Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Science

JavaScript Performance and Optimization
Removing bottlenecks

Linn Rosqvist Åkerblom
Abstract

This work is a study on performance issues found when performing different user interactions, e.g. changing folder or creating a new event in the calendar, in a Android application called Briteback. The study starts with the search for test tools that can be used when searching for bottlenecks in JavaScript code. It describes different tools and their main usage in testing. It continues with identifying performance issues and bottlenecks when running Briteback on a Android smartphone. Finally the thesis concludes with the bottlenecks discovered and optimization suggestions, with a comparison of the measurements from the original code vs the optimized code.

Keywords: Briteback, Performance, Optimization, Bottlenecks, JavaScript, Jquery, Chrome Developer Tools.
Foreword

The author is a student at the distance program Programvaruteknik at Mittuniversitetet i Östersund in Sweden. This report is a part of the thesis work that gives the student an exam in the program. The examiner, Ulf Jennehag, is stationed in Sundsvall, a city which is also part of the school. The supervisor on the school, Felix Dobslaw, is stationed in Östersund and has been a great help when writing this thesis. He has given good advises through the entire process and has been a helpful support. The supervisor at the company, David Erenger, has given a lot of helpful information and shared his knowledge. This has been a great support when questions have come up and when the authors own experience hasn’t been enough. The authors husband, Joakim Åkerblom, has been a patient support for proof-reading multiple drafts of this report.
# Table of Contents

Abstract .............................................................................................................. iii

Foreword ........................................................................................................... iv

Notation .......................................................................................................... vii

1  Introduction ............................................................................................... 1
   1.1  Briteback ............................................................................................ 1
   1.2  Time plan .......................................................................................... 1
   1.3  Background and problem motivation .................................................. 2
   1.4  Overall aim ......................................................................................... 3
   1.5  Scope .................................................................................................. 3
   1.6  Concrete and verifiable goals .............................................................. 3
   1.7  Outline ............................................................................................... 3
   1.8  Contributions ..................................................................................... 3

2  Theory .......................................................................................................... 4
   2.1  HyperText Markup Language ............................................................ 4
   2.2  JavaScript .......................................................................................... 4
   2.3  Document Object Model ................................................................... 4
   2.4  JQuery ................................................................................................ 4
   2.5  Velocity.js .......................................................................................... 4
   2.6  Redactor ............................................................................................. 4
   2.7  Backbone ........................................................................................... 5
   2.8  Exoskeleton ....................................................................................... 5
   2.9  JsPerf ................................................................................................. 5
   2.10  JSLitmus .......................................................................................... 6
   2.11  Firebug ............................................................................................. 7
   2.12  Chrome Developer Tools ................................................................. 7
      2.12.1  Timeline ...................................................................................... 7
      2.12.2  Profiler ....................................................................................... 9
   2.13  Technical overview of Briteback ....................................................... 10

3  Methodology ............................................................................................... 12
   3.1  Test tools .......................................................................................... 12
   3.2  Investigation - Finding bottlenecks ..................................................... 12
   3.3  Testing possible solutions .................................................................. 13
   3.4  Validate improvement ....................................................................... 13

4  Implementation ............................................................................................ 14
   4.1  A source of multiple bottlenecks ....................................................... 14
   4.2  Console debug and log ....................................................................... 14
   4.3  Replacing Backbone with Exoskeleton .............................................. 17
   4.4  Refactoring ComposeFragment ......................................................... 17
5 Results........................................................................................................20
5.1 Console debug and log.................................................................20
5.2 Replace Backbone with Exoskeleton........................................21
5.3 Refactoring ComposeFragment.................................................23

6 Conclusions......................................................................................25
6.1 Question Q1.................................................................................27
6.2 Question Q2.................................................................................27
6.3 Question Q3.................................................................................27
6.4 Question Q4.................................................................................28
6.5 Future work.................................................................................28
6.6 Ethical aspects.............................................................................29

References..........................................................................................30
Notation

Acronyms

CDT  Chrome Developer Tools
JS   JavaScript
UX   User experience
DOM  Document Object Model
HTML HyperText Markup Language
WYSIWYG What You See Is What You Get
1 Introduction

This chapter describes the background, purpose and goal of the study.

1.1 Briteback

Briteback [1] is a small company with ~5 employees, which started in 2014. They develop their own product with the same name – Briteback. In the future the company will be referred to as the company and the product as Briteback. The development of Briteback on the other hand isn't small. Approximately 20 people are working with the product in different ways - developing, designing, marketing, etc.

Briteback is a communication tool to be used within companies to handle internal and external communication. Instead of using different applications for e-mail, chat, calendar, etc you will only use one that connects them all, Briteback.

People tend to look at their phones for new notifications or check their e-mail way too often. For every time a person takes her mind off work and looks at her phone there is a restart time before she is back in her work flow again. In a company this can resolve in a lot of work time being lost, every day [2]. Briteback aims to be a solution for this problem. The user will be able to set the time when to receive notifications from Briteback, e.g. 10:00, 12:00, or whenever is a good time for the user to receive notifications and handle them.

Other features are to receive, send and respond to e-mails while logged in through e.g. Gmail or Outlook. When a company uses Briteback they will belong to a group and all the persons in that group will be able to chat with each other through an internal chat. There will be a calendar where the users can create new events/meetings, handle created events/meetings and even start a meeting request – where the invited users are able to answer in the calendar which suggested meeting appointments suits them.

One of Britebacks competitors is Hop. Both Hop and Briteback can be used with different, already existing accounts such as Gmail. The big difference is that Hop is for people that don't want emails to be strict and formal. Hop transform your emails to be more like an conversation in a chat thread. This isn't something Briteback aims for. When emails are composed or viewed, they look more like a real paper letter.

1.2 Time plan

Table 1 shows the time plan for the work that this report covers.
Table 1: Time plan

<table>
<thead>
<tr>
<th>Week</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Study the theory of the frameworks used</td>
</tr>
<tr>
<td>2</td>
<td>Study the theory of the frameworks used</td>
</tr>
<tr>
<td>3</td>
<td>Get settled at the company and search for test tools</td>
</tr>
<tr>
<td>4</td>
<td>Find performance issues and bottlenecks</td>
</tr>
<tr>
<td>5</td>
<td>Run and save performance tests on the found performance issues</td>
</tr>
<tr>
<td>6</td>
<td>Find solutions on the found performance issues and bottlenecks</td>
</tr>
<tr>
<td></td>
<td>Run new measurement on the solutions</td>
</tr>
<tr>
<td>7</td>
<td>Find solutions on the found performance issues and bottlenecks</td>
</tr>
<tr>
<td></td>
<td>Run new measurement on the solutions</td>
</tr>
<tr>
<td>8</td>
<td>Find solutions on the found performance issues and bottlenecks</td>
</tr>
<tr>
<td></td>
<td>Run new measurement on the solutions</td>
</tr>
<tr>
<td>9</td>
<td>Make the final work on the report and study</td>
</tr>
</tbody>
</table>

## 1.3 Background and problem motivation

Briteback is a project where the main part of code is written in JS, at the time of writing this approximately 52,400 lines excluding external modules. Briteback is not only developed by the persons that work on the company. A huge part of the application is developed by thesis students. The performance is always in mind when developing, but there can still be optimizations to be made. This thesis is about detecting performance issues when running Briteback as a user and find solutions for these, if possible. The time measured is only on the client side and no performance testing have been done on the server side. Script functions with a total time of 30 ms or greater will be investigated, and these are considered as performance issues. The limit of 30 ms is something the author self has decided, based on the fact that it should only take 16 ms to load a frame so it's not perceived as slow by the user [3].

At the time of writing, the performance on the client side isn't a problem, but since Briteback and its content is growing every day things might evolve to the worse. It's therefor better to have as good performance as possible from the beginning.

The first place where problems can occur is if good test tools is not found. It is important that the test tools show clear performance values. These values are to be used when comparing the old code to the new one. The next place where problems can occur is if no clear bottlenecks are found or if bottlenecks are found but they are not possible optimize.
1.4 **Overall aim**

The aim of the project is to find some distinct parts in the code that can be optimized. The best outcome is if the solution can be implemented in several parts of the existing code, so that large optimizations can be made. A user would not approve of a product that is full of yank and long loading time therefore the performance part in Briteback is important. A smooth running application is an obstacle less for adoption of Briteback.

1.5 **Scope**

The company experiences that the Android application is a bit slower than the web application, they have therefore asked the author of this thesis to focus on the Android application. Although the results may also affect the web applications performance since it shares a great amount of code.

1.6 **Concrete and verifiable goals**

The survey has an objective to respond to the following four questions:

Q1, Are there any specific test tool suitable for measuring the performance of the JS code of a web-based application?

Q2, Are there any verifiable performance issues in Briteback that can be measured with performance monitoring tools?

Q3, Are there any optimization solutions on the performance issues found?

Q4, Are there any more places where the found solutions can give better performance overall in the code?

1.7 **Outline**

Chapter 2 describes the most important technologies that Briteback use today. It also describes the libraries and methods that have been used or considered in this survey. Further chapters describe the method, implementation, result and conclusions of this study.

1.8 **Contributions**

Increased the performance when loading views by in average 33 ms and 25.5% on the affected functions, running Briteback beta 5.
2 Theory

This chapter describes the methods and technologies associated with Briteback and this study.

2.1 HyperText Markup Language

HyperText Markup language (HTML) [4] is the markup language that's commonly used when developing web pages. It's written with start- and end tags to describe elements. Browsers then read the HTML file and formats the document according to the HTML tags.

2.2 JavaScript

JavaScript [5] is a programming language, with its syntax derived from the language C. When used in web development, it's very common that it handles interaction with the user on the client side or alter displayed content on the web page.

2.3 Document Object Model

The Document Object Model (DOM) [6] is a representation of all the HTML with its tags, attributes and in which order they appear on the page. Each time content is changed, the DOM is changed and this can lead to performance issues or negative user experience (UX).

2.4 JQuery

With JQuery [6], programmers don't have to spend as much time to think of which browser the end user is using. JQuery is a JS library that has a great amount of functions to use, to make it easier to develop web applications.

When it comes to animation, JQuery isn't the best library to use since “JavaScript animation is fast. JQuery slows it down” [7]. JQuery is good in many ways but it has several design decisions that makes it suboptimal for animations.

2.5 Velocity.js

When it comes to animation it's preferable to use a JS animation library such as Velocity [7]. The syntax used in Velocity is pretty similar to the syntax used in JQuery, to make it easier for experienced JQuery developers.

2.6 Redactor

Redactor [8] is a WYSIWYG (What You See Is What You Get) editor that is used when composing a letter.
2.7 Backbone

Backbone [9] is a JS library to help the developer write code that handles the connection between model and views. Models is a representation of data that can be manipulated. If a change event is triggered by the model, the views connected to the model is updated to show the changed model. When using Backbone, the developer are freed from updating HTML manually. The module helps with the sync between user interface, logic and database actions. To be able to use Backbone a library named Underscore.js has to be included and it is also preferred to include jQuery.

2.8 Exoskeleton

An alternative to Backbone is the library Exoskeleton [10]. It's said to be both “faster and leaner” than Backbone. Here the developer can decide if Underscore.js is to be used or not, it’s not a criteria for Exoskeleton to work. To make Exoskeleton even faster, jQuery needs to be removed and replaced with the libraries Backbone.NativeView and Backbone.NativeAjax.

2.9 JsPerf

JsPerf [11] is a web based test tool where you can create test cases to compare the performance between the different cases. You simply navigate to the web resource http://jsperf.com and write your code in the text areas provided in the window. When you run your test cases you get a result for every test case with the operations per seconds. It also shows the most optimal case and the other cases compared to the best. Figure 1 shows how a completed test may look like. This test compares the fastness when triggering events with Backbone and Exoskeleton. The result shows that Backbone is 70% slower than Exoskeleton.

![Test runner](Image)

**Figure 1**: A completed test in jsPerf [12]
2.10 JSLitmus

Here we have to create our own HTML file and include a JS file containing code from *JSLitmus* [13]. Then create our own test case and run the HTML file in a web browser. Figure 2 shows a completed test in JSLitmus.
2.11 Firebug

![Firebug profiler](image)

Firebug [15] is a plug-in tool to Firefox. It has a timeline where you can see how much time the different methods use when running your program. Figure 10 shows an example of a capture in the timeline. Based on that you are able to see which method takes the longest time and may be possible performance issues.

2.12 Chrome Developer Tools

Chrome Developer Tools (CDT) [16] has support to connect a hand held device to your computer. With the device connected to your computer you can easily record profiles to get measurements when performing actions on your device in the Chrome browser.

2.12.1 Timeline

In the timeline you can record your current activities in the web browser. It records and then shows, with different colors, loading events, scripting, rendering and painting. Figure 4 shows a record made in the timeline when switching between folders in Gmail.
High columns in the timeline can be signs of performance issues. Illustrated in figure 5 we see performance issues when replying to an e-mail. There are two horizontal lines in the graph and it’s not preferable to have columns going over the
top line. Here the first four columns are above the top line and the yellow color tells us that they are a result of scripting.

![Timeline capture when replying to an e-mail](image)

**Figure 5: Timeline capture when replying to an e-mail**

### 2.12.2 Profiler

The profiler is a different tool in CDT. Here a record is made, but the result shown is only script, or HTML written in scripts. All functions called during the recorded action is presented on the screen after the recording has stopped. A profile that doesn't have performance issues is illustrated in figure 6. At the upper part, a time line is drawn but a bit different in its appearance from the timeline in figure 5. This illustrates how much work is done. Here the concern isn't on how high the chart goes, but how wide it is. In this example the columns aren't wide, so we don't have a problem.

![Record of a "good" profile](image)

**Figure 6: Record of a "good" profile**

Figure 7 shows a record of a bad profile. Here we can see that the timeline shows a wide and high chart. The wideness of the chart shows of heavy work being...
made and the highness of the chart tells us there are several functions calling other functions. In the lower part of the figure there are several colored areas. Every part is a function that has been called during the action recorded. The functions at the top to the left is the one called first, if that have functions below them, those functions have been called from within the first function. When hovering over a function we get detailed information that tell us how much self time and total time that function stands for and where the location of the specific function is found. The self time is the time the function self is responsible for creating. The total time is the time it takes to execute the function it self and the function calls from within the function.

![Figure 7: Record of a "bad" profile](image)

### 2.13 Technical overview of Briteback

Briteback works on computers, smart-phones (Android and Iphone) and tablets. It’s browser based, which means that it runs with the same JavaScript (JS) code in all these applications and works in most browsers.

Figure 8 shows a schematic view of the front end code in Briteback.

*Fragment* handles event and listeners and contains view and collections. *View* makes the fragments visual to the user. If something is changed in a collection this change is made visible in the view. *Collection* is a collection of models. *Model* tells what the object mail, calendar, calendar event etc contains. One could say that fragment is the controller in a model–view–controller pattern. *Provider* is an interface that connects the web worker to the model, regardless of what source the user is logged in from. The *web worker* is also an interface, and a separate thread that handles the connection between the provider and the email source. One web worker for Gmail, one worker for Outlook and one worker for Exchange.
The login is handled by a Grails server which speaks directly to the email sources to confirm user name and password.
3 Methodology

When doing the investigation, some test tools was used for finding bottlenecks. Possible solutions are tried and then validated again with these test tools.

3.1 Test tools

Searching for test tools was made in Google search engine with search terms “JavaScript profiler” and “JavaScript test performance”. That gave some ideas on tools to use, e.g. jsPerf, JSLitmus, Firebug and CDT. A question on the Q&A forum Stack Overflow [17] among with the results from the search showed to be helpful. In the Stack Overflow question, different people gave ideas on what they preferred when testing performance.

The hand held device that was used for testing is an Android smart phone, LG G2 mini.

JsPerf and CDT is the test tools used in this thesis. JsPerf was chosen based on its simple usage. There was no installation needed, it was available online and it had hardly no learning time. CDT was chosen partly since testing on actual hand held devices was possible. When running Briteback on an Android application it runs on Chrome browser and the fact that CDT is a plugin in Chrome makes it possible to test directly on the smart phone.

3.2 Investigation - Finding bottlenecks

Both the timeline and the profiler in CDT was used to find bottlenecks and performance issues.

Since there isn’t any specific visible yank in Britebacks application, a decision was made to record and investigate random actions. These recordings needs to be investigated to find bottlenecks. The first step in finding bottlenecks was simply to play around with the application in a phone. The action was then recorded in CDT for future investigation. If performance issues where detected in the timeline they where tested in the profiler as well.

Sometimes the time measured is a lot slower or faster than the average, so to be able to find a bottleneck multiple tests had to be made. In the profiler, 30 records of the same action was made to get a fair time measurement.
3.3 Testing possible solutions

There might be many ways to solve performance issues in a JavaScript application. Two ways have been looked at to improve the performance. One way is to change library, if the bottleneck is associated with an external library. Another way is to re-write code written by in-house developer.

3.4 Validate improvement

After changes have been made, new tests need to be done to assert that the changes have improved the performance.
4 Implementation

An attempt to optimize the code in Briteback was made by investigating the possible bottlenecks. With a lot of searching in books and on the web, different solutions as replacing different libraries or refactoring the code where considered for future use.

4.1 A source of multiple bottlenecks

Figure 9 shows a recording of replying to several recipients. Since the wideness of the function represents the total time spent, one can see that the top functions aren't the problem causers, they are simply calling other functions that take a lot of time. With a look a bit lower in the figure, one can see where the functions starts to spread into several. On the left side we have the Redactor function at the bottom, that is very wide, but with almost nothing under it self. This is a possible bottleneck. The same goes for the function in the middle lower part, Backbone as well as for the function to the right Compose._checkForExtras. The recordings with functions like these where documented for later possible optimization, an average time was calculated for each method.

These possible bottlenecks will be discussed in the coming sub-sections of chapter 4.

![Figure 9: Profile recording, reply all](image)

4.2 Console debug and log

A possible bottleneck to investigate was the profile recording from when a user replies to an e-mail.

Figure 10 shows the function ComposeFragment._checkForExtras (F1) in developer mode. The function is taking almost 20% of the entire time when clicking the button reply until the reply screen is up. That, plus the fact that
the function `Compose...ngeFrom (F2)` - called from F1, had a relatively large self time (the time the function itself is responsible for creating) made the author curious in investigate that function further.

The sources are highlighted in the figure to make a visualization of the problem. These console prints issues were resolved, simply by commenting them out in all affected functions.

All performance tests were performed in developer mode. Though when running the application in user mode these bottlenecks are not a problem since the debug messages are turned off in the user mode.
Figure 10: Zoom in on possible bottleneck
4.3 Replacing Backbone with Exoskeleton

Backbone handles the views in Briteback, hence is used in many places. The function that calls Backbone when the user wants to compose a new mail is `ComposeMetaView (F3)`. This function has a large total time, 136.5 ms and is therefore a great place to see if there could be any optimization made. As figure 1 shows, Exoskeleton is supposed to be faster than Backbone. Therefore an attempt to replace Backbone with Exoskeleton was made.

Since Exoskeleton and Backbone are closely related to each other, a declaration at one place needed to be changed to make all code using Backbone now using Exoskeleton instead.

4.4 Refactoring ComposeFragment

The functions `ComposeFragment._changeFrom (changeFrom)` and `ComposeFragment._addRecipient (addRecipient)`, are written by the developers at Briteback. This is a good place for refactoring. The test recordings have been made on replying to an email with 20 recipients.

The original code is visualized in text 1. When `addRecipient` was run, a check was made for every address if it was the user’s own address. This isn’t a necessary test since it is made in another place. This was resolved by temporary commenting out the code.

The function `changeFrom` is supposed to set the sender of the mail to the current user. The function had four if-clauses, that all check if the addresses provided are the users own. If true, another function is called that returns the users addresses. Since there where no else- or nested if-clauses this resulted in all 20 addresses were checked at least once.
The changed code is shown in text 2. A variable is set with the users own email addresses from recipients. If the variable is empty the same thing is made with sender, and the same procedure with cc and bcc. The big difference is that the first check is removed. Since _getOwnEmailAndAlias already checks if the address is the users, there is no need to make the check twice. If or when the variable aren't empty anymore, the rest of the addresses won't be checked since the program won't continue with the next if-clause.
JavaScript Performance and Optimization – Removing bottlenecks
Linn Rosqvist Åkerblom

_text: Code after optimization_

```javascript
function changeFrom(recipients, sender, cc, bcc) {
    var myEmails = this._getOwnEmailAndAlias(recipients);
    if(_.isEmpty(myEmails)) {
        myEmails = this.getOwnEmailAndAlias(sender);
        if(_.isEmpty(myEmails)) {
            myEmails = this._getOwnEmailAndAlias(cc);
            if(_.isEmpty(myEmails)) {
                myEmails = this._getOwnEmailAndAlias(bcc);
            }
        }
    }
    if(!_.isEmpty(myEmails)) {
        this.$html.find(".compose-address.from-label").text(myEmails[0].address);
    }
}
```
5 Results

This chapter show the results of the performed optimizations.

5.1 Console debug and log

As described in sub-chapter 4.2 a possible bottleneck was discovered when a user replies to an e-mail. This was due to console debug and log calls in the functions ComposeFragment._checkForExtras (F1) and Compose...nge- From (F2).

As table 2 shows, F2 had an average self time of 7.8 ms. When commenting out all the debug and log functions, the self time was almost non existing as seen in table 3.

<table>
<thead>
<tr>
<th>Table 2: Replying to an e-mail, original code</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Shortest self time</td>
</tr>
<tr>
<td>Longest self time</td>
</tr>
<tr>
<td>Average self time</td>
</tr>
<tr>
<td>Shortest total time</td>
</tr>
<tr>
<td>Longest total time</td>
</tr>
<tr>
<td>Average total time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: Replying to an e-mail, console-logs/debugs removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Shortest self time</td>
</tr>
<tr>
<td>Longest self time</td>
</tr>
<tr>
<td>Average self time</td>
</tr>
<tr>
<td>Shortest total time</td>
</tr>
<tr>
<td>Longest total time</td>
</tr>
<tr>
<td>Average total time</td>
</tr>
</tbody>
</table>

Further, all debug- and log functions where also removed in all the other functions called from F1 and F2. The change in time and percent is showed in table 4 and figure 11. F1 has decreased its total time with ~72% and F2 with ~95%.
Table 4: Replying to an e-mail, difference in measurements

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Removed log/debug</th>
<th>Change in time</th>
<th>Change in percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1 self time</td>
<td>3.5 ms</td>
<td>1.8 ms</td>
<td>-1.7 ms</td>
<td>-48.6 %</td>
</tr>
<tr>
<td>F1 total time</td>
<td>129.4 ms</td>
<td>31.4 ms</td>
<td>-93.5 ms</td>
<td>-72.2 %</td>
</tr>
<tr>
<td>F2 self time</td>
<td>7.8 ms</td>
<td>0.7 ms</td>
<td>-7.1 ms</td>
<td>-91.0 %</td>
</tr>
<tr>
<td>F2 total time</td>
<td>67.7 ms</td>
<td>3.5 ms</td>
<td>-64.2 ms</td>
<td>-94.8 %</td>
</tr>
</tbody>
</table>

Figure 11: Replying to an e-mail, difference in measurements

5.2 Replace Backbone with Exoskeleton

Table 5 shows the time using Backbone and Exoskeleton when the user clicks on the “compose” button. The function that calls Backbone is `ComposeMetaView (F3)`.

Table 5: Compose a letter, using different modules

<table>
<thead>
<tr>
<th></th>
<th>F3, Backbone</th>
<th>F3, Exoskeleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest self time</td>
<td>0.0 ms</td>
<td>0.0 ms</td>
</tr>
<tr>
<td>Longest self time</td>
<td>12.0 ms</td>
<td>2.0 ms</td>
</tr>
<tr>
<td><strong>Average self time</strong></td>
<td><strong>1.4 ms</strong></td>
<td><strong>0.3 ms</strong></td>
</tr>
<tr>
<td>Shortest total time</td>
<td>110.1 ms</td>
<td>65.4 ms</td>
</tr>
<tr>
<td>Longest total time</td>
<td>204.9 ms</td>
<td>128.8 ms</td>
</tr>
<tr>
<td><strong>Average total time</strong></td>
<td><strong>136.5 ms</strong></td>
<td><strong>73.1 ms</strong></td>
</tr>
</tbody>
</table>
The result was clear. As seen in table 6 and figure 12, the total time in F3 was reduced with 46%.

Table 6: Compose a letter, difference in measurements

<table>
<thead>
<tr>
<th></th>
<th>Backbone</th>
<th>Exoskeleton</th>
<th>Change in time</th>
<th>Change in percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3 self time</td>
<td>1.4 ms</td>
<td>0.3 ms</td>
<td>-1.1 ms</td>
<td>-78.6 %</td>
</tr>
<tr>
<td>F3 total time</td>
<td>136.5 ms</td>
<td>73.1 ms</td>
<td>-63.4 ms</td>
<td>-46.5 %</td>
</tr>
</tbody>
</table>

Figure 12: Compose a letter, difference in measurements

When performing a search for e-mails the function MailListFragment.openSearch (F5) calls Backbone. The same when creating a new event in the calendar, CalendarFragment.createEvent (F6), opening a mail, MailItemView (F7), replying to an email, ComposeMetaView (F8) and open a folder from Gmail, MailListFragment.openFolder (F9). As seen in table 7 and figure 13 the change is visible in all functions.

Table 7: Replace Backbone with Exoskeleton, difference in measurements

<table>
<thead>
<tr>
<th></th>
<th>Backbone</th>
<th>Exoskeleton</th>
<th>Change in time</th>
<th>Change in percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>F5 total time</td>
<td>150.3 ms</td>
<td>111.0 ms</td>
<td>-39.3 ms</td>
<td>-26.2 %</td>
</tr>
<tr>
<td>F6 total time</td>
<td>87.5 ms</td>
<td>52.6 ms</td>
<td>-34.9 ms</td>
<td>-39.9 %</td>
</tr>
<tr>
<td>F7 total time</td>
<td>143.9 ms</td>
<td>132.2 ms</td>
<td>-11.7 ms</td>
<td>-8.1 %</td>
</tr>
<tr>
<td>F8 total time</td>
<td>126.8 ms</td>
<td>108.8 ms</td>
<td>-18.0 ms</td>
<td>-14.2 %</td>
</tr>
<tr>
<td>F9 total time</td>
<td>170.6 ms</td>
<td>139.8 ms</td>
<td>-30.8 ms</td>
<td>-18.1 %</td>
</tr>
</tbody>
</table>
Refactoring ComposeFragment

The refactoring on two functions mentioned in sub-chapter 4.4 have been tested and measured on the function that call both these functions, ComposeFragment._checkForExtras (F4). The time measured in the original and the changed code is showed in table 8.

### Table 8: Reply all (20 recipients), original and re-factored code

<table>
<thead>
<tr>
<th></th>
<th>F4, Original</th>
<th>F4, Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest self time</td>
<td>1.0 ms</td>
<td>0.0 ms</td>
</tr>
<tr>
<td>Longest self time</td>
<td>8.9 ms</td>
<td>2.1 ms</td>
</tr>
<tr>
<td><strong>Average self time</strong></td>
<td><strong>2.9 ms</strong></td>
<td><strong>0.6 ms</strong></td>
</tr>
<tr>
<td>Shortest total time</td>
<td>68.2 ms</td>
<td>67.7 ms</td>
</tr>
<tr>
<td>Longest total time</td>
<td>147.9 ms</td>
<td>102.0 ms</td>
</tr>
<tr>
<td><strong>Average total time</strong></td>
<td><strong>91.0 ms</strong></td>
<td><strong>77.3 ms</strong></td>
</tr>
</tbody>
</table>

Table 9 and figure 14 shows the difference in time and percent after code refactoring have been made. Although the self time was small before, it's now almost non existing. The total time have increased by 13.7 ms or 15%, something that might be noticeable when replying to emails containing several recipients.
Figure 14: Reply all (20 recipients), difference in measurements

Table 9: Reply all (20 recipients), difference in measurements

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Refactoring</th>
<th>Change in time</th>
<th>Change in percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4 self time</td>
<td>2.9 ms</td>
<td>0.6 ms</td>
<td>-2.3 ms</td>
<td>-79.3 %</td>
</tr>
<tr>
<td>F4 total time</td>
<td>91.0 ms</td>
<td>77.3 ms</td>
<td>-13.7 ms</td>
<td>-15.1 %</td>
</tr>
</tbody>
</table>
6 Conclusions

It's preferable to use jsPerf before JSLitmus since the usage is easier in jsPerf. The user doesn't need to create his own HTML-file, the code is simply written or pasted into the text areas provided in the web browser. Developers in the company is using jsPerf to compare functions against each other when developing. This use is something that has made the task to find performance issues difficult, since different solutions are tested during development and not leaving many places to optimize. A lot of the performance issues is probably on the server side since there are large amounts of data exchanging in the background, but this is only a guess since this study have been focusing on the client side rather than the server side.

The `console.debug` and `console.log` are functions that is found repeatedly all over the code and these functions can be useful when developing, but when looking for performance issues it can be misleading. Since these functions are disabled in production, these issues won't affect the user. When searching for bottlenecks, a potential solution could be to write an empty function in the main-file. This function would override the console- log and debug functions. This way, the functions would run in developer mode, but since they are empty they wouldn't create any false bottlenecks.

When trying to optimize JavaScript, it's important to know what parts need to be optimized. A change of 2 ms could be a coincident in different measurements, while an improvement of 20 ms shows a significant optimization. To focus on the largest bottlenecks is a good start. But one have to consider the fact that if a minor bottleneck that is called many times get optimized, that might help the total performance a great deal.

While looking at a bottleneck you should identify whether the function is something written by a in-house developer or if it belongs to an external library. You don't want to change the code in an external library. Otherwise, consider the possibility to change the entire library into another.

An attempt to optimize Redactor would be to remove it and use another external library instead. The company have already tested several libraries and solutions, but ended up with Redactor as the best solution. Therefore, a try to optimize this part has not been made, despite Redactor being a large bottleneck.

Exoskeleton came up in a discussion concerning a replacement of Backbone. The company had at an earlier stage tried to implement Exoskeleton, but ran into code conflicts. They decided to not spend time on a replacement at that time. When the subject came up again the author was encouraged to try to make the replacement
again and maybe resolve the conflicts. Exoskeleton proved good performance measurement, but it isn't tested properly in the application. It's therefore not safe to replace Backbone entirely before several tests has been committed. Although, no problems where encountered when running the new test measurements. The company should consider replacing Backbone with Exoskeleton and make all needed test rather soon, especially since it is used often and makes a significant positive change in the performance.

Sometimes when a project grows bigger the code evolves and change. This might cause code rot, such as old code to do things that are not necessary anymore, things are done twice but in different places and code that is just hard to read and understand. Trying to refactor code could be a way to simplify further optimization.

When it comes to animations, there are no performance issues, neither with visually yank or in the time measurements. This is probably because Briteback already has implemented the library Velocity for all animations.

Automatic tests have been considered, though it seemed more of a problem to use scripts for testing rather than doing it manually. The recordings in CDT needs to be made manually, so a script that performs the action desired in the phone didn't seem viable.

It's difficult to search for bottlenecks. Large functions calling smaller functions, which in their turn calling smaller functions and so on, might look like bottlenecks. But when digging deeper, no function have a large self time. This means that there isn't much to optimize. Though there is several external modules being used, like in the example with replacing Backbone with Exoskeleton. It could be an idea to look at alternatives for other modules being used. However this could result in a huge amount of work. It's often not as easy to replace a module with another as it was with Backbone and Exoskeleton.

For instance, the library Underscore.js is something that Backbone needs in order to work. Briteback uses an alternative to Underscore, Lodash. Lodash is a library that originates from Underscore but is improved and faster. This can be compared with Exoskeleton that originates from Backbone, and also has been improved.

Something discovered under the search for optimization solutions is that you can't rely on all suggestions shared in books and over the Internet. If you find an solution that looks like it's faster and leaner, test it. Sometimes there are already written tests in jsPerf, but otherwise it's easy to write one of your own. Test everything and don't trust all you read, your code could be performing worse by implementing new code snippets that wasn't tested. For instance, tip #7 in [18] tells us to use the second argument to the $() function, i.e. the context. They say that it "is roughly equivalent to using the find() method". Something proven wrong by testing it in jsPerf, figure 15 shows that using context is 11% slower.
6.1 Question Q1

Are there any specific test tool suitable for measuring the performance of the JS code of a web based application?

There are several. Depending on what the user would like to test and in what browser she is developing the choices might be different. For this study the selected test tools are JSPerf and CDT.

6.2 Question Q2

Are there any verifiable performance issues in Briteback that can be measured with performance monitoring tools?

Yes there are. First of all, console prints gives a twisted result, showing several places with performance issues. When removing these, new performance issues are made visible.

Performance issues have been found when replying to emails with both one recipient and multiple recipients, composing a letter, performing a search for emails, creating an event in the calendar, opening an email and opening a folder.

In many of these tests there are two responsible libraries, Backbone and Redactor. They execute in a great amount of time. It's clear how important it is to choose the right library when developing, if you want your application to run smooth and without long loading time.

Redactor is something that was not investigated in this thesis. The company have chosen Redactor after tried several different libraries before it. It handle large tasks which is unavoidable, so an optimization was not possible here.

In some places the bottlenecks are a result from bad coding. Functions in the program that probably was written without any performance in mind.

6.3 Question Q3

Are there any optimization solutions on the performance issues found?
Some of the bottlenecks found was the console prints mentioned earlier. The performance is improved if these are removed. This optimization was very clear and shows a problem when searching for possible performance issues. To have a lot of debug messages is good when developing, but should considered being removed when not necessary anymore. Some debug messages can be left in the code for use in debug mode, but these should be picked with care. There is no need to have debug messages in every function called.

A transition to Exoskeleton from Backbone is a distinct improvement. Almost a 50% improvement on the affected function tested when composing a letter. The transition was also very easy and was therefore not a big problem for the company to perform. If this proves not to be a problem with the rest of the application then there shouldn't be a problem implementing Exoskeleton.

### 6.4 Question Q4

*Are there any more places where the found solutions can give better performance overall in the code?*

The console prints affects the entire application since these prints were found everywhere in the code. Almost every function inspected had a console log or debug in it. Though, this only affects the code in developer mode and not in production.

If Exoskeleton was to be used it would have impact on several parts of the program, since Backbone is used to handle views and models. In all actions recorded the result was clear, it's faster using Exoskeleton than Backbone.

### 6.5 Future work

A possible optimization that can be made in the future is to replace jQuery in Exoskeleton with the libraries `Backbone.NativeView` and `Backbone.NativeAjax`. This should, according to Exoskeleton's home page, make the code run even faster. Something not tested at this time in this project.

It's difficult to search for bottlenecks. Large functions calling smaller functions which in their turn calling smaller functions and so on, might look like bottlenecks. But when digging deeper only small function leafs are found in the call tree. A future work could be to investigate the code, to see if all functions is necessary or if they can be put together or even deleted through refactoring.

Angular is another possible library to replace Backbone with, though this library isn't as easy to learn as Backbone and Exoskeleton. The transition would therefore not be as easy as with Exoskeleton, though well worth evaluate for a suitable replacement for Backbone.
6.6 Ethical aspects

If the optimization solutions in this thesis was to be implemented in Briteback, performance improvement would lead to a more energy efficient execution of the application and in its turn lead to less energy used.

The life-span of a smartphone or a tablet battery is among other things limited by the total number of charge-cycles. A more efficient use of battery charge is thereby also a more efficient use of the costs and resources associated with the life-cycle of the battery.

The application changes discussed in this thesis does not change any user functionality of the Briteback application, and does therefore not have any other positive or negative ethical impact on the users or other actors.
References

[1] https://briteback.com/

http://en.wikipedia.org/wiki/Interruption_science

[3] Udacity, Browser Rendering Optimization, Lesson 1 – Frames
https://www.udacity.com/
Retrieved 2015-05-26

http://www.w3.org/standards/webdesign/htmlcss
Retrieved 2015-05-25

http://www.w3.org/standards/webdesign/script
Retrieved 2015-05-25


[8] Redactor WYSIWYG html editor
http://imperavi.com/redactor/
Retrieved 2015-06-15

[9] Backbone.js
http://backbonejs.org/
Retrieved 2015-05-20

[10] Exoskeleton – Faster and leaner Backbone
http://exosjs.com/
Retrieved 2015-05-20

https://jsperf.com/
[12] jsPerf, exoskeleton vs backbone events,  
http://jsperf.com/exoskeleton-events-vs-backbone-events  

[13] JSLitmus,  
http://broofa.com/Tools/JSLitmus/  

[14] JSLitmus: Demo test suite,  
http://broofa.com/Tools/JSLitmus/demo_test.html  

http://getfirebug.com/network  
Retrieved 2015-04-29

[16] Chrome DevTools Overview – Google Chrome  
https://developer.chrome.com/devtools  
Retrieved 2015-04-29

[17] Stack Overflow – How do you performance test JavaScript code?  
Retrieved 2015-05-25


[19] JsPerf, jQuery.find() vs context, W/o special case  
http://jsperf.com/jquery-find-vs-context-2  
Retrieved 2015-05-22