxis);

  // Display the image
  this.imageLabel.setPixmap(
    this.imagePixmap.transformed(
      trans, Qt.FastTransformation));
}
```

Listing 6. Function *rotTimerTimeout*

4.2 QtWeatherCameras: Live Road Weather

QtWeatherCameras is a mashup that utilizes the Google Maps JavaScript API and the road weather camera information available from Finnish Road Administration (http://www.tiehallinto.fi) – the government branch in Finland that is responsible for the highway network and road maintenance. The application utilizes the Google Maps API to calculate an optimal route between two chosen points on the map of Finland. The application then obtains information about the nearest road weather cameras along the route, and displays those cameras as markers on the map (see Figure 2). When the user clicks on any of the markers on the map, a live image and current weather conditions from the selected camera are fetched.

The displayed weather conditions include air temperature, road surface temperature and rain measurements. The weather camera image and weather conditions are displayed using a collapsible, semi-transparent widget placed over the map. The map underneath can be panned and zoomed freely.

At the implementation level, the *QtWeatherCameras* mashup uses Qt's *QWebView* web browser widget that has been placed in a *QVBoxLayout* layout component to allow smooth resizing of the application. The *QWebView* widget displays the map images from the Google Maps API.

After the user has selected a weather camera from the map, the application opens a semi-transparent widget called *ImageViewer* on top of the main widget. The widget consist of two *QLabel* components, a *QTextBrowser* and a *QPushButton* widget. The first *QLabel* is used for displaying the name of the weather camera. The image of the

camera is loaded into the second *QLabel* widget. The *QTextBrowser* widget is used for displaying weather conditions from the nearest weather station. Although it looks like a simple text box, the widget is a full-fledged rich text browser that accepts any HTML-formatted string as a parameter, and also supports hypertext navigation. The *QPushButton* widget is used for minimizing and expanding the *ImageViewer* widget. In addition, the mashup utilizes a widget *styleSheet* property that defines the customizations to the widgets' style, including their transparency.

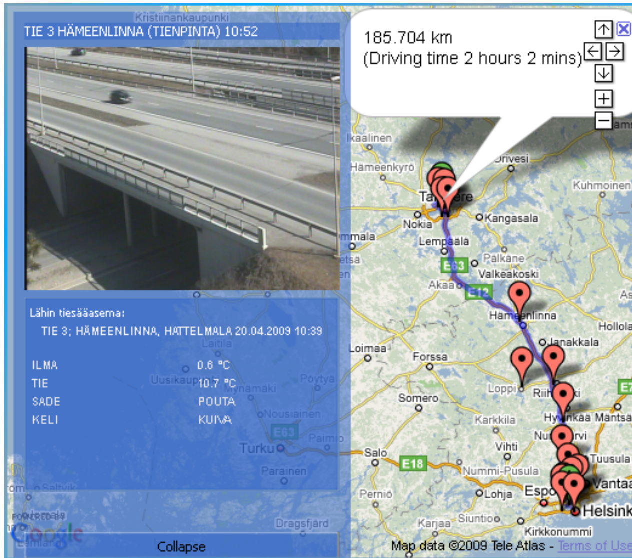

Fig. 2. *QtWeatherCameras* application running on a PC

Images and weather information are downloaded from the web server of Finnish Road Administration. Because the Finnish Road Administration does not provide any well-defined API for accessing the weather camera information, rather heavy parsing is needed in the *QtWeatherCameras* application to digest the information. Locations of road weather cameras are loaded into an array upon application startup. When a new route is created by the user, the application performs the selection of the nearest weather cameras and weather stations locally using the preloaded information. When the user clicks a marker representing a camera, web page containing camera data is loaded and parsed. Weather camera image will be passed on to the *ImageViewer* widget. The data from the nearest weather station is also parsed and passed to the *ImageViewer.*

4.3 QtMapNews: Geotagged RSS Feed Viewer

QtMapNews is a mashup that displays geotagged news items and other geotagged information utilizing the Google Maps API (see Figure 3). The application includes a *QTreeWidget* (tree view) component that lists a selection of predefined geotagged RSS feeds:

- *Earthquakes*: All the Magnitude 5 or greater earthquakes in the world in the past seven days.
- *Emergencies*: The last one hundred incidents/emergencies in Finland based on information available from *Finnish Rescue Service* (http://www.pelastustoimi.fi).
- *News*: Geotagged news from CNN, Yahoo and Yle (Finnish Broadcasting Service).

Fig. 3. *QtMapNews* application running on Nokia N810

The user can add more RSS feeds by pressing a QPushButton labeled "Add...", which will open a simple dialog to enter a new RSS feed. If the new feed is not a geo-coded GeoRSS feed, the QtMapNews application uses a publicly available RSS to GeoRSS converter service (http://www.geonames.org/rss-to-georss-converter.html) to geocode news items contained within the RSS feed. After the geocoding process, the items in the feed are displayed on the map as markers. When the user clicks on a marker, an overview of the news item is displayed on the map.

An interesting additional feature of the *QtMapNews* application is that it includes an embedded web browser to display more detailed information. Whenever the user clicks on a map item that contains an URL, a web browser view is opened inside the *QtMapNews* application (on top of the map) to display the contents of that web page. The web browser is implemented using a *QWebView* widget that is displayed on top of the map view when necessary.

4.4 QtScrapBook: Web Camera Scrapbook

QtScrapBook mashup (Figure 4) is a visual scrapbook that can be used for collecting and displaying static or dynamic images from the Web. The application is intended primarily for keeping track of the user's favorite web cameras. The application uses tabs (a *QTabWidget* object) to display multiple images from the web or local file system. To conserve screen space and to allow the application to be used on a mobile device, only a single image (a single tab) is displayed at a time. Images are updated on regular intervals based on a user-defined interval value that can be adjusted using a *QSlider* widget. A *QTimer* object is used for keeping track of time between updates. Images are fetched asynchronously utilizing the *QNetworkRequest* and *QNetworkAccessManager* classes.

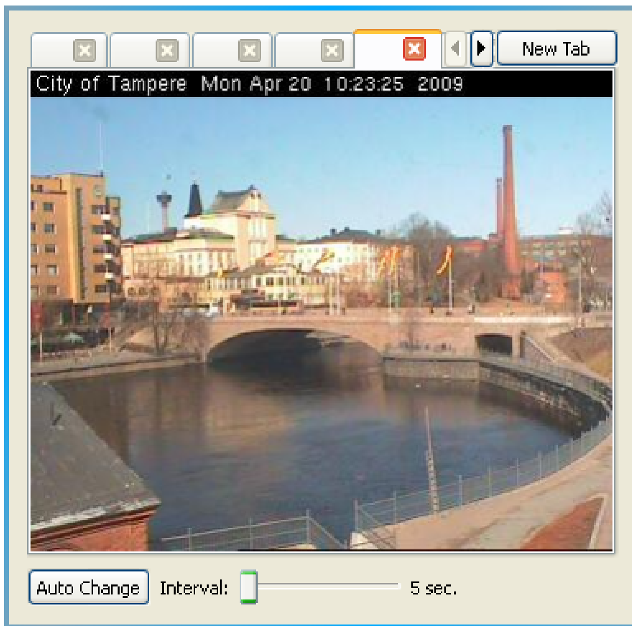

Fig. 4. *QtScrapBook* application running on a PC

The user can add new images and webcams to the application by dragging them from a web browser or from the file explorer of the host operating system. At the implementation level, this is accomplished using Qt's built-in drag-and-drop mechanism that enables the sending of drop events to the application. Every drop event holds MIME data that can be used for determining if the application should handle the event. In this case, the application accepts only those drop events that contain a web address (URL) or a path to a local file.

Image rendering within the *QtScrapBook* application is performed using the *QPainter* class and its *drawImage* method. This makes it possible to resize and scale images flexibly. When the user changes the size of the application, a paint event is sent to the application implicitly. The *drawImage* method is then invoked to (re)render the current image. Rendering is performed in the *Qt.KeepAspectRatio* mode so that the aspect ratio of the original image is always preserved. Furthermore, we utilize bilinear filtering (*Qt.SmoothTransformation* mode) to ensure smooth resizing of graphics.

4.5 QtComics: Comic Strip Viewer

QtComics application collects and displays comic strips from all over the world based on RSS feeds published on the Web. When the application is started, the user can select from a set of feeds that contain multiple comic strips. The selection is performed using a popup list (a *QComboBox* object) that lists the available feeds. To conserve screen space, the application displays only one strip at a time, as shown in Figure 5. The comic shown in this figure is from XKCD (http://xkcd.com/); reprinted with permission.

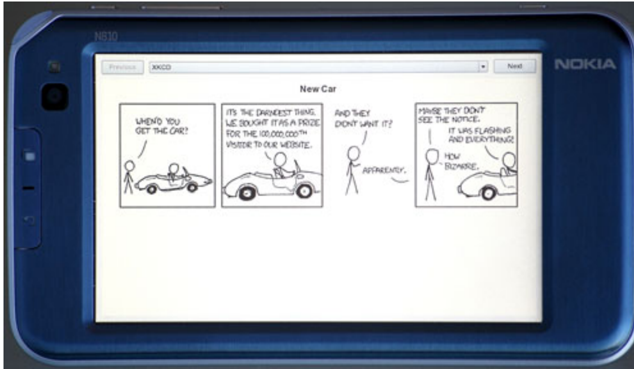

Fig. 5. *QtComics* application running on Nokia N810

At the implementation level, the *QtComics* application uses the *QNetworkAccess-Manager* and *QtNetworkRequest* classes to download the RSS feed asynchronously. The feed is parsed with a *QXmlStreamReader* object. A typical comic strip RSS feed item contains an HTML formatted string. The HTML code found inside the RSS feed item elements is stored into an array. The first array element is then shown inside *QWebView* web browser component. The *QWebView* object downloads content defined in the HTML code asynchronously and displays it on the screen. Thanks to the maturity of the Qt APIs used for accomplishing all this, the source code of the *QtComics* application is very short, only about 180 lines of JavaScript code.

5 Experiences and Discussion

In addition to the mashups described in the previous section we have developed a number of other mashups and web applications. These applications range from various map-based mashups to sports news tracking, weather forecast applications, media players and games. For instance, one of the applications is a mobile audio player that automatically collects artist information and other related information from different web sites. In the Web era, most of such information is available on the Web, albeit not necessarily in an easily digestible form.

All the applications have been written in JavaScript, utilizing the web browser, the JavaScript engine and the rich APIs provided by the Qt platform. While writing those applications, we have gained a lot of experience that is summarized in this section. We start from general observations related to mashup development, and proceed to comments that are specific to mashups on mobile devices. Finally, we summarize our experiences in developing mashups programmatically using Qt.

5.1 General Experiences and Comments

As we have already discussed earlier [13, 14], the majority of problems in web application and mashup development can be traced back to the fact that the Web was not originally designed to be a platform for active content and applications. The transition

from static web pages towards web applications is something that has occurred relatively recently, and the Web has not yet adapted fully to this transition. The problems in areas such as usability, compatibility and security are apparent when attempting to build web applications that run in a standard web browser.

From the viewpoint of mashup development, the two main problem areas are the lack of well-defined interfaces and insufficient security mechanisms. These two areas are discussed below.

Lack of well-defined interfaces. A key problem in mashup development today is the lack of well-defined interfaces that would describe the available web services in a standardized fashion. Although a number of web interface description languages exist, such as the Web Services Description Language (WSDL) [16] or the Web Application Description Language (WADL) [7], these languages are not yet in widespread use.

In general, only a fraction of the data, code and other content on the Web is available in a form that would make the content safely reusable in other contexts. Most web sites do not offer any public interface specification that would clearly state which parts of the site and its services are intended to be used externally by third parties, and which parts are implementation-specific and subject to change. In the absence of a clean separation between the specification and implementation of web sites, there are few guarantees that the reused services would remain consistent or even available in the future. This makes mashup development error-prone and the resulting mashups very brittle.

During the development of the mashups described in this paper, we found that only a small number of services, such as Google Maps and Flickr, offer a well-defined API through which these services can be used programmatically. In many cases, we had to parse HTML pages manually to scoop up the desired data from the web page. If there are subsequent changes in the format of the HTML page, the mashup that parses the page may suddenly stop working properly. This happened to us a few times, e.g., when developing the *QtComics* application.

Security-related issues. Another important problem in the creation of mashware is the absence of a fine-grained security model. The security model of the web browser is based on the *Same Origin Policy* introduced by Netscape back in 1996. The philosophy behind the same origin policy is simple: it is not safe to trust content loaded from arbitrary web sites. When a document containing a script is downloaded from a certain web site, the script is allowed to access resources only from the same web site ("origin") but not from other sites.

The same origin policy makes it difficult to build and deploy mashups or other web applications that combine content from multiple web sites. Since the web browser (the client) cannot easily access data from multiple origins, the mashing up of content must generally be performed on the server. Special proxy arrangements are usually needed on the server side to allow networking requests to be passed on to external sites.

The security problems of the Web present themselves in many other ways. Since there is no namespace isolation in the JavaScript engine, code and content downloaded from different web sites can interfere with each other. For instance, overlapping variable or function names in code downloaded from different sites will

almost surely result in errors that are very difficult to detect. Vulnerabilities based on this characteristics – collectively known as *cross-site scripting* (XSS) issues – have been exploited to craft phishing attacks and other browser security exploits. The possibility of such vulnerabilities is the reason why the same origin policy restrictions were originally introduced.

In the mashup development work described in this paper, we managed to bypass the limitations of the same origin policy by using Qt's networking primitives which do not adhere to the same origin policy. However, the namespace problems could not be avoided, and in a few situations overlapping variable declarations causes us considerable debugging headache, in spite of the relatively advanced debugging capabilities offered by Qt.

The key observation arising from all these problems is that there is a need for a *more fine-grained security model* for web applications. Until a more fine-grained security model and proper namespace isolation are available, mashup development is unnecessarily tedious and unsafe.

5.2 Comments Related to Mobile Mashups

Mashup development for mobile devices is still a new area. In principle, there should be little difference between mashups developed for mobile devices and the general Web. Ideally, as described in the *Mobile Web Best Practices* document of the World Wide Web Consortium [11], there should be just "One Web", meaning that the same information and services should be available to users irrespective of the device they are using.

In practice, One Web is still a dream, although we believe that over time most of the issues will be resolved [9]. In this subsection we discuss the main issues today, focusing on usability, connectivity and performance issues.

Usability issues. The mashups that we developed were not written only for mobile use. Rather, we intended them to be practical on desktop computers as well. However, since our target mobile device (Nokia N810) is stylus-operated and has a significantly smaller, 800x480 pixel screen than a typical desktop computer, usability problems could not be avoided. For instance, many of the Qt widgets used in our mashups are so large that they used excessive amounts of precious screen space. Initially, some widgets ended up being outside the viewable area. Font size differences gave us some problems, too. Fonts that look nice on desktop computers are not necessarily readable on the small screens of mobile devices.

Since our target device had a stylus, applications that require the precise use of a pointing device (e.g., the *QtWeatherCameras* application in which the user has to choose precise points on a map) are quite easy to use even on a small screen. However, we suspect that on other types of mobile devices, such as on conventional “candybar” mobile phones with only a numeric keypad, the use of such applications could be challenging.

Connectivity issues. The availability of a reliable Internet connection is vital for mashups. In mobile devices the network connection can often be slow, unreliable or unavailable altogether. The application developer should take this into account in the design of the applications, and provide feedback to the user when problems do occur.

In our mashup development work with Qt, sporadic connection blackouts did not usually pose major problems. Since our mashups run mainly on the client, the applications remain active if the network connection goes down. When using the Qt networking classes, the network requests will remain active until they are successfully completed, or they will timeout eventually if something goes wrong.

Performance issues. One of the main factors separating mobile devices from desktop computers is performance. Not only are mobile devices considerably slower than their desktop counterparts, but they usually have significantly less memory and storage capacity as well. Although processor and memory limitations will decrease over time, performance issues still cannot be ignored in developing mobile applications today. In the development of the mashups described in this paper, performance differences played a significant factor. Mashups such as *QtFlickr* and *QtWeatherCameras* require a lot of computation, and they are very slow on our target device. The performance problems are caused partially by the slow JavaScript engine used by Qt. In the last year or so, several high-performance JavaScript engines such as Apple's *SquirrelFish Extreme* (http://webkit.org/blog/214/) and Google's *V8* (http://code.google.com/p/v8/) were released, with performance improvements of more than an order of magnitude over conventional JavaScript interpreters. Once such engines become widely available, application response and load times should improve considerably.

5.3 Comments Related to Qt

One of the characteristic features of our mashups and the Lively for Qt system is that the majority of software development is performed programmatically, using *imperative* development style familiar from desktop software development. This is in contrast with traditional web technologies, which rely heavily on *declarative* languages such as HTML and CSS. In this respect, our applications bear close resemblance to applications developed with Rich Internet Application (RIA) platforms such as Adobe AIR [15] or Microsoft Silverlight [12].

Given that Qt APIs have been in development and use since the early 1990s, the Qt APIs are on par with the best RIA systems today. For instance, the expressive power of Qt APIs such as the *QXmlStreamReader* class saved us a lot of work when parsing complex XML data. Furthermore, API documentation and the available development and debugging tools for Qt are in good shape. In general, it was very easy to get started with Qt.

More generally, the combination of an existing, mature application framework with a built-in web browser and popular, fully dynamic programming language (JavaScript) turned out to be a powerful combination. With a JavaScript engine and only a few thousand lines of JavaScript code, it is possible to turn an existing, mostly static, binary, desktop-era application framework into a highly interactive, dynamic web development environment supporting mobile mashup development. Applications require no compilation, binaries or explicit installation, and yet they can utilize the full power of the existing, mature application framework.

On the negative side, although Qt is intended to guarantee platform-independence, we did experience some portability issues. For instance, some widgets or fonts refused to render themselves correctly on some target platforms. Apart from rendering

errors, event-handling differences and some performance-related issues on mobile devices, no other major issues were encountered, though.

The use of the JavaScript language for developing real applications is still a relatively new topic. When JavaScript is used as a programming language for developing full-fledged applications – as opposed to the conventional use of JavaScript as a *scripting* language – one has to be aware of its caveats and peculiarities. These topics have been summarized well by Crockford [3].

6 Conclusions and Future Work

In this paper we have presented an overview of the mobile web mashups that we have implemented in JavaScript on top of Qt – a cross-platform application framework recently acquired by Nokia. This work is part of a larger activity called *Lively for Qt*, an effort that has created a highly interactive, mobile web application and mashup development environment on top of the Qt framework. Here, we summarized our experiences in developing these applications, and included some source code to illustrate the development style.

Plenty of interesting avenues remain for future work. We are especially excited about the possibility of building *location-aware mashups* that have been customized to take into account the user's current location, utilizing the GPS satellite position information available in modern mobile devices. With increasingly reliable and inexpensive network connections, *collaborative mashups* that allow real-time collaboration between multiple mobile users are also becoming a reality. In general, the use of mashups and web applications in mobile devices offers entirely new possibilities that are beyond the reach of web services on desktop computers. Although various obstacles still remain, we are inspired by these possibilities and hope that this paper, for its part, encourages people to continue the work in this exciting new area.

Acknowledgments

This research has been supported by the Academy of Finland (grant 115485).

References

1. Clarke, J., Connors, J., Bruno, E.: JavaFX: Developing Rich Internet Applications. Java Series. Prentice Hall, Englewood Cliffs (2009)
2. Crane, D., Pascarello, E., James, D.: Ajax in Action. Manning Publications (2005)
3. Crockford, D.: JavaScript: The Good Parts. O'Reilly Media, Sebastopol (2008)
4. ECMA Standard 262: ECMAScript Language Specification, 3rd edn. (December 1999), http://www.ecma-international.org/publications/standards/Ecma-262.htm
5. Gibson, R., Erle, S.: Google Maps Hacks. O'Reilly Media, Sebastopol (2006)
6. Goodman, D.: Dynamic HTML: The Definitive Reference. O'Reilly Media, Sebastopol (2006)

7. Hadley, M.: Web Application Description Language Specification (November 9, 2006), https://wadl.dev.java.net/
8. Hanson, R., Tacy, A.: GWT in Action: Easy Ajax with Google Web Toolkit. Manning Publications (2007)
9. Mikkonen, T., Taivalsaari, A.: Creating a Mobile Web Application Platform: The Lively Kernel Experiences. In: Proceedings of the 24th ACM Symposium on Applied Computing, SAC 2009, Honolulu, Hawaii, March 8-12, pp. 177–184 (2009)
10. Mikkonen, T., Taivalsaari, A., Terho, M.: Lively for Qt: A Platform for Mobile Web Applications. In: The Proceedings of the Sixth ACM Mobility Conference, Mobility 2009, Nice, France, September 2-4 (2009) (to appear)
11. Mobile Web Best Practices 1.0. World Wide Web Consortium Recommendation Document (July 29, 2008), http://www.w3.org/TR/mobile-bp/
12. Moroney, L.: Introducing Microsoft Silverlight 2.0, 2nd edn. Microsoft Press (2008)
13. Taivalsaari, A.: Mashware: The Future of Web Applications. Sun Labs Technical Report TR-2009-181 (February 2009)
14. Taivalsaari, A., Mikkonen, T.: Mashups and Modularity: Towards Secure and Reusable Web Applications. In: Proceedings of First Workshop on Social Software Engineering and Applications, SoSEA 2008, L'Aquila, Italy, September 16 (2008)
15. Tucker, D., Casario, M., De Weggheleire, K., Tretola, K.: Adobe AIR 1.5 Cookbook. O'Reilly Media, Sebastopol (2008)
16. Web Services Description Language. World Wide Web Consortium (W3C) Specification (March 15, 2001), http://www.w3.org/TR/wsdl