Analyzing web applications: An empirical study

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Alex Nederlof
born in Zevenhuizen, the Netherlands
Analyzing web applications: An empirical study

Abstract

Due to the increased usage of JavaScript in web applications and the speed at which web technologies and browsers are evolving, web applications are becoming ever more complex. Our hypothesis is that these applications contain severe errors, take unnecessary performance penalties, and violate accessibility standards. This study analyzes such errors and tries to quantify the need for a tool that can help developers make web applications with less errors.

The research is conducted by first showing how much of the DOM is modified after the initial page load. This could indicate that static analysis does not suffice anymore. After that we quantify the amount of faults in the web application.

The research is done on 3,422 sites randomly selected from the internet. They were automatically analyzed using a crawler.

We conclude that the use of static analysis tools to prevent these faults does not suffice anymore. The errors and accessibility standard violations happen in dynamically generated DOM, which are not detectable by static analysis. The performance penalties are only visible through dynamic analysis.

We propose to develop a random testing tool based on a crawler that checks for these errors.

Our main contributions are the design of such a tool, the large dataset that we have gathered during this research and the quantification of both the level of dynamism of modern web applications and the fault-proneness of these applications due to this dynamism.

Thesis Committee:

Chair: prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
External supervisor: prof. dr. ir. A. Mesbah, UBC
Committee Member: dr. A. Zaidman, Delft
This thesis was written as part of my graduation for the Master of Science degree at the Technical University in Delft.

I want to express my mad love and thanks to both my parents, my girlfriend Marlon, and my sister and brother Nina and Olaf for always believing in me.

I am forever grateful for the excellent guidance, support, inspiration and trust that I have received from both Arie van Deursen and Ali Mesbah during my education.

I would also like to thank Bas Verheij for all our insightful discussions, the “UniversiteitFonds Delft”\footnote{http://universiteitsfonds.tudelft.nl/} for funding my trips to Canada, Georgios Gousios for letting me run my experiment on his server, and all my friends for all the good times we had during my study.
“Beware the irrational, however seductive. Shun the transcendent and all who invite you to subordinate or annihilate yourself. Distrust compassion; prefer dignity for yourself and others. Don’t be afraid to be thought arrogant or selfish. Picture all experts as if they were mammals. Never be a spectator of unfairness or stupidity. Seek out argument and disputation for their own sake; the grave will supply plenty of time for silence. Suspect your own motives, and all excuses. Do not live for others any more than you would expect others to live for you.”

- Christopher Hitchens
Contents

Preface iii

Contents v

1 Introduction 1

2 Background 3
  2.1 HTML 3
  2.2 JavaScript 5
  2.3 CSS 5
  2.4 Web application testing 6

3 Crawlers 9
  3.1 Classic crawlers 9
  3.2 JavaScript-enabled crawlers 10
  3.3 Challenges for JavaScript-enabled crawlers 10
  3.4 State navigation 11
  3.5 Invoking state changes 11
  3.6 Unreachable states 12
  3.7 Crawljax 13

4 Experimental set-up 15
  4.1 Conceptual set-up 15
  4.2 Implementation 16

5 Scaling up Crawljax 19
  5.1 Project organization 19
  5.2 New features 20
  5.3 Performance 24
  5.4 Overall statistics of performed work 25

6 Project big crawl 29
  6.1 Distribution of crawls 29
  6.2 Collection of statistics 31
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3 Development workflow</td>
<td>32</td>
</tr>
<tr>
<td>6.4 The resulting data set</td>
<td>33</td>
</tr>
<tr>
<td>7 Measuring Client-side DOM manipulation</td>
<td>35</td>
</tr>
<tr>
<td>7.1 States per URL</td>
<td>35</td>
</tr>
<tr>
<td>7.2 States without a URL</td>
<td>36</td>
</tr>
<tr>
<td>7.3 Post-load DOM comparison</td>
<td>37</td>
</tr>
<tr>
<td>8 The state of the web</td>
<td>39</td>
</tr>
<tr>
<td>8.1 HTML-5</td>
<td>39</td>
</tr>
<tr>
<td>8.2 Resource compression</td>
<td>41</td>
</tr>
<tr>
<td>8.3 Mobile ready</td>
<td>41</td>
</tr>
<tr>
<td>8.4 Volume</td>
<td>42</td>
</tr>
<tr>
<td>9 Operational errors</td>
<td>45</td>
</tr>
<tr>
<td>9.1 W3C Errors</td>
<td>45</td>
</tr>
<tr>
<td>9.2 Double ID tags</td>
<td>46</td>
</tr>
<tr>
<td>9.3 No Doctype</td>
<td>49</td>
</tr>
<tr>
<td>9.4 Broken layout</td>
<td>49</td>
</tr>
<tr>
<td>10 Performance analysis</td>
<td>53</td>
</tr>
<tr>
<td>10.1 Bottlenecks</td>
<td>53</td>
</tr>
<tr>
<td>10.2 Caching strategies</td>
<td>54</td>
</tr>
<tr>
<td>10.3 Compress Components</td>
<td>55</td>
</tr>
<tr>
<td>10.4 Put stylesheets at the top</td>
<td>55</td>
</tr>
<tr>
<td>10.5 Make scripts unblocking</td>
<td>55</td>
</tr>
<tr>
<td>11 Accessibility</td>
<td>59</td>
</tr>
<tr>
<td>11.1 Meta information</td>
<td>59</td>
</tr>
<tr>
<td>11.2 Navigation assistance</td>
<td>61</td>
</tr>
<tr>
<td>12 Discussion</td>
<td>63</td>
</tr>
<tr>
<td>12.1 Threats to validity</td>
<td>63</td>
</tr>
<tr>
<td>12.2 Recommendations for further development</td>
<td>64</td>
</tr>
<tr>
<td>12.3 Recommendations for further research</td>
<td>65</td>
</tr>
<tr>
<td>13 Related work</td>
<td>67</td>
</tr>
<tr>
<td>13.1 Crawling for exposure</td>
<td>67</td>
</tr>
<tr>
<td>13.2 Crawlers as a testing or analysis tool</td>
<td>68</td>
</tr>
<tr>
<td>13.3 Coping With JavaScript’s Dynamism</td>
<td>68</td>
</tr>
<tr>
<td>14 Concluding remarks</td>
<td>71</td>
</tr>
<tr>
<td>Bibliography</td>
<td>75</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In 2012 the United Nations declared the Internet a basic human right. Article two of the motion states:

“General Assembly at its sixty-sixth session, on freedom of expression on the Internet, [...] recognizes the global and open nature of the Internet as a driving force in accelerating progress towards development in its various forms”[1]

Due to its openness, versatility, and flexibility the Internet has grown to a necessity. Most people access the Internet on a daily basis through their web browser. We use the web to look up information, send emails and interact on social media platforms. As the technology evolved, the websites turned into web applications, allowing us to do more complicated things like on-line banking, tax administration, looking up driving directions on a map of the world, follow on-line courses at a digital university, or work on a single document with multiple people in real-time.

Since the web has become a necessity, we expect it to be well designed, fast, error-free, and accessible to everyone, including people with a physical disability. In this thesis we will see that this is far from true. The majority of the web applications either contain errors, are not as fast as they could be or are inaccessible to some users.

One might wonder why it is so hard to develop these applications correctly. The answer lies in the fact that the web was originally not designed for these complex applications. It was meant for document sharing between researchers in 1989 [12]. One researcher would put his or her documents in a special format called HTML on a server for others to see. They could in turn use a browser to download and view those documents. Because the documents could link to each other, researchers could easily build a body of knowledge.

Six years later a programming language called JavaScript was introduced that could run in a browser and modify the document at the client-side. Five years after that JavaScript was able to load data from the server without reloading the document. This allowed developers to use the web to build full fledged applications, instead of a collection of simple pages connected via hyperlinks. Since then, many applications that used to be installed on hard disk (e.g., desktop applications), have moved to the web.

[1]Human Rights Council, Twentieth session, Agenda item 3
1. **Introduction**

Another development that happened nearly parallel to the shift from desktop applications to web applications was the adoption of smartphones. Smartphones allow people to browse the web as well. This means that web applications have to work on screens that range from the size of the palm of a hand, to high-resolution desktop screens. To appreciate the impact of this, consider how different an email client looks on a smartphone compared to the desktop version.

These developments have made the web significantly more complex than it originally was designed for. As a result, testing web applications has become harder too. Testing complex software can be time-consuming but is a necessary investment to keep the number of bugs and errors down. For that reason companies like Google and Microsoft spend more time testing software than writing new software [68]. The amount of time needed to adequately test software depends on the complexity of the software and the quality of the tools developers have available. However, since the web is relatively young, it lacks some of the more mature testing tools that older platforms have.

Because of the added complexity and the lack of good testing tools, we anticipated that the web contains a significant number of errors. We also anticipated that developers often forget to adhere to certain best practices.

In this thesis we analyzed 3,422 web applications for such errors and quantified the need for a tool that can help developers make web applications with less errors. We do so by using a JavaScript-enabled crawler to inspect as many states as possible of a web application. By inspecting all states, we can run tests and analysis on a full application.

Our main research question is therefore:

*Using a JavaScript-enabled crawler to dynamically test web applications, what are the most severe malpractices that occur in practice, and to what extent is a JavaScript-enabled browser needed to expose these malpractices?*

The research will be conducted based on the following sub questions:

**RQ 1** Are static analysis, dynamic analysis, or both using a conventional crawler, adequate tools to inspect modern web applications?

**RQ 2** How far is the adoption of best practices and new standards in current web applications?

**RQ 3** What are severe errors that occur most frequently in practice?

**RQ 4** What are the performance issues that occur most frequently in practice?

**RQ 5** What are the accessibility issues that occur most frequently in practice?

**Outline** We will first give some background about the technologies discussed in 2 and 3. We then explain our experimental setup in 4, 5 and 6. After that we introduce three metrics to measure the dynamism of the web in 7. In Chapter 8 we give an overview of the state of the web. We then discuss operational errors in 9, performance penalties in 10 and accessibility in 11. We conclude with a discussion in 12, put this thesis into context in 13 and finish with our concluding remarks in 14.
Chapter 2

Background

This thesis will discuss the web and the use of a crawler to analyze it. To get a better grasp of these technologies, this chapter will introduce them in more depth. We will look at the three basic components that make up the client-side of the web: HTML, CSS and JavaScript. After that we look at what software testing encompasses and we introduce the concept of a crawler.

2.1 HTML

The web was originally designed to share documents. These documents were to be written in a predefined language called the HyperText Mark-up Language, proposed in 1989 [12]. HTML is a markup language meaning that it has special syntax that allows a viewer of such a document to apply some style to it [54]. The viewer in the case of HTML is the browser. For example, in HTML a heading is marked by surrounding it with $H_x$ tags where $x$ is the order of the heading. To make a heading of order 1, which is the biggest, one would write $<h1>Hello world!</h1>$.

HTML consists of elements. The previous example of the heading is such an element. The opening and closing parts are called tags. Tags can also have attributes like an id. for example this heading 1 has id $foo$: $<h1 id='foo'>$. Some elements do not have an opening and closing tag but close on themselves. These are called $void$ tags. An example of a void tag is the image tag, which also always contains the source attribute, specifying where the file containing the image is located: $<img src='my_image.jpg' />$. Elements can contain other elements which are called their child elements. These elements are said to be nested in their parent elements, all together making a tree structure.

HTML documents have the ability to link to other documents using Uniform Resource Links (URLs). This was part of the initial design so that it would be easy for researchers to link to each others documents and thus create a body of knowledge. This is done by specifying the hyperlink reference attribute in an anchor element like this: $<a href='other_page.html'>Go to the other page</a>$.

A HTML document should alway start with the DOCTYPE declaration that tells the browser which version of HTML it is using. After that, the main HTML element opens which contains two elements: the head and the body element. The head contains
2. Background

Figure 2.1: Example HTML

Several versions of HTML have been published since the original proposal. The most recent one is HTML-5 [19]. There are two independent committees that decide what the HTML standard will look like: the World Wide Web consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). Unfortunately they do not exactly work together. W3C has decided that the HTML-5 version is finished and made its specification immutable. New contributions will go into the HTML-5.1 standard. WHATWG on the other hand decided that HTML-5 is what they call a Living standard meaning it can change over time. This is confusing for developers at best. As Andrew S Tanenbaum once said:

“The nice thing about standards is that you have so many to choose from. Furthermore, if you do not like any of them, you can just wait for next year’s model.” [66]
2.2 JavaScript

JavaScript is a programming language that was designed to run in the browser[8] and interact with the DOM. It can also interact with the browser in a limited way, like showing the user a dialog or redirecting the browser to another URL. An important feature that was added later on was the ability to fetch and send data from and to a server, and store data in the browser storage engine. Communicating data instead of pages with the server is done using a special XMLHttpRequest[1]. This technique quickly became mainstream in what was called Asynchronous JavaScript and XML (AJAX). Although the name indicates it communicates XML, it can actually read any web resource from a server. This technique was the prime mover of what was later called the Web 2.0.

At the time of writing JavaScript is the dominant client-side scripting language. JavaScript can either be embedded into HTML in a script element or by referring to an external file containing the JavaScript source code. For example, Figure 2.3 shows a snippet of JavaScript code can be inserted in the HTML and will print hello world to the log once the document is loaded:

```html
<script>
window.onload = function() {
    console.log("Hello world!");
};
</script>
```

Figure 2.3: Example JavaScript

2.3 CSS

HTML is a markup language. The style of that mark-up can be specified in a Cascading Style Sheet (CSS). For example, if we would like to set the color of the heading one element we showed in 2.1 and give it a margin of five pixels we can write a CSS rule as shown in Figure 2.4. A rule starts by specifying the elements it applies to, followed by the rules for that element surrounded by two brackets.

```css
h1 {
    color: red;
    margin: 5px;
}
```

Figure 2.4: Example CSS

CSS was introduced in 1996 and has since seen three major versions. Since CSS 3 the standard has been divided into modules, each with their own version status. The latest stable version is CSS 3, although at the time of writing some modules of CSS 4 are marked as stable too[2]. Again, this is confusing for developers to keep up with.

2. **BACKGROUND**

since the standards change every couple of months, and browsers do not adopt them at the same pace.

### 2.4 Web application testing

Now that we have seen the basic components of a web application we will discuss the most common techniques currently used to test these applications. In this thesis we will only discuss automated testing. Automated testing means that a program, not a human, inserts some input into a program, triggers the program, and verifies that the program has gone to the desired new state using an oracle.

Oracles are mechanisms that determine whether a test has passed or failed \[48, 67\]. They range from **strong or specific** oracles to **generic or weak** oracles. Strong oracles verify that the output of a test is exactly a predefined value. This is very precise but also very sensitive. Weaker oracles check if the result of a test is in a certain range that is allowed, for example that the result of a function is always a positive integer. Weaker oracles often have the advantage that they can be reused in other tests, which makes them highly suitable for automated testing, in which many different inputs are generated.

In this thesis we will further make a distinction between **domain-specific** oracles and **generic** oracles. A domain-specific oracle is specific to one application (the domain). For example, it could verify that every page on a web application shows the company logo in top-left corner. A generic oracle is an oracle that can test software independent of the software its goal. For example it could check that all images actually load without an error.

A subclass of oracles are **invariants**. A invariant simply means that a given condition will always be true, or always be false \[14\]. The example of every page having a image in the top-left corner classifies as an invariant.

Automated tests can be further divided into **dynamic analysis** and **static analysis**. We will discuss the differences between the two in more detail.

#### 2.4.1 Static analysis

Static analysis is the testing or analyzing of resources without the runtime environment, in this case the browser. An example of this is a syntax validator. A syntax validator like the W3C HTML validator\[3\] is an oracle that accepts a string as its input and returns whether or not that string was valid HTML. It can also tell the developer where the errors occur. The syntax checker does not need a browser to run. It just looks at the input and compares that to the HTML specification.

Static analysis is classically not considered a test but for the purposes of this study it is since we use the result of static analysis, such as the syntax checker, as an oracle.

#### 2.4.2 Dynamic analysis

Dynamic analysis is the testing and evaluation of an application during runtime. In our case the runtime environment consists of the browser. An example of this is a scripted

\[3\]http://validator.w3.org
test. In a scripted test a developer writes a script that specifies a number of steps the test program has to repeat like clicking on buttons or inserting text into the document. After (or during these steps) certain oracles can be run.

The advantage that dynamic analysis has over static analysis is that it can test states that are only available at runtime, and thus hidden from static analysis. Static analysis on the other hand is often faster and less complicated to set-up, because the application does not have to run and guided to a particular state to be able to test.

At the time of writing, most testing tools for the web use static analysis. In this thesis we will try to quantify the need for dynamic analysis to test web applications.
Web crawlers, also known as spiders or wanderers, are programs that are designed to explore the web. In this chapter we first discuss the first crawlers to understand why crawlers were invented and how they work. After that we will go into the more recent JavaScript-enabled crawlers which have been at the heart of this research.

3.1 Classic crawlers

To start a crawl a crawler is handed a seed URL. It fetches the HTML from the server by opening that URL. After that it analyzes the HTML for whatever purpose the crawler was designed. While reading the HTML, it saves all the href attributes it encounters in anchor tags and puts them on its TODO list if it has not seen the URL before. Once the page is fully analyzed, it grabs the next URL from its TODO list and opens that. Then the analysis recurses. The crawler will repeat this process until its TODO list is empty. This process is shown in Figure 3.1 in pseudo code.

```plaintext
urls_seen = 0
urls_todo = { seed_url }
while not empty ( URLs_todo ) {
    url = take and remove first of URLs_todo
    add URL to URLs_seen
    html = download html for URL
    if error or html not valid
        Go back to while
    analyze the HTML file
    for each link found in the HTML document {
        if URLs_seen does not contain link
            add link to URLs_todo
    }
}
```

Figure 3.1: Crawler pseudo code
Web crawlers are almost as old as the World Wide Web itself. The first crawler was implemented by Matthey Gray in the spring of 1993. It was called the Wanderer and its goal was to measure the size of the web [27]. Soon after that in 1994, the first crawlers that indexed the web appeared [53]. After the publication of Google’s seminal paper on crawlers [16] some open source crawlers began to appear as well [30, 15, 58]. While only the latter of these crawlers are built to index the World Wide Web, all of them have ways to measure the web, build graphs of it, and generate statistics. Since then crawlers have evolved and are now being applied for many purposes, like archiving the Internet [17]. In 2001, researchers started using crawlers to infer state models of web applications to be able to generate tests [2, 10, 11, 36, 57].

3.2 JavaScript-enabled crawlers

When a web application uses JavaScript to manipulate the DOM, analyzing a web application by retrieving the HTML does not suffice anymore. Once the HTML is loaded into the browser and JavaScript runs, it can manipulate the DOM to generate content that a crawler would not see by analyzing the HTML. To be able to see that generated content the crawler would have to run JavaScript as well.

It was not until 2008 that the first JavaScript-enabled Crawler was introduced [45]. These crawlers work by using a browser to interact with the JavaScript runtime environment. This makes such a crawler a dynamic analysis tool, as opposed to the crawlers discussed so far that classify as static analysis tools.

3.2.1 States instead of documents

A JavaScript-enabled crawler does not only follow href attributes derived from anchor elements, but can interact with every element in the DOM, just as a user might do. After such an interaction it compares the content of the DOM to see if it has reached a new state. This means that a single document can have more than one state.

By keeping track of these state changes a graph can be build of an application. Such a graph consists of nodes that represent the states, and edges that represent interactions that caused a state change.

3.3 Challenges for JavaScript-enabled crawlers

Because JavaScript-enabled crawlers work with dynamic pages instead of static resources they face a number of challenges. We will discuss the three main challenges below as they have all influenced the outcome of this research. Some of these challenges are not exclusive to JavaScript-enabled crawlers but also apply for conventional crawlers.

3.3.1 State explosion

One of the biggest challenges JavaScript crawlers face is the huge amount of states a web application can have. Even if one only counts DOM changes as state changes the number of states is too big or even infinite with modern web applications. We will discuss three of the most prominent causes of this problem.
3.4. State navigation

**Dynamic application state**  Even a small web application can have an infinite number of states. Consider a simple TODO list application. Every time the crawler enters text that is added as a new TODO, the DOM changes significantly. If the crawler regards this as a new state, it might enter another task, ad infinitum.

**Time-based content**  An example of time-based content is a mail application that shows the current date and how long ago an email was received like “2 minutes ago”. If the crawler scans that same page one minute later, the DOM has changed substantially. This means there is a new DOM state every minute, per view that has a time representation like this in it.

**User tailored content**  Modern web pages often offer user-tailored content. One example is having a web application in three languages. That means having roughly three times the number states. Another example of user tailored content is a bookstore that has a section on the page that displays previously purchased items. Because this is unique for every user it implies that there are as many states as there are users.

The main problem for crawlers is to detect when a DOM change is relevant enough to count as a new state. However, it is hard to define a generic compare algorithm that can discriminate the relevant states from the irrelevant states for any given web application. To this extend, JavaScript-enabled crawlers work best when they have some foreknowledge of the application they are about to crawl. This way, elements that do not count towards a state change, like the three types of content mentioned above, could be excluded from the compare algorithm.

3.4  State navigation

To be able to explore all states crawlers have to be able to re-visit states to navigate to other states that still have to be analyzed. Even though browsers have a forth and back functionality built-in, this is only tied to the application state if the developers chooses to. And even if they do, it is a cumbersome error- prone task. This is why web applications often have a different state model than the one that can be derived from the URLs, making the navigation hard to automate [50]. This means that crawlers cannot expect to go to the previous state when they press the back button. They need a more robust system of navigating through the application.

3.5  Invoking state changes

State changes can be caused by many kinds of events in a web page [42]. Clickable’s are not limited to `<a href='x'>` elements. JavaScript allows one to add a click handler to practically any HTML element. Clicks are not the only events causing a state change. For example: there are other mouse events like hover, mouse-in, mouse-out, drag and drop, double click and right click, there are touch and gesture events [1] for

---

1[http://www.w3.org/wiki/Touch_Events](http://www.w3.org/wiki/Touch_Events)
3. CRAWLERS

tables and smartphones, there are events for elements being selected or unselected, there is an event for URL hash change, keyboard keys pressed, and many more.²

To reach all possible states, the crawler could invoke all possible events on all possible elements. But even if it would do that, the combination of those elements might be the key to going to the next state. For example, some applications have special states for when a user holds a keyboard key and then click an element. Trying out all possible combinations quickly leads to a combinatorial explosion and is therefore an inadequate solution.

The challenge for crawlers is to be either so fast and strong that they can try many of these combinations, or to be smart and discover which elements are listened to by JavaScript. Although finding which elements in JavaScript have listeners is possible, this does not cover the case of input combinations.

3.6 Unreachable states

Unreachable states are also known as “The Deep Web”. This term comes from conventional crawlers and stands for the part of the web that cannot be found by following links.¹ ³ Although JavaScript-enabled crawlers can find more than conventional crawlers, they face some of the same barriers.

So far we have discussed state changes that are discoverable by interacting with elements. There is however another type of input that often requires domain specific knowledge to trigger a state change: text input. This type of input often occurs in one of two themes: security, search and forms.

Security The simplest example is that of a username and password box. Inserting random characters into these boxes is unlikely to lead to a login, thus inhibiting the crawler from going any further. Security might also be in form of a user only being allowed to follow a certain path once, like registering for something. This prevents the crawler from going back and retrying it with different options potentially leading to new states. Finally there is the possibility that the crawler has no access at all to certain states for security reasons.

Searchable states Web pages that are the front-end of any kind of large database often have states that are only reachable by search. For example: on Wikipedia not all pages are connected.¹ Only by searching certain articles can be found. If the goal of the crawler is to check every article for broken links, the crawler needs to have a list of all strongly connected graphs as an entry point, so it can click from there to all other pages.

Forms When a web page makes use of a form, it is unlikely a crawler can guess all input combinations that will lead to all possible resulting states. When the crawlers wants to find every possible state it should not only know a correct state, it should also be able to trigger any state that for example gives a certain type of error or helper message.

²http://www.w3schools.com/jsref/ DOM_obj_event.asp
³http://mu.netso.ie/wiki/
3.6.1 Non connected states

The final category of unreachable states are states that are not connected in any way to another state. This is often a form of security known as security through obscurity. It can also be part of the design because that part of the application is conceptually different from the rest of the application. The only way a crawler can discover a page like this is if the crawler has a priori knowledge of the page. To be able to reach those pages the crawler needs an entry point into that part of the application [38, 61].

3.7 Crawljax

In this thesis, we will use a JavaScript-enabled crawler named Crawljax\(^4\) and its plugins. It was first introduced in 2008 [42] and is still under active development. To the best of our knowledge, there is no other open source crawler that is capable of crawling JavaScript web pages as thoroughly as Crawljax can.

3.7.1 Invoking state changes

Crawljax searches the DOM for a set of candidate elements that might invoke a state change by firing a click event. The predefined candidate elements are `<a>`, `<button>`, and `<submit>`, but this is configurable. Crawljax can also be configured to include or exclude elements using XPATH\(^5\) queries, allowing a more fine-grained configuration. At the time of writing, Crawljax only offers click events and no other interaction with elements like mentioned in [3.5]. Crawljax also offers the possibility to specify input data that the crawler will fill into input fields. If configured correctly it can overcome some of the challenges of finding hidden states as mentioned in [3.6]

3.7.2 Invoking state changes

Crawljax determines if a DOM change is significant by normalizing the DOM and then doing a string comparison between the string representations of both DOMs. Crawljax also offers the possibility to configure custom comparators [61], allowing users to configure Crawljax to be more, or less sensitive about certain DOM changes. If configured correctly this can limit or even eliminate the state explosion as mentioned in [3.3.1]. For example, if there is a DIV element that contains non-relevant data, that DIV can be excluded from the DOM comparison.

We can conclude that it is hard or even impossible for even a JavaScript-enabled crawler to find all possible states of an application if it has any of the hidden elements discussed in this section. Crawljax has the ability to access these states but only if the user has foreknowledge of the web application. Because we selected random web applications for this study, we do not have such knowledge. However, even by analyzing what we can find, we can already draw some interesting conclusions, as we will see in the following chapters.

\(^4\)http://Crawljax.com
\(^5\)http://www.w3.org/TR/xpath/
Chapter 4

Experimental set-up

Now that we have an understanding of the possibilities and limitations of JavaScript-enabled crawlers, we will discuss how we have used them in our empirical study. First we present our conceptual set-up. After that we share the most important implementation details.

4.1 Conceptual set-up

To test our hypothesis, we run the JavaScript-enabled crawler on web applications randomly selected from the Internet using a random URL generator[1]. The crawler clicks on predefined elements in a page and then checks the DOM for changes. If the DOM has been changed in any way it regards the change as a new state.

We equip the crawler with the following functionality:

- By saving the transitions between states a state-flow graph is constructed for the given web application. The nodes of this graph represent states and the edges represent an event on an element that caused the transition to another state.
- We capture all headers sent and received by the browser during a crawl using a proxy server.
- We capture a snapshot of the DOM from the browser and save it.
- The DOM converted to HTML and checked for validity.
- The state-flow graph of the application is saved for further analysis.

We measure the dynamism of a web application in three ways:

1. by comparing the amount of URLs seen versus the amount of states seen. This ratio shows an estimation of how much states there is behind a static URL,

2. by looking in the state-flow graph which state transition where clicks on anchor tags that have a href element. This shows us how much more a dynamic crawler finds versus a static (classic) crawler,

3. by comparing the HTML received from the server to the HTML seen in the browser after waiting for five seconds after the page is loaded. This gives JavaScript the time to do most if its DOM manipulation that are triggered when the application is loaded.

4.2 Implementation

The set up is implemented using several pieces of technology. We explain their configuration briefly in this chapter. Two of the products used need some further explanation which will be done in the two consecutive chapters.

4.2.1 HTML Validator

The HTML validator used is the official W3C HTML validator with the HTML-5 extension enabled.

4.2.2 Measuring the dynamism

We created a custom tool to measure the difference between the DOM received from the server and the DOM seen in the browser. The works by performing a HTTP GET request on the URL. It simultaneously directs a Firefox browser to that same URL. After both the request and Firefox have opened the URL it compares the received DOM and the DOM captured in Firefox. This tool is open source and available on Github.

4.2.3 Browsermob proxy

A modified version of the "BrowserMob" Proxy server was used as the proxy server. Modification were required because the original proxy server depended on old third-party libraries that conflicted with the crawler we use. In our modified version, we have brought the entire project up-to-date, fixed four bugs and cleaned up the code a little.

4.2.4 Operation Big Crawl

The crawlers ran on different machines and sent their results to one receiving server. That server contains a SQL database with all the results. It is this database that we would like to share with the research community. We will discuss the architecture of this application further in Chapter.

---

2http://validator.w3.org
3http://about.validator.nu/ Version 2.0
4version 22 and later on version 23
5http://github.io/alexnederlof/rendered-web/
6https://github.com/alexnederlof/browsermob-proxy
4.2.5 Crawljax configuration

During our study we have used Crawljax as the JavaScript-enabled crawler. Crawljax is configured to give an extensive insight into a web application yet only take a sample if the web application was too large. More specifically:

- It is configured to use a string-based comparison of the DOM between potential states. This is the most sensitive comparison available offering the best coverage of an application.

- To speed up the crawls, Crawljax is configured to click every interaction element only once. This means that if the resulting execution path of a certain interaction element in the exact same DOM state behaves differently based on a certain state (e.g., Time, JavaScript memory variable) it will only check one of those execution paths.

- Only HTML tags of type `<button>`, `<a>` and `<input>` are clicked upon. Input fields are filled with random values.

- Finally, we limit the maximum runtime for each crawl to two hours. This is a basic safeguard against crawl traps or gigantic applications like Wikipedia or Facebook.

Before this study could commence Crawljax had to be improved in several ways. In the next chapter, we will discuss the contributions made to Crawljax for it to be able to generate the result set presented in this thesis.

---

7version 3.2 and later on version 3.3
Chapter 5

Scaling up Crawljax

At the start of this thesis Crawljax had not been touched in almost a year. We encountered out-dated code, non-resolvable dependencies and build problems. Besides that, this research required some extra functionality and better performance. This chapter will describe the modifications made to prepare Crawljax for our research and future development.

5.1 Project organization

Since the inception of Crawljax in 2008 [42] there have been many developments in the Open Source community with regard to project management. Decentralized Version Control Systems (DVCS) are one of those developments. Another is the use of a publicly available workflow that tests and asserts the quality of the code every time a change is made. In this section, we discuss how we have adopted these novel concepts into the Crawljax workflow.

5.1.1 Decentralized version control

At the start of this research there were multiple researchers working on the project. However, code was not shared between them. This was mainly because there was no culture of sharing, and because sharing code seemed hard. Nevertheless for a research tool like Crawljax it is vital to have an active community of researchers and developers that regularly contribute to the project. Especially with the speed at which the web is evolving it is important for Crawljax to leverage the community to keep up. To achieve this goal, some project organization modernization was required.

To make the process of working with multiple researchers that are geographically distributed on the same code possible, we started using the DVCS Git and the Git branching model[1]. The source code is now hosted on GitHub[2] allowing everyone from that community to participate in the development. This has been a substantial success. There are now 57 forks of the Crawljax source and the project went from 7 contributors to 22.

5. **Scaling up Crawljax**

5.1.2 **Streamlined workflow**

Not only did the project move to Git, we also re-enabled the continuous integration service Jenkins\(^3\) set-up SonarQube\(^4\) for continuous quality monitoring and created a link between Jenkins and GitHub. When a developer pushes a change proposal to GitHub, it is now built, tested, its quality is assessed and the result of the tests are visible in GitHub. Another developer can then review that change, see that the build has passed and merge the branch into the production branch.

5.2 **New features**

With the new workflow in place, we could start to address the new features required for our research.

5.2.1 **Multiple distributions**

Crawljax used to ship as one executable that contained everything. This means that Crawljax users cannot choose to import only the parts of Crawljax that they need. Another disadvantage is that having all the source code in one project makes it harder for new developers to find what they need.

By splitting up Crawljax into multiple modules, we solved these problems. Crawljax is now available as different distributions. The **core** module is a jar that can be used to import Crawljax into other projects and control it programatically. The **cli** module allows developers to control Crawljax directly from the command line. The **example** module contains examples to show new users how to use Crawljax. Finally there is also a **test utility** module, which can be used by plug-in developers, to test basic functionality.

5.2.2 **The Crawl overview plug-in**

To give Crawljax users a better feeling of the crawl performed, Crawljax originally shipped with a **crawl overview** plug-in written by Danny Roest as part of his MSc thesis \(^61\). This plug-in offers a visualization of the state-flow graph where the nodes represent states and the edges represent interactions between those states. A screenshot of the state allowed to user to see which state it was. In Figure 5.1 a screenshot is shown of the old version after some changes to make the code compile and work again.

Unfortunately the plug-in had not been maintained and was broken and out-dated. To give developers a better feeling of what happened during a crawl we first made the plug-in work again. We then added new functionality. The graph is now interactive, meaning that developers can zoom in and out, drag the states around and get more information about a state or edge by clicking on it. A screenshot of this new graph is shown in Figure 5.2. Every state now also shows a snapshot of the DOM, some statistics, and a list of its interaction elements.

The are also some overall statistics available about the crawl, as can be seen in the screenshot in Figure 5.3.

\(^3\)https://jenkins-ci.org

\(^4\)http://www.sonarsource.com/
5.2. New features

Figure 5.1: The old overview plug-in

Figure 5.2: The new overview plug-in graph
Not only does the new plug-in report everything in the web interface, it also writes the result data in the JSON format to disk, to make it easier for researchers to do data mining on a set of results.

![Crawl results](image)

Figure 5.3: The new overview plug-in statistics

### 5.2.3 Implemented Statistics

While the crawl overview gives insight into the result of a crawl it does not instrument the progress of a crawl. Knowledge about its progress can be used to measure Crawljax performance and discover oddities during a crawl.

To gain this knowledge we equipped Crawljax with a statistics library[^6]. It can be used anywhere in Crawljax to register all kinds of events, measure times, distributions, etc.. Crawljax is by default configured to track the number of errors that occur, the number of times it gets lost and the running times of certain methods.

These observations can then be printed to the console, retrieved after the crawl is done, or observed from a statistics server. The number of times the crawler is lost for example, is an important metric for this study since it tells us how well Crawljax was able to cope with the navigation problem, explained in §3.4. We found out that Crawljax is often lost, due to the sensitivity of the way it resolves elements to click on. This is discussed further in §12.3.

[^6]: [metrics.codahale.com](http://metrics.codahale.com)
5.2.4 Crawling hidden anchors

The main idea of Crawljax is that it is able to click an element and see if it leads to a new state. However, when such an element is visually hidden Crawljax cannot click on the element. This is a known limitation of WebDriver\(^7\), the technology that Crawljax uses to interact with the browser Crawljax. For some web applications this is a significant problem because most of the navigation happens via menus that fold out as seen in Figure 5.4.

![Figure 5.4: A menu that folds out when the mouse hovers over it](image)

To circumvent this problem we created an optional configuration for Crawljax that tries to open these elements anyway. This is done by inspecting whether the element is an anchor element with a valid `href` attribute. If it is, we direct the browser to the URL specified in the `href`. There might be more robust ways of solving this problem which will be discussed in 12.2.

5.2.5 Excluding a tree of elements

Another problem we found concerning the clicking of elements is the exclusion of certain elements. Crawljax lets a user decide which elements should or should not be clicked. However it is not possible to specify that all _children_ of a certain element are not to be clicked.

This feature was a requirement for this research to be able to exclude sub trees of a DOM from analysis. For example, it is not relevant for the crawl to click on every element found in a Facebook plug-in on a web application, or on the links in an embedded Twitter stream. However, the user does not always know beforehand how those plug-ins load their interaction elements. So to exclude them from the crawl, we implemented a configuration that lets the user specify which elements in the DOM are containers of such plug-ins, and should be ignored all together.

\(^7\)http://docs.seleniumhq.org/projects/webdriver/
5. SCALING UP CRAWLJAX

5.2.6 Batch processing via CLI interface

To be able to crawl as many applications as we intended to do for this thesis, we needed a way to automate the start of a crawl using a script. To do this we chose to expand the command-line interface that came with Crawljax already.

The old command-line interface did not offer the flexibility required for this research. The new version has the ability to set several new options, including the browser type, maximum runtime, maximum states and number of browsers.

5.3 Performance

Crawljax was initially designed to be a single-threaded process. At a later stage, an attempt was made to make it work concurrently by Stefan Lenselink in light of his master thesis [46]. However, the concurrent implementation of that time did not adhere to best practices and contained many race conditions [26]. As a consequence Crawljax would often deadlock and did not efficiently make use of the resources available. To fix these problems, we redesigned a new concurrency model.

The new algorithm is based on the original Crawler algorithm presented in Figure 3.1. The difference is that the original algorithm works on a per-document basis and this algorithm works on a per-state basis. The new algorithm is described in Figure 5.5. In the new algorithm, every browser acts as a crawler process. This means there are as many crawlers as there are browsers. The more browsers a user chooses to use, the faster the crawl can go up to the point where either the bandwidth limit is reached, or the server cannot produce new resources any faster, or the client cannot run anymore browsers in parallel due to hardware restrictions.

```java
1 candidate_elements_seen = Ø
2 candidate_elements_todo = { seed_state }
3
4 while not empty ( candidate_elements_todo ) {
5    candidate_element = take and remove first of
6        candidate_elements_todo
7    add candidate_element to candidate_element_seen
8    path_to_state = find interactions to reach state
9    for each interaction in path_to_state
10       fire interaction
11
12   if state unreachable or element not found
13      Go back to while
14
15   scan the new state
16
17   for candidate_element found in the new state { 
18      if candidate_elements_seen not contains candidate_element
19        add candidate_element to candidate_elements_todo
20    }
21 }
```

Figure 5.5: Crawljax concurrent crawling pseudo code
Because the candidate_elements_todo queue is shared between crawlers the browsers are used as much as possible. Browsers are the most expensive resource during a crawl and thus the bottleneck. By making sure they are never idle we gained a significant performance boost.

The path_to_state is calculated using Dijkstra’s shortest path algorithm on the state flow graph [22]. The path is implemented as a list of interaction elements that have to be triggered to reach a certain state. The elements are fired until the given state is reached. If the state cannot be reached the crawler registers that it got lost and goes on to the next item in the TODO list.

5.4 Overall statistics of performed work

These contributions had a notable impact on the source code of Crawljax. In this section, we will report the result of these changes from a runtime perspective in terms of performance, and from a code quality perspective.

5.4.1 Performance

Thanks to the new concurrency model explained in 5.3 and the ability to find more elements explained in 5.2.4 we observed a significant increase in Crawljax its speed and coverage.

The overall speed of the crawl increased by 150%. This can be attributed to the new concurrency model which makes more effective use of the slowest resource: the browser. The amount of crawls that stopped because of a fatal error in Crawljax also went down from 33% to 0.6%. This is because the new concurrency model is much more resilient against known problems with the browser.

The coverage increased by 190%. Coverage in this case is the number of states found on a given web application, compared to the older version of Crawljax. This number is primarily influenced by the fact that Crawljax can now click most hidden elements, does not deadlock as often and is faster. Because it is faster, it will find more states when the crawler is only allowed to run for a maximum amount of time.

Finally the new version also has a much higher coverage of interaction elements. This is thanks to a fix in the core for a bug that we encountered during the refactoring causing edges to be lost before they could be saved. Now that all edges are saved to the graph, we find up to three times as many edges. This gives as a more accurate insight into the structure of the application.

5.4.2 Code statistics

While refactoring Crawljax we applied Robert Martin’s boy-scout rule, described by Martin Fowler in his post on opportunistic refactoring [40]. This means that any code that can be improved, will be improved right away. In this section, we will discuss some of the most notable statistics that resulted from this process of continuous improvement.

---

8The test were performed on a 2.8 GHz Intel Core i7, 8GB DDR3 RAM computer. The old and new version of the application were run on 100 applications directly after each other to get the best comparison.
5. SCALING UP CRAWLJAX

During this thesis 72 pull requests were accepted into the main branch during a period of six months. Nearly every month, we have released a new version of Crawljax so other researchers can take advantage of those new features and bug fixes.

During development, the quality of Crawljax was monitored by SonarQube. SonarQube keeps track of the number of coding standard violations in the project. During the this thesis, we solved 62% of these violations, while 261 issues still remain in the code.

Besides SonarQube the code was also assessed by the Software Improvement Group (SIG)\footnote{http://sig.eu}. The SIG measures code quality on a scale of 0.5 to 5.5 where 1 is the lowest quality when compared to other projects the SIG monitors. The score is divided into seven criteria. Figure 5.1 shows the result of our contributions to the Crawljax source code on each of these criteria. Overall we achieved a maintainability improvement of 0.34 points according to the SIG system, bringing the score from 3 to 4 stars out of 5, moving Crawljax from the top 65% in maintainability to the top 35%.

5.4.3 Community growth

As we mentioned earlier the number of contributors has increased from 7 tot 22. But the project is not only gaining traction amongst developers. We have seen a growth in downloads as well. Figure 5.6 shows the number of downloads per month from unique IP addresses since the start of this research. We can see the number grow from 9 downloads to 158 downloads in July.

![Figure 5.6: Downloads of Crawljax per month from unique IP addresses.](http://sig.eu)
5.4. Overall statistics of performed work

Table 5.1: Progress in code maintainability according to SIG between September 2012 and 2013

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Difference</th>
<th>Current value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume</td>
<td>-0.06</td>
<td>5.31</td>
</tr>
<tr>
<td>Duplication</td>
<td>-0.05</td>
<td>5.31</td>
</tr>
<tr>
<td>Unit size</td>
<td>+ 0.25</td>
<td>3.90</td>
</tr>
<tr>
<td>Unit complexity</td>
<td>+ 0.66</td>
<td>3.94</td>
</tr>
<tr>
<td>Unit interfacing</td>
<td>-0.06</td>
<td>3.65</td>
</tr>
<tr>
<td>Module coupling</td>
<td>+ 0.52</td>
<td>2.68</td>
</tr>
<tr>
<td>Component balance</td>
<td>+ 1.75</td>
<td>2.25</td>
</tr>
<tr>
<td>Overall Maintainability</td>
<td>+0.34</td>
<td>3.68</td>
</tr>
</tbody>
</table>

Points are measured on a scale of 0.5 to 5.5.
Chapter 6

Project big crawl

Now that we have seen how the crawler works, we will discuss how we put it into play. We set out to crawl as many applications as possible. However, crawling is a very resource-intensive process. On average it takes one computer one hour to perform a crawl, and it cannot run multiple crawls simultaneously.

To overcome this problem, we came up with a solution to automatically distribute the crawling process over multiple machines. This chapter will explain how we achieved that goal.

6.1 Distribution of crawls

Thanks to the new command line interface introduced in Chapter 5.2.6 we were able to start and control Crawljax from a script. We installed the script on 12 different computers. The script automatically downloads the latest version of Crawljax, checks if the environment is set-up correctly and then runs custom wrapper for Crawljax.

That wrapper communicates with the central Big Crawl server that distributes the URLs that have to be crawled. After it has received the URL the wrapper starts the crawl. Once the process complete, all the results are uploaded to the central server. If the process completes, fails or times out, the script starts over. This relation between the server, crawlers and database is shown in Figure 6.1.

The central Big Crawl server is a Java server that uses Jersey to create its REST interface. On top of the REST interface lies a web interface implemented in AngularJS. The graphs in the interface are shown using the D3 visualization library. The web interface shows the server status as can be seen in Figure 6.2, the status of the crawlers and intermediate statistics about the crawls as can be seen in Figure 6.3. The interface also lets a developer prioritize the next URLs to be crawled by the crawl clients. All data is stored in a PostgreSQL database.

1http://jersey.java.net
2http://angularjs.org
3http://d3js.org
4http://PostgreSQL.org
6. PROJECT BIG CRAWL

Figure 6.1: Big Crawl global architecture

Summary

<table>
<thead>
<tr>
<th>Global</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total crawls</td>
<td>3,068</td>
</tr>
<tr>
<td>Unique URL’s crawled</td>
<td>3,057</td>
</tr>
<tr>
<td>Total URLs seen</td>
<td>894,814</td>
</tr>
<tr>
<td>Total States seen</td>
<td>2,137,817</td>
</tr>
<tr>
<td>Average states per URL</td>
<td>18.67</td>
</tr>
<tr>
<td>Irrelevant URLs declined</td>
<td>1002</td>
</tr>
</tbody>
</table>

W3C Errors

<table>
<thead>
<tr>
<th>As reported by the W3C validator per site.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique errors</td>
<td>0.621</td>
</tr>
<tr>
<td>Total errors</td>
<td>34.387</td>
</tr>
<tr>
<td>Unique warnings</td>
<td>0.014</td>
</tr>
<tr>
<td>Total warnings</td>
<td>12,317</td>
</tr>
</tbody>
</table>

Exit statuses

<table>
<thead>
<tr>
<th>Exit status</th>
<th>Crawls</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXHAUSTED</td>
<td>1126</td>
</tr>
<tr>
<td>MAX_TIME</td>
<td>1942</td>
</tr>
</tbody>
</table>

Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>Crawls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0-1-g36a249c</td>
<td>17</td>
</tr>
<tr>
<td>1.0-133-g01f16d</td>
<td>86</td>
</tr>
<tr>
<td>1.0-134-g33afl14</td>
<td>281</td>
</tr>
<tr>
<td>1.0-2-g81a783</td>
<td>2684</td>
</tr>
</tbody>
</table>

State Visibility

| Overall visible states | 83.51 % |
| Visible states per site | 84.69 % |
| Min visible states on site | 0.05 % |
| Max visible states on site | 100.00 % |

Crawlers

<table>
<thead>
<tr>
<th>Crawler</th>
<th>Crawls</th>
</tr>
</thead>
<tbody>
<tr>
<td>disbb</td>
<td>29</td>
</tr>
<tr>
<td>dutfe</td>
<td>551</td>
</tr>
<tr>
<td>dutfi</td>
<td>631</td>
</tr>
<tr>
<td>dutki</td>
<td>419</td>
</tr>
</tbody>
</table>

Indexing

<table>
<thead>
<tr>
<th>Last seen</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header loading</td>
<td>100 %</td>
</tr>
<tr>
<td>State analysis queue</td>
<td>0</td>
</tr>
<tr>
<td>Visibility queue</td>
<td>126</td>
</tr>
<tr>
<td>Resource loading queue</td>
<td>0</td>
</tr>
</tbody>
</table>

Users

<table>
<thead>
<tr>
<th>Last seen</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawler</td>
<td></td>
</tr>
<tr>
<td>lbofneac.ubc.ca</td>
<td>2 months ago</td>
</tr>
<tr>
<td>Lextop.local</td>
<td>2 months ago</td>
</tr>
<tr>
<td>Big-Apple.local</td>
<td>2 months ago</td>
</tr>
<tr>
<td>arle</td>
<td>20 days ago</td>
</tr>
<tr>
<td>mnbay-inact1.eva.ubc.ca</td>
<td>12 days ago</td>
</tr>
<tr>
<td>bas</td>
<td>2 months ago</td>
</tr>
<tr>
<td>dutihi</td>
<td>an hour ago</td>
</tr>
<tr>
<td>dutisp</td>
<td>43 minutes ago</td>
</tr>
<tr>
<td>dutisj</td>
<td>27 minutes ago</td>
</tr>
<tr>
<td>all</td>
<td>a month ago</td>
</tr>
</tbody>
</table>
6.2 Collection of statistics

All results the server receives are immediately saved to the database to reduce the risk of data loss. Once the data is saved a pointer to the new data is put on a worker queue. This process is illustrated in Figure 6.4.

There are several statistics that are run on the incoming data: the headers are analyzed, the states are analyzed and the structure of the applications is analyzed. Each one of these tasks has its own worker queue and a limited number of workers. The amount of data that has to be analyzed far outnumbers the amount of memory the server has available. Even though the server we used had 12 gigabyte of RAM available, parsing four web applications simultaneously with 2000 states of 3 megabyte each is too much. To prevent out-of-memory exceptions the workers are only scheduled when enough resources are available to run the analysis. Because the results are received in unpredictable bursts, this is determined dynamically by inspecting the current memory usage before dispatching a worker.

After the results are analyzed the aggregated statistics are saved back into the database to make the calculation of statistics faster.

---

Figure 6.3: Big Crawl realtime headers statistics view
6. **Project Big Crawl**

6.3 **Development workflow**

Because of the many technologies used on top of each other and the complexity of the project, having adequate tests was quintessential for this research to succeed. Especially because we wanted to be able to continuously deploy improvements into the production. To achieve this we created a workflow that automatically tests and deploys these new versions.

When a developer wants to deploy his new version he runs a deploy script. That script will first start a virtual server that contains a PostgreSQL database with an identical configuration to the one used on the server that runs the Big Crawl service. This is done using the provisioning tools Vagrant\(^5\) and Chef\(^6\). After that it starts the Big Crawl server. Once the server has started without any errors the wrapper is started that crawls a predefined URL. When it is done it uploads the results to the server. The server in turn calculates all the statistics. After the calculations are done the test asserts that all data is correct using the server’s REST interface. The result of this set-up is that the entire technology stack required for crawling is tested.

Once the tests have succeeded and the developer feels confident about his new version he can browse to the distribution folder inside the project. That folder contains the latest assembly of the product. The folder is also a separate Git repository. If the

---

\(^5\)http://vagrantup.com
\(^6\)docs.opscode.com/chef_solo.html
developer commits the changes in this repository and pushes them to the central Git server all crawl clients will use that new version in the next crawl. This is because the client installations are in fact clones of that Git repository and they update before each crawl. This way, the distribution of new versions amongst clients is fully automated.

The server can be re-deployed using a script found in the script folder, which stops the old version of the service, deploys the new version, starts the new version and opens a console that shows the log to verify everything went well.

### 6.4 The resulting data set

Our empirical study resulted in a dataset with 3,422 applications that has over 96,144,592 HTTP headers and 2,565,981 DOMS. The applications include well known applications like Google.com, Facebook.com and Twitter.com. The applications span 101 different top-level domains, range from 1 to 26921 states, and include up to 2,253,278 errors according W3C.

The dataset also contains all errors and warnings per state reported by the W3C HTML validator. Finally the dataset contains a graph that contains the state flow graph of every web application describing the resulting state of every interaction element.

The total processing time to collect this data using about eight machines to crawl is an estimated 51 days. The server received approximately 61 new web applications per day to analyze. That is about 20 headers per second and one DOM every two seconds. The resulting database is 216 Gigabyte large.
Chapter 7

Measuring Client-side DOM manipulation

So far we have stated our hypothesis and proposed our experimental setup. In these five consecutive chapters we will discuss the results of our study.

This chapter will discuss the amount of DOM manipulation that happens on the client-side. The amount of manipulation is used as a guideline in the consecutive chapters.

Our hypothesis is that a lot of modern web applications use client-side DOM manipulations, instead of static document loading. As a result, using static analysis to test and analyze a web application does not suffice. To measure how much static analysis tools miss we propose three metrics that look at dynamism on the client-side:

1. The number of states per URL,
2. The number of states without a URL,
3. The number of post-load DOM manipulations.

We will discuss the results of each of those in this section.

7.1 States per URL

The first metric compares the number of states found compared to the number of URLs seen. In Figure 7.1 we see the distribution of states per URL, leaving some of the extreme cases out. There are no web applications underneath the diagonal because it is impossible to have fewer states than URLs using the DOM comparison we used. We can see that most applications diverge from a one-to-one ratio between URLs and state.

Our study shows that per URL there are 1.9 states. This means that if a static analysis tool would request a certain URL once, without any context, it would miss 51.8% of the states.

This number might be high because of the way we performed our state comparison. Consider an application that displays the current time on the page. That application
7. MEASURING CLIENT-SIDE DOM MANIPULATION

Figure 7.1: States per URL

would result in a different state every time the crawler opens the URL. Another example is an application displaying adds provided by an add-service. Each time the application is reloaded, there is a high chance it will show different adds, resulting in a different DOM. As we discussed in section 3.3.1 it is hard to distinguish such irrelevant states from the relevant states.

On the other hand, the number of URLs for this metric includes duplicate URLs with a different fragment identifier “#”. This means that when the metric is used without a browser (e.g., static analysis), the analysis would test even fewer states.

We conclude that there are many states per URL. This means that if a scripted browser test works on a URL basis, it would only test 51.8% of the available states.

7.2 States without a URL

The second metric addresses the number of states that are only reachable by interacting with the browser. This can either be because the web application required input and we inserted random text, or because the state transition was not caused by an anchor tag with a href attribute. By looking at the edges in the state-flow graph we did a breadth-first search of states triggered by an anchor tag with a href containing an URL (excluding the fragment identifier #). We marked those states as visible. All other states are marked as invisible. By comparing the number of visible versus invisible states we find the number of states hidden from a traditional crawler.

There are two cases we did not take into account: if an anchor tag is inserted by JavaScript on the client-side a traditional crawler would not have found it, since such a crawler does not run JavaScript. The other case is that during this research we have
only looked at anchor and input tags. If click-handlers would have been attached to other elements we have not triggered their actions.

We found that on average 85% of the graph is reachable through hyperlinks. The distribution has the first quantile at 80%, the median at 96% and the third quantile at 100%. Only 32% of the applications contained no invisible states.

We can conclude that the public applications we have investigated are indeed optimized for crawling since we can reach almost all states. However, the statistics we present are the under bound of the actual invisible states, due to the two cases described above. This means that although the applications appear to be optimized for traditional crawlers, those crawlers will still miss states on at least two out of three applications.

7.3 Post-load DOM comparison

The last metric we used is a measurement of the difference between HTML retrieved from the server and HTML observed in the browser after five seconds. We looked at the textual content in the HTML elements and the number of elements changed. This means that if an element is added or removed, this counts as a change. If any attributes are added or modified, this counts as a change too. If text is removed or added it counts as one point. If text is modified it is seen as an add and a remove operations and therefore counts as two. Table 7.1 shows the result of this metric on our dataset.

Table 7.1: DOM Manipulations after browser load

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>q₁</th>
<th>Median</th>
<th>q₃</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>23</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>3924</td>
</tr>
<tr>
<td>Elements</td>
<td>298</td>
<td>60</td>
<td>164</td>
<td>354</td>
<td>5745</td>
</tr>
</tbody>
</table>

Note: Results are rounded to the nearest integer

Another interesting observation is that 64% of the applications crawled modify the DOM after it has been loaded if we look at the textual content of the DOM. If we look at the number of elements changed, it is even more: using a threshold of 25 elements to make the analysis less sensitive we find that 89% of the applications are manipulating elements in the DOM. Setting the threshold to 100 elements shows us that 65% of the applications manipulate the DOM after the browser loads.

These manipulations mainly come from two sources: first and foremost is the fact that Firefox normalizes the HTML after it receives it. It does a best effort to repair any broken HTML by inserting missing tags and adding missing elements. However, the browser will never change the content of a DOM. The second source for changes is actual JavaScript. This can be either written by the developer or an included third party library. This is the only source for changing any of the content in the DOM. This method primarily shows us how much static analysis tools would miss if they just ran on the static HTML.

We conclude that at least 90% of the web application we investigated perform at least some client-side DOM manipulation after they have loaded. This means that even if a traditional crawler would find all states in an application, or there would exists a
test that covers all URLs, the test would still need to run in a browser to see all the possible states that occur.
Chapter 8

The state of the web

Before we continue with the errors and lack of adherence to best practices, this chapter will give some insight into the state of the web. There are many observations that can be made using our dataset. We have chosen the three that surprised us most: the usage of HTML, compression, and mobile web applications.

In the following chapters we will see many numbers concerning errors and lack of adherence to best practices. To put these numbers into context, we will also report on the volume of the web in this chapter.

8.1 HTML-5

The first thing we noticed is the wide adoption of the HTML-5 doctype.

HTML-5 was released in 2012 and is at the time of writing the newest HTML version. Its most notable improvements are new features like embedding media into an application, new elements or attributes that save developers from writing boilerplate code, and elements or attributes that make the web more semantic so it is better readable by computer programs. Making a web application easier to read by computer programs is important because those programs can help people with a disability to use the web and it helps programmers to automate certain tasks.

We found that 52% of the applications declare the HTML-5 doctype on their index page. Table 8.1 shows the percentages of applications using any of the new HTML-5 features, ordered by adoption. We measure the elements with an occurrence of at least one, two or three times. That way we try to see if an element was used only once, or whether it has been incorporated throughout the application. The last row in Table 8.1 shows us how often any of the HTML-5 features is used in an application. Even if we set the threshold to at least three elements 43% of the web applications have adopted the new HTML-5 features.

The features we discuss only concern the features that can be observed from analyzing the DOM [4,19]. We will discuss the top five of these features below.

Asynchronous JavaScript On top of the list is the asynchronous loading of JavaScript. This allows developers to import JavaScript in the correct place of in the HTML without blocking the page load. We will discuss this feature in more detail in Section 10.5 when we discuss performance in full detail.
8. The state of the web

Table 8.1: HTML-5 feature adoption

<table>
<thead>
<tr>
<th>Feature</th>
<th>At least once</th>
<th>At least twice</th>
<th>At least thrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous JavaScript</td>
<td>56.18%</td>
<td>39.77%</td>
<td>33.75%</td>
</tr>
<tr>
<td>Input placeholders</td>
<td>30.04%</td>
<td>15.47%</td>
<td>9.48%</td>
</tr>
<tr>
<td>Semantic tags</td>
<td>29.63%</td>
<td>28.21%</td>
<td>26.65%</td>
</tr>
<tr>
<td>New type of input</td>
<td>25.27%</td>
<td>4.60%</td>
<td>1.00%</td>
</tr>
<tr>
<td>Auto focus on input</td>
<td>8.07%</td>
<td>0.99%</td>
<td>0.07%</td>
</tr>
<tr>
<td>Canvas</td>
<td>5.82%</td>
<td>4.53%</td>
<td>3.56%</td>
</tr>
<tr>
<td>Deferred JavaScript</td>
<td>5.43%</td>
<td>4.46%</td>
<td>1.66%</td>
</tr>
<tr>
<td>Required on input</td>
<td>5.05%</td>
<td>3.60%</td>
<td>3.18%</td>
</tr>
<tr>
<td>Audio or Video</td>
<td>1.77%</td>
<td>0.55%</td>
<td>0.35%</td>
</tr>
<tr>
<td>Offline manifest</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Any of the above</td>
<td>62.82%</td>
<td>47.80%</td>
<td>42.68%</td>
</tr>
</tbody>
</table>

Count is based on any state having the element more than the specified number of times.

Input placeholders  Input placeholders offer developers the option to set a text in a input element that is shown until the users inserts something. For example, the text “Search...” in a search box. This used to be done by writing some boilerplate JavaScript that inserts the text into the box and removes it as soon as the users touches the box. Then the users empties the box it puts the text back. Using this new features cleans up that boilerplate code.

Semantic tags  We found it interesting to see a wide adoption of semantic tags. Semantic tags do not contribute to the regular functionality of a web application. They provide extra structure to the DOM tree, which can be used by, e.g., screen readers. We will address this in more detail in Section 11 when we cover accessibility in full detail. It is the only feature in this top five that has nothing to do with JavaScript.

New input types  The adoption of new input types is also well on its way with one in four applications adopting it. The new input types allow a developer to specify that a input element only accepts for example dates, or email addresses. Browsers now have the ability to validate those input types for developers. This saves the developers from a lot of boilerplate code that validates the input manually. Another advantage is that on mobile browsers (and some desktop browsers) the input medium changes depending on the input type. For example, when the input type is date, a mobile browser will open a date picker. When the input type is email, a mobile browser will use a special keyboard that shows characters used in email addresses, like the “@”.

Auto focus  The last feature to make it to the top 5 is the autofocus feature which like most other features on this list saves the developer from writing boilerplate code. In this case the developer does not have to write code that lets the user interface focus on a certain input element after the page is loaded.
We can conclude that of the new HTML-5 features, the features that save developers from writing boilerplate are the most popular ones. The two exceptions are the ability to load JavaScript asynchronously and semantic tags.

The least adopted feature is the use of the offline manifest, which allows developers to specify which resources should be stored on the client-side and which resources should be loaded from the server. It can even be used to run a web application completely offline.

8.2 Resource compression

In contrast with the wide adoption of HTML-5, the adoption of compressing a web application’s content is still lacking behind. By inspecting the headers of all HTTP requests we observe that 20.0% of the resources were compressed. Among the compressed resources, Gzip appears to be the most popular technology governing 99.96% of all the compressed resources. Compressing resources can reduce their size by up to 80%. This can make a page load substantially faster. We will discuss this in more detail in Section 10.3 when we discuss performance in full detail.

8.3 Mobile ready

As mobile browser usage increases applications have to get prepared to show their content on screens significantly smaller than the desktop screens most web applications were designed for. Figure 8.1 shows Cisco’s forecast on mobile device usage up to 2017. We can see that the majority of mobile devices are smartphones.

One way an application can indicate it is mobile-ready is by setting the viewport meta tag in the header. In our dataset 12.4% of the applications declare this tag on their index page. Note that an application might not have the meta tag declared on its index page because they redirect requests with a certain resolution to a special mobile version of their application. This case we did not investigate.
8. The state of the web

Figure 8.1: Cisco Forecasts 11.2 Exabytes per Month of Mobile Data Traffic by 2017

8.4 Volume

To get a grasp of the size of applications, we looked at several factors. To test how much resources can be found using a JavaScript-enabled crawler, we tracked the number of JavaScript and CSS resources that are imported after the index page is loaded. We count the number of embedded script elements and style elements, as well as the number of imported scripts and style sheets. We observed that on average, 29.65 CSS resources and 636.24 JavaScript resources were loaded when the crawler navigated further into the application. We can see the distribution of the sites based on the percentage of their resources that are loaded after the index page in Figures 8.2 and 8.3. There is a notable difference between JavaScript and CSS in these graphs. We observe that more sites tend to load all the CSS they need at the index page. JavaScript on the other hand more often loaded later on in the application. In both figures we see a trend at 0% to 5% and 95% to 100%, meaning the sites either load everything at the index, or nearly nothing at the index page. We did not find an explanation for the peak we see at around 50% in Figure 8.2.

The overall sizes of the resources we found can be seen in Table 8.2.

---

8.4. Volume

Figure 8.2: CSS loaded after the index

Figure 8.3: JavaScript loaded after the index
### Table 8.2: Web application sizes

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>$q^\frac{1}{2}$</th>
<th>Median</th>
<th>$q^\frac{3}{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM size (Kb)</td>
<td>63</td>
<td>13</td>
<td>32</td>
<td>76</td>
</tr>
<tr>
<td>URLs</td>
<td>305</td>
<td>11</td>
<td>72</td>
<td>306</td>
</tr>
<tr>
<td>States</td>
<td>699</td>
<td>45</td>
<td>306</td>
<td>945</td>
</tr>
<tr>
<td>CSS imports</td>
<td>16.94</td>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>JavaScript Imports</td>
<td>330.94</td>
<td>2</td>
<td>11</td>
<td>53</td>
</tr>
</tbody>
</table>

Note: URLs and States are per site. Other statistics are overall.
Chapter 9

Operational errors

Now that we have discussed the state of the web and we have observed the volume of these applications we are going to discuss the errors and other malpractices that occur in web applications. We started off by applying the W3C validator\(^1\) to every state all the web applications in our dataset. From those results we took the three most severe warnings: double id tags, lacking Doctype and broken layout. We will discuss each of these errors in more detail in the consecutive chapters.

9.1 W3C Errors

Table 9.1 shows an overview of the number of errors detected by the W3C HTML validator. The table shows the average and quantiles of the distribution of errors and warnings, per web application. Since larger web applications tend to have a higher number of errors, we also show the total number of errors and warnings normalized on the number of URLs and states the web application has.

Table 9.1: W3C Errors and warnings as reported by the W3C validator per web application

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>q(\frac{1}{2})</th>
<th>Median</th>
<th>q(\frac{3}{4})</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique errors</td>
<td>58.3</td>
<td>12.0</td>
<td>30.0</td>
<td>69.0</td>
<td>12388.0</td>
</tr>
<tr>
<td>Total errors</td>
<td>23,984.1</td>
<td>263.0</td>
<td>3,991.5</td>
<td>18,625.8</td>
<td>2,253,278.0</td>
</tr>
<tr>
<td>Unique warnings</td>
<td>3.7</td>
<td>0.0</td>
<td>1.0</td>
<td>3.0</td>
<td>475.0</td>
</tr>
<tr>
<td>Total warnings</td>
<td>8,070.8</td>
<td>0.0</td>
<td>30.0</td>
<td>825.5</td>
<td>1,065,212.0</td>
</tr>
<tr>
<td>Total errors per URL</td>
<td>805.5</td>
<td>10.5</td>
<td>32.1</td>
<td>144.4</td>
<td>506,404.0</td>
</tr>
<tr>
<td>Total errors per State</td>
<td>34.6</td>
<td>6.2</td>
<td>14.4</td>
<td>33.0</td>
<td>2,482.1</td>
</tr>
<tr>
<td>Total warnings per URL</td>
<td>182.9</td>
<td>0.0</td>
<td>0.5</td>
<td>7.8</td>
<td>63,127.5</td>
</tr>
<tr>
<td>Total warnings per State</td>
<td>12.5</td>
<td>0.0</td>
<td>0.2</td>
<td>2.9</td>
<td>1,882.9</td>
</tr>
</tbody>
</table>

Note: The unique errors are grouped by type. This means that if a developer makes the same mistake twice, it will count the error only once.

\(^1\)http://validator.w3.org/
9. Operational errors

9.2 Double ID tags

Manipulating the DOM from JavaScript often happens by querying the DOM for a certain element and then doing operation on that element or its children. Another use of the ID attribute is that when it is put behind the fragment identifier # in the URL, the browser will try to scroll to that element. Giving the element a unique ID attribute is the fastest and most unambiguous way to select the element of interest. For that reason, the ID attribute of any element should always be unique. Declaring the same ID more than once can result in the browser selecting the wrong element to scroll to or perform the edit or delete operation on.

In our study, we found 52.8% sites that contain states with ambiguous IDs. If we look at those sites and we calculate how many states contain the same ID more than once we see it happening on 35.3% of the states. The distribution of the percentage of states with a ambiguous ID can be seen in Figure 9.1. We see that although most applications make the mistake only on 0% to 5% of their states, there are many applications that make the mistake more often.

A reason this error might occur so often is that when the developer chooses an ID for an element, he or she needs to know which IDs are already taken to be able to make up a new unique ID. However, when the developer is doing this, he or she cannot see all the IDs of elements that might be added at runtime using DOM manipulations. That is because in the development environment, most developers only have static analysis tools at hand, that cannot check these DOM manipulations.

http://www.w3schools.com/tags/att_global_id.asp
9.2. Double ID tags

9.2.1 Case studies

To get a better understanding of these errors we looked at three web applications on our dataset in more detail.

PeK WM

Pek WM[^3] is the website of an OpenSource linux window manager. It contained the most double IDs of all web applications in our dataset. Their website contains a history of all the code contributed to the product. One view shows the difference between two versions with optional annotations on a line that differs. A screenshot of this view can be seen in Figure 9.2. The markup is done using a table and one line is presented as shown in Figure 9.3, where the line has the id `src/pekwm.hh80`. It seems that the ID is based on the line in the original file this annotation refers to. This piece of HTML is clearly generated by JavaScript or the server. The problem is that when there are multiple annotations on one line, the same ID is used more often.

[^3]: www.pekwm.org

Figure 9.2: Screenshot of Pek WM version comparison.
9. Operational errors

![Facebook Data Use Policy](https://www.facebook.com/about/privacy/other)

Figure 9.4: Screenshot of Facebook’s Data Use Policy.

```html
<tr class="commit_diff_line" id="src/pekwm.hh50">
<td class="commit_line_number">50</td>
<td class="commit_diff_line">static void forceReload (void) {
    _cfg_files.forceReload ();
}</td>
</tr>
```

Figure 9.3: Sample of the website of Pek WM’s DOM

The web applications appear not to use the ID reference in JavaScript but only as a fragment identifier. When the user browses to [https://www.pekwm.org/projects/pekwm/commits/5964#src/pekwm.hh80](https://www.pekwm.org/projects/pekwm/commits/5964#src/pekwm.hh80) the browser will show the correct line. This means that the web application does not comply with the meaning of an ID, namely that it is unique, but that it does work for this application, since the double IDs always occur on the same line.

**Facebook** To review a more popular web application we also inspected Facebook. In the part of their application that states their “data use policy” as shown in Figure 9.4, the headers of all paragraphs have the same id: `<h3 id="facebookcontent" ... />`. This means that if one would want to send a URL to a given paragraph on that page, it will always point to the first occurrence of the ID, which is the first paragraph. It is not possible to point to another paragraph. From that perspective, this page is broken.

**Evernote** On the website of Evernote, a note-taking utility, we encountered a subtree of the DOM that is hidden in the browser. This part of the DOM is only visible when using certain mobile devices, creating a mobile-friendly version of their web application. However, in the mobile version, the same IDs are used as in the desktop version. Because the mobile version is not a separate page but part of the page with the desktop version some layout ID’s occur twice. For this particular application, this is a problem since we encountered this line of code in their JavaScript: `var containerWidth = $('#skitch_content').innerWidth();`

4[http://facebook.com](http://facebook.com)
This piece of JavaScript sets the width of the container to the width of the container with ID “skitch_content”. However, since that ID occurs twice in the DOM, one for the mobile version, and one for the desktop version, the width will be set to the first of the two elements. This might not be what the developer intended. Finding this bug is hard since no warning is shown when an ID query returns more than one result.

In these three case studies we have seen how ambiguous IDs can be no problem for the functionality, limit the functionality, or actually break the functionality. Assuming the developer of the web application understands the concept of IDs these errors could have been prevented by notifying the developer of the problem before shipping the website to production.

9.3 No Doctype

Every HTML document should always start with a DOCTYPE declaration\(^5\). It tells the browser which version of HTML it is working with. When no DOCTYPE is specified, the developer cannot be sure which CSS rules will be applied because the browser will run in Quirks mode\(^6\). In Quirks mode the browsers violates the HTML standard and its according CSS standards in order to avoid “breaking” pages build for browsers from the late ’90s. Since Internet Explorer 6 is near extinction (except in China)\(^1\) and Quirks mode its purpose is for browsers from before that time, developers should not have that mode enabled.

In our research we found that 17.3% of the sites we visited the Doctype is not defined or invalid on the index page. If we look at all the states in an application, we found that 61.6% of the web applications have some state in their application that does not define a Doctype.

This error is especially dangerous because it potentially ruins the style of a page without showing any errors depending on the browser version. This is typically something that is hard to test so it is easy for the error to sneak into production. We can see the result of such a difference in rendering in Figures 9.5 and 9.6. The pages are rendered with exactly the same HTML except that 9.5 does not specify a Doctype. To make the difference visible, we have set the background of the page to red and that of the body to green. We can see that in Quirks mode, the green area is much bigger than the red area, even though the CSS and HTML of both pages is identical.

9.4 Broken layout

To be able to display a web page correctly it is best if the browser receives valid HTML. Valid HTML in this case means that the HTML complies to the HTML standard specified in the Doctype. However, during our research we encountered a substantial amount of incorrect HTML, which we will discuss in this section.

HTML consists of 135 element\(^7\) like \(<p>Hello world</p>\) that have an opening tag (\(<p>\) and a closing tag (\(</p>\)) and their children or content in between. However,
16 of these elements called *void elements* do not require a closing tag because they cannot have any content so they close on themselves. An example of that is the image tag: `<IMG />`. Although that seems like an easy-enough rule to remember there are exceptions: when the browser runs in HTML-4 mode there are 16 exceptions but in HTML-5 mode there are 78 exceptions to this rule [54] [19]. Those elements can omit their closing tag all together but they are not allowed to close on themselves. It is easy for a developer to forget the correct usage or placement of any of these 135 elements each with their own attributes and the 78 exceptions.

Developers can introduce incorrect HTML at the server-side but also on the client-side. JavaScript allows a developer to insert any string into the DOM, which the browser will then try to interpret. Besides string insertion, JavaScript is also perfectly capable of adding elements or attributes to places where they do not belong according to the HTML version specified in the Doctype (if any).

In 13.0% applications we found invalid HTML caused by the absence of a manda-
tory closing tag or a closing “>” for the specified HTML version. In all other cases, the browser will **try to make the best of it.** For example, when a `DIV` element is not closed, the browser has to assume that whatever follows is a child of that same `DIV`.

In 19.8% applications there are elements missing or misplaced. For example, we found 189,332 states in 659 applications that used elements that are unknown to the specified version of HTML. This can either be a non-existing element or an element that was introduced in a later version of HTML. In the former case they will be ignored by the browser, in the latter case most browsers will render the element anyway.

Finally there are 9.3% applications that have a problem with their attributes like a double attribute or an element missing a required attribute. For example, we found 5,099 states in 72 applications where the `img` element was missing its required `src` attribute that tells the browser where to load the image from. As a result, the browser will not be able to show the user the image.

Just like a missing Doctype this error occurs without a warning. Without proper tooling, a developer might remain completely oblivious to the fact that he or she is shipping these errors into production.
Chapter 10

Performance analysis

Using a JavaScript-enabled crawler also gave us insight into the performance aspects of the application. In this section we will evaluate the adoption of best practices to reduce page load time. A page load is the process that starts when the users directs the browser to a URL, and ends when the page is displayed in the browser. These best practices are derived from research done by Yahoo [33], Google [32] and others [41, 65]. They can be applied to the resources loaded for the initial page, as well as to any resources loaded later on through javascript code.

To help developers, these companies have developed their own analysis tools called respectively YSlow\(^1\) and Google Page Speed\(^2\). They offer insight into the number of resources downloaded, the time it took to download them, and the time it took to render them in the browser. They also offer suggestions to improve performance using best practices. However, these tools only work on one page load. We on the other hand have applied the technique to every page load throughout the application. In doing so we were able to create a performance overview of an entire application.

We will first discuss what the biggest bottleneck is during a page load. After that we go into best practices that help to speed up a page load. There is an extensive list of best practices provided by Google and Yahoo. We will focus on the issues that have the most impact on performance.

10.1 Bottlenecks

The biggest bottleneck during a page load is the number of resources the browser needs to download before it can render the page. Browsers limit the number of simultaneous requests per host. At the time of writing this is limited to between six or eight simultaneous connections in the latest browser versions [3]. This means that when a page requires 10 resources from the same host to render, it will load the first 6 to 8 resources, and then load the rest when a connection is available. Only when all resources are loaded, the page will be rendered and then displayed. Resources that can be fetched asynchronously are fetched as soon as a connection is available.

Besides the limitation on the number of parallel downloads, every HTTP request has the overhead of the request and response headers. This gets worse when the re-

\(^{1}\)http://yslow.org

\(^{2}\)https://developers.google.com/speed/page-speed/
quests happen via SSL/TLS because of the handshake that precedes the start of a session and the decryption that takes place for each request [56].

Now that we know that the number of requests should be limited, and that the browser first renders, and then displays the page, we will discuss best practices that help the page load time.

### 10.2 Caching strategies

To reduce the number of HTTP requests the HTTP protocol offers the ability to cache resources or do a *check-before-load* process. In HTML-5 a new type of caching is introduced that could replace this. In this section we will discuss each of them.

**Headers** When a resource declares a *Cache-Control* or *Expires* header, this instructs the browser how long or until when a certain resources can be cached. When no strategy is defined, a developer cannot be certain whether the resource will be cached or not [24].

Besides the ability to cache resources the HTTP protocol also offers a *check-before-load* process. This happens using so called *ETags*. An ETag is a unique string for a resource, much like a hash. When the resource’s cache period has expired, the client sends a request for the resource with the last-received ETag of the resource. If the server concludes that the ETag is the same on the server the resource is not modified and the server returns status code 304: *Not modified* without any content. The client can then keep the resource in its cache for another period without having to download the file from the server.

By capturing all the headers during a crawl we gained insight into how often these techniques have been adopted and where developers still have room for optimization. We found that only 57.1% of the intercepted headers define an explicit cache strategy. Ideally all headers should have this strategy enabled.

27.4% of the headers made use of the ETag feature. ETags are not always necessary. When either the calculation of the ETag is expensive or when a developer is certain the resource is only valid for a given time, sending a client a cache strategy is a better choice than using ETags.

**Cache manifest** In HTML-5 there is also an explicit cache control available using the *application cache* [19]. This technique uses a special file called the *cache manifest* that lists all the resources that can be cached permanently in the browser. When a user opens the web application, the browser only has to check if the cache manifest file has been changed. If it has, all changed resources are reloaded. Otherwise the application starts using the cached resources.

Even though nearly half of the applications claim to use HTML-5, not a single web application in our dataset used this technology. This comes as no surprise since this technique is still very young.
10.3 Compress Components

If the browser does have to load a resource, it is best to compress it before its transferred. Even smartphones, which are arguably the slowest Internet devices at the time of writing, suffer almost no performance penalty from decompressing these resources, taking into account the transfer time of the download. That is because compression algorithms can reduce the size of the resource by up to 70%. The exception to this rule of thumb are resources that are smaller than 150 bytes or images and other binary files. That is because images and binary files are often already zipped and therefore cannot be compressed any further.

Looking at our data, we see that only 20.0% of the headers used compression. Unfortunately our dataset does not tell us what type of file was transferred but we find it unlikely that the 79.98% of uncompressed resources were all binary files of smaller than 150 bytes.

Amongst the compression techniques, GZip is the most popular compression format. It was standardized by the GNU project [21] and at the time of writing, all current browser versions support it. Of the compressed resources 99.65% used GZip. The second-most popular technique seems to be Deflate, which was used 0.4% of the time.

10.4 Put stylesheets at the top

When the browser loads a page, it first receives the HTML, then parses the DOM, reads the stylesheets from the head element and then starts rendering the content of the body element. While rendering, it uses a CSS engine that compiles the rules defined in the stylesheets. If a stylesheet is added after the head element, the browser is forced to re-compile the rules and re-render the page. This is an expensive process and should be avoided. Therefore, the style sheets should always be on top.

However, we found that 56.5% applications violate this best practice at some point on their applications. Of these applications, the distribution of the percentage of states that violate this practice can be seen in Figure [10.1]

64.1% Of the applications repeat the error on more than 10% of their states. It seems like that 64.1% is not aware of the problem, and therefore repeats it throughout the problem. The other 35.9% might be aware of the problem, but are unaware that their applications contain that problem.

10.5 Make scripts unblocking

CSS resources are not the only resources with a potential performance penalty. Developers should also be aware of the different ways to load JavaScript. Importing JavaScript can be done in two ways: embed code in the DOM in a script tag or keep an empty script tag that links to a JavaScript file. Since HTML-5 a developer can also specify the defer and async attribute.

When JavaScript is declared in the head element, as it should be, the browser will block until the resource is parsed and its execution has started. If the script tag links to an external document, this means it has to be downloaded before it can be parsed
10. Performance analysis

CSS loaded outside the Head element

Figure 10.1: Percentage of states where CSS is loaded outside the head element

and executed. All the while the user will get no visual feedback. To circumvent this, developers worked around the problem by adding the JavaScript in the body element, right before its end tag <!--body>. That way the DOM is rendered and drawn before the script loads and the user can see the page, albeit without the JavaScript interaction.

Because this technique is a work-around a new standard arrived with HTML-5 that allowed the developers to load a resource without blocking the browser from the head element. This is done by declaring the defer or async property. Both load the resource without blocking and can specify a method within the resource that should be run as soon as the browser has loaded the resource. The difference between the two is that when two resources are loaded with the async attribute they can start as soon as the script is loaded. When they are loaded with the defer attribute, they are run when the browser is finished parsing. This is useful for scripts that do not write to the DOM. This process is illustrated in Figure 10.2

Figure 10.2: JavaScript execution using async, defer or regular loading

56
We have already seen in Figure 8.1 that 56.18% of the web applications are already using the `async` attribute and that 5.43% are using the `deferred` attribute. We also looked at the distribution between the import of JavaScript in a blocking way versus a non-blocking way. Blocking is when it is an in-line script, or an import in the header without a `async` or `defer` attribute. Otherwise it is non-blocking. We assume that when a script is not in the `head` element it is placed right before the `</body>` tag. In Figure 10.3 we see the distribution of sites, according to the percentage of non-blocking JavaScript. The average lies at 39.81%. We also observe that a large proportion of the sites only have blocking JavaScript.

Figure 10.3: Distribution of sites, according to the percentage of blocking JavaScript

One might wonder if it is always necessary to block the page for this script, or that the developer is unaware of the fact that the script is blocking. We will discuss this further in 12.3.

---

3 All credits to http://peter.sh/experiments/asynchronous-and-deferred-javascript-execution-explained/
Chapter 11

Accessibility

Accessibility refers to the practice of making web applications usable for people with a physical disability. Tim Berners-Lee, W3C director and inventor of the World Wide Web, formulated its importance as follows:

“The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect”

To allow people with a disability to use any web application, a developer has to comply to certain rules. In this section we will look at some of these rules and whether they are adopted in practice. The rules we look at concern people who are visually impaired. These people use so-called screen readers to view a web page. A screen reader parses the HTML just like a browser does, but unlike a browser it also reads the page aloud to the user. For people with both visual and hearing impairments a screen reader can read a web page and render it in Braille.

Reading content from HTML can be a tricky process for a screen reader considering the structure of a web application. If the screen reader would start reading top to bottom, like a book, it might take some time on some web applications to reach the important information.

Another problem is that the visual layout of a web application tells people without a disability a lot about the context of the information. When reading top-to-bottom, this context is much harder to convey.

To cope with these two problems the HTML standard comes with rules to help screen readers present the information in a correct way, allowing disabled people to use the web just was anyone else.

11.1 Meta information

One way to help a screen reader present the user with the correct information is by inserting meta information in elements. We will discuss three of these techniques in this section.

11.1.1 Images and figures

When the page is read by a screen reader all information that is contained in images or figures is lost to the reader. To prevent this from happening developers can specify
11. Accessibility

an alt or longdesc attribute that explains what the image is about. In Figure 11.1 we see the distribution of sites and the percentage of images that are annotated with the required attribute. We see that there is a tendency to either do it with all images, which is good, or to do it with no images at all, which is mostly bad.

![Percentage per of image meta info](image)

Figure 11.1: Distribution of sites and the percentage of images they have annotated with the required alt or longdesc attribute.

11.1.2 Tables

Tables offer users a visually structured presentation of information. Now consider a screen reader reading a table row by row, column by column. The larger the table is, the higher the chance a user loses track of where the screen reader is.

To make tables useful for users of a screen reader, the table should have a summary, which tells the users the most important information. Besides that, it should declare the header rows or columns so that the reader has a context of the information that is being read aloud.

In our dataset we found that of the 2212 web applications with a table in any of their states, 91.4% have a table on their site without heading rows or columns. 98.4% of the sites have a table without a summary or caption. Only 5.6% of the sites adhere to the specification.

Almost a decade ago it was common practice to use tables as a layout mechanism a website. This is considered a bad practice today. Those applications will not have a caption, summary or headers because that is not what the table was designed for. However, it is still just as bad for screen readers since they do not expect layout to be
11.1.3 Labels for input elements

Even if an input element uses a placeholder, it might not always be clear to the user what the program expects him to put in. For that reason, most interfaces also surround the input element with information about the input. The official way to do this is to put the information about an input in a label element. That label has as an attribute for which points to the id of the input element.

Although this is not hard to implement, 36.2% of the sites with input elements do not declare an appropriate label for an input ID. The sites that do use this technique only apply it to 19.67% of their input elements on average. Just 3 out of the 2306 sites applied the technique to all input elements.

11.2 Navigation assistance

Besides the meta information concerning one element, the screen reader also works better with some knowledge about the structure of the application. It is especially useful for a screen reader to know how the page is constructed in terms of the roles of particular sections of the application. That way, the screen reader can for example offer the user to first read the navigation section of the page, before reading the content. This makes for a much better user experience than when the screen reader starts to read the whole page from top to bottom.

Before HTML-5, developers could communicate the intent of certain elements using the role attribute. For example, the container containing the navigation would be written like \(<\text{div role='navigation'}>\). In HTML-5, this has become easier by introducing containers with these roles that are very common. So instead of writing a div, the element is called \(<\text{nav}\>\). Other new semantic elements include header, footer, figure and summary.

In Figure 8.1 we saw that the adoption of semantic tags made it to the top 5 HTML-5 features in terms of adoption. 29.63% are using it. In Figure 11.2 we can see how many sites implement these practices. We observe that 60% of the web applications do not give the screen reader any kind of information about the structure of the page. 25% of the sites use both semantic and roles and 16% of the sites use either one of the two methods.
Figure 11.2: Usage of navigation indicators for screen readers
Chapter 12

Discussion

Now that we have presented our most important findings, we will discuss these findings and our research methodology. After that we discuss our recommendations for future research and development.

12.1 Threats to validity

During this study some assertions were made based on our dataset, and the tools we have used. In this section, we discuss the threats to the validity of these assertions.

12.1.1 Internal validity

Crawljax performance improvements In Section 5.4.1 we concluded that the new concurrency framework led to a higher coverage and a faster crawl. These two factors influence each other since some crawls were stopped after a certain time. The faster a crawler is, the more states it will find in a fixed amount of time. That performance boost has made the coverage higher in such cases.

Page load time In the Section 10 we discussed best practices that can lead to a faster page load. We left out the specifications of the computer the calculations are performed on, and its bandwidth to the server. On a relatively slow computer, with the web application available at a very high speed, the bottleneck may move from the number of HTTP requests, to amount of rendering that has to be done by the browser. Our conclusions are based on what should happen theoretically in most cases, based on today’s hardware.

State detection While discussing the DOM metrics in Section 7 the state of the web in 8 the performance in section 10 and accessibility in 11 we drew a number of conclusions based on the number of states we found. As discussed in 3.3.1 it is hard to determine whether a changed DOM is actually a conceptually different state, or that the DOM has not changed in a relevant way. This may cause several sites to have a high number of states per URL. All metrics derived from the number of states containing some property can be deluded in that sense.

Crawl completeness During our crawl we only interacted with anchor, button and input elements. This means that we only crawled a subset of all potential states.
12. Discussion

All conclusions based on the completeness of a site are therefore in fact the under bound of spectrum, since there can be many more states.

Another factor that comes into play that some sites were not crawled exhaustively. In Section 7.2 we only looked at exhausted crawls but all other conclusions based on either exhausted crawls or crawls that were stopped after exactly two hours.

Placement of non-blocking scripts In 10.5 we assumed that when a script tag is not in the head element, it is placed right before the </body> tag. We see this happening most in practice. However, when it is not placed right before the body ends but somewhere else in the DOM, the same blocking consequences apply as when it is placed in the head element.

12.1.2 External validity

Web applications change continuously In Section 5.4.1 we compared the coverage and speed of the crawler by crawling a number of web applications twice. To be able to make an accurate estimation, we crawled multiple sites. However, to make the numbers more meaningful, we crawled public sites so that the measured performance is the actual performance we can expect from crawling the public web. These web applications are not under our control and might change significantly during the crawl, influencing our measurement. However, we tried to find sites that are content-wise as static as possible.

Another subject where dynamism comes in to play is when we crawl a web application for our result set. Consider the case where the crawler is about to finish, and the web application is updated to a new version. The crawler will then consider the new version as a series of new states. It cannot make the distinction between an old or new version of a site. In the most extreme case, the site could continuously be changing, even though it has a small number of states, and the crawler will interpret this as one large web application.

The analysis is as good as Crawljax performs This research completely relies on the JavaScript-enabled crawler called Crawljax. All assumptions were made on the bases that Crawljax behaves as it proclaims.

Web applications selected are as random as the tool provides We have no knowledge of the algorithm behind the random web application tool we have used. We only crawl each web application once. We do not know how that web application obtained or distributes the web applications it distributed to us.

12.2 Recommendations for further development

Clicking all hidden elements In this study, all anchor, button and input elements were interacted with. There are many more elements that could have triggered actions, but trying them all out took too much time. If Crawljax could detect which elements actually have actions attached to them, this would lead to a significant speed improvement.
Create a plug-in ecosystem Crawljax was designed, and is still used as a research tool. Because of that, interesting research has been done using Crawljax’s plug-in architecture [61, 44, 13, 62, 9, 28, 23, 49]. Of these plug-ins however, only few are usable for third party developers because they are either out-dated or broken. It would be an advantage if Crawljax would have an ecosystem where these plug-ins would be kept alive, and tested with every release, so that they can evolve with the main product. That way every one can keep up leveraging the insights these tools offer.

Time the page loading During this resources we took a theoretical approach to the performance of a website. We did not measure the time it took to load a page, or download a resource from the server. This is certainly possible with the experimental setup that we have proposed. Adding this information could inform a developer where the application struggles with performance and which resources are the heaviest to load.

12.3 Recommendations for further research

Security In this study we did not look at security, though there is great potential to research the possibilities with Crawljax. There have been successful experiments with Crawljax before [37, 63, 13], but not on a large scale. Topics for interesting research include:

- mapping the different paths that are available with different security privileges,
- inspecting headers for security parameters,
- inspect different OAuth scopes required to enter different parts of a web application.

Static JavaScript and CSS analysis In this study we have looked at errors in HTML by doing a static analysis on dynamically loaded DOMs. By doing the same for dynamically loaded JavaScript and CSS we might gain insight into how much errors occur in those other two of the three components that make up the majority of the web. In Section 8.4 we have seen that websites make heavy use of these technologies. We also saw that a significant number is loaded after the index page. This means that analyzing these dynamically allocated resources is necessary to get a good overview.

JavaScript runtime errors JavaScript is notorious for failing silently when an exception is thrown. There are tools that can detect such errors1. By using such a tool on a big corpus we could gain insight into how many errors websites tend to miss because of silent failures.

Should JavaScript block? In Section 10.5 we have seen that the majority of web applications have blocking scripts in them. This might not always be necessary. By investigating which scripts actually modify the DOM before it is rendered we

---

1http://errorception.com/
could gain insight into the number of scripts that actually need to be blocking, and the ones that could be imported in an asynchronous fashion. The latter would give such sites a faster page load.

**Better fingerprinting** At the heart of our research lies the way we define a new state by making a comparison on the DOM. During this study we have done so by comparing the DOMs as two strings. This is a rather sensitive comparison. There are other ways to compare DOM trees known in literature \([39, 29, 18, 64]\) but they have not been tested for this type of crawling yet.

Every aspect of crawling improves by leaps when a better DOM comparison can be made. We have shown in Section 7 that a number of web application are very dynamic. This means that states can change every time the browser opens them. Consider the case where a web application shows the current time, no state will have the same DOM twice. This makes state navigation hard for the crawler since it cannot verify in which state it is. Once better fingerprinting is in, more effective navigation could be implemented as well because the crawler will not be lost as often.

**True mobile readiness** In 8.3 we have seen that 12.36% of the web applications indicate that the application can be viewed on a device with a small screen. As we mentioned, this is not the only way to indicate an app is mobile-ready. It would be interesting to do a crawl where one would run a mobile browser and a desktop browser and see if the web application is actually different. This should give a better overview of how much of the web is mobile-ready.
Chapter 13

Related work

In this chapter we discuss how this research relates to other research that has been presented in the past. This thesis touched roughly upon three themes: crawling to gain exposure, crawling to test applications, and analyzing modern web application’s JavaScript and CSS. We will discuss how our research differs from other studies for each of these themes.

13.1 Crawling for exposure

The nature of crawlers has always been to discover as much data presented in web pages or applications as possible. The first documented crawler discovered pages to measure the size of the web [27]. Soon after Google followed with the goal to index the entire web [16]. Soon after that a number of open source crawlers appeared that tried to achieve the same goal or help developers and researchers gather statistics [30, 15].

When the first web applications arrived that required input to reach certain states, crawlers that could insert data followed suit [55, 38]. The term “deep web” was first explained in 2001 [11] and covered both input-dependent states and states only reachable from JavaScript.

In 2004 Alvarez et al proposed a method using a smaller headless browser to be able to execute JavaScript code and extract links from a dynamically generated anchor elements [5]. The research unfortunately does not present any empirical results.

It was not until 2008 that the first JavaScript enabled crawler Crawljax was invented that could cope with dynamic pages [42]. The paper tested the JavaScript enabled crawler on six applications, and shows it finds substantially more states by interacting with JavaScript. In 2013 a larger study was done to investigate the dynamism of the web using this crawler. The study covered 500 web applications but with a maximum crawl depth of three levels and a maximum of 50 states [9]. It differs from our study in that it interacted with additional HTML elements that we did not include: DIV, SPAN and IMG. In our research we covered 3,422 web applications and did not impose any of the other boundaries except for a time constraint. As a result, our dataset includes applications with up to 26,921 states. Since only 28% of the applications we covered contain less than 50 states, we can conclude that our dataset is far more extensive. Assuming all 500 applications had 50 states, our dataset contains 112 times more states.
13. RELATED WORK

13.2 Crawlers as a testing or analysis tool

Both traditional crawlers and JavaScript-enabled crawlers have been used to test web applications. The W3C consortium for example offers a “validation suite”¹ that makes use of a traditional crawler to validate the HTML, CSS and internationalization of the discovered pages.

Crawljax was originally designed to crawl and index applications that made use of AJAX communication. However, since its inception, the authors realized its true potential lay in testing [45, 47, 28]. It has since been used to find JavaScript smells [23], to find cross browser incompatibilities [62, 44], to assess CSS coverage [43], to inspect the security of web widget interactions [13], to generate regression tests [61], and to perform mutation testing [49].

The work that is most similar to Crawljax in terms of testing is a tool called Webmate, introduced in 2012 [20]. Webmate has been used to automatically generate tests, and to test web applications for cross-browser incompatibilities. Although the authors expect Webmate to perform better than Crawljax in terms of coverage, no persuasive empirical data has ever been advanced or adduced that supports that claim. Unfortunately Webmate is proprietary software, preventing us from using it in this study, or to make a comparison to Crawljax in terms of performance.

13.3 Coping With JavaScript’s Dynamism

As JavaScript became more popular it also attracted more research. JavaScript is a dynamically typed language. This makes it easy for developers to make type errors that can only be seen at runtime. For that reason, a lot of research has been conducted that attempts to do some static analysis of JavaScript’s types. In 2005 and later in 2009, attempts have been made to present a static program analysis that can infer type information for JavaScript programs. As a result, common programming errors could be detected statically and programming comprehension can increase. [6, 35].

In 2010, Richards et al conclude from a large empirical study that:

“Rigidly static type systems are unlikely to be usable with JavaScript; any applicable type system must be open to the very real possibility of object protocol changes. Any typing framework which depends on the typing of function parameters will struggle with JavaScripts high degree of variadicity.” [60].

In 2011, Jensen presented the first static analysis tool that is capable of analyzing the control flow and data flow between JavaScript and the browser [34].

In 2011, Richards et al did an analysis of dynamically generated JavaScript, that is then evaluated and run in the browser. This is often considered a bad practice. However, they found that of the 10,000 sites they tested, 82% used this functionality [59].

In 2012, Nikiforakis et al analyzed the JavaScript imports of 10,000 sites and concluded that there is a considerable number of web applications that include JavaScript

¹https://validator-suite.w3.org/
code from external sources that are not taking all the necessary security-related precautions and thus could be compromised [52].
Chapter 14

Concluding remarks

In this thesis we conducted an empirical study of 3,422 web applications using dynamic analysis. The analysis was performed using a JavaScript-enabled crawler that controls the browser and is thus able to execute JavaScript. By using a proxy server we gathered all HTTP requests and responses that were sent and received during the crawl.

To make the results more accurate, we fixed a number of bugs in Crawljax, the crawler we used in this study. We also improved its speed by 150% and its coverage by 190%. By reducing the number of fatal crashes from 33% to 0.6% we made the crawler more robust. Finally we added new features, improved the quality of the code, and made the project more accessible to new contributors. These contributions led to a community with more contributors and downloads of the product than ever before, paving the way for future developments of Crawljax.

We created an experimental-setup where crawlers are distributed over multiple computers to speed-up the crawling process. Because of this unique set-up, we were able to present the largest data-set ever produced using a JavaScript-enabled crawler outnumbering the second-largest study in 112 times the state size. The 216 Gigabyte large dataset contains 3,422 web applications, 96,144,592 HTTP request headers, and 2,565,981 DOMs.

Using this extensive dataset we were able to answer our research questions and draw the following conclusions:

RQ 1: Are static analysis, dynamic analysis, or both using a conventional crawler, adequate tools to inspect modern web applications? To analyze a web application to its full extend the analysis tool needs to cover as much aspects of the application as possible. Do determine the necessity of a JavaScript-enabled crawler, we measured the dynamism of the web in Chapter 7. We proposed three metrics to determine dynamism:

The first metric compared the states per URL. It showed that there are on average 1.9 states per URL. This means that when a static analysis tool would work by testing pages on a URL basis, it would only test 51.8% of the available states.

The second metric looked at the number of states that are invisible to a traditional crawler. It showed that on average 85% of the graph is reachable through hyper-
CONCLUDING REMARKS

links. This is the lower bound of invisible states since we have not looked at all interaction elements.

The third compared the HTML received from the server to the DOM observer in the browser after 5 seconds. We see that 90% of the sites perform DOM manipulations after they are loaded. 64% Change text inside the DOM after they are loaded.

We conclude that a tool based on static analysis, or dynamic analysis using a traditional crawler is not capable of inspecting all states of a web application.

RQ 2: How far is the adoption of best practices and new standards in current web applications? In Chapter 8 we listed the occurrence of the top 10 of HTML-5 features seen in practice, the amount of resource compression, the number of sites that are mobile-ready, and the volume of the applications we analyzed.

We conclude that: there is a high adoption of HTML-5 features that save programmers from writing boilerplate, the amount of resource compression was much lower than we expected, and 12.4% of the applications identifies itself as mobile-ready.

RQ 3: What are severe errors that occur most frequently in practice? In Chapter 9 we took the most severe errors according to the W3C-validator and inspected them in more detail. We found that 52.8% of the websites contain ambiguous IDs. This can lead to misplaced DOM manipulations. We also found that 17.3% of the applications do not define a Doctype, or have an invalid DocType. This causes the browser to enter an unpredictable render mode. Finally we observed that of the sites we analyzed 13.0% contain broken HTML, 9.3% have errors concerning their attributes, and 19.8% have misplaced elements.

We conclude that a substantial amount of web applications contain severe errors that can potentially break the application.

RQ 4: What are the performance issues that occur most frequently in practice? By analyzing all the resources used on a web page and looking at how the DOM is actually rendered we found out that there is still much to gain in the area of performance. Unlike existing analysis tools we inspected performance for all states of the application. In Chapter 10 we give an overview of how best practices are used throughout an the applications we have inspected.

We conclude that especially the compression of all resources and the place and type of JavaScript imports is an often occurring performance penalty.

RQ 5: What are the accessibility issues that occur most frequently in practice? In Chapter 11 we inspected to what extent these web applications adhere to accessibility standards for people with any kind of physical impairment. We observed that 36.2% of the sites give no information for a screen reader about the context of an input field. 60% of the applications offer no information for a screen reader about the structure of the web applications.
We conclude that a substantial number of applications still have a number of opportunities to improve their accessibility. The lack of necessary meta information makes these applications inaccessibly to disabled users.

Our main research question during this research was:

Using a JavaScript-enabled crawler to dynamically test web applications, what are the most severe malpractices that occur in practice, and to what extent is a JavaScript-enabled browser needed to expose these malpractices?

We conclude that from our random sample of the web, we can see that these web applications are very dynamic. Static analysis or dynamic analysis using a traditional crawler does not suffice for these applications.

We have shown that many of these web applications contains numerous errors, do not adhere to best practices, and are mostly inaccessible to people with a disability.

Finally, we have demonstrated that building a tool on top of a JavaScript-enabled crawler like Crawljax helps developers find these mistakes and malpractices.

The time has come to leverage the technologies we have presented to their full extent, enabling developers to improve the quality of their web applications.

In 2012 the United Nations declared the Internet a basic human right. Let us allow anyone with that right to have access to high quality applications.
Bibliography


