Aalto University
School of Science
!Fixme **Set degree program** Fixme!

Kimmo Puputti

Mobile HTML5

Implementing a Responsive Cross-Platform Application

Master's Thesis
Espoo, !Fixme **Add English date** Fixme!

DRAFT! — Monday 23rd January, 2012 — DRAFT!

Supervisor: Professor Petri Vuorimaa, Aalto University
Instructor: Risto Sarvas D.Sc.(Tech.)

Aalto University
School of Science

Aalto University
School of Science
!Fixme **Set degree program** Fixme!

ABSTRACT OF
MASTER'S THESIS

Author:	Kimmo Puputti		
Title:	Mobile HTML5 Implementing a Responsive Cross-Platform Application		
Date:	!Fixme **Add English date** Fixme!	**Pages:**	vi + 29
Professorship:	Media Technology	**Code:**	T-110
Supervisor:	Professor Petri Vuorimaa		
Instructor:	Risto Sarvas D.Sc.(Tech.)		
!Fixme **Add English abstract** Fixme!			
Keywords:	!Fixme **Add English keywords** Fixme!		
Language:	English		

Aalto-yliopisto
Perustieteiden korkeakoulu

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Kimmo Puputti		
Työn nimi: !Fixme **Add Finnish title** Fixme! !Fixme **Add Finnish subtitle** Fixme!			
Päiväys:	!Fixme **Add Finnish date** Fixme!	**Sivumäärä:**	vi + 29
Professuuri:	Mediatekniikka	**Koodi:**	T-110
Valvoja:	Professori Petri Vuorimaa		
Ohjaaja:	Tohtori Risto Sarvas		
!Fixme **Add Finnish abstract** Fixme!			
Asiasanat:	!Fixme **Add Finnish keywords** Fixme!		
Kieli:	Englanti		

Acknowledgements

!Fixme **Add acknowledgements** Fixme!
Thank you.

!Fixme **Decide city...** Fixme!, !Fixme **Add English date** Fixme!

Kimmo Puputti

Contents

0.1 Thesis Git repository info

Build time: **Monday 23rd January, 2012 14:58**

Git HEAD:

```
commit 73f4581d60083749bd115799fbfab37e7ae9a390
Author: Kimmo Puputti <kpuputti@gmail.com>
Date:   Mon Jan 23 13:31:44 2012 +0200

    Add rest of results sections.
```

Repository status:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   results.tex
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Mobile OS Type	**Skill Set Required**
Apple iOS	C, Objective C
Google Android	Java (Harmony flavored, Dalvik VM)
RIM BlackBerry	Java (J2ME flavored)
Symbian	C, C++, Python, HTML/CSS/JS
Windows Mobile	.NET
Windows 7 Phone	.NET
HP Palm webOS	HTML/CSS/JS
MeeGo	C, C++, HTML/CSS/JS
Samsung bada	C++

Table 1.1: Required skill sets for different mobile platforms. [1]

Chapter 1

Introduction: Smartphone Market and the Need for Cross-Platform Support

1.1 Smartphone Landscape

1.2 HTML5

1.2.1 History

1.2.2 Markup

1.2.3 CSS3

1.2.4 JavaScript APIs

1.2.5 Related APIs

1.3 Modern Mobile Web Application Architecture

1.3.1 Single-Page applications

1.3.1.1 JavaScript MVC Libraries

1.3.2 Responsive Design

1.3.3 Progressive Enhancement

1.3.4 UI Libraries

1.3.4.1 jQuery Mobile

1.3.4.2 jQTouch

1.3.4.3 Sencha Touch

1.3.5 Hybrid Applications

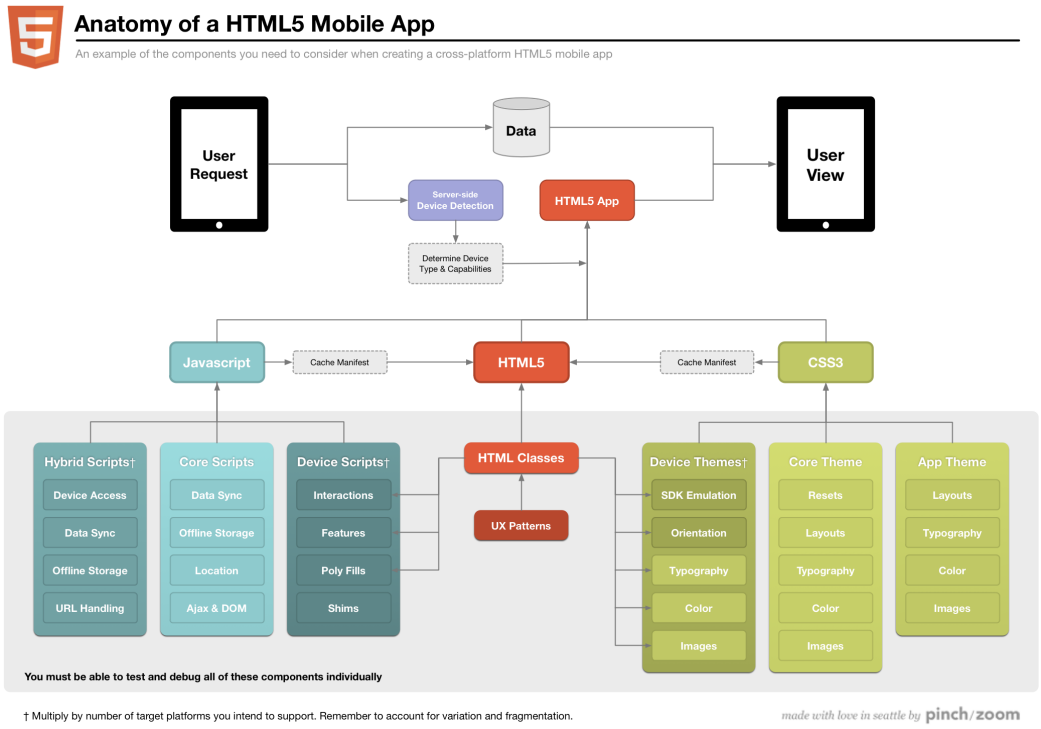

Figure 1.1: HTML5 Mobile Application Anatomy (citation needed)

1.4 Performance Guidelines

There are several web application performance best practices and related guidelines. According to Souders [2], only 10–20% of the end user response time is spent generating and transferring the HTML document from the web server to the client. Therefore, most of the optimization should be done in the frontend for best improvement opportunities. Below we list the performance guidelines defined by Souders [2, 3].

- **Make Fewer HTTP Requests**

  According to Souders, 80–90% of the end user response time is spent downloading components in a page other than the requested HTML page. Therefore, the simplest way to improve the response time is to reduce the number of HTTP requests needed to get all the required components.

  There are several ways to reducing the number of needed HTTP requests. Combining images into sprites, inlining images, or combining separate JavaScript and CSS files result in fewer components needed to download in a page.

- **Use a Content Delivery Network**

  As web applications are deployed and become accessible worldwide, latency might become an issue for users far from the application's web servers. Geographically distributed servers allow for serving the application as close to the user as possible.

- **Add an Expires Header**

  Avoiding a HTTP request altogether is the best option for reducing the response time when downloading the components in a page. Good caching strategies help browsers to know which resources are valid and for how long until they should be updated.

  The Expires header in HTTP tell the client how long a resource is valid, and especially far future Expires headers reduce the need for downloading an updating the components in a page after the initial download.

- **Gzip Components**

  Compressing HTTP responses is an easy and effective way to reduce the size of the data needed to transfer across the network. Compression is supported widely in web browsers and the impact of reduced response

sizes is huge. Using Gzip, the response size is reduced generally about 70%.

- **Put Stylesheets at the Top**

  Putting the CSS files to the top of the document allows the page to load progressively and the browser show visual feedback to the user as early as possible.

- **Put Scripts at the Bottom**

  Because scripts block parallel downloads, they should be included to the page after all other resources. They also block progressive rendering of all content below them in the HTML document, and should therefore be at the bottom of the document.

- **Avoid CSS Expressions**

  CSS expressions are a way to dynamically set CSS properties in Internet Explorer by evaluating a JavaScript code in a stylesheet. However, despite the obvious upsides, the expressions are evaluated at such a high frequency that they negatively impact the performance.

- **Make JavaScript and CSS External**

  There are performance tradeoffs between making JavaScript and CSS external versus inlining them in the HTML document. In the typical case, however, making them external enables the browser to leverage the HTTP caching semantics and thus reduces the needed network transfer.

- **Reduce DNS Lookups**

  Apart from cached DNS lookups, the browser typically needs 20–120 milliseconds to look up the IP (?) address for a given hostname. The cache lifetime of a lookup depends on the TTL (?) value of the DNS record and having the components of a page distributed across several domains might accumulate into a noticeable response time.

  There is also a trade off between unique hostnames and allowed parallel connections and therefore these settings should be configured based on the application architecture and needs.

- **Minify JavaScript**

  Because JavaScript is an interpreted language that must be sent to the web browser as source code, minifying the code reduces the required

network transfer. Minifiers and obfuscators optimize the size of the source code by stripping extra whitespace and comments as well as renaming variable and function names to shorter ones without changing the interpreted behavior of the code.

- **Avoid Redirects**

  Rerouting any component in a page takes time, and avoiding any kind of redirects improves the response times.

- **Remove Duplicate Scripts**

  Including a resource several times serves no purpose but is actually quite common. Developers should make sure to include resources only once.

- **Configure ETags**

  ETags (?) are a mechanism in HTTP for servers and browsers to validate cached resources. The typical default values set by commonly used web servers might hurt performance, and should thus be configured properly to address the application architecture and needs.

- **Make Ajax Cacheable**

  Highly dynamic web sites have a lot of Ajax (?) functionality, and developers should make sure all the requested URLs for data fetching follow the performance best practices such as having the proper caching in place.

- **Splitting the Initial Payload**

  Nowadays, web sites include a lot of resources and JavaScript functionality, but only a small part of the downloaded components are used in the typical use cases of the application. Splitting the resources into bundles that can be lazily downloaded when first needed reduces the initial payload needed to transfer on application startup.

- **Loading Scripts Without Blocking**

  Most browsers block the downloads of other resources when scripts are being downloaded and executed. There are several ways to circumvent this behavior to allow browsers download scripts in parallel with other resources as well as with other script files.

- **Coupling Asynchronous Scripts**

Related to the previous item, when using parallel downloads with scripts that are dependent on each other, race conditions might occur due to the varying order of download and execution. Therefore, asynchronous scripts dependent on each other should be coupled to preserve the correct order of execution.

- **Positioning Inline Scripts**

  Inline scripts do not introduce a HTTP request, but they can still block parallel downloads of other resources and they might affect also the progressive rendering of the page. With the correct positioning of the scripts, these problems can be handled properly.

- **Writing Efficient JavaScript**

  After networking, the obvious place to optimize the runtime speed of a web application is the JavaScript code.

  Because the whole UI (?) and the JavaScript code run in the same browser thread, there can be only one thing happening at a time. Long running functions block the UI from updating and can result in bad UX (?).

  Splitting the running code into properly sized chunks, appropriately leveraging the asynchronous patterns of JavaScript in the application architecture, understanding the details and slow parts of the DOM API, and using several JavaScript programming best practices can result in big improvements in the perceived application performance. [4]

- **Scaling with Comet**

  For real-time data-driven applications, there are various optimization techniques related to optimizing the constant data transfer between the server and the client. The collection of there various technologies is unofficially called Comet.

- **Going Beyond Gzipping**

  Although Gzipping is widely supported in web browsers, there are cases when it is not supported or when the support is not indicated. Stripping extra content such as unneeded whitespace and comments reduces the payload size for uncompressed responses. There are also ways to detect Gzip support if the client does not directly indicate that.

- **Optimizing Images**

Images typically tend to account for a large portion of the page weight, and since the page weight is highly correlated to the response time, images are a natural target for optimization. There are several ways to optimize images either with lossy or lossless conversions.

- **Sharding Dominant Domains**

  By tuning the amount of unique hostnames used for serving all the resources of an application, parallel downloads can be better leveraged. Also, by using HTTP 1.0 with proper Keep-Alive headers or HTTP 1.1 with proper persistent connections the parallel downloads can be tuned for better performance.

- **Flushing the Document Early**

  Some web application frameworks allow flushing parts of the document to the user before the whole document is generated. This enables progressive rendering and gives faster feedback to the user and thus improves the perceived performance.

- **Using Iframes Sparingly**

  Iframes enable developers to embed a separate HTML document inside another document. They are useful in sandboxing external documents in the same view, but the iframe element is the most expensive DOM element related to the page performance.

- **Simplifying CSS Selectors**

  There are several ways to choose elements in CSS stylesheets to apply the defined properties to. Some selectors are faster than others and some have terrible performance.

Chapter 2

Research Question: HTML5 - Hype versus Realities?

Chapter 3

Methods: Example Application and Library

3.1 Qt Developer Days 2011 Conference Schedule Application

The Qt Developer Days[1] is a conference for developers using the Qt cross-platform application and UI (?) framework[2]. We created a mobile web application with contextual and personalized session information and daily schedule for the conference.

3.1.1 Application Architecture

The conference schedule[3] is a single-page application (citation needed)with a lightweight backend written in Python using the Django Web Framework[4].

The backend provides the static assets (JavaScript, CSS (?), images, etc.) and an API (?) for persisting session feedback to a MySQL[5] relational database. It also generates the HTML5 AppCache (citation needed)offline cache manifest file based on the categorized device type.

The frontend is a JavaScript application written using the Backbone[6] MVC (?) framework. Other used JavaScript libraries include Underscore[7]

[1] `http://qt.nokia.com/qtdevdays2011/`
[2] `http://qt.nokia.com/`
[3] `http://m.qtdevdays2011.qt.nokia.com/`
[4] `https://www.djangoproject.com/`
[5] `http://www.mysql.com/`
[6] `http://backbonejs.org/`
[7] `http://underscorejs.org/`

for data manipulation, jQuery[8] for DOM (?) API abstraction, Handlebars[9] for templating, and Modernizr[10] for feature detection. The HTML5 Mobile Boilerplate[11] was used as an initial markup structure for the application. The architecture of is depicted in Figure 3.1.

Wireless networks can be unreliable in conference settings, so offline support was also added using several different JavaScript techniques and HTML5 APIs.

The application was designed for touch screens on various platforms and screen sizes. The layout adjusts to the available space and provides rich interactive components. Integration to social networking services was also added as an additional functionality.

!FIXME **add screenshots on different devices (at least phone and tablet** FIXME!

3.2 JSONCache JavaScript Library

JSONCache is a lightweight JavaScript library for fetching JSON (?) data in unreliable networks. The library was designed especially to handle unreliable mobile networks with connection problems and short interruptions. The goal is to avoid networking as long as possible and failing gracefully if the network connections are not stable.

JSONCache provides two main functionalities: data caching and attempting to fetch the data multiple times.

The caching layer uses the client side localStorage (citation needed)cache of HTML5 (?). Data requests can be done using the JSONCache API (?) which always checks the local cache first before opening any network connections. If the data is already in the cache, the cached data is checked for validity and if the data has not been expired, it is returned immediately. If the data is not in the cache or it has been expired, a new network request is made and the received data is cached and returned. The expiration time of a data item can be configured in the library settings.

JSONCache also tries to fetch the data multiple times to handle small interruptions in network connections. !FIXME **add example and explain that it is very common** FIXME!. If a data fetch fails, a new fetch is issued after a timeout (defined in the configuration). On subsequent attempts the

[8] `http://jquery.com/`
[9] `http://handlebarsjs.com/`
[10] `http://www.modernizr.com/`
[11] `http://html5boilerplate.com/mobile`

timeout is increased, and after a defined number of attempts the fetch error is issued.

Figure 3.2 shows an interactive demo of the JSONCache library. The demo[12] simulates the caching and fetching functionality of the library by simulating a unreliable network based on the configuration.

[12] `http://kpuputti.github.com/JSONCache/demo/index.html`

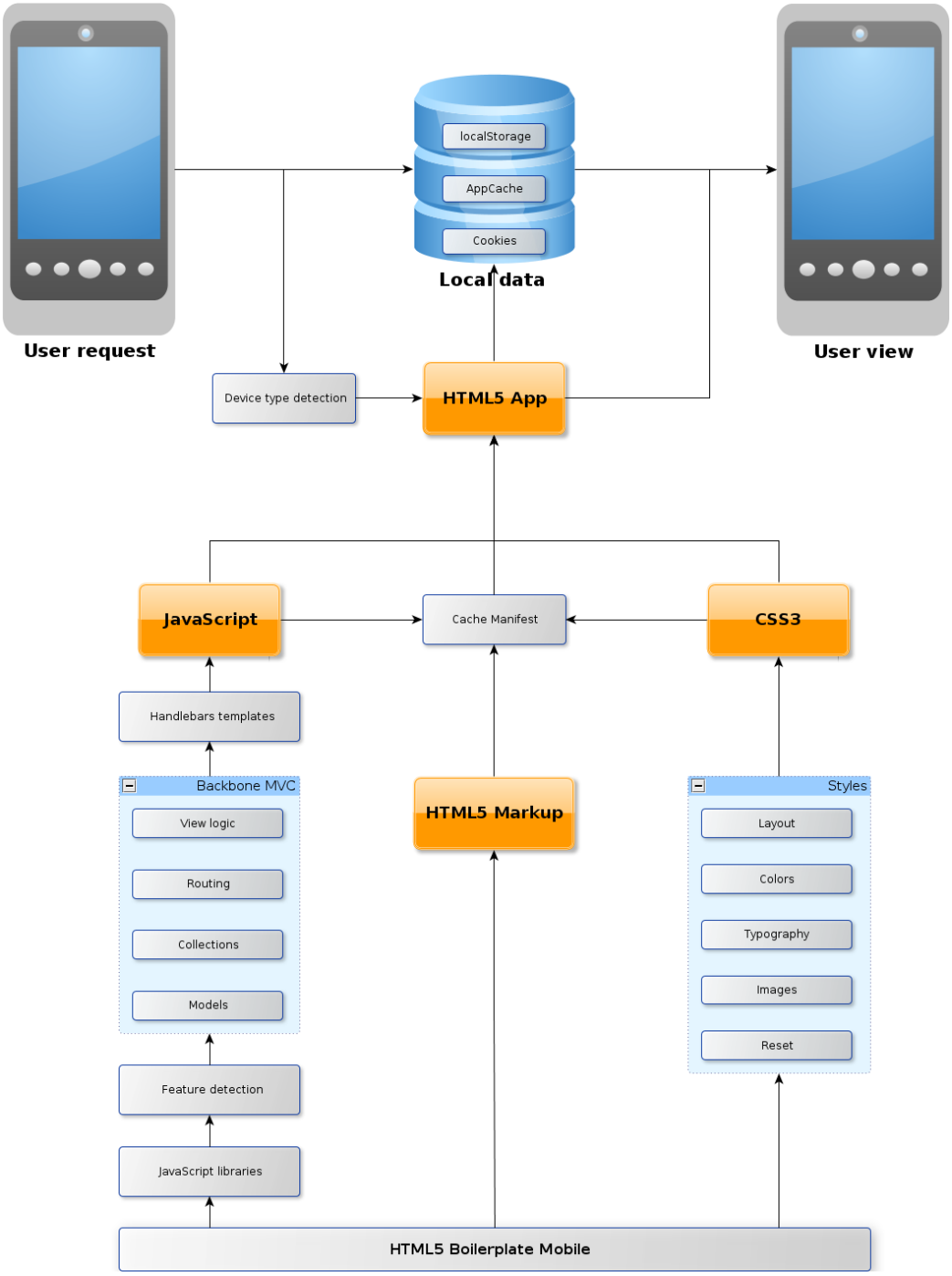

Figure 3.1: Conference schedule application architecture.

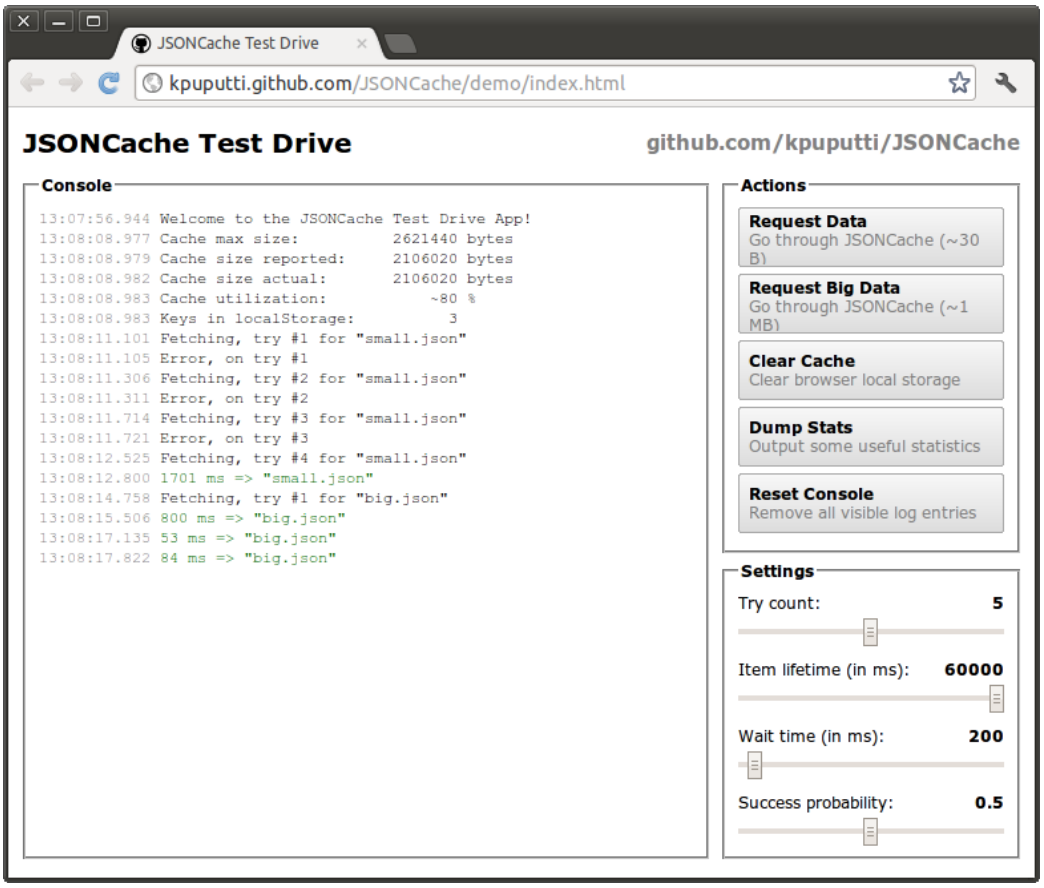

Figure 3.2: Interactive JSONCache demo.

Chapter 4

Results: What Was Good and Where Were the Compromises

4.1 Targeting Different Platforms

Despite the web browser being the unified environment for different platforms, there are lots of differences between various devices. The form factors vary from tiny mobile screens to touch screen tablets and desktop monitors and each device and platform has its own feature set. There are also known bugs in the browsers that have to be handled.

Therefore, means to detect the user's device are needed. Here we present two such means: device detection and feature detection. Both of these were used in our conference application.

4.1.1 Device Detection

The User-Agent (UA (?)) HTTP (?) header contains detailed information of the web browser and platform where the request originates. As we can see from Table 4.1.1 (!Fixme **Check table ref number** Fixme!), we can extract platform and browser specific information from the UA header.

In the conference application, device detection was used in the backend to provide a different offline AppCache manifest to different device groups. The detection was also used in defining the assets to be preloaded in the application. The devices were divided into four categories based on the rules defined in Table 4.1.1 (!Fixme **Check table ref number** Fixme!). There were serious limitations in this approach, and compromises had to be made.

First, there is no way to surely know if the device actually is what it reports itself to be. Second, the most important thing to know when generating the screen specific assets in the manifest file would have been the

Device	**Platform**	**User-Agent**
Samsung Nexus S	Android 2.3.4	Mozilla/5.0 (Linux; U; Android 2.3.4; en-us; Nexus S Build/GRJ22) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1
Apple iPhone	iOS 3.1.3	Mozilla/5.0 (iPhone; U; CPU iPhone OS 3_1_3 like Mac OS X; de-de) AppleWebKit/528.18 (KHTML, like Gecko) Version/4.0 Mobile/7E18 Safari/528.16
Apple iPad	iOS 5.0	Mozilla/5.0 (iPad; CPU OS 5_0 like Mac OS X) AppleWebKit/534.46 (KHTML, like Gecko) Mobile/9A334
Unknown	Android	Opera/9.80 (Android; Opera Mini/6.5.26571/26.1023; U; de) Presto/2.8.119 Version/10.54

Table 4.1: Example User-Agent strings.

screen size. However, this information is not present in the UA header. We could have listed all the assets for all the devices, but then the list of offline assets would have grown too much and, for example, have large images also for older mobile phones.

Despite the drawbacks, the received advantages of this approach outweighed the possible compromises. The worst that could happen was that the device was wrongly classified and the proper resources were not downloaded for offline use.

Getting platform and browser information from the UA header might look tempting and useful, but it is considered a bad practice to detect a device from it and provide device specific bug fixes or additional features. The header can easily be changed and some browsers or browser plugins even provide preconfigured values for certain browsers or devices for spoofing. Also, the device specific bug fixes might become obsolete with platform updates, and the application might break due to invalid expectations. This is why feature detection is generally the recommended option whenever possible.

Rule	Device Type
'iPad' in UA	highres
'iPhone' in UA	iphone
'Android 3' in UA	highres
'mobile' (case insensitive) in UA	mobile
'MIDP' in UA	mobile
'Opera Mobi' in UA	mobile
'Opera Mini' in UA	mobile
otherwise (desktop computer)	highres

Table 4.2: Device type detection rules.

4.1.2 Feature Detection

Feature detection is an important concept in Progressive Enhancement design (See Section 1.3.3). A lot of the HTML5 related JavaScript APIs are still unsupported in several platforms, but browser developers are constantly filling the gaps. Therefore, it is important to check whether a certain feature is supported and provide graceful fallback mechanisms for browsers lacking the functionality.

Doing runtime feature detection provides the possibility to give additional functionality to modern browsers and instant support for devices that add the feature support in the lifetime of the application. In the conference application (!FIXME **ref needed?** FIXME!), we used the Modernizr feature detection library (!FIXME **already ref earlier, add bib entry?** FIXME!) to check for HTML5 features.

For example, the user could add sessions to his or her favorites by clicking the star in the agenda or on the session details view (!FIXME **add screenshot?** FIXME!). The favorites were then listed on the home view together with information about the time left for them to begin.

We used HTML5 localStorage for storing the favorites in the user's web browser. By using Modernizr, we detected localStorage support and showed the favorite stars only in browsers that supported the functionality. For all other browsers, the stars were simply hidden and users could not add favorites. We could have also provided a fallback mechanism for persisting the favorites to the backend, but for simplicity and because we targeted mostly modern platforms, this approach was considered as reasonable.

Property	Description	Value
height	Height of the viewport.	pixel value or 'device-height'
width	Width of the viewport.	pixel value or 'device-width'
initial-scale	Initial zoom level.	float value (0.01–10)
minimum-scale	Minimum zoom level.	float value (0.01–10)
maximum-scale	Maximum zoom level.	float value (0.01–10)
user-scalable	Enables/disables zoom.	'yes' or 'no'
target-densitydpi	Visual pixel density.	dpi value, 'device-dpi', 'high-dpi', 'medium-dpi', or 'low-dpi'

Table 4.3: Viewport meta tag configuration for Android.

4.2 Targeting Different Screens

Probably the biggest difference in various devices and form factors is the screen size, resolution, and dimensions. Web applications should adjust to the available space and flexible handle screen orientation and window size changes.

First, to target mobile and tablet platforms, the viewport meta information should be indicated in the document. The following tag was used in the conference application:

```
<meta name="viewport" content="width=device-width,
                               initial-scale=1.0">
```

The viewport meta tag was first introduced in Apple's iPhone and afterwards ported to other platforms, such as Android. The possible configuration options and default values might vary between platforms. Values accepted by Android are shown in Table 4.2 (!Fixme **Check table ref number** Fixme!) (citation needed). iOS devices also support these same properties.

If we do not set the viewport configuration tag, the device uses its own default values for the properties. For example, the default value for the width property is 980 pixels in iOS (citation needed), which is clearly defined for web sites targeting desktop browsers. Without setting this value to something smaller and more appropriate in a mobile context, the whole application is very wide and has small and unreadable text in the initial zoom level.

In the viewport configuration we used for the conference application (as defined above), we set the viewport width to 'device_width'. This makes the

application width to adjust to the visual pixels of the device screen and works well with screens of different sizes and dimensions. The only other viewport property we set is the initial scaling. This is set to 1.0 to force the browser to render the application without any initial zooming.

In addition to the viewport configuration, we used media queries (citation needed)to use better background images for high resolution screens. We also dynamically set the map view (!FIXME **Add screenshot?** FIXME!) images based on the screen dimensions so that we could provide smaller images for smaller screens and high resolution images for tablets and other devices with larger screen estate.

4.3 Handling Different Orientations

As shown in the previous section, screen sizes and dimensions vary between devices. In addition to handling different resolutions and dimensions, we must also handle screen orientation changes. The width and height of the touch screens are usually different, and the user can hold the device either in portrait or in landscape mode and in any point switch between these two.

In the conference application, we wanted to have different header and footer background images for different orientations. We also needed to redraw the agenda view when the screen width changes since the items on the schedule needed to be dynamically positioned to the available space.

Mobile browsers trigger an 'orientationchange' event whenever the device orientation changes. We listened to this event, inferred the orientation from the screen dimensions, and executed the wanted functionality for the event. We also had to do a fallback for Mobile IE (?) browser to listen to the window resize event because the browser does not support the orientation change event.

4.4 Handling Mobile Networks

One of the biggest problems in mobile web applications is the often slow and unreliable network. Our conference application was designed for a context where the application cannot trust on the networking but should still manage to handle interactions and persist application state. Also being a conference where people come from around the world, the network data transfer cost might be surprisingly high, and thus bandwidth should be saved whenever possible.

4.4.1 Minimizing Data Transfer

The best approach to minimize data that needs to be transferred is to avoid the transfer whenever possible, for example, with proper caching. However, with initial download or with dynamic data, the second best option is to minimize the size of the data needed to be transferred.

First, we made sure the data was minimized and compressed with Gzip. Second, using JSON (?) instead of XML (?) in Ajax (?) requests saves bandwidth and needed effort of the browser to process the data. Third, using the offline manifest ensured that the application assets and data needed to be downloaded only once, and using localStorage we could store the application state locally to the browser avoiding the network completely.

4.4.2 Caching

Caching on different levels of the application stack is one of the most important optimizations that should be done. Caching can be done in the client side using HTML5 APIs, on the HTTP level letting the browser handle it complying to the HTTP caching header semantics, or in various levels of the backend application stack.

In the conference application, we put the most focus on the HTTP caching. Following the performance guidelines specified in Section 1.4, we created unique URLs (?) for all different versions of all static resources (images, CSS, JavaScript, and AppCache manifest files) and set a far future expires header for them. This way we could tell the browser to cache all resources as far as possible and updating the resources was handled by changing the version number in their corresponding URLs.

In addition to the HTTP-level caching, using the AppCache manifest file told the browser to cache all needed resources to a more persistent offline cache, which minimized needed downloads on application startup if the resources were already in the cache.

Client side caching was used in saving the user specific state in the conference application and experimented with the JSONCache library specified in Section 3.2. Using localStorage, we can persist data in the browser and avoid networking if the cached data is still relevant.

JSONCache handles the localStorage caching automatically, with only user configuration needed for setting the data lifetime. Every time the data is rerequested, the local cache is checked first, and networking can be avoided altogether.

4.4.3 Preloading

One way to prevent UI (?) slowness due to flaky networks is to preload resources and data that is expected to be used later on. In the conference application we predownloaded background images and other graphics in the application initialization.

For example, downloading the header and footer background images for both orientations made the device orientation change more responsive because otherwise the browser would have started to download the images after the orientation had already changed. With preloaded images the browser just had to switch the image and render it instantly without any networking.

4.4.4 Offline Support

Using HTML5 AppCache offline manifest file and storing application state to localStorage, we provided full offline compatibility for the conference application. With the offline manifest, we specified the needed resources for all device types as categorized by the rules defined in Section 4.1.1. The offline cache also made subsequent application startups faster since the cache is more persistent than the HTTP cache in browsers.

The offline support was especially critical for the conference application since the conference had indeed very bad wireless network. Without the offline support, users would not have been able to check the session schedule during the conference.

However, the only thing needing the network was the session feedback functionality. The application had a feedback form for all sessions, and the submitted data was persisted in the backend. In offline mode, this functionality was not available. Going further, we could extend the offline support by saving the given feedback, for example, to localStorage and sending it later to the backend server when the network connection is open again.

!FIXME **Add dev headaches: device categories, refreshes in dev, cache updates** FIXME!

4.4.5 Handling Interruptions

Small interruptions are common in mobile networks (citation needed). For example, the user might have a stable network connection, but after walking into an elevator the connection drops for a moment. Then after exiting the elevator the device reconnects to the network. Applications should expect these interruptions and should not fail immediately with brief interruptions in flaky networks.

JSONCache library introduced in Section 3.2 had a functionality to overcome these issues. The library tries to download the requested data multiple times, and fails only when the configured maximum attempt count is reached. With every iteration, a timeout is set for a new request, and the timeout is increased after each failed attempt. This approach works very well, and together with localStorage caching lets data updates circumvent small network interruptions failing only when the network connection seems to be completely down.

4.5 Animations

Animation and transitions, if not overused, can be a valuable addition to the UX (?) of an application. For example, having a simple sliding animation between different views makes the application more uniform and pleasing to the eye.

There are several ways to animate elements in a web application. The simplest is to use CSS3 (?) animations. However, the performance of the animations is not yet good enough for a cross-platform mobile application. We tried to animate the view changes in our conference application, but even a simple cross-fade did not have good enough performance in all target platforms.

Using progressive enhancement techniques (see Section 1.3.3) we could have provided enhanced experience for the platforms that support animations well, but only iOS devices performed well enough, so for simplicity we did not use any animations.

4.6 Following JavaScript Best Practices

There are lots of best practices and conventions that have been developed by the web developer community. A lot of these tried and tested techniques are outlined in the HTML5 Boilerplate (citation needed)that we used as our base for the conference application. Here we present some techniques and tools that help to improve application performance and reduce bugs.

4.6.1 JSLint

JSLint[1] is a code quality tool for JavaScript. There are several JavaScript features that are suboptimal for performance or code maintainability. JSLint

[1] `http://jslint.com/`

also checks for JavaScript syntax and convention violations, which is valuable because the code will be sent in the source form to be interpreted by the browser.

We had automatic JSLint checking integrated in the Emacs (citation needed)editor that we used for all JavaScript programming, which helped to notice common errors as early as possible and made the code cleaner.

4.6.2 Lazy initialization

Postponing work as long as possible is a valuable optimization technique. In lazy initialization we minimize initialization work in application startup to render the initial view fast. We then initialize additional views only when they are requested.

Implementing lazy initialization needs more work than simply doing all initialization work in the application startup, but the received benefits are worth the extra effort. In the conference application we tried to postpone all work to be done as late as possible and doing as little work as possible for faster execution.

4.6.3 Efficient DOM Manipulation

After mobile network issues and data transfers, DOM manipulation is one of the first things to optimize for performance. There are several known performance issues, but the biggest and most common issue is updating a number of elements at once. [4]

The application might, for example, refresh the contents of a list, and with an overly simplistic (but still common) implementation would create the list items and add them to the list container one by one. This causes the browser to reflow the page after each insertion and might add up to UI (?) artifacts and slowness.

One approach to handle updating several elements at once is to use document fragments. With document fragments, several elements can be added to one fragment, which can then be added to the element container. This has no effect on the DOM (?) tree itself, but it requires only one reflow from the browser. One other solution is hiding the container while its contents is modified, and showing after the modification are done. [4]

We used these techniques in the conference application to minimize UI (?) reflows to improve the perceived performance.

4.6.4 Efficient Event Handling

In an interactive web application, there is lots of event listeners and handler functions. For example, a list of dozens or hundreds of items might have one or even several event handlers for each item in the list. This obviously becomes a burden especially in mobile devices with limited processing power and memory.

One way to minimize event listeners is to use event delegation [4]. In event delegation we attach only one event listener to a parent element of the elements that we want to listen for events. Then in the parent event handler we check the target element of the event and handle the wanted functionality based on the target.

One other optimization for touch screen is to use native touch events instead of traditional mouse events such as click. Mobile browsers typically have a delay or 300 milliseconds after a touchstart event until the the click event is fired (citation needed). This is because the browser waits if the user is doing a double tap instead of a single tap and a delay is needed before a double tap can be excluded. If we bind our event handlers to the touch events instead of click events, we can immediately dismiss this delay altogether and make the UI (?) components a lot more responsive.

We used event delegation and touch events, for example, in the main navigation of the conference application to get the best performance and responsiveness in changing the page views.

4.7 Performance Analysis

We made a quantitative analysis of the conference application performance by using two different tools: YSlow and Page Speed. These tools analyze the web performance practices of a web page and provide optimization guidelines. Many of the rules used in these tools are derived from or based on the guidelines defined by Souders [2, 3] and specified in Section 1.4.

4.7.1 YSlow

YSlow is a website analyzer originally developed by Steve Souders. It checks the website against the rules defined in Section 1.4.

!Fixme **Add YSlow results screenshot** Fixme!

4.7.2 Page Speed

Page Speed (citation needed)is an open-source project by Google for analyzing and optimizing web site performance best practices. We used the Google Chrome browser extension to analyze the conference application against the performance rules defined in Page Speed. The results are pictured in Figure 4.1.

We were very happy with the Page Speed score of 92 out of 100. A lot of the performance rules analyzed by Page Speed are similar to the guidelines listed in Section 1.4, but there are also additional rules.

The only real problem in the score was the 'Optimize Images' rule. We had not optimized the images used in the application, but instead used the images provided by the designers. Going further, we could have saved a lot or bandwidth by optimizing the images with tools such as Pngcrush[2].

Of the other notes in the results, 'Defer parsing of JavaScript' could have been avoided by adding a 'defer' attribute to all the script tags in the document. The reason for this rule is that scripts block page rendering as defined in Section 1.4. However, since we followed the guideline 'Put Scripts at the Bottom', this rendering issue is avoided. The only script in the document head was Modernizr, which must be included before the page is parsed because it creates the essential HTML5 tag support for older browsers and must do so before the tags are parsed.

The 'Minify JavaScript' note was probably due to the Handlebars templating library not being minified. All the JavaScript libraries were included in their minified form, but Handlebars library was only available unminified. We also did not want to minify it ourselves to avoid breaking any functionality. All other JavaScript files were minified and concatenated to avoid extra HTTP requests.

The 'Minify CSS' note was not seen as important since CSS compression does not yield big improvements and because the CSS files were already Gzipped delivered. The 'Remove query strings from static resources' note means that query parameters like '?123' should be removed from the end of the URLs (?) because they might not be cached in some proxies. We did not change this because the query strings in the static assets were an essential part of our caching strategy.

[2] `http://pmt.sourceforge.net/pngcrush/`

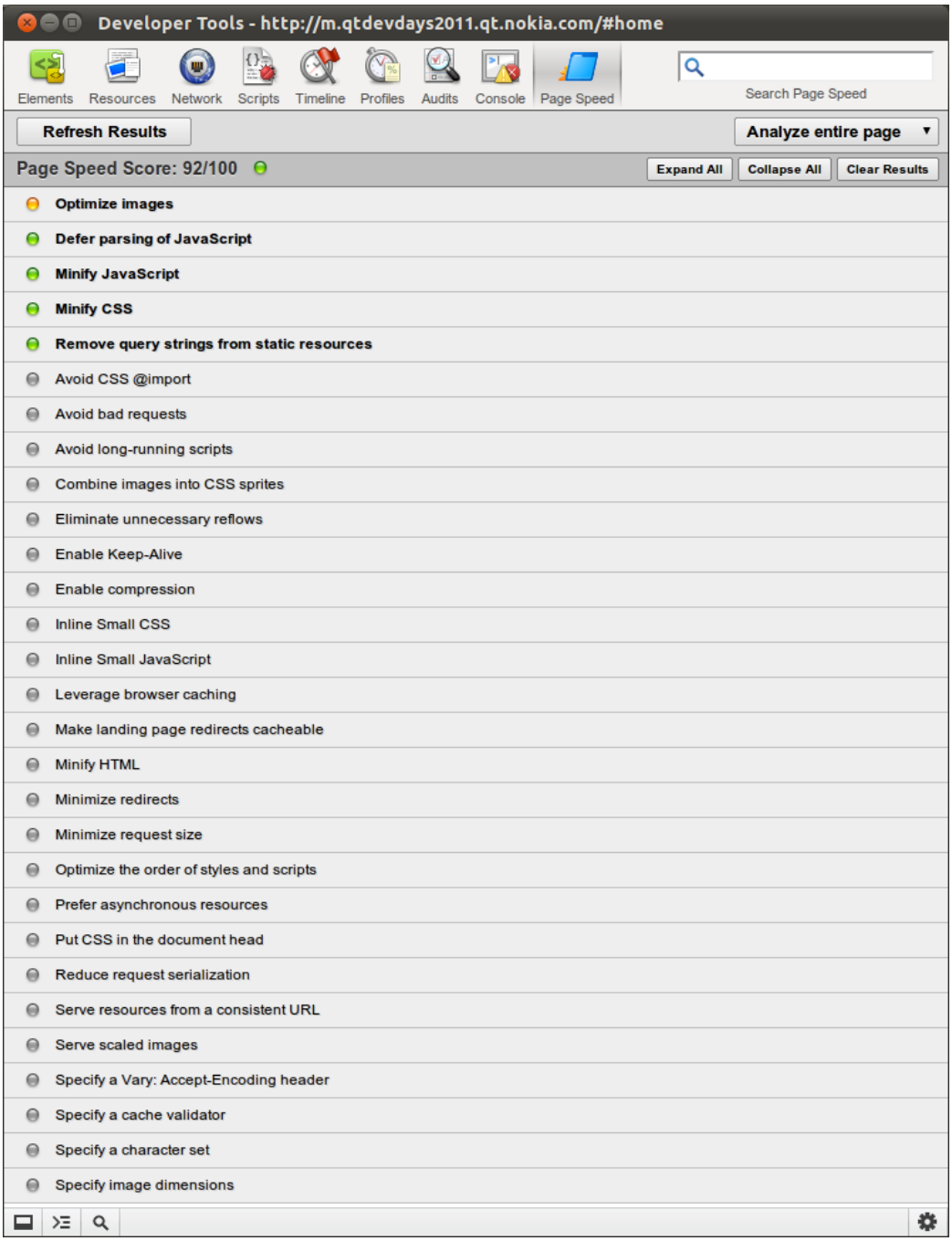

Figure 4.1: Page Speed results for the conference application.

Chapter 5

Discussion: Bright Future Ahead for HTML5

Bibliography

[1] Charland, A., and Leroux, B. Mobile Application Development: Web vs. Native. *Communications of the ACM 54*, 5 (2011), 49–53.

[2] Souders, S. *High Performance Web Sites*. O'Reilly Media, 2007.

[3] Souders, S. *Even Faster Web Sites*. O'Reilly Media, 2009.

[4] Zakas, N. C. *High Performance JavaScript*. O'Reilly Media / Yahoo Press, 2010.