Challenges for Web Analytics Applications on Mobile Platforms

WILLIAM ZHANG

KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2013

Challenges for Web Analytics Applications on Mobile Platforms

W I L L I A M Z H A N G

DD221X, Master's Thesis in Computer Science (30 ECTS credits)
Degree Progr. in Computer Science and Engineering 270 credits
Royal Institute of Technology year 2013
Supervisor at CSC was Jeanette Hellgren Kotaleski
Examiner was Anders Lansner

TRITA-CSC-E 2013:015
ISRN-KTH/CSC/E--13/015--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

Challenges for web analytics applications on mobile platforms

The mobile industry has seen tremendous growth over the last decade, with both mobile hardware and software rapidly improving. This opens up new possibilities for mobile applications and allows them to expand into areas that were previously reserved for desktop applications. One of these fields is web analytics, which Vizzit wishes to explore by creating a prototype mobile web analytics application.

This master thesis project investigates whether mobile devices are a suitable platform for web analytics applications, and what challenges are involved, both from a general, top-down perspective, as well as a more detailed developer perspective.

The results of the investigation suggests that it is definitely possible to create a mobile based web analytics application, but not without challenges and limitations. For example, tablets proved to be much more suitable than smart phones because of their larger displays. One must also consider crucial factors such as the application type (native or web-based) and target platform and their implications. For example, JavaScript is used for almost all mobile web development, but there are some limitations and challenges posed by the language itself. Finally, for data-driven applications such as web analytics programs, efficiency problems and usability issues also arise when fetching large amounts of data remotely.

Sammanfattning

Utmaningar för webbanalysapplikationer på mobila plattformar

Mobilindustrin har upplevt en enorm tillväxt det senaste decenniet, med förbättringar inom både hårdvara och mjukvara. Detta gör det möjligt för mobila applikationer att expandera in i nya områden som tidigare var exklusiva för skrivbordsapplikationer. Ett av dessa områden är webbanalys, inom vilket Vizzit önskar att undersöka möjligheten för en mobil webbanalysapplikation genom att utveckla en prototyp.

Detta examensarbete kommer att analysera huruvida mobila enheter är lämpliga för webbanalysapplikationer, och vilka svårigheter som detta medför, både generella och utvecklingsrelaterade utmaningar.

Resultaten av analysen antyder att det är fullt möjligt att utveckla en mobil webbanalysapplikation, men inte utan vissa utmaningar och begränsningar. Till exempel visade det sig att surfplattor var mycket lämpligare än smart phones p.g.a. att de hade större skärmar. En annan utmaning är valet av applikationstyp (webbaserat eller nativ) och målplattform. Eftersom JavaScript även används för mobil webbutveckling, utgör språket i sig en ganska stor utmaning, och ger upphov till en rad svårigheter och begränsningar också. För dataintensiva applikationer som webbanalysprogram finns det även problem med prestanda och användbarhet då stora mängder data måste hämtas från en fjärrserver.

Table of Contents

1 Introduction

During the last decade, the mobile phone has evolved from being a simple tool used mainly for voice communication to a platform for complex applications. These range from pure recreational applications to full-fledged business applications. In fact, Gasimov et al. (2010) claim that "*Along with digital wallet and keys, mobile phones have become one of the three things people generally carry with them*" (p. 78).

In recent years, tablets such as the iPad have also gained notable popularity, and together with smart phones, they fuel the rapidly expanding mobile application market. With this trend in mind, Vizzit wishes to evaluate the suitability of a web analytics application for mobile devices, as well as the challenges involved.

2 Background

In this section, a brief background of Vizzit will be given, as well as an introduction to basic concepts and terminology concerning web analytics and mobile development.

2.1 Vizzit International AB

Vizzit AB is a privately owned, Swedish company which specializes in web analytics. It was founded in 2002 and has since built a large customer base, including many Swedish government agencies and authorities.

Vizzit currently provides two main products, V2 and WebMaster, both of which are statistical tools, but with different purposes. V2 is a traditional statistics tool that provides analytical reports of a web site, presenting various types of statistical data such as page visits, page views and periodical trends. WebMaster on the other hand, focuses on so-called "meta-statistics" for the site. Examples of this can be the date and time of when a page was edited, the name of the person that did the edit and the duration of the edit.

2.2 Terminology

- HTML5: the latest revision of the HTML markup language. It introduces many new features including improved multimedia and graphical support, improved usability features such as drag and drop, and web storage.
- CSS3: the latest revision of the CSS style sheet language. As with HTML5, CSS3 introduces new language features that simplify more advanced aesthetic effects such as box shadows, gradients and rounded borders. It also provides built-in support for animations such as fading and transitions as well as simple 3D-effects.

- JavaScript: a dynamically typed scripting language that supports multiple programming paradigms. It has historically been used mainly for simple animations and user interactions on web pages, but in recent years, it has evolved into a more comprehensive role with the advent of AJAX and increasing complexity of web applications. Currently, it is one of the most popular web programming languages and the *de facto* standard client side scripting language.
- AJAX: an acronym for *Asynchronous JavaScript And XML*. AJAX is a set of web development techniques used to enable asynchronous communication between client and server. It is typically used in cases where parts of a web page need to be updated (e.g. a progress bar), but at the same time, a full page reload is redundant since the rest of the page remains the same.

2.3 Web analytics

Web analytics is a relatively new field that emerged after the introduction of the first web browsers during the mid-nineties. During the Internet boom, the web analytics industry experienced a rapid growth of 200%, but shrank by 7% when the Dot-com bubble burst in 2000-2001 [16]. It has since then recovered, and even expanded into new fields such as mobile web analytics. Nearly all web analytics is based on two methods of data collection, web log analysis and page tagging [20]. This section will provide a brief overview of these.

2.3.1 Data collection methods

Web log analysis is the oldest method, relying on web server logs to obtain the data. Most web servers record the web traffic in these logs, which typically contain detailed information about each web request, such as the type of resource that was requested, the date and time of the request, and information about the computer that made the request. The most common log format is W3C's (World Wide Web Consortium) Extended Common Log Format [1], in which the information in one log entry is organized into *fields*; some common fields are listed in the table below:

Field name	Description
date	The date of the transaction.
time	The time at which the transaction completed.
c-ip	The IP address of the client making the request.

cs-method	The type of the request (either GET or POST).
sc-status	The HTTP status of the request, e.g. 200 for OK or 404 for page not found.
cs(User Agent)	The user agent (i.e. browser) that made the request. Also includes information about the operating system of the client.
cs-uri-stem	The server stem, which is usually the part in the URL that comes after the domain (but not including the query part).
cs-uri-query	The query part of the URL, which typically consists of GET query parameters.

Table 2.1: Overview of some common ECLF log fields.

An alternative method of data collection that has gained popularity during recent years is page tagging. One of the reasons is that it is more convenient than log file analysis, especially for companies that do not have access to their own web servers[1]. Furthermore, because JavaScript is used, page tagging can provide features that log-based measurement cannot, e.g. screen size detection and measurement of events for specific HTML elements. The basic method is to inject a bit of JavaScript code into the web page, which is activated when the page loads. At this point visitor data is collected and submitted to a special data collection server. The main disadvantage of page tagging is that it has a small impact on performance, since each page visit involves an extra DNS-lookup, as the IP-address of the data collection server must be obtained. The script itself must also be parsed by the user's browser. Consequently, page load times are slightly increased.

2.3.2 Common web analytics concepts

Web analytics can be daunting for newcomers due to the large amount of technical terms and other jargon. This section will explain some of the most common terms and concepts.

- Page: an analyzable unit of content that can be requested, thus counting as a page view. Although most commonly web pages, a "page" can also be a media file, a document or an AJAX-requested resource.

[1] This is usually the case for companies that use third-party hosting services, in which case the servers are hosted at data centers.

Note: for the rest of this section, the term "page" will refer to the unit described above, unless otherwise specified.

- Referrer: the page that originally generated the request for the current page. A referrer is usually identified by its URL. It is very common to distinguish between *internal* and *external referrers*; the former being a page within the web site, while the latter is a web page outside of the web site, e.g. an external link. In some cases it is also useful to distinguish *search referrers*, which are referrers whose URLs are generated by a search engine.
- Page views: the number of times a page has been *successfully* requested. For example, if a requested page is missing, or there was an internal server error, the request is not considered successful, and is therefore not counted as a page view.
- Page visits: this metric is often confused with page views; the difference is that a page visit is really a visit, i.e. if the user reloads the page, it still counts as one visit, but two page views.
- Session time: the duration of a page visit.
- Unique visitors: the number of unique visitors within a specific period. The most accurate method of keeping track of a unique visitor is through user authentication, but since most websites do not require users to log in, cookies are often used instead. Unfortunately, cookies are often deleted by users (and in some cases even blocked), which affects the accuracy of this method.

2.4 Mobile applications

2.4.1 Mobile platforms

The mobile market is currently dominated by a few major platforms. In a 2008 study by Hammershøj, Sappupo and Tadayoni [8], these were identified as:

- iOS (Apple)
- Android (Google)
- Symbian (Nokia)
- Blackberry (RIM)
- Windows Mobile (Microsoft)
- WebOS (Palm)

However, in the rapidly evolving mobile industry, the market shares have changed; in a more recent report from Gartner [15], Symbian has lost considerable market shares to mainly Android and iOS, while WebOS is not even among the top six anymore:

Operating system	Market share Q2 2012 (%)	Market share Q2 2011 (%)
Android	64.1	43.4
iOS	18.8	18.2
Symbian	5.9	22.1
Blackberry	5.2	11.7
Bada	2.7	1.9
Windows mobile	2.7	1.6
Others	0.6	1.0

Table 2.2: Comparison of worldwide market share of mobile operating systems between Q2 2012 and Q2 2011. Source: Gartner [15].

2.4.2 Types of mobile applications

Mobile applications are typically divided into two categories, native applications and web applications. This section will provide a quick overview of the key characteristics of each category.

2.4.2.1 Native applications

Native applications are what people mostly refer to when speaking of mobile applications. They are typically distributed through so-called "app-stores", such as Apple's *App Store* or Google's *Play Store*, and are installed on the memory card of the device. Native applications are developed specifically for a certain platform, and are written in languages such as Java, Objective-C or C++, making them incompatible with other platforms.

2.4.2.2 Web applications

In contrast to native applications, mobile web applications are not bound to any specific platform, since they are executed within the browser. Instead of downloading them through an app-store, they are usually accessed directly through a URL; in other words, no installation is needed. Web applications are developed using the standard web development technologies, i.e. HTML, CSS and JavaScript.

2.4.2.3 Hybrid applications

Hybrid applications aim to avoid the inherent disadvantages of both native and web applications. A common approach is to create a native application that acts as a wrapper and contains a Web view component that has the same rendering engine as a browser. The rest of the application is essentially a web application. However, hybrid applications are still

typically slower than native applications since it needs to translate the JavaScript function calls into native.

2.4.3 Mobile development frameworks

As mobile applications grow in complexity, mobile development frameworks are becoming an increasingly important tool for developers, as they help reduce development time and effort considerably. These are mainly used for mobile web development, mainly due to the diversity of web applications. For native applications targeted at a single platform, these frameworks are rarely needed. Instead, so-called software development kits (SDK) provided by the manufacturers provide most functionality in the form of library functions and components. This section will provide a brief overview of some popular frameworks.

2.4.4 jQuery mobile

jQuery mobile is one of the most popular mobile web development frameworks, and builds on top of the jQuery library. It facilitates mobile web development by providing an additional layer of abstraction, hiding platform- and browser-specific details. It also includes a large number of pre-made mobile UI components that can be used out-of-the-box, as well as simplifying touch-based input.

jQuery mobile applications are written exclusively in HTML5, CSS3 and JavaScript, and can be run on all major mobile platforms, including Android, iOS, Blackberry, Symbian and Windows Phone 7. This is a popular alternative for developers coming from a jQuery background.

2.4.5 Sencha Touch

As with jQuery mobile, Sencha Touch uses HTML5 and CSS3 to enhance user experience, and provides a large number of user interface components. It is an extension of the Ext JS framework, which is a JavaScript library similar to jQuery. One of its key strengths is that it encourages the use of the MVC design pattern, which improves code structure. It also provides a native packaging feature that allows for the creation of hybrid applications. A key disadvantage of Sencha Touch is its size, which is 569KB if all features are enabled. In comparison, jQuery mobile has a size of 92KB[2]. While hardly an issue in most cases, larger files generally increase the time it takes to load and parse the files in the browser.

[2] The comparison was done between the minified JavaScript files of Sencha Touch 2.0.1.1 (sencha-touch-all.js) and jQuery mobile 1.1.1 (jquery.mobile-1.1.1.min).

2.4.6 PhoneGap

PhoneGap focuses exclusively on the creation of hybrid applications. A unique feature is that it is compatible with applications developed using other web frameworks, such as jQuery mobile or Sencha Touch. In other words, it is possible to "wrap" a web application developed using either of these frameworks in a PhoneGap application, and thereby gain access to features reserved for native applications.

Theoretically, PhoneGap would seem to be the perfect solution as it incorporates the cross-platform compatibility of web applications as well as having access to all the hardware features of a native application. Unfortunately, as with all frameworks, there is a performance penalty due to an additional layer of abstraction; this is quite noticeable on slower devices with CPUs below 600 MHz [17].

3 Aim

The aim of this master's thesis is to investigate whether mobile platforms are a suitable platform for web analytics, and if so, what challenges will be faced. More specifically, the following questions should be answered:

1. Are mobile devices a suitable platform not only from a technological standpoint, but also in terms of usability?
2. What are the challenges associated with developing "heavy" applications such as web analytics software for mobile platforms, and is it possible to overcome them? If so, how?

4 Constraints

The prototype application is a proof-of-concept, and therefore, features that would be present in a commercial application such as multiple user support and help sections are omitted. Furthermore, the application was developed as a mobile web application which means that it requires an HTML5 and CSS3 compatible browser. Since the application is a client-side program, it depends on a server to provide the required data. It is assumed that Vizzit can provide the required data.

Currently, the application is only available in Swedish as the majority of Vizzit's customers are government agencies and other public institutions that are based in Sweden. Also, it will not support orientation changes; the user interface is only designed for landscape mode.

5 Method

In order to identify the suitability of mobile platforms for web analytics applications and identify notable challenges associated with it, a prototype web analytics application was developed.

5.1 Development

At the beginning of the project, a number of decisions were made that lay the foundation for the rest of the project. These included the choice of platform and application type, development environment and methodology and evaluation of any third-party frameworks. It also involved identifying the target group.

5.1.1 Platform

It was decided fairly early on that the prototype application would be a mobile web application. The reason for this was mainly because web applications are cross-platform compatible and are easy to maintain. Because no hardware features are required for this application, the greatest incentive for native applications is thus rendered irrelevant. As a consequence, there is no need to choose a specific platform/operating system.

5.1.2 Development methodology

The application was developed using an iterative software development model with quick development cycles and continual dialog and feedback from the customer (i.e. Vizzit). The motives behind this are discussed in detail in section 6.3.3.3. A general concept of the process is illustrated in the figure below:

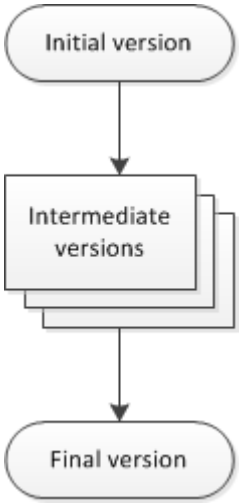

Figure 5.1: A simple illustration of the basic idea of the iterative development model. After an initial prototype is developed, the customer is continually involved and new requirements are identified with each new intermediate version.

In this case, an initial prototype of the application was developed early in the process. One of the fundamental requirements from Vizzit was that the user interface was to be integrated with the website, so that the user would interact with it when using the application instead of just passively looking at numbers and charts, as with many other web analytics applications. For this purpose, the user interface would be as minimal as possible, allowing the user to focus on the website. In the prototype application, the user interface consists of three main components:

- A menu bar which serves as the main navigational tool.
- A bottom panel that mainly provides contextual information (such as the active mode, currently selected period, status, etc.) and statistical data.
- A number of larger panels slide in and out of the screen, typically used when more detailed information is to be presented. Examples of these are the start page and tracked item views (see Figure 6.1, Figure 6.7, Figure 6.8 and Figure 6.9).

Despite numerous changes to requirements in subsequent iterations, this setup was kept throughout, and can still be seen in the final version.

5.1.3 Program design

Somewhat related to development methodology is the issue of the overall architectural design of the application. This is important because it results in good, maintainable code that is easy to extend in case future modifications or extensions are required. To achieve this, both design patterns and software design principles were used.

Design patterns can be thought of as templates or descriptions of solutions to recurring software design problems. They are language-agnostic, and are therefore not limited to any particular language. However, because of the relatively small size of the application, only two design patterns were used.

The first pattern, MVC, is an acronym for Model View Controller and is a classical pattern that has its roots in the 1970's. It was originally conceived as an architectural solution for programs with GUIs (Graphical User Interface), and is still widely used today, and has even spawned a number of variants such as MVVM (Model View ViewModel) and MVP (Model View Presenter). The basic idea of MVC is to separate concerns, and defines different types of objects to handle three distinct types of tasks:

- Models: responsible for abstracting and holding application data.
- Views: responsible for presenting the data, which can either be fetched directly from the model or supplied by the controller.

- Controllers: the "brains" of the application; responsible for handling user interaction, and updating the models and the views.

There are many variants of MVC, and the one used in this project uses one in which the second pattern, Observer, is used to automate certain aspects of the interactions between models, views and controllers. In essence, the Observer pattern is a publisher-subscriber system, where certain objects can subscribe to a set of events of another object. When an event is fired, all subscribers are notified, and can take appropriate action. In the context of MVC, the most common application of the Observer pattern is to let views subscribe to model events; when the model is updated by the controller, it fires an appropriate event, after which the associated view is notified, and can update the user interface accordingly. The benefit of this is that the controller needs to do less work since the views can update automatically, simplifying the code. The code also becomes more loosely coupled, which is highly desirable in software design as it increases flexibility and facilitates unit testing.

In contrast to design patterns, software design principles do not apply to any specific situation or problem, but are more general guidelines. The design principles used in this project are described below.

- *Minimize coupling, maximize cohesion*: actually two principles, but they are almost always used and referred to in conjunction with each other. By minimizing coupling, program components (e.g. classes) become less dependent on each other, resulting in more flexible and robust code. High cohesion is achieved by letting a class be responsible for only a few, related tasks.
- *Separation of concerns*: a program should be divided into modules with distinct, non-overlapping roles. An example of this can be seen in the MVC patterns, where each of the modules (models, views and controllers) has a unique, well-defined role.
- *Abstraction principle/DRY (Don't repeat yourself)*: these two principles state essentially the same thing, namely that every significant function should be present in only one place, and thus reduce code duplication.
- *Principle of least astonishment*: when writing the code, common coding conventions should be followed, and non-trivial parts should be commented in order to make it easier for other programmers to read and maintain.

5.1.4 Frameworks

A number of JavaScript development frameworks and plugins were used to simplify and speed up the development process. These are summarized in the table below:

Framework	Main areas of usage	Description
jQuery	DOM manipulation, event handling, remote communication (AJAX)	jQuery is an extremely popular, general-purpose JavaScript library. It eases development by introducing powerful new language constructs that allow the user to avoid many awkward or inconvenient aspects of JavaScript.
Twitter Bootstrap	Manipulation and creation of user interface components	Bootstrap is a front-end JavaScript library that provides stylish web controls such as dropdown lists, dialog boxes and carousels. All of these are very flexible and the look-and-feel can be customized.
Hammer.js	Touch-event handling	Hammer provides out-of-the-box support for almost any type of touch event, from simple swiping to more complex ones such as pinch-zoom.
Flot	Creation of graphs and charts	Flot is a powerful plotting library that allows the user to create almost any type of chart, including bar graphs, line graphs and pie charts.

Table 5.1: An overview of the different JavaScript libraries used in this project.

5.1.5 Persona

To help establish a target group, a persona was created.

Background	Per is a CMO (Chief Marketing Officer) at a privately held company. He is in his early 40s and has a Master's degree in economics. He has fairly average computer skills, but has little or no knowledge of web analytics. The company has recently handed out iPads to all managers.
Key goals	As a marketing manager, Per would like to gain more insight into how the company's website is performing. More specifically, he would like to track the development of certain pages and visitor flows, especially during marketing campaigns. However, their current statistics tool is too complicated and he finds it difficult to relate all the values, graphs and charts to the actual website. There is also no way to define custom

	visitor flows.
Usage scenario	With the new application, Per can conveniently get the latest information about the company's website from his iPad. He uses it on a weekly basis, during which he checks the latest status of his tracked pages and flows.

This persona was then used throughout the development process and influenced both the design and functional requirements.

5.2 Empirical measurements

A simple empirical study was done to analyze mobile constraints regarding bandwidth (see 6.2.5). As discussed in the next section, bandwidth is a major challenge with a mobile web analytics application because of the large amount of data involved. To illustrate the difference between caching and not caching and its impact on loading times, a comparison was made between the average loading times of both cases. The methodology was as follows:

1. Loading times were measured for the Statistics mode (see 6.1.2.1) for the start page for the website of Köping municipality (http://www.koping.se). The loading time was defined as the timespan between the first AJAX call after activating the Statistics mode and when all data had been fetched. The time was measured by using the getTime() function of the Date class in JavaScript, which returns the number of milliseconds since Jan 1st, 1970. More specifically, the time was recorded just before the first AJAX call, and was recorded again after all data had been fetched, after which the difference was calculated, yielding the timespan in milliseconds.
2. The process described above was repeated five times for when caching was enabled and disabled respectively. For each sample, a point estimation was then calculated: $\mu^*_{obs} = \bar{x}$
3. As a measurement of the dispersion of the sampled values, the standard deviation s was also calculated as follows:

$$s = \sqrt{\frac{1}{n-1}\sum_{j=1}^{n}(x_j - \bar{x})^2} = \sqrt{\frac{1}{n-1}\left(\sum_{j=1}^{n}x_j^2 - \frac{1}{n}\left(\sum_{j=1}^{n}x_j\right)^2\right)}$$

6 Results and discussion

6.1 The prototype application

Based on the persona, the primary focus of the prototype application was centered on providing the user a means to study visitor behaviors in an accessible manner. The application comprises three main areas of functionality: Actual behavior, Desired behavior and Tracked behaviors. These are designed to help the user answer the following questions respectively:

1. How are visitors currently navigating throughout the site?
2. How would I *like* them to navigate? In other words, what are my intended navigational patterns?
3. How has the navigational pattern changed over the recent period? Are the visitors actually following the intended navigational pattern?

What separates this application from other popular web analytics software is that the user interface of the application is an overlay on top of the web site being analyzed. Consequently, the user actually interacts with the web site when working with the application.

6.1.1 Start page

After a user logs in, the start page is shown, which summarizes the changes and developments since the user last logged in:

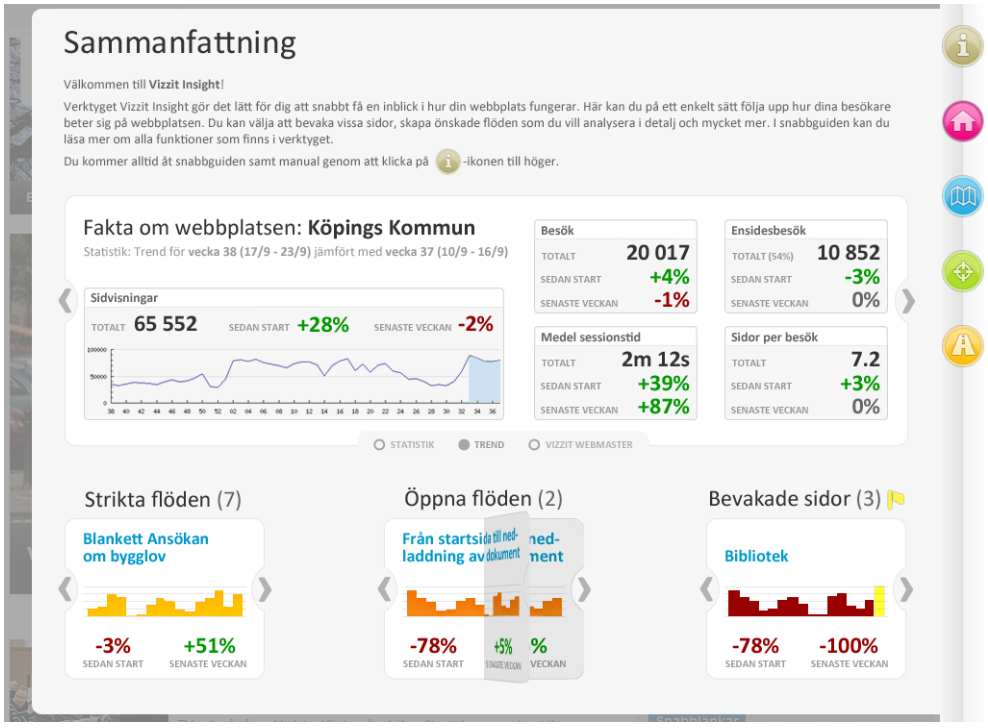

Figure 6.1: The start page, which summarizes recent trends for the website.

The lower part of the start page contains shortcuts to each tracked item.

6.1.2 Actual behavior

This category includes features that allow the user to see the current state of the web site, such as statistics and visitor flows. More specifically, it comprises the following three modes:

6.1.2.1 Statistics

This mode provides common web metrics such as page views, page visits and average session time, including information about how they have changed over the last weeks or months. Some of the information is actually integrated into the web page, such as the key values for each link on the page, which show up upon activation. The default key value is page views but can be changed. In the panel, the same key value is shown for the current page, as well as a graph indicating how it has changed over a longer period of time. When this mode is active, clicking a link on the page will cause the application to navigate to that page, updating the statistics data.

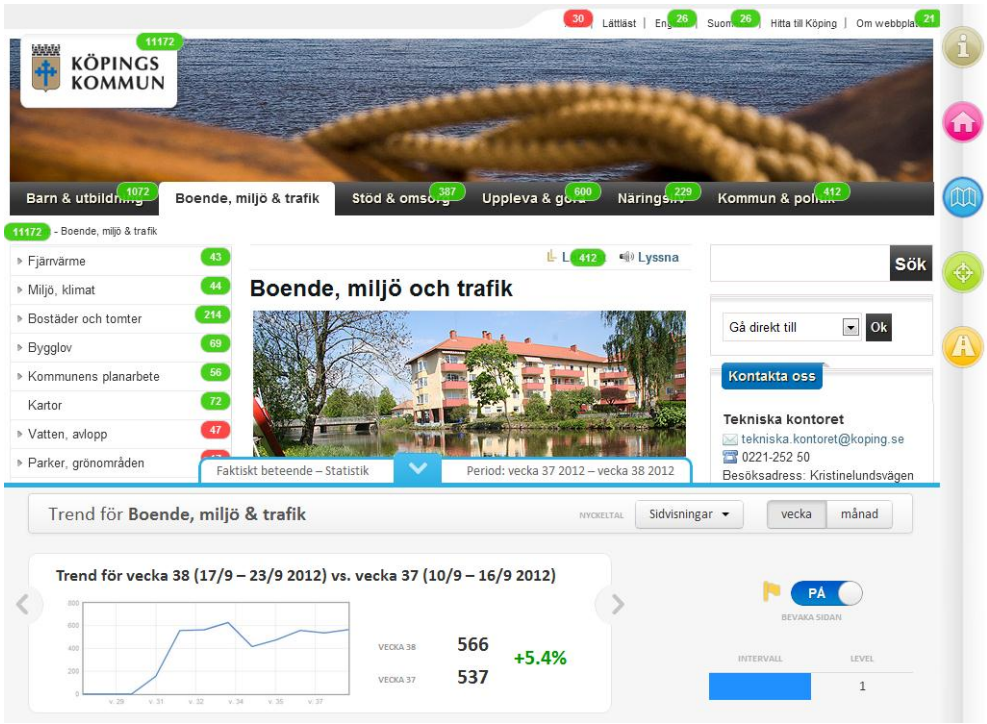

Figure 6.2: The statistics mode view. The badges for each link has a numerical value that represents the currently selected key value, while the color denotes the trend (green means increase, red means decrease).

6.1.2.2 Snapshot

Just looking at key values can be insufficient in cases where the user is more concerned with navigational patterns. The snapshot mode is designed to provide a quick overview of

how visitors come and go from a certain page by displaying the top five pages to and from that page. The arrows on the sides can be used to view older or newer snapshot data.

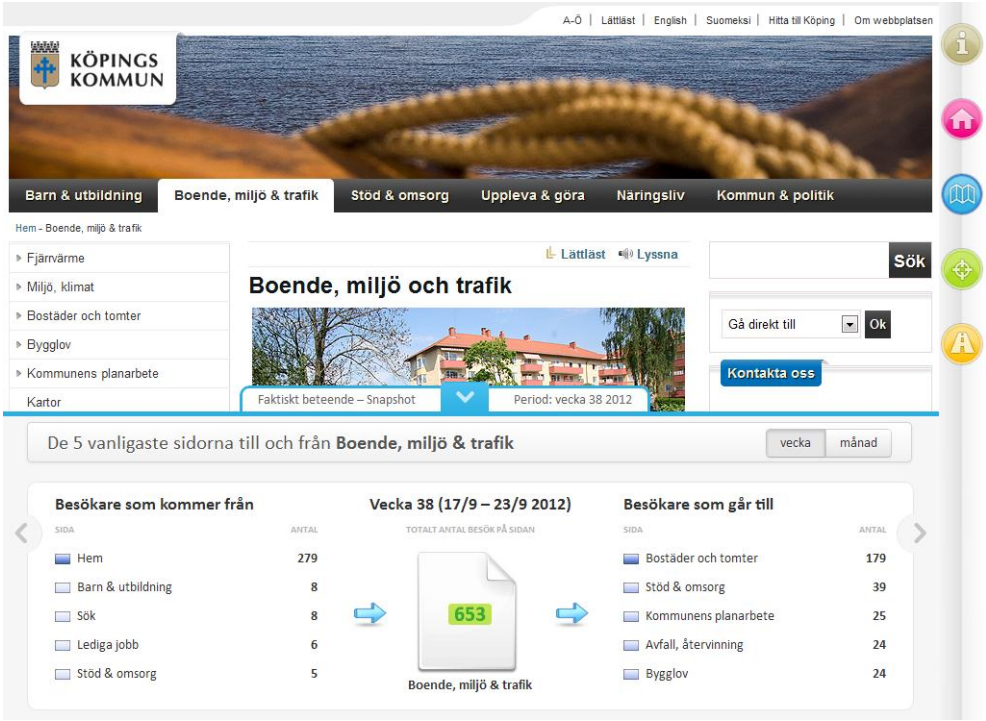

Figure 6.3: The snapshot mode view.

6.1.2.3 Navigation paths

One of the problems with the Snapshot tool is that it has a limited scope since it only involves one step to and from the selected page. For cases where the user wishes to see a more complete navigational pattern, this mode is more suitable as it provides actual paths to and from a given page. As with the snapshot mode, the period can be switched by using the arrows on the sides.

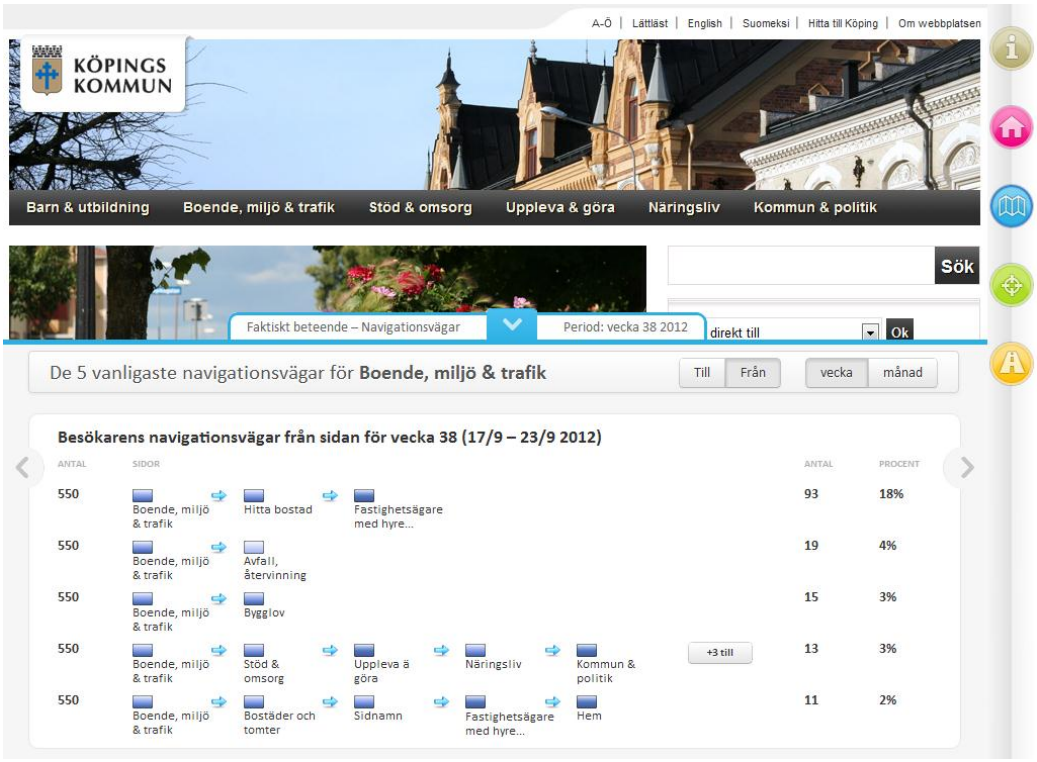

Figure 6.4: The navigation paths view. In this case, only the top five navigation paths *from* the selected page are shown. This can be toggled using the buttons in the toolbar at the top of the panel.

6.1.3 Desired behavior

Sometimes, when visitors are not following your intended navigational pattern, it can be useful to see how they deviate from it. One way of doing this is to first define the desired visitor flow, and then putting it under observation. The functions in the Desired behavior category are designed especially for defining visitor flows.

6.1.3.1 Strict flow

This is the recommended way of defining a visitor flow. The user essentially records the flow by navigating through the website. Each time a link is followed, the page is temporarily saved in the list in the panel. When the user is done, he/she can save it. However, this mode requires that the user knows each step in the flow exactly; also, the number of steps are limited to five for performance reasons (when viewing statistical data for the tracked flow later on).

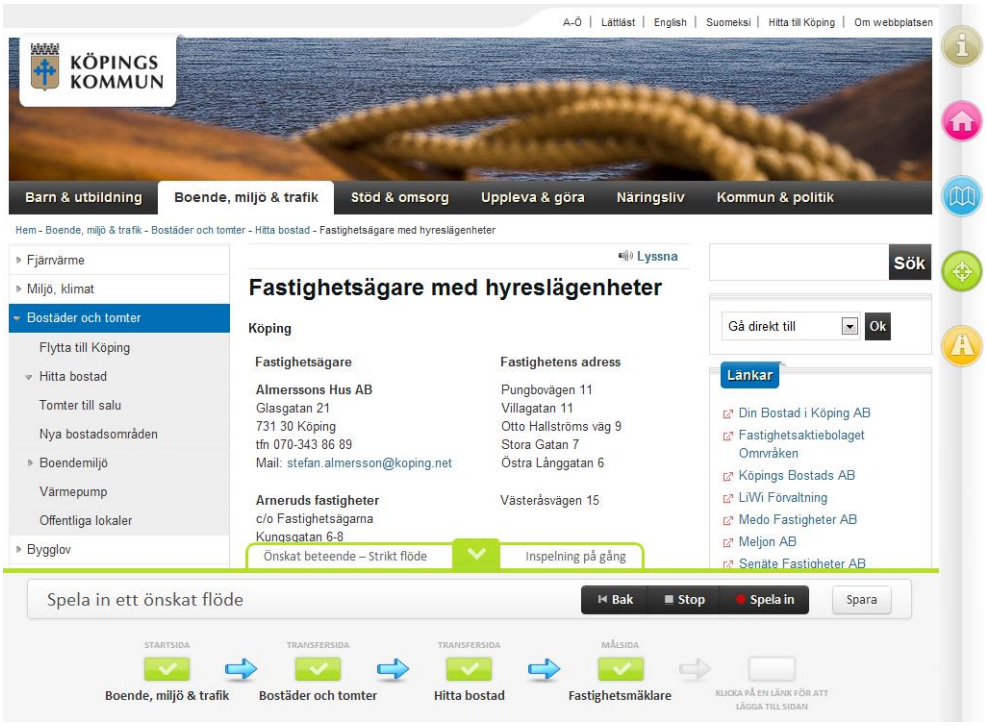

Figure 6.5: The strict flow recording view.

6.1.3.2 Open flow

This mode provides an alternative to the strict flow mode, which requires the user to know each step of the flow in advance. In contrast, this mode only requires a starting point and a finishing point (i.e. the target page).

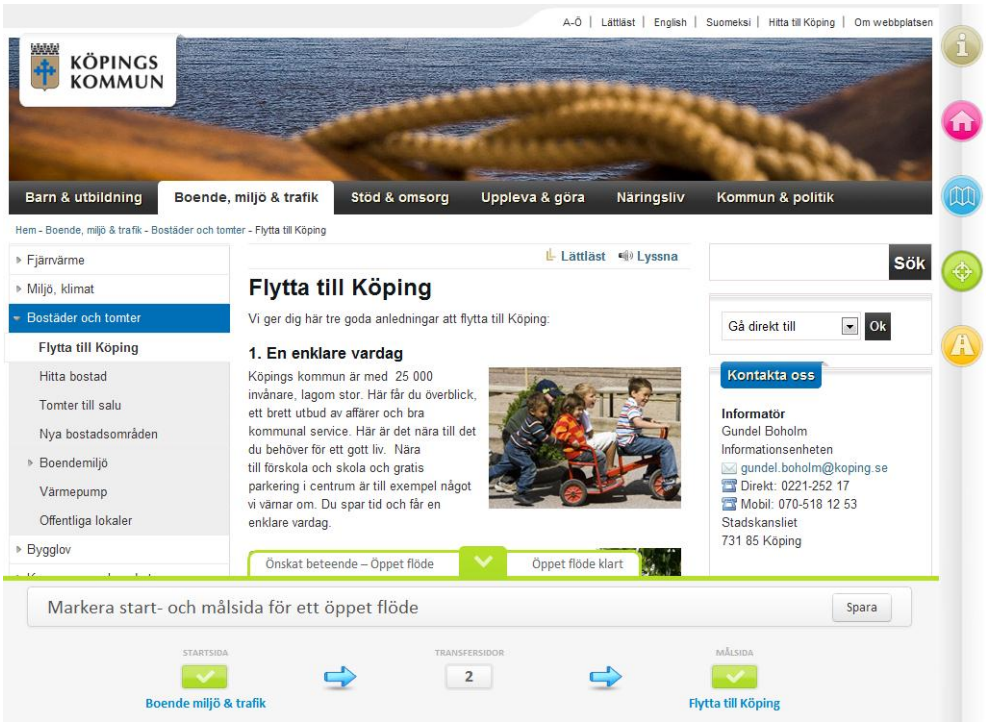

Figure 6.6: The open flow recording view.

6.1.4 Tracked pages and flows

From an analytical perspective, this group of features is perhaps the most important one, as it allows users to track the development of a certain page or user-defined visitor flow over time. It comprises three views, which correspond to the trackable items: pages, strict flows and open flows.

6.1.4.1 Tracked pages

Virtually any page on the web site can be tracked, after which the user can view detailed statistical information about it, including trends, various key values and comparisons between different periods, as illustrated below:

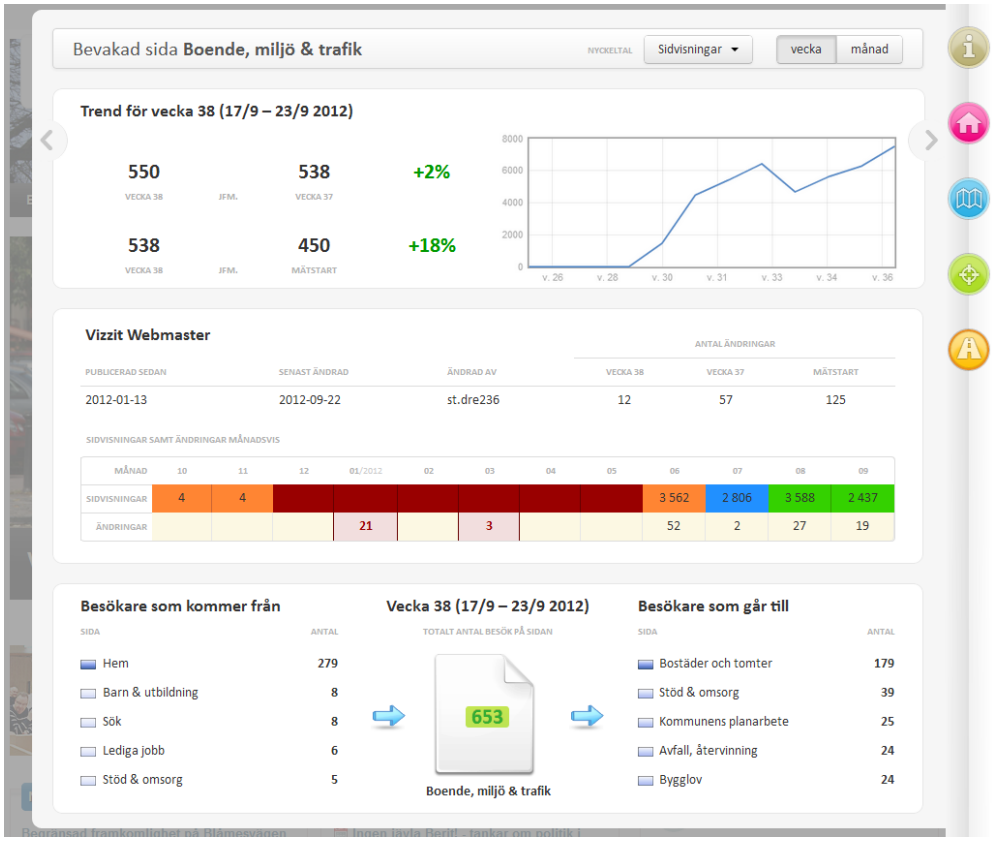

Figure 6.7: The tracked pages view.

6.1.4.2 Tracked strict flows

After a strict flow has been recorded (and saved), the user can view detailed information about it. This information is mainly concerned with the number of visitors that followed the recorded flow, the number of drops (i.e. navigated to other pages) between each step in the flow and how that has changed over time. As a convenience feature, the top five navigation paths from the starting page of the recorded flow are also provided, so that the user can quickly compare the number of visitors that followed the recorded flow with the how the visitors *actually* navigated.

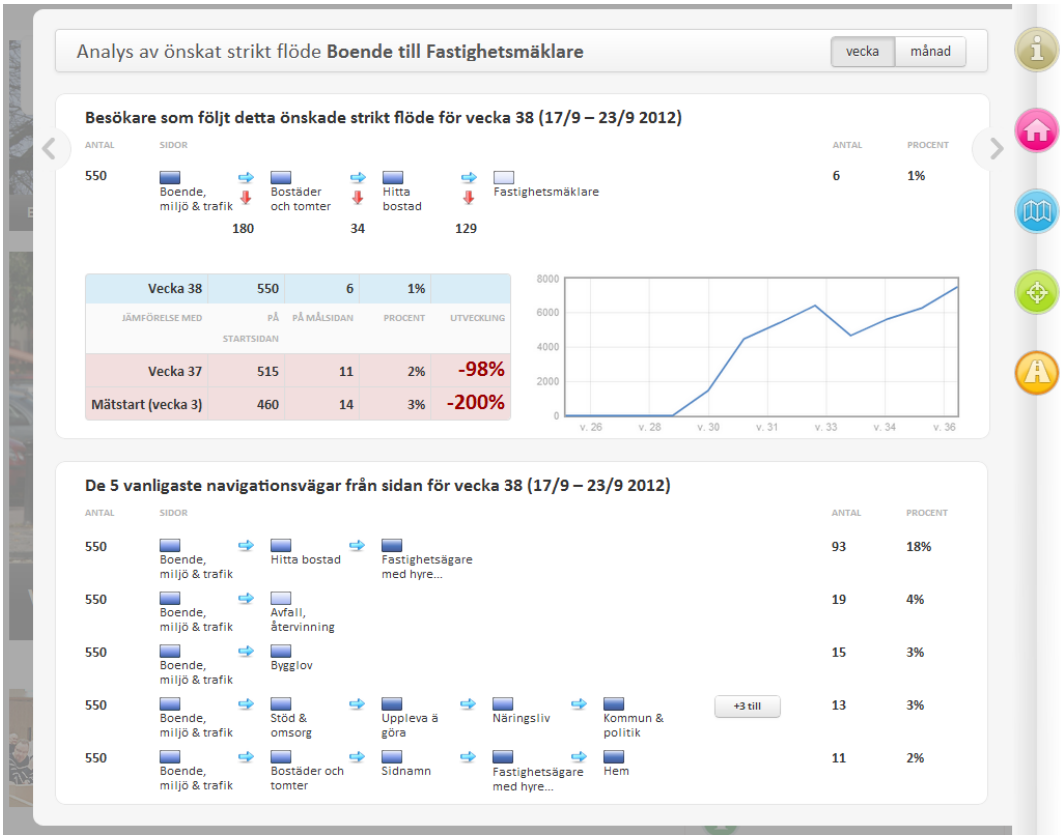

Figure 6.8: The tracked strict flows view.

6.1.4.3 Tracked open flows

Unfortunately, the amount of detailed information cannot be provided for recorded open flows as for strict flows. The reason is that because open flows can match any visitor pattern as long as they start and end at the specified pages, there is no way to know the number of drops as with a strict flow. Another consequence of this is that there can be a very large number of flows that match this pattern; and as such, only the top five (i.e. with the most visitors) open flows are shown. Although not shown in the screenshot below, the intermediate steps for each flow can be displayed by pressing the button in the middle.

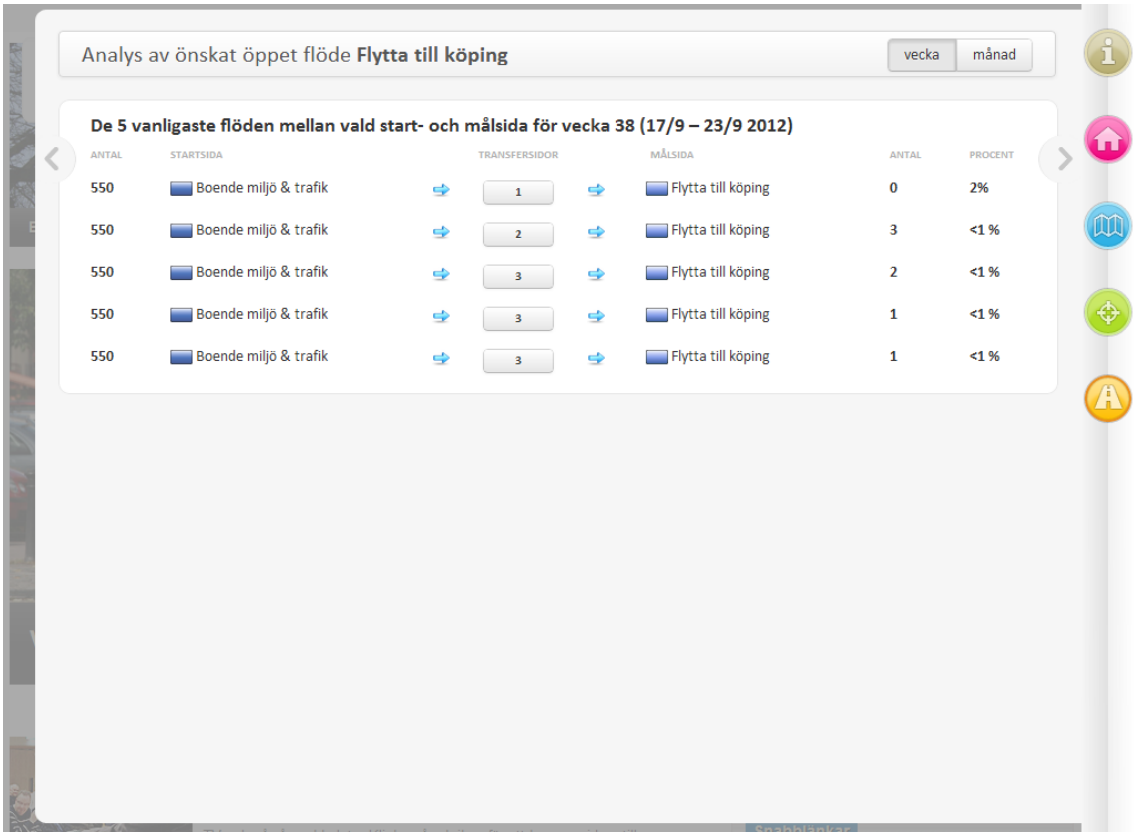

Figure 6.9: The tracked open flows view.

Before answering the questions from section 3, a clarification of what a mobile device is may be necessary. The definition is: "a small, hand-held computing device, typically having a display screen with touch input and/or a miniature keyboard and weighing less than 2 pounds (0.91 kg)." [18]. Thus, strictly speaking, in addition to smart phones and tablets, the term mobile device also encompasses devices such as portable media players, handheld game consoles and PDAs. However, in this report, the term mobile device will refer to smart phones and tablets unless specified otherwise.

6.2 Suitability of mobile devices

One of the biggest challenges that any mobile application faces is the varying display sizes of mobile devices. What may look good on a tablet with a large display may be entirely distorted on a smart phone with a much smaller display. This is especially true for web analytics applications that need to present large quantities of data, often in a graphical form such as a table or a graph. For the prototype application in this project, one of the biggest differences between it and other web analytics applications is that the entire user interface is integrated with the website. While this has some advantages, it also requires a fairly large display. On the small displays in smart phones, it turned out that there was no way to fit the entire user interface; to do so would cause the text and user controls to be so small that

they would be impossible to distinguish. On the other hand, to make the text readable, only a fraction of the interface would fit on screen (see screenshot below). In this case then, although technically possible, the prototype application is virtually unusable on smart phones due to usability issues.

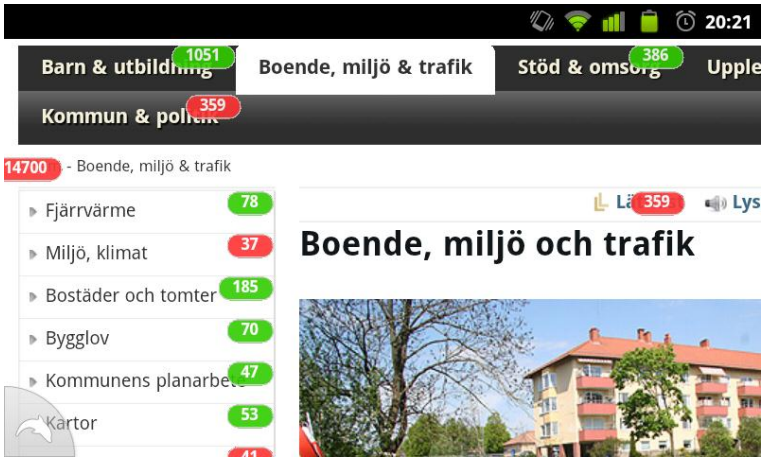

Figure 6.10: On the significantly smaller smart phone displays, only a fraction of the user interface is visible when zoomed in. This leads to an unreasonable amount of panning which is very tiresome. Compare to Figure 6.2: The statistics mode view., which is the same view, but on a tablet.

This being said, there are examples of web analytics applications that work fairly well on small displays, such as Google Analytics:

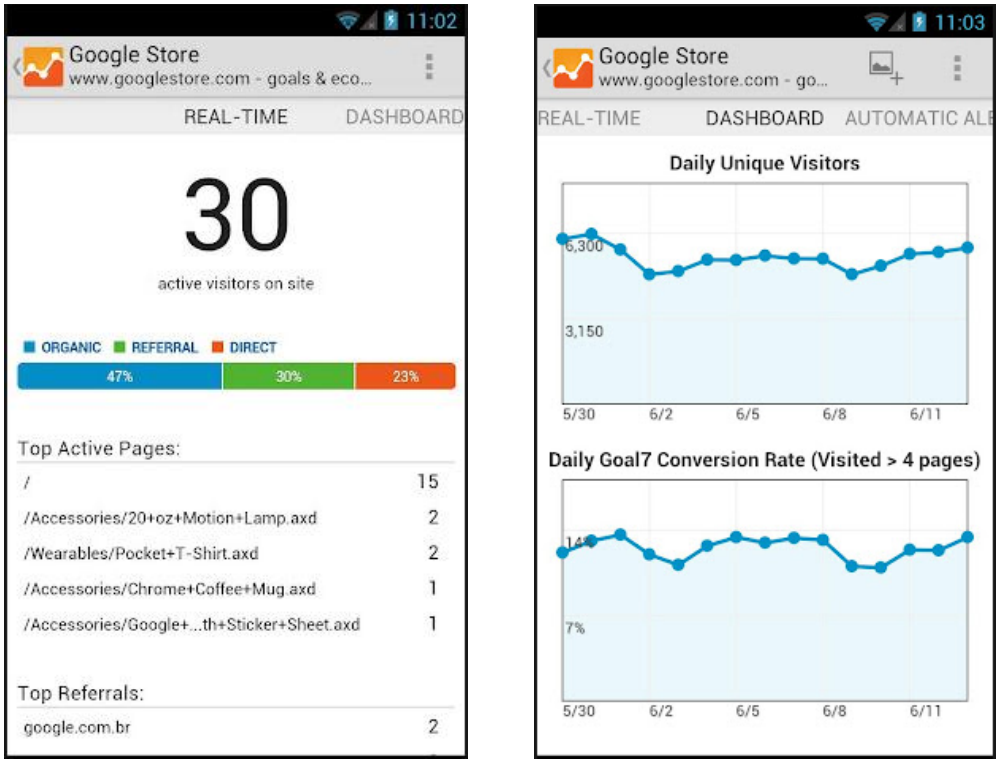

Figure 6.11: Google Analytics user interface on a smart phone. Source: Google Play.

Small displays are not the only form of constraint that determines the suitability of mobile devices. According to Murugesan and Ventkatakrishnan [9], there are a host of different constraints:

- Low display resolutions and small displays: the limited size of the display and comparatively low resolutions makes it difficult to display full-sized web pages in a mobile device.
- Low bandwidth: low bandwidth (e.g. non-3G devices) seriously limits the capabilities of heavy, data-driven applications. This is especially true for mobile web applications.
- Security risks: mobile phones usually contain more personal information than a PC, and are more prone to loss.
- Limited user-interaction capabilities: user input on mobile devices is achieved through keypads, or virtual keyboards on touch devices. Traditional text-based input is more error prone and can cause frustration.

It is important to keep in mind that, because the study was carried out in 2005, there have been subsequent advances of mobile technology that render some of these constraints less relevant. For example, Murugesan et al. argued that:
"Further, the display resolution of hand-held device screen is poorer than that of a desktop or notebook computer. Resolution of a typical PDA is 240 x 320 pixels, whereas the resolution of a typical notebook computer screen is 1600 x 1200 pixels. This makes it hard to present standard Web pages on mobile devices." [9, p. 200]

However, current smart phones typically support resolutions that rival that of a PC screen, such as Samsung's latest smart phone, the Galaxy S3 which has a resolution of 720 x 1280 pixels[3]. On the other hand, it is doubtful whether this really solves the problem, since although the content will "fit" on the screen, it will in most cases be too small to see without zooming. Consequently, it is still a constraint since content must still be customized for mobile devices.

Satyanarayanan [4] has reached similar conclusions, and has identified four fundamental constraints for mobile devices:

- *Mobile elements are resource-poor relative to static elements*: for the same cost, static elements such as stationary computers will always have more processing power,

[3] Specifications from Samsung's official web site:
http://www.samsung.com/global/galaxys3/specifications.html

mainly because mobile elements are constrained by their size and finite power source.

- *Mobility is inherently hazardous*: because mobile devices are carried around most of the time, they are more prone to damage and loss.
- *Mobile connectivity is variable in performance and reliability*: the quality of service (QoS) depends on the user's location; while high-bandwidth connectivity such as WiFi may be available at home or at the office, the user may have to rely on much slower, non-3G network technologies such as GPRS or EDGE. In addition, tablets often come in different versions, some without support for mobile network connectivity.
- *Mobile elements rely on a finite energy source*: perhaps the biggest disadvantage when compared to traditional, stationary computers. Not only does this limit the performance of mobile devices, but it also requires both hardware and software to be as energy-efficient as possible.

Almost all of the constraints mentioned so far have influenced the development and design of the prototype application, albeit in varying degrees. Considerable effort has been spent on minimizing the negative effects of these constraints, the details of which are discussed below.

6.2.1 Limited screen size

As mentioned earlier, it became evident in early tests that smartphones were not suitable for running the application due to the layout and design of the user interface (see section 6.2). Consequently, it was decided that the application was to run on tablets only, and as such, this constraint became largely irrelevant since the user interface did not suffer from any of the layout and rendering problems that the smart phones did.

6.2.2 Limited processing power

Mobile devices will always be at a disadvantage compared to stationary computers in terms of performance and raw processing power. High-performance hardware generates large amounts of heat, which usually requires the components to be cooled in a controlled fashion. For desktop computers and laptops, the most common method is through the usage of fans and heat-sinks. For mobile devices on the other hand, cooling is a bigger problem since the size constraint prevents them from using the same cooling techniques. Although studies have been carried out to investigate the possibility of active cooling in mobile devices [3, 14], most of them currently still rely on passive cooling, which limits

processing power (faster chips = more heat). Fortunately, the prototype application involves very limited calculations and thus requires relatively little processing power.

6.2.3 Security issues

Unlike computers that remain stationary at home or the office, users tend to bring their mobile devices with them. Unfortunately, this also raises some security issues, since they become more prone to theft or loss. For example, most users have their phones or tablets powered on when on the go; while this is convenient, if the device were to be stolen or lost, practically anyone could pick it up and gain access to the contents of the device, including personal information. For this reason, the prototype application requires the user to log in upon application launch. Also, because mobile devices are more prone to loss, all user-defined content such as tracked flows and pages are saved to a cloud service.

6.2.4 User-input

Because mobile devices do not have the traditional keyboard and mouse combination, common user input controls such as text boxes should be kept to a minimum, since text input is generally not as convenient on a mobile device as on a keyboard. Furthermore, in most cases standard user interface components such as checkboxes and radio buttons must be adapted since they are usually too small to be comfortable on touch screens. In fact, both Google and Apple recommend "tappable" elements (e.g. buttons, checkboxes) to be around 44-48 px, which is roughly 7-10 mm [22, 23]. This mostly applies to web applications since native applications use SDKs which already provide mobile-friendly controls. As an illustration, consider the following comparison between standard HTML controls and mobile-customized controls from the jQuery Mobile library:

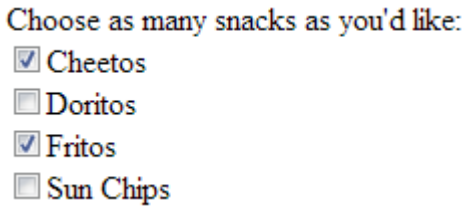

Figure 6.12: Standard HTML checkbox controls.

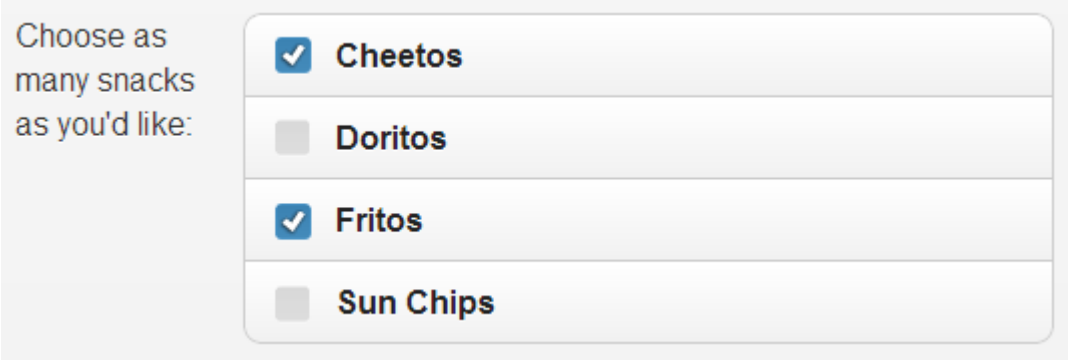

Figure 6.13: jQuery Mobile enhanced HTML checkbox controls.

In the images above, all HTML elements are portrayed with their actual size. As one can see, the standard-sized controls are much more difficult to select on a touch screen compared to the enhanced controls due to their small size.

6.2.5 Bandwidth

Web analytics applications have a single purpose, namely to present various statistical data for a website. As a consequence, these types of applications typically require more data than say, a simple calendar application. In the prototype application, all modes require some form of data to be requested from the server. This is usually not a problem for desktop computers as they often have stable, high-speed broadband connections. For mobile devices however, the connectivity can vary from high-speed Wi-Fi connections to low-speed data network connections and in some cases no connectivity at all. Also, in the latter case, network operators often impose some type of volume or price-based limit (e.g. a maximum of 1GB per month). This poses a considerable challenge as the application must be much more conservative with data requests. To minimize unnecessary or redundant data consumption, two different strategies were used. Firstly, the size of the required resources such as scripts, images and other files were minimized. Secondly, caching was heavily used to ensure that the requested data was kept around as long as possible in case it was needed again. The remainder of this section will discuss these in more detail.

Beginning with the optimization of the size of data requests, I used one of the most basic methods, which involved minimizing the size of included CSS and JavaScript files. Since the application was written entirely in JavaScript, there were a large number of JavaScript files included. In such cases, a so-called minifier script is often used to remove whitespace and comments from the included scripts. The rationale behind this is that, because JavaScript is an interpreted language and the browser reads each file as they are written,

there is no point in keeping whitespaces and comments. The same idea can be extended to CSS files, for which there are similar tools.

The final technique relates to keeping already requested data as long as possible. The majority of the user interactions involve displaying statistical data of some form. Hence, instead of requesting new data every time data needs to be presented, caching is used to reduce the amount of new requests. Since JavaScript does not allow traditional I/O-operations such as reading and writing to disk (due to security concerns), I implemented a simple in-memory cache. The basic strategy is as illustrated in this flowchart:

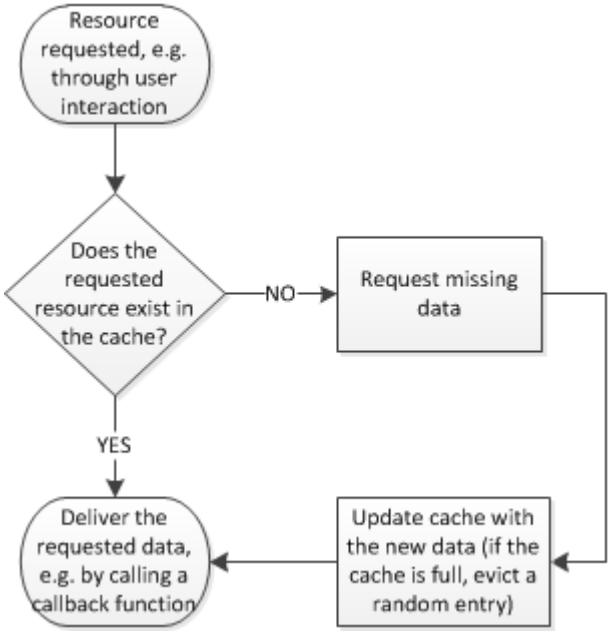

Figure 6.14: Flowchart illustrating the basic strategy when a remote resource such as statistics data is requested.

In addition to reducing data usage, caching also decreased loading times (provided the data exists in cache) since it is always faster to read data from the cache than to fetch it remotely.

To prove this, several measurements of the loading times of non-cached data versus cached data were done for the Statistics mode (see 5.2 for a more detailed outline of the method). They yielded the following results:

Non-cached	Time (seconds)
Sample 1	5.025
Sample 2	5.900
Sample 3	6.560
Sample 4	6.316
Sample 5	4.444
Point estimated average	**5.649**
Standard deviation	0.892

Table 6.1: The loading times for the Statistics mode without caching, including a point estimated average and standard deviation.

Cached	Time (seconds)
Sample 1	0.019
Sample 2	0.022
Sample 3	0.023
Sample 4	0.023
Sample 5	0.019
Point estimated average	**0.021**
Standard deviation	0.002

Table 6.2: The loading times for the Statistics mode with caching, including a point estimated average and standard deviation.

From the results we can see that when there is a huge difference in average loading time between when data is cached and when it is not. Also, for each average loading time, the standard deviation was fairly small, meaning that the values are fairly concentrated around the point estimated average.

Another optimization that could have been done is CSS sprites. When including images using CSS, the easiest and most straightforward way is usually to specify the URL to the image directly, with a rule such as the following:

```
background-image: url('/images/icon.png');
```

Each rule such as the one above makes an independent HTTP request when fetching the image. In a web application with a large amount of icons and other images, this quickly becomes a problem, since a large number of HTTP requests are inefficient and increases bandwidth usage as well as loading times. By using the CSS sprite technique, this can be avoided; instead of making separate requests for each image, all images are placed on a single image which is loaded only once, typically during the first request to the server. Subsequently, when an image is needed, a suitable offset is calculated and the image is cropped accordingly. However, due to time constraints, this was never implemented.

6.2.6 Finite energy source

Since nearly all mobile devices rely on batteries, both hardware and software must be as energy efficient as possible. An effective method of maximizing battery life through hardware is by underclocking the processors in mobile devices, since it turns out that underclocking a more powerful processor usually drains less battery than "full-clocking" a less powerful processor. In fact Apple uses this technique in almost all their smart phones [19].

From a software point of view, programs can also be more energy efficient by minimizing the amount of unnecessary operations through careful planning. In a study conducted by Miettinen and Nurminen from 2010 [15], the choice of JavaScript library also affects the energy consumption for mobile web applications, as well as the type of connectivity (3G or WLAN). However, as the authors point out themselves, the study was done on a small scale, and the results should to be further verified by other, preferable larger-scale studies.

6.2.7 Limited storage options

For native applications, reading and writing content to disk (usually an internal or external SD-card) is in most cases straightforward, with library functions are provided by the platform's software development kits (SDK).

Unfortunately, it is not as easy for web applications. Due to security restrictions, web applications (including mobile web applications) are not allowed to access system resources; consequently, they cannot perform I/O-operations such as reading and writing to disk. Traditionally, web applications have circumvented this restriction by using cookies. However, cookies were not designed for this purpose, and are not a good choice for storing data because:

- Cookies are limited to 4KB of data, which in many cases is inadequate for storing any form of complex data.

- Cookies are included in every HTTP request, which has a considerable impact on performance. They are also unencrypted, allowing anyone to read its contents.
- Cookies are often blocked or deleted by users, making them an unreliable form of storage.

Fortunately, HTML 5 provides a new feature, local storage, which addresses this very problem. It allows web applications to store relatively large amounts of data in a more persistent way using key-value pairs [21]. Although the choice is ultimately up to each vendor, the W3C Web Storage draft recommends a maximum size of 5MB per domain, which is a huge improvement over cookies. Unfortunately, local storage does not provide any kind of internal structure for organizing data; it is essentially a mapped collection of key-value pairs. This is somewhat problematic if the data needs to be organized in a more structured fashion.

6.3 Challenges of mobile development

There are a number of significant challenges involved with mobile development, which can be divided into three broad categories:

- The diversity of platforms and other technologies.
- Mobile-specific requirements and constraints.
- Development-related challenges.

This remainder of this section will discuss these challenges in greater detail.

6.3.1 Diversity of technologies

6.3.1.1 Choosing the right platform

As mentioned in 2.4.1, there are a number of different platforms on the mobile market today. These are generally incompatible with each other, greatly complicating matters. According to Gasimov, Tang, Phang and Sutanto [10], there are certain factors that should be considered when deciding the platform:

- Geographical region: the market share of each platform is highly dependent on the region. For example, in Q3 2007, Symbian was the leading platform in Asia, while it had less than 10% of the market in North America.
- Compatibility with other platforms: it is important to be aware of compatibility issues not only between different operating systems, but also different versions of the same operating system. As such, parts of the application code may have to be tailored for the different versions.

- Special hardware and software requirements: some applications might depend on a highly specialized or a proprietary feature that are not supported by certain platforms.
- Market trend: for projects with long development times (i.e. more than a year), it is a good idea to consider any major future trends. A good example that illustrates the importance of this is that Symbian has lost 75% of its market share in just one year (see Table 2.2: Comparison of worldwide market share of mobile operating systems between Q2 2012 and Q2 2011. Source: Gartner [15].

6.3.1.2 Choosing the right application type

Another crucial decision is one concerning the type of the application. As mentioned in 2.4.2, there are two main types of mobile applications, native and web applications. These two types are fundamentally different, with inherent advantages and disadvantages; it is thus wise to decide the type of application as early in the development process as possible. This section will compare these two types as well as the somewhat less common hybrid applications. It will also discuss what type was chosen for this project, and the rationale behind it.

The greatest strength of native applications lies in their ability to take full advantage of the device's capabilities. This can be both in terms of hardware and in software. In the first case, that would typically be features such as GPS, camera or onboard graphics, while the latter case usually involves creating widgets or some form of integration with stock applications, e.g. calendar or e-mail. Native applications are also capable of running offline, and can run as background processes. On the other hand, because they are usually written in a platform-specific language such as Java or C++, they are constrained to that platform[4]. This could be avoided by creating separate versions for each platform, and while it is technically possible, it is not cost-effective; unless the application requires certain features (e.g. hardware-accelerated graphics), it is usually better to consider other alternatives.

Mobile applications seek to mitigate the issue of cross-platform incompatibility. Written entirely in HTML, CSS and JavaScript, they are executed entirely within a mobile browser, which effectively acts as an abstraction layer. Consequently, web applications can run on any platform provided the browser has JavaScript enabled[5]. Another advantage is that no

[4] It is possible to circumvent this issue by using third party frameworks such as PhoneGap. However, this requires the application to be written in HTML, CSS and JavaScript, and is thus more of a solution for web applications. It also adds another layer of overhead that may not be desirable in all cases.
[5] Most current mobile web applications also make heavy use of HTML5 and CSS3. Thus, in practice, a browser that supports HTML5 and CSS3 is required.

installation is required, since the application is accessed directly through a URL. This also greatly facilitates maintenance, allowing the developers to update the application much more frequently. The Achille's heel of web applications is the inability to access the same system resources that native can freely access. They are also somewhat less performant that native applications since interpreted languages (e.g. JavaScript), are generally slower than compiled languages (e.g. Java, C++). However, the gap is rapidly closing as the mobile industry is spending huge amounts of resources on improving the performance of JavaScript engines [24].

The prototype application was developed as a web application, mainly because Vizzit wants to have the option of distributing it across multiple platforms. Although only iOS is supported at the moment, by creating the prototype in the form of a web application, it will require little or no additional work to make it run on other platforms, such as Android. Additionally, since the application does not use any special device features such as camera or GPS, the biggest advantage of native applications becomes more or less irrelevant. Finally, although it is possible that performance could be somewhat better in a native application, it is arguable whether this would result in a noticeable difference, since the only graphical effects used are CSS3 animations, which can be hardware-accelerated.

6.3.2 Mobile-specific requirements and constraints

Some of the various requirements and constraints discussed earlier (see section 6.2) also present themselves as significant challenges for mobile development, such as limited bandwidth and screen sizes. Roman Longoria states a number of rules of thumb for mobile application development [11]:

- Every pixel counts
- Every round trip counts
- Avoid data entry
- Keep you navigation model simple and clear
- Don't try to shove a desktop application into a mobile device

Most of these points have already been mentioned earlier, but they serve as a nice illustration of the importance to consider these factors. However, one important point that has not been mentioned before is the last one. Currently, the majority of mobile applications are developed on PCs and later deployed on mobile devices. For this purpose, most native SDKs include emulators. Web applications do not require any, as they can be tested directly in the browser. An important thing to keep in mind then, is that although

development is conducted on the PC, it is not a desktop application, meaning that what may look or work well on a desktop computer may not necessarily work equally well on a mobile device. In other words, there is a significant difference between desktop and mobile applications; a desktop application cannot simply be shoved into a mobile device, and *vice versa*.

6.3.3 Development-related challenges

This category of challenges relate to how different programming languages compare to each other, and their inherent strengths and weaknesses.

6.3.3.1 Native development

As stated in 2.4.1, the current mobile market is dominated by five operating systems, namely Android, iOS, Symbian, Blackberry and Windows Mobile. Each of these platforms provide SDKs, which generally include an integrated development environment, libraries and other tools. These tools often include emulators as development is mostly done on a PC, in which case it is convenient to perform some of the tests in the emulator instead of having to copy the program to the device each time something needs to be tested. However, this does not mean one can solely rely on emulators, as there can still be some discrepancies between them and the actual devices. One of the main challenges that must be faced when developing a native application is the choice of platform. Not only does one have to consider factors such as compatibility issues and market trends (see 6.3.1.1), but also the different technologies the power each platform, and their inherent strengths and weaknesses.

6.3.3.2 Web development

6.3.3.2.1 Choice of frameworks

When developing a mobile application, it is often advantageous to use one or more mobile development frameworks. Although it is certainly not required, it will involve much more work and in many cases, reinventing the wheel. For example, the jQuery framework provides a large number of convenience functions, and one of the most extensively used ones in the prototype application is the ajax() function, which facilitates AJAX calls. Using jQuery's ajax() method, an AJAX call can typically be done in a couple of lines:

```
$.ajax({
    // The URL of the server-side resource to request
    url: 'http://ajaxurl',
    // The type of the expected response. Usually XML, but in this case
    // JSONP is used in order to enable cross-domain communication
    dataType: 'jsonp',
    // Any additional data that needs to be passed to the server
```

```
    data: {
        funcName: 'LOGIN_EXTERNAL',
        args: JSON.stringify({
            'username': username,
        })
    },
    success: function (data) {
        // If the AJAX call is successful, the notify the user that
        // he/she is logged in
        // ...
    },
    error: function (xhr, status, error) {
        // If the AJAX call fails, display an error message
        // ...
    }
});
```

Without jQuery, providing the same functionality would require much more code.

However, using too many frameworks has an impact on performance. Because they are usually included as JavaScript and CSS files, the more frameworks that are used, the larger the size of each page request becomes. The challenge then, lies in finding a balance between convenience and performance. For this project, I have tried to minimize the number of libraries, while still maintaining the desired functionality. For instance, jQuery Mobile was initially used to provide touch support, but later on I found a much smaller library, Hammer.js, that provided the same functionality. While jQuery mobile has a size of 92 KB, Hammer.js has a size of just 2 KB.

6.3.3.2.2 JavaScript

"JavaScript is built on some very good ideas and a few very bad ones."– Douglas Crockford

In contrast to the vast array of platforms and technologies, the core of any mobile web application is built using HTML, CSS and JavaScript. Thus, developing a web application is easier than a native application in the sense that no platform needs to be prioritized or left out. However, although HTML and CSS are fairly straightforward to grasp, JavaScript is fundamentally different from the programming languages used for native development (e.g. Java or C++), and presents some challenges of its own.

Despite its superficial syntactical similarity with traditional imperative languages such as C++, Java and C#, JavaScript is quite different under the covers. Firstly, whereas many modern, imperative programming languages such as those just mentioned are statically-typed and focus heavily on OOP[6], JavaScript is dynamically-typed and relies on

[6] Object-oriented programming: a programming paradigm in which data and operations are organized into "objects", which are often comprised of data fields and methods. Common OOP concepts include encapsulation, coupling, inheritance and polymorphism.

prototypical inheritance. Although JavaScript is technically an object-oriented language (in fact, objects are the *only* entity), it has some unique characteristics that are quite different from the C-family languages. In fact, according to Crockford [30], "*JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java*". Consequently, programmers coming from these languages often mistakenly think that they already know half the language before realizing how different JavaScript truly is due to the syntactical similarities. The rest of this section will discuss some of JavaScript's key differences compared to the C-family languages.

One of the biggest differences is that JavaScript is dynamically typed. Compared to static languages, there are three main differences:

- Variables do not need to be declared with a type; the type is resolved at runtime instead of compile-time. For example, in Java a variable is typically declared as follows:

  ```
  int count = 5;
  ```

  In JavaScript on the other hand, the declaration would look like:

  ```
  var count = 5;
  ```

- The source code is never compiled, but is interpreted (at runtime). JavaScript is typically interpreted by a JavaScript engine, e.g. Google's V8 JavaScript engine which is used in the Google Chrome web browser.
- Objects can be manipulated at runtime. In JavaScript, this means that methods and properties can be freely added, edited and removed on the fly. This is generally impossible for static languages since they are compiled into binary files in which the code structure cannot be changed at runtime.

One of the greatest strengths of dynamic languages is that they are very flexible, which in some cases is advantageous, such as in the early stages of prototyping (when requirements often change). Another benefit is more concise code since there are no type declarations to clutter the code. For example, in C#, creating a new instance of a generic pair class could look like the following:

```
MyGenericPair<string, List<int>> pair = new MyGenericPair<string,
List<int>>("foo", new List<int>() { 1, 2, 3 });
```

The corresponding declaration in JavaScript would be:

```
var pair = { "foo", [1, 2, 3] };
```

The true intentions of the first code snippet is somewhat obscured by all the type declaration code, while the second code snippet is more clear and thus easier to comprehend.

However, there are also a number of significant disadvantages with dynamic typing. Firstly, although the omission of potentially cumbersome type declarations leads to terser code, it can also make it harder to read, which especially true when reading poorly commented code that someone else has written. Secondly, the lack of strict typing allows variables to be passed around freely. Although this gives maximum flexibility and can be very powerful when used appropriately, it also greatly increases the amount of runtime errors, since there is no compile-time type-checking. During the development of the prototype application, I wasted countless hours on strange errors caused by simple type errors and spelling mistakes.

A somewhat related issue is that, since dynamically typed languages are interpreted, the IDE can only provide limited assistance such as auto-completion and documentation lookup, which decreases productivity since more typing must be done. It is also riskier, since a common task such as renaming a variable requires a manual string search and replace. Finally, statically typed languages often discourage so-called meta-programming, whereas dynamic languages do not. An example of this is the `eval()` expression in JavaScript, which is extremely dangerous since it takes any string as input and executes it. Besides the security issues, it is also often considered bad practice to use this type of functions.

The dynamic nature of JavaScript is not the only fallacy. One of the most confusing aspects of the language is its flexibility, which is also happens to be one of its greatest strengths. It embraces multiple programming paradigms, including functional, imperative and object-oriented programming styles. However, it is not a "true" object-oriented language in the sense that there are no classes, and hence, common object-oriented concepts such as interfaces, inheritance and polymorphism are not supported natively. They can however be simulated using different patterns.

Another difficulty is that one task can often be achieved in multiple ways. For instance, a class can be simulated[7] using a few different patterns, two of which are described below:

[7] Recall that everything in JavaScript is an object; therefore, there is no such thing as a class. As a result, classes can only be simulated using functions, which also happen to be objects.

```
/* Method 1
 * The simplest pattern and most efficient pattern
 * because the methods in the prototype are shared
 * between all instances of the object. However,
 * this pattern does not support access modifiers.
 */
// The "constructor", which takes 3 arguments
var Person = function(name, gender, age) {
    // Data fields
    this.name = name;
    this.gender = gender;
    this.age = age;
}

// Public methods
Person.prototype = {
    getName: function() {
        return this.name;
    },

    setName: function(newName) {
        if(!checkName(newName))
        {
            throw new Error('Name is not valid');
        }
        this.name = newName;
    },

    checkName: function(nameToCheck) {
        ...
    }

    ...
};

/* Method 2
 * A somewhat more complex pattern that supports access
 * modifiers, but is less memory efficient because all
 * the methods are copied with each new instance.
 */
var Person = function (name, gender, age) {
    // Private data fields
    var thisName = name,
        thisGender = gender,
        thisAge = age;

    // Private methods
    var checkName = function (nameToCheck) {
        ...
    };

    // Public methods
    var getName = function() {
        return thisName;
    };

    var setName = function (newName) {
        if(!checkName(newName))
        {
            throw new Error('Name is not valid');
        }
        thisName = newName;
```

```
    };

    // Return function references to getName and setName to make them public
    return {
        getName: getName,
        setName: setName,
    };
}
```

As one can see, the two patterns above are quite different syntactically, and demonstrate the power and flexibility of JavaScript. However, for newcomers it can be rather overwhelming to evaluate the advantages and disadvantages between all the different patterns.

Usually considered as an advanced topic, closures are so fundamental and integrated with the language that any serious JavaScript developer must have a good understanding of it. Without going into too much detail, the basic idea of closures is that when a function is declared within another function, the inner function can access all the local variables in the outer function. In fact, Method 2 in the code snippet above uses closures to simulate private members; this pattern is known as the Module pattern [25]. Each of the private methods has access to the private "fields" because the fields are actually private variables in the outer function (Person).

Closures are very powerful, and can in many cases facilitate development as they reduce the need to pass around data as parameters. This is especially useful for asynchronous operations involving callback functions. However, over-usage can lead to devastating performance degradations since closures are easy to create and can sometimes become very large. An example of this is given in the next section.

Until recently, JavaScript had a rather bad reputation, and was generally not regarded as a "real" programming language among the professional programming community [5]. This can partly be attributed to the fact that in the early days, JavaScript was mainly used for simple user interactions, animations, and DOM manipulation in web pages. However, with introduction of Web 2.0, this has changed rapidly. Web applications are becoming richer and more complex, with JavaScript-based technologies such as AJAX becoming the norm. In fact, node.js, a platform for building network applications such as web servers, is written almost exclusively in JavaScript, which proves that it is not only limited to creating simple client-side programs.

In large, complex applications, a good program design is crucial. Without it, maintainability and extensibility become very difficult, and as the program evolves, the code may eventually have to be rewritten entirely. The main method of achieving this is through the

use of design patterns and other software design principles. This is usually quite straightforward in object-oriented programming languages such as Java or C#, but presents quite a challenge in JavaScript. The main reason for this is the lack of built-in support for fundamental object-oriented concepts and entities such as encapsulation and classes. However, as shown on the previous page, all of these can be simulated in JavaScript, but they come with their own drawbacks. As an example, let us return to the Module pattern example. Although encapsulation, which is an important step towards better program design, is achieved, there are two significant performance trade-offs. Firstly, because closures are used, each private method will have access to the local variables (including the other private methods) in the outer function by creating a closure. This leads to a considerable overhead. Secondly, the approach of Method 2 causes an independent copy of the entire class to be copied for each new instance of the Person class. In contrast, Method 1 takes advantage of JavaScript's prototypal inheritance by declaring the methods in the object's prototype property. This way, only one copy of the prototype is created, and all instances of the Person class will share it. Unless the class is intended to be a singleton (i.e. a class which can have only one instance), the performance loss is significant, as Method 2 consumes more memory.

6.3.3.3 Software process models

Traditional software projects often follow the classical waterfall model, for which the development process can be divided into three phases: planning, implementation and deployment. During the planning phase, various requirements and constraints are defined, and the overall software architecture is designed based on these, often using abstract modeling methods such as UML[8]. In the implementation phase, the software design is realized as a program or a set of program units. Also, each unit (i.e. component or module) is tested independently; this is known as unit testing. In the final deployment phase, the entire system is tested as a whole to ensure that it meets the requirements originally defined in the planning phase. After testing, the system is delivered to the customer.

The waterfall model is easy to understand and has well-defined, clear-cut stages. Unfortunately, the development process is rarely as tidy in practice; it is very common to discover new problems and challenges during the planning and implementation phases, which in turn may require adjustments to the original requirements and constraints. Another problem is that the waterfall model states that all requirements must be decided in

[8] UML (Unified Modeling Language) is a powerful modeling language commonly used in object-oriented software design.

the planning phase before continuing to the implementation phase. However, this is an unrealistic requirement, since customers often come with new requirements during the course of the project.

An alternative to the rigid waterfall model is iterative or evolutionary development. The general idea is to quickly create a prototype based on an initial specification, and then continually working with the customer to explore additional requirements based on their feedback. This gives much more flexibility, and the customer can continually add more requirements. However, the software structure and overall program design tends to deteriorate with continual changes. It is also difficult for managers to get an overview of the progress, since it is unfeasible to produce extensive documentation for every iteration.

Considering the dynamic nature of mobile applications, an iterative development approach is often the better choice. As Rahimian and Ramsin [12] argue in their paper, market trends change rapidly within the mobile industry, much more so than the traditional software market. This in turn causes user and software requirements to change. Instead, they advocate the use of agile methods (a form of iterative development) with quick development cycles, which allows the software process to quickly adapt to new requirements. Mobile applications are also generally smaller than desktop applications, which makes it easier to maintain a good software structure even with continual changes.

7 Conclusion

Mobile devices have become very popular during last few years, and are becoming increasingly powerful. The prototype application of this project proved that mobile devices are definitely suitable even for complex programs such as web analytics applications. However, while technically possible, there are certainly some non-trivial challenges that one must face when developing a mobile web analytics application. These challenges can be split into two broad categories: those that are inherent to mobile devices in general and those that apply only to mobile web applications.

The first category pertains mainly to various constraints of mobility, such as small displays, finite energy source and limited processing power and user input options. In this case, the constraints had varying degrees of impact on the prototype application. While limited processing power and user input options certainly had to be taken into consideration, they were somewhat less relevant in this case because the application did not involve any heavy processing, and user interaction mostly consisted of simple and intuitive touch gestures such as tapping and swiping. The display size, however, proved to be a significant

challenge. Due to the design of the user interface, the prototype application required a lot of screen space due to the large amount of information being presented. Thus smart phones were proved to be unsuitable; only tablets had displays that were large enough. Finally, this category also involves other forms of challenges. For example, when developing a mobile application, one must decide whether to make it web-based or native, and in the latter case, the platform as well. These choices are far from trivial, and each alternative has their own advantages and disadvantages. When making the decision, one needs to consider both economic factors (e.g. market trends) as well as technical factors, which typically involves analyzing and comparing each platform.

While bandwidth is strictly speaking a mobile constraint, it is much more of an issue for mobile web applications than for native applications, and especially web analytics applications due to their data-driven nature. This results in an abundance of data requests to the server, which increases traffic volume and loading times. A good solution for this problem turned out to be caching, which greatly reduced both loading times and traffic volume.

Beyond this, the most significant challenge for mobile web development is the JavaScript language itself. While the platform-specific languages (e.g. Java or C++) that are used to develop native applications have been used to create large-scale applications for years, JavaScript was until recently considered a "toy" language by the professional programming community. However, with the increasing popularity of both desktop and mobile web applications, JavaScript has become the *de facto* web scripting language, and has managed to shake off this reputation. In fact, it has already expanded beyond the confines of the browser, and is used in server-side frameworks such as node.js. Yet, writing large, complex programs in JavaScript introduces is not as straightforward as in Java or C++, much due to some rather unique characteristics of the language. For example, the lack of classes in combination with prototype-based inheritance makes it harder to implement traditional software design patterns and principles. Although it *can* be done thanks to the flexibility of JavaScript, it requires much more work, and comes with some trade-offs, especially performance-wise. This problem has been realized by both Microsoft and Google, who have introduced their own versions of the language, TypeScript and Dart. Both aim to simplify the process of creating large-scale applications by introducing static typing and traditional object-oriented concepts such as classes, interfaces and class-based inheritance. Both TypeScript and Dart can be compiled into JavaScript, making them compatible with current browsers. Whether they will be able to replace JavaScript completely remains to

see, but in the near future, JavaScript will probably retain its position as the undisputed (client-side) web programming language.

Bibliography

Journal articles

1. Sen, Arun & Dacin, Peter A., Pattichis, Christos. *Current Trends In Web Data Analysis, Communications of the ACM*, vol. 49, no. 11, pp. 85-91.
2. Sears, Andrew & Shneiderman, Ben. *High precision touchscreens: design strategies and comparisons with a mouse*, International Journal of Man-Machine Studies, no. 34, pp. 593-613.
3. Tan, F.L. & Tso, C.P., *Cooling of mobile electronic devices using phase change materials*, Applied Thermal Engineering, vol. 24, no. 2-3, 2004, p 159-169.

Reports

4. Satyanarayanan, M. *Fundamental Challenges in Mobile Computing*, Carnegie Mellon University 1996.
5. Mikkonen, Tommi & Taivalsaari, Antero, *Using JavaScript as a Real Programming Language*, Sun Microsystems 2007.
6. Meijer, Erik & Drayton, Peter. *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*, Microsoft Corporation 2005.
7. Burby, Jason & Brown, Angie, *Web Analytics Definitions*, Web Analytics Association 2007.

Conference proceedings

8. Hammershøj, Allan & Sapuppo, Antonio & Tadayoni, Reza. *Challenges for mobile application development*, 2010 14th International Conference on Intelligence in Next Generation Networks (ICIN 2010) / Second International Workshop on Business Models for Mobile Platforms (BMMP 2010), 2010.
9. Murugesan, San & Venkatakrishnan, Balasubramanian Appiah. *Addressing the challenges of Web applications on mobile handheld devices*. in *Mobile Business*. International Conference on Mobile Business (ICMB 2005), 2005, pp. 199-205.
10. Gasimov, Anar & Tan, Chuan-Hoo & Phang, Chee Wei & Sutanto, Juliana. *Visiting Mobile Application Development: What, How and Where*, 2010 Ninth International Conference on Mobile Business /2010 Ninth Global Mobility Roundtable (ICMB-GMR 2010), 2010, pp. 74-81.

11. Longoria, Roman. *Designing Mobile Applications: Challenges, Methodologies, and Lessons Learned*, Ninth International Conference on Human-Computer Interaction, 2001, pp. 91-100.

12. Rahimian, Vahid & Ramsin, Raman. *Designing an agile methodology for mobile software development: a hybrid method engineering approach*, 2008 Second International Conference on Research Challenges in Information Science, 2008, pp. 337-342.

13. Grønli, Thor-Morten & Hansen, Jarle & Ghinea Gheorghita. *Android vs Windows Mobile vs Java ME: a comparative study of mobile development environments*, PETRA '10 Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments, 2010.

14. Walsh, E & Grimes, R. & Walsh, P., *The performance of active cooling in a mobile phone* in *Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008. 11th Intersociety Conference on*, 2008, p 44-48.

15. Miettinen, Antti P. & Nurminen, Jukka K., *Analysis of the Energy Consumption of JavaScript Based Mobile Web Applications*, Mobile Lightweight Wireless Systems – Second International ICST Conference. MOBILIGHT 2010.

Web pages

16. Gartner (2012-08-14) *Gartner Says Worldwide Sales of Mobile Phones Declined 2.3 Percent in Second Quarter of 2012*, http://www.gartner.com/it/page.jsp?id=2120015 [2012-09-25]

17. Margus Pala (2011-09-02) *PhoneGap vs native Android and iPhone app performance and features*, http://marguspala.com/phonegap-vs-native-android-and-iphone-app/ [2012-09-08]

18. Wikipedia (2012-07-02) *Mobile device*, http://en.wikipedia.org/wiki/Mobile_device [2012-07-07]

19. Wikipedia (2012-09-17) *Underclocking*, http://en.wikipedia.org/wiki/Underclocking [2012-09-30]

20. Wikipedia (2012-06-25) *Web analytics*, http://en.wikipedia.org/wiki/Web_analytics [2012-06-30]

21. W3C (2011-12-06) *Web Storage*, http://dev.w3.org/html5/webstorage/ [2012-08-10]

22. Android Developers (2012-10-02) *Metrics and Grids*, http://developer.android.com/design/style/metrics-grids.html [2012-10-02]

23. iOS Developer Library (2012-09-19) *User Experience Guidelines*, http://developer.apple.com/library/ios/#DOCUMENTATION/UserExperience/Conceptual/MobileHIG/UEBestPractices/UEBestPractices.html#//apple_ref/doc/uid/TP40006556-CH20-SW20 [2012-10-02]

24. ACM Queue (2011-04-12) *Mobile Application Development: Web vs. Native*, http://queue.acm.org/detail.cfm?id=1968203 [2012-08-03]

25. Addy Osmani (2012-08-13) Learning JavaScript Design Patterns, http://addyosmani.com/resources/essentialjsdesignpatterns/book/ [2012-09-13]

26. Digital Analytics Association (2010-01-20) *Optimizing the Online Business Channel with Web Analytics*, http://www.digitalanalyticsassociation.org/blogpost/533997/89328/Optimizing-the-Online-Business-Channel-with-Web-Analytics [2012-08-20]

27. Mark Pilgrim (2012-09-12) *Dive Into HTML5*, http://diveintohtml5.info/storage.html [2012-09-14]

28. Douglas Crockford (2010-09-30) *JavaScript: The World's Most Misunderstood Programming Language*, http://www.crockford.com/javascript/javascript.html [2012-08-25]

29. mobiThinking (2012-08-13) Mobile applications: native v Web apps – What are the pros and cons?, http://mobithinking.com/native-or-web-app/ [2012-08-20]

White papers

30. Lionbridge, *Mobile Web Apps vs Mobile Native Apps: How to Make the Right Choice* 2012.

Books

31. Crockford, Douglas. *JavaScript: The Good Parts*, O'Reilly Media/Yahoo Press 2008.

32. Harmes, Ross & Diaz Dustin. *Pro JavaScript Design Patterns*, Apress 2008.

33. Sommerville, Ian. *Software Engineering* (8th ed.), Pearson Education 2007.

34. Blom, Gunnar & Enger, Jan & Englund, Gunnar & Grandell, Jan & Holst, Lars. *Sannolikhetsteori och statistikteori med tillämpningar* (5th ed.), Studentlitteratur 2009.

TRITA-CSC-E 2013:015
ISRN-KTH/CSC/E--13/015-SE
ISSN-1653-5715

www.kth.se