Haaga-Helia
University of Applied Sciences

Browser's Memory Profiling Automation

Lucy Liu

Bachelor's Thesis
Tietojenkäsittelyn koulutusohjelma
2017

Haaga-Helia
University of Applied Sciences

Abstract

13.9.2017

Author(s)
Lucy Liu

Degree programme
Tietojenkäsittelyn Koulutusohjelma

Report/thesis title
Browser's Memory Profiling Automation

Number of pages and appendix pages
57 + 10

This is a commission thesis done for Comptel Oyj's Necterday UI Platform, Team Terra. The commissioning party wanted to have automated memory profiling for browser so it would be easier for the developers to see the quality of their code and fix the possible memory leaks.

Memory leak happens when the memory that is allocated to perform some operation is not freed after that given operation is completed, resulting in that the program will eventually use up all the memory that it needs to work normally.

We can do memory profiling manually by using browser's developer tools to record the performance. During the recording user should interact with the application and repeat the same actions for multiple times to get better feedback. Developer tool is able to draw a graph where user can see how much memory has been allocated. Usually it is recommended to suspect memory leak when the graph resembles a sawtooth curve.

Automation has many advantages, but most importantly it saves developers or testers time. Especially tests that need a lot of precise inputs are better handled automatically than manually. To be even thinking about automation, one should have the manual testing side in "flawless" state. Automation initial cost is high and it is not cheap to change or alter the test cases.

As there has been no previous attempts to do this memory profiling automation, at least in Team Terra, we have to pretty much start from the zero and do the basic research and planning. In this work, we use Team Terra's current tools for testing and development, but also introduce some new tools to get the log of memory profiling visualized.

In the implementation, we use Google Chrome's command line flags to start the memory profiling from command console. First, we run this command along with automated unit tests using Karma, then we do the same for automated functional tests that are handled by Nightwatch.js. These commands will give us profiling log file that is in raw JSON-format.

The file has to be translated into valid JSON, it includes irrelevant information also, so we have to parse the data and create new file. We chose ElasticSearch and Kibana, to be our database and user interface tool for showing infographic based on the data, respectively. In our parsing-script we also send the new data to the database where the UI tool, Kibana, gets that data and draws charts according to it.

In the end, we use Jenkins CI to do the final automation so it runs this whole process with given bash commands on certain time intervals.

Keywords
Memory profiling, memory leak, automation, performance testing, Nexterday UI Platform, Google Chrome

Haaga-Helia
University of Applied Sciences

Abstract

13.9.2017

Tekijä(t)
Lucy Liu

Koulutusohjelma
Tietojenkäsittelyn Koulutusohjelma

Raportin/Opinnäytetyön nimi	**Sivu- ja liitesivumäärä**
Automatisoitu muistin profilointi selaimelle	57+10

Opinnäytetyön nimi englanniksi
Browser's Memory Profiling Automation

Tämä opinnäytetyö on toimeksianto Comptel Oyj:n, Nexterday UI Platform, Terra-tiimille. Toimeksiantaja halusi automatisoidun muistin profiloinnin selaimelle, jotta tiimin sovelluskehittäjillä olisi helpompi tarkistaa koodin laatu ja havaita mahdolliset muistivuodot.

Muistivuoto tapahtuu, kun muistia, jota on varattu suorittamaan jotain operaatiota, ei vapauteta suoritetun operaation jälkeen. Näin ohjelmisto käyttää lopulta senkin muistin, jonka se tarvitsee toimiakseen normaalisti.

Muistin profilointia voidaan suorittaa manuaalisesti käyttämällä selaimen kehittäjätyökaluja nauhoittamaan selaimen suorituskykyä. Nauhoituksen aikana käyttäjän on tarkoitus tehdä jotakin toimintoa toistuvasti moneen kertaan kyseisellä sovelluksella, jotta saataisiin laadukasta palautetta suorituskyvystä. Kehittäjätyökalu piirtää kuvaajan, mistä käyttäjä näkee, kuinka paljon muistia on varattu eri vaiheissa. Tavallisesti on hyvä epäillä muistivuotoa, jos kyseinen kuvaaja muistuttaa sahalaitaa.

Automaatiolla on monia etuuksia, muun muassa se, että kehittäjät ja testaajat säästävät paljon aikaa. On hyvä automatisoida erityisesti sellaiset testit, jotka tarvitsevat paljon täsmällisiä syötteitä. Ennen automatisoinnin harkitsemista pitäisi manuaaliset testit olla ”täydellisiä”. Automaation alkukustannukset ovat korkeita, mutta testien muuttaminen automatisoinnin jälkeen saattaa myöskin tulla kalliiksi.

Koska tätä aihetta ei olla varsinaisesti aiemmin tutkittu, ainakaan Terra-tiimissä, on aloitettava perustutkimuksesta ja suunnittelusta. Tässä työssä käytetään Terra-tiimin nykyisiä työkaluja testaamiseen ja kehitykseen ja otetaan myös käyttöön pari uutta työkalua.

Tässä implementaatiossa käytetään Google Chromen komentorivikytkimiä, jotta muistin profilointi saataisiin käynnistettyä komentoriviltä. Ensiksi käynnistämme komentorivin automatisoitujen yksikkötestien kanssa Karma-työkalun avulla. Tämän jälkeen sama tehdään Nightwatch.js työkalulla, joka hoitaa funktionaalisien testien automatisoinnin. Komennot antavat syötteenä takaisin profiloinnin lokitiedoston, joka on raakaa JSON-formaattia.

Kyseisen tiedoston data pitää kääntää validiksi JSON:iksi, joten tiedostoa pitää parsia. Näin, saadaan myös muistin profilointia ajatellen turha tieto pois. Parsimisen päätteeksi luodaan uusi tiedosto. ElasticSearch valittiin tietokannaksi parsitulle datalle ja Kibana-työkalulla saadaan piirrettyä ja näytettyä infografiikkaa kyseisen ptofilointidatan pohjalta. Loppuautomaatio tapahtuu Jenkins CI:n avulla, jolle annetaan tarvittavat komentorivikomennot.

Asiasanat
Muistin profilonti, muistivuoto, automaatio, suorituskyky testaus, Nexterday UI Platform, Google Chrome

Table of contents

1 Introduction

This thesis is done as a commission for Comptel Oyj. Projects objective is to develop automated memory profiling for Google Chrome browser. During the first half of the thesis, the commission party is introduced, we will go through the basic theory of memory management and test automation. The early chapters will teach us about memory leaks, what they are and how to recognize and possibly fix them presented with some case examples. This will lead us to performance topic, where the memory profiling as part of testing is introduced.

Automation means that we are making something that would normally need manual work to work on its own. This means that before we automate anything, we need to first know how to do something manually. We will go through basics of using Google Chrome's DevTool's to do performance and memory profiling manually. After this we go through the reasoning of why we would want to automate this, and we will learn not only benefits but also defects of automation.

The other half of this thesis consists the detailed introduction to the project, the background, objectives and risks of it. We will also go through the relevant existing tools that the commission party is using which leads us to the implementation plan and finally to the progress of implementation.

Implementation progress chapter includes not only the successes but also problems and defects that were found or transpired during the process. This might be good for future backlog, and also to define the requirements of the setup.

2 Terms and definitions

API	Application Programming Interface. “A set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service.” (Google Dictionary in Firefox browser, 2017a)
BDD	Behaviour-Driven Development. Software development methodology in which an application is specified and designed by describing how its behavior should appear to an outside observer. (Rouse, 2014)
CI/CD	Continuous Integration/Continuous Development. Development practice that has certain workflow pipeline. Works tightly with version control system.
DB	Database
DevTools	Developer Tools, set of tools that are built in browsers for developers.
DOM	Document Object Model. Cross-platform and language-independent application programming interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document. (Wikipedia, 2017a)
ElasticSearch	RESTful search engine. In this thesis referred often as database
GC	Garbage Collection in system’s memory management
GraphicsMagick	Image Processing System (GraphicsMagick, 2017)
IE11/IE	Internet Explorer 11/ Internet Explorer

JSON	Javascript Object Notation, format for data storing and exchanging. Data has to be text when it is exchanged between browser and a server, which is why JSON is used because it is text (W3Schools, 2017a)
Kibana	Platform to visualize the data. Connected with ElasticSearch
Nightwatch.js	An automated testing framework
Node	“A point in a network or diagram at which lines or pathways intersect or branch” (Google Dictionary in Firefox browser, 2017b)
Node.js	A JavaScript runtime built on Chrome's V8 JavaScript engine.
npm	Node Package Manager. Package manager for Node.js.
OSS/BSS	Operation Support System/Business Support System. “Operated together by telecommunications service providers, are used to support a range of telecommunication services.” (Wikipedia, 2016)
TDD	Test Driven Development. Software development methodology where tests are written first and then just enough of production code to fulfill that test and refactoring. (Ambler, 2017a)
Unit test	Test of a certain part (unit) of the source code

3 Introduction of the commissioning party

Comptel Oyj is Finnish software development company that was founded 1986; the company celebrated its 30th anniversary in 2016. The company has many offices around the world, and employee-wise Finland's office is the biggest one. Comptel is specialized in telecommunication software development and has currently over 300 customers in more than 90 countries, the estimation of the quantity of served end-users is more than 1.2billion. The three biggest teleoperators in Finland, Elisa, Saunalahti, and Telia are all Comptel's clients. (Comptel.com, 2017)

Comptel Oyj develops, sells and delivers products and services that support telecom operators' Operation and Business Support System (OSS/BSS). The popular products are fulfillment and analytics softwares. Fulfillment software is meant to automate the telecom operator's operation process. "Fulfilment is one of the key processes for any operator. This process is responsible for providing customers with their requested products and services in a timely and correct manner" (Comptel intra).

In 2015, Comptel introduced Operation Nexterday. Operation Nexterday is acting as a movement, but it started as a book. Nexterday's mission is to advice operators on how they "can radically update their sales, marketing, and service strategies in response to customers' increasing digital service demands," (Vänttinen 2016). It guides operators on how they can become a better digital company with better customer experience.

This thesis is done for one of the Comptel's front-end development team, Team Terra, which produces user interface components, also known as the Nexterday UI Platform, to other development teams. Team Terra uses various of methods, frameworks and front-end tools. Comptel mainly follows Agile and SAFe (Scaled Agile Framework) development guidelines, there are also some other frameworks that individual development teams might apply to their work process. Some frameworks that Team Terra has applied to their work process are a mix of Kanban, Scrum, and Extreme Programming.

4 Memory management in JavaScript

While memory profiling collects various of data, usually the interesting part is the data and events from JavaScript, since all the code is happening there. JavaScript is garbage collected language, if garbage collection (GC for short) fails or doesn't work properly it means there's a possible memory leak in the system.

4.1 Memory graph

To make it easier to comprehend how browsers handle JavaScript performance, we shall think memory as a graph. All the primitive types, Boolean, string, number, and objects, can be visualized in memory graph as number of interconnected points.

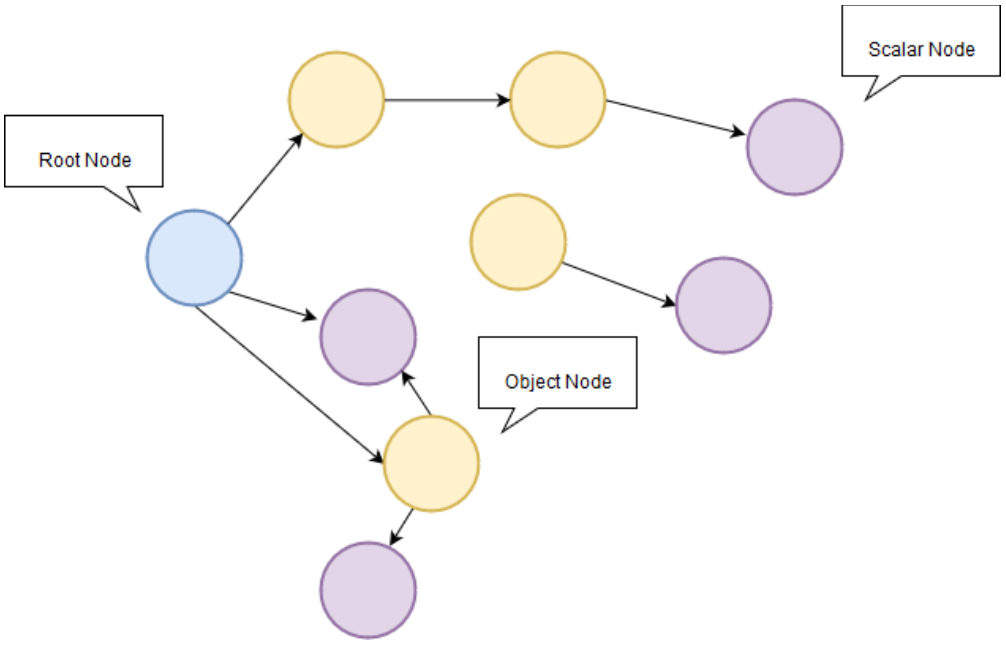


Figure 4.1 The start of the graph is the root node. Root could be a browser window or Global object of a Node module. Root node's garbage collection is not controlled by developers (Osmani, 2014)

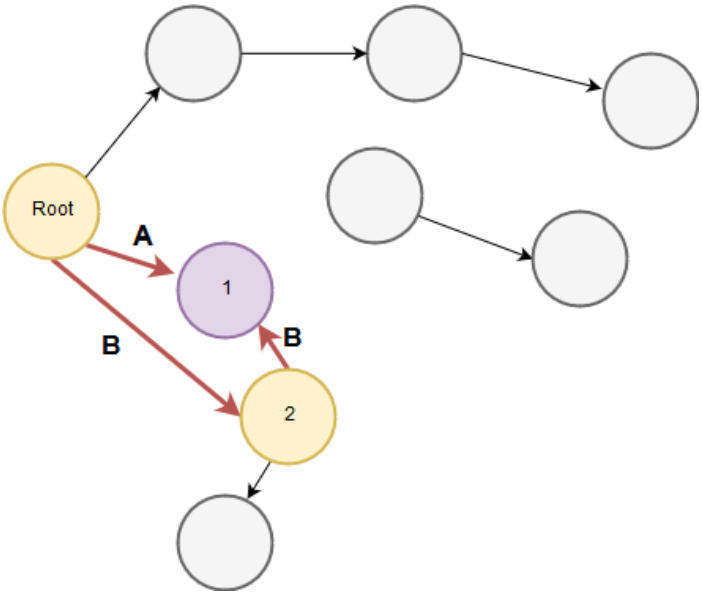


Figure 4.2 A value's retaining path(s) (Osmani, 2014)

Figure 4.2 shows the retaining tree of the purple node. The arrows show the path(s) in the graph that is keeping the object from being classified as memory. In this graph, the root node is retaining tree for the purple node (1). There are two different paths from the root node to the purple node. One path is straight from root to the purple node, we shall name it as path **A**; another path **B** goes through another yellow node (2). Garbage collection will take the purple node if both of its paths to the root node are terminated and it is entirely detached from that root (Osmani, 2014).

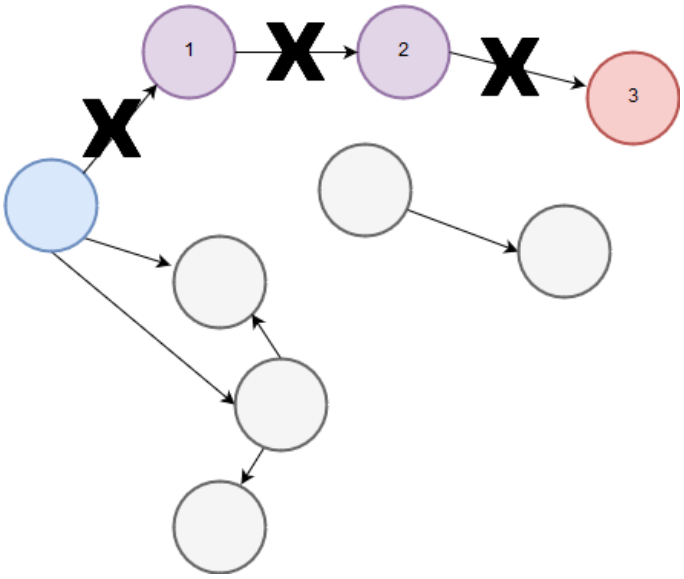


Figure 4.3 Removing a value from the graph (Osmani, 2014)

If we want to remove a node from the graph, we can cut the edges. If we want to remove a red node (3) in figure 4.3, we can cut from any edge with “X”. By cutting the edge closest to the red node, we remove that particular node (3) only. Cutting one from the other two edges will also remove one or both purple nodes (1, 2). (Osmani, 2014)

4.2 What is garbage and garbage collection?

Any node that doesn’t have a retaining path in the system is garbage. In other words, Garbage Collector will attempt to collect all values that cannot reach the root node. GC first goes through all the live values in the system. When it detects dead values, it will return the memory used by them to the system. Garbage Collection can be thought as a big recycling system. (Osmani, 2014)

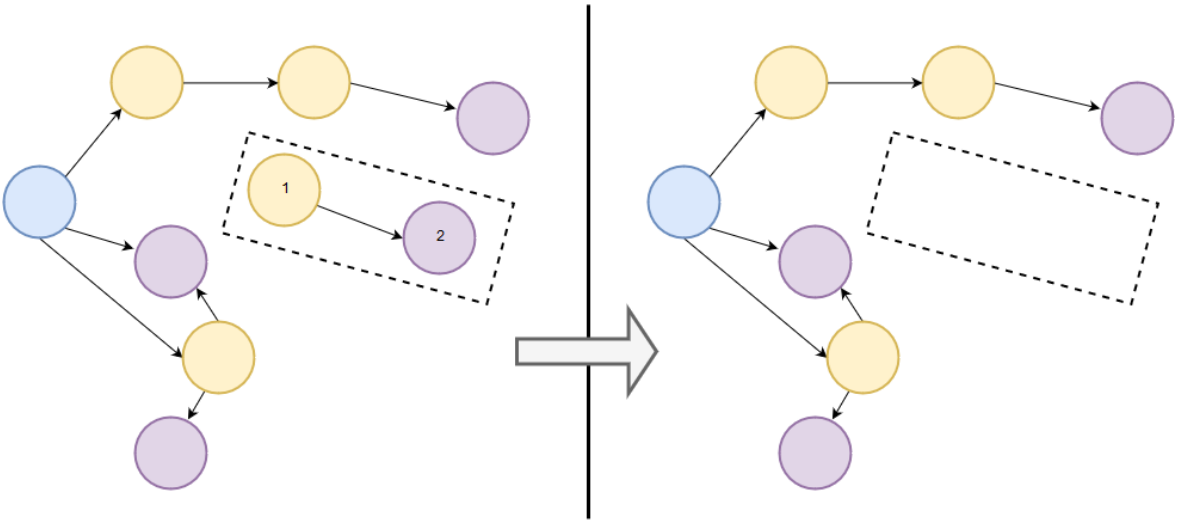


Figure 4.4 Nodes that don’t have a path from root becomes garbage, which is properly collected by Garbage Collector. Here we can see garbage collection of the nodes 1 and 2 (Osmani, 2014)

A shallow size of a memory means the memory that object can hold by itself. Memory can also be held by references to other objects; this prevents those objects from being automatically disposed by GC. Retained size is the memory that is freed when object itself and its depended objects along are deleted (Kearney, 2017)

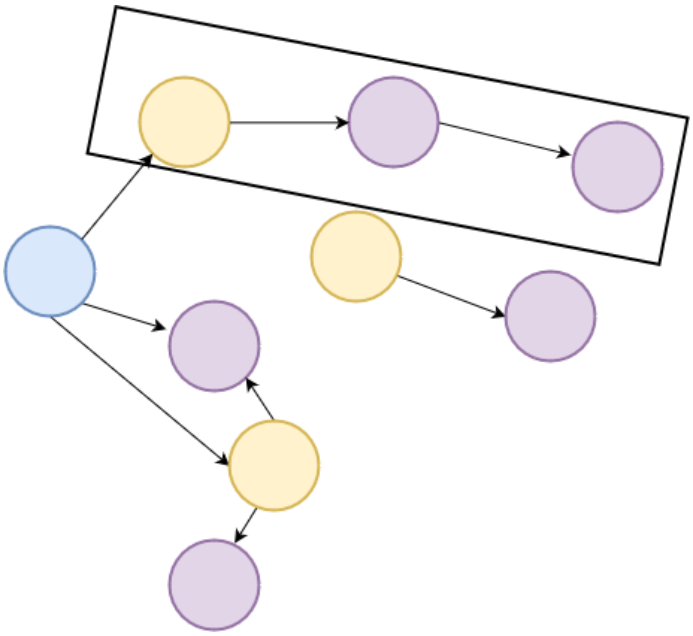


Figure 4.5 Yellow node's retained size is itself plus the purple nodes' sizes (Osmani, 2014)

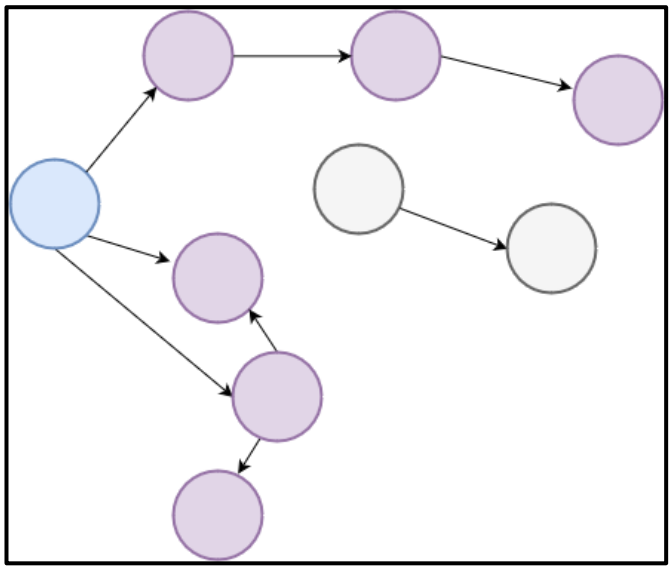


Figure 4.6 Blue root's retained size is all the memory being used by the JavaScript (Osmani, 2014)

4.3 What is memory leak?

If you type "memory leak" in the Google search in Firefox browser (2017c), it will give you following definition: "a failure in program to release discarded memory, causing impaired performance or failure". Memory leak happens when the memory that is allocated to perform some operation is not freed after that given operation is completed, resulting in that the program will eventually use up all the memory that it needs to work normally. If we

think about previous memory graph, a memory leak is when node(s)/data that we removed are still reachable from the root node, meaning that there's still a retaining path to some other value. It is suggested to suspect memory leak when the user realizes that the program has started to work gradually slower for no any obvious reason.

One common cause of memory leak is detached DOM (Document Object Model) nodes. According to Basques (2017) from Google, "A DOM node can only be garbage collected when there are no references to it from either the page's DOM tree or JavaScript code." Detached node means that it is removed from DOM tree, but there's still other code that references it. Figure 5.7 is a good practical example of DOM leak from Auth0 blog by Sebastian Peyrott (2016).

```
var elements = {
    button: document.getElementById('button'),
    image: document.getElementById('image'),
    text: document.getElementById('text')
};

function doStuff() {
    image.src = 'http://some.url/image';
    button.click();
    console.log(text.innerHTML);
    // Much more logic
}

function removeButton() {
    // The button is a direct child of body.
    document.body.removeChild(document.getElementById('button'));

    // At this point, we still have a reference to #button in the global
    // elements dictionary. In other words, the button element is still in
    // memory and cannot be collected by the GC.
}
```

Figure 4.7 Example code that will create memory leak (Peyrott, 2016)

As an example of solving a memory leak, there is a good sample from IBM developerWorks website by Bhattacharya and Sundar that shows a memory leak situation caused by circular reference:

```
<html>
<body>
<script type="text/javascript">
document.write("Program to illustrate memory leak via closure");
window.onload=function outerFunction(){
  var obj = document.getElementById("element");
  obj.onclick=function innerfunction(){
    alert("Hi! I will leak");
  };
  obj.bigString=new Array(1000).join(new Array(2000).join("XXXXX"));
  // This is used to make the leak significant
};
</script>
<button id="element">Click Me>/button>
</body>
</html>
```

Figure 4.8 Event handling memory leak pattern (Bhattacharya & Sundar, 2007)

In Figure 4.8 you see a closure in which a JavaScript object (`obj`) contains a reference to a DOM object (referenced by the id `"element"`). The DOM element, in turn, has a reference to the JavaScript `obj`. The resulting circular reference between the JavaScript object and the DOM object causes a memory leak.
One solution to the memory leak in Figure 4.8 is to make the JavaScript object `obj` null, thus explicitly breaking the circular reference, as shown in Figure 4.9. (Bhattacharya & Sundar, 2007)

```
<html>
<body>
<script type="text/javascript">
document.write("Avoiding memory leak via closure by breaking the circular reference");
  window.onload=function outerFunction(){
  var obj = document.getElementById("element");
  obj.onclick=function innerfunction()
  {
    alert("Hi! I have avoided the leak");
    // Some logic here
  };
  obj.bigString=new Array(1000).join(new Array(2000).join("XXXXX"));
  obj = null; // This breaks the circular reference
  };
</script>
<button id="element">Click Me>/button>
</body>
</html>
```

Figure 4.9 Break the circular reference (Bhattacharya & Sundar, 2007)

Good rules for how to avoid coding memory leaks accidentally:

- Avoid circular object references

- Avoid using long lasting references to DOM elements that you do not need anymore.
- Use the right variables. One common type of memory leak is caused by accidentally using global variables.
- Free unused event listeners
- Keep an eye on local cache and what you store there.
(Osmani, 2014)

4.3.1 Memory leak patterns - The Sawtooth Curve

If the memory usage chart after profiling is saw-tooth shaped, it means that a lot of shortly lived objects have been allocating. When the line drops back to the baseline, it means a GC has occurred. If the done actions are not expected to have any retained memory as an outcome, and the DOM node count does not drop back to the baseline where it began, it is highly possible that there is a memory leak.

Usually, browser's garbage collection should bring the memory back to where it was during the starting point. The memory usage goes up until garbage collection then it goes back down to the starting point and then starts to go up again. This process is repeated multiple times. This kind of curve, with proper garbage collection, indicates that there's no leakage.

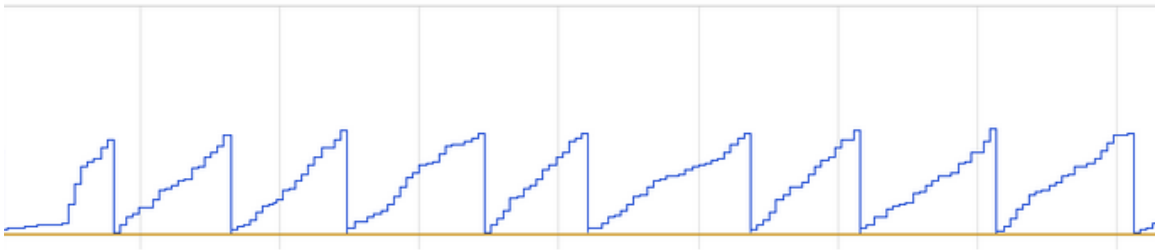


Figure 4.10 Normal memory usage curve without leak (Kerr, 2015)

Figure 4.11 shows the not wanted saw-tooth curve. We can see from the figure that there is garbage collection happening, but the graph never goes down to the starting point. Instead, it is only growing. On this case, it most likely means there is a leak, and further investigation is recommended.

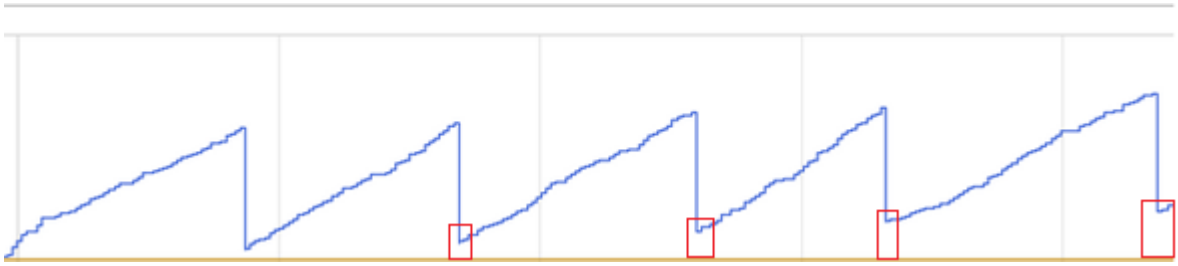


Figure 4.11 Graph never goes down to starting point. Red box indicates the rising points (Kerr, 2015)

Sometimes the memory leak in the graph might not be saw-tooth shaped but resemble the shape of stairs instead. If the stairs never go down even with garbage collection, it might also indicate a possible memory leak. However, the application might be intentionally made so that it uses gradually more memory. So, it depends on what the app is doing and whether it is supposed to do so. “It’s not a bug, it is a feature,” said by a developer and in this case, it is actually true. We can think about infinite scroll in websites and apps as an example. Twitter uses an infinite scroll with “lazy load” effect, which means more tweets (posts/items) will be loaded automatically to the feed when you reach the bottom of the screen. The previously loaded tweets are still there, they are still in the DOM, which is why the memory used by them cannot be released. Once you close the Twitter, the reserved memory is not needed anymore. Thus, creating a healthy stair-shaped curve. Eventually, you might have some problems with your memory resources if you scroll too much. “However, this is not a memory leak - it's just increasing memory usage.” (Kerr, 2015.)

5 Performance & Browser's Memory Profiling

This chapter will explain what memory profiling is, what part of testing it is and why it is important to profile memory during software development process.

5.1 Performance nowadays

In today's world, most of the devices refresh their screens 60 times a second and the users expect a responsive performance of the webpages that they visit. The pages should be interactive and operate smoothly, giving users an enjoyable browsing experience. The tolerance for slow loading and "janky" performance is only getting lower. Refreshing something on 60 times a second means that each of the individual frames has a little bit over 16ms to work and sort everything (1 second / 60 = 16.66ms). In this 1 second, the browser has to handle the input whereafter JavaScript start typically executing jobs that affect the visual parts of the page. The states of the page get updated, and this often triggers a layout change, where the browser re-layout all the elements inside the page and then also paint all the elements. Painting in here means filling in the pixels, e.g. drawing text, colors, images and borders. Finally, all the work is composited on the user's screen. This everything is supposed to happen 60 times in a short span of 1 second (Lewis, 2017.)

Figure 5.1 Browsers pixel-to-frame pipeline. Not every part of the pipeline is being activated on every frame e.g. if the layout is not changed the pipeline will jump straight to painting

5.2 Performance testing

As defined by Graham & al. (2008, 48), "The process of testing to determine the performance of software product." The primary goal of performance testing is to develop effective enhancement strategies for maintaining acceptable system performance. Performance testing is a process of gathering information and analyzing it. The data from this process is collected and used to predict when load levels will use too much system resources. The results of non-functional tests give developers insight into system performance and response time under real-world conditions. Response time is the amount of

time a user must wait for a Web system to react to a request, e.g. a search (Nguyen 2001, 43). Profiling during the construction phase helps to minimize the cost of defects.

5.3 Memory Profiling is Non-functional black-box testing

Memory profiling is counted as non-functional black-box testing, where the interests are in how well or how fast something is done. It is testing something that is needed to measure on a scale of measurement, for example, time to respond. Non-functional testing includes, for example, performance, load, stress and usability testing. (Graham, Veenendaal, Evans & Black 2008, 47.)

Memory profiling is “a process of investigating and analyzing a program’s behavior to determine how to optimize the program’s memory usage” (Telerik 2017). Memory profiling helps the developers to test the written program’s performance quality; developers use it to detect possible memory leaks. Memory profilers are tools for performance analysis.

5.4 Cost of defects/change

Why we would want to practice memory profiling at regular times can be explained with software development’s cost of defects (aka. Cost of change) chart. Memory profiling is helping to keep the cost of change curve flatter.

The cost of finding and fixing defects rises exponentially as software development progresses through its life cycle stages. The figure below presents the traditional cost defects curve when looking at the whole lifecycle of software development in the waterfall model.

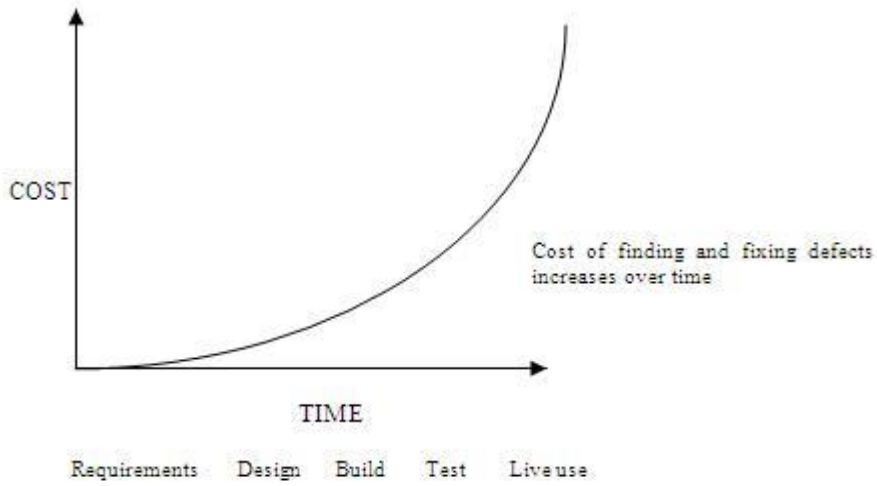


Figure 5.2. The cost of change in normal waterfall model software development (ISTQB Exam Certification)

Comptel currently practices agile software development, which flattens the curve of the cost of defects. As Team Terra development practices also have part of extreme programming, we also have the Kent Beck's (creator of extreme programming) cost of change curve which is flat since the feedback loop is much shorter in extreme programming (Ambler, 2017c)

> Short feedback loops are an important aspect not only of testing but also of the complete agile software delivery cycle. Transparency in decision making is needed for long-term success in providing high-quality software. (Gregory & Crispin 2014, 15)

In Agile software development, there's various of feedback loops, some of them are for example unit tests, code reviews, continuous integration, daily scrum and sprints. Memory profiling is part of testing which supports feedback (Puckett, 2011.)

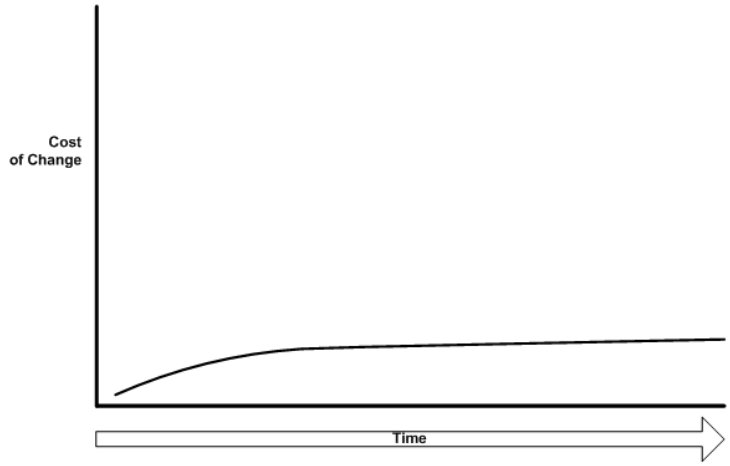


Figure 5.3 Kent Beck's cost of change curve (Ambler, 2017c)

Scott W. Ambler, software engineer and author of various books and papers related to the agile framework (Ambler, 2017b), thinks that the true cost of change curve is not as flat as Kent Beck presents it, but instead rises gently over time (Figure 5.4). The cause might be because the code base most likely grows over the time, which means that there's increasingly bigger chance that any occurring change might affect multiple things. Ambler's view of the cost of change in agile software development is not as dramatic as presented in the waterfall model, to keep it like this, we have to follow the guidelines of agile working and keep the source code quality good and feedback loops short. (Ambler, 2017c)

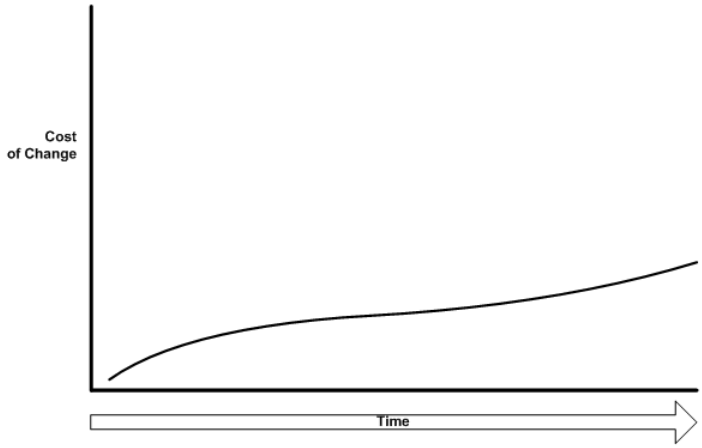


Figure 5.4 Scott Ambler's view of true cost of change curve (Ambler 2017c)

5.5 Manual memory profiling

Browser's memory profiling is manually done by using that given browsers' developer tools (DevTools). Usually, there should be some built-in profiling tools. There used to be separate software or plugins like Firebug for Firefox for profiling in web development, but as the browsers started to have DevTools with profilers integrated into themselves, it became easier to just use the browser's developer tools instead of third party software/plugin. In the example of Firebug, it is not being developed or maintained any longer as a separate software but was merged with the Firefox developer tools in 2016 (Odvarko, Walker, 8.2.2016). Every developer tool is a little bit different, but they usually have a tab called "Performance" and "Memory," these are the features we are interested in memory profiling.

How the memory profiling works is that in the browser we open DevTools, navigate to the right tab of the tool and then start recording (profiling). When the recording starts, the user interacts with the chosen web page. These interactions are, for example, clicking buttons or fields, typing some input, basically "activating" something on the webpage and they will create events that the profiler records. Usually, during the profiling, the same operation is done multiple times, this is called repetitive testing which helps the developers to find possible memory leaks. After the wanted operations have been done, developer ends the recording session after which DevTool presents different graphs of the system's performance and operations that happened.

Figure 5.5 Chrome DevTools Performance tab after recording something

In Figure 5.5, a finished performance profiling can be seen. It can be quite overwhelming output when seeing it the first time, but it is actually divided into clear sections. The information can be read as individual sections, but also together since it is supposed to resemble a timeline. Inside the red box are some labels (FPS, CPU, NET and HEAP) that describe the “function” of that line. So, for example, FPS, which stand for Frames Per Second, shows the frame output. If we see a sudden drop in frame rates, we might want to know the cause of it, thus we can just use the timeline presentation layout in our advantage and search the information around the same vertical line. The heap line graph is presented twice; the first one is to see the overall graph, to see the bigger picture of it. The second heap graph is bigger and can be filtered, in the figure 5.5 only JS heap filter is selected. Above the first heap graph, there are screenshots that show the actual change that happened in the user interface. In the middle of the two heap graphs, there is a flame chart of the functions. This same graph without markings can be found bigger in the appendix 2.

Usually, during the first time of memory profiling, it is more common to record only with the “Performance” -tool which gives more general data. If something dubious is spotted, the next step would be using the “Memory” -tab, depending on the browser it lets you take heap snapshots and record allocation timelines/profiles, which helps the developer to pinpoint the reason for possible memory problems. Screenshots from Google Chrome,

Firefox and Internet Explorer 11/Edge DevTool's Performance and Memory tab are in appendices; Appendix 1 has a browser DevTool comparison table that was created during this browser DevTool research. The table does not cover every single thing in the DevTools, the comparison was done little bit perfunctorily focusing mainly on memory profiling.

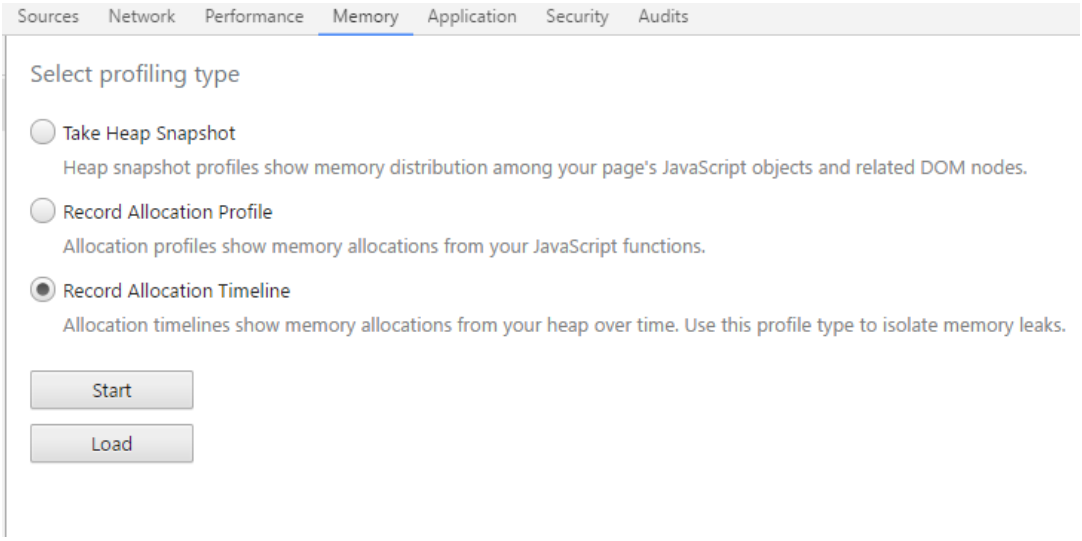


Figure 5.6 Google Chrome DevTool Memory Tab

Let's have an example of recording allocation timeline in Chrome. Like the Performance -tool, this works similarly, the recording start by clicking start button. This mode records heap allocations and shows allocated memory as pillars. The pillars are blue when the Chrome is using the memory, when the memory is freed the pillars will turn grey. If there are pillars or part of them staying blue, it might indicate that there is some memory leaking. (Kerr, 2015).

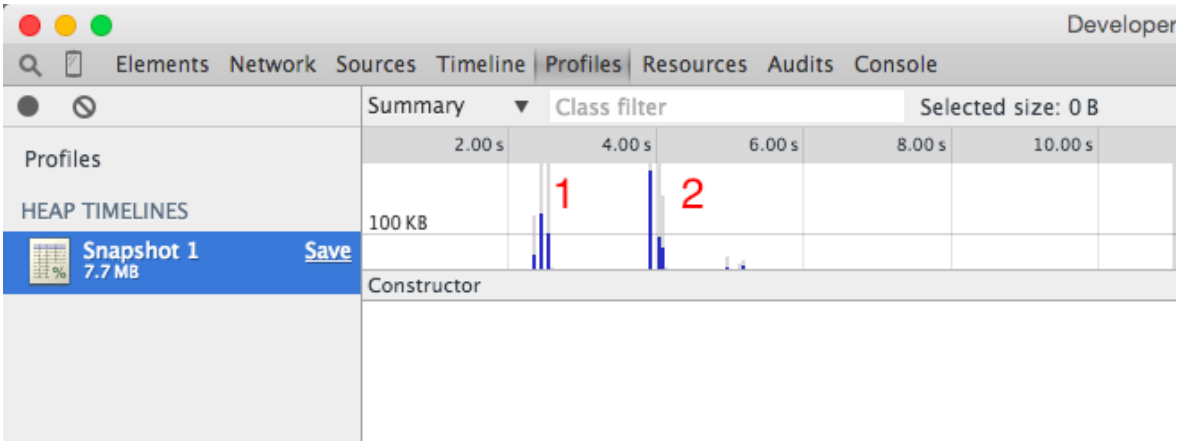


Figure 5.7 View of allocation timeline (Kerr, 2015)

This (Figure 5.7) is old version of DevTools, but the basics are the same so we shall use this situation as an example. This example is by Dave Kerr, he used a simple photo album application for this recording. The starting point is the app's homepage, and the first allocations happen when he navigates to one of the albums. On point 2, he navigates back to the homepage and some of the memory that was just allocated is freed. Some memory is still in use for the homepage itself, and probably for the transition that is happening. Looking at how much memory was freed, Kerr suspects that there might be a leak. He selects the 1st three pillars to inspect what was left in the memory (also potential memory leaks) in the view below.

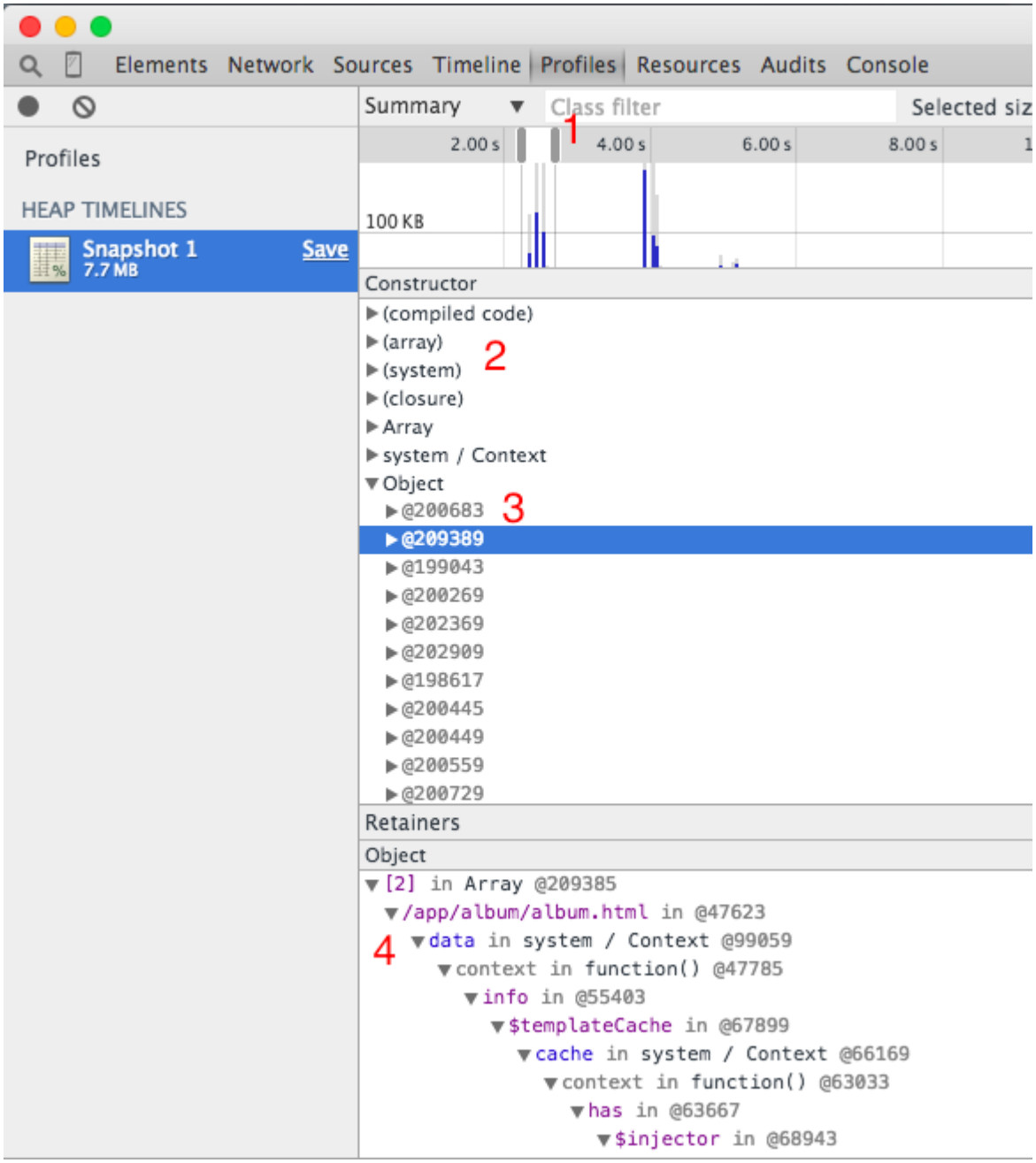


Figure 5.8 Information gotten by selecting the first three pillars (point 1) (Kerr, 2015)

So, in the Figure 5.8 we see points 1-4 that Kerr has marked. Point 1 being the memory left from the allocation. Point 2 shows the different types data in memory, point 3 is the instance of the data, JavaScript object instance in this case. Finally, point 4 shows the retaining path of that object. By investigating the data Kerr finds out that AngularJS template cache is using some data, which mean that the problem most likely not a leak, since AngularJS caches the template that is used to render the album when it is first visited. This means that technically, more memory should be freed during next time when visiting the album and returning back to home page. The cache is already there so the new allocated memory is just for the page itself. (Kerr, 2015).

Performance and allocation timeline recordings are just couple ways from many to track a memory leak manually. Profiling is not hard to do like this, but will definitely be time consuming and tedious if we have a lot of data to go through, which is why automation is considered.

6 Test Automation

In software development, project managers and developers face the challenge of building the application within a tight schedule and with minimal resources. Automation will eventually become a necessity in bigger projects in order to manage the technical debt and make time for other testing activities. Automated regression tests can inform the team quickly if they have broken the source code. Automated tests also act as good living documentation for the application. (Gregory & Crispin 2014, 209). Automation's purpose is to enhance the production performance by making the phases faster and to adapt better to changing requirements. Automated software testing ensures the accuracy and stability of the software through each build. Test automation cannot and will not fully replace manual testing.

6.1 Why automated testing?

Manual software testing is done by a someone going through the system and interacting with it by variation of input, clicking and other usage combinations. The results are compared to the expected output and behavior. Manual tests are repeated often because of the changes in the source code during the development, basically tests should be repeated every time a source code is modified. Multiple and different operating environments, operating systems and hardware configuration are also reasons for running manual tests many times. Manual testing is heavy for labor and likely to have errors, it does not support the same kind of quality checks that are possible with automated testing tool (SmartBear, 2017)

Repeating tests manually takes a lot of time, and it also costs a lot. Once created, automated tests can be run repeatedly, easily and they are much faster than manual tests. Automated software testing can drastically reduce the time in repetitive testing, thus also reducing the costs. Automation helps to run long-lasting tests unattended with the possibility to run same tests on different hardware configurations, using different operating systems or databases. Test coverage is better after automation; every test run can execute thousands of different complex test cases, this could be quite hard with manual testing. Automation also gives upper hand when we want to simulate mass of users interacting with the system; it is quite impossible to perform a controlled web-application test manually with thousands of users (SmartBear, 2017)

As mentioned before, manual testing is prone to errors. It is humane to make errors; even very careful testers will probably make mistakes during repetitive manual testing. Even if

the mistake is very small, in the end, it will still affect the result and increase the used time on testing. One benefit of automated test is that it can replicate the same steps precisely on each test run execution. The actions performed by the automation are always the same and executed at configured time interval, they have a precise schedule. Automated tests can run every time source code changes. Testers and developers freed from repetitive manual tests have more time and energy to deal with other things, so automation increases team's work motivation too. This also improves the team's velocity.

Automating a test affects only how economic or evolvable it is. After automation is implemented, an automated test is generally more economic. However automated tests cost more to create and maintain. When the initial automation of a test is done well, it will be cheaper to implement in the long term; it is more expensive to automate one test than to run it once manually.

The next figure (6.1) is a Keviat diagram, which shows the four quality attributes of a test case. The solid lines represent a test case that is done manually and the dashed lines represent an automated test. When a manual test is automated for the first time we use more work effort in the automation, which is why it will be less evolvable and economic. After some time, when that automated test has been run multiple times, it will become much more economic compared to the manual test (Fewster & Graham 1999, 5.).

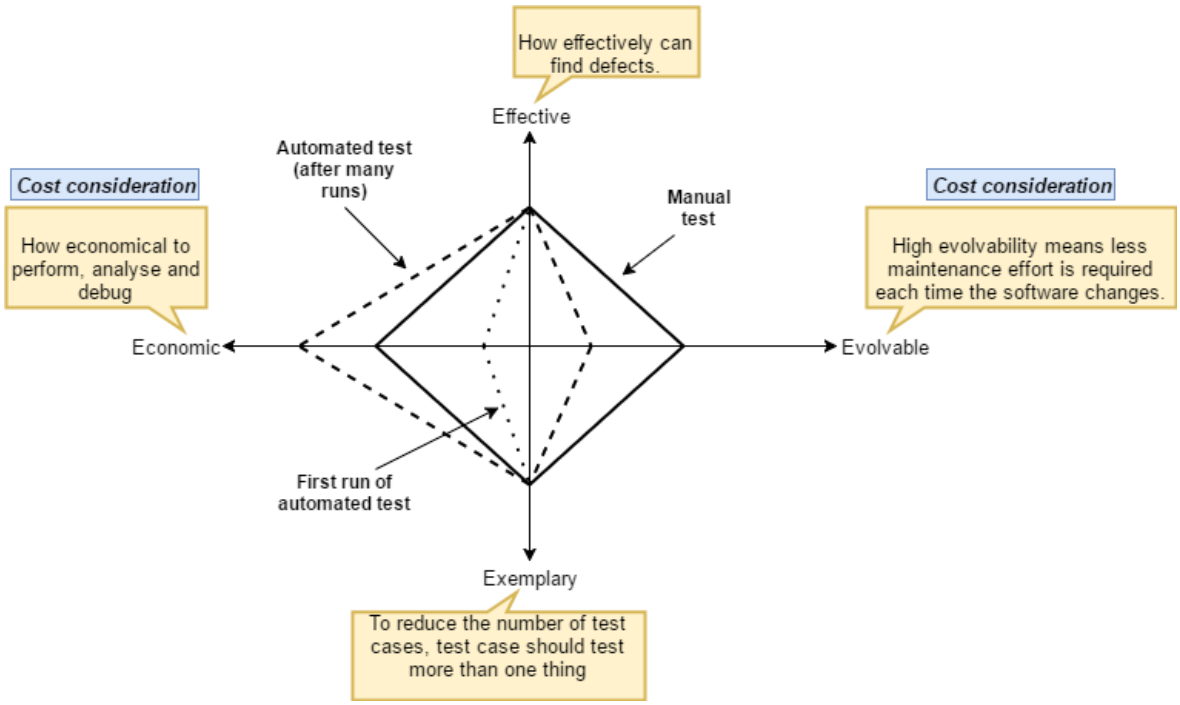


Figure 6.1 "The "goodness" of a test case can be illustrated by considering the four attrib-utes in this Keviat diagram. The greater the measure of each attribute the

> greater the area enclosed by the joining lines and the better the test case." (Fewster & Graham 1999, 5.)

> "Automated testing requires a higher initial investment but can yield a higher return of investment. Skills and training are required to be a successful with any automated testing tool. Every company doing automated testing still does some amount of manual testing", (Fernandes & Fonzo 2013, 3.)

Automation itself is important part of continuous integration (CI) practice. CI practise has following steps: Working in version control. Code is modified in own branches and committed at least once a day. Changes are included in the new integration builds, this happens several times a day on separate machine. Automated tests check the changes from multiple angles to ensure that they work, 100% of the tests should pass for every build. Broken builds should be fixed right away. After these steps, the new built is deployed to production. This is also known as continuous integration pipeline.

6.2 Disadvantages of automated testing

Test automation might give too much confidence, expecting that automation will solve everything and many new bugs will be found after a test is automated. If nothing has really changed, a test is unlikely to find new defects; this applies vice versa also, it is more likely to find new defects when something has changed in the source code, software or environment itself. Of course, not finding actual defects does not mean that the software is perfect. The test might be incomplete, or perhaps they might contain defect themselves. "If the expected outcomes are incorrect, automated tests will simply preserve those defective results indefinitely" (Fewster & Graham 1999, 11). Sometimes the test tool itself may be buggy or dysfunctional. Third-party software products are also vulnerable to defects which raises the chance of technical problems.

Automated test need a lot of maintenance. The effort to maintain tests discourages test automation plans and initiatives often even "killing" them. When the software changes, it is usually necessary to update some, if not all, of the tests so they can be run successfully against the updated product giving reliable and updated data of the current system. Automated tests need continual attention; unmaintained tests will take longer and longer to run over the time (Gregory & Crispin 2014, 248). Automated tests can and will most likely be abandoned if it seems like updating and maintaining those tests take more effort in comparison of running the tests manually. Not only the test themselves need maintenance but

also the servers where they are executed. Test automation initiative should not stop because of high maintenance costs. (Fewster & Graham 1999, 11)

Automation uses a lot of resources, especially in big software companies with many tests that have a lot of test data going through them. Automated tests are in danger if the server is slow or even fully down. Sometimes the fault might not be in the server, but maybe the testers forget or don't care about the capacity limit of the reserved servers. Running one or two builds in test tool is clearly easier than running a dozen of them. If there is "traffic" in a test tool, sometimes a test that could be run in 20minutes might take hours during the traffic.

Automation is also not a good idea if the tests themselves are not good, e.g. bad coverage. Other factors like poor testing practise and unrealiable documentation also speak against automation. "It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing." (Fewster & Graham 1999, 11). It is important to keep in mind that test automation itself is not testing, there is still needed for someone to write the tests that are automated.

6.3 When to automate, when not to, how to decide what to automate?

In chapter 7.2 there was listed already some disadvantages that may be the reasons of why a test should not be automated, but it is important to remember that automated testing does not replace manual tests. There will always be some tests that is easier and more cost effective to do manually. Sometimes the automation of a test might be so difficult that it is more economic to keep the testing manual.

It makes more sense to keep manual tests when:

- Tests are run very rarely
- If the functions change drastically, are entirely new or change frequently, creating automated scripts may be waste of time and less cost effective.
- Strategic application functions where we want to pay specific attention.
- Manual testing is only option. Functions that must be validated by humans. These are for example: Usability, look-and-feel and tests.

(Fernandes & Fonzo 2013, 9.)

On contrast, automated testing is good idea especially when:

- There's a lot of regression testing
- We need fast high-level evaluation on the quality of a build and making yes/no decision on deeper testing, AKA. Smoke testing

- Tests are repetitive and don't change often
- Validation happens with a lot of different inputs and large data sets (i.e. login and search), also known as Data Driven Testing.
- We are testing load and performance.

(Fernandes & Fonzo 2013, 10.)

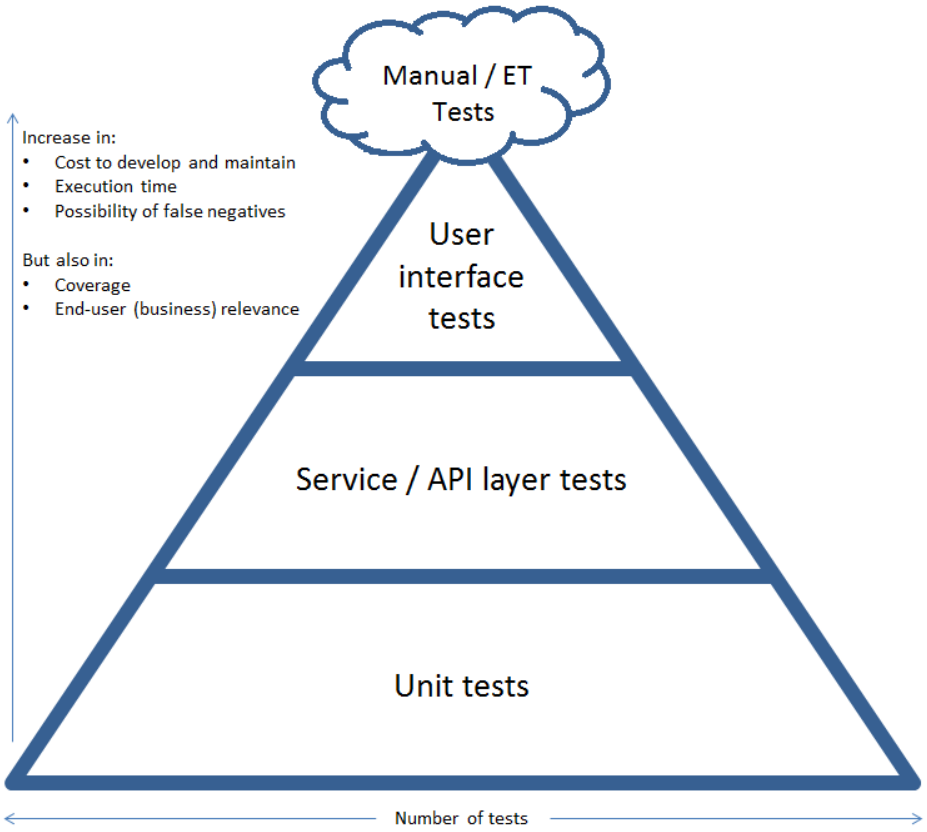


Figure 6.2 Based on Mike Cohn's Test Automation Pyramid (Dijsktra, 26.6.2014)

Test Automation Pyramid was introduced by Mike Cohn in "Succeeding with Agile", and it shows how automation should be done. According to Gregory and Crispin in "More Agile Testing" (2014, 223) the benefit in Test Automation Pyramid is based on "how fast the feedback is received. The lowest level, the unit tests, get the fastest feedback by running with every commit of new code". The higher we go on the pyramid, the slower the feedback is. Gregory and Crispin have modified the original pyramid by adding the little cloud on the top representing manual and exploratory tests that cannot be fully automated. Apparently, many teams require some manual regression tests to supplement their automated checks. Over the years, developers have made their own adaption of the original Test Automation Pyramid. The original pyramid does not fit to all situations, and some assumptions also have been changed due to advances in technology.

We now know more about the advantages and disadvantages of automation, we understand the possible risks and requirements that lead to cost effective and “good” automation.

7 Introduction of the project

Now that we have educated ourselves with the necessary background theory needed, we can focus more on the project itself. This chapter will introduce the background of the project, the goal of it and the possible risks that might jeopardize it. The currently used tools will be introduced and also the previous way of working in regard of memory profiling.

7.1 Background of the project

Currently, the front-end developers have to do the memory profiling and test result comparison manually, and this takes a lot of time and effort. Memory profiling is done to discover possible memory leaks. Automated memory profiling would help the developer's work by ensuring the quality of the code already during the coding process. Early prevention of memory leaks will improve the end product's performance which is important to Comptel's customers. Software installation life expectancy might be up to 10 years.

Manual memory profiling was explained in chapter 6.4. As a brief reminder, it means manually inputting the data or interacting with the components' fields, buttons, and other controls while using browser's developer tools to record the memory usage. Profiling is currently not often practiced in the front-end team. The whole "Automated memory profiling" is a quite uncharted area, meaning that there's no previous work or data in Comptel that this work is based on. This project is supposed to pioneer the automated memory profiling, so other developers can develop it further, which is why there may be some unknown factors. The development and testing environment for this work is going to be Team Terra's internal test application. It was also decided that the usage will be limited for only Team Terra developers and not released for external use, for now.

7.2 Currently used tools

If we were to list all the "official" tools that the current front-end developers use, the list would be too large, which is why we'll will only list those that are closely related to this project's topic. These tools are all open source and can be divided into four different main categories: front-end libraries and frameworks, tools for building, tools for testing and finally, tools related to build/test environment.

7.2.1 Front-end libraries and framework

- React: “A JavaScript library for building user interfaces” (npm, 2017a)
- Redux: A framework that controls states in a JavaScript app (GitHub, 2017a)
- Bluebird: JavaScript promise library, helps with handling asynchronous code that has a lot of callbacks (GitHub, 2017b)
- Lodash: “A modern JavaScript utility library delivering modularity, performance & extras” (Lodash, 2017)
- Moment.js: JavaScript library for handling dates and time, e.g. parsing and manipulating (Moment.js, 2017)

7.2.2 Tools for building

- Babel: JavaScript compiler, “community-driven tool that helps you write the latest version of JavaScript” (GitHub, 2017c)
- WebPack: JavaScript bundler. “Packs many modules into a few bundled assets. Code Splitting allows to load parts for the application on demand. Through ‘loaders’, modules can be for example, CommonJS, CSS, Images, JSON, CoffeeScript and developer’s own customised content.” (Github, 2017d)
- PostCSS: “A tool for transforming CSS with JavaScript” (PostCSS, 2017)

7.2.3 Tools for testing

- Karma: Test runner for JavaScript, “tool that allows you to execute JavaScript code in multiple real browsers” (Github, 2017e)
- Enzyme: JavaScript testing utilities for React. Enzyme's API is meant to be intuitive and flexible by mimicking jQuery's API for DOM manipulation and traversal (npm, 2017b)”
- Chai: “A BDD / TDD assertion library for node and the browser that can be delightfully paired with any JavaScript testing framework” (npm, 2017c)
- Selenium: Automates browsers by using WebDriver API on the server side (SeleniumHQ, 2017)
- Nightwatch.js: An automated testing framework based on Node.js that is run against Selenium/WebDriver server. Nightwatch.js is the client side implementation of the WebDriver API specification (GitHub, 2017f)

7.2.4 Build/Test Environment

- Web browsers: Mozilla Firefox, Google Chrome, Microsoft Edge and Microsoft Internet Explorer 11

- Jenkins CI and GitLab CI: Continuous Integration Servers (CI servers). Used for automation. On every push to the repository, the CI servers will automatically run the tests (Figure 7.1).

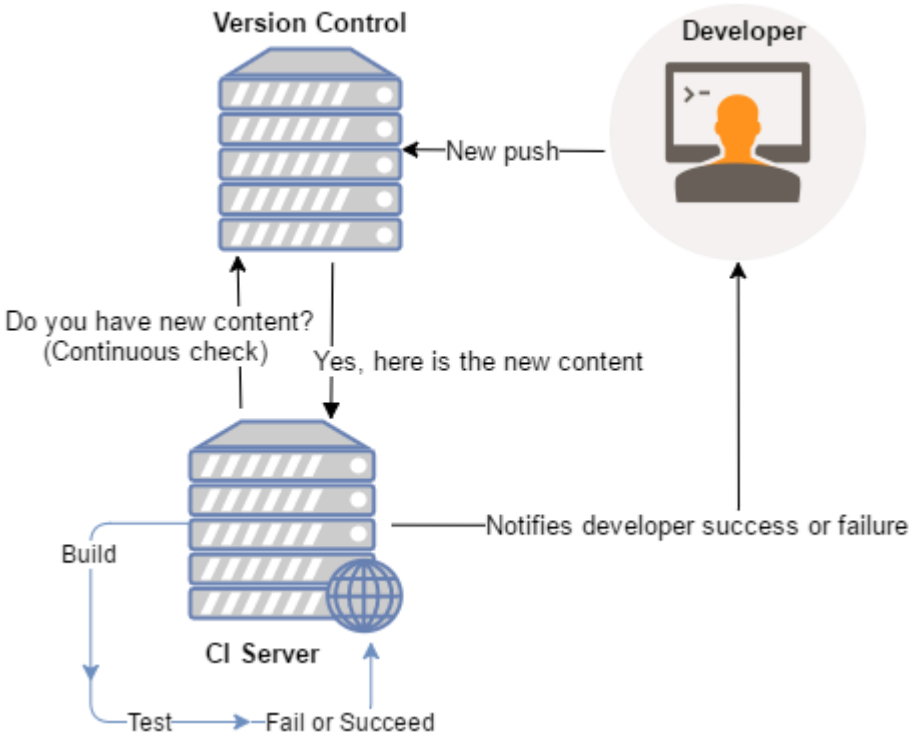


Figure 7.1 CI Servers' workflow

7.3 Objective

The ultimate goal is to have a working solution for automating the memory profiling. As mentioned, this is a little bit uncharted area in front-end development of Team Terra, so the job is to do the base work, find the right tools to execute the automation and fit the solution(s) in Comptel's product. This project's "product" is the toolset, JavaScript -file and documentation for other developers to guide them how to set up the automation.

7.4 Risks of the project

One of the risks is if the development/testing environment becomes unstable because of other development. This should not happen in an ideal case, but there's always a chance that, for example, some broken or more likely buggy code, might end up in the source

code that this project is run against. This can be partly prevented by using an own, project-specific branch in revision control and keeping it on update.

New technology for me can also be counted as a risk. There might be some tools, software or other information that I am not familiar with or don't know at all. Getting to know entirely new tools or other things slows down the project. Of course, one of the reasons and objects of this thesis is to also learn along the process; the learning process already starts from the theoretical part of this thesis.

The overall time reserved for this project might not be enough. Especially the reserved time for the implementation might be doubled. This is because the topic is quite new for the company, there hasn't been previous "groundwork" before this project. The tool library is also quite large, so there are quite many dependencies. In order to make the automation work, there are many things that have to be stable and working.

Then there are also hardware or total system failures. Very unlikely but still possible. Any code produced will be automatically saved in the version control if committed, which is why often committing is practiced.

8 Implementation plan

This chapter will focus on the implementation planning. We will go through the chosen tools for the implementation and the reason they were chosen.

8.1 Browser selection

Because of the project size and deadline, the project will be developed only on Google Chrome. Currently, the Nexterday UI Platform officially supports only two latest versions of Chrome, Edge, and Firefox. IE11 is the only supported version from Internet Explorer. Per the statistics of the most popular browsers published by W3Schools, Chrome, Internet Explorer, Firefox and Safari are four of the most popular browsers currently. Google Chrome became most popular web browser in 2012 and has held the title ever since, which is the main reason this project will be done against Google Chrome.

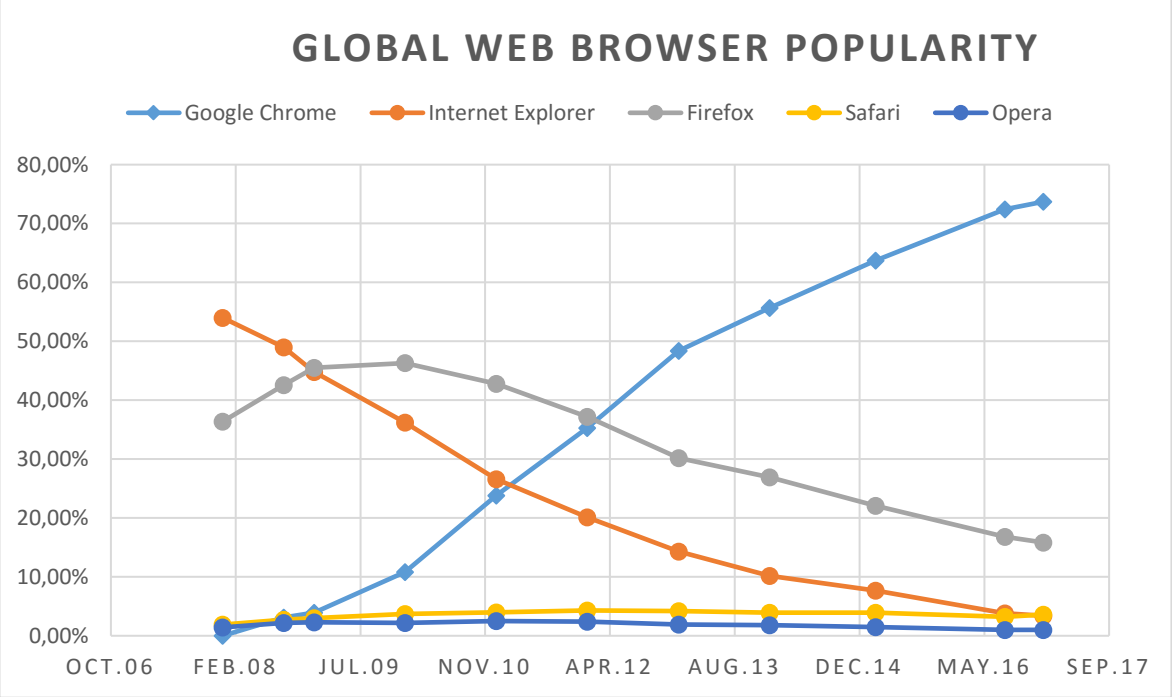


Figure 8.1 Web browser statistics from 2008 to 2017. Top 5 of the globally most popular browsers (W3Schools, 2017b)

The selection was mainly based on the browser popularity and how good the manual memory profiling seemed to be. Some of profiling functions from Chrome were mentioned in earlier chapter 5.5 Manual memory profiling. Chrome browser seemed to be the best suited overall candidate to deploy this on because of its capabilities and how well the browser supports the current setup of the Terra team.

8.1.1 JavaScript Engine

The computer itself does not understand the JavaScript language; JavaScript engine converts JavaScript into something that the computer understands, machine code. Jen Looper from Telerik's Developer Network (2015) defines that JavaScript engine's job "is to take the JavaScript code that a developer writes and convert it to fast, optimized code that can be interpreted by a browser or even embedded into an application." There are multiple different engines because each one of them is designed for a different browser. (Looper, 2015).
To cover the most popular browsers' engines:

- V8 JavaScript engine is open source and developed for the Google Chrome web browser; it is also used in Node.js, a cross-platform JavaScript runtime environment (Developers, 2017).

- SpiderMonkey is developed for Firefox browser, and it is also used in some Adobe applications and GNOME desktop environment (Wikipedia, 2017b)

- Trident and Chakra engines are used in Internet Explorer 11 and Edge browsers, respectively. Chakra was originally closed source but became open sourced in 2016 (Wikipedia, 2017c)

The commission party uses Node.js currently. Therefore, the V8 engine is also used. This adds to the reasons why this thesis is done for and against Google Chrome.

8.2 Development environment

All the tools are from the list that was previously mentioned in chapter 8.2. It is encouraged to use the existing tools (since there is already quite a lot) if needed new tools can be introduced. Development will use Windows 10 operative system for testing environment.

8.3 Action steps

To automate something, we need first to have something to automate. Because this has not been done earlier, we do not really know if it is even possible, which is why we have to start from the basics. We need to start from the manual actions, moving to semi-automation, and then make it fully automated. Semi-automation stands here, for example, to be able to run something with a command line perhaps. The first goal is to profile unit tests

with Karma, then larger functional tests with Nightwatch.js. After this, we want to somehow visualize the data. At this point, we have pretty much all the manual steps needed from getting the data to the actual visualization, which leaves us to only automate it with Jenkins CI somehow.

9 Implementation progress

This chapter and its subchapters will concentrate on the execution in practise. The flow of the work will be described, relevant parts of the written code among the shell commands will be shared. New tools will also be introduced and explained here.

9.1 Chrome flags

We tried to start and run browser developer tools (DevTools) by outer console/terminal command line. After some research, it was found that Google Chrome has a feature called flags that can be enabled or disabled. These flags are usually functions that have not been fully implemented or standardized or are just not necessary for normal Chrome user, e.g. debugging flags. By running following command: `$ ./chrome.exe --js-flags="--help"` , all the currently possible and available Chrome flags will be listed. The list is very long, so it will not be attached to this work, but the same list can also be found online automatically updated by Peter Beverloo.

At first we tried to start up Google Chrome with flags, we were not successful with Cmder terminal (one type of 3rd party console for windows) for some reason, so we run the same command in GitBash (another type of console) which gave some response.

After researching a bit, a chosen combination of wanted flags was made. We navigated to the chrome.exe, in this case it was located in Program Files (x86), and ran the Chrome with flags:

```
$ chrome.exe --trace-startup=disabled-by-default-memory-infra --enable-heap-profiling=task-profiler --trace-startup-file=/tmp/foo1.json  --trace-startup-duration=30
```

Table 1 Google Chrome flag descriptions (Beverloo 2017)

Flag	Description
--trace-startup	No description
--trace-startup-file	If supplied, sets the file which startup tracing will be stored into, if omitted the default will be used "chrometrace.log" in the current directory. Has no effect unless --trace-startup is also supplied. Example: --trace-startup --trace-startup-

	file=/tmp/trace_event.log As a special case, can be set to 'none' - this disables automatically saving the result to a file and the first manually recorded trace will then receive all events since startup.
--trace-startup-duration	Sets the time in seconds until startup tracing ends. If omitted a default of 5 seconds is used. Has no effect without --trace-startup, or if --startup-trace-file=none was supplied.

```
C:\Program Files (x86)\Google\Chrome\Application
λ ls -lh
total 1.3M
drwxr-xr-x 1 cpt2rll 1049089    0 Apr 12 03:00 59.0.3067.6/
drwxr-xr-x 1 cpt2rll 1049089    0 Apr 12 13:35 SetupMetrics/
-rw-r--r-- 1 cpt2rll 1049089  404 Apr 12 03:00 chrome.VisualElementsManifest.xml
-rwxr-xr-x 1 cpt2rll 1049089 1.2M Apr 11 12:56 chrome.exe*
-rw-r--r-- 1 cpt2rll 1049089  80K Dec 13 10:41 master_preferences

C:\Program Files (x86)\Google\Chrome\Application
λ chrome.exe --trace-startup --trace-starup-file=/tmp/foo1.json --trace-startup-duration=10
```

Figure 9.1 Starting Google Chrome with flags as shell command

Opening Chrome with these flags allows me to interact with the view while Chrome traces my actions for 10 seconds, after 10 seconds, it will create a raw JSON file in the given directory. The created JSON-file can be imported to Google Chrome DevTools. As the JSON is raw and not “true” JSON, meaning that it’s not following the accepted JSON formatting rules, we would need to parse it later in the development work.

9.2 Implementing to Karma

```
const path = require('path');
const moment = require('moment');

 // Creates json file with date and time in the filename.
 const filename = moment().format('YYYYMMDD-HHmmss');
 const traceFile = path.join(process.cwd(), `${filename}-karma.json`);

 karmaConfig.set({
   browsers: [
     // http://karma-runner.github.io/1.0/config/browsers.html
     // @see https://github.com/karma-runner/karma-chrome-launcher
     'ChromeHeadless', // By default run tests in ChromeHeadless
   ],

   customLaunchers: {
     // Custom options for Chrome
     // @see http://peter.sh/experiments/chromium-command-line-switches/
     // @see https://github.com/v8/v8/wiki/Profiling%20Chromium%20with%20v8

     ChromeProfiling: {
       base: 'Chrome',
       flags: [
         '--trace-startup',
         `--trace-startup-file=${traceFile}`,
         '--trace-startup-duration=10',
       ]
     }
   },
/**
 *  Run tests in Chrome for memory profiling:
 *    'npm test -- --chrome-profile'
 *    or
 *    'karma start --chrome-profile'
 *  Will overwrite the default PhantomJS
 * @see https://github.com/karma-runner/karma-chrome-launcher
 */
if (karmaConfig.chromeProfile) {
  karmaConfig.set({
    browsers: [
      'ChromeProfiling',
    ],
  });
}
```

Figure 9.2 karma.config.js configuration. This is a combination of multiple code snippets from the actual source code to just demonstrate the crucial parts to get the unit test profiling up and running. This code will not work as like this

Next step was to put the commands in karma.config.js file (Figure 9.2), so it can semi-automatically (still have to use command line for it to start) run the unit test profiling in Chrome. Karma is one of the test automation tools.

For the sake of better identification and transparency, date object function was added in the filename which currently is in the format of “YYYYMMDD-HHmmss-karma.json.” We also want the JSON-file to be created into the currently working repository. Moment.js was used to get the date and time, it is a JavaScript library meant for “parsing, manipulating, and formatting dates” (Moment.js, 2017). Typing `npm test -- --chrome-profile` in the console will run the script.

9.3 Implementing to Nightwatch.js

As shortly introduced in chapter 7.2.3. Nightwatch.js uses Selenium servers for running the functional tests. To configure the profiling to run through Nightwatch.js, we modified nightwatch.conf.js file. In the test settings, we created new browser setting called “chromeProfile” and added the earlier arguments (flags) for chrome in desired capabilities. Next figure will show the final result of the “chromeProfile” configuration. These commands still won’t create the wanted JSON-file. Apparently, the Selenium must be run on localhost and with port 4444 (previous 5555). We had to install selenium-standalone, which is a standalone package that makes it trivial to have Selenium server up and running, since we did not have it previously.

Figure 9.3 Profiling in Nightwatch.js

To start the Nightwatch.js profiling, We have to run the local server first: `npm start`, then run the standalone selenium: `selenium-standalone start`, and finally running the shell command for Nightwatch.js:

`npm run test-webdriver -- -e chromeProfile`.

Adding `--tests <componentName>` specifies individual component's tests that we want to run, without this addition all the currently existent tests will be run.

9.3.1 Additions

At this point of implementation, couple more lines and specifications was added for the flags.

```
'--trace-startup=disabled-by-default-memory-infra',
'--enable-heap-profiling=task-profiler',
'--profiling-flush',
```

MemoryInfra, which is Chrome's timeline-based profiling system is enabled. Also, enabled heap profiling and profiling flush, which flushes the profile data periodically. The data is flushed at the exit by default, "but sometimes there are bugs that prevent the at exit handlers from getting run" (The Chromium Projects, 2017).

9.3.2 Problems

After some test runs, it came to realization that there's a big problem in using trace-startup with trace-startup-duration -flags. It was found that the given duration must be smaller than the actual execution time of Nightwatch.js going through the tests. This is because the data will disappear after the browser is shutdown, which is after the end of the test in the current scenario. So, if we would put duration as 20s, and the test is done in 19s, JSON file will not be created. We also discovered that the defined duration of the tracer has to be few seconds less than the component's tests. So, for example, if normally the tests would run in 25s, we would have to put the tracer for maybe 20s. This is because the closing of browser takes some time.

Since we have so many of tests and most likely the number of them will only increase, this is a big flaw and affects the automation. It would mean that the duration time has to be changed every time based on the test and its duration. Meaning that this flag must be specified by individual component's duration.

To bypass this problem for now, to just get initial "sample data", we will do some recordings by manually defining the time on each component. Actual possible solution would be to script all the tests to run on empty for given time, this way the browser closes **after** the tracer has gone through the tests.

Other problem occurred with autocomplete component. As we are using GraphicsMagick to take the screenshot in visual testing, it will throw error message when we try to run profiling against autocomplete-component. The given error is "`gm convert: geometry does not contain image (unable to crop image)`". This can also be bypassed if we comment out code regarding screenshots either in autocomplete's tests, or disable it all together in "takeScreenshot.js"-file. In the latter file, we just comment out code regarding GraphicsMagick:

```
  // http://www.graphicsmagick.org/GraphicsMagick.html#details-
//crop
    const command = `gm convert "${info.screenshotPaths.temporary-
FullScreenImagePath}" -crop
${sizes.w}x${sizes.h}+${sizes.x}+${sizes.y} "${info.screenshot-
Paths.currentImageDestination}"`;
    execSync(command, {
      encoding: 'utf8',
      timeout: EXEC_TIMEOUT
```

```
});
```

9.3.3 Average Nightwatch.js test run time

To get an idea of how long is the average time in seconds on each test, couple chosen component tests were run for ten times, put them on table and calculated the average time. This was done both in the office and home's WIFI-network (Figure 9.4). What was surprising is that the tests ran faster with home connection, which is around 50mbps, compared to office connection which should be more than five times faster. Connection speeds are of course just theoretical numbers and office renovation reduced the WIFI support point and hence, the more realistic WIFI speed per user varies from 25mbps to 137mbps (at least in the laptop used for this development). Important factor in connection speed is also the number of users that are using the same connection. The speed can be faster during morning and late afternoon compared to noon and early afternoon, since there are less people in the office during early and late hours. At home, there are just couple of users sharing the same connection.

The chosen components for the time estimation “test” are: Autocomplete, Datepicker, Lightbox, List and Sortable Group. The selection was based on how many functional tests they have, the one's that had more than others were selected.

	Run in seconds (HOME WIFI ~50MB)													
Component	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**	**Run 8**	**Run 9**	**Run 10**	**Average**	**Median**	**Min**	**Max**
Autocomplete	19	20	20	19	20	20	19	20	21	20	19,8	20	19	21
Datepicker	18	19	19	18	18	19	19	19	19	19	18,7	19	18	19
Lightbox	31	31	33	32	32	33	32	32	32	33	32,1	32	31	33
List (generic)	39	38	39	41	42	38	38	39	44	41	39,9	39	38	44
Sortable Group	45	52	45	39	39	39	39	39	41	41	41,9	40	39	52

	Run in seconds (OFFICE WIFI)													
Component	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Run 6**	**Run 7**	**Run 8**	**Run 9**	**Run 10**	**Average**	**Median**	**Min**	**Max**
Autocomplete	27	28	28	27	28	27	26	27	28	27	27,3	27	26	28
Datepicker	29	25	26	26	27	27	27	26	27	26	26,6	26,5	25	29
Lightbox	45	34	34	49	32	40	33	33	34	32	36,6	34	32	49
List (generic)	43	44	44	47	43	43	43	44	48	44	44,3	44	43	48
Sortable Group	44	46	45	45	49	49	44	44	44	45	45,5	45	44	49

Figure 9.4 Nightwatch.js tests' runtime in seconds. Average, median, minimum and maximum's are shown.

9.4 JSON-files into charts, ELK-stack

Now that we have some profiling results, we want to find a tool that let us see the file in a visual format, charts, and graphs. Chrome DevTools and chrome://tracing gives this option to import and inspect the elements, but what the commissioning party wants is that

there could be a separate app, open-source one would be most ideal, that shows these data in different, customizable, formats. The main idea is that the app would fetch the file itself, to eliminate those manual processes in the middle, that way the whole system could be automated.

At first, a tool named DataDog which is one type of monitoring service, was considered, but according to one of the company's cloud technical consultant, DataDog is not very suitable in my case, because it's a real-time monitor, e.g. monitoring the current CPU usage. What we want is to show historic events, the results of the tests that have been run already. The specialist recommended either of the following combination:

- Grafana (metric tool) + influxdb (database)
- ELK stack (Logstash + ElasticSearch + Kibana), refer to Figure 9.5

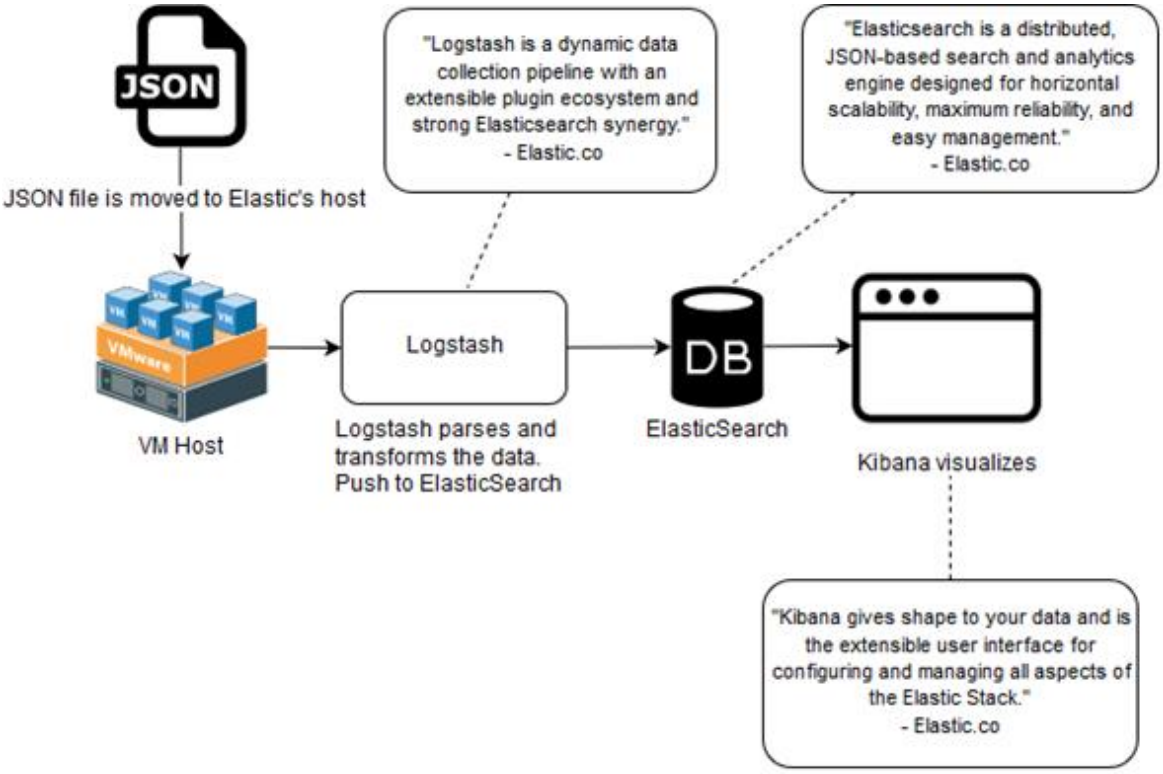


Figure 9.5 Elastic stack's workflow

Originally, Grafana was considered because this tool's name had come up earlier also, but the cloud technical consultant personally recommended ELK or Elastic stack more because of the way it saves and shows data, especially since the JSON files that the tracing produces is rather large (can be more than 100mb). Hence, ELK stack is used for visualization.

The cloud technical consultant helped to get a host for proto and Kibana up and running. The documentation of the whole process of setting up ELK stack is documented internally by the commissioning party. The following three chapters will explain briefly the setup and configuration on ElasticSearch, Logstash and Kibana, respectively.

9.4.1 ElasticSearch

The first step is to install ElasticSearch, the database. After that we want to configure it to store the data in question. A test output “20170508-094847-nightwatch-list” from “List”-component’s profiling was used here. The JSON-file in question is 137MB, quite big, but for now we are only interested to extract and store data that is related to memory allocation. The following figure shows the data we are using.

```
{
    "pid": 11084,
    "tid": 0,
    "ts": 1881874086,
    "ph": "v",
    "cat": "disabled-by-default-memory-infra",
    "name": "periodic_interval",
    "args": {
        "dumps": {
            "allocators": {
                "blink_gc": {
                    "attrs": {
                        "size": {
                            "type": "scalar",
                            "units": "bytes",
                            "value": "1a0000"
                        }
                    },
                    "guid": "48793998f4611faf"
                },
                "blink_gc/allocated_objects": {
                    "attrs": {
                        "size": {
                            "type": "scalar",
                            "units": "bytes",
                            "value": "cdd8"
                        }
                    },
                    "guid": "24c6af4bfa9c916e"
                },
                "malloc": {
                    "attrs": {
                        "size": {
                            "type": "scalar",
                            "units": "bytes",
                            "value": "284000"
                        },
                        "virtual_size": {
                            "type": "scalar",
                            "units": "bytes",
                            "value": "32a000"
                        }
                    },
                    "guid": "64f1affa0cf3312d"
                },
                ...
            }
        }
    }
}
```


Figure 9.6 Data that will be imported to ElasticSearch instance

The events are named “periodic-interval”, and we are particularly interested in the malloc blocks. Malloc is memory allocation function used in C language.
The documents that are created will have pid, tid, ph, cat, name and memory_allocated fields. The latter mentioned field is extracted from args.dumps.allocators.malloc.attrs.size.value. These events are defined by Chrome itself when it creates the tracing file. To import these values, we define an index and the mapping for that index to include these mentioned fields. (Comptel Wiki)

- **name:** The name of the event, as displayed in Trace Viewer
- **cat**: The event categories. This is a comma separated list of categories for the event. The categories can be used to hide events in the Trace Viewer UI.

- **ph:** The event type. This is a single character which changes depending on the type of event being output. The valid values are listed in the table below. We will discuss each phase type below.
- **pid**: The process ID for the process that output this event.
- **tid**: The thread ID for the thread that output this event.

(nduca, dsinclair, 2016)

```
1. curl -XPUT 'localhost:9200/tracing/'
2. curl -XPUT 'localhost:9200/tracing/_mapping/entry' -H 'Content-Type: application/json' -d'
  {
    "properties": {
      "pid": {
        "type": "long"
      },
      "tid": {
        "type": "long"
      },
      "ts": {
        "type": "long"
      },
      "cat": {
        "type": "keyword"
      },
      "name": {
        "type": "keyword"
      },
      "memory_allocated": {
        "type": "long"
      },
      "run_time": {
        "type": "date",
        "format": "yyyy-MM-dd HH:mm:ss"
      }
    }
  }'
```

1. Creates an index
2. Adds an entry mapping and defines the types of various fields

Figure 9.7 ElasticSearch index creation

Now there's an empty index where documents can be imported to.

9.4.2 Parsing JSON and sending to DB

Originally the plan was to use Logstash, which is part of the Elastic set. But the reason why the idea was deserted, is because the parsing configuration file, which is also Logstash's configuration file, would be in the host side and not in the actual Nexterday UI Platform's repository. This means that it would be outside of version control, which is against the current working practise. Configuration file's content is something that might change quite a lot and drastically after the initial creation, which is why it would be ideal if it's in the version control. The other reason we would want to keep it in the same server as working repository, is so the threshold to edit it would be as low as possible. Getting in the host is not difficult, but it would be better to make it so that any additional steps can and will be avoided. The workflow as visualized in Figure 10.4 is thus changed a little, but main idea is still same (Figure 9.8).

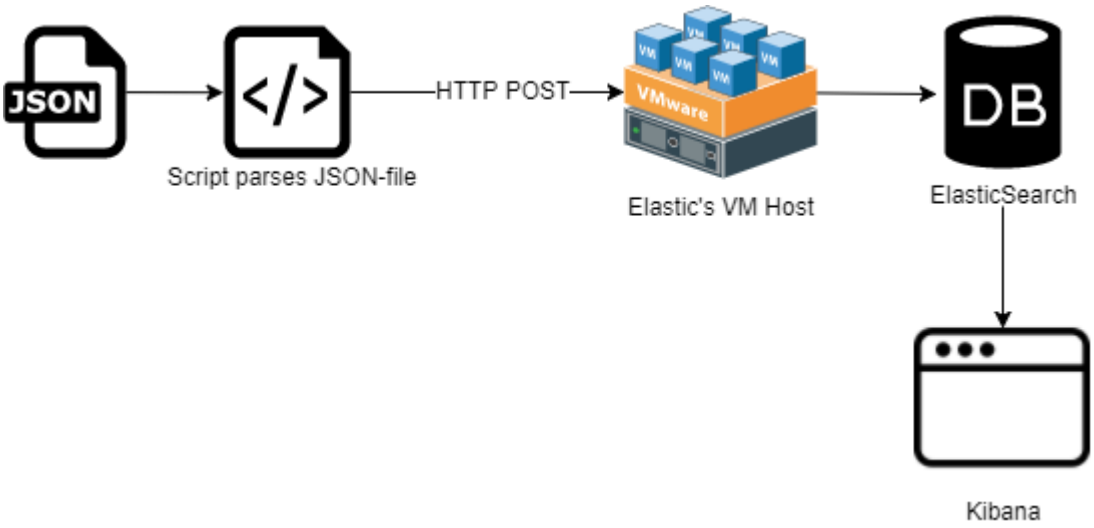


Figure 9.8 New workflow from JSON-file to Kibana

As said, new plan is to parse the JSON file before pushing it to ElasticSearch in host server. A new file called profiling-parser.js is created and the configuration of parsing is scripted there. We use JSONStream and Event-Stream npm package to help us to stream the JSON data, which is then parsed into right format and finally sent to the DB in host server using http POST request.

```
  //Filtered data will be visualized
  const list = d.filter(function (item) {
    return item && item.name && item.name === 'periodic_interval' && item.args;
  }).map(function (item) {

    // NOTE: if malloc.attrs.size.value is hexadecimal, you need to convert it into decimals!!
    const mem = parseInt(item.args.dumps.allocators.malloc.attrs.size.value);
    if (!isNaN(mem)) {
      return {
        'pid': item.pid,
        'tid': item.tid,
        'ts': item.ts,
        'cat': item.cat,
        'name': item.name,
        'memory_allocated': mem,
        'run_time': datetime,
        'component': componentName,
      };
    }
    return null;
  }
).filter((item) => {
  return item !== null;

});
```

Figure 9.9 Parsing and filtering data

We have to change the default port that ElasticSearch is listening to [“0.0.0.0”] in order for our POST requests to work, default port is [“127.0.0.1”]. This is done by modifying the ElasticSearch’s config file by adding `network.host: [“0.0.0.0”]` under Discovery.

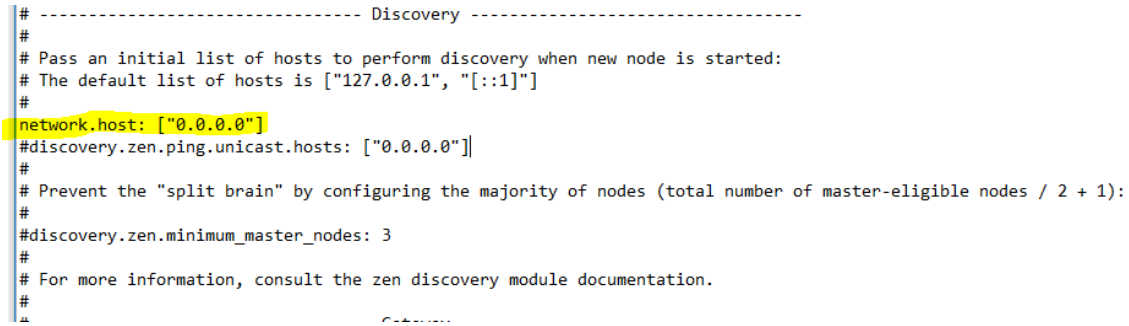


Figure 9.10 Changing Elastic's config, "elasticsearch.yml", to bind port 0.0.0.0

We create one POST request **per object**, there will be error 400 if there’s only one big POST request for every object. The whole code for parsing, formatting and sending to host can be found in attachments.

Figure 9.11 Using POST request to send data to ElasticDB

9.4.3 Kibana

Kibana is one of the Elastic.co's products. It is a tool made for visualizing various of log files. After installing it and made it to run as background process all we have to do is to edit the config file kibana.yml under etc/kibana/ to make it bind the right address.

```
# Kibana is served by a back end server. This setting specifies the port to use.
server.port: 5601

# Specifies the address to which the Kibana server will bind. IP addresses and host names are both valid values.
# The default is 'localhost', which usually means remote machines will not be able to connect.
# To allow connections from remote users, set this parameter to a non-loopback address.
server.host: " IP address "

# The URL of the Elasticsearch instance to use for all your queries.
elasticsearch.url: "ElasticSearch's URL "
```

Figure 9.12 Kibana's config file. This figure only shows the edited parts of the whole file

Now we can access Kibana. Like mentioned in chapter 9.4, this is not focusing on the whole installation and setting up of Elastic's tools. Figure 9.13 shows the outcome in Kibana after data has been sent to database.

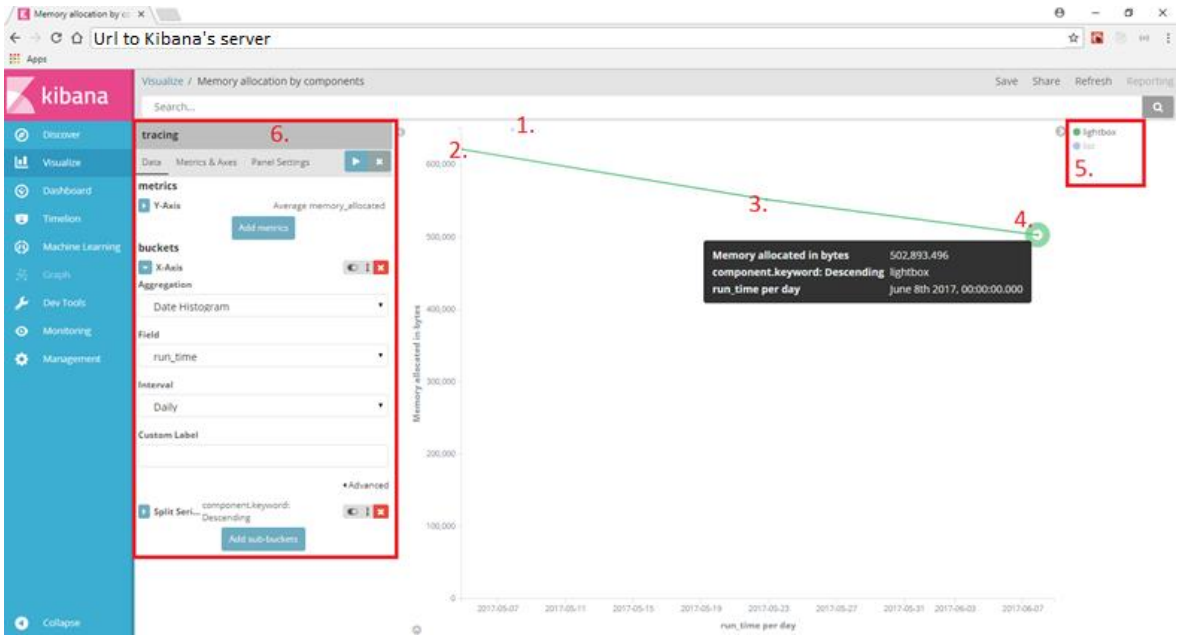


Figure 9.13 Kibana's view after data has been sent to database

It is probably little hard to see from this screenshot (Figure 9.13), but there are currently two different components data visualized, "lightbox" and "list". Point 1 shows the currently only data we have from list. Lightbox on the other hand already has data from couple different days (2, 3, 4). Point 4 shows the hovering feature in this tool. When we hover our mouse over some data point, it will show us more detailed information of that point. In current case, it shows us "Memory allocated in bytes", the component name and date when the data was created. Kibana connects separate dots which gives us a line diagram and of course also gives us the legend (5) that shows which component is which colored line. Point 6 is the configuration panel for the diagram, where we can change the axis orientation to suit our needs, give a title to whole diagram, and most importantly set the interval we want to see the data in. In this particular figure, the interval is set to "Daily". Other possible interval settings are for example by hourly, weekly or monthly. It can even show as

detailed as by millisecond. In our case, millisecond is not defined in our timestamp creation, so it should be added there.

9.4.4 Problems

Apparently, the working environment has been using Chrome's "dev channel" versions, which is unstable version. Dev channel is usually couple versions ahead from the stable version, it gives the user a chance to use and try unreleased new features and fixes right away, of course the catch is that those are unstable. The problem that occurred was that the parsing conditions didn't seem to match anymore when using the new profiling files in the developer browser.

Table 2 Chrome's version history (Wikipedia, 2017d)

Major version	**Release date (Linux, macOS, Windows)**	**Layout Engine**	**V8-engine version**	**Info**
56.0.2924	2017-01-25	Blink 537.36	5.6.326	
57.0.2987	2017-03-09	Blink 537.36	5.7.492	
58.0.3029	2017-04-19	Blink 537.36	5.8.283	
59.0.3071	2017-06-05	Blink 537.36	5.9.211	Current stable version (as checked 20.6.2017)
60.0.3112	2017-06-08	Blink 537.36	6.0	Current Beta channel
61.0	2017-06-09	Blink 537.36	6.1	Current Dev channel
61.0	2017-04-15 (macOS & Windows)	Blink 537.36	6.1	Current Canary channel

Tracing files done in Dev Channel version, Chromium 61.0, did not have any memory-infra categories nor "periodic_interval" as item names. After reverting to stable Chromium version 59.0.3071, items with previous conditions were found. Just by looking at the version numbers (Table 2), we can see that V8-engine's version has changed in the middle,

while it might be relevant but it still shouldn't be causing the problem as the previous category was not related to V8. Going through the changelog of version 61.0 showed that there were some changes to the memory-infra, one that might be interesting in our case is the one presented in the following figure 9.14.

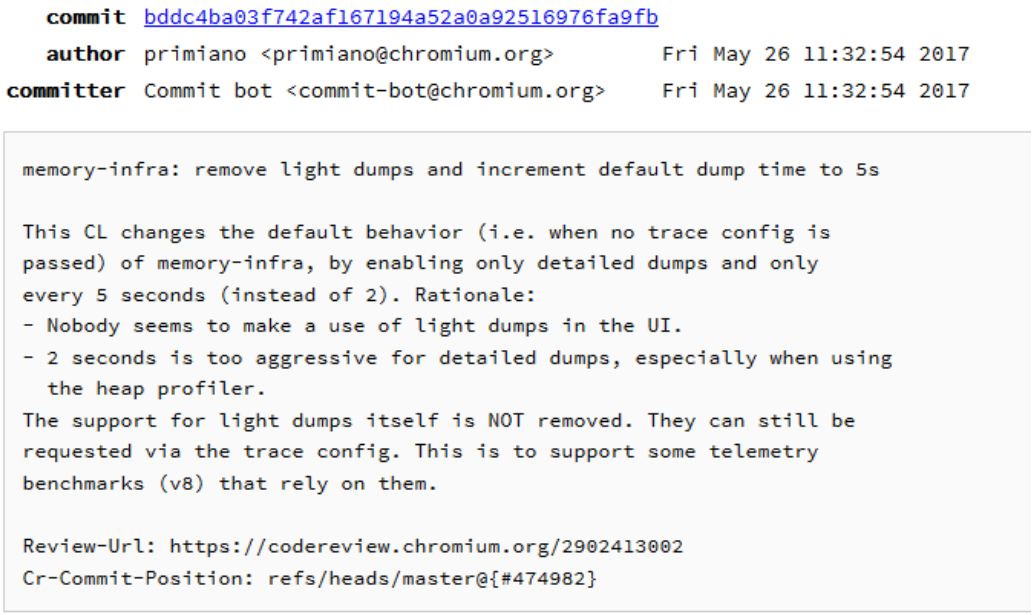


Figure 9.14 From changelog of Chromium (Dev Channel) version 61.0

However, this should also not apply in our case as it regards only "light dumps". Dumps that we are visualizing have been even over 600kb, so those should not be classified as light dumps.

The exact solution for this problem remains open. It might be that it happens only when not using stable version of Chromium, which means that using a stable version should be one of the requirements. For now, using stable version seems as a good time saving solution for this project. We should wait and see if the problem comes again when the stable version gets major updates.

9.5 Jenkins CI configuration - Final automation

To summarize previous parts, we should now have two working shell commands.

1. `npm run test-webdriver -- --hub-host localhost -e chromePro-file --tests <test/component>`
2. `node profiling-parser.js <path/to/the/file.json>`

The 1st command works if you have localhost and selenium running, it starts up the tests with the tracer and the outcome is a raw JSON type file. The 2nd command will start the

parser script that includes the POST request to the host. We parse the raw JSON file with the custom conditions. The conditions for this project were introduced in chapter 10.4.1 together with creating index for ElasticSearch.

After this, we'll create a new job, "freestyle-project", in Jenkins CI. We can set the Jenkins CI to build periodically e.g. every hour, night or day. A nightly build setting was a request from the commissioner. We shall name this new Jenkins CI job/project "nightwatch-memory-profiling". We define the source code's repository and branch. In our case the testing should be done with master branch, but other individual branches are also possible.

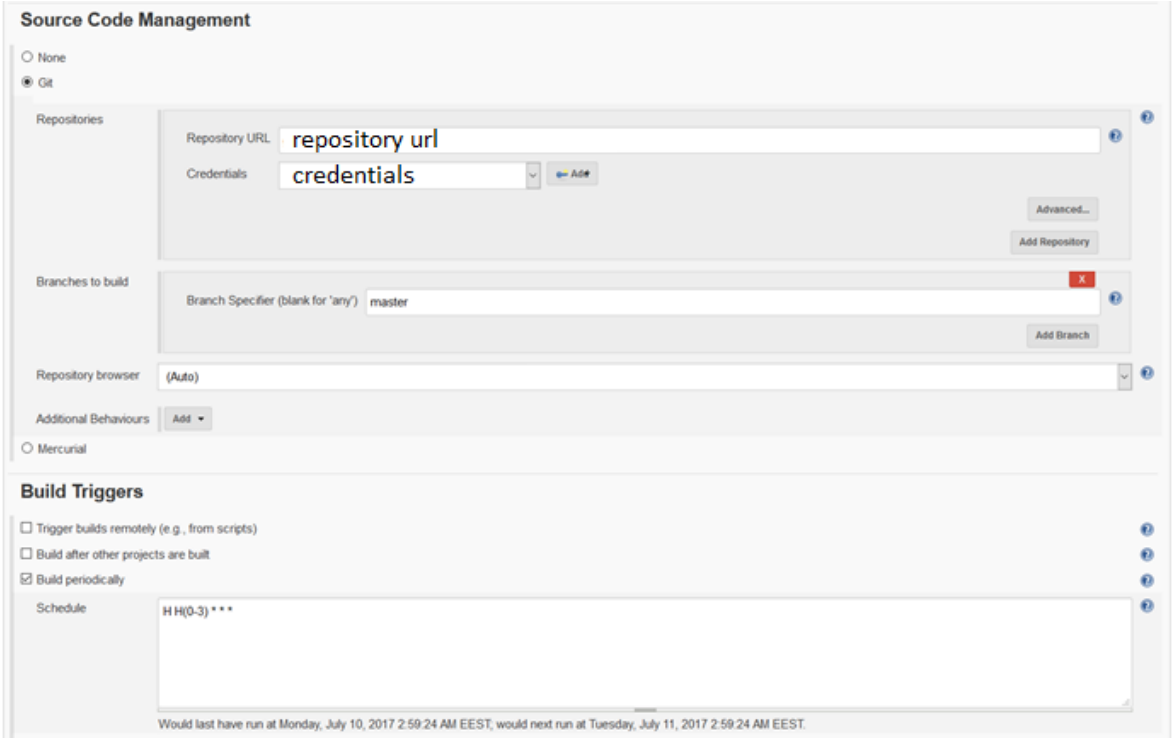


Figure 9.15 Defining repository and build schedule in Jenkins CI.

Under Jenkins CI's "Build Environment", we check "Add timestamps to the Console Output". Then, under "Build" section, we input our commands under the "Execute Shell" and save the changes. Now it should run the memory profiling every night. We want the build to run nightly because during the daytime the test pipeline is busy with other functional tests and development. Nightly builds won't affect the normal daytime developers by slowing down connection or test pipelines.

This is at least how it should work. Unfortunately, due to the fact that we cannot get Selenium to the Jenkins CI build, at least not in very easy way, we cannot really finish the ac-

tual automation yet. Jenkins CI builds happens in a total “clean-state”, the testing environment won’t have anything that you haven’t given to it. A solution was to get Docker, a container platform, to have the Selenium and other dependencies.

Docker has isolated containers that have everything that is needed to make a software run. It can include e.g. code, runtime, system tools and libraries, basically anything you can install on a server. Docker is not a VM though, it does not pack a whole operating system. Docker “guarantees that software will always run the same, regardless of where it’s deployed” (Docker, 2017).

As there were some issues with using Docker, because the CI environment itself was already a virtual host, the automation will have to wait till it is available, meaning that Docker will be put as requirement. After it is available, it should be quite easy to run the build on Jenkins, just need to add the new or updated set of shell commands in the job.

10 Conclusion

We could say that the implementation was almost 100% success, as the only problem is to get the automation script running. As briefly mentioned before, there are still many things that can be improved and fixed, but the base of “Memory profiling” is now there, so future improvements won’t need as much of research, hopefully. Of course, it is important for Team Terra, and also Nexterday UI Platform to always have high quality which is why continuous improvement is also part of the continuous integration practice.

We can’t yet say or see the “statistics” of how much this project has helped the development process and code quality as in performance, but it is supposed to improve the feedback loop. In Comptel’s office, it is quite common to use these “radiator screens” to show Jenkins’ test builds where different colors indicate the status of the build, e.g. red means failure. Commissioning party was thinking if it would be possible to show Kibana view in the radiator screen. This way developers could see if there were some significant changes, for example developers can get a visual que from it if there’s a sudden drop, so they can make some further investigations on a possible memory leak.

Automating memory profiling helps developers to get feedback faster, but at least with current parsing, it’s doesn’t show enough data to pinpoint the problem to specific part of a code. Of course, by running the script separately against individual components, we get to know in which component has the problem. As mentioned in the theory part of test automation, Chapter 6, manual tests are still need and they cannot be replaced fully.

In this case, automation is still cost-effective solution, it does save time and it doesn’t seem to be very expensive to upkeep. The setting up is little bit more complex than expected, as it needs quite many tools (ElasticSearch, Kibana, Jenkins etc.) to show the data as infographic. The memory capacity for Kibana and ElasticSearch is also a possible future risk. There has not been a decision yet on the exact time span of profiling’s to be stored but eventually the old data is to be deleted entirely.

Looking back to the implementation plan, I would say that it was overall quite good and well followed. We could’ve maybe added a phase for analyzing the gotten data as that took quite a lot of time. In the theory part, it was also brought up how tests should not be automated before the test cases are well thought through, in this case we didn’t have to think it that much as the tests are existing Nexterday UI Platform tests. Of course, we should look more into on what kind of data we are parsing, what info from the memory do

developers want and need, currently we only collect certain malloc data that matches the search condition, the feedback doesn't tell us that much yet. The automation might not be as cost-effective as one would want just yet, but it definitely has the ingredients for further improvements and development. Previously in the chapter 6.3 we went through some basic practices and rules on when automation is good or bad idea, and it was mentioned that automation is good if it's for performance testing. Even with current simple parsing, we are already getting some feedback on how much memory is being allocated, we only need to "enlarge the scope". We want to have more memory related data.

As it was revealed in chapter 9.5 that we can't yet finalize the automation, we still have the feature ready and all that is needed is to have the Docker. The commissioning party estimated that they could get everything done for this feature in one day approximately, but it's more of a timing question.

11 General afterthoughts and self-evaluation

During the development of this thesis there were not only office renovation but also huge architectural changes in the Nexterday UI Platform, as well as in development tools. For example moving from Mercurial to git and smaller repository sizes.

Originally, I thought that I wouldn't need to attend daily scrum, this was also recommended by my commissioning party's supervisor because I technically was not directly working on Nexterday UI Platform. But now after I think about it, not attending to daily scrums did backfire a little. This is because I was oblivious to big changes that affected my developing environment. Nothing lethal errors happened, thanks to version control I could always rollback to previous revisions, but I guess I could have saved some time if I were to know some things beforehand.

Most frustrating thing, aside of inexperienced coding, was all those dependencies that affected to the work. If I were to come up with a solution for one thing, it would later come to me that the fix can't be implemented because of some other thing.

Being my own project manager was little tough to me. I got often stuck on things that might've not been relevant at that point of development, this took a lot of time. I think that as a developer, you try to see the big picture, you might realize that most likely the current implementation does not work that well with other things and dependencies in the future and you want to avoid that. But of course, the main idea was to get out "a minimum viable product", so that this memory profiling has the minimum necessities to run and work.

As expected, learning new tools really takes a lot of time. Also, as a beginner in JavaScript, coding also was very slow. However, as a lot of studying has happened, hopefully it also means that I have learned a lot. This project has given me a good insight to development process and basics to many tools e.g. Karma and Nightwatch.js. I also got educated well on test automation.

References

Ambler, S.W. 2017a. Introduction to Test Driven Development (TDD). URL: http://agiledata.org/essays/tdd.html. Accessed: 24.2.2017.

Ambler, S.W. 2017b. Scott W. Ambler's Home Page. URL: http://www.ambysoft.com/scottAmbler.html. Accessed: 23.2.2017.

Ambler, S.W. 2017c. Examining the Agile Cost of Change Curve. URL: http://www.agilemodeling.com/essays/costOfChange.html. Accessed: 24.2.2017.

Basques, K. 2017. Fix Memory Problems. URL: https://developers.google.com/web/tools/chrome-devtools/memory-problems/. Accessed: 7.4.2017.

Beverloo. P. 2017. List of Chromium Command Line Switches. URL: http://peter.sh/experiments/chromium-command-line-switches/. Accessed: 15.4.2017.

Bhattacharya, A. & Sundar, K.S. 2007. Memory leak patterns in JavaScript. URL: https://www.ibm.com/developerworks/library/wa-memleak/. Accessed: 3.4.2017.

Comptel 2017. Life is Digital Moments. URL: http://www.comptel.com/. Accessed 8.2.2017.

Developers. 2017. Chrome V8. URL: https://developers.google.com/v8/. Accessed: 13.3.2017.

Dijsktra, B. 26.6.2014. The test automation pyramid. On test automation. URL: http://www.ontestautomation.com/. Accessed: 16.3.2017

dsinclair@ & Nduca@ 2016. Trace Event Format. URL: https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview. Accessed: 21.3.2017.

Docker, 2017. What is Docker? URL: https://www.docker.com/what-docker. Accessed: 10.7.2017

Fernandes, J. & Fonzo, A.D. 2013. When to Automate Your Testing (and When Not To). URL: http://www.oracle.com/technetwork/topics/qa-testing/whatsnew/when-to-automate-testing-1-130330.pdf. Accessed: 1.3.2017.

Fewster, M. & Graham, D. 1999. Software Test Automation. ADDISON-WESLEY & ACM Press. New York.

GitHub, 2017a. Redux. URL: https://github.com/reactjs/redux. Accessed: 9.6.2017.

GitHub, 2017b. Bluebird. URL: https://github.com/petkaantonov/bluebird. Accessed: 9.6.2017.

Github, 2017c. Babel. URL: https://github.com/babel/babel. Accessed: 9.6.2017.

Github, 2017d. Webpack. URL: https://github.com/webpack/webpack. Accessed: 9.6.2017.

Github, 2017e. Karma. URL: https://github.com/karma-runner/karma. Accessed: 9.6.2017.

GitHub, 2017f. Nightwatch.js. URL: https://github.com/nightwatchjs/nightwatch. Accessed: 9.6.2017.

Google Dictionary, Firefox-browser. 2017a. URL: https://www.google.fi/search?q=API&ie=utf-8&oe=utf-8&client=firefox-b&gfe_rd=cr&dcr=0&ei=OMm4WbTLNcmFZrKIqMAI. Accessed: 8.9.2017.

Google Dictionary, Firefox-browser. 2017b. URL: https://www.google.fi/search?hl=en&q=Dictionary#dobs=node. Accessed: 8.9.2017.

Google Dictionary, Firefox-browser. 2017c. URL: https://www.google.fi/search?hl=en&q=Dictionary#dobs=memory%20leak.

Graham, D., Veenendaal, E.V., Evans, I. & Black, R. 2008. Foundations of Software Testing. Cengage Learning EMEA. London.

GraphicsMagick. 2017, URL: http://www.graphicsmagick.org/. Accessed: 25.8.2017.

Gregory, J. & Crispin, L. 2014. More Agile Testing. Addison-Wesley. New York.

Kerr, D. 2015. Fixing Memory Leaks in AngularJS and other JavaScript Applications. URL: https://www.codeproject.com/Articles/882966/Fixing-Memory-Leaks-in-AngularJS-and-other-JavaScr. Accessed: 2.3.2017.

Kearney, M. 2017. Memory Terminology. URL: https://developers.google.com/web/tools/chrome-devtools/memory-problems/memory-101. Accessed: 10.2.2017.

Lewis, P. 2017. Rendering Performance. URL: https://developers.google.com/web/fundamentals/performance/rendering/. Accessed: 20.3.2017.

Lewis, W.E. 2000. Software Testing and Continuous Quality Improvement. AUERBACH. Florida.

Lodash, 2017. A modern JavaScript utility library delivering modularity, performance & extras. URL: https://lodash.com/. Accessed: 5.6.2017.

Looper, J. 2015. A Guide to JavaScript Engines for Idiots. URL: http://developer.telerik.com/featured/a-guide-to-javascript-engines-for-idiots/. Accessed: 23.5.2017.

Moment.js. 2017. URL: https://momentjs.com/. Accessed: 25.8.2017.

Nguyen, H.Q. 2001. Testing Applications on the Web. WILEY. New York.

npm, 2017a. React. URL: https://www.npmjs.com/package/react. Accessed: 18.5.2017.

npm, 2017b. Enzyme. URL: https://www.npmjs.com/package/enzyme. Accessed: 18.5.2017.

npm, 2017c. Chai. URL: https://www.npmjs.com/package/chai. Accessed: 18.5.2017.

Odvarko, J.H & Walker, J. 8.2.2016. Merging Firebug into the built-in Firefox Developer Tools. Firebug web development evolved - development blog. URL: https://blog.getfirebug.com/2016/02/08/merging-firebug-into-the-built-in-firefox-developer-tools/. Accessed: 13.2.2017.

Osmani, A. 2014. JavaScript Memory Management Masterclass. URL: https://speakerd.s3.amazonaws.com/presentations/5e0b4c70100801328edc52d4282366ee/Speaker-Deck-_Memory_Management_Masterclass__DevTools___2_.pdf. Accessed: 9.2.2017.

Peyrott, S. 26.1.2016. 4 Types of Memory Leaks in JavaScript and How to Get Rid Of Them. Auth0 blog. URL: https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/. Accessed: 27.3.2017.

Puckett, D. 2011. The Importance of Agile Feedback Loops. URL: https://www.infoq.com/news/2011/03/agile-feedback-loops. Accessed: 25.4.2017.

Rouse, M. 2014. Behaviour-Driven Development (BDD). URL: http://searchsoftwarequality.techtarget.com/definition/Behavior-driven-development-BDD. Accessed: 2.3.2017.

SeleniumHQ. 2017. What is Selenium? URL: http://www.seleniumhq.org/. Accessed: 10.3.2017.

SmartBear. 2017. Why Automated Testing? URL: https://smartbear.com/learn/automated-testing/. Accessed: 22.2.2017.

Telerik 2017. URL: http://www.telerik.com/. Accessed: 1.2.2017.

Vänttinen, A. 5.4.2016. Nexterday Acknowledged as a Campaign of the Year by Business Intelligence Group. Compelling Conversations - ComptelBlog. URL: http://comptelblog.com/2016/04/nexterday-award/. Accessed: 8.2.2017.

W3Schools 2017a. JSON - Introduction. URL: https://www.w3schools.com/js/js_json_intro.asp. Accessed: 16.3.2017.

W3Schools 2017b. Browser Statistics. URL: https://www.w3schools.com/Browsers/default.asp. Accessed: 16.3.2017.

Wikipedia, 2017a. Document Object Model. URL: https://en.wikipedia.org/wiki/Document_Object_Model. Accessed: 14.4.2017.

Wikipedia, 2017b. SpiderMonkey. URL: https://en.wikipedia.org/wiki/SpiderMonkey. Accessed: 13.3.2017.

Wikipedia, 2017c. Chakra (Jscript engine). URL: https://en.wikipedia.org/wiki/Chakra_(JScript_engine). Accessed: 13.3.2017.

Wikipedia, 2017d. Google Chrome Version History. URL: https://en.wikipedia.org/wiki/Google_Chrome_version_history. Accessed: 20.6.2017.

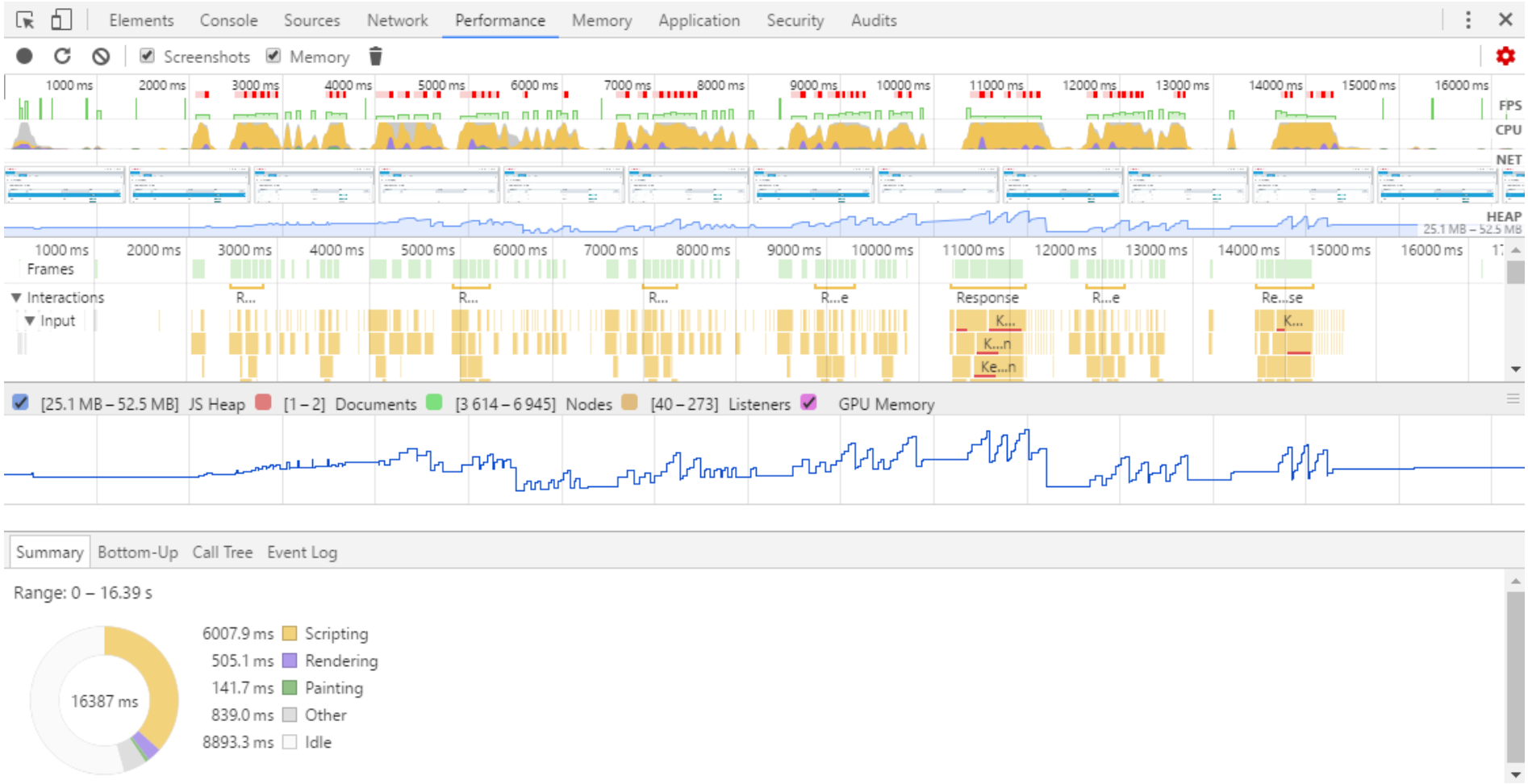
Wikipedia. 2016. OSS/BSS, URL: https://en.wikipedia.org/wiki/OSS/BSS. Accessed: 5.2.2017.

Appendices

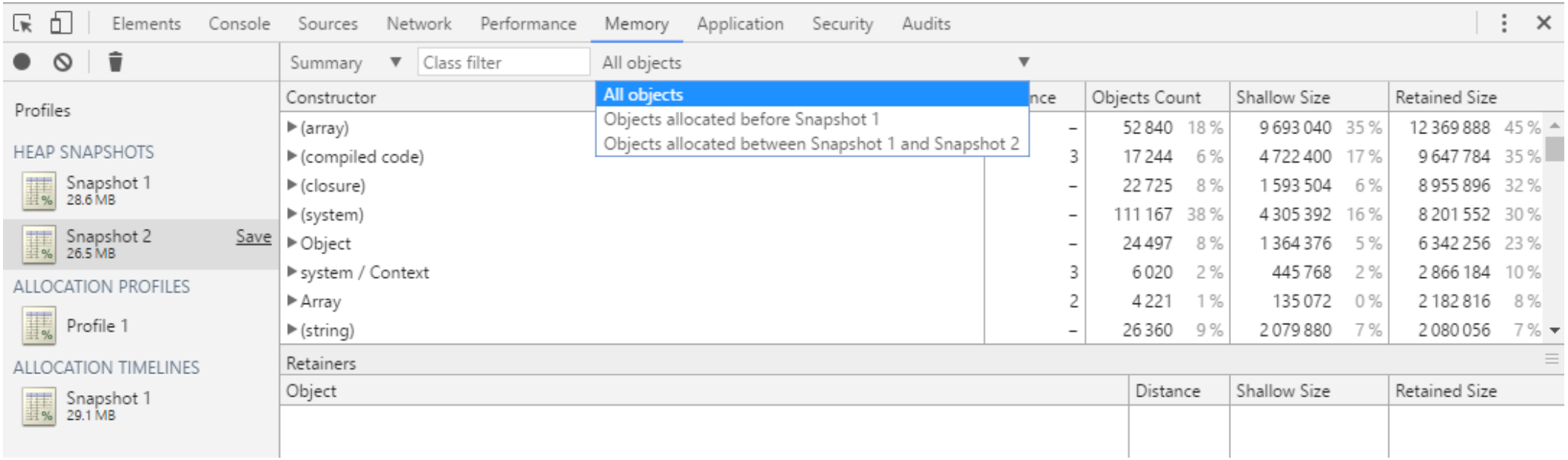
Appendix 1. Browser Comparison Table

#	Feature	Google Chrome	Internet Explorer 11/Edge	Firefox
1.	Performance Profiler	✓	✓	✓
	FPS usage report	✓	✓ Live graph	✓
	Heap Report	✓	✗	✗
	CPU Usage graph	✓	✓	✓
	Flame chart	✓	✗	✓
	Report shows screenshots of the page	✓	✗	✗
	Heap graph	✓	✗	✗
	Details of the event	✓ Also graphical	✓	✓ Even tells the reason of GC
	Manual garbage collection initiation	✓	✗	✗
	Filtering of graphical view - To be able to show more or less graphical information	✓	✗	✓
2.	Memory Profiler	✓	✓	✓
	Heap Allocation Profile	✓ Live graph	✗	✗
	Allocation Timeline	✓	✓ ☹ Snapshots from previous run will be deleted	✓
	Taking snapshots	✓	✓	✓ In Performance profiler
	Snapshot comparison	✓	✗	✓
3.	Code coverage. How much CSS and JS was used vs how much was loaded	✓	✗	✗
4.	Import & Export the report	✓	✓	✓
5.	Source Mapping. - Clicking the function leads to the debugger/sources -tab and shows the source code	✓	✓	✓
6.	JavaScript Engine	V8	IE11: Trident Edge: Chakra	Spidermonkey & Rhino

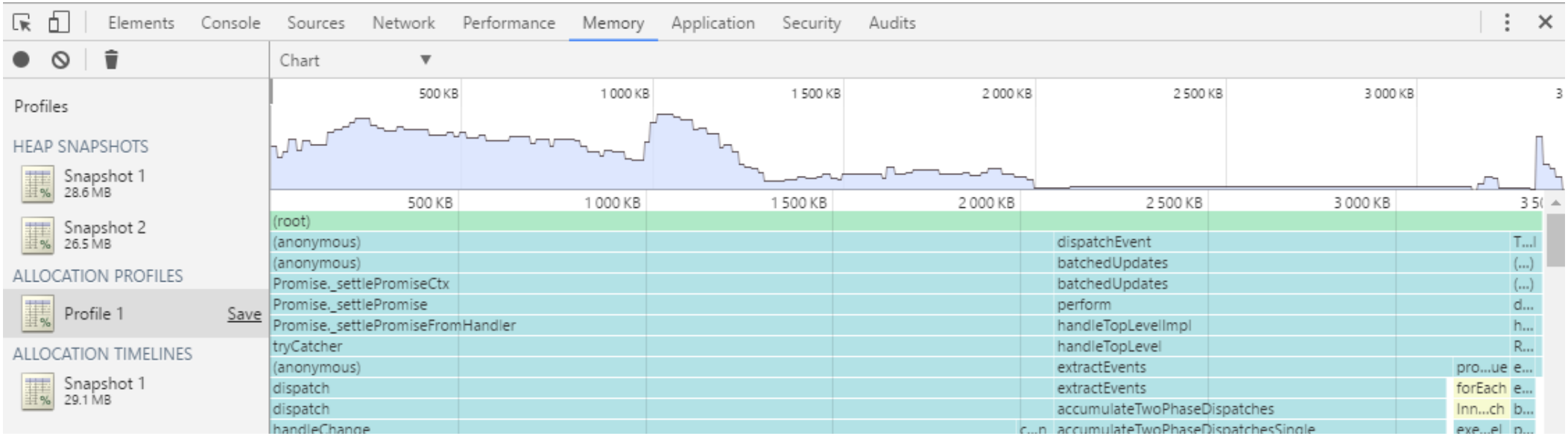
Appendix 2. Google Chrome DevTool Memory Profiling



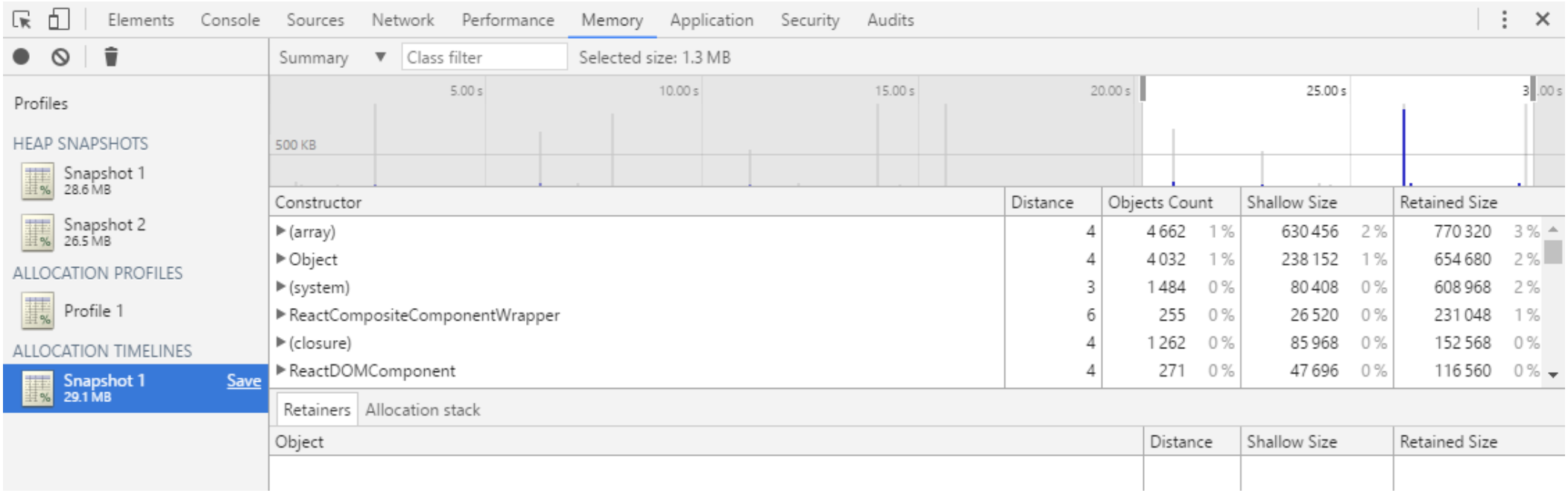
Appendix 3. Google Chrome Heap snapshot



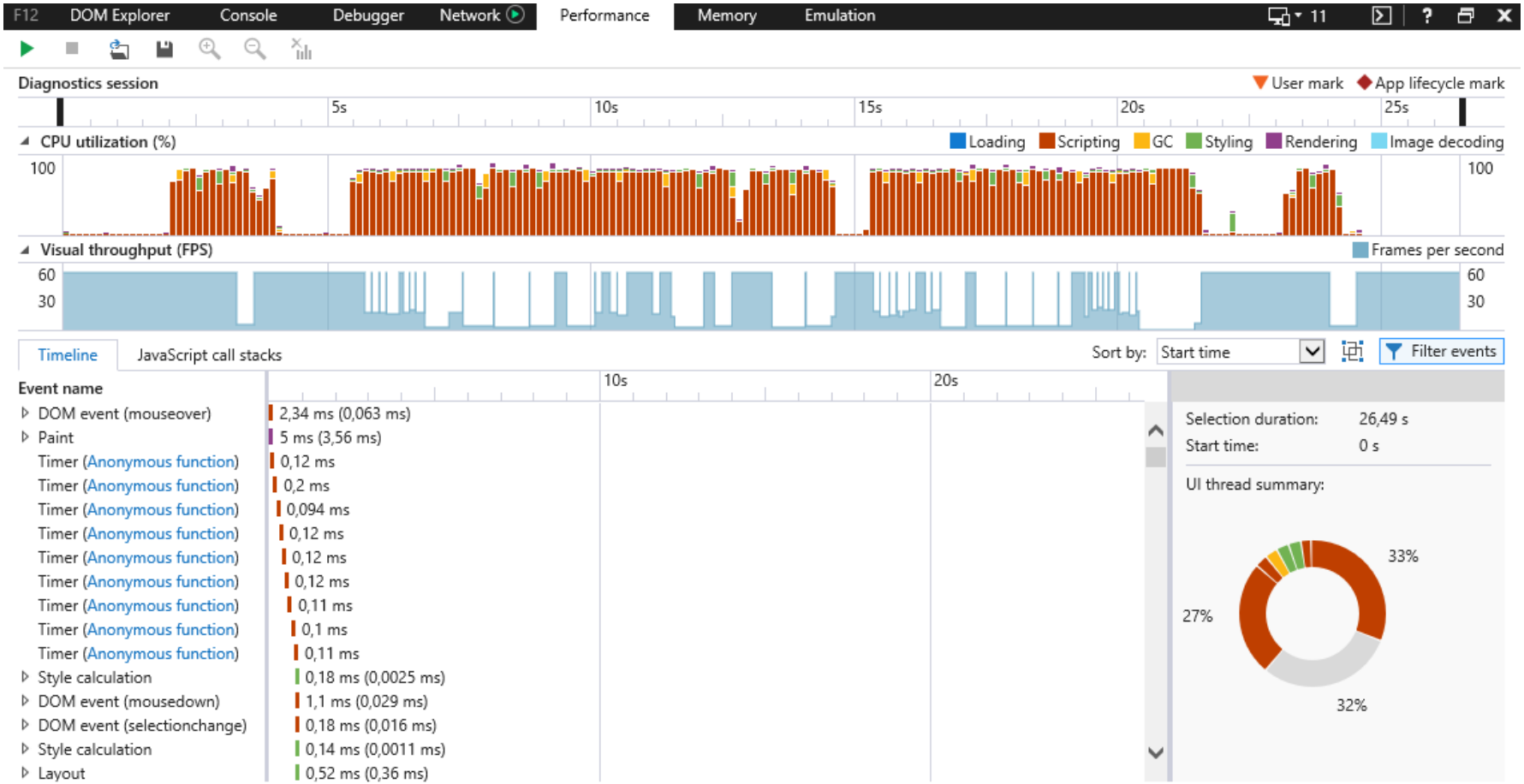
Appendix 4. Google Chrome Allocation Profiling



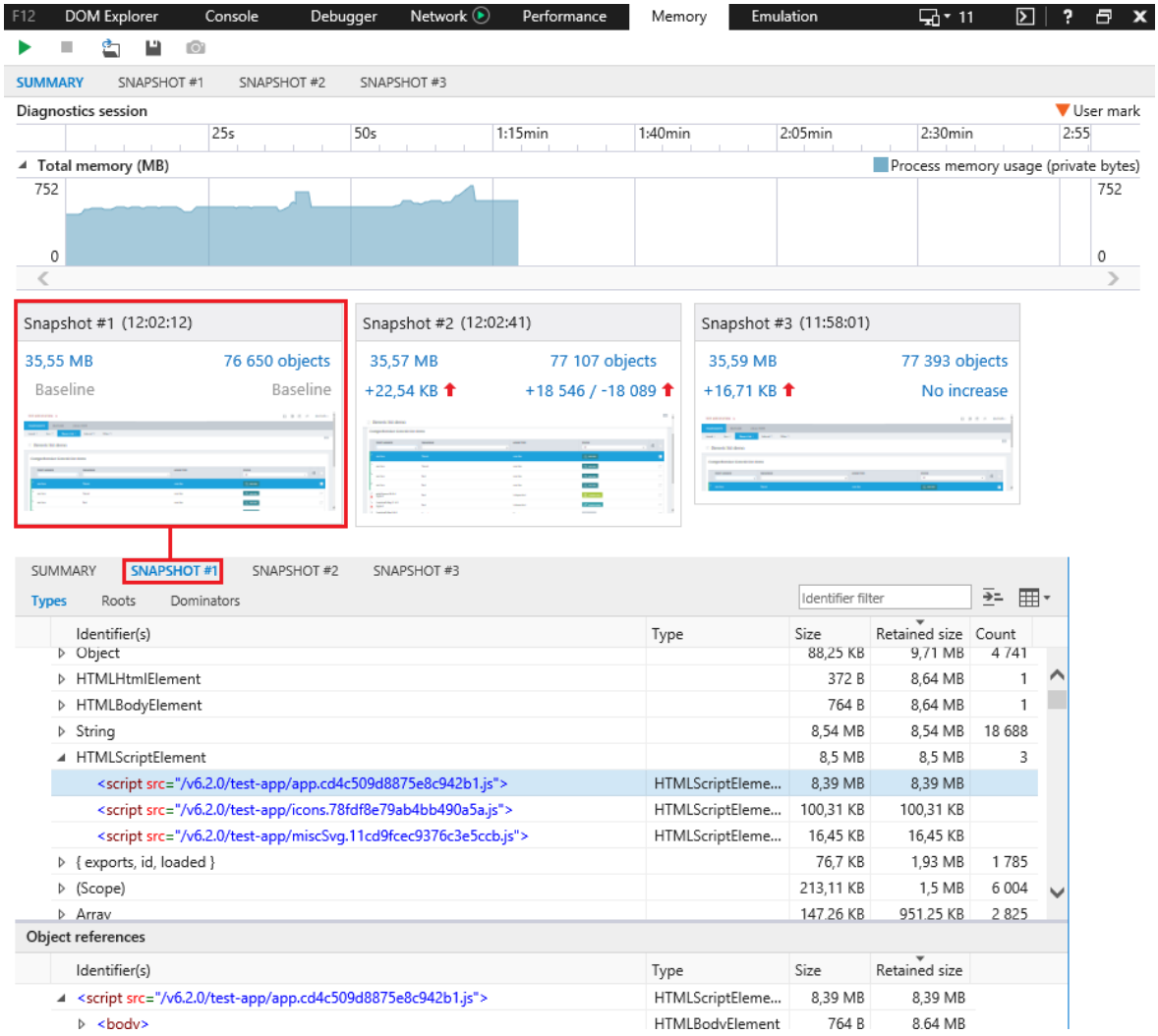
Appendix 5. Google Chrome Allocation Timeline



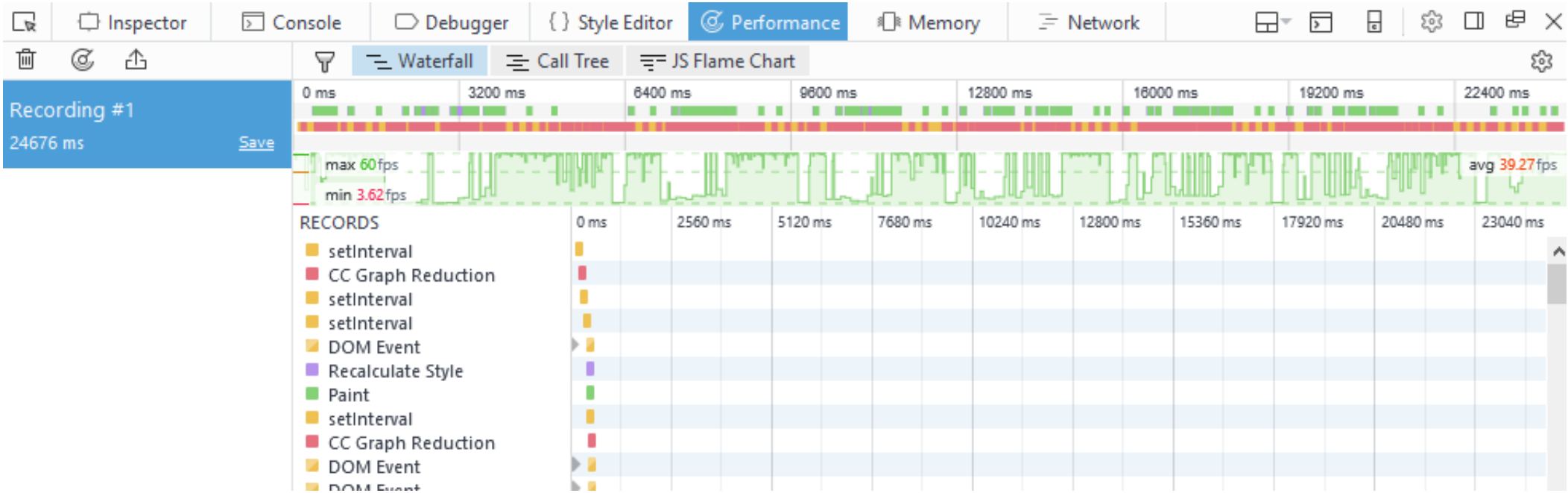
Appendix 6. Edge/IE11 Devtool Performance Tab



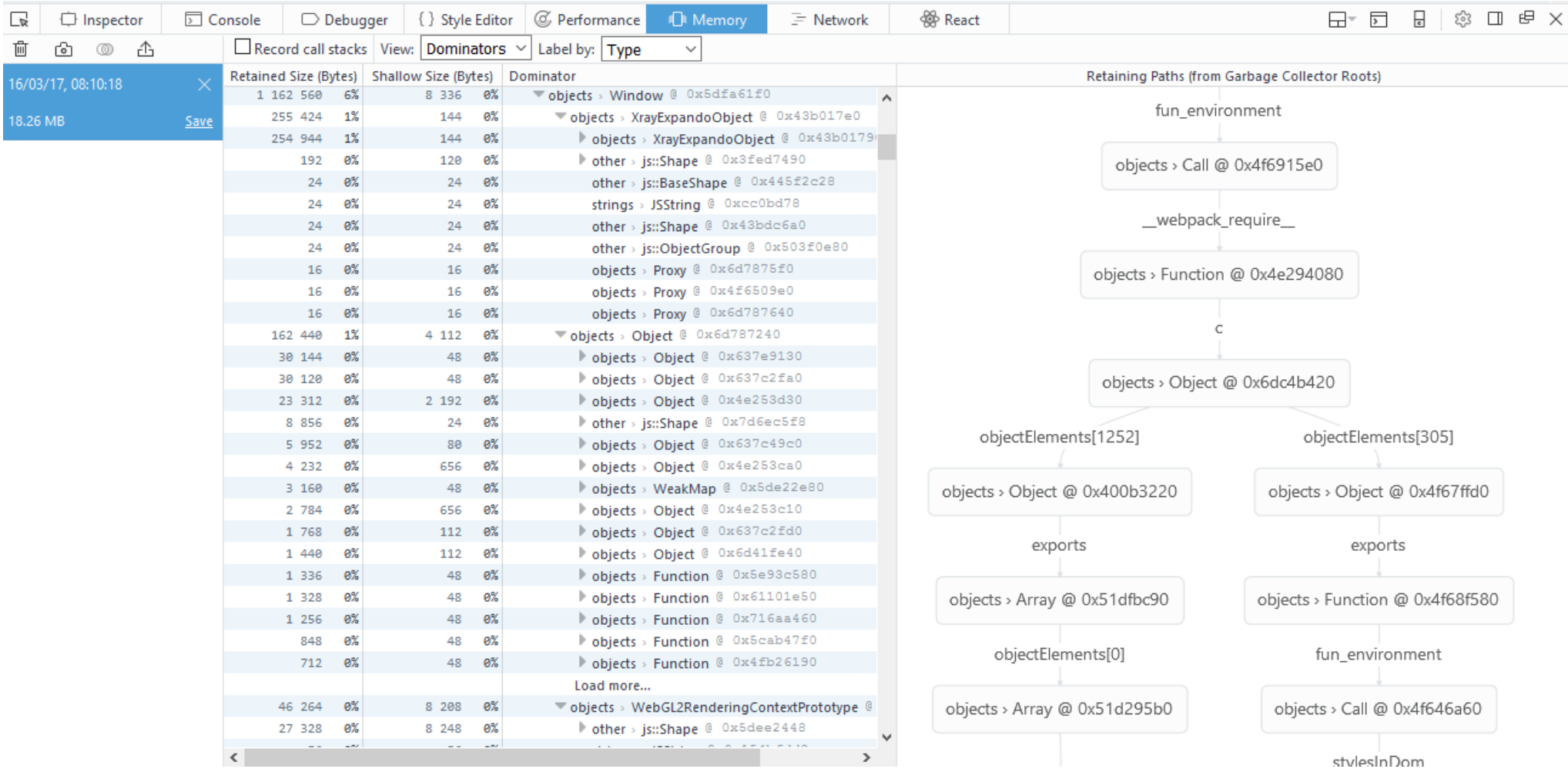
Appendix 7. Edge/IE11 Memory Tool



Appendix 8. Firefox Performance Tool



Appendix 9. Firefox Memory Tool



Appendix 10. profiling-parser.js

```
const http = require('http');
const fs = require('fs');
const JSONStream = require('JSONStream');
const es = require('event-stream');
const path = require('path');
const program = require ('commander');
const moment = require ('moment');

program
  .usage('Path to file')
  .parse(process.argv);



if (program.args.length !== 1) {
  console.error('ERROR! Not a valid argument.')
  process.exit();
}

function fsExists(path) {
  try {
    fs.accessSync(path);
    return true;
  } catch (error) {
    console.log('Are you sure the given path is right? :', path
);
    return false;
  }
}


if (fsExists(program.args[0])) {

  const jsonData = program.args[0];
  const basename = path.basename(jsonData);

  const split = basename.split('-');
  const date = moment(split[0], 'YYYYMMDD').format('YYYY-MM-DD
');
  const time = moment(split[1], 'HHmmss').format('HH:mm:ss');
  const timestamp = `${date}${time}`;
```

```
   const componentName = split.pop().split('.').shift();

   const getStream = function () {
     const stream = fs.createReadStream(jsonData, {encoding:
'utf8'});
     const parser = JSONStream.parse('*');
     return stream.pipe(parser);

   };
   getStream()
     .pipe(es.mapSync(function (d) {
      // let timestamp;
       if (!(d instanceof Array)) {
        //  timestamp = d['trace-capture-datetime']; //gives GMT+0
and sometimes HH:M:s. Time is different than the one created "man-
ually" aka. file creation time.
         //console.log(d);

         return;
       }

       //Filtered data will be visualized
       const list = d.filter(function (item) {
         return item && item.name && item.name === 'periodic_in-
terval' && item.args;
       }).map(function (item) {
         // NOTE: if malloc.attrs.size.value is hexadecimal, you
need to convert it into decimals!!
         const mem = parseInt(item.args.dumps.allocators.mal-
loc.attrs.size.value);
         if (!isNaN(mem)) {
           return {
             'pid': item.pid,
             'tid': item.tid,
             'ts': item.ts,
             'cat': item.cat,
             'name': item.name,
             'memory_allocated': mem,
             'run_time': timestamp,
             'component': componentName,
           };
         }
         return null;
       }
```

```
    ).filter((item) => {
      return item !== null;

    });

      console.log(d.length, list.length);

      if (list.length === 0) {
        console.log('Could not find data that match the require-
ments');
        return;
      }

      const outgoing = {
        data: list

      };

      const out = JSON.stringify(outgoing, null, '\t');

      //console.log(out);

      //create a new JSON file to store parsed data
      const filename = path.join(`parsed-${basename}`);
      fs.writeFileSync(`${filename}`, out, 'utf8');


          //Using REST API to send data to host
      const options = {
        hostname: 'hrh7ds185.comptel.com',
        port: 9200,
        path: '/tracing/entry',
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        }
      };

      // one http.request() per list
      const oneRequest = (outgoingData) => {
        const req = http.request(options, (res) => {
          console.log(`STATUS: ${res.statusCode}`);
          console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
          res.setEncoding('utf8');
```

```
            res.on('data', (chunk) => {
              console.log(`BODY: ${chunk}`);
            });
            res.on('end', () => {
              console.log('No more data in response.');
            });
          });

          req.on('error', (error) => {
            console.error(`problem with request: ${error.mes-
sage}`);
          });

        // write data to request body
          req.write(outgoingData);
          req.end();

        }; //end of oneRequest

        list.forEach(function (item) {
          const showStatus = JSON.stringify(item);
          oneRequest(showStatus);
        });

      }
));
 }
```