Masaryk University
Faculty of Informatics

Automated visual testing of web application

Diploma thesis

Juraj Húska

Brno, 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Juraj Húska

Advisor: Mgr. Marek Grác, Ph.D.

Acknowledgement

I would like to thank Mgr. Marek Grác, Ph.D., the advisor of my thesis, for his help, comments and time spent helping me with this work.

I am profoundly grateful to Bc. Lukáš Fryč for his guidance and support throughout my work.

My deepest thanks go to my parents, who made it possible for me to study at this brilliant university.

I would also like to say thanks to the Czech Republic for allowing me to study at the Masaryk University. Similarly, I would like to express my thanks to Red Hat company, I cooperated on this thesis with. Particularly to Mgr. Pavol Pitoňák for allowing me to dedicate some of the work time to this thesis.

I greatly appreciate the support of my girlfriend during writing of this thesis, as well as help of all my close friends.

Abstract

This diploma thesis describes the creation of the tool and defines a process which will enable automated visual testing of a web application. The process improves the effectiveness of the QA team. A research to find out what output of such a tool would be most useful is conducted. The visual testing is based on picture comparison. The most effective way of storing and defining the picture base, which all the other pictures will be compared with is determined. The usage of such a tool in Continuous Integration environment is described. An analysis of the already made solutions is done before the tool creation itself. One of the thesis parts consists of the deployment of the tool and the process on a particular web application.

Keywords

automated visual testing, picture comparison, web UI testing, testing

Table of contents

1 Introduction

To make sure that the released software is bug free, one has to test it properly. Testing of software is the only assurance of quality.

In the testing process for an application with a user interface, one of the last steps is to ensure that all of the promised functionality is delivered, and the design of the user interface is not broken.

Many software developing companies are ensure this by manually checking all possible use cases provided by their developed software's UI. This tedious activity is very time consuming and error prone, thus also quite expensive.

The focus of this thesis is to mitigate the need for manual testing of user interfaces by introducing a tool which would automate the major part of it. This would enable the so called automated visual testing. The goal of such a tool is to increase the effectiveness of the quality assurance team.

The thesis consists of five main parts: the first one being a broad introduction to automated visual testing and an explanation of the a motivation behind it.

The second part analyses the already existing solutions, summarizes their drawbacks as well as advantages.

The third part formulates the hypothesis which we are trying to prove or disprove by deploying the tool on a particular web application. It also introduces a process which is necessary to adhere to in order for the created tool to be effective.

The fourth part describes the implementation details of the developed tool, provides a list of components we reused or developed to obtain the final result, as well as a justification of why we chose a particular component to integrate with.

The last part deals with the particular deployment of the tool on a real web application, its example usage within Continuous Integration systems, and within the cloud environment.

2 Definition of terms

Here we would like to define some of the important terms used in the whole thesis.

- Screenshot - a screen capture.
- Pattern - a screenshot of a web application, representing the correct state of the application, made in the first execution of a visual testing test suite, all other later screenshots will be compared with.
- Sample - screenshot of the web application, representing a current state of the application.
- Diff - a generated picture, which is made when differences are found during picture comparison of pattern and sample. It clearly shows these differences by making colorful only those different parts.
- Web Manager - a web application made as a part of this thesis to view the results of automated visual testing, and for taking immediate action over these results, to change configuration of the visual testing for subsequent runs of the test suite.
- Test suite - a set of automated functional tests for a web application.
- Test suite run - a particular execution of a test suite at a particular time, with a particular browser, etc.

3 Visual testing of software

Testing of software in general is any activity aimed at evaluating an attribute or capability of a program and determining that it meets its required results [1]. It can be done either manually by the actual use of an application, or automatically by executing testing scripts.

If the application under test also has a graphical user interface (GUI), then one has to verify whether it is not broken. The visual testing of an application is an effort to find out its non-functional errors, which expose themselves by changing the graphical state of the application under test.

A typical example can be a web application in which GUI is programmed usually with a combination of HyperText Markup Language (HTML) and Cascading Style Sheets (CSS). HTML is often used to define the content of the web application (such as page contains table, pictures, etc.), while CSS defines the structure and appearance of the web application (such as color of the font, absolute positioning of web page elements, and so on).

The resulting web application is a set of rules (CSS and HTML) applied to a static content (e.g. pictures, videos, text). The combination of rules is crucial, and a minor change can completely change the visual state of the web application. Such changes are very difficult, sometimes even impossible to discover by automated functional testing of the application. It is because functional tests verify the desired functionality of the web application, and disregard web page characteristics such as red color of heading, space between two paragraphs, and similar.

That is why a visual testing has to take place. Again, it is done either manually, when a tester goes through all of web application's use cases, and verifies that the application has not broken visually. Or it is performed automatically, by executing scripts which assert the visual state of an application.

In this thesis we are going to focus on the visual testing of web applications only. As we mentioned above, the way a web page looks like is mainly determined by CSS script. There are two ways of automated testing used:

1. asserting the CSS script
2. comparing screen captures (also known as screenshots) of new and older versions of the application.

In this thesis we are going to work with comparing screenshots only, as it is a method which is more likely to reveal a bug in the visual state of an application under test.

3.1 Visual testing in release testing process

Nowadays software is often released for a general availability in repetitive cycles, which are defined according to a particular software development process such as Waterfall [2], or Scrum

[3].

Testing of software has an immense role in this release process. Automated tests are often executed continuously, as they are quicker to run than manual tests, which are carried out at a specific stage of the release process.

For example in RichFaces[1] Quality Engineering team[2] visual testing was done manually, before releasing the particular version of RichFaces library to a community. In practice it involves building all example applications with new RichFaces libraries, and going through the use cases with a particular set of web browsers.

To be more specific, consider a web page with chart elements showing a sector composition of gross domestic product in the USA (as figure 3.1 demonstrates). To verify its visual state is not broken, it would involve e.g.:

1. Checking the size, overflowing and transparency of all elements in charts.
2. Checking colors, margins between bars.
3. Putting a mouse cursor over specific places in the chart and verifying whether a popup with more detailed info is rendered in the correct place.
4. Repeat this for all major browsers[3] and with all supported application containers[4].

3.2 Need for automation

The chapter 3.1 tried to outline how tedious and error prone might manual visual testing be. From our experience in the RichFaces QE team, any activity which needs to be repeated, and does not challenge the tester's intellect enough, becomes a mundane activity. The more one repeats the mundane activity, the more likely a mistake is introduced: one forgets to try some use cases of an application, overlooks some minor errors, etc.

Automated visual testing addresses these shortcomings, as it would unburden human resources from mundane activities such as manual testing, and would allow spending their time on more intellectually demanding problems. However, it introduces other kinds of challenges and needs to be implemented wisely. The following are minimal requirements for a successful deployment of an automated visual testing.

1. RichFaces is a component based library for Java Server Faces, owned and developed by Red Hat
2. Quality Engineering team, is among other things, responsible for assuring the quality of a product
3. Major browsers in the time of writing of this thesis are according to the [4]: Google Chrome, Mozilla Firefox, Internet Explorer, Safari, Opera
4. Application containers are special programs dedicated to providing a runtime environment for complex enterprise web applications, e.g. JBoss AS, Wildfly, Apache Tomcat

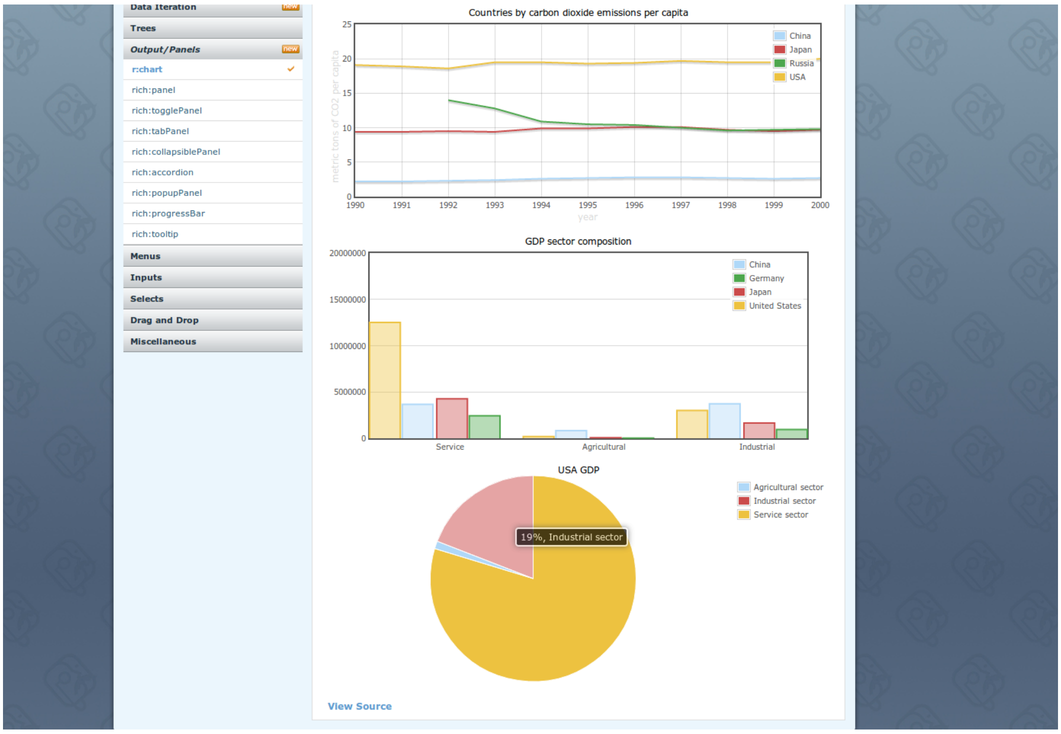

Figure 3.1: RichFaces chart component shown in Showcase application

3.3 Requirements for automation

The overall cost of the automation has to be taken into consideration. It is necessary to take into account higher initial cost of automation, and the consequences it brings, such as an increased time to process relatively huge results of testing and the cost of test suite maintenance.

Therefore, to foster effectiveness in quality assurance teams while keeping the cost of automation reasonably low, automated visual testing would require:

1. A low cost of the test suite maintenance;
2. a low percentage of false negative or false positive tests results;
3. a reasonable time to execute the test suite;
4. a concise yet useful test suite output;
5. a support for Continuous Integration systems[5].

5. Continuous Integration (CI) system is a software to facilitate a practice of CI, which in short is about

3.3.1 Low cost of test suite maintenance

A test suite needs to reflect the development of an application under test. Therefore, with each change in the application, the test suite usually has to be changed as well. Making a change in the test suite can often introduce a bug and cause false negative or false positive tests results.

To keep this cost as low as possible, the test suite script has to be readable and meaningful, so that when the change is about to be introduced, it is clear where and how it should be done.

A test framework in which the test suite is developed should provide high enough abstraction. This would enable better re-usability for various parts of the test suite, while lowering the overall cost of maintenance.

Specifically for visual testing, when done by comparing screen captures, it is very important how well a repository of screen captures is maintainable. Secondly, how reference captures (those correct ones, other screen captures will be compared with) are made.

3.3.2 Low percentage of false negative or false positive results

False negative test results incorrectly indicate a bug in an application under test, while it is a bug in the test suite itself. They are an unwanted phenomenon as they take time to process and assess correctly.

False positive tests results hide actual bugs in an application. They provide incorrect feedback by showing the tests as passing, even when there is a bug in the application.

Specifically for visual testing, when it is done by comparison of screen captures, it is very easily broken by small changes on a page. For example, if the page outputs a current date, then it breaks with any date which is different. There have to exist techniques which would prevent these situations.

3.3.3 Reasonable time to execute a test suite

A reasonable time is quite a subjective matter, but in general, it depends on how many times e.g. per day one needs to run the whole test suite. Nowadays, the trend is a Continuous Integration, in which a developer or a tester commits changes of an application several times per day to a shared source code mainline. Each such commit should trigger the test suite, which verifies whether the change did not introduce an error to the application.

According to creators of Continuous Integration practice, the whole point of CI is to provide a rapid feedback. A reasonable time for them is 10 minutes. If the build takes more time, every minute less is a huge improvement (considering that a developer/tester runs the test suite several times a day).

merging all developer copies with a shared mainline several times a day [5].

3.3.4 Concise yet useful test suite output

One of the drawbacks of automated testing is its ability to produce a huge amount of logs, test results etc. The output therefore needs to be as concise as possible, while still providing useful information. A tester needs to be able to quickly recognize where the issue might be. The best situation would be if the tester did not need to run the test again in order to spot the issue. The output should give him a clear message where the issue is.

For visual testing specifically, this can be achieved by providing a tester with screen captures together with the difference between the old version and the new one.

3.3.5 Support for Continuous Integration systems

This is quite easy to achieve, but still, a developer of a tool for visual testing should have this in mind beforehand. Current CI systems support a variety of build systems, for various platforms and languages. For example, Jenkins supports build systems like Maven or Gradle, but it can also run shell scripts.

4 Analysis of existing solutions

As we introduced in 3.3, there are many aspects which need to be taken into consideration when automating visual testing. The following analysis is going to compare existing solutions to automated testing with those requirements in mind, while introducing different approaches to visual testing.

The representative list of tools for comparison was made also according to the ability to be used in enterprise context. In an enterprise company, there is a stress on stability and reliability of the employed solutions. It is quite a vague requirement, and it is usually hard to determine which tools are a good fit for enterprise companies, however, some indicators, which we used as well, might be helpful:

- Is a project actively developed? When was the last release of the project, or how old is the last commit to a source code mainline?
- How many opened issues does the project have? When was the last activity with those issues ?
- What is the quality of the source code? Is it well structured? Does it employ the best development practices?
- Does the project have a user forum? How active are the users?
- Is a big enterprise company behind the solution, or otherwise sponsoring it ?
- What are the references if the project is commercialized ?

For each tool in the following sections we are going to show an example usage and its main drawbacks together with some basic description.

4.1 Mogo

Mogo [6] approach to visual testing can be in short described as:

1. Set of URLs of an application under test is provided to a cloud based system.
2. Application URLs are loaded into various browsers, detection of broken links is done.
3. Screenshots are made and are compared with older screenshots from the same URL to avoid CSS regressions.

There is no programming script required, therefore it can be used by less skilled human resources. It can be configured in shorter time, and thus is less expensive.

4.1.1 Mogo drawbacks

The drawbacks of this approach are evident when testing dynamic pages, whose content is easy to change. Applications which provide rich interaction options to the end user, and which state changes by interacting with various web components (calendar widget, table with sorting mechanism etc.), require a more subtle way of configuring what actions need to be done before the testing itself. Mogo is suitable for testing static web applications, not modern AJAX enabled applications full of CSS transitions.

The above mentioned drawbacks might lead to a bigger number of false negative test results when used with real data (any change, e.g. showing actual date may break testing), or to a bigger number of false positive test results when such a tool is used to test mocked data [1].

4.2 BBC Wraith

Wraith is a screenshot comparison tool created by developers at BBC News [7]. Their approach to visual testing can be described as:

1. Take screenshots of two versions of web application by scripting either PhantomJS 4.2.1, or SlimerJS[2] by another JavaScript framework called CasperJS 4.2.2 [20].
2. One version is the one currently developed (which runs on localhost[3]), and the other one is a live site.
3. Once screenshots of web page from these two different environments are captured, a command line tool `imagemagic` is executed to compare screenshots.
4. Any difference is marked with blue color in the created picture, which is the result of comparing two pictures (It can be seen in Figure 4.1).
5. All pictures can be seen in a gallery, which is a generated HTML site (It can be seen in Figure 4.2).

To instruct the BBC Wraith tool to take screenshots from the web application, one has to firstly script PhantomJS or SlimerJS to interact with the page, and secondly, create a configuration file, which will tell the PhantomJS instance which URLs need to be loaded and tested. PhantomJS script is one source of distrust for this tool, and therefore is introduced further.

1. Mocked data is data made up for the purpose of testing, so it is consistent and does not change over time
2. SlimerJS is very similar to PhantomJS 4.2.1, it just runs on top of Gecko engine, which e.g. Mozilla Firefox runs on top of. [10]
3. In computer networking, `localhost` means this computer. [11]

4.2.1 PhantomJS

PhantomJS [8] is a stack of web technologies based on headless[4] WebKit[5] engine, which can be interacted with by using its JavaScript API.

For the sake of simplicity we can say that it is a browser which does not make any graphical output, which makes testing with such an engine a bit faster and less computer resources demanding.

One can instruct PhantomJS to take a screenshot of a web page with the following script:

```
var page = require('webpage').create();
page.open('http://google.com/', function(status) {
  if(status === 'success') {
    window.setTimeout(function() {
      console.log('Taking screenshot');
      page.render('google.png');
      phantom.exit();
    }, 3000);
  } else {
    console.log('Error with page ');
    phantom.exit();
  }
});
```

When executing such a script it will effectively load `http://google.com/` web page, waits 3000 milliseconds, and after that, creates a screenshot to the file `google.png`.

In most environments it will be sufficient to wait those 3000 milliseconds in order to have the `www.google.com` fully loaded. However, in some resource limited environments, such as virtual machines[6], it is not necessarily enough. It will result in massive number of false negative tests. There is a need for a more deterministic way of finding out whether the page was loaded fully at a given time and taking of the screenshots itself can take place.

Another problem we noted in the previous script is its readability. It is written in a too low level way (one has to control HTTP status when loading a page). Secondly, there is a need to explicitly call `page.render('google.png');` in order to take a screenshot. Script which would test a complex web application would be full of such calls. Together with a poor choice of naming created screenshots (a user has to choose a name wisely), it might lead to problems when maintaining such a test suite.

PhantomJS API is wrapped by CasperJS, which is further described below.

4. Headless software do not require graphical environment (such as X Windows system) for its execution.
5. WebKit is a layout engine software component for rendering web pages in web browsers, such as Apple's Safari or previously a Google Chrome [9]
6. Virtual machines are created to act as real computer systems, run on top of a real hardware

4.2.2 CasperJS

CasperJS is a navigation scripting and testing utility written in JavaScript for the PhantomJS and SlimerJS headless browsers. It eases the process of defining a full navigation scenario and provides useful high-level functions for doing common tasks [20].

The following code snippet shows a simple navigation on a Google search web page. It will load *http://google.com* in a browser session, type into the query input string *MUNI*, and submit it.

```
casper.start('http://google.com/', function() {
  // search for 'MUNI' from google form
  this.fill('form[action="/search"]', { q: 'MUNI' }, true);
});

casper.run(function() {
  this.exit();
});
```

The problem with this script, which we identified, is its low-level abstraction of the browser interactions. It makes tests less readable, and thus more error prone when a change needs to be introduced.

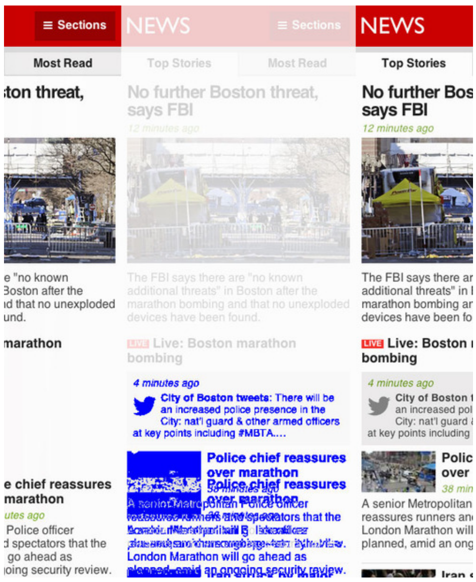

Figure 4.1: BBC Wraith picture showing difference in comparison of two web page versions [12]

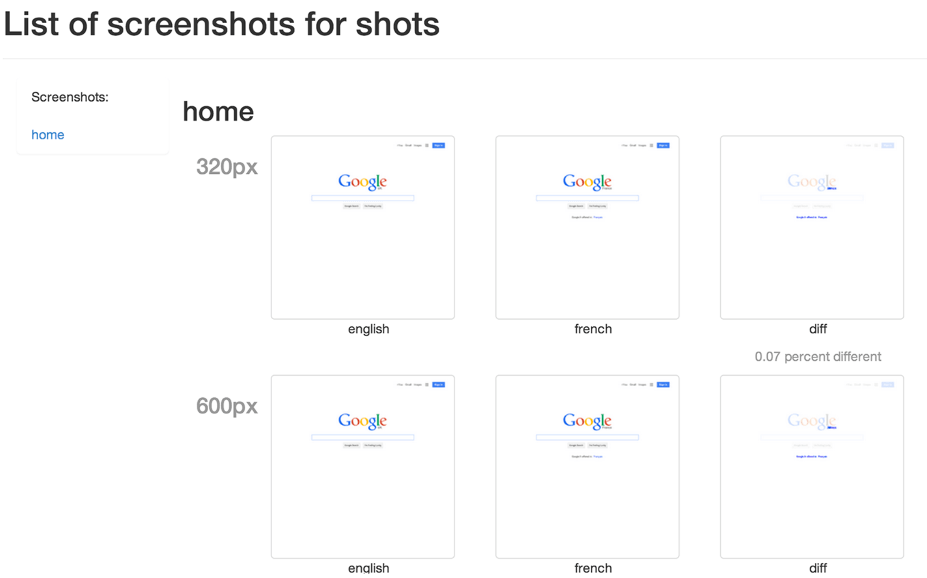

Figure 4.2: BBC Wraith gallery example [13]

4.2.3 BBC Wraith drawbacks

Two of the drawbacks were described in the previous sections, 4.2.1 and 4.2.2.

Another problem which might occur when testing with BBC Wraith is cross browser compatibility. As it supports only PhantomJS, one cannot assure that the page will be looking the same in all major browsers. The incompatibility comes from the fact that browsers interpret CSS rules differently, and because they have different JavaScript engines. Thus for example, a web page might look different in Google Chrome and Microsoft Internet Explorer, and PhantomJS will not register these issues.

4.3 Facebook Huxley

Another visual testing tool [15], supported by Facebook, Inc. [14], uses a similar approach in terms of comparing taken screenshots. The process of taking them and the process of reviewing results are different, though.

1. One creates a script which would instruct Huxley tool, which web pages screenshots should be taken from. Such a script might look like:

```
[test-page1]
url=http://localhost:8000/page1.html
```

```
[test-page2]
url=http://localhost:8000/page2.html

[test-page3]
url=http://localhost:8000/page3.html
```

2. One runs Huxley in the Record mode. This is the mode where Huxley loads the pages automatically in a fresh browser session, and a tester by hitting the Enter key instructs Huxley to take a screenshot. Screenshots are stored in a folder with a test name (one given in the square brackets in the example above), together with a JSON[7] file describing mainly how long should Huxley wait, when doing visual comparison, to have a tested web page fully loaded. Time is measured during this record mode.

3. To start visual testing, one has to run Huxley in the Playback mode. Huxley will start a fresh browser session, and will playback loading of the pages, with waiting for the pages to be fully loaded.

4. When there is a change in an application, Huxley will log a warning, and takes a new screenshot automatically. In continuous integration environments, one can instruct Huxley to finish with error, in case screenshots are different. In that case, one can run Huxley again with an option to take new screenshots (if the change is desired, and is not an error).

The main drawback of Facebook Huxley we can see is similar to BBC Wraith and that is its non-deterministic approach to waiting for a fully loaded web page. It is again a fixed amount of time, which can be different from environment to environment. The waiting time can be configured, however, it is still quite error prone, as for the first visual testing run 4 seconds would be enough, and for another run would not.

Secondly, it lacks a proper way of viewing results of comparisons, leaving only one option to check the command line output, together with manual opening of the screenshots. This would degrade cooperation among various QA team members, and it is harder to deploy in a software as a service cloud solution[8], where such a cooperation might take place.

4.4 Rusheye

Rusheye [18] is a command line tool, which is focused on automated bulk or on-the-fly inspection of browser screenshots and comparison with a predefined image suite. It enables

7. JSON stands for JavaScript Object Notation, a standard format that uses human readable format to transmit data objects [16]

8. Software as a service is on demand software, centrally hosted, accessed typically by using a thin client via web browser [17].

automated visual testing when used together with Selenium 1 project [19].

The process has subtle differences in comparison with the previous solutions. It consists of these steps:

1. Screenshots are generated, for example by Selenium 1 tool, while functional tests of web application are executed.
2. First screenshots are claimed to be correct (are controlled manually). They are called patterns.
3. After a change the web application under test, another run of Selenium 1 tests generates new screenshots. They are called samples.
4. Patterns and samples are compared, their visual difference is created, and the result is saved in an XML file.
5. The results can be viewed in a desktop application, Rusheye manager [22].

Rusheye has one very important feature, which other tools lack. It is the possibility to add masks on particular parts of the screenshots. Those parts are ignored when two screenshots are compared. It is a huge improvement to prevent false negative tests, as some variable parts (such as actual date, etc.) can be masked from comparison, and thus their change will not break the testing.

4.4.1 Rusheye drawbacks

The core of the Rusheye is only able to compare screenshots generated by some other tool. Integration with Selenium 1 is advised, however, functional tests written in Selenium 1 suffer from the same problems [21] as scripts written for BBC Wraith 4.2.2, and that is bad readability caused by their low level coupling with HTML and lack of higher abstraction.

Another problem we can see is only a desktop version of the tool for viewing results (Rusheye Manager). A cooperation on some test suite among QE team members and developers would be more difficult as they would need to solve persistence of patterns, samples and descriptor of the test suite.

4.5 Conclusion of analysis

All previously listed tools have some useful features, which we would like to be inspired with. However, all of the solutions lack something that we suppose to be an inevitable part of an effective automated visual testing.

Figure 4.3 summarizes the requirements we have for a visual testing tool, and the fact whether the tool satisfies the particular requirement.

Tests readability is a problem we discussed with a particular tool description. It is a level of abstraction over underlaying HTML, in which the application is written. It is quite a subjective matter, however, there are some clues by which it can be made more objective. For example, the way how tests are coupled with the HTML or CSS code. Because the more they are, the less they are readable [21]. A scale we used for evaluation supposes “insufficient” as lowest readability, which in the long term run of the test suite might cause a lot of issues.

By tests robustness we suppose a level of stability which have tests, when are executed with a particular tool. It means how likely there are false positive and false negative tests, whether they are caused by a not fully loaded page, or by dynamic data rendered on the page. If the robustness is low, one can find a red mark in a particular cell, a green one otherwise.

Cross browser compatibility issue deals with the ability to test web application across all major browsers 3.

By cloud ready features we mean whether a tool has a web based system for reviewing results, and thus enables cooperation across QA team members and developers of the particular software.

Continuous Integration friendliness in this context means the fact whether a tool is suitable for usage in such systems. It actually means whether output of the tool is clear enough, how much work a tester would be required to do manually to deploy such a tool in a test suite. Whether testers would need to review just logs to find visual testing failure, or whether it will be somehow visible, e.g. whole test would fail.

	Test script readability	**Tests robustness**	**Cross browser compatibility**	**Cloud ready**	**CI friendliness**
Mogo	*not applicable*	✗	✓	✓	✓
BBC Wraith	*insufficient*	✗	✗	✓	✗
Facebook Huxley	*insufficient*	✗	✓	✗	✓
Rusheye + Selenium 1	*insufficient*	✓	✓	✗	✓

Figure 4.3: Existing solutions features comparison

As Figure 4.3 shows, none of the tools met our requirements fully. Therefore, we decided to proceed with developing a new tool, which would address all issues, and which would integrate existing parts of the solutions when it is reasonable. The creation of a new process which would enable effective usage of such a tool by QA team members is inevitable. The following chapters describe this new tool and the new process.

5 New approach

Each tool by its definition introduces a process for a visual testing. While we recognized shortcomings (described in chapter 4.5), we realized a need for a different approach to visual testing. The approach came from two and a half years of developing and maintaining the functional test suite [1] for RichFaces project [2].

5.1 Hypothesis

It should be enough to have just a functional test suite for an end to end testing of an application. Scripts from functional testing interact with the application sufficiently, therefore, another script taking screenshots during such interactions is redundant.

This redundancy is expensive because quality assurance teams need to maintain more test suites. A change in the application needs to be introduced in more places, which allows errors to sneak in test suites.

At the same time we do believe that a script for functional testing should be abstracted from a visual testing as much as possible. This means that explicit calls to methods which take screenshots should be optional. A functional test script should take screenshots automatically during testing, in each important state of an application. By this, we will achieve better readability of the functional tests' scripts.

There should certainly be an option to make screenshots explicitly, however, a need to facilitate such option should be sporadic. This will be achieved by fine-grained configuration options. A tester should be able to configure on a global level, meaning for all tests from one place, as well as on test level, in which situations a screenshot should be taken.

The base of screenshots, which will serve as a reference for a correct state of the web application, will be made automatically during the first run of the created tool. Screenshots should be made available automatically for all interested parties (developers, testers, etc.).

A viable solution seems to be introducing a web interface as a system for managing results of visual testing comparison. In this system (called a manager in this thesis), a user should be able to decide about the results of visual testing. More particularly assess the results, whether there is a bug in application or not. This web interface will foster cooperation between interested parties.

By following the above mentioned principles, we will achieve greater effectiveness of a quality assurance team. More particularly, we will massively decrease the amount of time they need to spend on manual testing.

---

1. The RichFaces test suite is available at https://github.com/richfaces/richfaces-qa
2. RichFaces is a component library for Java Server Faces web framework [23]

5.2 Process

The whole solution for visual testing would need to include a reaction to these problems:

1. Executing a visual test suite for the first time to generate patterns, which new screenshots in subsequent tests executions will be compared with.

2. Executing the visual test suite for the second and more times, to generate new screenshots, which are called samples in this thesis, for comparison with patterns generated in the fist run.

3. Review results, and take an action when there is a false negative result, or bug in the application.

4. Executing the visual test suite when a new test is added, deleted, or otherwise changed.

For simplicity, in the first stage of the development, we suppose the third problem will be solved by re-executing the whole test suite again, as it is done in the first run of the test suite. The overall process can be described with the following subprocesses.

5.2.1 First execution of functional tests

Figure 5.1 denotes steps needed to generate patterns. It is a prerequisite for the visual testing.

Screenshots are generated during the execution of the tests. If all functional tests pass, those screenshots can be claimed as patterns, and are uploaded to a storage. Screenshots generated in tests which failed are ignored: they are not included in the visual testing. Such functional tests need to be fixed, made to be passing, to include them to visual testing.

An optional part is reviewing of the screenshots. If there is any issue with patterns, they should be thrown away, and tests will be rerun.

If patterns are correct, a tester can proceed with subsequent runs of the test suite to start the actual visual testing.

5.2.2 Subsequent execution of visual testing

Figure 5.2 shows how subsequent execution of visual testing together with functional testing would look like. The first step is the same as in the previous process, functional tests are executed, and screenshots are taken. Secondly, patterns are downloaded from the storage, and after that the actual visual testing can start.

Newly created screenshots, called samples in this thesis, are pixel to pixel compared with downloaded patterns. The result is stored in a descriptor file, and differences between the particular pattern and sample are visually stored in a generated picture.

If any differences are found, generated pictures together with their corresponding samples and patterns are uploaded to a remote storage.

Users should be able to review the results from now on, where they will find the particular run according to a time stamp when the run was executed. They should be able to asses the results, with displayed triplet, consisting of the pattern, sample and their difference. They should be able to say whether it is a false negative test result, in which case they should be able to take an action to prevent such results in the future. One of such actions can be applying masks, which is further described in chapter 4.4.

The tool should be configurable, so such false negative results are rather sporadic, instead, failed visual comparison tests should reveal a bug in the application under test. In that case it is in the user's responsibility to file such a bug in a corresponding issue tracker. The generated pattern, sample and difference can be used as a useful attachment to an issue report, which would better describe the actual and expected result.

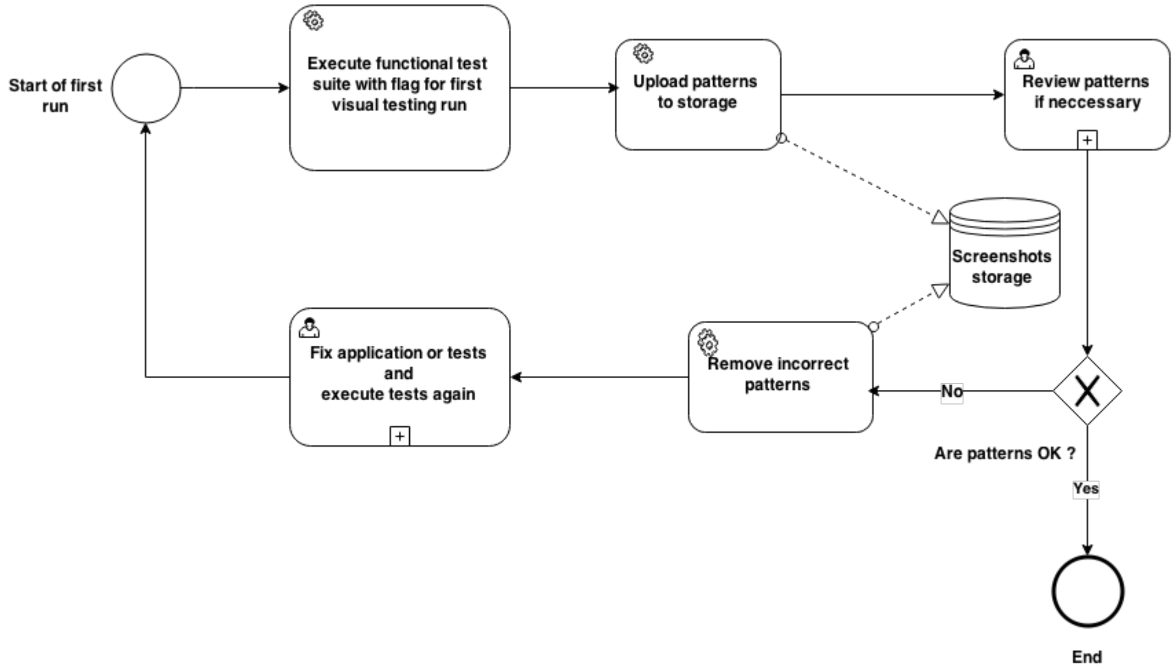

Figure 5.1: Process to generate patterns during the first execution of functional tests.

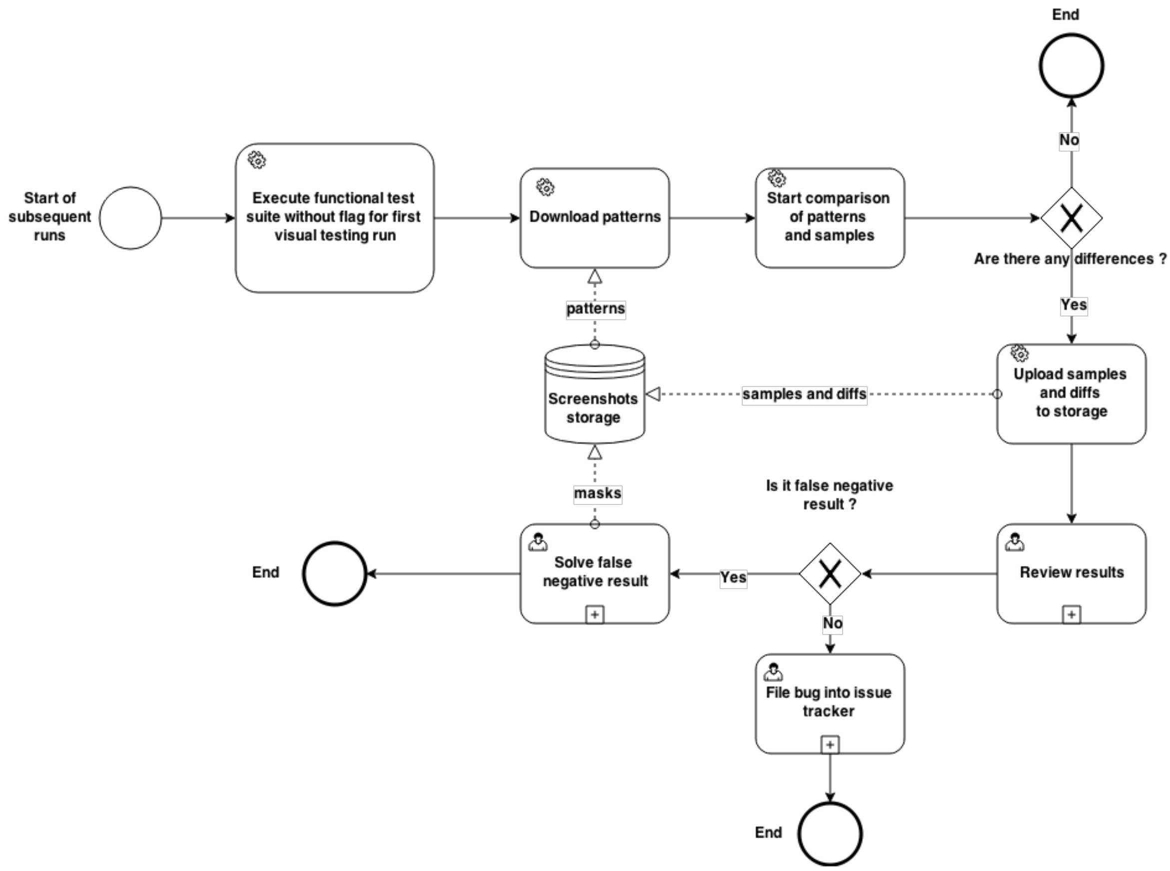

Figure 5.2: Subsequent execution of visual testing together with functional tests.

5.3 Analysis of useful tool output

To create a tool which is widely accepted by a community of testers, who are interested in a visual testing, we have to take usefulness of the tool as a priority. Such a tool has to show results of the visual comparison in such a way that it is a pleasure to use it. At the end of the day, the amount of time spent with using the tool has to be dramatically smaller than doing the manual visual testing.

Therefore, we conducted a research among IT experts, which took a form of brainstorming on the visual testing web application user interface. The purpose of such a web application is to show results and allow a tester to take an immediate action over them.

The brainstorming took place on Arquillian developer forum [24], which is daily accessed by hundreds of users. At the time of writing this thesis, it has more than 355 views and 11 replies. We can say that there are users interested in such a tool, and we have gained valuable feedback, together with many interesting ideas about what features such a web application for reviewing results should have.

We started with a description of the tool together with proposals for graphical user interface design (mockups), and asked the IT community for their opinions, what they miss on such an interface and what is, on the other hand redundant.

You can see created mockups in appendix A. The web application is logically divided into three screens. The first one (can be seen in appendix A.1) shows the front page of the application. Its purpose is to show test suites, which are groups of tests written for a particular application under test. Together with a web browser they are tested on, they represent a test suite for visual comparison testing.

The second screen (can be seen at A.2) shows a particular executions of the test suite. These test suite runs are unambiguously named according to the time and date they were executed.

The third created mockup shows comparison results for a particular test suite run. The comparison result consists of three screenshots. A pattern, created during the first run of the test suite. A sample, which was taken during this particular test execution, and a picture called diff, which denotes differences between the pattern and the sample.

On the third mockup you can also see two buttons. Their purpose is to allow a tester to take an immediate action upon results. The pattern is correct button is used when the result of the comparison is a false negative, or when the result denotes a bug in the application. The sample is correct button is used when there is an anticipated change in the application under test. In that case the newly created sample should be used as the pattern in the following comparisons.

Based on users' insight, we complemented the final web application with the following features:

- Indicate number of failures versus total number of tests.
- Revision number of application under test. For example Git[3] commit id.
- Display together with screenshots also the name of the functional test, and a name of the test class the test resides in.
- The triple pattern, sample, diff should be displayed in a way that it is not hard to spot the actual difference. Putting them side by side is not a good option. We had to think out some different approach.
- Together with time stamp of the run, we should also show an order number of the run.

3. Git is source code version system, http://git-scm.com/

6 Implemented tool

To support our hypothesis and the process we figured out, a set of tools needed to be created. As we did not want to reinvent a wheel, where it is feasible we used an already existing tool and integrated it to the final result.

Figure 6.1 depicts the component diagram of the implemented solution. It is only a high level picture of the overall solution. Particular components are described in detail in the following chapters.

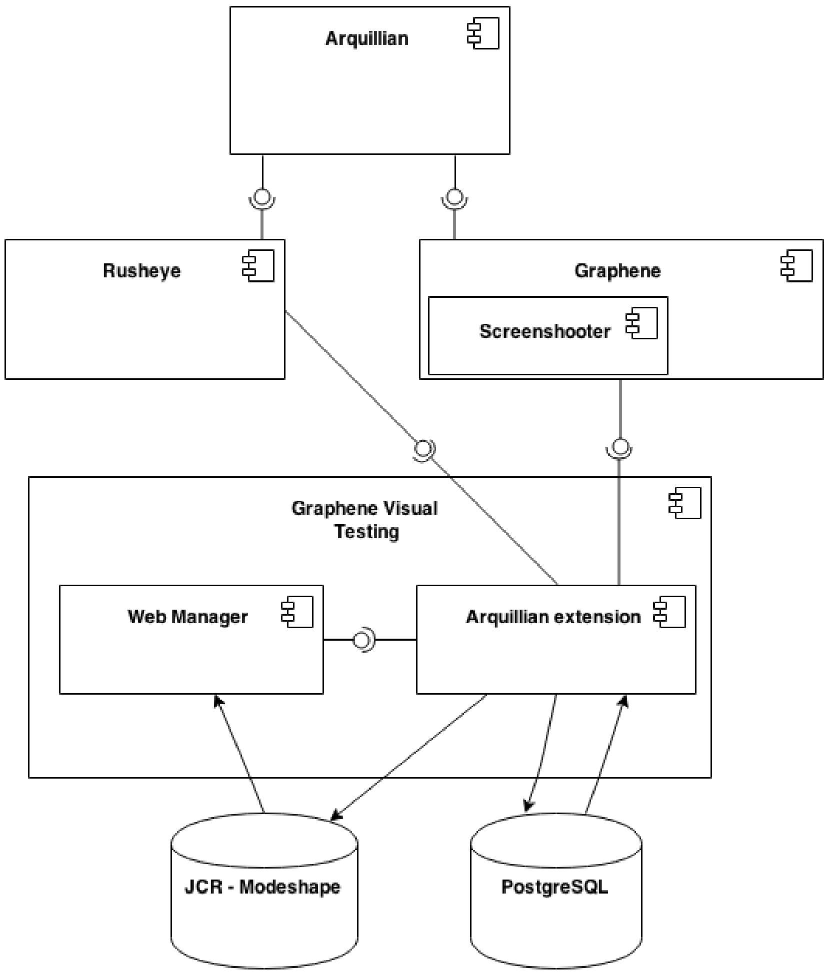

Figure 6.1: Component diagram of the implemented solution for visual testing.

6.1 Arquillian

Arquillian[1] is a testing framework, which facilitates integration testing. It automatizes various parts of the integration testing, which allows testers to develop actual tests.

It can, for example, start an application container before testing and stop it after the test's execution is done, it can deploy the application under test to that container, populate the database with testing data, start web browsers, test on mobile devices, and provide useful output with videos and screenshots from testing.

In other words, it manages the life cycle of all integration components your application integrates with.

It has very nice architecture, which resembles an event based machine. It is easily extensible, as it supports Service provider interface pattern [25]. It can cooperate with all major browsers, which makes it a cross browser solution.

All these features made it a good candidate to integrate our solution with. We did not need to develop any feature for this project.

6.2 Arquillian Graphene

Arquillian Graphene is an extension to Arquillian, thus it fully supports its life cycle of test. Its main purpose in integration tests is to control a web browser. By using its API, a tester is able to, for example, click on the button on the web page, write some characters into text inputs or otherwise interact with a web application under test.

Its core is a wrapper over well known project Selenium (also known as WebDriver) [2]. It is a W3C standard[3], which guarantees stability, making it a good candidate to build our solution with.

Graphene's type safe API[4] supports design patterns such as Page Object[5] or Page Fragment[6].

Those features form an API with a high abstraction level, and encourage a tester to develop more readable tests. Those are attributes which the already existing solutions lack (see 4.5 for more information).

1. More information about Arquillian at http://arquillian.org/
2. More information about Selenium project at http://www.seleniumhq.org/
3. WebDriver working draft standard available at http://www.w3.org/TR/webdriver/
4. Type safe API in Java programming language enforces obeying various rules before compilation of source code, thus decreases a change to introduce an error.
5. Page Object pattern encapsulates a web page into object, makes tests more readable [26].
6. Page Fragments pattern divides a web page further into reusable components which encapsulates a web widget functionality, making tests more reusable and readable [27].

6.3 Graphene screenshooter

For the purpose of visual testing, we needed to implement one addition to Arquillian Graphene. A component which would facilitate taking of screenshots of the application under test in a uniform manner. Developed addition implement a common interfaces defined in Arquillian Recorder extension [7].

We had two main requirements on this screenshots taking extension, which came directly from the analysis of already existing solutions (see chapter 4):

1. A tester should be able to configure this extension so it takes screenshots automatically. A script for a functional test should be unaware of such screenshots taking behavior. It will stay clean and more readable (see chapter 4.5 to see more background information for this requirement). The configuration should be done on a global level, in other words for the whole test suite.

2. The tester, on the other hand, should be able to explicitly enforce taking of a screenshot at any place in the functional test script. This should be just an optional feature, used rather sporadically.

The second point is developed in all the existing solutions. To enhance readability we required from our solution to have implemented also point number one.

The global configuration is done where all configuration takes place for Arquillian framework. It is in `arquillian.xml` file.

Listing 6.1: Example of screenshooter configuration in arquillian.xml

```
<extension qualifier="screenshooter">
  <property name="takeBeforeTest">true</property>
  <property name="takeAfterTest">true</property>
  <property name="screenshotType">PNG</property>
</extension>
```

In the example configuration 6.1, one can see that screenshots are automatically made after two events: before the test, which is effective just after the web application is loaded in a browser, and after the test.

We also started with an experimental feature, which is not fully available, and that is an option to take screenshots after every action made by Selenium in browser[8].

7. Arquillian Recorder is an extension which among the other things defines interfaces for screenshot taking, and video recording from test executions. We have cooperated on defining this common interface as well. More information at https://github.com/arquillian/arquillian-recorder

8. See more information about `takeOnEveryAction` at https://github.com/arquillian/arquillian-graphene/tree/master/extension/screenshooter

6.4 Rusheye

We have already described some of the Rusheye features in chapter 4.4. It is important that it is only a command line tool, so integration with such a tool would require either executing its binary, packaged in a `.jar` file, or calling its main[9] method. This is not a good software design because it is hardly extensible and error prone when there is a change introduced in either of the integrated parts.

Therefore, we decided to introduce an integration layer in Rusheye, which would cooperate with Arquillian event based system. It is mostly a realization of the Observer pattern[10].

Rusheye observes events such as: `StartCrawlingEvent`, or `StartParsingEvent`, and reacts according to them. It starts crawling of patterns and creates a test suite descriptor (XML file which describes where the particular screenshoots for a particular functional test are stored). This is done after the first run only (see chapter 5.2.1). In subsequent runs `StartParsingEvent` is fired, and Rusheye starts the actual visual comparison, it compares patterns with samples on a pixel to pixel basis.

After crawling or parsing is done, it fires `CrawlingDoneEvent` or `ParsingDoneEvent` events respectfully, so that other integrated modules can wire up.

In result, such event based architecture makes a loosely coupled system, which is easily extensible.

Listing 6.2: Example of StartParsingEvent observer

```
public void parse(@Observes StartParsingEvent event) {
    // Initialization code ommited
    parser.parseFile(suiteDescriptor, event.
        getFailedTestsCollection());
    parsingDoneEvent.fire(new ParsingDoneEvent());
}
```

6.5 Graphene visual testing

It is a project which has two purposes.

- to integrate and control Rusheye with Arquillian;
- and to provide a way for reviewing results of visual comparison.

For those two purposes, two sub-projects were created, and are described bellow.

9. Java main method is an entry point to the program.
10. Observer pattern - `http://en.wikipedia.org/wiki/Observer_pattern`

6.5.1 Arquillian extension

As it was written previously, this extension is focused on controlling Rusheye and storing or retrieving created screenshots.

As the 6.3 listing shows, it wires up to Arquillian life-cycle, as it listens to the `AfterSuite` event. If it is a first execution of the test suite, then it just fires the `StartCrawlingEvent` event, which is observed by Rusheye. After crawling is done, it stores the created suite descriptor and screenshots to a Java Content Repository (JCR - see chapter 6.6).

If it is not a first run, it firstly downloads screenshots and the suite descriptor, and then fires a `StartParsingEvent`, which is again observed by Rusheye.

The result of parsing (XML file describing which patterns and which samples were different, and a path to the created diffs) are also uploaded to a JCR.

Listing 6.3: AfterSuite observer to controll Rusheye

```
    public void listenToAfterSuite(@Observes AfterSuite event) {

        String samplesPath = scrConf.get().getRootDir()
                                             .getAbsolutePath();

        if (visualTestingConfiguration.get().isFirstRun()) {
            crawlEvent.fire(new StartCrawlinglEvent(samplesPath));
        } else {
            String descAndPtrDir = serviceLoader.get()
                    .onlyOne(DescriptorAndPatternsHandler.class)
                    .retrieveDescriptorAndPatterns();

            startParsingEvent.fire(new StartParsingEvent(
                    descAndPtrDir,
                    samplesPath, failedTestsCollection.get())
                    );
        }
    }
```

A communication between this extension and a JCR is done via JCR Rest API. To issue a http request, we use Apache HttpComponents project[11].

6.5.2 Web application to view results

It is a web application for reviewing results, but also for an active change of the visual testing configuration.

11. Apache HttpComponents - `http://hc.apache.org/`

The application back-end is developed with the use of Java Enterprise Edition[12] stack. It includes using technologies like: Java Persistence API [13] plus PostgreSQL[14] for a persistence layer. Enterprise Java Beans[15] for controllers code (Model-Viewer-Controller pattern[16]). It exposes Java API for RESTful Services (JAX-RS[17]) endpoints to enable communication with the Arquillian extension (see chapter 6.5.1), and with its client part (HTML front end) in a RESTful way.

The application is deployed on a WildFly application server[18]. We chose this server because it is an open source software, backed by a huge community of users, and by a big enterprise, Red Hat[19]. Another very important thing for us was its speed and availability in cloud environments (see chapter 7.3).

The front end is developed with the use of AngularJS, which is a MVC framework for creating a Single Page Applications (SPA[20]. It is complemented with Twitter Bootstrap framework, to provide a visually pleasant UI.

The design of particular screens is in accordance with the analysis described in section 5.3. You can find screenshots from the application in the [DOPLNIT APPENDIX].

For better convenience, and to verify the possibility to deploy the application on the cloud, we have deployed it on Platform as a Service, OpenShift by RedHat cloud[21].
Please see chapter 7.3 for more information about the deployment (how to log in, etc.).

The following figure 6.2 depicts the possible use cases with the web application, to give a better picture of what can be achieved with this application.

The most important here are `Reject Pattern` and `Reject Sample` functionalities, as they allow a tester to react to results. `Reject Pattern` will change in the storage for all screenshots the pattern with sample. This functionality is used when there is an expected change in the application, and we want to use the new sample as a pattern in the subsequent runs.

`Reject sample` is used when the results is either a false negative (in which case the sample is just deleted, or when there is a bug in the application under test, in which case a tester is supposed to create a bug report. The created sample and diff will serve as a good help when describing the bug.

There are planned extensions to this basic functionality, described in section 8.

12. Java Enterprise Edition - `http://www.oracle.com/technetwork/java/javaee/overview/index.html`
13. JPA - `http://en.wikipedia.org/wiki/Java_Persistence_API`
14. PostgreSQL database - `http://www.postgresql.org/`
15. Enterprise Java Beans - `http://en.wikipedia.org/wiki/Enterprise_JavaBeans`
16. Model-Viewer-Controller - `http://en.wikipedia.org/wiki/Model-view-controller`
17. JAX-RS - `https://jax-rs-spec.java.net/`
18. WildFly application server - `http://www.wildfly.org/`
19. Red Hat - `http://www.redhat.com/en`
20. SPA - `http://en.wikipedia.org/wiki/Single-page_application`
21. OpenShift by RedHat - `https://www.openshift.com/`

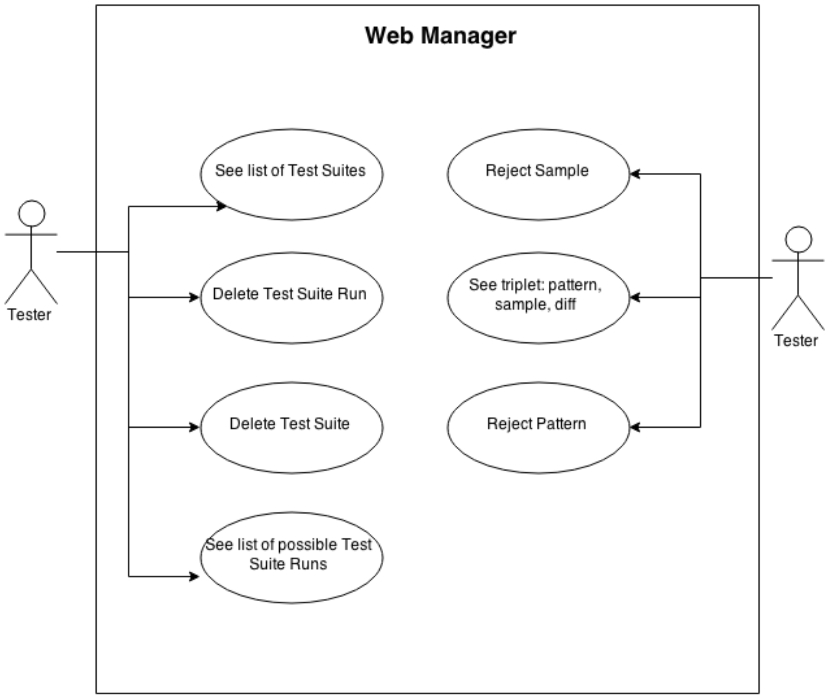

Figure 6.2: Use case diagram for Web Manager web application.

6.6 Storage for images

As we had to think wisely when developing UI for Web Manager, to enable successful emplyment of the tool among a community of testers; in the same way, we had to think beforehand about storage for created images (screenshots).

We have these requirements for storing the images:

- The chosen solution has to provide a way for storing a large amount (hundreds) of pictures;
- it should be a scalable solution;
- it should provide a solid performance when uploading and downloading stored pictures;
- it should be capable to ensure security for data, authentication and authorization when accessing pictures;
- it should be a cloud friendly solution.

We were choosing from these options, which we chose by a careful analysis of patterns:

1. Store screenshots as Binary large objects (BLOBs [22]) in a relational database, such as PostgreSQL.

2. Store screenshots in a file hosting service such as Dropbox[23], or Google Drive[24]. Store just URLs to relational database.

3. Store screenshots in a Java Content Repository (JCR[25]). Store just URLs to a relational database.

The number one option has some obvious advantages. Databases are a superior solution where transactional integrity between the image and metadata are important. Because it is more complex to manage integrity between database metadata and file system data, and within the context of a web application, it is difficult to guarantee that data has been flushed to disk on the file system [28].

However, the way to store pictures in database as BLOBs has some disadvantages too. Database storage is usually more expensive than file system storage; database access cannot be accelerated for example by configuring a web server to use the system call to asynchronously send a file directly to a network interface, as it can be done for file system access [28].

Option number two seems to be a viable solution for smaller test suites. It does not suffer from the problems which a database suffers from. In a the later stage of development of our tool we would like to provide this option to users of our tool. For now, we would like to provide a more flexible solution, which does not vendor lock in to some providers, a solution which is free of charge when big storage space is required.

Therefore, we chose option number three as the pilot way for storing screenshots. JCR is a specification and also a Java API, thus a best fit for our Java based application. We liked what kind of data and access patterns JCR are very good at handling [29]:

- Hierarchical data;
- files;
- navigation-based access;
- flexible data structure;
- transactions, versioning, locking.

Data we need to store is naturally hierarchical. See figure 6.3 to see in what hierarchy we need to store generated screenshots, XML suite descriptor, and XML result descriptor (configuration files for Rusheye module).

22. BLOB - `http://en.wikipedia.org/wiki/Binary_large_object`
23. Dropbox - available at `https://www.dropbox.com/`
24. Google Drive - available at `https://www.google.com/drive/`
25. JCR - specification available at `https://jcp.org/en/jsr/detail?id=283`

JCR are good at storing files, which is exactly what we are looking for, as pictures will be the main artifacts which we need to persist.

Navigation based access is often used for application dealing with hierarchical data. In our application we often need to work only with subset of the hierarchy - for example to display triplet pattern, sample, and diff for a particular test suite run.

We wanted to provide a flexible data structure. By this tool stays extensible to other approaches and new features for later development (see section 8).

Transactions, versioning, and locking are sweet points which will be used in later development. They are important, as we want our solution to be scalable (scale out[26]), and we want to allow cooperation of testers on a particular test suite (their actions on results will need to be transactional).

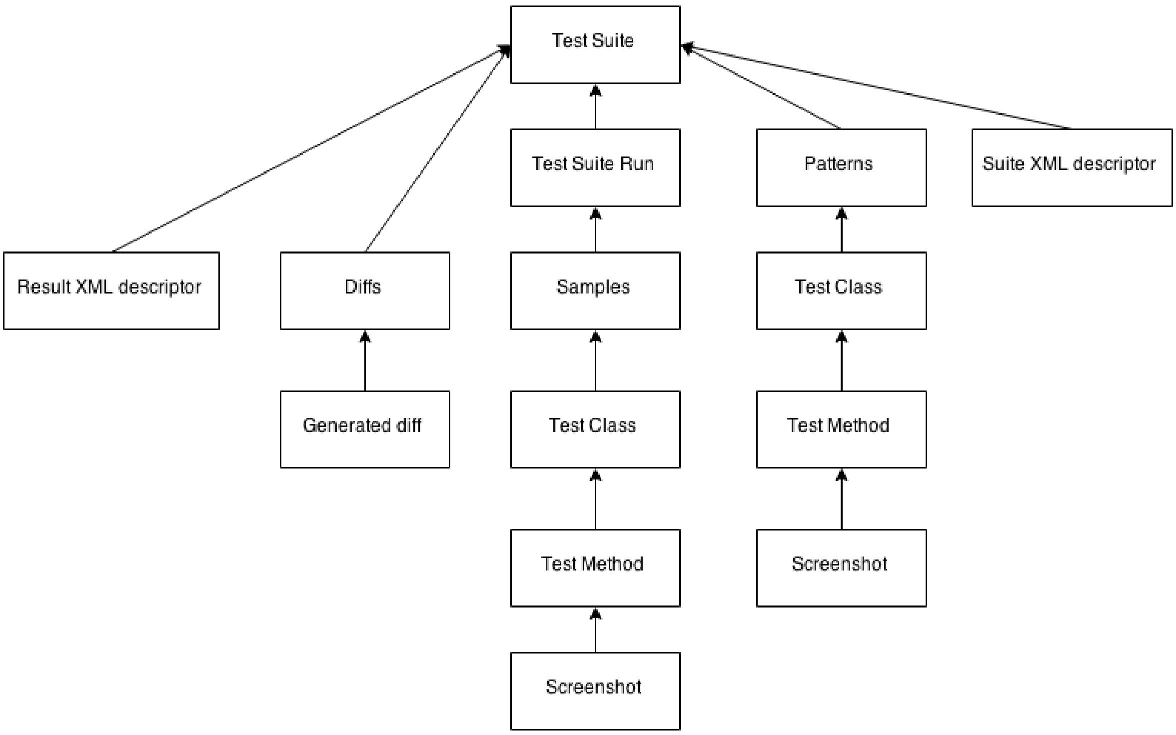

Figure 6.3: The hierarchy of nodes in our JCR deployment.

We chose JCR implementation ModeShape[27]. There are number of advantages we saw, when comparing this implementation with reference implementation of JCR, the Apache

26. Horizontal scaling (scale out) - `http://en.wikipedia.org/wiki/Scalability`
27. JBoss ModeShape by Red Hat - `http://modeshape.jboss.org`

Jackrabbit[28]: The development is backed by Red Hat, the same company we chose the application server from (see 6.5.2). They cooperate well with each other, and there is plenty of documentation on how to integrate those two systems. ModeShape also by default exposes a RESTful API for accessing and modifying the content of the repository. We are utilizing this feature in the Web Manager (see 6.5.2), when screenshots which we want to display in the client browser, do not have to be firstly streamed to the WildFly application server, and then served to the client, but they are directly streamed to the client browser, as it knows the URL of the screenshot.

For the future development we like its support for WebDAV protocol[29], and possibility to cluster multiple ModeShape instances.

28. Apache Jackrabbit - `http://jackrabbit.apache.org`
29. WebDAV protocol - `http://en.wikipedia.org/wiki/WebDAV`

7 Deployment of tool and process

After implementing the tool, to prove or disprove hypothesis from section 5.1 we need to deploy the tool and the process (see section 5.2) on a real application. The following chapters describe such deployment, real world use cases and best practices when using our tool.

To have a better picture about the systems and environments in which each part of the visual testing will be executed, the following sequence diagram was created:

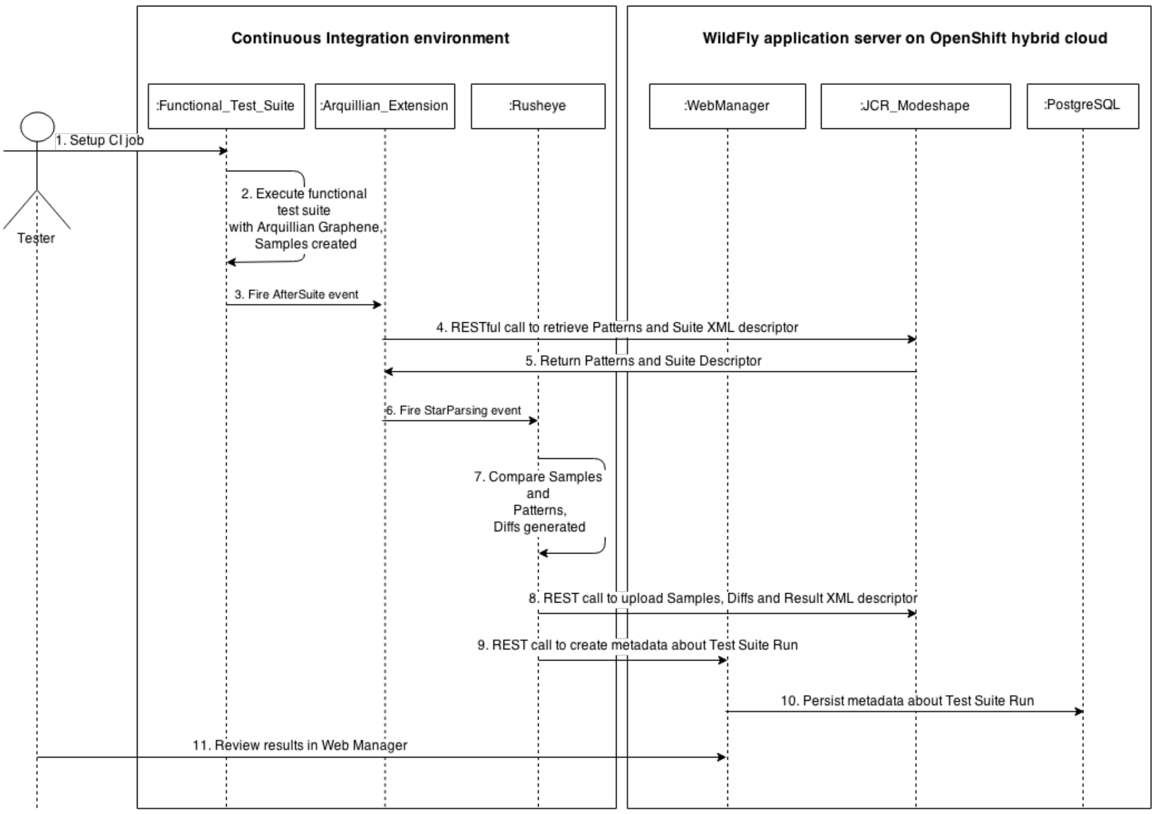

Figure 7.1: Whole visual testing solution sequence diagram.

There are omitted processes which are not important for visual testing, or processes which describe how data is transported into the Web Manager, already described in section 6.6.

7.1 Actual visual testing

We chose RichFaces Showcase application[1] and its functional test suite[2] to try our tool and process on. It is a Java EE application, with many libraries needed to be deployed alongside with the application. One of them is RichFaces core library.

We chose a real world use case to try the tool and the process. It is: to verify visual state of the application after an upgrade of the core RichFaces library, from version 4.5.0.Final to 4.5.1.Final. We proceed as follows:

1. Web Manager was deployed to OpenShift cloud on a WildFly 8.2.0.Final cartridge (see chapter 7.3).
2. Functional test suite was run. It tested RichFaces Showcase application with core libraries of 4.5.0.Final version.
3. During first execution of tests, patterns were created and uploaded to the cloud.
4. Test suite was run several times to stabilize visual testing.
5. When the visual testing was stable enough (there were no more than 4 differences), the test suite was run in a way that it tested RichFaces Showcase with core libraries of the 4.5.1.Final version.
6. Results were analyzed.

We deployed the Web Manager to the cloud, as that would be its most probable environment. It made conditions for testing more difficult because we ran the test suite from Europe, and the servers where the Web Manager was deployed, are located in the US. Moreover, we chose servers with low RAM (512MB) and with reduced CPU performance. The speed of the Internet was 1,1 Mbit/s for download, and 0,6 Mbit/s for upload. These limited conditions were chosen on purpose because if the tool would work sufficiently in such conditions, it would have even better performance in better conditions.

After the test suite was run for the first time and patterns were created, we ran it several times (10) to stabilize results. Initially, there were many differences (about 30 out of about 400 visual comparisons) found by our tool. All of them were false negative results. The reason why we got lot of false negative results was hidden mainly in random data by which the application was filled. Particularly, rows for tables consist of random data in the application

1. RichFaces Showcase - Screenshot from application is shown in figure 3.1. Source code is available at `https://github.com/richfaces/richfaces/tree/master/examples/showcase`. Application is hosted at `http://showcase.richfaces.org/`
2. Test suite is written in Arquillian Graphene framework, and the source code is available in the same repository as the application itself.

(with each deployment on the server there is different data shown in table components[3]). Secondly, there were unstable test results because of timing issues. For example, there is a slightly different delay for RichFaces tooltip[4] component to be shown.

Therefore, we have decided to improve the stability of visual testing by introducing a way to exclude some tests or whole test classes from visual testing (they are still run in functional tests, as they are stable enough there). We introduced a JAVA annotation `@VisuallyUnstable` to the Graphene visual testing extension API (see 6.5.1). Listing 7.1 depicts how one particular test can be excluded from visual testing, and listing 7.2 shows how a whole test class, and all its testing methods can be excluded from visual testing at once.

Listing 7.1: Exclude functional test from visual testing by annotating it with @VisuallyUnstable

```
@Test
@VisuallyUnstable
public void testClientTooltipWithDelayComponent() {
  //actual testing code ommited
}
```

Listing 7.2: Exclude whole test class from visual testing by annotating it with @VisuallyUnstable

```
@VisuallyUnstable
public class ITestExtendedDataTable {

  @Test
  public void testFiltering() {
  }

  @Test
  public void testSorting() {
  }

  //and other tests ommited
}
```

The main part of the real world use case we wanted to test was upgrading of the RichFaces core library to the 4.5.1.Final version. We chose this use case, because it was mainly in release

3. RichFaces table component in Showcase - `http://showcase.richfaces.org/richfaces/component-sample.jsf?demo=dataTable&sample=arrangeableModel&skin=blueSky`
4. RichFaces tooltip component - `http://showcase.richfaces.org/richfaces/component-sample.jsf?demo=tooltip&skin=blueSky`

testing process (see chapter 3.1) when manual testing was conducted and there was a new version of RichFaces core library available.

The test suite was run in a way that it tested the RichFaces application with 4.5.1.Final libraries. Initially as we expected we got many differences (more than 200). The reason was that the Showcase application shows on the bottom, in web application's footer its version. The version is visible on all screens, thus there were so many differences. Secondly, there were some expected changes, due to fixes in the application[5].

Figures 7.2, 7.3 show generated diffs for these expected changes. Because they affect a lot of visual comparisons, we had to do something to decrease the number of false negative tests. We used the Arquillian Rusheye (see section 4.4) feature of masks to do it. It is a way to make arbitrary parts of the web page excluded from visual comparison. Currently, Rusheye supports the so called selective alpha masks, which are pictures with a transitive (alpha) layer and also with non transitive parts which cover the parts of the patterns and samples we would like to exclude from pixel to pixel comparison [30]. Our tool is currently not supporting the feature of masks, as we did not recognize it as an inevitable part of proving/disproving of our hypothesis (see chapter 5.1). It is indeed a very useful feature, which we will add to the tool in the next developmental stages (see section 8).

7.1.1 Results

Finally, after applying all the previously mentioned methods for decreasing the number of false negative test results, we came to an acceptable number of generated diffs (4). Reviewing such results would take minimum time (5 min maximum) for a tester familiar with the application.

Indeed, we had to add to the final result the time which we spent with applying masks, and excluding unstable tests from visual testing (30 min). However, most of these activities needs to be done only once for the test suite. It can be reused in subsequent releases of the RichFaces framework (the mask to cover the changing version of RichFaces will remain the same, the same for the mask for covering the menu with components).

It is a very good result when taking into consideration the following facts:

- Visual testing was done automatically, so during this time the human resources (testers) were able to do some more intellectually demanding testing than just exploratory manual testing, which means clicking through the application.
- This manual testing needs to be done for all major browsers separately. Each manual testing takes approximately 30 min. If there are 5 major browsers currently used (see section 3), it is about 150 min of manual testing.

5. These changes are available in an online repository:
1. `https://github.com/richfaces/richfaces/commit/203a7421f7daa594ce8d16c810a379a75dafa805`
2. `https://github.com/richfaces/richfaces/commit/715607080a4c15bff90af6546353e4b21a8391ee`

- If we suppose that reviewing of automatized jobs would take 25 min (5 min for each), we can say that we save 125 min of time for quality assurance team human resources. It is about 83.33 % improvement.

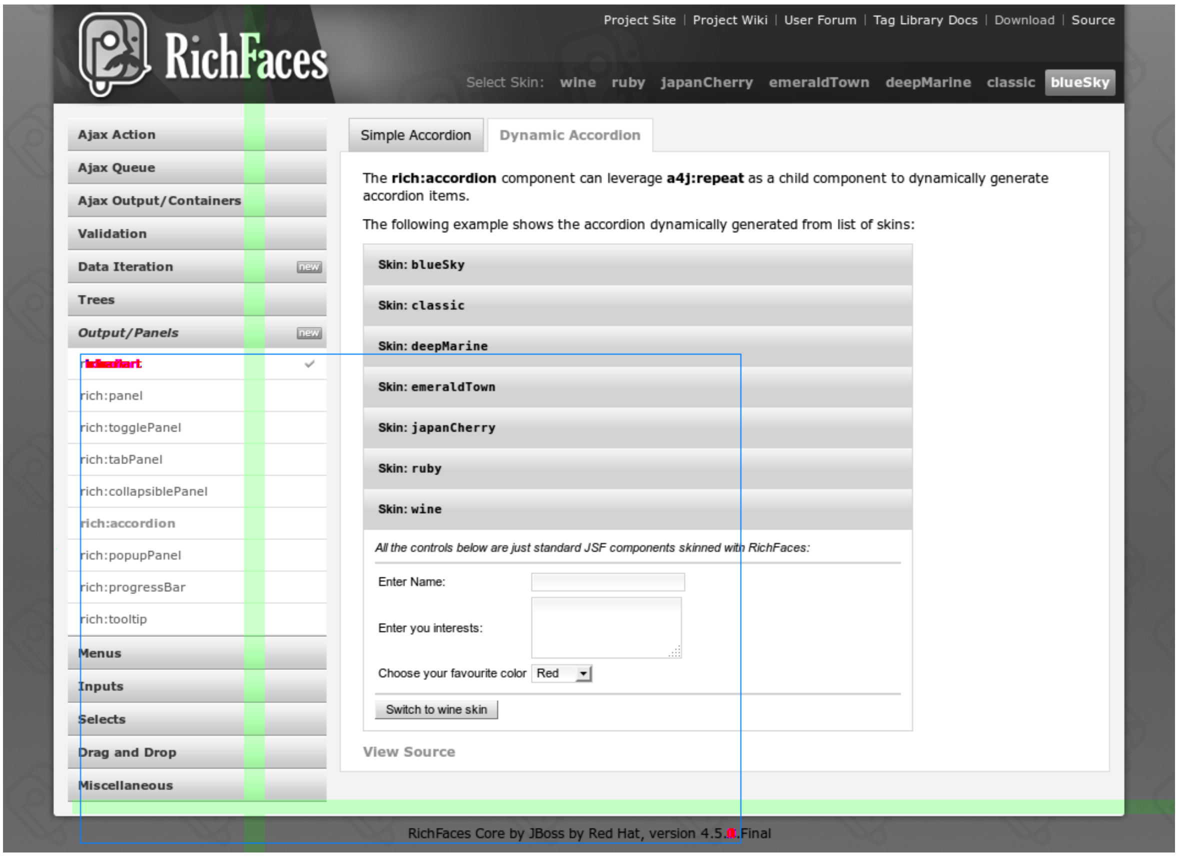

Figure 7.2: RichFaces chart component renamed in application menu.

7.2 Usage with CI

One of our requirements for the tool is that it can be used within Continuous Integration systems. This is quite an easy condition to meet. The only requirement is that the process of visual testing needs to be automatized, so that it will not need user interaction.

As we built our solution as an integration with Arquillian Graphene, this requirement is fulfilled automatically. The required way of running Arquillian Graphene tests is via Apache

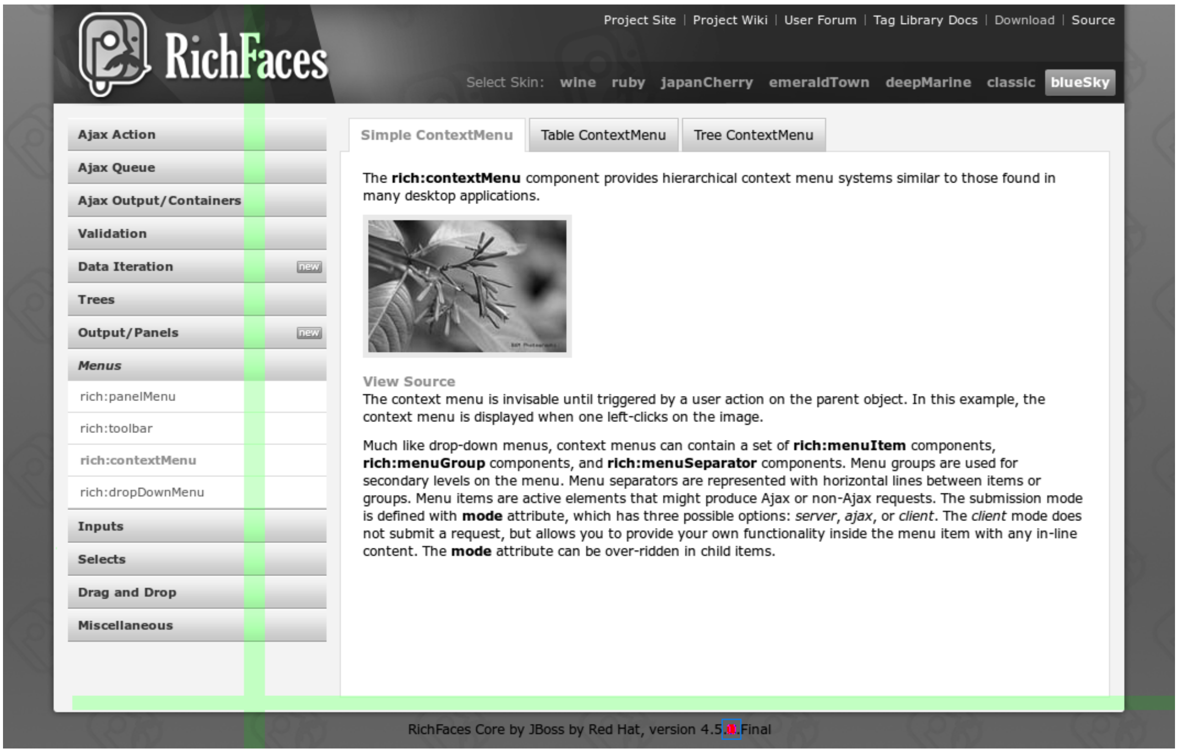

Figure 7.3: RichFaces version changed after it was upgraded.

Maven[6] build.

Functional tests are executed in Continuous Integration environment (see figure 7.1). During this testing, screenshots are made. After testing, pixed to pixel comparison takes place (in the same CI environment). When visual testing is done, created diffs and samples are uploaded into the JCR and can be reviewed later via Web Manager. The tester is able to take an immediate action over the results. He can reject the pattern or sample (it will change the configuration files on the server), which effectively changes the next executions of visual testing in CI environment, as all configuration files are downloaded before each execution of tests.

Test suite and Web Manager are wired via the Arquillian configuration file. Listing 7.3 shows what information is needed to be set in order to pair your test suite and the web manager (the description of all configuration values is included in the documentation of the source code, attached in CD attachment, see D).

Listing 7.3: Example of configuration for Graphene Visual Testing Extension

6. Apache Maven - `http://maven.apache.org/`

```
1 <extension qualifier="visual-testing">
2         <property name="testSuiteName">richfaces-showcase-firefox-
              whole</property>
3         <property name="firstRun">false</property>
4         <property name="failBuild">false</property>
5         <property name="jcrContextRootURL">http://jbosswildfly-
              jhuska.rhcloud.com/modeshape-rest/graphene-visual-
              testing/default</property>
6         <property name="managerContextRootURL">http://jbosswildfly-
              jhuska.rhcloud.com/</property>
7         <property name="jcrUserName">msuser</property>
8         <property name="jcrPassword">CAus6Vj6QsX-</property>
9 </extension>
```

The following text describes the current setting for visual testing form listing 7.3. On line number 2, there is a unique value to distinguish this particular configuration in Web Manager (JCR and PostgreSQL data). On line number 3, when visual testing is done for the first time, this has to be true, patterns are created. In subsequent runs it should be set to false.

On line 4, when visual testing fails, the whole Maven build fails if this is true. This is a very important feature, because it enables CI to indicate to the tester that the visual testing has failed. Otherwise a tester would need to find it out from logs (very time consuming), or to check the Web Manager. For example Jenkins CI[7], when build fails, it is indicated to the tester with a red sign. When build passes, it is blue. This provides instant information about build status, and saves tester's time.

Line 5, 6, 7, 8 are the information needed to wire the test suite and JCR, as well as to wire it with the Web Manager (see attachment E for more information on how to obtain these values).

7.3 Cloud ready

The improvement of the effectiveness of a quality assurance team was our primary goal when developing the tool. This includes easy deployment of the tool either in the organization infrastructure or in a cloud environment, particularly its web part, the Web manager for reviewing results (see 6.5.2).

We chose OpenShift open hybrid cloud by Red Hat to prove the concept that our solution is deployable to a cloud. One of the reasons is that it supports WildFly application server, which integrates well with ModeShape JCR repository 6.6.

To login to the application[8] one has to use `msuser`, and password `CAus6Vj6QsX-`.

7. Jenkins CI - `http://jenkins-ci.org/`
8. Application is available at `http://jbosswildfly-jhuska.rhcloud.com/`

The whole process of deploying Web Manager to OpenShift is described in Appendix E. By following it, it is very easy to deploy the Web Manager application.

`graphene-visual-testing-webapp`

8 Known issues and future plans

We supposed created tools mainly as a proof of concept, which proves our hypothesis (see 5.1). Therefore, we chose those features to the tool to be implemented first, which were essential for proving or disproving the hypothesis.

We are aware of the fact that there are a lot of features missing to have a production ready tool. The implementation of such, however, is out of the scope of this thesis. At the same time we do believe that the tools can be used right now to increase the effectiveness of the quality assurance team. The following known issues (or in other words future plans to extend the tool) need to be kept in mind:

1. Support for Rusheye feature of masking those parts of the web site, which need to be excluded from visual testing, has to be applied manually at the moment. In future developments, it will be possible to do this from the Web Manager during the reviewing of results.

2. Web Manager application security needs to be improved. Right now there is only Basic authentication employed. So in order to work with the application securely, one has to work with it via SSL channels. REST API calls are not secured either.

3. Performance of the application can be improved. Right now we are using eager fetches from databases, and we also do not optimize the client side of the Web Manager in any way.

4. Small Web Manager UI improvements, such as identification of the particular runs with source code management ids (e.g. Git commit id of particular commit in the application, which is tested).

9 Conclusion

The aim of this thesis was to create a tool and a process for a visual testing of a web application, to improve the effectiveness of a quality assurance team. We were obliged to do a research, to find out what output of such a tool would be most useful. Before implementing the tool, an analysis of the already existing solutions for visual testing based on picture comparisons was supposed to be carried out. Based on this analysis and research, we were supposed to design the best way for storing created screenshots, intended for visual comparison. Last but not least, we had to keep in mind, when designing the tool and process, its cooperation with Continuous Integration systems. The improvement of effectiveness of the quality assurance team was supposed to be proved on an actual deployment of the tool and the process to a real world web application.

We can conclude that all of the thesis targets were met. We have created the tool, and the process, which can demonstrably improve the effectiveness of a quality assurance team in some cases by 83%.

The major contribution of this thesis is the created tool and the process itself. It introduces a new approach to visual testing, where visual testing is carried out alongside functional testing of an application. Because of this, testers can develop and maintain only one script, which controls and assert various states of the application under test.

By a careful picking of frameworks (Arquillian, Graphene, Rusheye) and technologies which the final solution was built on, we have achieved that the test suite for a visual testing (the same as for functional testing) is well maintainable, and thus better prepared for reacting to the constantly changing application under test.

We have conducted a research among IT experts on what output of the tool for reviewing results of visual testing would be the most useful. Based on the research results, we have designed a web application called the Web Manager. With this application a tester can take an immediate action over particular results. We have proved that this web application is easily deployable in cloud environment (OpenShift by Red Hat). Last but not least, we have proved its usages in Continuous Integration systems as for example Jenkins.

We have learned a lot of new technologies during the developing of the tool. We have deepened our knowledge of Java EE and JavaScript. We have learned AngularJS framework, Arquillian extensions mechanism, Twitter Bootstrap. We have successfully integrated our solution with a Java Content Repository, and deployed the whole solution to the cloud. We have contributed to open source projects, which we integrate with (list of them is available in appendix B).

There are many possible extensions to this thesis. The Web Manager UI can be improved to display more information about visual testing, even to enhance user the experience with it. Support for applying Rusheye masks from the Web Manager is a very welcomed addition, which we are planning to implement and provide the whole solution as an open source application.

A Appendix A - GUI mockups of Web Manager UI

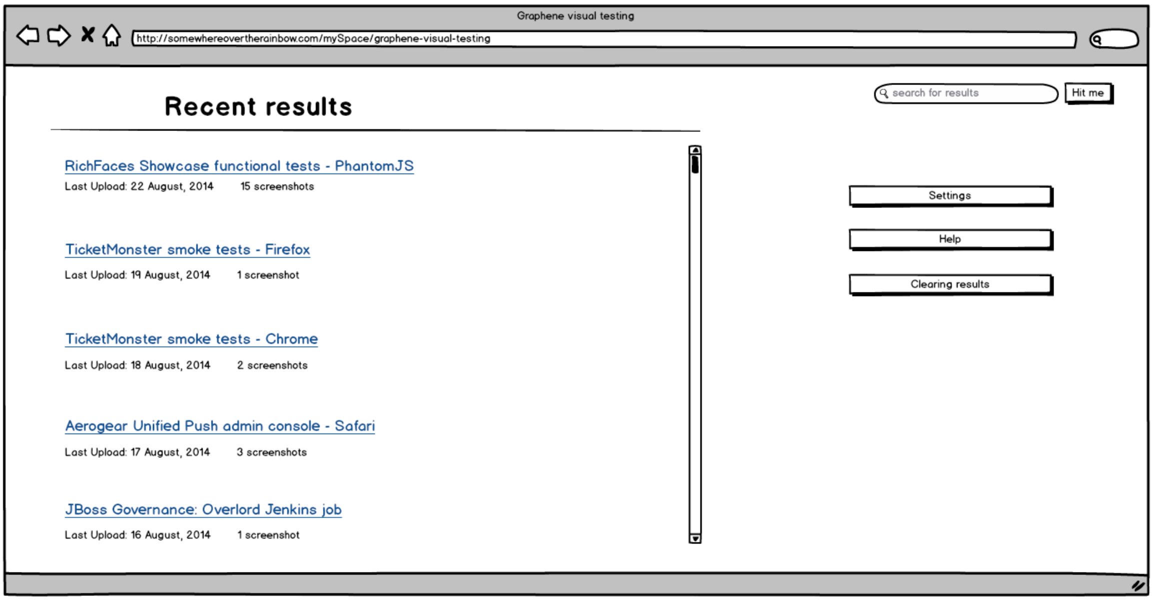

Figure A.1: GUI mockup for result viewer web application - front page.

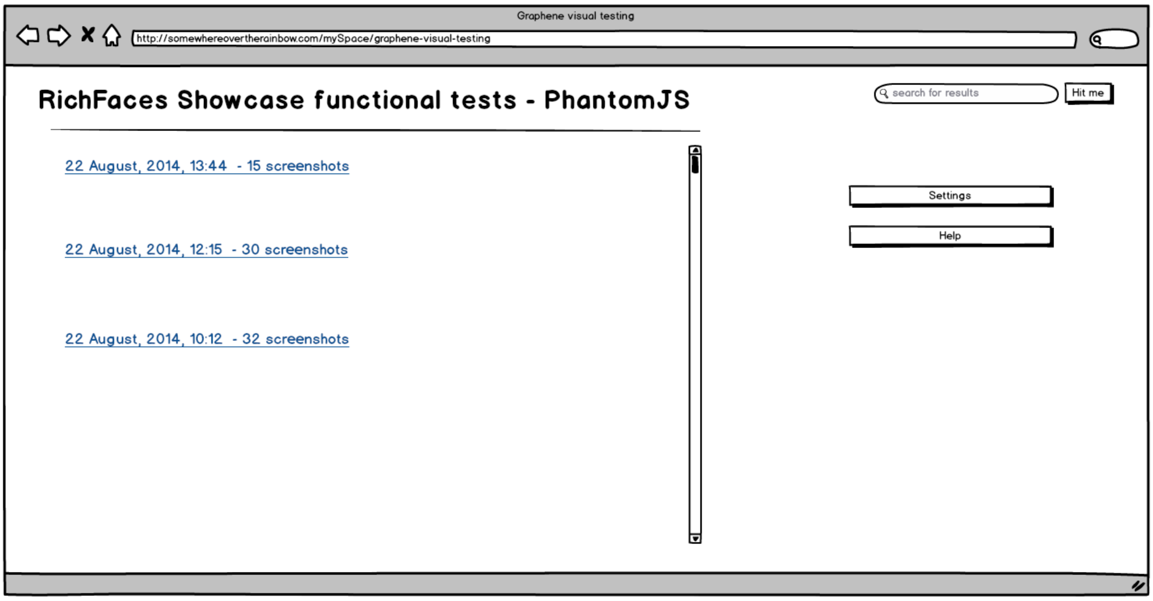

Figure A.2: GUI mockup for result viewer web application - particular test suite run.

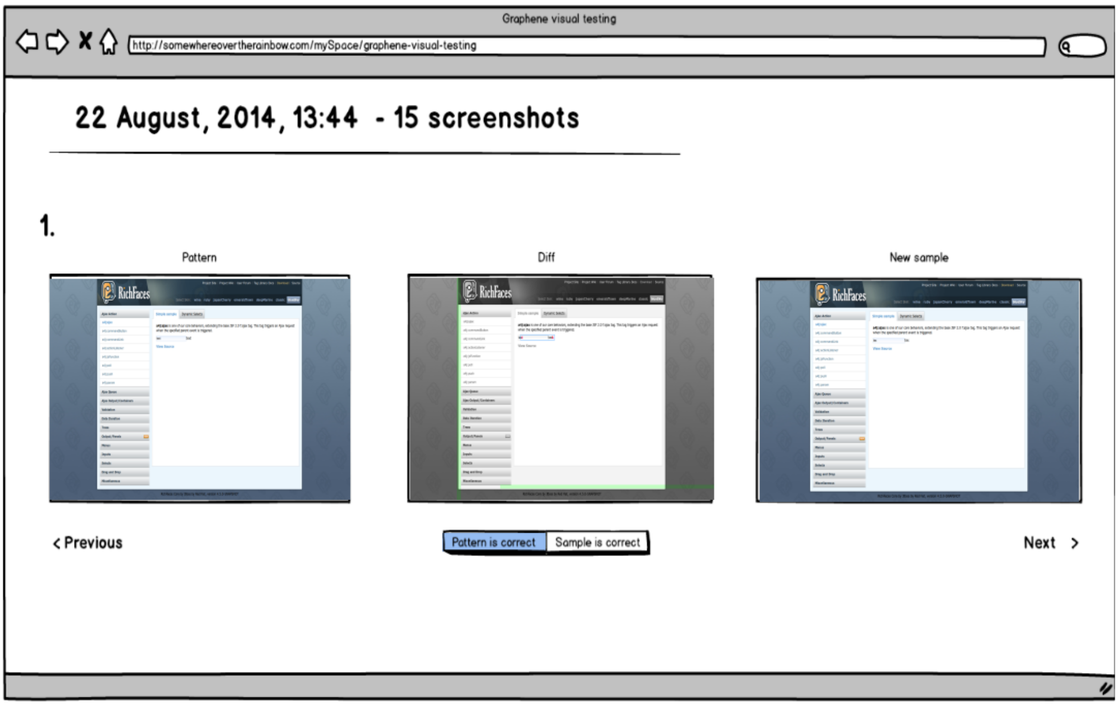

Figure A.3: GUI mockup for result viewer web application - actual comparison results.

B Appendix B - List of contributions to open source projects

1. Arquullian Rusheye, particularly source code available in CD attachment directory:
2. RichFaces Showcase application, particularly source code available in CD attachment directory:
3. Graphene Screenshooter, an extension to Graphene, which enables taking of the screenshots during functional testing. Available at: `https://github.com/arquillian/arquillian-graphene/tree/master/extension/screenshooter`.
4. Add to Graphene Interceptors feature way to intercept in order. Feature request tracked with this issue: `https://issues.jboss.org/browse/ARQGRA-423`.

C Appendix C - Screenshots from Web Manager UI

Figure C.1: Web Manager screenshot - list of test suites.

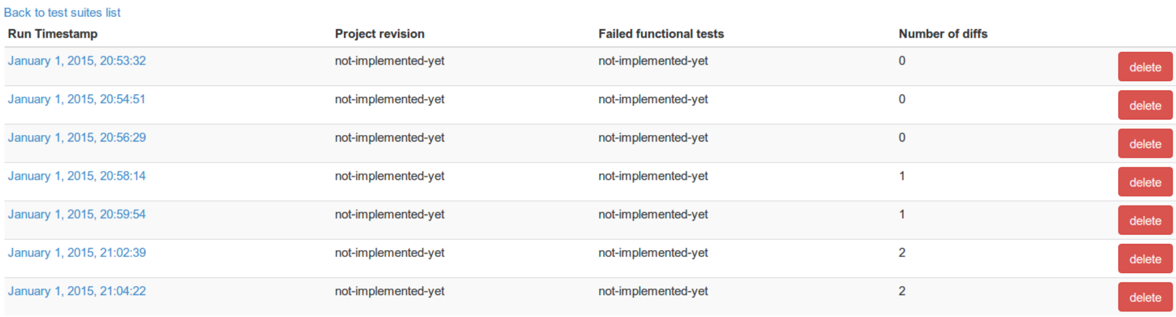

Figure C.2: Web Manager screenshot - list of test suite runs.

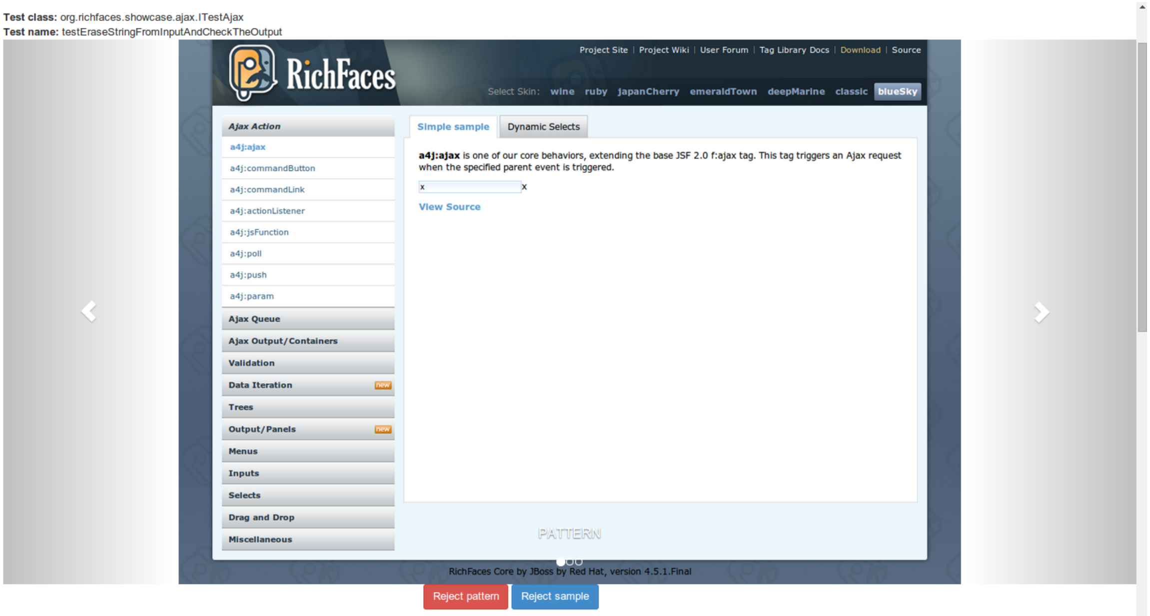

Figure C.3: Web Manager screenshot - test suite run results.

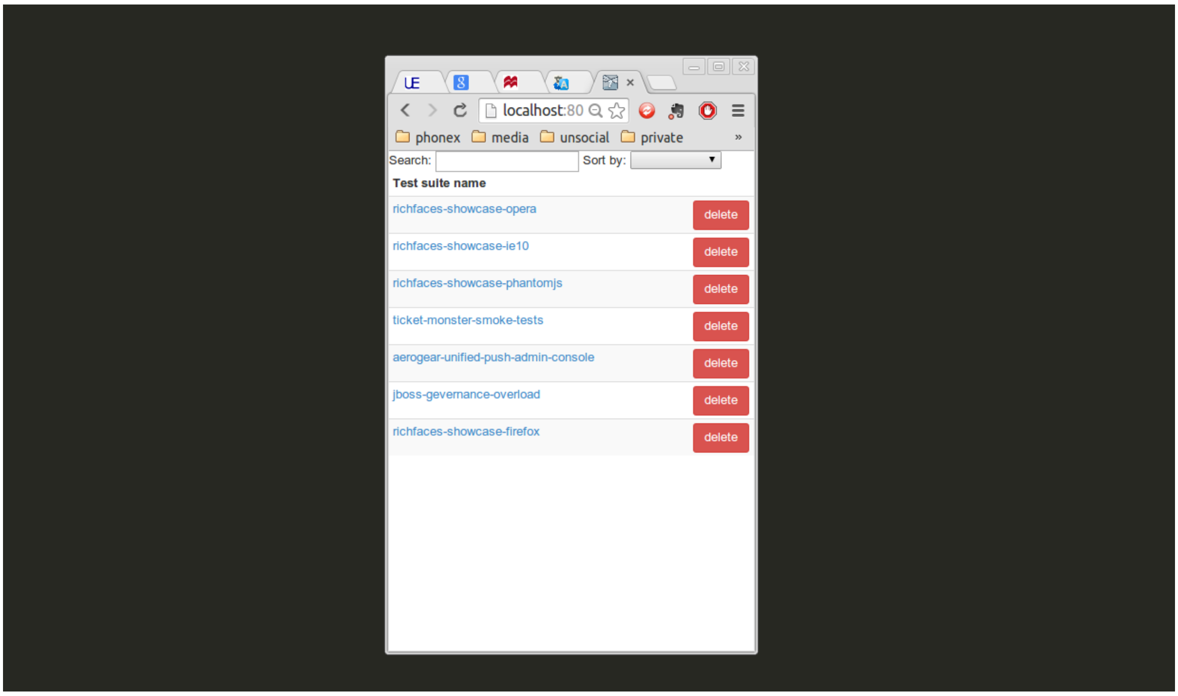

Figure C.4: Web Manager screenshot - test suite list responsive.

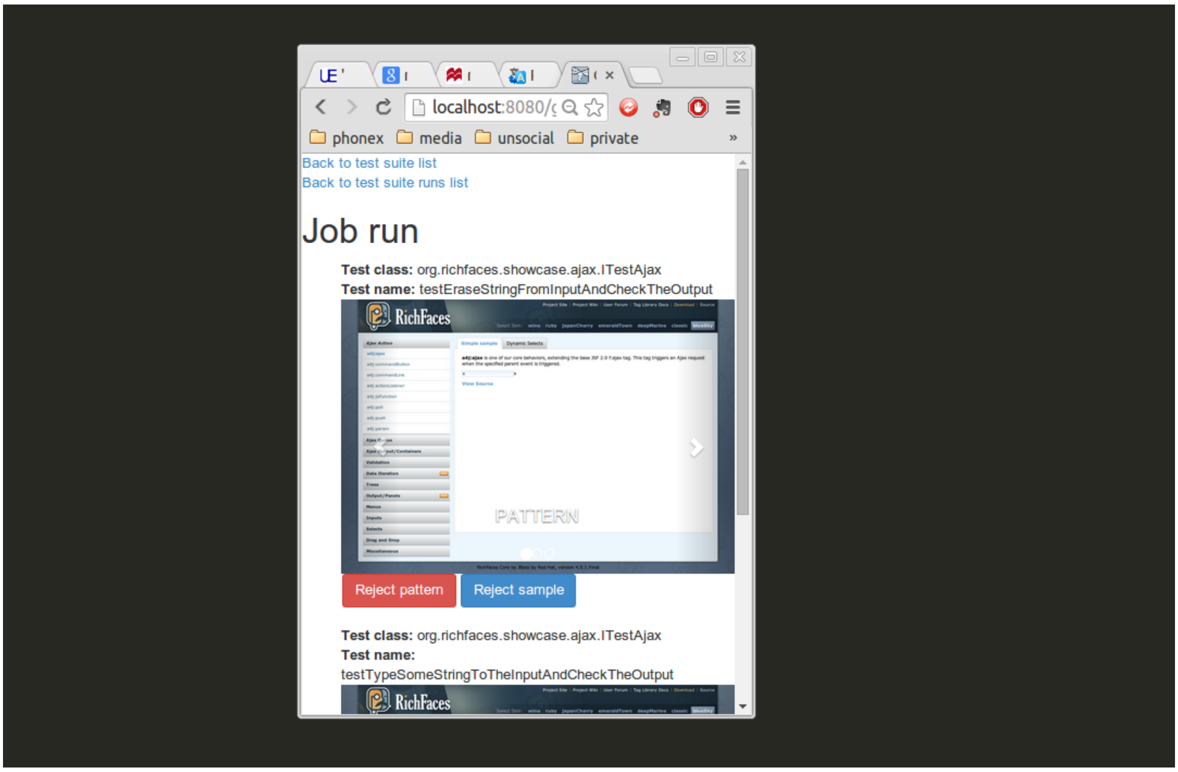

Figure C.5: Web Manager screenshot - test suite run results responsive.

D Appendix D - CD Attachment

CD attachment popis

E Appendix E - How to deploy Web manager on OpenShift

1. Create an account at `https://www.openshift.com`
2. Add application, under Java category, choose WildFly Application Server 8.2.0.Final
3. Choose public URL, wished gear size (we used small), scaling options (we did not used scaling), region of your choice.
4. Click in the created application from Applications menu.
5. Add PostgreSQL database.
6. Under remote access, find out the address to ssh into the application from command line.
7. Run `wget http://downloads.jboss.org/modeshape/4.1.0.Final/modeshape-4.1.0.Final-jboss-wf8-dist.zip`
8. Unzip the downloaded artifacts, and follow this `https://docs.jboss.org/author/display/MODE40/Installing+ModeShape+into+Wildfly` tutorial to enable Modeshape in WildFly installation.
9. Clone the GIT repository of your application.
10. Remove pre-generated `pom.xml`.
11. Build and put `graphene-visual-testing-app.war` into deployments directory of the cloned repository.
12. Alter the `standalone.xml` file, so it combines `standalone-modeshape.xml` and original `standalone.xml`.
13. It should contains all necessary configurations for ModeShape. More information in the tutorial referenced in step 8.
14. Commit and Push the changes into the remote Git repository.
15. Your application should be available at the public URL of the application you created in OpenShift + add the end of the path: graphene-visual-testing-webapp
16. The login and password to the application should be same as the ones for create WildFly user in step 8.

F Appendix F - Getting started with visual testing

1. Consider please figures 5.1, 5.2, 6.1, 7.1, to have a better picture what is the process behind visual testing.

2. Add to your Maven pom.xml dependencies section following dependency:

```xml
<dependency>
  <groupId>org.jboss.arquillian.extension</groupId>
  <artifactId>graphene-visual-testing-arq-ext-api</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

3. Add to your profiles section in Maven `pom.xml` following profile:

```xml
<profile>
  <id>visual-testing</id>
  <dependencies>
   <dependency>
     <groupId>org.jboss.arquillian.extension</groupId>
     <artifactId>graphene-visual-testing-extension-impl</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <scope>test</scope>
   </dependency>
  </dependencies>
</profile>
```

4. Copy prepared binaries to a local Maven repository. They can be found in CD attachments in a directory: /binaries/.m2/repository

5. Follow instructions from E to deploy Graphene Visual Testing Web manager app to OpenShift cloud. If you want to deploy it locally, use WildFly 8.2.0.Final. It is very similar to deploying to OpenShift.

6. Put following minimal configuration to `arquillian.xml` configuration file of your Graphene functional test suite:

```xml
<extension qualifier="visual-testing">
  <property name="testSuiteName">uniquely-name-your-test-suite</property>
  <property name="firstRun">true</property>
  <property name="jcrContextRootURL">http://urlToDeployedJCR/contextRoot</property>
  <property name="managerContextRootURL">http://pathToDeployedWebManager</property>
  <property name="jcrUserName">createdUserName</property>
  <property name="jcrPassword">createdPassword</property>
</extension>
```

7. Run functional test suite for the first time to generate patterns. Do not forget to run the maven build with profile `visual-testing`.

8. Change property with name `firstRun` in `arquillian.xml` to `false`.

9. Run functional test suite again. See results in `http://pathToDeployedWebManager/graphene-visual-testing-webapp`

Bibliography