

LIU-ITN-TEK-A--15/021--SE

# Automated end-to-end user testing on single page web applications

Tobias Palmér

Markus Waltré

2015-06-10



**Linköpings universitet**  
TEKNISKA HÖGSKOLAN

LIU-ITN-TEK-A--15/021--SE

# **Automated end-to-end user testing on single page web applications**

Examensarbete utfört i Medieteknik  
vid Tekniska högskolan vid  
Linköpings universitet

**Tobias Palmér  
Markus Waltré**

Handledare Karljohan Lundin Palmerius  
Examinator Stefan Gustavson

Norrköping 2015-06-10

## Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

## Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>

## *Abstract*

Competencer wants to be sure users experience their web product as it was designed. With the help of tools for end-to-end user testing, interactions based on what a user would do is simulated to test potential situations.

This thesis work is targeting four areas of end-to-end user testing with a major focus on making it automatic. A study is conducted on test case methods to gain an understanding of how to approach writing tests. A coverage tool is researched and built to present a measure of what is being tested of the product. To ease the use for developers a solution for continuous integration is looked at. To make tests more automatic a way to build mocks through automation is implemented.

These areas combined with the background of Competencers application architecture creates a foundation for replacing manual testing sessions with automatic ones.

# Contents

<b>Abstract</b>	<b>i</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vi</b>
<b>Abbreviations</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Purpose . . . . .	2
1.3 Problem Definition . . . . .	2
1.4 Thesis Outline . . . . .	2
<b>2 Background</b>	<b>4</b>
2.1 Importance of Testing . . . . .	4
2.2 Value of Mocks . . . . .	5
2.2.1 Keeping Mocks up to Date . . . . .	5
2.3 Small, Medium, Large Tests . . . . .	6
2.4 Languages . . . . .	6
2.5 HTTP Request and Response . . . . .	7
2.6 Frameworks and Tools . . . . .	8
2.6.1 Selenium . . . . .	8
2.6.2 Protractor . . . . .	9
2.6.3 Mocha and Chai . . . . .	10
2.7 Architecture at Competencer . . . . .	12
2.7.1 Django . . . . .	12
2.7.2 Node . . . . .	13
2.7.3 Gulp . . . . .	13
2.7.4 Angular . . . . .	13
<b>3 Related Work</b>	<b>14</b>
3.1 Test Case Methods . . . . .	14
3.1.1 Equivalence Partitioning . . . . .	14
3.1.2 Boundary Value Analysis . . . . .	15
3.1.3 Decision Table . . . . .	16
3.1.4 State Transition Analysis . . . . .	16

---

3.2	Test Case Design Guide	17
3.3	Mocks	17
3.3.1	Mocking Third Party APIs	18
3.3.2	Prism	18
3.3.3	ngMock and ngMockE2E	19
3.4	Continuous Integration	19
3.5	Coverage	21
3.5.1	Code Instrumentation	21
3.5.2	Data Gathering	21
3.5.3	Coverage Analysis	22
3.6	Complexity Factors	22
3.6.1	Logical Lines of Code	22
3.6.2	Parameter Count	22
3.6.3	Cyclomatic Complexity	22
3.6.4	Cyclomatic Complexity Density	23
3.6.5	Halstead Complexity Measures	23
3.6.6	Maintainability Index	24
<b>4</b>	<b>Theory</b>	<b>25</b>
4.1	Test Case Design Guide	25
4.2	Mocks	25
4.2.1	Ways of Rigging Data for Tests	25
4.2.2	Mocking Tool	28
4.2.3	Unique Responses	29
4.3	Coverage	29
4.3.1	Elements	30
4.3.2	Global and Local State URL's	31
4.3.3	Test Adequacy Criteria	32
4.4	User Session Data	32
<b>5</b>	<b>Implementation</b>	<b>34</b>
5.1	Setup and Execution	34
5.2	Test Case Methods	36
5.2.1	Decision Table	36
5.2.2	State Transition Analysis	37
5.2.3	Boundary Value Analysis	38
5.3	Test Case Design Guide	39
5.4	Building Mocks	39
5.4.1	Automatic Mocks with Prism	39
5.4.2	Manual Mocks with ngMockE2E	40
5.5	Building Mock Tool	41
5.5.1	Empty Database	42
5.5.2	Url Counter	42
5.5.3	File Name Generator	43
5.6	Coverage	43
5.6.1	Code Instrumentation	44
5.6.2	Data Gathering	47

---

5.6.3	Coverage Analysis . . . . .	49
5.7	Continuous Integration . . . . .	53
<b>6</b>	<b>Results</b>	<b>54</b>
6.1	Tools . . . . .	54
6.2	Test Case Methods . . . . .	55
6.3	Test Case Design Guide . . . . .	55
6.4	Continuous Integration . . . . .	55
6.5	Mocks . . . . .	56
6.6	Coverage . . . . .	59
<b>7</b>	<b>Discussion</b>	<b>60</b>
<b>8</b>	<b>Conclusion</b>	<b>65</b>
	<b>Bibliography</b>	<b>67</b>
	<b>A Implementation of URL Counter</b>	<b>70</b>
	<b>B Config File for Protractor</b>	<b>71</b>
	<b>C Setup Commands for Codeship</b>	<b>72</b>
	<b>D Competencer Architecture</b>	<b>73</b>
	<b>E Test Case Design Guide</b>	<b>74</b>
	<b>F Protractor-E2E-Coverage Tool</b>	<b>77</b>

# List of Figures

2.1	HTTP request and response . . . . .	8
2.2	Selenium RC architecture, image is remake from [1] . . . . .	9
2.3	Protractor architecture . . . . .	10
2.4	Example of a Mocha reporter with passing tests, image from [2] . . . . .	11
2.5	Example of a Mocha reporter with a failed test, image from [2] . . . . .	12
3.1	Equivalence partitioning example with hard limits 15 and 300 . . . . .	15
3.2	Boundary value analysis example with hard limits 15 and 300 . . . . .	15
3.3	State transition login example . . . . .	17
3.4	Prism mode records information flow . . . . .	19
3.5	Prism mode mocks information flow . . . . .	19
3.6	Overview structure of how continuous integration works, image from [3] . . . . .	20
4.1	Example flow over active database setup for testing . . . . .	26
4.2	Example flow over populating a database for tests . . . . .	27
4.3	Example flow over tests with manually created mocks . . . . .	27
4.4	Example flow over automatically created mocks . . . . .	28
4.5	Overview structure of using prism as a middleware . . . . .	29
4.6	Overview structure of data flow for coverage . . . . .	30
5.1	State transition analysis example for a login flow . . . . .	37
5.2	Storing elements in session storage . . . . .	45
5.3	Overall statistics for end-to-end coverage . . . . .	49
5.4	Coverage summation on a by type basis . . . . .	50
5.5	Coverage summation on a by state basis . . . . .	50
5.6	Coverage details for a single state . . . . .	51
5.7	Coverage information for an element and its events . . . . .	51
5.8	Comparison of coverage reports visualized element (top) and web page element (bottom) . . . . .	52



# List of Tables

3.1	Example of decision table with conditions and results, F being false and T being True . . . . .	16
3.2	Decision table example with three conditions, taken from [4] . . . . .	16
4.1	Chosen HTML elements for coverage, from [5] . . . . .	31
4.2	Chosen events for HTML elements, from [6] . . . . .	31
5.1	Color indications based of coverage percentage . . . . .	52
5.2	Color indications based of coverage percentage . . . . .	52
6.1	Comparison of execution times for three test case methods . . . . .	55
6.2	Time comparison for two tests between real backend and mocked backend	56
6.3	Complexity measures for a Prism setup file . . . . .	57
6.4	Complexity measures for Prism functions where A: function proxyMiddleware, B: function setupPrism, C: function customMockFilename, D: function connectInit, E: function middleware . . . . .	57
6.5	Complexity measures for ngMockE2E mock file . . . . .	57
6.6	Complexity measures for ngMockE2E mock file functions . . . . .	58
6.7	Complexity measures for ngMockE2E mock file without injected JSON . .	58
6.8	Complexity measures for ngMockE2E mock file functions without injected JSON . . . . .	58
6.9	Complexity measures for multiple ngMockE2E mock files A: file loginMock, B: file editProfileMock, C: file marketplaceMock, D: file changePasswordMock . . . . .	59

# Abbreviations

<b>API</b>	<b>A</b> pplication <b>P</b> rogramming <b>I</b> nterface
<b>BVA</b>	<b>B</b> oundary <b>V</b> alue <b>A</b> nalysis
<b>CI</b>	<b>C</b> ontinuous <b>I</b> ntegration
<b>CSS</b>	<b>C</b> ascading <b>S</b> tyle <b>S</b> heets
<b>DT</b>	<b>D</b> ecision <b>T</b> able
<b>E2E</b>	<b>E</b> nd-to- <b>E</b> nd
<b>EP</b>	<b>E</b> quivalence <b>P</b> artitioning
<b>HTML</b>	<b>H</b> yper <b>T</b> ext <b>M</b> arkup <b>L</b> anguage
<b>HTTP</b>	<b>H</b> yper <b>T</b> ext <b>T</b> ransfer <b>P</b> rotocol
<b>JS</b>	<b>J</b> ava <b>S</b> cript
<b>JSON</b>	<b>J</b> ava <b>S</b> cript <b>O</b> bject <b>N</b> otation
<b>LOC</b>	<b>L</b> ogic (lines) <b>O</b> f <b>C</b> ode
<b>MI</b>	<b>M</b> aintainability <b>I</b> ndex
<b>NPM</b>	<b>N</b> ode <b>P</b> ackage <b>M</b> anager
<b>SPA</b>	<b>S</b> ingle <b>P</b> age <b>A</b> pplication
<b>STA</b>	<b>S</b> tate <b>T</b> ransition <b>A</b> nalysis

# Chapter 1

## Introduction

This masters thesis is carried out at Competencer and Linköping University department of technology and science. The aim for this master thesis is to come up with a solution for automatic testing on Competencers web application with focus on user interaction tests. This chapter covers the background for the thesis along with the purpose and problem definitions. It also covers information about Competencer and presents the thesis outline.

### 1.1 Background

Testing is a practice to assure that a product works as intended. At Competencer this is important because there is a possibility to lose costumers if the product doesn't behave as it's suppose to. Testing on Competencers frontend is today only done with manual testing and Competencer wants a way of knowing that their users will get the same experience designed for them with more consistency. This can be accomplished by covering their frontend with user interaction tests, referred to as end-to-end tests in this thesis. These end-to-end tests would test different user interacting scenarios in the same way as a user would interact with the web page. If these end-to-end tests passes it assures that the interactions on the web page is doing what they are expected to do.

Competencer wants to have an automated testing process to save both time and resources so that they can focus on developing the product, instead of maintaining tests. They do not want to have an extra backend server to maintain either, meaning that the tests should be able to run without a backend server. It can be hard to know what parts of the product is best to test and to maintain a good structure of the written test cases. Therefore Competencer desires an efficient way to keep track of their tests and guidelines for writing test cases. They also want a way to see what parts of their product have been tested and what would be recommended to test next.

## About Competencer

Competencer is an online platform connecting clients and coaches with a digital video tool. Competencer states this is in the following way:

*"our mission is to make professional advice more accessible, as well as more efficient, and flexible. Through our marketplace and online platform we connect advisors with clients in an easy way and make it possible for them to meet in our digital meeting room, no matter where they live."* [7]

## 1.2 Purpose

The purpose of this masters thesis is to build a foundation for testing web applications in a fully automatic process with a focus on user interaction tests. This will be done to ease the implementation of end-to-end user testing on Competencers application so that Competencer will be able to completely replace their current manual testing sessions with fully automated ones. The purpose is to also give Competencer the possibility to measure and evaluate these tests, which is difficult today since they are performed by hand. Furthermore the purpose is also to build an automatic process, enabling a tester with direct feedback from the test suites.

## 1.3 Problem Definition

The following problems are the ones decided to investigate and answer in this thesis:

1. How can a comparison of different end-to-end testing methods lead to a more effective way of writing test cases?
2. How can a coverage report be built for end-to-end tests without the knowledge of source code and what information should be analyzed to help a tester?
3. How can end-to-end tests be automated and how can feedback from it improve a development process?
4. How does automatic mock creation compare to manual mock creation and how can an automatic process be achieved when mocking backend?

## 1.4 Thesis Outline

Chapter 1 includes an introduction, purpose and problem definition for this thesis. In chapter 2 a background describing the situation for this project is presented. Chapter 3 highlights relevant work and chapter 4 covers theory developed from the relevant work.

Chapter 5 includes implementation of the testing tool, mocks and other areas. Chapter 6 presents this thesis results and findings. Chapter 7 discusses the results and the final chapter 8 presents this thesis conclusions.

# Chapter 2

## Background

This background chapter explains why testing is important, how it supports developing reliable software and the value of using mocks. Here the technical background for this project is presented including the tools and frameworks used.

### 2.1 Importance of Testing

When developing software and writing code it's inevitable for errors and bugs to be included and situations to be missed. Requirements for a software program changes over time and mistakes emerge during changes. In a typical program it is estimated that around four to eight errors exist within 100 program statements [8]. Some companies put down up to half of their time and resources to debug and test their product [8].

Testing applications has a base in a psychological perception of trust towards a product or application. A user might be off-put to let a company handle its credit card details if the same company have trouble handling a message, rendering an image or a link. When working with web applications the attention span for a user is short and the expected quality is exceedingly high, in comparison to desktop software. In order not to lose a customer to a competitor it is vital to make sure the quality of the product is high, or in other terms - well tested.

#### Definition of testing

*“Testing is the process of executing a program with the intent of finding errors.”* [8]

The cost of errors in software development follows the same principle as how expensive it is to retract a product from the market. Detecting something early on is cheap, fast and doesn't hurt business reputation while a critical failure on a live version is vastly more expensive.

When testing web applications there are multiple tiers, layers and approaches to take for evaluating a system. It is possible to test the view-, business- or data tier on both a system level and unit level <sup>1</sup>.

## 2.2 Value of Mocks

Mocks is a practice to simulate an object and mimic its behaviour in a static way. In this thesis they are used to mimic the database communication. Mocking for end-to-end testing can be made in many different ways and which one that is best suited is ultimately decided by the specified testing goal. If one aims at performing an integration test of an application it's important that the interaction and responses are generated from the actual underlying architecture and in that case mocking isn't an appropriate method. In this thesis case mocking is good out of a few different reasons:

1. Decouple backend and frontend
2. Backend highly tested
3. Double environment handling for tests
4. Increased difficulty in maintaining tests

At Competencer the current single page application is built with high a decoupling between backend and frontend with only an API as a bridge in between. This architecture combined with high code test coverage for the backend makes it redundant and expensive to test both backend and frontend in end-to-end tests. Furthermore setting up an environment that tracks states and requirements for both backend and frontend takes additional time and effort and is harder to maintain. From these reasons it's beneficial to include mocks in end-to-end testing to exclude the interaction with the database. Since the backend is already highly tested an assumption that the responses from the databases are correct is made.

### 2.2.1 Keeping Mocks up to Date

One downside of mocking the backend is that if something gets updated, the mocks are affected by this update and needs to be reviewed and updated. Once the mocks are built they are static, meaning they will always act the same way. If the mocks are not updated they can miss changes in the database and the tests will test old code.

---

<sup>1</sup>Unit Level: It refers to individual units of the source code

## 2.3 Small, Medium, Large Tests

According to Wittaker et. al [9] small test, such as unit testing, lead to code quality where medium to large tests lead to product quality. Medium tests is on a higher order than unit testing, such as a function calling another function, and large tests can test the application as a whole. This thesis will focus on end-to-end tests, a higher order large test, which is testing a product in a way that simulates the interactions an actual user would perform. For instance this could simulate the user logging in to an application, sending a message, checking the sent message and then log out. This enables a tester to be sure that the product behaves in way that is to be expected from a users viewpoint. A software solution only tested with unit testing might have perfect code coverage and logic error identification but lack knowledge if all the pieces are put together correctly. Even smaller end-to-end tests can give a lot of assurance that the product works in a way that is expected by relatively small cost.

*“Small tests lead to code quality. Medium and large tests lead to product quality” [9]*

## 2.4 Languages

Common languages used in this thesis are described in the following paragraphs.

### HyperText Markup Language

HTML tags are read by the browser and translated into a visible element on the web page. When writing tests that interact with elements on the page, HTML is used to target the correct element, then JavaScript is added to interact with the element.

### Cascading Style Sheets

Cascading style sheets (CSS) is used to alter the appearance of HTML elements to give a visual expression. CSS is utilized to make the coverage report more appealing, which can be read more about in sections 3.5 and 5.6.

### JavaScript

Chapman [10] describes JavaScript like this:

*“JavaScript is a programming language used to make web pages interactive. It runs on your visitor’s computer and doesn’t require constant downloads from your website.”*

Lately it’s not only used to make web pages interactive but can also be used for client logic and even as backend. The javascript frameworks used to write and run the actual tests are Protractor see section 2.6.2, Mocha see section 2.6.3 and Chai see section 2.6.3.

### JavaScript Object Notation

JavaScript Object Notation (JSON) is a text format for storing JavaScript objects. It



was developed for JavaScript but works well for many languages due to its simple and intuitive structure. It is often used to represent configuration information or to implement communication protocols [11]. In this thesis JSON is used to represent configuration information for Node Package Manager (NPM) so that Node knows what dependencies and versions to install when setting up the environment. JSON is also used to store requests from the client and responses from the backend when mocking it. An example of a stored JSON is presented in the snippet below:

```
1 {
2   "requestUrl": "/api-token-auth/",
3   "contentType": "application/json",
4   "method": "POST",
5   "statusCode": 200,
6   "count": 1,
7   "payload": {
8     "username": "hello@competencer.com",
9     "password": "competencer"
10  },
11  "data": {
12    "token": "0d80b5b63fdf4100d652dfd61950e3b3330bb5b4"
13  }
14 }
```

---

Here the *payload* are sent with method POST to the *requestUrl*. The *data* and the *statusCode* is a response from the server. More about requests and responses can be read about in the following section.

## 2.5 HTTP Request and Response

HyperText Transfer Protocol (HTTP) is used to carry communication between a client and a server, see figure 2.1. The server has endpoints that are like paths or addresses that the client can call. These endpoints are called by sending a HTTP request to an URL that represents the endpoint. The endpoints can be called with or without payload depending on how the endpoint is designed. Payload is the attached data that is sent together with the request. For example, in the code snippet in section 2.4, the call is a request to log in a user. To log in, the user needs an authentication token from the server that is unique for the user. The users name and password are sent as a payload to the request url, where the server retrieves the payload and checks if the user is registered. If the user is registered the server sends back a response with a status code of 200, this particular status code means that everything is good. The server will also send back an authentication token which will authorize the user. If the user isn't registered the server will instead respond with another status code of 400 that translates into *"bad request from client"*.

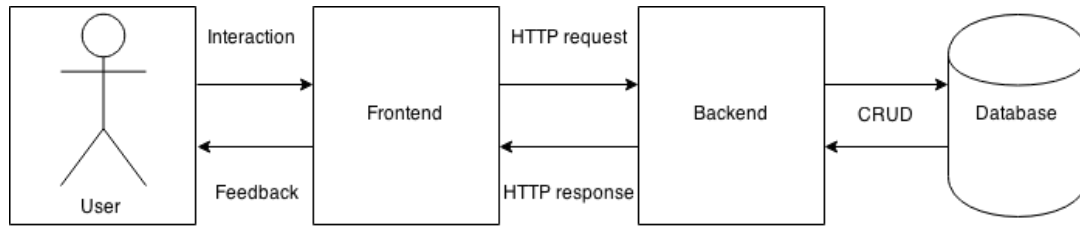


FIGURE 2.1: HTTP request and response

Where CRUD stands for the common database actions Create, Read, Update and Delete.

## 2.6 Frameworks and Tools

There are a lot of different testing frameworks and testing related frameworks available to use. A testing framework provides functionality for writing tests. It can provide a sea of different features depending on its scope.

A test runner is the environment that the tests are developed in and the tool that runs the tests inside the browser. It can either run the tests directly in a browser that is installed locally or it can use a server that has drivers for several different browsers.

### 2.6.1 Selenium

Selenium's founder Jason Higgins was working on a web application at ThoughtWorks [9], this web application was targeting Internet Explorer which at that time was used by the majority of the market. There was only one problem, he was getting bug reports from early adopters of Firefox and when he fixed these bugs he would break the application in IE. In 2004 he started developing what would become Selenium Remote Control (Selenium RC), Selenium RC consists of a Java server that will start and kill the browser and also works as a HTTP proxy see figure 2.2. This server uses JavaScript to control the interactions in the browser [1].

Before selenium was stable Simon Stewart had started working on something called WebDriver at Google. It was a different approach to web application testing than Higgins Selenium. With the use of automation API's the WebDriver was integrated into the browser itself. This made it possible to do things that Selenium couldn't handle but on the other hand as it was integrated to the browser it wasn't easy to make it compatible with new browsers. This was also one of Selenium's strong sides, it could adapt to new browsers with almost no effort. The two projects merged in 2009 and got the official name Selenium WebDriver [12].

Today most end-to-end testing libraries is a wrapper for the Selenium WebDriver which makes it possible to write end-to-end tests in many coding languages.

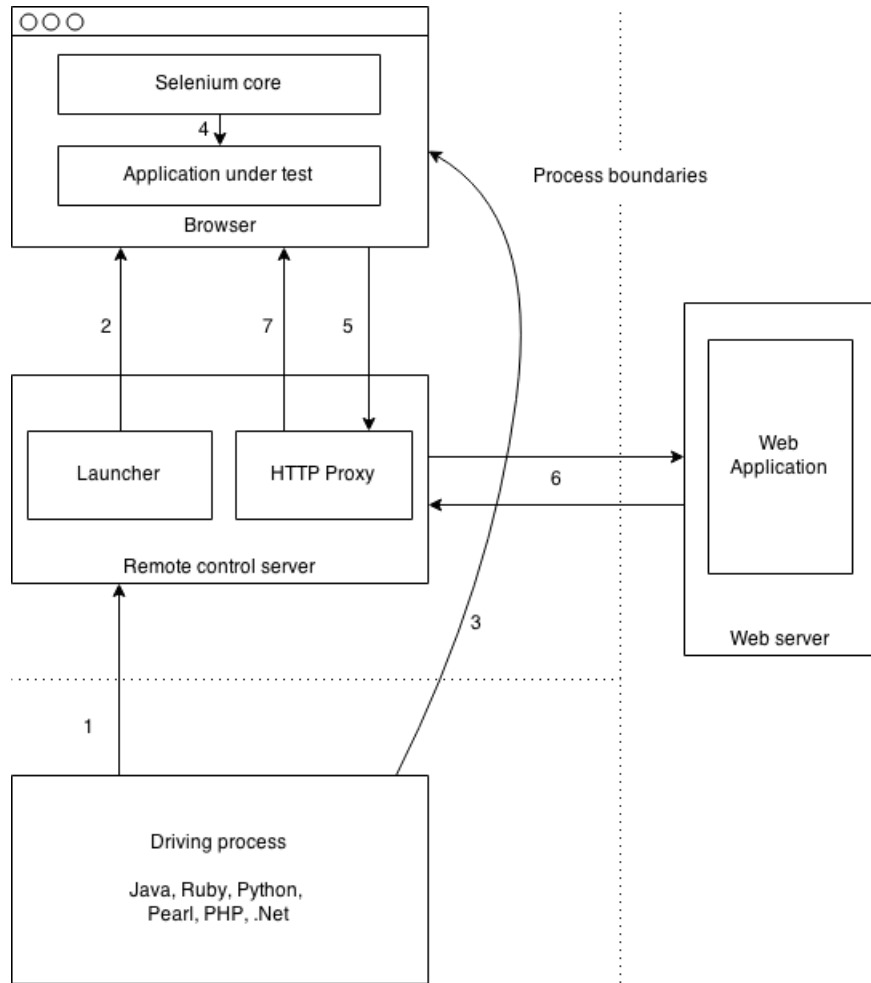


FIGURE 2.2: Selenium RC architecture, image is remake from [1]

Short explanation of what is happening in figure 2.2 from [1]:

1. A client connects to the remote control server
2. The launcher part of the RC server starts a Selenium core in the browser
3. The client can feed the selenium core with code which the core translates to JavaScript
4. The core runs JavaScript in the browser to interact with things on the page
- 5,6 and 7. When browser receives a request to load a page it will get the content from the real page through the remote control server proxy and render it.

### 2.6.2 Protractor

Protractor is one of the wrappers for Selenium WebDriver written in JavaScript. It has support for testing specific practices related to coding with Angular, such as repeaters and bindings. It's developed to perform high level testing like user interaction scenarios. It can be viewed as tool for interacting with a website the same way as a user would.

The tests are written in a test-framework like Mocha, see 2.6.3, together with a test-runner like Protractor. It's possible to run the tests without Selenium on Chrome derived browsers with a Chrome driver. However, if the test is going to be tested in multiple browsers the easiest way is to use Selenium. Then the webdriver part of Selenium converts the tests to JSON requests that then are sent to the Selenium server. The server itself cannot interact with the page so it uses browser drivers to do that. When a request for a specific test suite<sup>2</sup> is completed a test-framework will cover the reports.

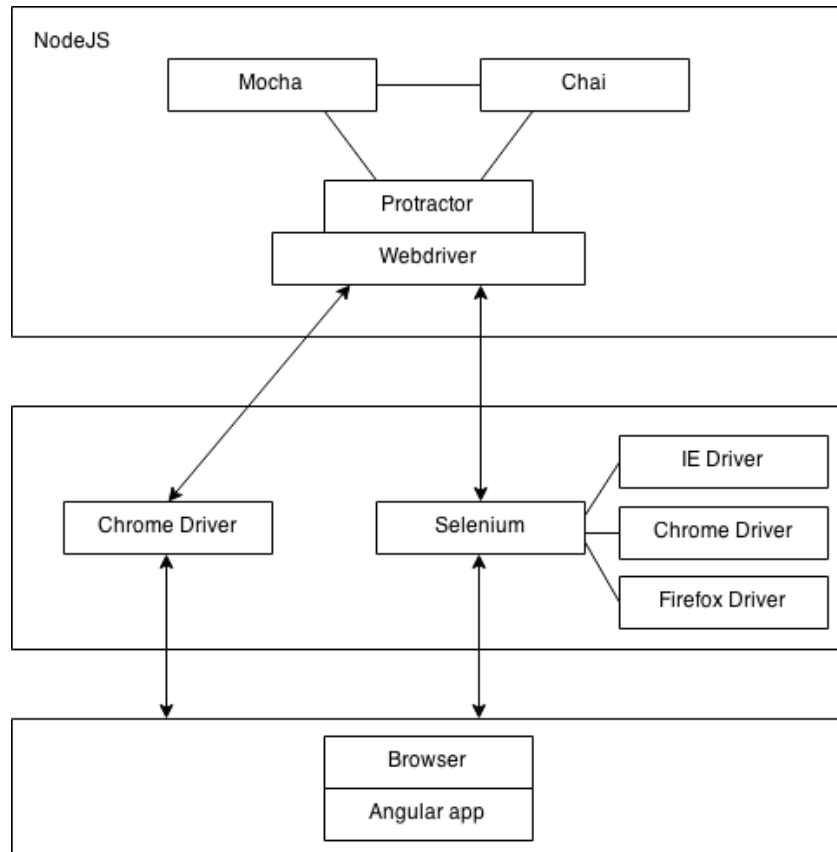


FIGURE 2.3: Protractor architecture

In figure 2.3 it is shown how protractor interacts with Selenium, Mocha and Chai that is used together with the Angular application. The Mocha, Chai and Protractor are used for writing the tests. The tests are sent by the WebDriver part of Protractor to either the local Chrome driver or to a Selenium server that will use the browsers drivers to run the tests inside the browser.

### 2.6.3 Mocha and Chai

Both Mocha [2] and Chai [13] are JavaScript testing related frameworks, Mocha is the part of testing that actually triggers the tests and reports the results. In the second code

<sup>2</sup>Test Suite: It refers to a group of test cases that are related

snippet in section 5.1 *describe* and *it* calls are made. This is what mocha is used for in the tests. Inside these *it* calls *expect* calls are made, these are from the Chai library which is an assertion library. Mocha and Chai work very well together. In figure 2.4 an example of how the Mocha reporter looks like when the tests are passing is displayed. In figure 2.5 an example of how Mocha reports a failed test is presented.

A terminal window titled "mocha — bash" showing the output of a Mocha test run. The prompt is "λ mocha (master): make test". The output shows a test suite for "Array" with two methods: "#indexOf()" and "#pop()". Each method has two test cases, all of which passed, indicated by green checkmarks. The final output is "✓ 3 tests completed (5ms)" and the prompt returns to "λ mocha (master):".

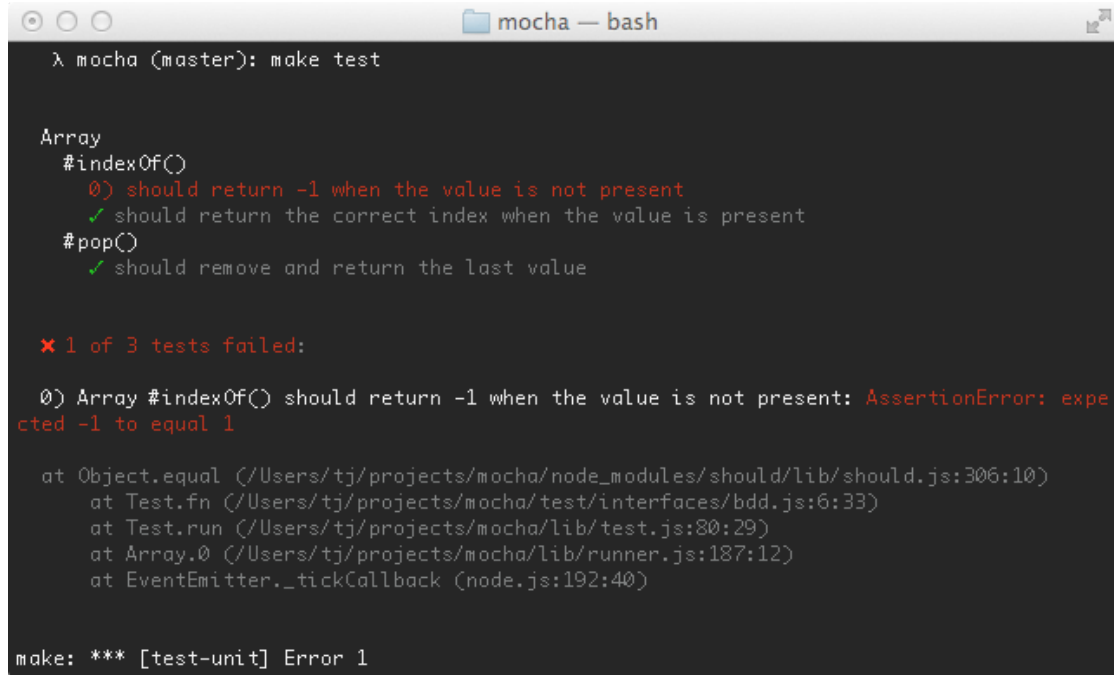
```
λ mocha (master): make test

Array
  #indexOf()
    ✓ should return -1 when the value is not present
    ✓ should return the correct index when the value is present
  #pop()
    ✓ should remove and return the last value

✓ 3 tests completed (5ms)

λ mocha (master):
```

FIGURE 2.4: Example of a Mocha reporter with passing tests, image from [2]



```
λ mocha (master): make test

Array
  #indexOf()
    0) should return -1 when the value is not present
    ✓ should return the correct index when the value is present
  #pop()
    ✓ should remove and return the last value

✖ 1 of 3 tests failed:

 0) Array #indexOf() should return -1 when the value is not present: AssertionError: expected -1 to equal 1
    at Object.equal (/Users/tj/projects/mocha/node_modules/should/lib/should.js:306:10)
    at Test.fn (/Users/tj/projects/mocha/test/interfaces/bdd.js:6:33)
    at Test.run (/Users/tj/projects/mocha/lib/test.js:80:29)
    at Array.0 (/Users/tj/projects/mocha/lib/runner.js:187:12)
    at EventEmitter._tickCallback (node.js:192:40)

make: *** [test-unit] Error 1
```

FIGURE 2.5: Example of a Mocha reporter with a failed test, image from [2]

## 2.7 Architecture at Competencer

This thesis is done at Competencer and testing is performed on their product. Part of the technical background for this work is from the current architecture at Competencer, which can be seen in appendix D.

From the architecture we can see that Competencer uses Angular as their frontend framework and Django as their backend framework. They also rely on Node and Gulp for other tasks, such as enhancing workflow. In the following sections these frameworks will be described in more detail.

### 2.7.1 Django

Django is an open source web framework written in Python aimed at building scalable web apps with ease [14]. The core of Django is a model-view-controller pattern where the model is a relational database. The framework also includes tools for running a development server, middleware functionality, site maps, admin interface and much more. At Competencer Django is used in conjunction with the database PostgreSQL as their backend.

### 2.7.2 Node

Node is an event driven framework designed for building network applications that scale well [15]. Node creates an runtime environment on the server-side for running and working with network applications. The framework is written in JavaScript, is open-sourced and can be run on all major platforms. One important feature is the node package manager (NPM) which enables easy install and updating of modules. Node comes bundled with functionality for creating web servers, networking tools and file system writing.

### 2.7.3 Gulp

Gulp is a streaming build system written in JavaScript as a Node module [16]. Gulp aims at enhancing tasks such as building process and running certain types of environment making development processes faster and more convenient. This tool can for instance enable building of source files, minifying scripts, bundling files, compressing images, running test suites and much more.

### 2.7.4 Angular

Angular is a web application framework for building dynamic views in HTML and works by creating a single page application [17]. Angular interprets a HTML page and binds specific tags and attributes to variables in controllers, allowing two way data binding. At Competencer Angular is used as the primary frontend framework for building web applications.

# Chapter 3

## Related Work

This chapter presents related work for this thesis. That includes different test case design methods, test case design guide, mocking information, continuous integration, related coverage information and complexity factors used for later evaluation.

### 3.1 Test Case Methods

Test cases are derived from requirements and specifications of the software application. This is often described by an external specification of how the software is expected to behave. A test designer would look at an isolated task, its related requirements and decide its valid and invalid input and then what the correct output would be. When testing software it is practically, and sometimes even theoretically, impossible to try all cases. For instance an input field for text on your website can be entered in a near infinite number of combinations. Test case methods are good for breaking down large number of possible test cases to a few cases. This way testing will be more efficient and practical.

#### 3.1.1 Equivalence Partitioning

Equivalence partitioning is a testing method that is suitable on inputs with hard limits. Equivalence partitioning, also known as equivalence cases, is a way to group cases that can be treated as the same. Then one case of each case group can be tested to ensure that it would work for all cases in that section.



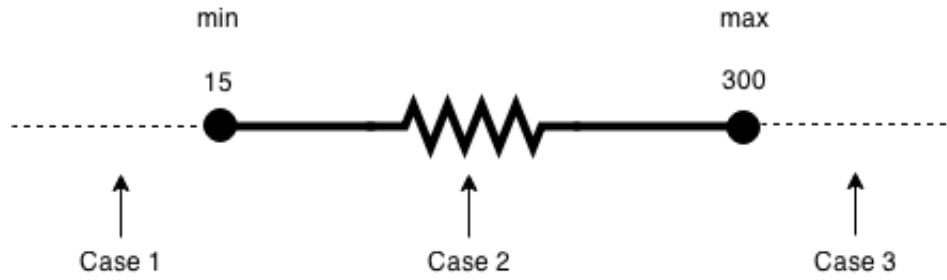


FIGURE 3.1: Equivalence partitioning example with hard limits 15 and 300

In figure 3.1 an example of input that has a valid input range of 15 through 300 is illustrated. All possible inputs have here been divided into three equivalence cases. Case 2 contains all valid inputs and choosing one inside the range will be enough. Case 1 and case 3 tests invalid inputs by looking at the lower and upper outside ranges. With this method an unmanageable amount of test cases have been reduced to three, by for instance testing 7, 123, 377.

### 3.1.2 Boundary Value Analysis

When testing input values it's widely known that more errors occur close to the edges of the domain. With boundary value analysis (BVA) test cases are built around the boundary instead of an arbitrary point in the range. The BVA method is an improved equivalence partitioning method since it captures more edge cases.

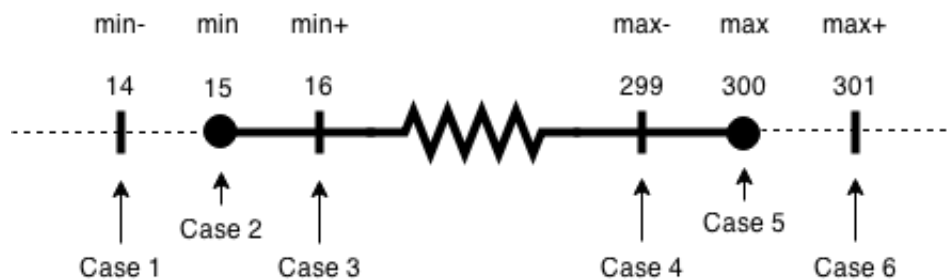


FIGURE 3.2: Boundary value analysis example with hard limits 15 and 300

Boundary value analysis on the example in figure 3.2 would generate six test cases. The test cases are defined by testing the limits 15 and 300 and their closest values. Myers et al. [8] says that

*"if practiced correctly, [boundary value analysis] is one of the most useful test-casedesign methods."*

He continues to say that the hard part of using it is to identify the boundaries correctly.

### 3.1.3 Decision Table

Decision table is a suitable testing method for multiple inputs that combined generates specific outputs. Decision table testing is good for creating a structure for complex logic and is intuitive for both testers and developers to work with. However the size of tables grow very fast and can quickly become unmanageable, therefore it's a good practice to split into sub tables as much as possible.

<b>Condition</b>				
Enter valid username	F	F	T	T
Enter valid password	F	T	F	T
<b>Result</b>				
Error message	X	X	X	
Log in user				X

TABLE 3.1: Example of decision table with conditions and results, F being false and T being True

In table 3.1 an example of a decision table is displayed with two conditions that can either be true or false. Each of these combinations will produce a result that should be tested against. This example is easy to work with since it's only four rules, but it quickly becomes heavier by adding more conditions. Three conditions would give eight rules; four would give 16 and so on following a  $2^x$  formula. Having a complete decision table gives a good overview of cases one might have otherwise missed. It also gives the opportunity to choose which ones to implement and which ones to disregard.

<b>Conditions</b>								
New customer(15%)	T	T	T	T	F	F	F	F
Loyalty card(10%)	T	T	F	F	T	T	F	F
Coupon(20%)	T	F	T	F	T	F	T	F
<b>Actions</b>								
Discount(%)	X	X	20	15	30	10	20	0

TABLE 3.2: Decision table example with three conditions, taken from [4]

Table 3.2 shows a decision table example with different outcomes dependent on different combinations of discount offers.

### 3.1.4 State Transition Analysis

Most testing techniques involve finding errors in an isolated state. State transition analysis (STA) goes beyond this to find errors that might occur between different states. Classic exhaustive testing over different states in a complex application is time consuming and difficult to perform. By defining states and determining loops the amount of test cases to design can be limited. STA can also be viewed as a finite state machine.

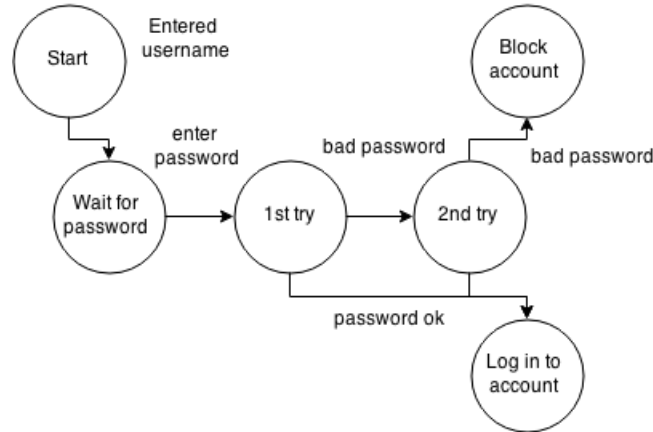


FIGURE 3.3: State transition login example

From a STA we can define a test condition for each state as well as for each state transition. This normally isn't very effective but with a state diagram it's easier to create useful test cases and detect most probable use cases.

## 3.2 Test Case Design Guide

Test case templates are widely used amongst companies that writes a lot of tests. This is a document where the person writes what to test and how to test it before writing actual code. This is done to keep track of the tests created and to make is easier to write the actual test. An example of such a template is Dr. Assassas template [18] where a tester can fill in description, pre-conditions, test steps, post-conditions and some more information. This is a generally good template, however, there are some missing information that would make it fit an end-to-end test better. Test case templates especially created for end-to-end tests could not be found.

## 3.3 Mocks

In the process of writing tests the response is compared to an expected value. The response form the database has an expectancy beforehand. This means that it's possible to isolate the test from the database by hijacking the request and storing the response as a fake response. When a request goes out it doesn't go to the database but instead to a mock that returns what the database is supposed to return. This will decouple the tests from requiring a backend.

Mackinnon et al. [19] explains mocks like this:

*”Until a choice is made, we can write a mock class that provides the minimum behaviour that we would expect from our database. This means that we can continue writing the tests for our application code without waiting for a working database.”*

### 3.3.1 Mocking Third Party APIs

Third party API's are hard to test because the lack of control over sent requests. According to [20] and [21] mocks should only be built on parts that are owned. These third party mocks often result in being more complex and hard to maintain than expected. Freeman [21] says that

*”Mock-based tests for external libraries often end up contorted and messy to get through to the functionality that you want to exercise.”*

When mocking an external API there is no assurance that the test actually works, or as Kolsjö [22] says, it's just a passing test. It will probably work when the test and mocks are written but if the external API changes the mock will be wrong and the test will continue to pass until both the test and mock are updated.

### 3.3.2 Prism

The tool Prism [23] is inspired by the VCR project [24], which is a tool for recording HTTP requests and responses 2.5. Prism is designed with the aim to help and speed up the process of working with frontend development by recording and playing back HTTP requests for testing or development [25].

Prism supports four different modes; proxy, record, mock and mockrecord. Here only record and mock are going to be discussed as they are the primary features of an automatic recording tool.

Prism works by adding a custom built middleware to the existing middleware setup. This enables it to be in-between routing of API requests and adds functionality for handling these requests. A normal use would be to set the tool in record mode, see figure 3.4, and let all HTTP requests pass and on the requests return save the response as a JSON file.

When the recorded mocks are completed it's possible to enable the middleware Prism to redirect HTTP requests to stored responses with the mode mock, see figure 3.5. This mode will listen to the API request and then try to lookup a stored mocked version of that specific API request. A prerequisite for this is that each request must have been seen in the record mode, otherwise there is no mock to match the request. This makes it possible to remove the backend altogether and in a sense can be seen as an HTTP cache.

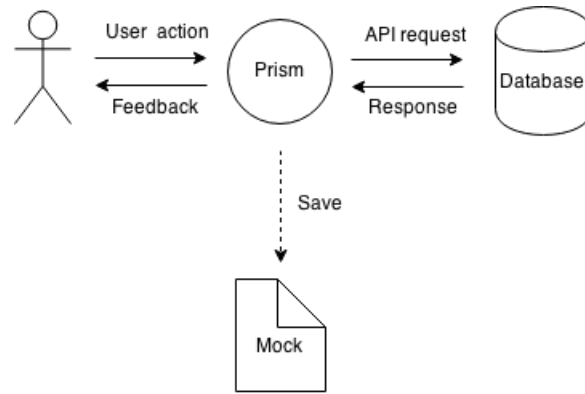


FIGURE 3.4: Prism mode records information flow

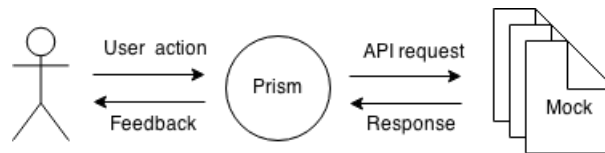


FIGURE 3.5: Prism mode mocks information flow

### 3.3.3 ngMock and ngMockE2E

The modules ngMock [26] and ngMockE2E [27] are modules developed for the framework Angular. NgMock adds support for mocking Angular services and injects them as dependencies to other services. They are created with the aim of supporting unit testing, with the functionality that they can be inspected and controlled in a synchronous manner. Mocks in ngMock works by simulating what the actual service would return by injecting the faked service into test suites.

The other module, ngMockE2E, is very similar to ngMock in that it mocks services and can be injected across test suites. The biggest difference is that it's tailored for end-to-end tests where it mocks HTTP requests by returning the expected responses over and over again, like an actual server [28]. However both of these modules requires the developer to manually create each mock response.

## 3.4 Continuous Integration

Continuous Integration (CI) is one of the twelve foundation practices of extreme programming and is a good way for continuously building and testing the code. According to Martin Fowler [29] CI is

*“a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible”.*

CI doesn't have to run tests in order to be used. It can also be used to build code for detecting compiling errors. However it's a common practice to also have tests running on the CI to be able to discover more bugs. The CI runs the tests and gives a report back to the developers so that they can fix eventual bugs.

*“As a result projects with Continuous Integration tend to have dramatically less bugs, both in production and in process” [29].*

Except for the increased chance of finding bugs, using a CI server allows the developers to integrate their code every day. This enables them to detect integration errors every time they have changed or built something. This frequent integration will save time because developers spend less time integrating their code later on in the project. An overview of how continuous integration works can be seen in figure 3.6.

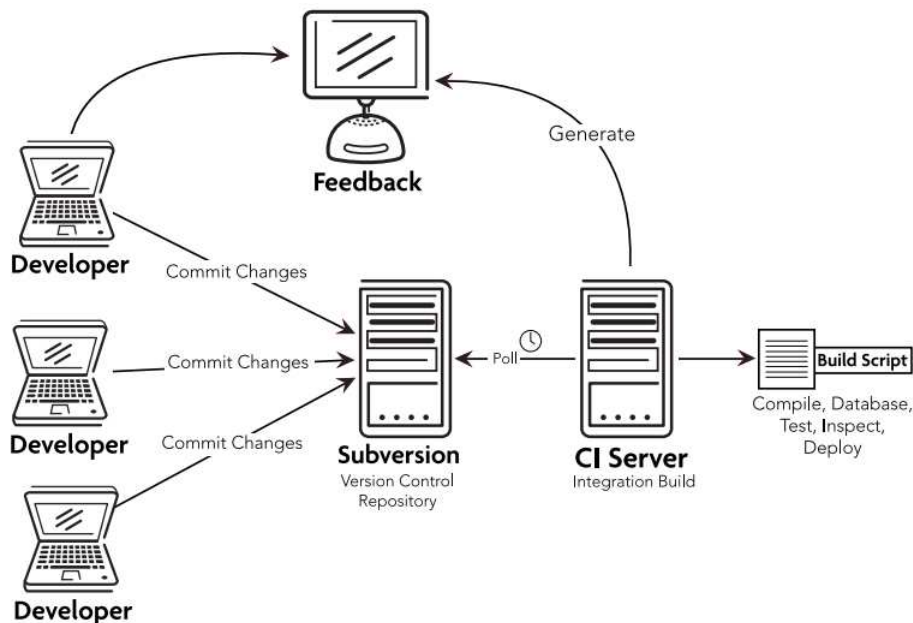


FIGURE 3.6: Overview structure of how continuous integration works, image from [3]

By connecting a version control<sup>1</sup> repository<sup>2</sup> to the CI server, it will get notified by the repository if there are any new commits<sup>3</sup>. The CI server will then sync the repository

<sup>1</sup>Version control: Is a system for tracking changes on files over time

<sup>2</sup>Repository: Is a project in version control system

<sup>3</sup>Commit: Is making changes permanent in version control system

and run a chain of commands, configured by the a user. This is done to set up a copy of the deploy environment and run tests to detect as many errors or bugs as possible before the code is deployed<sup>4</sup>.

## 3.5 Coverage

Code coverage is a tool for assisting in the assurance of well-tested code. By looking at what code gets executed during the tests runtime a measurement of how much the tests are covering can be retrieved. A program is not reliable if just all test are passing as it might just touch on a portion of the code [30].

The purpose of code coverage is not to give a definitive statement if a program is finished with testing or not. It merely gives an indication of the state of the codebase, extra low coverage percentage means that a low amount of code have been tested. Coverage can also help in tracking the state of an application over time as the percentage gives an indication over the amount of tested parts. More importantly it gives a tool to narrow down areas or parts of the code that dont get attention and are untested.

Building and using a coverage tool consists of three parts: code instrumentation, data gathering and coverage analysis [31].

### 3.5.1 Code Instrumentation

Instrumenting a program give the possibility of being able to monitor a piece of software. The process can be utilized to record performance, speed or error tracing in various ways. These recorded features can then be used in a context outside of the software to give an indication of its health, with different metrics.

Instrumentation of a piece of code will listen to what is being triggered and send that data to a data hook<sup>5</sup> for further processing. This practice will not detect any information on parts the program never visited.

### 3.5.2 Data Gathering

Data gathering, also known as data collection, is the process of retrieving and measuring data from a test. In relation to code coverage its goal is to listen on the outputs of the code instrumentation and then build it into a usable structure. This gathered data will then be the basis for answering the proposed questions in the tests and to decide whether a hypothesis was successful. The gathered data for testing software would be the information needed to answer if parts of the code are tested or not.

---

<sup>4</sup>Deploy: Is a process for releasing a certain version of a software

<sup>5</sup>Hook: It allows other modules or parts to react on changes

### 3.5.3 Coverage Analysis

With the provided data from a data gathering process a coverage analysis will perform strategies to decide a softwares health and to give recommendations on areas that need attention. Coverage analysis will analyze the given data and decide what items, objects or functions that have a poor or good status.

## 3.6 Complexity Factors

Complexity factors are formulas to measure a perceived difficulty level of code. A rough approach is to count the number of lines in the source code which then could express a measure of time required to look through the program. Different complexity factors give different hints on where attention should be focused. These complexity factors is used to compare manual versus automatic mocks.

### 3.6.1 Logical Lines of Code

Logical Lines of Code (LOC) is a physical count of logical statements in a program. This performance measure is often seen as a relatively poor one as the most compact solution isn't always the best one, making a pure count misleading.

### 3.6.2 Parameter Count

Parameter count looks at the number of parameters that go into a function and is a way of measuring dependency injection. However this measure doesn't account for dependency within a parameter, making for instance a configuration object count as only one. A low score is desirable.

### 3.6.3 Cyclomatic Complexity

Cyclomatic complexity is a quantitative measure of linearly independent paths through a program. In 1976 Thomas J. McCabe developing this metric by evaluating paths and nodes in a control flow graph. A low score is desirable.

$$\text{Cyclomatic complexity} = E - N + 2P \quad (3.1)$$

Where  $E$  is the number of edges in the flow graph,  $N$  is the number of nodes in the flow graph and  $P$  is the number of connected components.



### 3.6.4 Cyclomatic Complexity Density

Cyclomatic complexity density is an extension of cyclomatic complexity where a percentage score is retrieved by comparing to logical lines of code. A low score is desirable.

$$\text{Cyclomatic complexity density} = \text{cyclomatic complexity}/\text{LOC} \quad (3.2)$$

### 3.6.5 Halstead Complexity Measures

Halstead complexity measures were developed in 1977 by Maurice Howard Halstead and aims at measuring relationship between static code and results. Difficulty refers to the difficulty to actually write, extend or understand the program. Effort is a combined measure of volume, size of the program, and the difficulty that can be viewed as a mass of work. Effort can be estimated to actual time to program and can also give an estimate of expected bugs to detect.

The variables used in some of the following equations are:  $n_1$  = the number of distinct operators,  $n_2$  = the number of distinct operands,  $N_1$  = the total number of operators and  $N_2$  = the total number of operands

$$\text{Program vocabulary: } n = n_1 + n_2 \quad (3.3)$$

$$\text{Program length: } N = N_1 + N_2 \quad (3.4)$$

$$\text{Calculated program length: } N_c = n_1 * \log_2 n_2 + n_2 * \log_2 n_1 \quad (3.5)$$

$$\text{Volume: } V = N * \log_2 n \quad (3.6)$$

$$\text{Difficulty: } D = n_1/2 * N_2/n_2 \quad (3.7)$$

$$\text{Effort: } E = D * V \quad (3.8)$$

$$\text{Time required to program (s): } T = E/18 \quad (3.9)$$

$$\text{Delivered bugs: } B = E^{2/3}/3000 \quad (3.10)$$

$$\text{Program level: } L = 1/D \quad (3.11)$$

### 3.6.6 Maintainability Index

Maintainability index (MI) is a metric to evaluate the entire module or program and was designed by Paul Oman and Jack Hagemester in 1991. Three other metrics; effort 3.8, cyclomatic complexity 3.1 and LOC are used in a weighted average to give the entire program a score of how maintainable it is. A higher score is better where 171 is maximum obtainable.

$$MI = 171 - 3.42 \ln \bar{E} - 0.23 \ln \bar{C} - 16.2 \ln \bar{L} \quad (3.12)$$

Where  $\bar{E}$  is mean effort,  $\bar{C}$  is mean cyclomatic complexity and  $\bar{L}$  is mean logic lines of code.

# Chapter 4

## Theory

This chapter covers information about the theories developed in this thesis from the related work. It includes a concept for a test case design guide, usage of mocks, structure of a coverage tool and potential usage of user session data.

### 4.1 Test Case Design Guide

The test case template Dr. Assassa [18] has written is missing information that is important to know when writing a test case for an end-to-end test. Assassas template covers most things that are generally good to know when writing a test. These are usually conditions necessary for setup, performed actions during tests with their expected response and conditions present after a completed test. With end-to-end tests it's important to know which states the tests are visiting. This is important to know where the elements are interacted with and because some HTTP requests are page specific. This means that if a test is failing because of the mocks or that the test is rewritten, it is easy to know what test cases to rerun for recording the mocks that needs to be rerecorded. Another thing that is good to know when writing a test is what method(s) to use, therefore this is also added to the test case design guide.

### 4.2 Mocks

Mocks presented in 3.3 mention the definition of what mocks aim to achieve. Here the related works are continued to present this thesis theory of how to utilize mocks.

#### 4.2.1 Ways of Rigging Data for Tests

There are different ways that the data could be rigged to run the tests. These options are to use an active database, populate a database with data or to mock the database,

either manually or automatically. The following subsections explain the different methods in more detail.

### Active Database

One form of end-to-end testing is called acceptance test and it's a way of testing all components in a system without any faked responses. This is done to ensure that each component actually works like it should in the live version. These tests cost more in terms of time and effort in setup and execution but will generate a result that matches the expected real user interactions more closely.

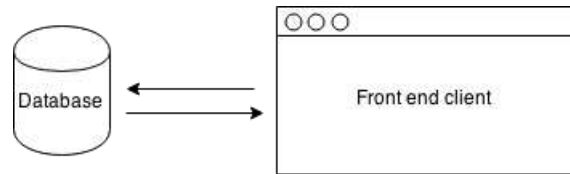


FIGURE 4.1: Example flow over active database setup for testing

It's performed by setting up a frontend client and a backend server, see 4.1, running live for the test suites. Each test on the platform is then run on the actual user interface and interactions are sent to the backend server over the API. A benefit from this solution is that test backend and frontend are tested simultaneously. There are however some drawbacks which tends to make integration testing less useful. Setting up an environment that holds and supports both the backend and frontend takes more time and effort than two separated environments. Usually, or hopefully, the backend and frontend are two very isolated areas and tend to have little to none shared codebase.

### Populate Database

This method is similar to active database setup where both a server and a frontend client is created. Working with a live backend server can create problems when manipulating data since the data is manipulated between testing cycles. Running a test suite that manipulates data will then create a different state after the test suite, making reruns impractical. To compensate for this problem preselected information is generated before building tests.

This can be done either manually or automatically, where dumping data from a live database is a common practice. When the data in the database is at a good state it would be retrieved and saved by a so called data dump. Next time tests are started, a new clean database is set up and populated by the dumped data, see 4.2. This enables tests to be run multiple times as the data before executing tests suites is static.

### Manual Mock Creation

Manual mock creation is most commonly used in unit testing but still contribute to other areas such as end-to-end testing. This can be done in two different ways; by mocking

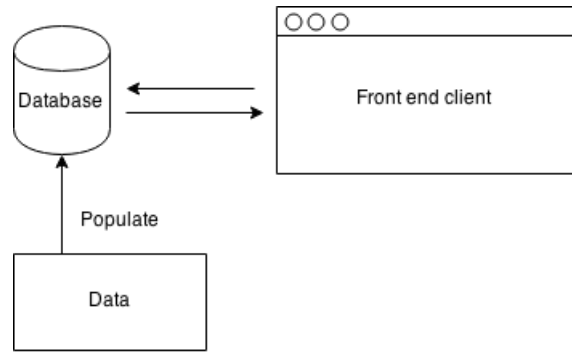


FIGURE 4.2: Example flow over populating a database for tests

the actual database architecture and its corresponding tables, or by mocking responses a user triggers in the product. For instance a test suite that tests a login page that responds with an authentication token can be mocked into a static file.

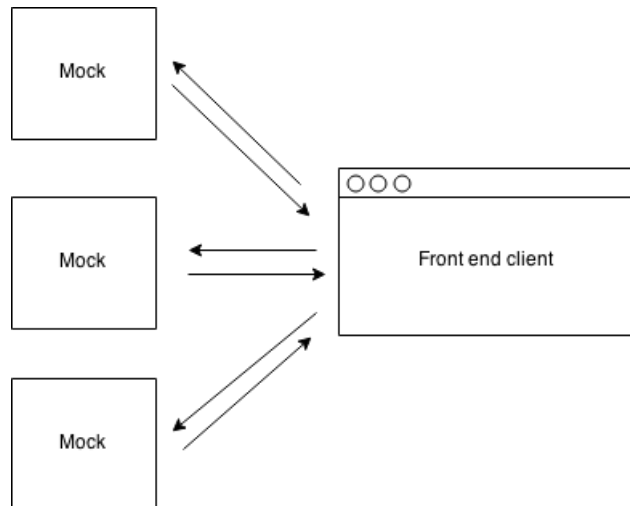


FIGURE 4.3: Example flow over tests with manually created mocks

Using mocks for tests decouples the backend completely and isolates the frontend client for tests. This gives a better performance and narrows down occurring errors into a more controlled environment. It's also one of the more common methods since it's relatively easy to build and yields a higher return than many of its counterparts. Problems associated with mocking is a higher demand on initial time investment, see 4.3, for building the mocks. Services with many dependency injections might require additional work to mock all involved parts. Keeping these mocks up to date, see 2.2.1, is not a straightforward task and can take extra work.

### Automatic Mock Creation

One way of creating mocks by automation is to record HTTP responses from HTTP requests. The tool that is running the mock creation will listen to each HTTP request

and save its header and response information. With this data it's now possible to playback what was recorded next time it's called.

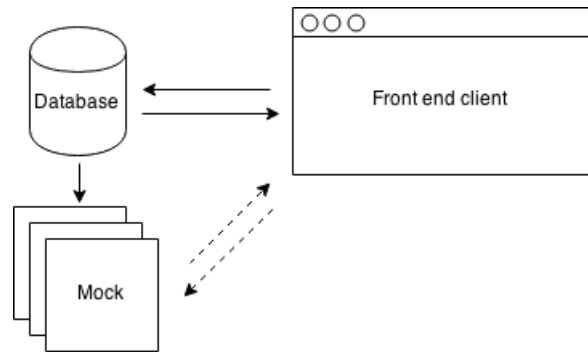


FIGURE 4.4: Example flow over automatically created mocks

With the tool up and running all tests are performed on a live setup. All requests are saved into mocks for later reuse. So the next time tests are running it will first check if it has a recorded version and then sends that one back, see 4.4. A recorded test suite can then be run without any backend at all and makes it easier to maintain on a staging environment. To add new tests a developer simply runs the tests locally which will build the new unseen data.

A positive aspect of this approach is the extensive saving of time to build mocks for test suites. It also matches the real image of responses very well since it records actual responses. It can be harder to maintain since it's an automatic solution where a recorded suite is not intended to be updated manually.

## 4.2.2 Mocking Tool

Prism 3.3.2 is used to build mocks for end-to-end testing by the *automatic mock creation* theory. Here the tool is extended to better suite the needs of current testing requirements since the identification of unique mocks where insufficient in the original tool.

Building mocks is done through an automatic recording tool that is capturing HTTP requests so it can be saved for later use. It's automatic since it injects itself as a middleware in the existing environment and listens, records and replays HTTP requests and responses. These features eliminates the need for manual upkeep of mocks and reduces the required time to manage mocks. A smaller drawback is the loss of control in the details inside a mock request or response. For instance if a request is dependent on a timestamp in its payload it would be hard to use it with an automatic tool.

Figure 4.5 illustrates the current setup of middlewares for relaying information to the right targets at Competencer. Currently three ports are used to relay information and calls to the frontend, backend and a realtime component. In this setup it's the backend that is going to be eliminated in testing so then Prism is attached onto that target port,

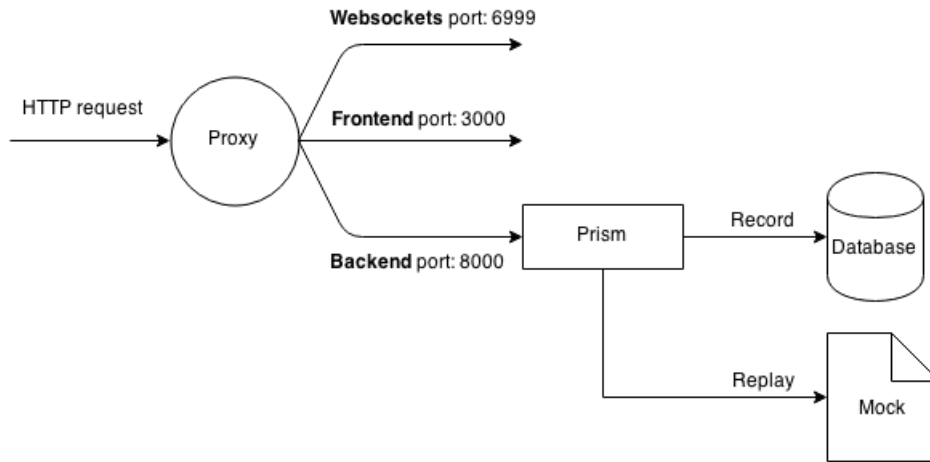


FIGURE 4.5: Overview structure of using prism as a middleware

8000. Prism takes control if it should relay information forward to the backend or if it should get information from mocks.

### 4.2.3 Unique Responses

When working with an *automatic mock creation* tool, Prism, it's important to distinguish each request as a unique one. Otherwise problems might arise with mocks being writing over each other or new ones not being detected.

A mocking tool needs to be able to distinguish between a good API response and a bad one, HTTP status 200 and 400 respectively. Both statuses can be produced on the same endpoint and is therefore not enough to just look at an API endpoint. This is where the current version of Prism is falling short and why an extension was necessary.

#### Proposed structure for unique endpoints

API endpoint + Payload + Headers => Unique response

Given an endpoint, its headers and a payload it should produce an unique response. The reason behind this statement is that this is the exact information that the backend receives, which it uses to perform a specific operation. Therefore it should be enough to return a matching level of granularity in the response based on the same parameters.

## 4.3 Coverage

Section 3.5 describes the fundamentals of using coverage over a code base. The major difference between standard practice methods for unit testing and end-to-end testing is the access to source code. A test in end-to-end testing will not be able to give any

indication whether a certain part or statement have been evaluated or tested from the source code. Testing coverage of end-to-end tests is here defined as a measurement of all elements that are possible to interact with.

The goal here is to locate what elements a user can interact with and then save which one the user does interact with. This will create a coverage report that maps the areas covered of possible user interactions, something that is closely related to the goal of end-to-end testing. A form of state based approach is taken for building coverage over end-to-end tests where the states are triggered by user interactions. Coverage will gather information based on the visual interface presented to the user and data analyzing will be done on the HTML.

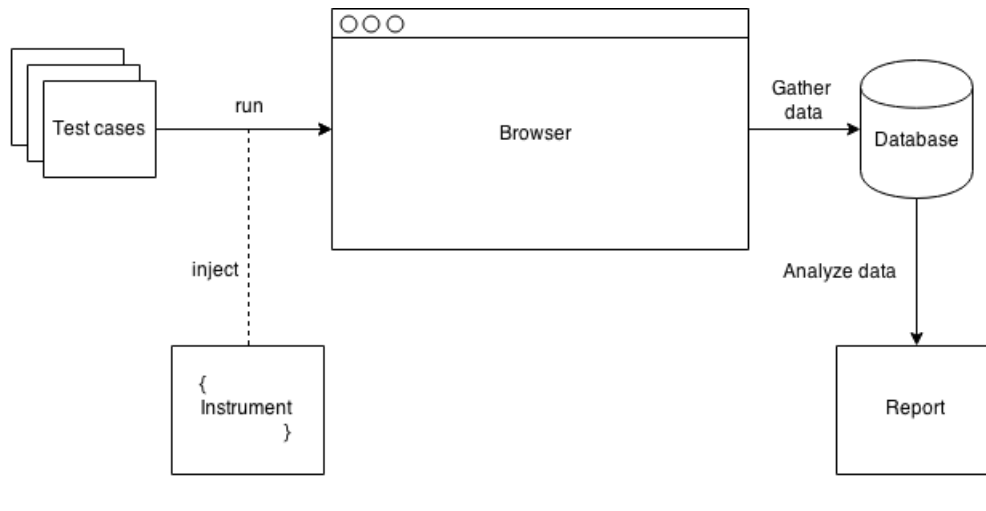


FIGURE 4.6: Overview structure of data flow for coverage

Figure 4.6 displays the structure of the tool and the flow of information for building coverage over an end-to-end solution. For each test case that is executed the browser is instrumented and then a process for gathering data from the browser is performed. This data is saved into a database and upon completion of all the test cases it's analyzed for presentation in a report.

The browser is created and hosted by Selenium and all interactions to and from the browser are going through it. Protractor is used as the bridge between the frameworks used for the web application and the web driver host Selenium. Protractor supports hooks to perform operations on test setup, test case completion and test teardown. With these hooks a Node plugin is built for Protractor to instrument and analyze the tests.

### 4.3.1 Elements

To build a coverage report for end-to-end tests first elements and events that should be used for the report needs to be defined.



Detecting events on visual elements in the browser is done with help from the documentation on HTML. HTML is a global standard [32] and all elements and their attached events are already defined. From [5] all elements that are possible to interact with can be determined, as well as their attached events [6]. In this process a selection is made to not get an overflow of information for the coverage analysis. The selection is made to include elements used for navigation and other elements that are primarily interacted with. Only events that are triggered by a user here are taken into consideration.

Element	Description
<button>	represents a clickable button.
<form>	represents a document section that contains interactive controls to submit information to a web server.
<input>	is used to create interactive controls for web-based forms in order to accept data from the user.
<select>	represents a control that presents a menu of options.
<textarea>	represents a multi-line plain-text editing control.
<a>	defines a hyperlink

TABLE 4.1: Chosen HTML elements for coverage, from [5]

Element	Input	Click	Invalid	Focus	Blur	Change	Submit
<button>		X					
<form>							X
<input>	X	X	X	X	X	X	
<select>		X				X	
<textarea>	X	X		X	X	X	
<a>		X		X	X		

TABLE 4.2: Chosen events for HTML elements, from [6]

In table 4.1 all elements used for coverage are displayed and in table 4.2 chosen related events are presented.

### 4.3.2 Global and Local State URL's

Working with HTML elements instead of source code introduces a few differences. For instance a web page appears in multiple states when the actual HTML markup visible to the user can be built on demand. Source code in contrast is often built beforehand or visited upon runtime. This creates a problem where for instance a button can appear on multiple pages. That button in itself is essentially the same one over multiple pages even though it could have different actions tied to it.

A concept of local and global elements is therefore introduced. The definition of **local** is related to the state URL of the current webpage. **Global** refers to all states encapsulated under a webdomain, e.g. www.competencer.com.

#### Example of state URLs

- /
- /login
- /account/profile

An element can have multiple coverage metrics since it can refer to its local state as well as its global state. If it's interacted on half of its available events on a state named */login* it has a 50% local coverage. The global coverage for that element can't be lower but it could be higher if other events for that element have been triggered on other states.

This concept works for an element on a specific state as well as on the aggregated state itself. A single state have a local coverage and a global coverage since other states can have shared elements that are tested.

### 4.3.3 Test Adequacy Criteria

For a coverage report to produce results that are usable it's important to define when and how a test is completed and if it passes. Since normal practices for testing can't be applied a custom criteria will be developed to better suit the environment.

The test adequacy criteria for this coverage tool is on two levels; events and elements. The criteria for these levels are based on that an event is only triggered when a user makes an interaction. A visual element is the starting point for triggering events and the definition of accessible actions for a user.

#### Criteria for a passed event and a passed element

1. An event is considered a pass when an interaction has triggered it
2. An element is considered a pass when **one** of its own events have passed

## 4.4 User Session Data

Elbaum et al. [33] writes about how user session data can be used to create test suites for unit tests in a more effective way. This could also transfer to end-to-end test if data from user activity on the web page were to be collected. The data about what elements are interacted with could be used to cover the most used parts of the web page with tests. This could be combined with the coverage report to see what elements to prioritize in the test cases to be written.

The data that should be withdrawn from user sessions to get a good representation of what to test next could be things like:

- Number of interactions with an element

- What type of element
- What type of interactions with the element (hover, click, etc.)
- How many times pages are visited
- What pages are visited
- Time spent on pages
- Time spent waiting on action

The coverage shows what is already tested and what is not. This together with the data from users browsing the web page in the list above could be used to generate a list of recommendations of user heavy elements or states that should be tested. This would be done in order to make it easier for testers to decide what to test. Possible ways to decide what to test next is to check what elements have the highest frequency of use or check what pages are visited the longest time. It is the things users use most that needs to be tested.

This recommendation list could give examples like *This element have a high priority* or *Users spend a long time on page X*. A test case from this recommendation list could easily be created and it would save time for the tester to figure out what to test next. It would be like having an automatically generated backlog for the tester(s). There are a lot of possibilities with a method like this.

# Chapter 5

## Implementation

This chapter covers how implementations of the various parts of this thesis is carried out. The implementations are explained with figures and code snippets to make it easier to understand and replicate.

### 5.1 Setup and Execution

To write tests with Protractor, Mocha and Chai a Protractor configuration file needs to be written. In this configuration file a path to the Selenium server is defined, what framework to use and what tests to run are also included. This configuration file is included in appendix B.

When writing tests it's a good practice to use page objects. This means creating an object for a certain page that will have variables tied to the elements on that page. It can also contain functions such as a login function. This makes the code reusable rather than inlining it in the tests. An example of a page object can be seen in the snippet below:

```
1 var LoginPage = function() {
2   this.usernameField = element(by.model('auth.user.username'));
3   this.passwordField = element(by.model('auth.user.password'));
4   this.loginButton = element.all(by.tagName('form')).get(0).element(
      by.tagName('button'));
5   this.errorMessageBox = element(by.css('[ng-show="auth.error"]'));
6   this.get = function() {
7     browser.get(browser.baseUrl + '/login');
8   };
9   this.login = function(username, password) {
10    this.usernameField.sendKeys(username);
11    this.passwordField.sendKeys(password);
12    this.loginButton.click();
13  };
14 };
15 module.exports = LoginPage;
```

The above snippet shows how a page object for a login page could look like. It includes variables for the username field, password field, login button, error message box, login function and a function that navigates to the login page.

A testfile using the page object could begin like the snippet below:

```
1  'use strict';
2  // Import chai framework
3  var chai = require('chai');
4  var chaiAsPromised = require('chai-as-promised');
5  chai.use(chaiAsPromised);
6  var expect = chai.expect;
7  // Import login page object
8  var LoginPage = require('../pages/login.page.js');
9  var loginPage = new LoginPage();
10
11 describe('Login test', function(){
12   it('Should go to login page', function(){
13     loginPage.get();
14     expect(browser.getCurrentUrl()).to.eventually.equal(
15       browser.baseUrl + '/login');
16   });
17   it('Should be a username field', function(){
18     expect(loginPage.usernameField.isPresent()).to.eventually.equal(
19       true);
20   });
21   ...
22 });
```

---

First the Chai assertion library is imported to enable expectations that checks if a test passes or fails. The *chai-as-promised* adds promises to the Chai library, which enables the tests to wait for a promise. If not the test would be faster than the page is loading and all tests would fail.

The *describe* function is used to divide parts of tests and to get a more granular report. The *it* functions is what is evaluated to true or false. In the code snippet above there are two tests, first tests if the browser is on the correct page and the seconds tests if the username input field is present.

After the tests have been written it can be executed with the following bash command, assuming current folder contains the *e2e.conf.js* file, see [B](#).

```
protractor e2e.conf.js --suite logintest
```

To prevent the need to start up the frontend server which needs to be done before running tests like described in section [5.1](#), a gulp task is created. What this task simply does is to start the frontend server and then run the tests with only one command needed. The gulp task for running all tests looks like the snippet below.

---

```

1  'use strict';
2  var gulp = require('gulp');
3  var $ = require('gulp-load-plugins')();
4  var connect = require('gulp-connect');
5  var browserSync = require('browser-sync');
6  gulp.task('protractor-only', function (done) {
7    var testFiles = [
8      'test/e2e/**/*.js'
9    ];
10   gulp.src(testFiles)
11     .pipe($.protractor.protractor({
12       configFile: 'test/e2e.conf.js'
13     }))
14     .on('error', function (err) {
15       // Make sure failed tests cause gulp to exit non-zero
16       throw err;
17     })
18     .on('end', function () {
19       connect.serverClose();
20       done();
21     });
22 });
23 gulp.task('protractor', ['serve', 'protractor-only']);

```

---

Now the only bash command needed to run all tests is the following:

```
gulp protractor
```

There are also tasks created to for running tests with mockrecord and for running tests without backend against the recorded mocks.

## 5.2 Test Case Methods

In this section the implementation of the three methods DT, STA and BVA are shown with code and figures. The full implementations are not presented, only the vital parts are shown to understand how they would be implemented. The following subsections shows an interpretation of how these methods could be implemented as there is no exact way.

### 5.2.1 Decision Table

The first case where both username and password are invalid, from table 3.1, is implemented in the following way:

```

1  describe('Logintest, N = Not Valid, Y = Valid', function() {
2    it('Username: N, password: N', function() {

```

```

3     loginPage.usernameField.clear();
4     loginPage.passwordField.clear();
5     loginPage.login(wrongUsername, wrongPassword);
6     expect(loginPage.errorMessageBox.getText()).
      to.eventually.not.equal("");
7   });
8
9   ...
10  });

```

First the username and password fields are cleared, then an attempt to login with wrong username and password is made. After the login attempt, an error message box containing an error message should not be empty.

## 5.2.2 State Transition Analysis

A login STA diagram is shown in 5.1 and the following code snippet is an implementation of the first two states shown.

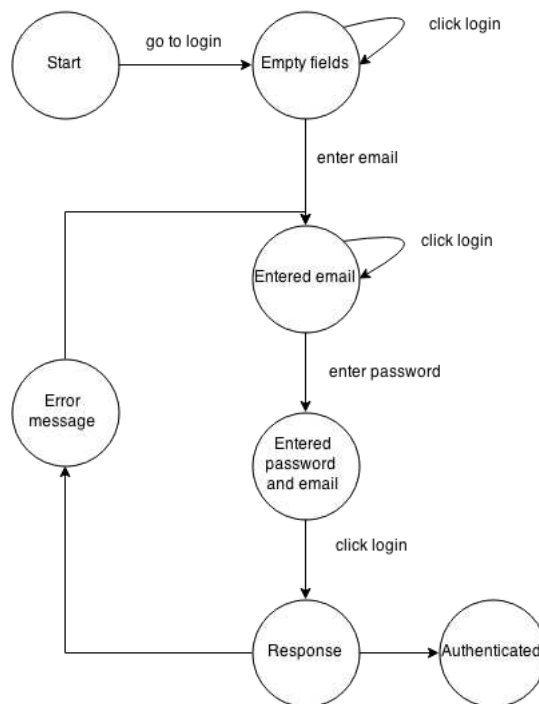


FIGURE 5.1: State transition analysis example for a login flow

```

1 describe('Logintest, A = Action, S = State', function() {
2   it('A: Go to login, S: S0 -> Empty fields', function() {
3     loginPage.get();
4     expect(browser.getCurrentUrl()).to.eventually.equal(
      browser.baseUrl + '/login');
5     expect(loginPage.usernameField.getAttribute('value')).
      to.eventually.be.empty;

```

```
6     expect(loginPage.passwordField.getAttribute('value')).
7     to.eventually.be.empty;
8   });
9   it('A: Enter email, S: Empty fields -> Entered email', function() {
10    loginPage.usernameField.sendKeys(wrongUsername);
11    expect(browser.getCurrentUrl()).to.eventually.equal(
12    browser.baseUrl + '/login');
13    expect(loginPage.usernameField.getAttribute('value')).
14    to.not.eventually.be.empty;
15  });
16  ...
17  });
```

---

In the snippet above the first state gets the login page and checks that the browser is on the correct page. After that it also checks that the input fields are empty. In the state after that it inputs a wrong username and checks that it's still on the login page and that the field is not empty.

### 5.2.3 Boundary Value Analysis

The BVA method, from figure 3.2, is implemented in the following way:

```
1  var tryDuration = function(duration, shouldWork){
2    describe('Trying duration:' + duration, function() {
3      editProfile.offerings.create.duration.sendKeys(duration);
4      if(shouldWork) {
5        it('Should not be an error', function() {
6          expect(editProfile.offerings.create.error.getText()).
7          to.eventually.equal("");
8        });
9      }
10     else {
11       it('Should be an error', function() {
12         expect(editProfile.offerings.create.error.getText()).
13         to.eventually.not.equal("");
14       });
15     }
16   });
17 }
18 describe('Boundry value analysis duration test', function(){
19   tryDuration(14, false);
20   tryDuration(15, true);
21   tryDuration(16, true);
22   tryDuration(299, true);
23   tryDuration(300, true);
24   tryDuration(301, false);
25 });
```

---

In the snippet above the *tryDuration* function tries the passed duration and then checks if and if not there is an error message.



## 5.3 Test Case Design Guide

The test case design guide, see appendix E, is created to help testers keep track of what is tested, how it's tested and how to build a test. The guide has fields for name of the test and a short description of what the test intends to test. Furthermore, the guide has a field for dependencies, this is the field where test cases that needs to be run before the test case that is being created are listed. The visited pages field is to keep track of what pages different tests are going through, this can be used to optimize the testflow. Tests can have preconditions, for example a user that has to be logged in to be able to perform the test. In the methods field the methods to be used should be listed, these are chosen from BVA, STA, DT or ad hoc <sup>1</sup> where the ad hoc method is a custom made test. The test table is there to fill in everything that is being tested. The post-condition field contains everything that has been changed after the test is completed. These guidelines will help keep the testing structured and make it easier to create and build a test.

## 5.4 Building Mocks

This section presents different implementations for the tools Prism and ngMockE2E. The included code snippets are there to help understand the structure and to ease reproducing the approach.

### 5.4.1 Automatic Mocks with Prism

Building mocks with Prism, 3.3.2, requires focus on the setup rather than the individual mocks. Since the mocks are built automatically the investment into a single mock is negligible. It's the surrounding setup for a single mock that is created demands time and thought.

```
1 function setupPrism(mode, name) {
2     prism.create({
3         name: name,
4         context: '/api',
5         host: 'localhost',
6         mode: mode,
7         mocksPath: './test/mocks',
8         port: targets.backend.port,
9         hashFullRequest: true,
10        mockFilenameGenerator: customMockFilename,
11        clearOnStart: false
12    });
13 }
```

---

<sup>1</sup>Ad Hoc: Is a method that doesn't use either planning or structure

Above snippet shows a setup of Prism that takes a name and a mode as inputs. The tool is targeted onto the backend port which is the area that is targeted for mocks. The mocking context for Prism is the endpoint `/api` and the rest is ignored.

```

1  function connectInit(baseDir, args) {
2      setupPrism(args.mode, args.name);
3
4      connect.server({
5          root: baseDir,
6          port: targets.frontend.port,
7          livereload: false,
8          middleware: function(connect, opt) {
9              return [
10                 prism.middleware,
11                 proxyMiddleware,
12                 modRewrite([
13                     '!\\.\\.\\w+$ /index.html [L]',
14                 ])
15             ]
16         }
17     });
18 }

```

---

Function `connectInit` contains the setup for using a frontend server on localhost<sup>2</sup>. It's targeted against the frontend port and has a middleware functionality to redirect requests. Prism is instantiated by `setupPrism` and is the first in a series of middlewares. Prism will act as a proxy if the context is wrong or if the mode is not set to mock requests.

### 5.4.2 Manual Mocks with ngMockE2E

The frameworks ngMock and ngMockE2E both rely on the angular service `$httpBackend` [34]. The main difference is that ngMock is designed for unit tests and ngMockE2E is designed for end-to-end tests. Unit testing only requires the response once but end-to-end testing needs to keep getting the response continuously [35], for example an authentication check can happen multiple times. NgMockE2E enables this feature which makes it good for the end-to-end tests.

The mocks in ngMockE2E are created in an Angular module called `run` where the `$httpBackend` is passed. This module is placed inside a function that is exported, so it then can be required in a test file.

```

1  var httpBackendMock = function() {
2      angular.module('httpBackendMock', ['ngMockE2E'])
3      .run(function($httpBackend) {
4          //Mocks here
5      })
6  };

```

---

<sup>2</sup>Localhost: It's a hostname that refers to the current machine

```
7 module.exports = httpBackendMock;
8
9 // importing the module
10 var MockedBackend = require('../utils/mockedBackend.js');
```

---

Through the *MockedBackend* it's now possible to add the mock module to Protractor which runs it with the mocks. This is done inside a test and can be viewed below.

```
1 describe('A test', function() {
2
3   //Add the mock
4   browser.addMockModule('httpBackendMock', MockedBackend);
5
6   it('Should test something', function() {
7     expect(something.getText()).to.eventually.equal("something");
8   });
9 });
```

---

If a test requires a response from a backend to pass it's now possible to mock that request. The request that the test requires needs to be done manually so that the response can be stored in a JavaScript object. This variable will then be used in the return response of the mock, see snippet below.

```
1 var authToken = {
2   token: "26b9fa-SOME-TOKEN-78681ff"
3 };
4
5 var correctLogin = {
6   username: 'theUsername',
7   password: 'thePassword'
8 };
9
10 $httpBackend.whenPOST('/api-token-auth/', correctLogin)
11   .respond(function(method, url, data, headers) {
12     return [200, authToken, {}];
13 });
```

---

When a test makes a POST request to the url */api-token-auth/* with the *correctLogin* details, it's suppose to return 200 and an authentication token. This backend call is now stored in a mock, so once the request goes out the method *whenPOST* catches it and returns 200 and the token that is expected. It's also possible to choose if some requests are going to go through to the real backend, this is helpful when the mocks are under development.

## 5.5 Building Mock Tool

This section contains the setup for the backend to run a mocking tool as well as the extensions made to the tool Prism, such as an url counter and a mock file name generator.

### 5.5.1 Empty Database

With mocks in end-to-end testing it's a requirement to have knowledge about the initial state the data is in when running tests. This is necessary by the fact that an end-to-end test can modify information during testing. Running two tests in a row could break the tests since the first one can change the data.

#### Setup for running an empty database

```
#!/bin/sh
```

```
FRONTEND_TEST=true python manage.py flush --noinput  
FRONTEND_TEST=true python manage.py syncdb --noinput  
FRONTEND_TEST=true python manage.py runserver
```

The approach used is to start all tests from a clean slate since it's easy to grasp. The benefits from this approach is that it's easier to maintain and easier for other people to get started with. Some actions might take longer to test through this practice since some features aren't present in the initial state. Before starting to record mocks a backend server is setup that will first flush all data and then sync the database for migration purposes.

### 5.5.2 Url Counter

Something that rose during testing when using stored mocks for replay is that it's a static method. A database is a dynamic organism and will give different responses depending on where in the queue it's called. This became a problem when the previous statement about unique responses held true, but still made tests crash. A unique response might be unique under that timestamp but since a mock is static it does not reflect changes over time.

For instance if a user first updates the profile and then reverts that profile back to its original state, the intermediary state is lost. The endpoint for retrieving that users profile is the same but the content inside is changed over time.

The solution to this was to add an internal tracker of how many times an endpoint had been called. This url count tracker then updates the formula for an unique response.

#### Structure for unique endpoints with url counter

```
API endpoint + UrlCounter + Payload + Headers => Unique response
```

Implementation of the url counter can be found in [appendix A](#).

### 5.5.3 File Name Generator

The recorded response result is stored in a JSON file that later on is used by the mock mode. For this interaction to work seamlessly the storing and finding of files must match identically to the uniqueness of a HTTP response. A custom file name generator is built on the assumptions on what makes a request unique. This generator is used to first store the file in record mode and to then look up files in mock mode.

The following snippet displays the implementation of the file name generator.

```

1  function customMockFilename(config, req, status, count) {
2      var maxLength = 255;
3
4      var fileName = req.url.replace("/\\/|\\_|\\?|\\<|\\>|\\\\\\\\|\\:|\\*|\\\\|\\'\\/g
5      ,'_') + '_' + req.method + '_';
6      var payload = JSON.stringify(req.body);
7      var auth = !req.headers.authorization ? '' :
8      req.headers.authorization;
9
10     var shasum = crypto.createHash('sha1');
11     shasum.update(fileName + payload + auth + count);
12     var hash = shasum.digest('hex');
13
14     return fileName.substring(0, maxLength - hash.length) + hash + '.
15     json';
16 }

```

The file name generator takes the request url, request payload, authorization header and count as parameters for building a hash. The hash is the unique key that maps each request that is entered to the function. Using a hash as the file name is not very usable as navigating between jumbled numbers and characters is far from intuitive. The relation between a seemingly random string and the content it represents is indecipherable. The file name is therefore built to first display what url was requested, then what method it was requested with and at the end append the hash.

#### File name by hash

17464e8ebd541096ce02318baf267f7b0b1607f3.json

#### File name by hash and request

\_api\_users\_1\_set\_password\_POST\_17464e8ebd541096ce02318baf267f7b0b1607f3.json

## 5.6 Coverage

This section presents the implementation of the coverage tool built for end-to-end tests. It's divided into parts of code instrumentation, data gathering and coverage analysis. A coverage tool is a utility for measuring what is covered in the product by the test suites.

### 5.6.1 Code Instrumentation

Code instrumentation, see 3.5.1, for coverage is performed after each test case. It is necessary to perform this operation often since a test can modify content and enable options on a page that would go missed if code instrumentation were not updated. A web application also updates its state, URL, throughout the tests and will display significantly different content over time making it important to continuously update instrumentation. Since a test case can be viewed as a user action the instrumentation will be updated after each one.

#### Browser Injection

The coverage tool need to withdraw data about elements that are interacted with. To do this a script is injected into the browser that queries all elements available, of the types predefined in 4.3.1. These are then attached with event listeners to give an output when they have been called.

Code snippet of the major parts in the browser injection:

```
1 browser.executeScript(function() {
2   ...
3
4   // Elements we want to investigate
5   DOMcomponents.forEach(function(DOMtype) {
6
7     // find all elements of the node type
8     var DOMitems = helper.getNodes(DOMtype.type);
9
10    DOMitems.forEach(function(item) {
11
12      // add one eventlistener for each event
13      DOMtype.events.forEach(function(event) {
14
15        item.addEventListener(event, function() {
16          // output information on triggered event
17          // needs to be info to be caught by the browserlogs
18          console.info('CoverageE2E', event, item.outerHTML,
19            window.location.pathname);
20        });
21      });
22    }
23  });
24 });
25
26 return DOMcomponents;
27 }, self.config.elements).then(function(DOMcomponents) {
28   ...
29 });
```

---

The above code steps through each specified element that is going to be analyzed and gathers them from the browser. The nodes are gathered by querying all elements of that type and then returns an array from the helper function. This element node then gets an event attached for each associated event that is required. When one of these events then has been triggered an output is sent to the browser logs. The output contains information about what event triggered it, information about the element as well as current state it was triggered on. After all elements have been attached with event listeners the browser injection then returns the elements back from the browser to the node module for storage.

This browser injection is running after each test case which can count up to be several thousands, if not more. Appending event listeners over and over would clutter the logs and make browser slow and hard to work with since the amount of event listeners would implode. In order to account for this the session storage in the browser is utilized to keep track of which elements and events that have previously been seen. Figure 5.2 shows the flow of using session storage in the browser.

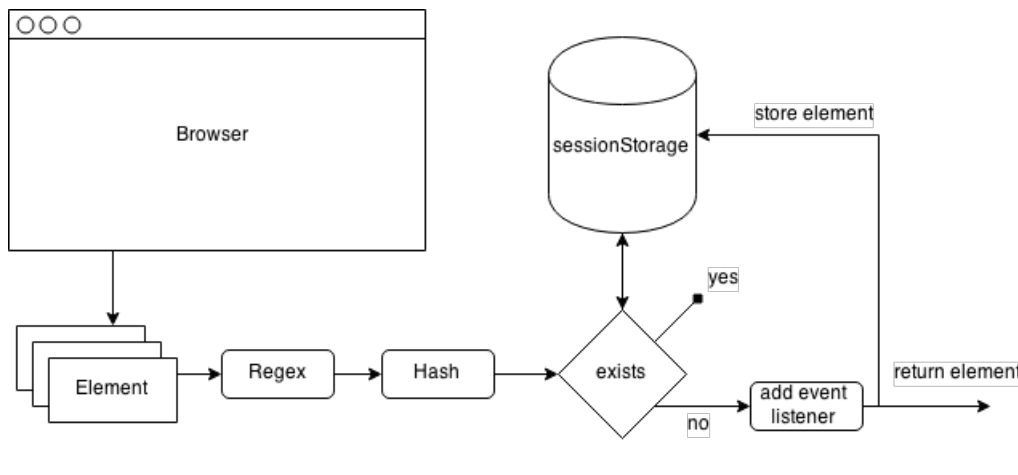


FIGURE 5.2: Storing elements in session storage

Attaching event listeners in conjunction with session storage:

```

1 DOMitems.forEach(function(item) {
2   var hash = helper.hashCode(item.outerHTML + url);
3
4   // check if eventlistener exists
5   if(!window.sessionStorage.getItem(hash)) {
6
7     // if not, add one eventlistener for each event
8     DOMtype.events.forEach(function(event) {
9
10      item.addEventListener(event, function() {
11        // output information on triggered event
12        // needs to be info to be caught by the browserlogs
13        console.info('CoverageE2E', event, item.outerHTML,
14          window.location.pathname);
15      });
16    });
17   }
18 }

```

```

16     });
17
18     // store eventlistener in sessionStorage
19     window.sessionStorage.setItem(hash, 'CoverageE2E');
20 }
21
22 });

```

---

To keep track of elements seen, the outer HTML of an element is hashed to give a comparison reference. This hash is then looked up in current session storage and if undefined the injection will continue to add event listeners. These are then stored with hash in session storage so further interactions won't append new event listeners.

One caveat working with AngularJS as a framework is its use of CSS classes for looking up states of elements. It uses specific classes for specifying if an input for example have been visited or if it has been entered. This makes identifying elements harder as an element, by its outer HTML, would be considered two separate ones after interacting over it. To address for these changes an implementation of stripping out classes is performed before hashing an element.

### Regular Expression for stripping classes

```
/\s\bclass=("[^"]+")/g
```

Helper functions available for hashing and retrieving elements in the browser:

```

1  var helper = {
2    hashCode: function (s) {
3      var clean = helper.cleanElement(s);
4      return clean.split("").reduce(function(a,b){a=((a<<5)-a)+
      b.charCodeAt(0);return a&a},0);
5    },
6    cleanElement: function(s) {
7      // remove html classes for hash
8      // so we don't get duplicate on things like .ng-touched
9      var r = '/\s\bclass=("[^"]+")/g';
10     return s.replace(r, ' ');
11   },
12   getNodes: function(type) {
13     // return NodeList
14     var arr_nodes = document.querySelectorAll(type);
15     // convert to array
16     return Array.prototype.slice.call(arr_nodes);
17   }
18 }

```

---



## 5.6.2 Data Gathering

From the browser injection in code instrumentation the elements that have been seen can be returned back to the node module for storage. For speed improvements all information about the element is built and calculated in the browser before returning it to the module. The reason for it being a question about speed is the interaction with the browser through a browser injection is dependent upon promises. Resolving promises for multiple element would take a significantly longer time and initial testing showed a near triple increase on test suites runtime.

When the elements are returned from the browser it is stored in an object acting as a database:

```
1 CoveragePlugin.prototype.storeElement = function(element, type) {
2   var self = this;
3
4   var hash = self.hash(element.item);
5   var index = _.findIndex(self.DOMElements, {'url': element.location
6     });
7
8   // element structure
9   function buildElement() {
10    return {
11      'hash': hash,
12      'element': element.item,
13      'css': element.css,
14      'type': type,
15      'tested': false,
16      'events': []
17    }
18  }
19
20  // if the url hasn't been seen
21  if(index === -1) {
22    var urlObj = {
23      'url': element.location,
24      'elements': [buildElement()]
25    }
26    self.DOMElements.push(urlObj);
27  }
28  // if the item on the location hasn't been seen
29  else if(_.findIndex(self.DOMElements[index].elements, {'hash': hash
30    }) === -1) {
31    self.DOMElements[index].elements.push(buildElement());
32  }
```

The above snippet shows that the element is here stored with hash as its unique key. This part of data gathering is mainly storing the element itself into a structure with persistent lookup on state and hash. The element stored here hasn't yet been tested or analyzed.

At the end of each completed test the logs from the browser is retrieved and saved. This is done after each test since operations in the browser might clear the content of the logs, such as navigation to a different domain and back, which would resolve in a loss of data.

Storing logs into a stack for later parsing:

```
1 CoveragePlugin.prototype.saveLogs = function(config) {
2   var self = this;
3
4   if(this.browserLogAvailable) {
5     browser.manage().logs().get('browser').then(function(log) {
6       self.logs.push(log);
7     });
8   }
9 };
```

---

After all test suites have completed and the browser environment is completed the data is updated in storage through parsing the logs. Parsing logs includes stepping through the saved outputs from the browser and updating the existing data structure. Each relevant entry in log contain information about the element, where it was seen and what type of event triggered it. This makes it easy to hash the element and update the existing structure to include the event that has been seen.

Parsing logs and updating element:

```
1 CoveragePlugin.prototype.parseLogs = function(config) {
2   var self = this;
3
4   if(this.browserLogAvailable) {
5     this.logs.forEach(function(log) {
6       var warnings = log.filter(function(node) {
7         return (node.level || {}).name === 'WARNING';
8       });
9
10      warnings.forEach(function(elem) {
11        var m = JSON.parse(elem.message);
12        if (m.message.hasOwnProperty('parameters')) {
13
14          var p = m.message.parameters;
15
16          if(p[0].value === self.name) {
17            self.updateElement(p[1].value, p[2].value, p[3].value);
18          }
19        }
20      });
21    });
22  }
23 };
```

---

The information is then saved into a JSON file for further usage and presentation.

### 5.6.3 Coverage Analysis

Last step in building coverage is to analyze the gathered data and make a digestible report. The raw data is available and to make it more usable and extra layer of interpolation is added on top.

Each element is extended with all the events that never was triggered so an element in itself have a representation of all possible events. Then it calculates a coverage percentage over that event coverage. It then proceeds to iterate over states to detect if the same element on an another place have been seen. If it detects another occurrence it compares that elements events with its own to generate a global status over seen events. This then results in a global coverage percentage over events for that element.

Each state calculates how many elements have been tested both locally and global, by the same manor as with events. A summation and calculation is based on type of elements seen in the state.

#### Hierarchy

The report is divided into three parts: overall statistics, summation by type and presentation by state. The overall statistics conclude of the amount of elements seen, how many were tested and that resulting coverage percentage as seen in figure 5.3.



FIGURE 5.3: Overall statistics for end-to-end coverage

Summation by type is a presentation of each type of element seen across the test suites. The total count of the element type is presented along with how many were tested as well as the coverage percentage, see figure 5.4.

The last section is a list of all states that the test suites were run across. The top level of this section is presented in the order the states occurred in the tests. A state is named by the URL and displays information about the amount of elements it saw on that state as well as the local coverage percentage, see figure 5.5.

This last section contains more information as the states contains the elements as well. Drilling down the section it is split into three sub parts. The sub parts, figure 5.6, have the information about local and global coverage percentage for the aggregated elements of that state. It also contains information about the coverage per a type by type basis. The last sub part is a list of all elements found on that list sorted by element type.

135	a	4 Tested	3% Coverage
8	button	2 Tested	25% Coverage
25	form	1 Tested	4% Coverage
42	input	7 Tested	17% Coverage
2	select	0 Tested	0% Coverage
1	textarea	0 Tested	0% Coverage

FIGURE 5.4: Coverage summation on a by type basis

▶ /	48 Elements	2% Coverage
▶ /register	17 Elements	12% Coverage
▶ /login	8 Elements	0% Coverage
▶ /account/profile/	51 Elements	2% Coverage

FIGURE 5.5: Coverage summation on a by state basis

Each element can then be further grained to show more detailed information. An element shows if it was tested, its own hash, its own html markup and a visual representation of that element. It also presents events that have been tested both locally and globally, see 5.7.

This structure is built for easy access into the important information needed to make good actions towards building and updating test cases. First level gives the user a grasp of information about the state of the tests. The second will indicate if it's any particular type of object that is underrepresented during test suites.

Viewing test state by state will quickly narrow down where testing have been insufficient. It quickly can be seen which elements that have been missed and can get an information

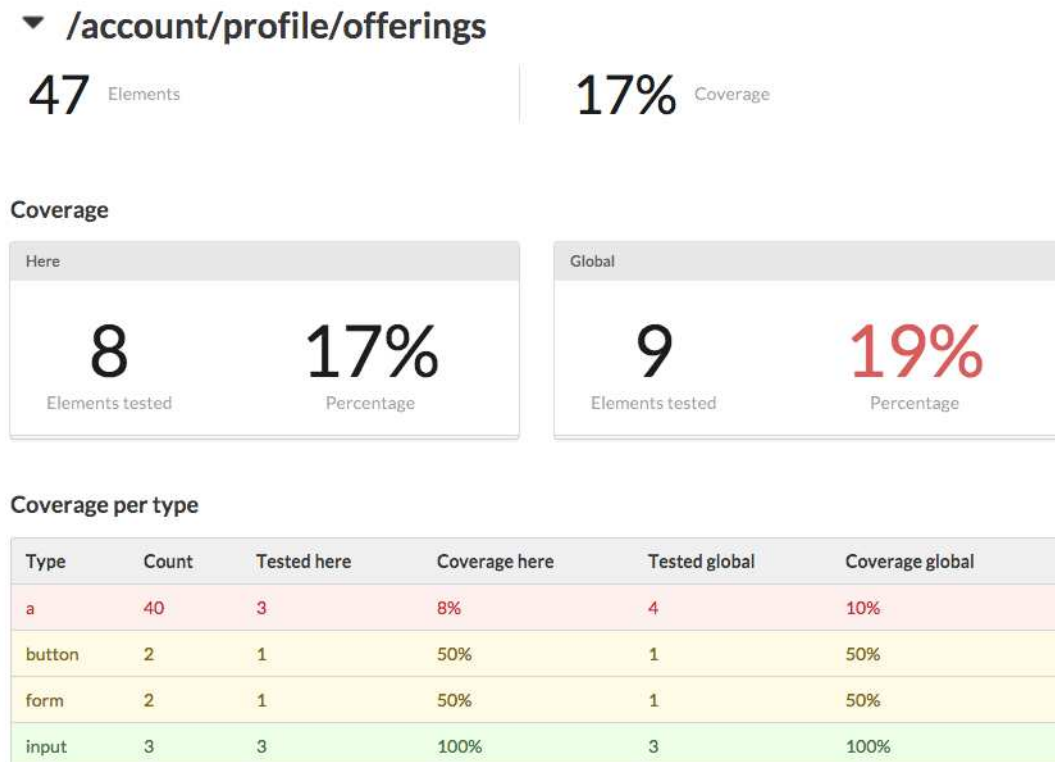


FIGURE 5.6: Coverage details for a single state

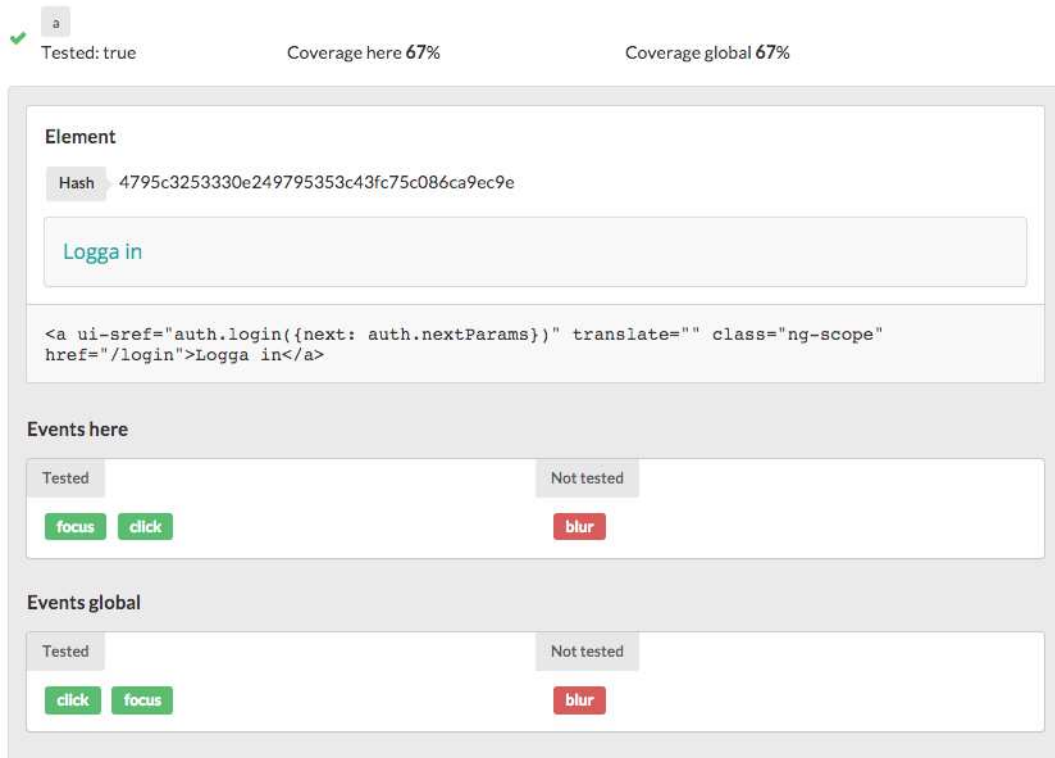


FIGURE 5.7: Coverage information for an element and its events

about markup and its visual representation. This makes for an efficient flow for a user to go back and update tests to target that element, or event.

## Indicators

Some indicators are added to the report to ease the use of it. Percentage counters are given a color to give a direction towards the area where attention is needed. The actual cutoffs for these percentages have no significant grounds for what is considered good or not as they are only there to help point on areas of interests.

Coverage percentage	Color
<25%	Red
>= 25%, <75%	Orange
>= 75%	Green

TABLE 5.1: Color indications based of coverage percentage

Test status	Indicator
Tested	Check mark
Untested	Cross mark

TABLE 5.2: Color indications based of coverage percentage

## Visualizing Elements

To help users use the report in an effective manor a visual element is added into the presentation. The visual element is a close representation of the tested element in the application. It's constructed by compiled all associated CSS classes attached to the element. This works well for single elements but loses representation for nested types such as forms.

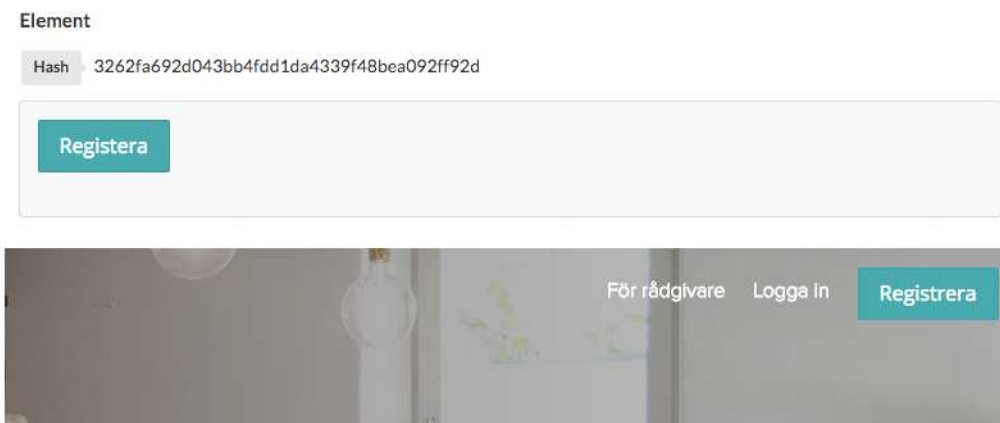


FIGURE 5.8: Comparison of coverage reports visualized element (top) and web page element (bottom)

## 5.7 Continuous Integration

Codship [36] is the chosen CI server because it is free and has an intuitive and easy interface on the web. The first thing to do is to connect the CI server with the Github repository, this is easily done by logging in with Github credentials on Codship. When this is done and Codship is allowed access to the repositories it is possible to choose which repository to run on the CI server. Then the build commands can be set up to run the environment needed for the project. These are the commands that will be run after the CI server gets a new version of the repository. For the environment Competencer uses, the commands seen in appendix C needs to be run. The hashtags are comments.

When this `gulp protractor:mock` command is run everything should be set up for it to run the tests. It will connect to the selenium WebDriver server and run the `protractor:mock` task that will run the test suites against the mocks. For this to work the tests must have been run with the mock tool in record mode first so that the recorded mocks exists in the repository.

# Chapter 6

## Results

This chapter covers the results from this thesis, which includes results from tools, test case methods, continuous integration, mocks and coverage.

### 6.1 Tools

After looking at different testing frameworks to build end-to-end tests for an angular built single page application the preferred and chosen frameworks and tools are:

- Mocha [2.6.3](#) which is a JavaScript framework for writing tests and reporting results
- Protractor [2.6.2](#) which is a JavaScript end-to-end testing framework that is built especially for angular, it makes it easy to get and test elements in an Angular single page application. Protractor also contains Selenium WebDriver [2.6.1](#) which is a server that translates the tests and runs JavaScript in a browser to interact with the page
- Chai [2.6.3](#) which is a JavaScript assertion library that works well together with Mocha and Protractor
- Prism [3.3.2](#) which is a record and playback mocking tool that is used to mock the backend. This tool is chosen because of its record and playback functionality which makes the mocks more maintainable and easier to create than using manually created mocks
- Custom built coverage tool [5.6](#) named Protractor-e2e-Coverage which is built because there were no tools that counted interactable elements during a test to build an end-to-end coverage report
- Codeship which is a continuous integration [3.4](#) server used for automatically running tests when integrating new code



## 6.2 Test Case Methods

A decision table login test, a STA login test and a BVA test for two numerical input fields is compared. The login tests for STA and DT contains an input field for a username and an input field for a password. The BVA test contains two numerical input fields, one for duration and one for price. This means that the three compared tests all contain two input fields but the BVA test is more suited for BVA. In the table 6.1 the total execution time for each test is presented, these times however does not include the time it takes to navigate to the page where the test is going to be performed. The times are only summed from the parts of the test that treats the actual method.

Method	Decision table	State transition	BVA
Time to execute a test with two input fields	11463 ms	12568 ms	13744 ms

TABLE 6.1: Comparison of execution times for three test case methods

As seen in table 6.1 the execution time for the three methods are marginal. There are some limitations to this comparison as there is considerable difference between the operating systems Windows and Mac. There is also a difference running tests on different browsers and times will vary between repeated runs, even on the exact same setup.

## 6.3 Test Case Design Guide

To be able to keep a good structure of the tests created and a way to easier maintain tests a test case design guide E was created for a test designer to fill in, one for each test case. This will also make it easier to build and maintain tests.

## 6.4 Continuous Integration

Developers want to test their code often so that bugs and errors are detected early on and a very good way to achieve this is to have the tests running on a CI server 3.4. This way the code will be tested when a developer integrates it through git. This is the preferred way to automatically run tests on the developers code and it's described how this was implemented in section 5.7.

When having the tests running on a CI server and the goal is to only test the frontend, the backend server should not be hosted on the CI server aswell. This will mean that developers have an extra copy of the backend server to maintain.

## 6.5 Mocks

This section covers results from evaluating mocks with complexity measures for the tools ngMockE2E and Prism.

### Live and Mocked Backend

A comparison was made to evaluate differences in time to run between a live backend and a mocked one. The time comparison is marginal between running tests against a live backend and running tests against a mocked backend as seen in the table 6.2 below.

Test suite	Live backend	Mocked backend	Backend requests
Change password test	32 tests passing (34s)	32 tests passing (32s)	19
Register test	15 tests passing (21s)	15 tests passing (19s)	9

TABLE 6.2: Time comparison for two tests between real backend and mocked backend

Comparing a live backend setup and a mocked backend with execution times generated a very small difference, 6.2. The first test took 32 seconds for a mocked backend and 34 seconds for a live backend, which is a difference in 2 seconds or 6%. The second test had the same time difference of 2 seconds but in this case it was a 10% difference. Since the first test used 19 backend requests and the second just 9 it can be determined that the difference in execution time lies in setup and not in request calls. The difference in execution time between a live backend and a mocked one is therefore 2 seconds for setup, which is negligible in this scenario. It's not possible to make a preferred choice from execution time between a live or mocked backend.

The tests in 6.2 are performed on a Macbook Air 13" mid 2013 with Selenium running the tests in Chrome.

### Complexity for Prism Setup

Since the complexity is non-linear for Prism the logic will only be on the actual setup of the automatic mocking tool and its integration with the current flow, rather than on the mocks themselves. The saved mocks through Prism is saved as JSON which doesn't contain any logic.

The setup file that is analyzed contains instructions for a frontend server, two proxy servers, configuration for mocks and setup for Prism as a third proxy middleware. The file also contains information about the taskrunners and a mock file name generator.

From table 6.3 the tool Prisms setup file has a high maintainability of 123. And from table 6.4 the combined Halstead effort is 5 minutes and 26 seconds which is considered low. This gives an indication that it's easy to work with.

LOC	79
Mean parameter count	12
Cyclomatic complexity	7
Cyclomatic complexity density	9%
Maintainability index	<b>123</b>

TABLE 6.3: Complexity measures for a Prism setup file

In the setup file there are four anonymous <sup>1</sup> functions and five declared functions. The four anonymous functions will be ignored for detailed presentation as they just make a function call in this case, contributing a negligible complexity.

	A	B	C	D	E
LOC	7	10	7	8	1
Parameter count	3	2	3	2	2
Cyclomatic complexity	3	1	1	5	1
Cyclomatic complexity density	43%	10%	14%	63%	100%
Halstead difficulty	6	3	5	8	2
Halstead volume	180	179	376	271	50
Halstead effort	1023	447	1952	2168	113
Time required to program	67s	25s	1m 48s	2m	6s
Delivered bugs	0.034	0.019	0.052	0.056	0.008
Program level	29.5	51.3	19.2	17.9	128.4

TABLE 6.4: Complexity measures for Prism functions where A: function proxyMiddleware, B: function setupPrism, C: function customMockFileName, D: function connectInit, E: function middleware

### Complexity for ngMockE2E Mock

First a mock for the login test is evaluated containing four different HTTP responses; bad authentication, good authentication, booking list and current user response.

LOC	148
Mean parameter count	21
Cyclomatic complexity	1
Cyclomatic complexity density	1%
Maintainability index	<b>93</b>

TABLE 6.5: Complexity measures for ngMockE2E mock file

In the login test file there is one declared function: *loginMock*, and six anonymous functions. Since five out of the six anonymous functions are defined in the exact same way only one out of the identical ones will be included in the table.

Isolating data from functionality the estimated time to program drops from **29 minutes and 28 seconds** to **1 minute and 16 seconds** which is the combined time from *loginMock* and all anonymous functions in table 6.8.

<sup>1</sup>Anonymous: Is a function that defined without an identifier or name

	loginMock	anonymous	anonymous x5
LOC	1	140	1
Parameter count	0	1	4
Cyclomatic complexity	1	1	1
Cyclomatic complexity density	100%	1%	100%
Halstead difficulty	2	9	2
Halstead volume	43	3634	37
Halstead effort	101	31407	63
Time required to program	6s	29m 5s	4s
Delivered bugs	0.007	0.332	0.005
Program level	138.3	3.0	189.5

TABLE 6.6: Complexity measures for ngMockE2E mock file functions

LOC	14
Mean parameter count	21
Cyclomatic complexity	1
Cyclomatic complexity density	7%
Maintainability index	<b>144</b>

TABLE 6.7: Complexity measures for ngMockE2E mock file without injected JSON

	loginMock	anonymous	anonymous 1-5
LOC	1	6	1
Parameter count	0	1	4
Cyclomatic complexity	1	1	1
Cyclomatic complexity density	100%	17%	100%
Halstead difficulty	2	4	2
Halstead volume	43	234	37
Halstead effort	101	908	63
Time required to program	6s	50s	4s
Delivered bugs	0.007	0.0313	0.005
Program level	138.3	32.0	189.5

TABLE 6.8: Complexity measures for ngMockE2E mock file functions without injected JSON

From table 6.9 it's evident that the lines of logic are very high for some test files as well as time required. Since the mocks are test suite specific and not reusable the Halstead effort needs for each test required to be summed. This leads to a total required time for manual mocks to be high for large applications. One thing that isn't possible to tell from table 6.9 is that some of these mocks share the same requests, which creates redundancy, duplication and a higher difficulty to update responses.

	A	B	C	D
Logical LOC	148	396	1600	335
Mean parameter count	21	17	17	17
Cyclomatic complexity	1	1	1	1
Cyclomatic complexity density	1%	0%	0%	0%
Maintainability index	<b>93</b>	<b>67</b>	<b>38</b>	<b>71</b>
Combined Halstead effort	31823	212376	1319293	150949
Combined Halstead time required	29m 28s	3h 16m 39s	20h 21m 34s	2h 19m 46s
Combined delivered bugs	0.335	1.187	4.010	0.945

TABLE 6.9: Complexity measures for multiple ngMockE2E mock files  
A: file loginMock, B: file editProfileMock, C: file marketplaceMock, D: file changePasswordMock

## 6.6 Coverage

Coverage [3.5](#) is a tool or measurement of how much an application is covered from tests in terms of how much is tested. In end-to-end testing interactable elements are tested and there was no framework for this so a custom one was built, see [section 5.6](#) of how. This custom built end-to-end coverage tool counts all interactable elements on each page the tests visit and calculate the coverage from the tests. The coverage report is shown on a separate web page where the user can browse the pages and elements to see what was tested and what was not.

The coverage tool could be extend to incorporate saved information from user session data [4.4](#) gathered from the live product. With the help of this additional data the coverage report could generate a priority list of elements and areas to improve on. This would make it a lot easier for testers to know what to test.

# Chapter 7

## Discussion

This chapter discusses the result this thesis concluded in. It touches on the different areas in results and adds reflections to the different sections.

### Tools

The tools and frameworks used in this thesis are primarily chosen for their fit with the existing setup that was available. For instance the testing tool Protractor works well with the frontend framework Angular, which was used in advance. Although it's possible to use tools not built for this environment it would have required more time and overhead to get it up and running. Frameworks with a bigger community were also thought of as better suitable than those with a small amount of contributors. Community size was however just a tipping point decision maker if two frameworks were considered near equal.

### Test Case Methods

The three methods, BVA, DT and STA take roughly the same amount of time to write. The BVA method is the fastest one because it's very straight forward. A field with a numeric input and only two limits will generate six cases: *min-*, *min*, *min+*, *max-*, *max*, *max+*, see [3.1.2](#).

It takes longer time to write a test with the STA method. The states of a process must be defined and after that the transitions between the states must be drawn. This can be hard, especially if the process has a lot of transitions and/or if it's hard to find states. The STA method can get out of hand if there are too many transitions since it will take an unreasonable amount of time to test every combination of transitions.

The decision table method is intuitive to code when the decision table is complete but it's the making of the table that can require an investment of time. It doesn't seem that hard at first but choosing the conditions is a skill in order to keep the number of

conditions at a reasonable amount. Only the most basic conditions can be in a decision table because it has  $2^x$  possible outcomes, where  $x$ =number of conditions.

The three methods evaluated are difficult to draw distinct conclusions on as they are suited for different types of tests. The STA method takes some time to get started with and is good for tests where there are clear states or processes, like a payment process or a booking process. The DT method also takes some time to get started with and it's hard to decide which conditions to include. It can't be too many conditions as the number of outcomes quickly adds up to an unmanageable amount. For test that include inputs that combined generate different outcomes the DT method is recommended. The BVA method is suited for numerical inputs with limits. It's very straight forward as long as the limits of the inputs is known.

The errors that the methods detect differ between the methods since they don't test for the same things. The DT tests that different combination of inputs leads to the correct outcome. The STA method explores different transitions between states and checks if the transition taken led to the expected state. The BVA however is as mentioned before suited for numerical inputs or inputs with hard limits and will detect errors that occur near the limit boundaries, see section 3.1.2.

As seen in table 6.1 the difference in execution times are marginal. The test case methods execution time only differs 17% from slowest to fastest, which isn't a strong enough indication that either one of them should be used based on execution time alone.

From the execution time comparison and the difference in errors that these methods detect none of the methods are thought of as better than the other. These three methods should be used together for best performance. When none of them are suited for the test situation, an ad hoc model should be used.

### Test Case Design Guide

After reworking Assassas test case template [18] to make it fit end-to-end tests in a better way it became very handy. Even though the test case design guide E is only used once when writing a test, that time it went a lot easier and quicker to actually write the test following the written guide. The guide might still be missing some field that a tester could find important, this could have been brought to attention if a user study was made. The time for that was unfortunately not found within this thesis. Another big plus is that the test cases written gets documented in a structured way. The choice of adding more fields to the guide can be discussed but this guide is mainly created to help the tester write the test. This means that if unnecessary fields were added it will take longer time to fill in and it could be confusing for the person who will follow it and create the test. The decision to keep it simple with only the most important fields was made. The test case design guide successfully fulfills the coveted goal.

### Continuous Integration

Continuous integration helps developer to be more efficient through optimizing tasks such as code building and testing. With the help of pushing code and testing automatically onto a CI, bugs can be found faster. In comparison if a CI wasn't utilized then bugs would be found later on which then could have more impact on other new areas.

Integration of tests also helps finding building errors and gives developers a constant feedback. This constant feedback is important as it notifies exactly when a bug or error is introduced. Making it easier to find what part of the code that is creating this breaking change. By taking care of testing automatically it becomes a more used practice instead of relying on testers and developers to remember to constantly test their new versions.

## Mocks

NgMockE2E is a very straight forward tool in the sense that it's viewed as a static file containing information about the mocked HTTP request. However its features are limited and setting up multiple mocks stack up in terms of time, complexity and cost. The isolation of information and reuse of components between mocks are practically nonexistent and because of that the correlation between the amount of mocks and effort required is near linear, as can be seen through Halstead metrics in 6.9.

A discovered drawback for ngMockE2E is that it's served static to the frontend. This makes the module limited since it cannot load external resources or wait on promises inside the module. Not being able keep module functionality separated from mock data creates a harder environment to work with. Some responses from the backend can contain several thousands lines in JSON which is not suited for being inlined. Not having the functionality to separate responses into self-contained files will generate duplicates.

If the tool could be used without having to inline responses inside the mocked module, the result would be very different as seen in tables 6.5 and 6.6 compared with tables 6.7 and 6.8. The separation of concern is here very visible in measurements of complexity.

Prism is a more sophisticated tool that is well suited for larger scaled applications, as the amount of mocks or HTTP requests to simulate isn't a limiting factor. Working with an automatic recording tool for mocks demands a higher precision on setup and preconditions as each mock needs to be unique. Prism takes longer time to get a grasp of but will demand significantly less upkeep in time, space and cost for future development and extensions making it the preferred option.

One caveat of working with an automatic tool is the fact that the mocks are not designed to be manipulated manually. This might not be necessary but the absence of functionality to edit a mock could be troublesome. Some scenarios that might need a different response than the actual one from the database is not possible to create.

Important thing to notice is that even though Prism's score on the maintainability index, **123 6.3**, and Halstead effort, **5703 6.4**, is very good it's not a definitive score. According to Paul Omand and Jack Hagemester in *"Metrics for assessing a software systems*



*maintainability*” the higher score the better, with 171 being max and then going to negative infinity. They made the statement that below 65 on the maintainability index is considered an indication of poor quality and above 85 to be of high quality.

However having a score below 65 is considered a clear sign that it’s going to be hard to maintain. This can be seen from ngMockE2E tests scoring between **38** to **93** and the experienced feeling that it’s difficult to work with, especially on a large scale.

## Coverage

The coverage tool built in this thesis presents an intuitive way of finding elements that are tested in a single page application. Elements are presented with overall statistics as well as categorized into states to make navigation easy and structure similar to interaction with the web page.

Currently the coverage tool will display nested elements without a proper visualized element. The practice developed here will only calculate styled properties on the actual element and not the nested ones, making a form appear visually significantly different from the real version.

Elements presented across the web page might run into being counted as the same. Since elements are hashed based on their outerHTML property some elements that are marked up exactly the same but used differently will be viewed as the same. In the coverage report the elements will still have a local and global property to determine where the element was interacted with. However it’s possible for two elements to appear on the same state and be viewed as the same, even if it’s unlikely.

When detecting unique elements all classes from an element is stripped, which is a rather rough approach. This is utilized from the fact that the Angular framework append and remove classes based on interactive states of an element. Therefore it could make sense to just remove classes that are Angular specific instead of all classes presented. However this isn’t necessarily beneficial since the tool is developed as a plugin for Protractor which in itself is Angular specific.

One issue that this report didn’t have time to cover is the situation where a state is never interacted with. If a state is never visited in a test suite it will not show in the coverage report, which could give a false indication. A way to find all possible states would give a better picture of test coverage of a product.

Test adequacy criterias current limitation is that doesn’t determined which events are more important. If one event have been interacted with the element is considered tested but it gives no indication if it missed to interact with one of its more important events. A possible other approach is to define a tested element as if all possible events have been triggered but this was considered to be a to strict restriction for the tool to be useful.

Combining user session data with a coverage report could help developers target important sections. If an object from user session data has a high frequency in interaction and

---

is untested in the coverage report it could be utilized to give a strong indication of next primary action to take. Trying out this concept was outside the scope of this thesis.

## Chapter 8

# Conclusion

This chapter covers the conclusions for this thesis and answers the problem definitions.

*How can a comparison of different end-to-end testing methods lead to a more effective way of writing test cases?*

The comparison of the different testing methods does not directly lead to a more effective way of writing test cases. However the comparison lead to a knowledge of when to use which method. This knowledge itself lead to the creation of a test case design guide [E](#). Following this test case design guide makes test case writing more effective.

*How can a coverage report be built for end-to-end tests without the knowledge of source code and what information should be analyzed to help a tester?*

Exactly how a coverage report can be built for end-to-end testing can be read about in section [5.6](#). This is done without the knowledge of the source code by adding eventlisteners to interactable elements during the tests to get information. This information is later displayed to help the developer with information of what is tested and what is not. Information from users interacting with the web page can be used to improve the coverage with recommendations on what to test next. Information that could be interesting to store is how many times an element is interacted with and how long time users spend on a certain page. With this information a priority list of what should be tested next could be generated inside the coverage tool. This would help testers to cover the most important parts first. However, the implementation of such a thing was not within the timeframe of this thesis.

*How can end-to-end tests be automated and how can feedback from it improve a development process?*

End-to-end tests can be automated by running them on a continuous integration server. This server will run the tests every time someone integrates their code. The server will

also give developers feedback on failing tests which helps them to quickly narrow down the error so it can be solved directly.

*How does automatic mock creation compare to manual mock creation and how can an automatic process be achieved?*

Manual mock creation is a method that takes linear time to build as a tester needs to create every mock that is going to be used. In manual mocking the responses from requests need to be inlined in the mocks since the tool can't load external resources. Automatic mock creation however has a longer time for setup but after that all the mocks are created automatically when running the tests. An automatic mock process is achieved by using the JavaScript framework Prism and building a custom middleware proxy that handles redirection of requests. Other things that need to be customized are file name for the mocks, structure for unique endpoints and an url counter. Exactly how this mocking tool is built can be read about in section [5.5](#). Automatic mock creation is preferred for end-to-end user testing to save time and for its ease of keeping the mocks up to date.

# Bibliography

- [1] Selenium 1 (selenium rc). URL [http://www.seleniumhq.org/docs/05\\_selenium\\_rc.jsp](http://www.seleniumhq.org/docs/05_selenium_rc.jsp). Accessed March 25th, 2015.
- [2] Mocha, javascript testing framework. URL <http://mochajs.org/>. Accessed May 21th, 2015.
- [3] Amit Agarwalla. Continuous integration (ci) stack, 2014. URL <https://angraze.wordpress.com/2014/02/09/continuous-integration-ci-stack/>. Accessed May 22th, 2015.
- [4] What is decision table in software testing? URL <http://istqbexamcertification.com/what-is-decision-table-in-software-testing/>. Accessed March 24th, 2015.
- [5] Html element reference, . URL <https://developer.mozilla.org/en-US/docs/Web/HTML/Element>. Accessed May 12th, 2015.
- [6] Event reference, . URL <https://developer.mozilla.org/en-US/docs/Web/Events>. Accessed May 12th, 2015.
- [7] Competencer about. URL <https://www.competencer.com/aboutus>. Accessed May 25th, 2015.
- [8] Glenford J Myers and Corey Sandler. The art of software testing. *New Jersey. John Wiley & Sons*, 2004.
- [9] James A Whittaker, Jason Arbon, and Jeff Carollo. *How Google tests software*. Addison-Wesley, 2012.
- [10] Stephen Chapman. What is javascript. URL <http://javascript.about.com/od/reference/p/javascript.htm>, 2013.
- [11] Sang Shin. Introduction to json (javascript object notation). *Presentation http://www.cse.iitd.ac.in/cs5090250/JSON.pdf*, 2010.
- [12] Jev Zelenkov. Make the browsers test for you. part 1: Selenium webdriver, 2014. URL <http://www.jzelenkov.com/posts/make-the-browsers-test-for-you-part1/>. Accessed March 19th, 2015.

- 
- [13] Chai, javascript assertion library. URL <http://chaijs.com/>. Accessed May 21th, 2015.
- [14] The web framework for perfectionists with deadlines. URL <https://www.djangoproject.com/>. Accessed May 27th, 2015.
- [15] Node.js. URL <https://nodejs.org/>. Accessed May 27th, 2015.
- [16] gulp.js - the streaming build system. URL <http://gulpjs.com/>. Accessed May 27th, 2015.
- [17] Angularjs - superheroic javascript mvw framework, . URL <https://angularjs.org/>. Accessed May 27th, 2015.
- [18] Ghazy Assassa. Software engineering, test case template and examples. URL [http://faculty.ksu.edu.sa/ghazy/CSC342\\_Tools/Test%20Case%20Template.pdf](http://faculty.ksu.edu.sa/ghazy/CSC342_Tools/Test%20Case%20Template.pdf). Accessed May 27th, 2015.
- [19] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2000.
- [20] Mike Lazer-Walker. Test doubles: Mocks, stubs, and more, 2014. URL <http://www.objc.io/issue-15/mocking-stubbing.html>. Accessed March 19th, 2015.
- [21] Steve Freeman. Test smell: Everything is mocked, 2007. URL <http://www.mockobjects.com/2007/04/test-smell-everything-is-mocked.html>. Accessed March 19th, 2015.
- [22] Joakim Kolsj. Only mock the things you own, 2014. URL <http://rubyblocks.se/2014/03/02/only-mock-the-things-you-own/>. Accessed March 19th, 2015.
- [23] Connect-prism, . URL <https://github.com/seglo/connect-prism>. Accessed March 19th, 2015.
- [24] Vcr, . URL <https://github.com/vcr/vcr>. Accessed March 19th, 2015.
- [25] Sean Glover. Record, mock, and proxy http requests with grunt-connect-prism, 2014. URL <http://randomom.com/blog/2014/06/record-mock-and-proxy-http-requests-with-grunt-connect-prism/>. Accessed March 19th, 2015.
- [26] ngmock, . URL <https://docs.angularjs.org/api/ngMock>. Accessed March 19th, 2015.
- [27] ngmocke2e, . URL <https://docs.angularjs.org/api/ngMockE2E>. Accessed March 19th, 2015.
- [28] Ken Rimple. Angularjs corner the ngmock and ngmocke2e libraries, 2014. URL <http://chariotsolutions.com/blog/post/angularjs-corner-ngmock-ngmocke2e-libraries/>. Accessed March 19th, 2015.

- 
- [29] Martin Fowler. Continuous integration, 2006. URL <http://martinfowler.com/articles/continuousIntegration.html>. Accessed May 12th, 2015.
- [30] Brian Marick. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pages 16–18, 1999.
- [31] Elinda Kajo-Mece and Megi Tartari. An evaluation of java code coverage testing tools. In *BCI (Local)*, pages 72–75, 2012.
- [32] Html5 a vocabulary and associated apis for html and xhtml. URL <http://www.w3.org/TR/html/>. Accessed May 12th, 2015.
- [33] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher. Leveraging user-session data to support web application testing. *Software Engineering, IEEE Transactions on*, 31(3):187–202, 2005.
- [34] httpbackend, . URL <https://docs.angularjs.org/api/ngMock/service/%24httpBackend>. Accessed May 18th, 2015.
- [35] Ken Rimple. Angularjs corner the ngmock and ngmocke2e libraries, 2014. URL <http://chariotsolutions.com/blog/post/angularjs-corner-ngmock-ngmocke2e-libraries/>. Accessed May 18th, 2015.
- [36] Codeship. URL <https://codeship.com/>. Accessed May 12th, 2015.

# Appendix A

## Implementation of URL Counter

```
1  'use strict';
2
3  var _ = require('lodash');
4
5  function UrlCounter() {
6    // Structure { url: '/api/', count: 1}
7    var _db = [];
8
9    this.getCount = function(url) {
10     var item = getItem(url);
11
12     if(exists(item)) { // if it exists increment
13       item.count += 1;
14       return item.count;
15     }
16
17     else buildItem(url); // else create
18     return 1;
19   };
20
21   function getItem(url) {
22     return _.find(_db, {'url': url});
23   };
24
25   function exists(obj) {
26     return typeof obj !== 'undefined';
27   };
28
29   function buildItem(url) {
30     var _obj = {
31       'url': url,
32       'count': 1
33     }
34
35     _db.push(_obj);
36   };
37 }
38
39 module.exports = UrlCounter;
```

---



## Appendix B

# Config File for Protractor

```
1 exports.config = {
2   // Specify the path to the selenium standalone server
3   seleniumServerJar: '../path/selenium-server-standalone-2.45.0.jar',
4   // Specify the path to the chromedriver
5   chromeDriver: '../path/chromedriver',
6   // Base url
7   baseUrl: 'http://localhost:3000',
8   // Framework
9   framework: 'mocha',
10  // Capabilities to be passed to the webdriver instance.
11  capabilities: {
12    'browserName': 'chrome'
13  },
14
15  /**
16   * Suites
17   * these suites need a specific order
18   * as they create objects other depend on
19   */
20  suites: {
21    firstTest: ['e2e/firstTest/*.js'],
22    secondTest: ['e2e/secondTest/*.js'],
23    ...
24  },
25  // Before tests
26  onPrepare: function () {
27    browser.driver.manage().window().setSize(1440, 900);
28    browser.driver.manage().window().setPosition(0, 0);
29  },
30  // Mocha options
31  mochaOpts: {
32    ui: 'bdd',
33    reporter: 'spec', //spec, tap, dot, progress, list, nyan, min
34    timeout: 100000
35  }
36 };
```

---

## Appendix C

# Setup Commands for Codeship

```
# Install and use a version of the Node Version Manager
nvm install 0.10.25
nvm use 0.10.25

# Choose where globally installed node packages gets cached
npm config set cache "${HOME}/cache/cache/npm/"

# checkout frontend from repository root to enable Codeships cache control path
git subtree split --prefix=FRONTEND_FOLDER --branch=frontend
git checkout frontend

# The node packages is installed
npm install

# install bower and gulp globally to get access to environment variables
npm install -g bower gulp

# The bower dependencies are installed
bower install

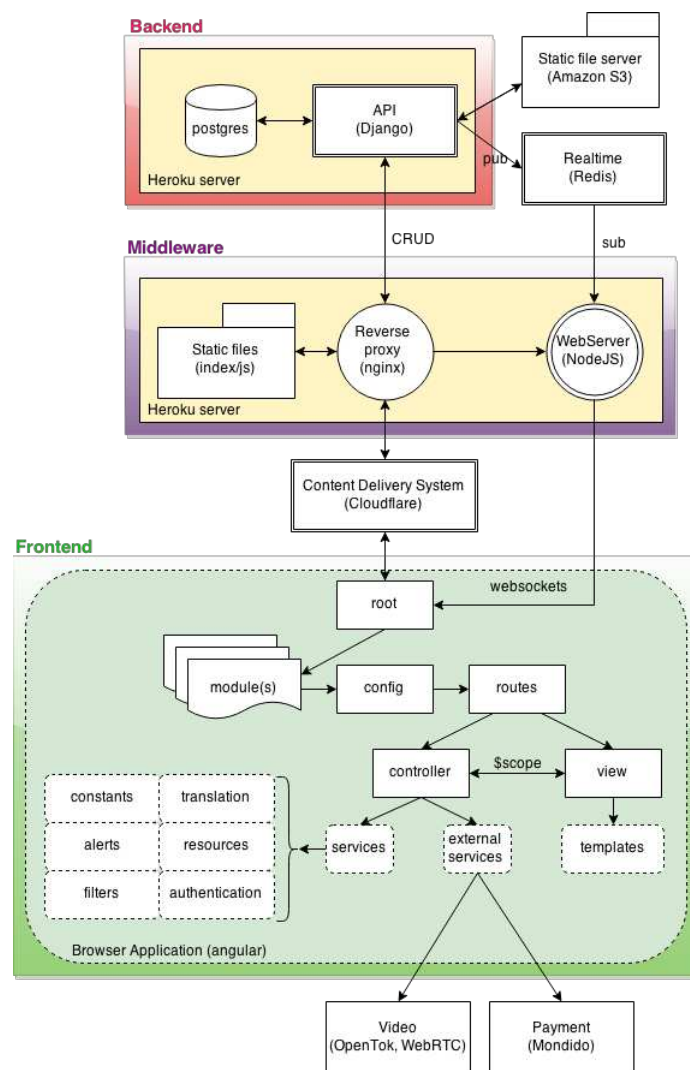
# The webdriver gets updated
./node_modules/protractor/bin/webdriver-manager update

# The webdriver server is started to run in the background
nohup bash -c "webdriver-manager start 2>&1 &" && sleep 9

# The testsuites are run via a custom protractor task for gulp
gulp protractor:mock
```

# Appendix D

## Competencer Architecture



# Appendix E

## Test Case Design Guide

### E.1 Introduction

This test case design guide is made for end-to-end tests on Competencers frontend but works well for test cases in general on single page web applications. These guidelines are made for the person creating the tests and for the person who will maintain the tests. If these guidelines are followed it will help maintaining a good and clear structure that makes tests easy create, work with and maintain.

### E.2 Starting Conditions

The tests should be built for an empty database. The tests should also be run in a specific order so that the first tests that are run are the tests that the later ones might depend on. This means that as the tests are running it will gradually populate the database but it should always start with an empty database so that every testrun runs on the same conditions.

### E.3 Choosing Test Method for Inputs

The choice of method depends on what the test does and what elements it is going to test. The efficiency of these methods depend on what you apply them on. It is possible to combine these methods in the same test, for best result follow the guidelines below:

For processes that spans over multiple states or views: **State transition analysis**

Inputs that gives different outcome depending on combination of inputs: **Decision table** If the number of inputs are greater than 3, divide the inputs into subgroups.

Inputs with hard limits, for example numeric inputs: **Boundary value analysis**

If it is vital that the input field does not break anything, for example credit card fields:  
**Ad hoc** Test everything: numerics, non numerics, special characters, html, few characters, many characters etc.

## E.4 Test Case Design Template

**Test suite name:**

**Short description:**

**Dependencies:**

Other test suites that your test depend on. For instance registering users as a dependency for editing profile.

**Visited pages:**

Pre-conditions
1.

**Method(s):**

Tests	
Action	Expected system response
1.	
2.	
3.	

Post-conditions
1.

## E.5 Text Case Design Template Example

**Test suite name:** Edit settings

**Short description:** A user should be able to enter settings and change password, update country and similar items.

**Dependencies:** Register

Other test suites that your test depend on. For instance registering users as a dependency for editing profile.

**Visited pages:** /settings/account, /settings/payments, /settings/payouts

Pre-conditions
1. User is logged in
2.

**Method(s):** Ad hoc

Tests	
Action	Expected system response
1. Enter Markus as new first name	First name placeholder is updated to Markus
2. Clicks Update	Displays a message with confirmation of updated information
3.	

Post-conditions
1. Users first name is updated to Markus
2.

## Appendix F

# Protractor-E2E-Coverage Tool

```
1 var q = require('q'),
2     crypto = require('crypto'),
3     fs = require('fs'),
4     path = require('path'),
5     _ = require('underscore'),
6     wrench = require('wrench');
7
8
9 var CoveragePlugin = function() {
10     this.DOMElements = [];
11     this.logs = [];
12     this.browserLogAvailable = false;
13     this.name = 'CoverageE2E';
14     this.outdir;
15     this.config = {
16         elements: []
17     };
18 };
19
20 CoveragePlugin.prototype.hash = function(elem) {
21     var shasum = crypto.createHash('sha1');
22     // remove html classes for hash
23     // so we don't get duplicate on things like .ng-touched
24     var r = '/\s\bclass=("[^"]+")/g';
25     shasum.update(elem.replace(r, ' '));
26     return shasum.digest('hex');
27 }
28
29 CoveragePlugin.prototype.updateElement = function(event, obj, url) {
30     var self = this;
31
32     var hash = self.hash(obj);
33
34     var index = _.findIndex(self.DOMElements, {'url': url});
35     var elem = _.findIndex(self.DOMElements[index].elements, {'hash':
36         hash});
37
38     if(elem !== -1) {
39         var element = self.DOMElements[index].elements[elem];
```

```
39
40     element.tested = true;
41
42     if(element.events.indexOf(event) === -1) {
43         element.events.push(event);
44     }
45 }
46 }
47
48 CoveragePlugin.prototype.storeElement = function(element, type) {
49     var self = this;
50
51     var hash = self.hash(element.item);
52     var index = _.findIndex(self.DOMElements, {'url': element.location
53         });
54
55     // element structure
56     function buildElement() {
57         return {
58             'hash': hash,
59             'element': element.item,
60             'css': element.css,
61             'type': type,
62             'tested': false,
63             'events': []
64         }
65     }
66
67     // if the url hasn't been seen
68     if(index === -1) {
69         var urlObj = {
70             'url': element.location,
71             'elements': [buildElement()]
72         }
73         self.DOMElements.push(urlObj);
74     }
75
76     // if the item on the location hasn't been seen
77     else if(_.findIndex(self.DOMElements[index].elements, {'hash': hash
78         }) === -1) {
79         self.DOMElements[index].elements.push(buildElement());
80     }
81 }
82
83 CoveragePlugin.prototype.parseLogs = function(config) {
84     var self = this;
85
86     if(this.browserLogAvailable) {
87         this.logs.forEach(function(log) {
88             var warnings = log.filter(function(node) {
89                 return (node.level || {}).name === 'WARNING';
90             });
91
92             warnings.forEach(function(elem) {
93                 var m = JSON.parse(elem.message);
```



```

91         if (m.message.hasOwnProperty('parameters')) {
92
93             var p = m.message.parameters;
94
95             if(p[0].value === self.name) {
96                 self.updateElement(p[1].value, p[2].value, p[3].value);
97             }
98         }
99     });
100 });
101 }
102 };
103
104 CoveragePlugin.prototype.saveLogs = function(config) {
105     var self = this;
106
107     if(this.browserLogAvailable) {
108         browser.manage().logs().get('browser').then(function(log) {
109             self.logs.push(log);
110         });
111     }
112 };
113
114 CoveragePlugin.prototype.setup = function(config) {
115     var self = this;
116     self.outdir = path.resolve(process.cwd(), config.outdir);
117
118     if(config.elements) {
119         self.config.elements = config.elements;
120     }
121
122     browser.manage().logs().getAvailableLogTypes().then(function(res) {
123         self.browserLogAvailable = res.indexOf('browser') > -1;
124     });
125 };
126
127 CoveragePlugin.prototype.postTest = function(config) {
128     var self = this;
129     var deferred = q.defer();
130
131     browser.executeScript(function() {
132         var helper = {
133             hashCode: function (s) {
134                 var clean = helper.cleanElement(s);
135                 return clean.split("").reduce(function(a,b){a+=((a<<5)-a)+
136                 b.charCodeAt(0);return a&a},0);
137             },
138             cleanElement: function(s) {
139                 // remove html classes for hash
140                 // so we don't get duplicate on things like .ng-touched
141                 var r = /\s\bclass=( [^ ]+ )/g ;
142                 return s.replace(r, ' ');
143             },
144             getNodes: function(type) {

```

```
144     // return NodeList
145     var arr_nodes = document.querySelectorAll(type);
146     // convert to array
147     return Array.prototype.slice.call(arr_nodes);
148   }
149 }
150
151 // Elements and events we want to investigate
152 var DOMcomponents = arguments[0];
153
154 var url = window.location.pathname;
155
156 DOMcomponents.forEach(function(DOMtype) {
157   var DOMitems = helper.getNodes(DOMtype.type);
158
159   DOMitems.forEach(function(item) {
160     var hash = helper.hashCode(item.outerHTML + url);
161
162     // check if eventlistener exists
163     if(!window.sessionStorage.getItem(hash)) {
164
165       // if not, add one eventlistener for each event
166       var events = DOMtype.events;
167       events.forEach(function(event) {
168         item.addEventListener(event, function() {
169           // needs to be info to be caught by the browserlogs
170           capture
171             console.info('CoverageE2E', event, item.outerHTML,
172               window.location.pathname);
173         });
174       });
175
176       // store eventlistener in sessionstorage
177       window.sessionStorage.setItem(hash, 'CoverageE2E');
178
179       // get computedCss on element, doesnt look at nested
180       var css = window.getComputedStyle(item).cssText;
181
182       DOMtype.elements.push({'item': item.outerHTML, 'css': css,
183         'location': url});
184     }
185   });
186 });
187
188 return DOMcomponents;
189 }, self.config.elements).then(function(DOMcomponents) {
190
191   DOMcomponents.forEach(function(DOMtype) {
192     var elements = DOMtype.elements;
193     elements.forEach(function(elem) {
194       self.storeElement(elem, DOMtype.type);
195     });
196   });
197 });
198 }
```

```
195     self.saveLogs();
196     deferred.resolve();
197   });
198
199   return deferred.promise;
200 };
201
202 CoveragePlugin.prototype.outputResults = function(done) {
203   var self = this;
204
205   try {
206     fs.mkdirSync(self.outdir);
207   } catch (e) {
208     if (e.code !== 'EEXIST') throw e;
209   }
210
211   // build coverage file
212   var outfileCoverage = path.join(self.outdir, 'coverage.json');
213   fs.writeFileSync(outfileCoverage, JSON.stringify(self.DOMElements))
214     ;
215
216   // save config setting
217   var outfileConfig = path.join(self.outdir, 'config.json');
218   fs.writeFileSync(outfileConfig, JSON.stringify(self.config.elements
219     ));
220
221   // copy report folder
222   wrench.copyDirRecursive(__dirname + '/report', self.outdir + '/
223     report', {forceDelete: true}, done);
224 };
225
226 CoveragePlugin.prototype.postResults = function(config) {
227   var self = this;
228   var deferred = q.defer();
229
230   self.parseLogs();
231   self.outputResults(deferred.resolve)
232
233   return deferred.promise;
234 };
235
236 var coveragePlugin = new CoveragePlugin();
237
238 exports.setup = coveragePlugin.setup.bind(coveragePlugin);
239 exports.postTest = coveragePlugin.postTest.bind(coveragePlugin);
240 exports.postResults = coveragePlugin.postResults.bind(coveragePlugin)
241   ;
242 exports.CoveragePlugin = CoveragePlugin;
```

---