

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Visual testing something catchy

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

Some people helped me a lot and some not at all. Nevertheless, I would like to thank all.

Abstract

This thesis is very important!

Keywords

key word1, and so on

Table of contents

1	Introduction	3
2	Visual testing of software	4
2.1	<i>Visual testing in release testing process</i>	4
2.2	<i>Need for automation</i>	5
2.3	<i>Requirements for automation</i>	6
2.3.1	Low cost of test suite maintenance	7
2.3.2	Low percentage of false negative or false positive results	7
2.3.3	Reasonable time to execute a test suite	7
2.3.4	Concise yet useful test suite output	8
2.3.5	Support for Continuous Integration systems	8
3	Analysis of existing solutions	9
3.1	<i>Mogo</i>	9
3.1.1	Mogo drawbacks	10
3.2	<i>BBC Wraith</i>	10
3.2.1	PhantomJS	11
3.2.2	CasperJS	12
3.2.3	BBC Wraith drawbacks	13
3.3	<i>Facebook Huxley</i>	13
3.4	<i>Rusheye</i>	14
3.4.1	Rusheye drawbacks	15
3.5	<i>Conclusion of analysis</i>	15
4	New approach	18
4.1	<i>Hypothesis</i>	18
4.2	<i>Process</i>	19
4.2.1	First execution of functional tests	19
4.2.2	Subsequent execution of visual testing	19
4.3	<i>Analysis of useful tool output</i>	21
5	Implemented tool	23
5.1	<i>Arquillian</i>	24
5.2	<i>Arquillian Graphene</i>	24
5.3	<i>Graphene screenshooter</i>	25
5.4	<i>Rusheye</i>	26
5.5	<i>Graphene visual testing</i>	26
5.5.1	Arquillian extension	27
5.5.2	Web application to view results	27
5.6	<i>Storage for images</i>	29
6	Deployment of tool and process	33

6.1	<i>Actual visual testing</i>	34
6.1.1	Results	36
6.2	<i>Usage with CI</i>	37
6.3	<i>Cloud ready</i>	37
7	Possible extensions	39
8	Conclusion	40
A	Appendix A - GUI mockups of Web Manager UI	41
B	Appendix B - List of contributions to opensource projects	44
C	Appendix C - Screenshots from Web Manager UI	45
D	Appendix D - CD Attachment	46
E	Appendix E - How to deploy Web manager on OpenShift	47
F	Appendix F	48
	Bibliography	48

1 Introduction

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

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

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

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

The thesis consists from five main parts: a broad introduction to automated visual testing, and reasoning a motivation behind it.

Second part analyses already existing solutions, summarizes their drawbacks as well as advantages.

Third part formulates a 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 inevitable to adhere in order to be effective with the created tool.

Fourth part describes implementation details of developed tool, list of components we reused or developed to obtain a final result, as well as a reasoning why we choose particular component to integrate with.

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

2 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 actual using of an application, or automatically by executing testing scripts.

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

Typical example can be a web application, which GUI is programmed usually with combination of HyperText Markup Language (HTML) and Cascading Style Sheets (CSS). HTML is often used to define a content of the web application (such as page contains table, pictures, etc.), while CSS defines a 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 not possible to find out by functional tests of the application. It is because functional tests verify a desired functionality of the web application, and do not consider web page characteristics such as red color of heading, space between two paragraphs, and similar.

That is why a visual testing has to take a place. Again, it is done either manually, when a tester by working with an application, is going through all of its use cases, and verifies, that the application has not broken visually. Or automatically, by executing scripts which assert a 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 how web page looks like is mainly determined by CSS script. There are two ways of automated testing used:

1. asserting the CSS script
2. or 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 of screenshots only, as it is a method, which more likely reveals a bug in a visual state of an application under test.

2.1 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. While 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 RichFaces¹ Quality Engineering team² 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 to go through its use cases with a particular set of web browsers.

To be more specific, consider please a web page with a chart elements showing a sector composition of gross domestic product in the USA (as figure 2.1 demonstrates). To verify its visual state is not broken, 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 a specific places in the chart, and verifying whether a popup with more detailed info is rendered in a correct place.
4. Repeat this for all major browsers³, and with all supported application containers⁴.

2.2 Need for automation

The chapter 2.1 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 tester's intellect enough, become a mundane activity. The more one repeats the mundane activity, the more likely an mistake is introduced: one forgets to try some use cases of an application, overlooks some minor errors, etc.

Automated visual testing addresses this shortcomings, as it would unburden human resources from mundane activities such as manual testing, and would allow spending their time on intellectually more demanding problems. However, it introduces another kind of challenges, and needs to be implemented wisely. 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 the other things responsible for assuring a 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 provide a runtime environment for complex enterprise web applications, e.g. JBoss AS, Wildfly, Apache Tomcat

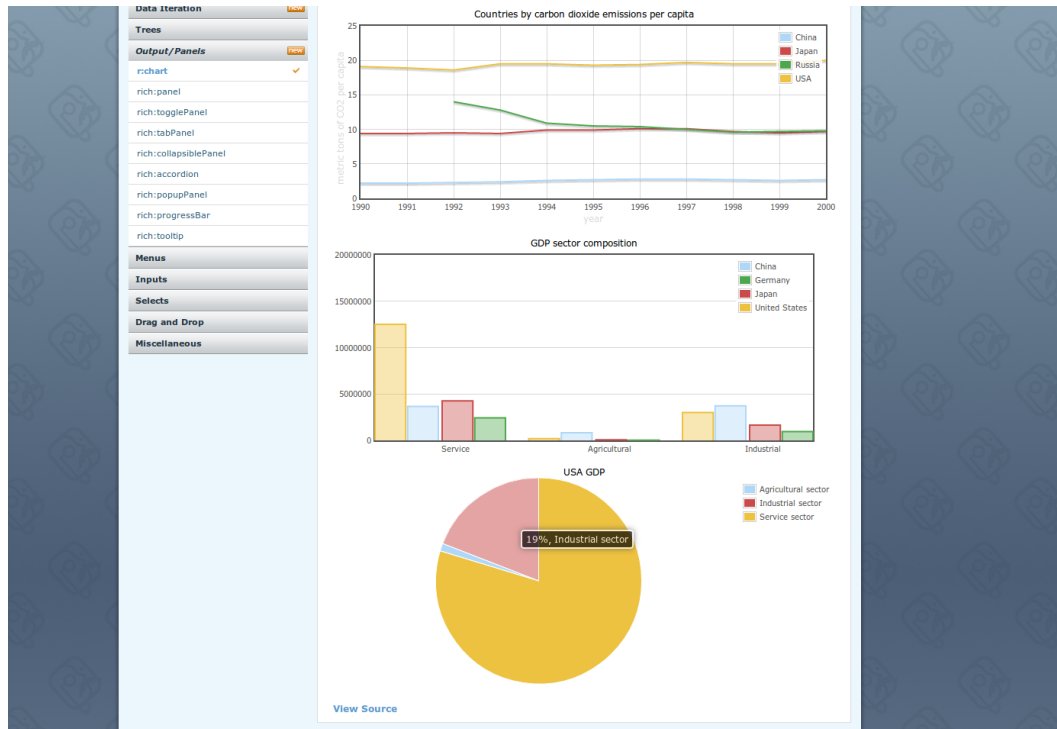


Figure 2.1: RichFaces chart component shown in Showcase application

2.3 Requirements for automation

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

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

1. A low cost of a 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 systems⁵.

5. Continuous Integration (CI) system is software to facilitate a practice of CI, which in short is about merging all developer copies with a shared mainline several times a day [5].

2.3.1 Low cost of test suite maintenance

A test suite needs to reflect a development of an application under test. Therefore, with each change in the application, it is usual that the test suite 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 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. That 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 (those correct ones, other screen captures will be compared with) screen captures are made.

2.3.2 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 unwanted phenomenon as they take time to process and assess correctly.

False positive tests results hide actual bugs in an application. They provide an 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 to be broken by small changes on a page. For example if the page outputs a current date, then it would break with every different date. There has to exist techniques, which would prevent these situations.

2.3.3 Reasonable time to execute a test suite

Reasonable time is quite subjective matter, but in general, it depends on how many times e.g. per day one needs to run whole test suite. Nowadays trend is a Continuous Integration, when 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 introduced 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 a developer/tester runs test suite several times a day).

2.3.4 Concise yet useful test suite output

One of drawbacks of automated testing is its ability to produce huge amount of logs, test results etc. The output therefore needs to be as concise as possible, while still providing an useful information. A tester needs to be able to quickly recognize where the issue might be. The best situation would be if the tester does 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 difference of old and new version.

2.3.5 Support for Continuous Integration systems

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

3 Analysis of existing solutions

As we introduced in 2.3, there are many aspects which need to be taken into consideration when automating visual testing. 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 an ability to be used in enterprise context. In an enterprise company, there is a stress on stability and reliability of employed solutions. It is quite vague requirement, and it is usually hard to find out 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 the project has? When was the last activity with those issues ?
- What is the quality of the source code? Is it well structured? Does it employ 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 following sections we are going to show an example usage and its main drawbacks together with some basic description.

3.1 Mogo

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

1. One provides set of URLs of an application under test to a cloud based system.
2. Application URLs are loaded in 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.

3.1.1 Mogo drawbacks

Drawbacks of this approach are evident when testing dynamic pages, which content is changed easily. Applications which provide rich interaction options to an end user, and which state changes by interacting with various web components (calendar widget, table with sorting mechanism etc.), require 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.

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 tests results when such a tool is used to test mocked data ¹.

3.2 BBC Wraith

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

1. Take screenshots of two versions of web application by scripting either PhantomJS 3.2.1, or SlimerJS² by another JavaScript framework called CasperJS 3.2.2 [20].
2. One version is the one currently developed (which run on localhost³), 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 a created picture, which is the result of comparing two pictures (It can be seen at Figure 3.1).
5. All pictures can be seen in a gallery, which is a generated HTML site (It can be seen at Figure 3.2).

To instruct 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, creates a configuration file, which will tell the PhantomJS instance which URLs need to be loaded and tested. PhantomJS script is one source of distrust to this tool, and therefore is introduced furthermore.

1. Mocked data is made up data for purpose of testing, so it is consistent and does not change over time
2. SlimerJS is very similar to PhantomJS 3.2.1, 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]

3.2.1 PhantomJS

PhantomJS [8] is stack of web technologies based on headless⁴ WebKit⁵ engine, which can be interacted with by using of 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 a engine a bit faster and less computer resources demanding.

One can instruct PhantomJS to take a screenshot of a web page with 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 machines⁶, it does not have to be enough. It will result in massive number of false negative tests. There is a need for more deterministic way of finding out whether the page was loaded fully in 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 poor way of naming created screenshots (a user has to choose name wisely), it might lead to problems when maintaining such a test suite.

PhantomJS API is wrapped by CasperJS, which is furthermore 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

3.2.2 CasperJS

CasperJS is 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].

Following code snippet shows a simple navigation on Google search web page. It will load *http://google.com* in a browser session, will type into the query input string *MUNI*, and will 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 is needed to be introduced.

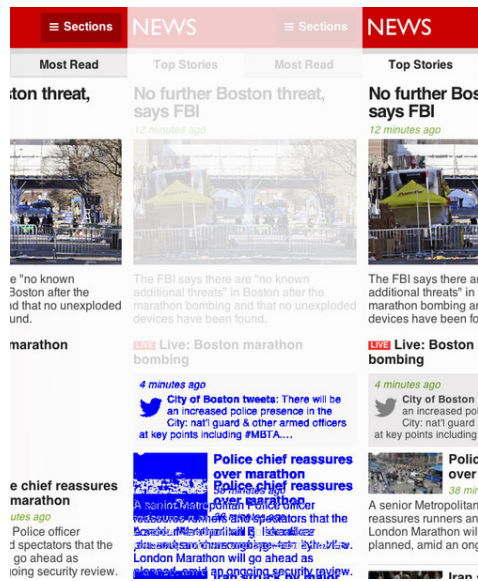


Figure 3.1: BBC Wraith picture showing difference in comparison of two web page versions [12]

List of screenshots for shots

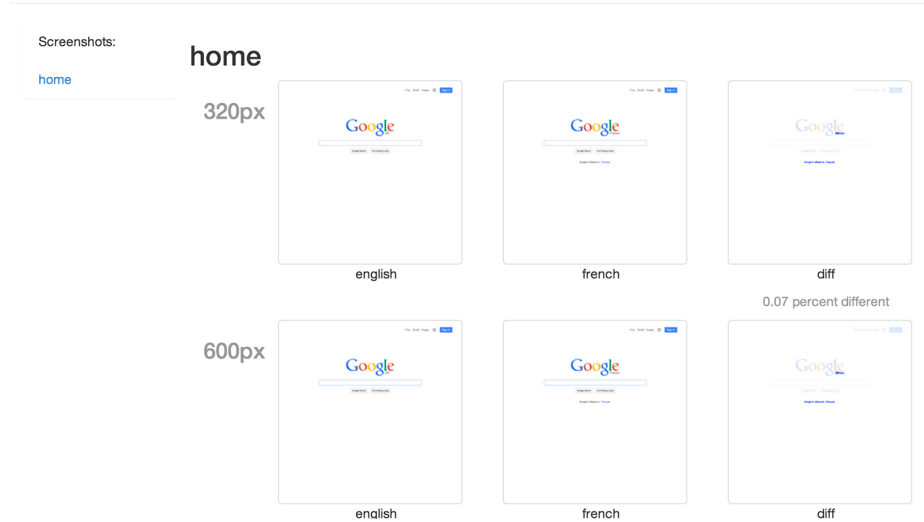


Figure 3.2: BBC Wraith gallery example [13]

3.2.3 BBC Wraith drawbacks

Two of the drawback were described in the previous sections, 3.2.1 and 3.2.2.

Another problem which might occur when testing with BBC Wraith, is cross browser compatibility. As it supports only PhantomJS, and therefore, one can not 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 web page might look differently in Google Chrome and Microsoft Internet Explorer, and PhantomJS will not catch this issues.

3.3 Facebook Huxley

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

1. One creates a script which would instruct Huxley tool, what 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. That is the mode when Huxley loads the pages automatically in a fresh browser session, and a tester by hitting Enter keyboard button instructs Huxley to take a screenshot. Screenshots are stored in a folder with test name (one given in the square brackets in the example above), together with a JSON⁷ 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 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).

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 time to wait can be configured, however, it is still quite error prone, as for first visual testing run e.g 4 seconds can be would be enough, and for another run would not.

Secondly it lacks a proper way of viewing results of comparisons, leaving with 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 solutions⁸, where such a cooperation might take a place.

3.4 Rusheye

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

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].

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

The process has subtle differences in comparison with previous solutions. It consists from 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 in web application under test, another run of Selenium 1 tests generates new screenshots. They are called samples.
4. Patterns and samples are compared, its visual difference is created, and result is saved in an XML file.
5. The results can be viewed in a desktop application, Rushey manager [22].

Rushey has one very important feature, which another tools lack. It is a 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 protect from false negative tests, as some all the time changing parts (such as actual date, etc.) can be masked from comparison, and thus their change will not break testing.

3.4.1 Rushey drawbacks

Core of the Rushey 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 3.2.2. 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 (Rushey Manager). 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.

3.5 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, what we suppose to be an inevitable part of an effective automated visual testing.

Figure 3.3 summarizes 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 underlying HTML, in which the application is written. It is quite 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 long term run of the test suite might cause lot of issues.

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

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

By cloud ready features we suppose whether tool has 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 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	<i>not applicable</i>				
BBC Wraith	<i>insufficient</i>				
Facebook Huxley	<i>insufficient</i>				
Rusheye + Selenium 1	<i>insufficient</i>				

Figure 3.3: Existing solutions features comparison

As Figure 3.3 shows, none of the tools met our requirements fully. Therefore, we decided to proceed with developing of a new tool, which would address all issues, and which would

integrate existing parts of the solutions when it is reasonable. Creation of a new process which would enable effective usage of such tool by QA team members is inevitable. Following chapters describe this new tool and the new process.

4 New approach

Each tool by its definition introduces a process for a visual testing. While we recognized shortcomings (described in the chapter 3.5), we realized a need for a different approach to the visual testing. The approach came from 2 and half years of developing and maintaining the functional test suite ¹ for RichFaces project ².

4.1 Hypothesis

It should be enough to have just a functional test suite for a end to end testing of an application. Scripts from functional testing interacts with the application sufficiently, therefore, another script which during such interactions take screenshots 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 on more places, which allows errors to sneak in test suites.

In the same time we do believe, that script for functional testing should be abstracted from a visual testing as much as possible. Meaning that, explicit calls to methods which take screenshots, should be optional. 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.

For sure there should 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 to configure on global level, meaning for all tests from one place, as well as on test level, in what 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 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 results of visual testing. More particularly asses the results, whether there is a bug in application or not. This web interface will foster cooperation between interested parties.

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

1. The RichFaces test suite is available at <https://github.com/richfaces/richfaces-qa>

2. RichFaces is component library for Java Server Faces web framework [23]

4.2 Process

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

1. Executing a visual test suite for the first time to generate patterns, which are new screenshots in subsequent tests executions will be compared with.
2. Executing the visual test suite for second and more times, to generate new screenshots, which are called samples in this thesis, for comparison with patterns generated in the first 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 development, we suppose the third problem to be solved with re-executing the whole test suite again, as it is done in the first run of the test suite. Overall process can be described with following subprocesses.

4.2.1 First execution of functional tests

Figure 4.1 denotes steps needed to generate patterns. It is a prerequisite for the visual testing.

Screenshots are generated during 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 test need to be fixed, made to be passing, to include it 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.

4.2.2 Subsequent execution of visual testing

Figure 4.2 shows how subsequent execution of visual testing together with functional testing would look like. The first step is same as in previous process, functional tests are executed, and screenshots are taken. Secondly, patterns are downloaded from the storage, and after that actual visual testing can start.

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

If there are found some differences, 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 he will find the particular run according to a time stamp when the run was executed. They should be able to assess the results, with displayed triplet, consisting of pattern, sample and their difference. They should be able to say whether it is 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 more described in chapter 3.4.

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 user responsibility to file such a bug in a corresponding issue tracker. Generated pattern, sample and difference can be used as a useful attachment to a issue report, which would better describe actual and expected result.

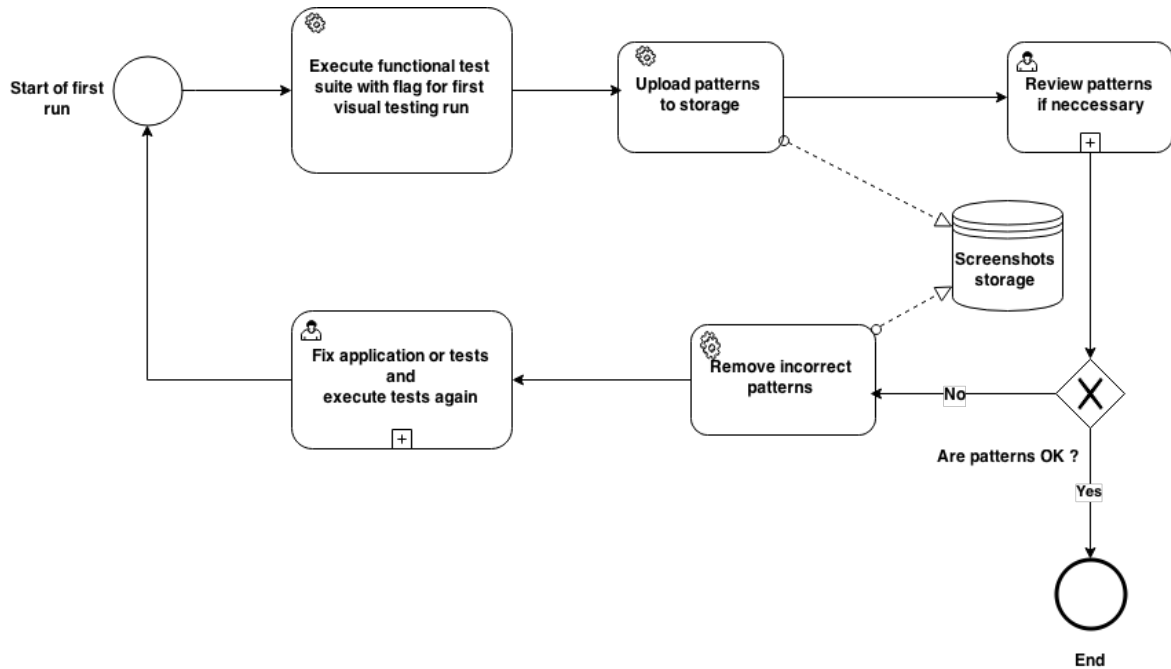


Figure 4.1: Process to generate patterns during first execution of functional tests.

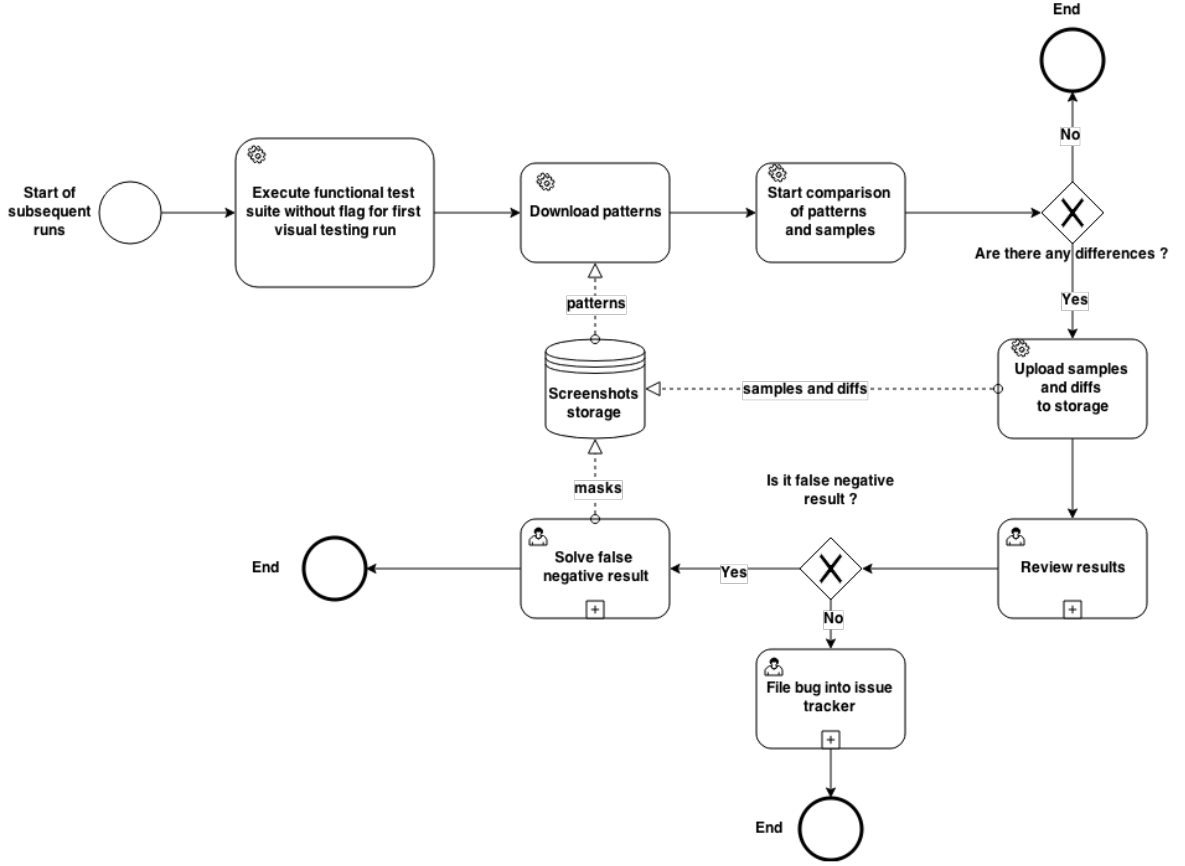


Figure 4.2: Subsequent execution of visual testing together with functional tests.

4.3 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 tool has to show results of the visual comparison in a way, that it is a pleasure to use it. In the end of the day, the amount of time spent with using of the tool has to be dramatically less 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. 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. By the time of writing of 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 a valuable feed back, together with many interesting ideas what features such a web application for reviewing results should have.

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

You can see created mockups in appendix A. The web application is logically divided into three screens. First one (can be seen in appendix A.1) is showing 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, represent a test suite for visual comparison testing.

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 a time and date, they were executed.

Third created mockup shows comparison results for a particular tests suite run. The comparison result consists from three screenshots. A pattern, created during 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. They purpose is to allow a tester to take an immediate action upon results. 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. Sample is correct button, is used when there is an anticipated change in the application under test. In that case the new created sample should be used as the pattern in next comparisons.

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

- Indicate number of failures versus total number of tests.
- Revision number of application under test. For example Git³ 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/>

5 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 already existing tool, and integrated it to the final result.

Figure 5.1 depicts component diagram of implemented solution. It is only a high level picture of overall solution. Particular components are described in detail in following chapters.

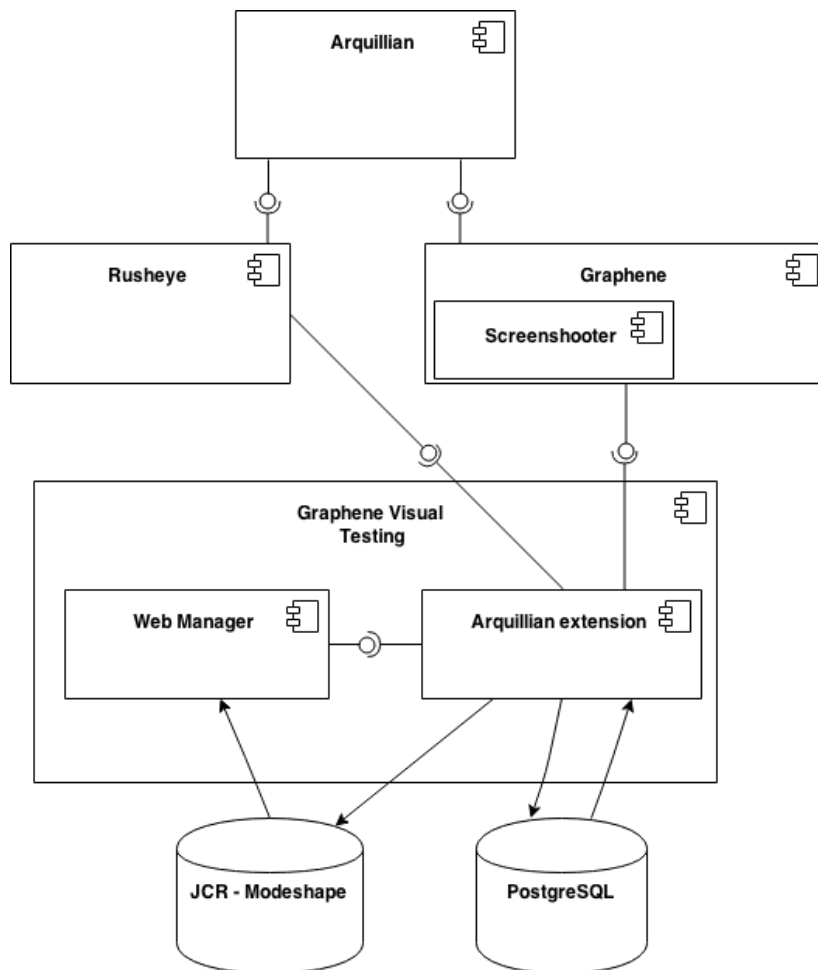


Figure 5.1: Component diagram of implemented solution for visual testing.

5.1 Arquillian

Arquillian¹ 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 application container before testing, and to stop it after tests execution is done, can deploy application under test to that container, populate database with testing data, start web browsers, test on mobile devices, and to provide useful output with videos and screenshots from testing.

In other words, it manages life cycle of all integration components your application integrate 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.

5.2 Arquillian Graphene

Arquillian Graphene is an extension to Arquillian, thus 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 standard³, which guarantees stability, making it a good candidate to build our solution with.

Graphene's type safe API⁴ supports design patterns such as Page Object⁵ or Page Fragment⁶.

Those features form an API with high abstraction level, and encourage a tester to develop more readable tests. Those are attributes, which already existing solutions lack (see 3.5 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].

5.3 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 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 3):

1. A tester should be able to configure this extension so it takes screenshots automatically. A script for functional test should be unaware of such screenshots taking behavior. It will stay clean and more readable (see chapter 3.5 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 screenshot at any place in the functional test script. This should be just an optional feature, used rather sporadically.

Second point is developed in all 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 take place for Arquillian framework. It is in `arquillian.xml` file.

Listing 5.1: 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 5.1, you can see, that screenshots are automatically made after two events: before test, which is effectively 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 browser⁸.

7. Arquillian Recorder is an extension which among the other things defines interfaces for screenshot taking, and video recording from tests executions. We have cooperated on defining this common interfaces 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>

5.4 Rusheye

We have already described some of the Rusheye features in chapter 3.4. Its 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. That 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 realization of Observer pattern¹⁰.

Rusheye observes to events like: `StartCrawlingEvent`, or `StartParsingEvent`, and reacts according to it. It starts crawling of patterns, and creates a test suite descriptor (XML file which describes where are the particular screenshots for a particular functional test stored). This is done after first run only (see chapter 4.2.1). In subsequent runs `StartParsingEvent` is fired, and Rusheye starts actual visual comparison, it compares patterns with samples on pixel to pixel basis.

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

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

Listing 5.2: Example of `StartParsingEvent` observer

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

5.5 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

5.5.1 Arquillian extension

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

As listings 5.3 shows, it wires up to Arquillian life-cycle, as it listens to **AfterSuite** event. If it is a first execution of test suite, then it just fires **StartCrawlingEvent** event, which is observed by Rusheye. After crawling is done, it stores created suite descriptor, and screenshots to a Java Content Repository (JCR - see chapter 5.6).

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 what patterns and what samples were different, and a path to created diffs) are also uploaded to a JCR.

Listing 5.3: AfterSuite observer to controll Rusheye

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

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

    if (visualTestingConfiguration.get().isFirstRun()) {
        crawlEvent.fire(new StartCrawlingEvent(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 are using Apache HttpComponents project¹¹.

5.5.2 Web application to view results

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

11. Apache HttpComponents - <http://hc.apache.org/>

The application back-end is developed with use of Java Enterprise Edition¹² stack. It includes using of technologies like: Java Persistence API¹³ plus PostgreSQL¹⁴ for a persistence layer. Enterprise Java Beans¹⁵ for controllers code (Model-Viewer-Controller pattern¹⁶). It exposes Java API for RESTful Services (JAX-RS¹⁷) endpoints to communicate enable communication with Arquillian extension (see chapter 5.5.1), and with its client part (HTML front end) in a RESTful way.

The application is deployed on a WildFly application server¹⁸. We chose this server, because of its an open source software, backed with huge community of users, and by a big enterprise, Red Hat¹⁹. Very important for us was also its fastness and availability in cloud environments (see chapter 6.3).

The front end is developed with use of AngularJS, which is a MVC framework for creating a Single Page Applications (SPA²⁰). It is complemented with Twitter Bootstrap framework, to provide a visually pleasant UI.

The design of particular screens are in accordance with the analysis described in section 4.3. You can find screenshots from the application in the [DOPLNIT APPENDIX].

For better convenience, and to verify possibility to deploy application on the cloud, we have deployed it on Platform as a Service, OpenShift by RedHat cloud²¹. Please see chapter 6.3 for more information about the deployment (how to log in, etc.).

Following figure 5.2 depicts possible use cases with the web application, to give a better picture 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 on 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 pattern in subsequent runs.

Reject sample is used when the results is either 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. 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 7.

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/>

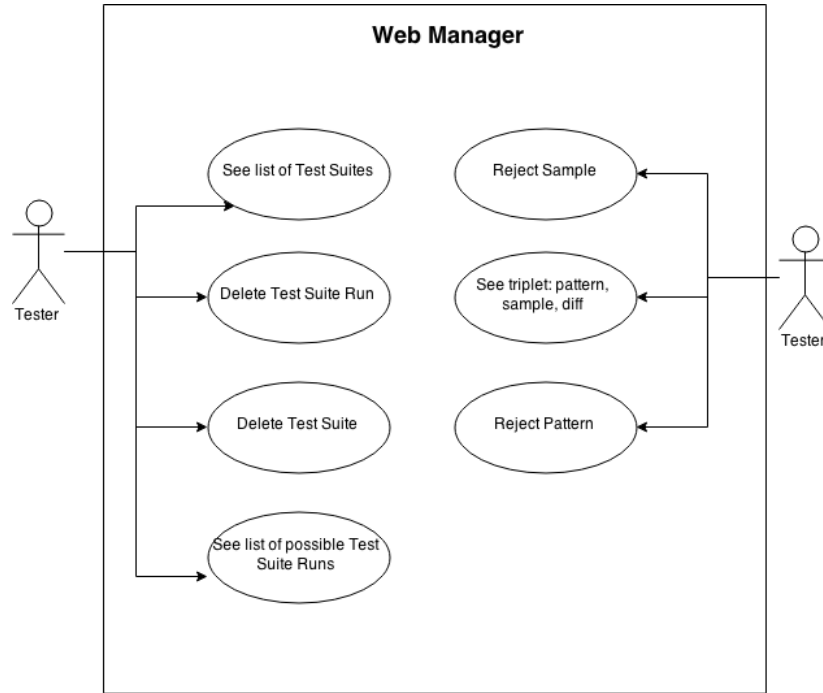


Figure 5.2: Use case diagram for Web Manager web application.

5.6 Storage for images

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

We have these requirements for storing of the images:

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

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

1. Store screenshots as Binary large objects (BLOBs²²) in a relational database, such as PostgreSQL.
2. Store screenshots in a file hosting service such as Dropbox²³, or Google Drive²⁴. Store just URLs to relational database.
3. Store screenshots in a Java Content Repository (JCR²⁵). Store just URLs to 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 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 can not be accelerated for example by configuring a web server to use system call to asynchronously send a file directly 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 database does. In a 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 more flexible solution, which does not vendor lock in to some providers. Which is free of charge when big storage space is required.

Therefore, we chose option number three as 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;
- and transactions, versioning, locking.

Data we need to store is naturally hierarchical. See figure 5.3 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, this 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 need to work often only with subset of the hierarchy - for example display triplet pattern, sample, and diff for a particular test suite run.

We wanted to provide flexible data structure. By this tool stays extensible to other approaches and new features for later development (see section 7).

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 out²⁶), and we want to allow cooperation of testers on a particular test suite (their actions on results will need to be transactional).

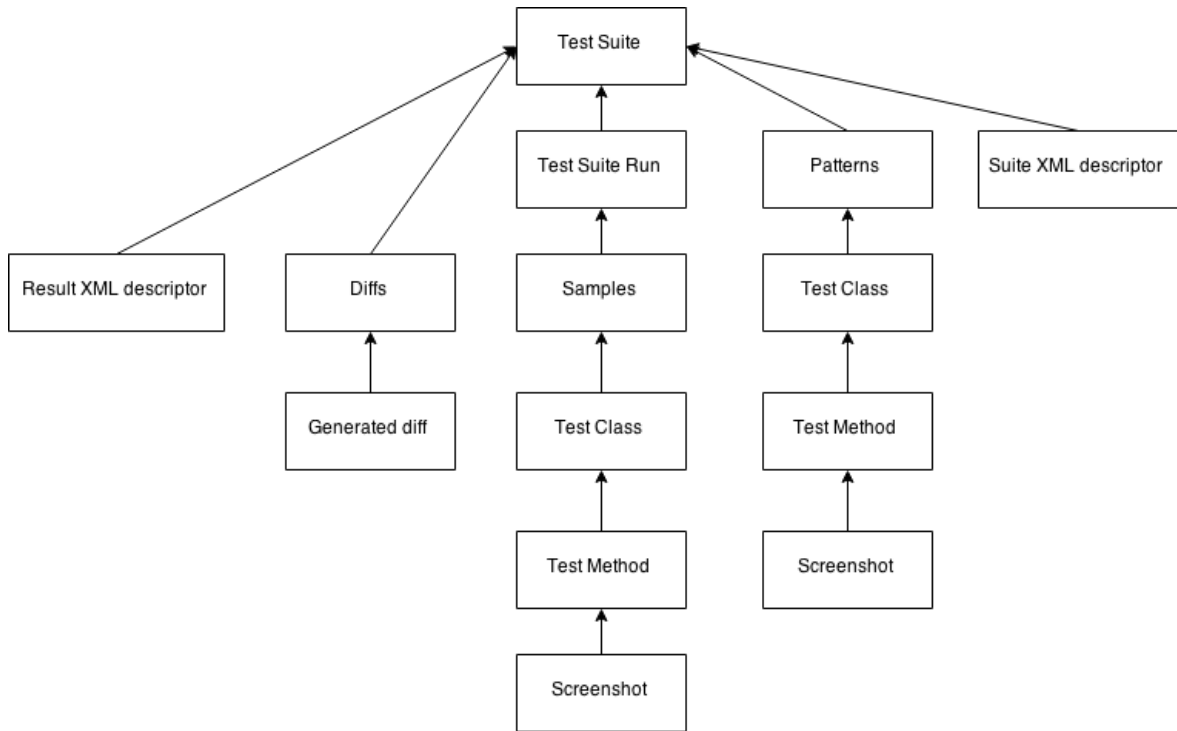


Figure 5.3: The hierarchy of nodes in our JCR deployment.

We chose JCR implementation ModeShape²⁷. 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>

Jackrabbit²⁸: The development is backed by Red Hat, the same company we chose the application server from (see 5.5.2). 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 5.5.2), when screenshots 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 protocol²⁹, and possibility to cluster multiple ModeShape instances.

28. Apache Jackrabbit - <http://jackrabbit.apache.org>

29. WebDAV protocol - <http://en.wikipedia.org/wiki/WebDAV>

6 Deployment of tool and process

After implementing the tool, to prove or disprove hypothesis from section 4.1 we need to deploy the tool and the process (see section 4.2) on a real application. Following chapters describe such deployment, real world use cases and best practices when using our tool.

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

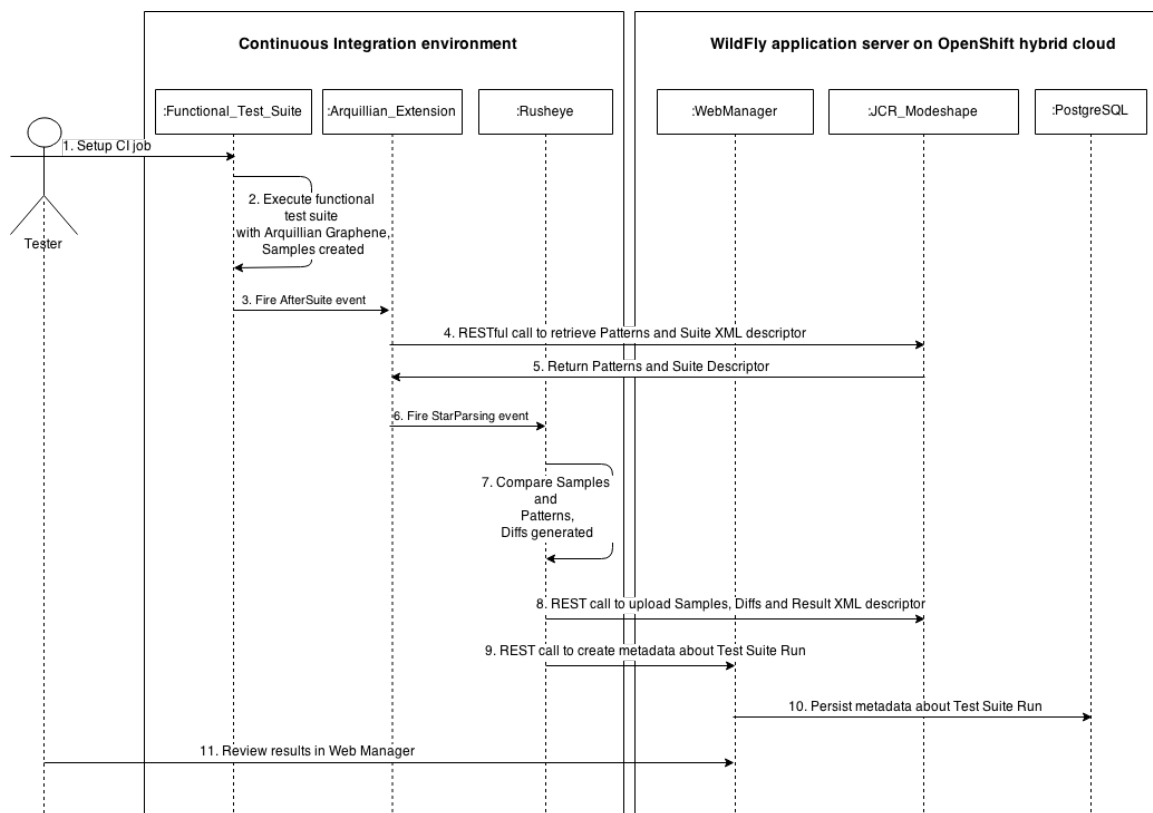


Figure 6.1: Whole visual testing solution sequence diagram.

There are omitted processes which are not important for visual testing, or processes which describes how data is transported into the Web Manager, already described in section 5.6.

6.1 Actual visual testing

We chose RichFaces Showcase application¹, and its functional test suite² 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. Verify visual state of the application after upgrade of the core RichFaces library, from version 4.5.0.Final to 4.5.1.Final. We proceed as follows:

1. Web Manager were deployed to OpenShift cloud on a WildFly 8.2.0.Final cartridge (see chapter 6.3).
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 version 4.5.1.Final.
6. Results were analyzed.

We deployed Web Manager to the cloud, as that would be its the most probable environment. It made conditions for testing more difficult, because we run the test suite from Europe, and the servers where the Web Manager was deployed on, are located in the US. Moreover, we chose servers with low RAM (512MB), and with reduced CPU performance. The speed of 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 on better conditions.

After test suite was run first time, and patterns were created, we run 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 we got lot of false negative results, was hidden mainly in random data, by which was the application filled. Particularly, rows for tables consists from random data in the application

1. RichFaces Showcase - Screenshot from application is shown by figure 2.1. 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 tables components³). Secondly, there were unstable tests results because of timing issues. For example there is a slightly different delay for RichFaces tooltip⁴ component to be shown.

Therefore, we have decided to improve the stability of visual testing by introducing a way how to exclude some tests, or whole test classes from visual testing (they are still run in functional tests, as there they are stable enough). We introduced a JAVA annotation `@VisuallyUnstable` to the Graphene visual testing extension API (see 5.5.1). Listing 6.1 depicts how one particular test can be excluded from visual testing, and listing 6.2 shows how whole test class, and all its testing methods can be excluded at once from visual testing.

Listing 6.1: Exclude functional test from visual testing by annotating it with `@VisuallyUnstable`

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

Listing 6.2: 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 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 2.1), when manual testing was conducted. When there was a new version of RichFaces core library available.

The test suite was run in a way, that it tested RichFaces application with 4.5.1.Final libraries. Initially as we expected we got many differences (more than 200). The reason was that Showcase application shows on the bottom, in 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 application⁵.

Figures 6.2, 6.3 show generated diffs for these expected changes. Because they affect lot of visual comparisons, we had to do something to decrease the number of false negative tests. We used Arquillian Rusheye (see section 3.4) feature of masks to do it. It is a way, how to make arbitrary parts of the web page excluded from visual comparison. Currently, Rusheye support so called selective alpha masks, which are pictures with a transitive (alpha) layer, and also with non transitive parts which covers the parts of the patterns and samples, we would like to exclude from pixel to pixel comparison [30]. Our tool is not currently supporting feature of masks, as we did not recognized as a inevitable part of proving/disproving of our hypothesis (see chapter 4.1). It is indeed a very useful feature, which we will add to the tool in next development stages (see section 7).

6.1.1 Results

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

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

It is a very good result when taking into consideration 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 is 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/715607080a4c15bfff90af6546353e4b21a8391ee>

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

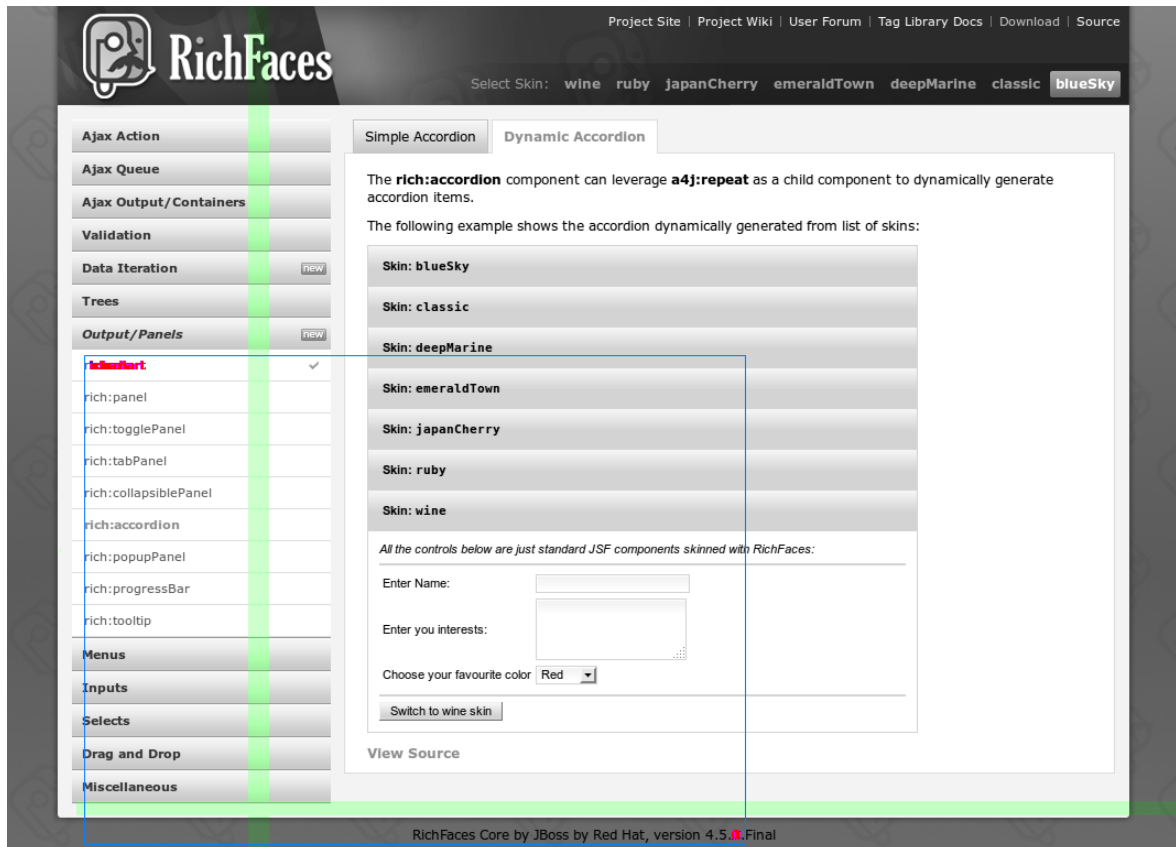


Figure 6.2: RichFaces chart component renamed in application menu.

6.2 Usage with CI

6.3 Cloud ready

Improvement of effectiveness of a quality assurance team was our primary goal, when developing the tool. This include 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 5.5.2).

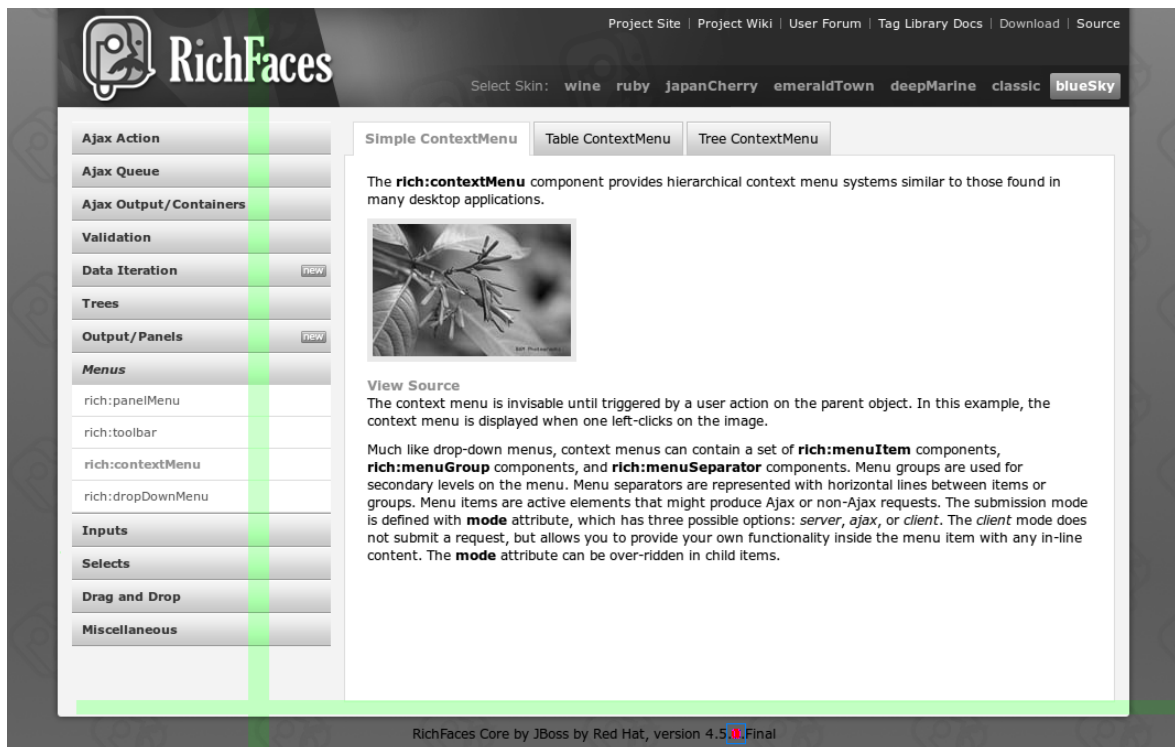


Figure 6.3: RichFaces version changed after it was upgraded.

We chose OpenShift open hybrid cloud by Red Hat, to proof 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 5.6.

To login into the application⁶ one has to use `LOGINNAME`, and password `PASSWORD`.

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.

6. Application is available at <http://jbosswildfly-jhuska.rhcloud.com/graphene-visual-testing-webapp>

7 Possible extensions

8 Conclusion

What I developed, What I improved, What can be better, Possible ways of extensions: Open-Shift cartridge

A Appendix A - GUI mockups of Web Manager UI

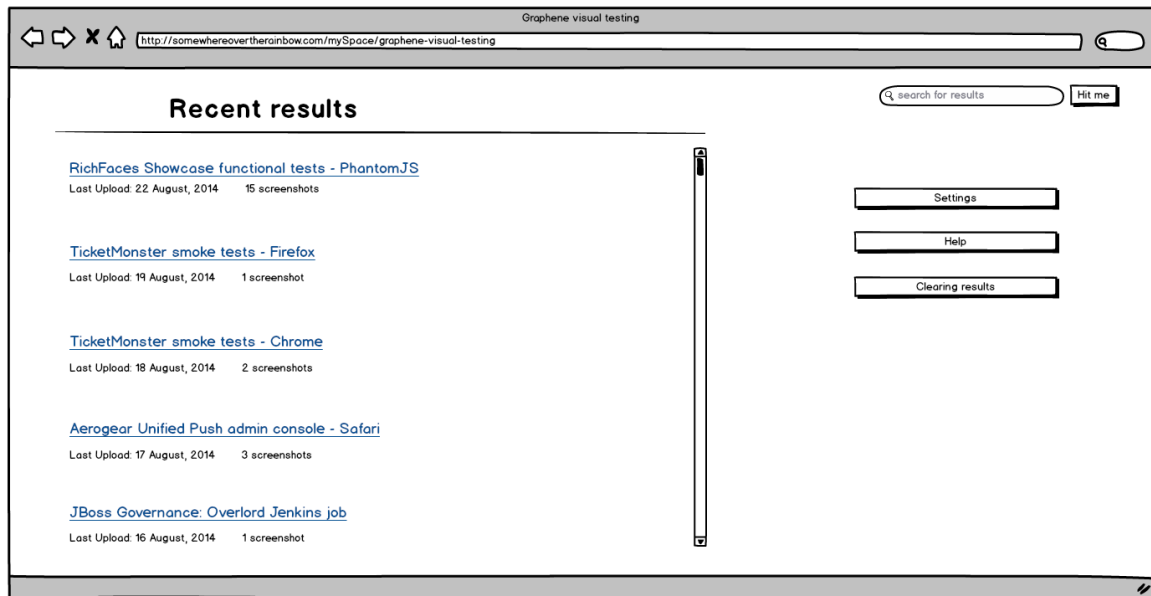


Figure A.1: GUI mockup for result viewer web application - front page.

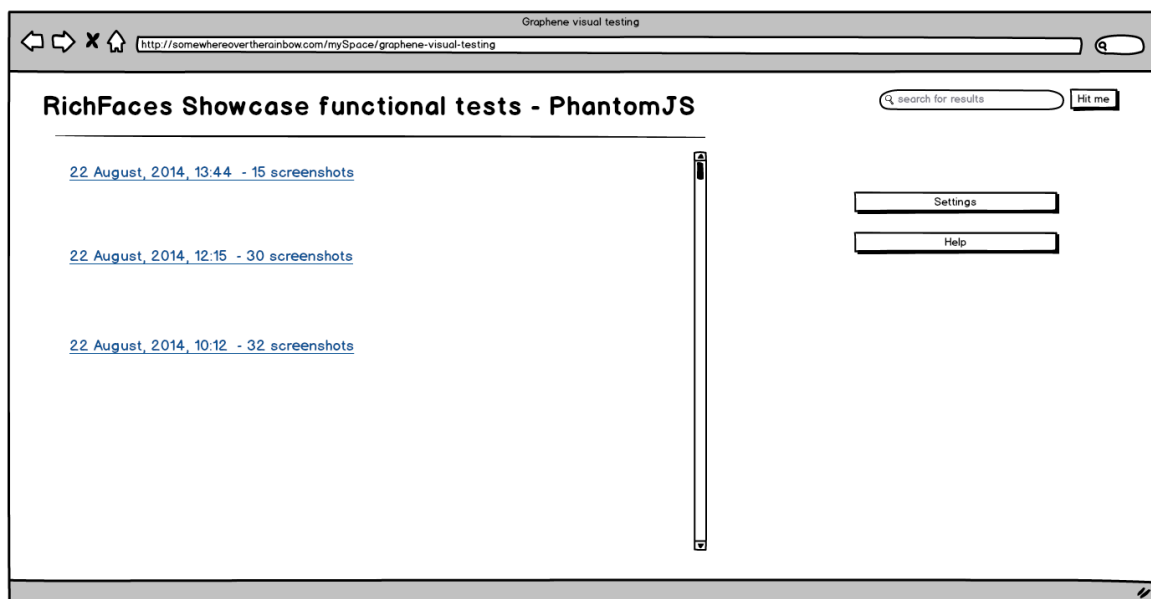


Figure A.2: GUI mockup for result viewer web application - particular test suite run.

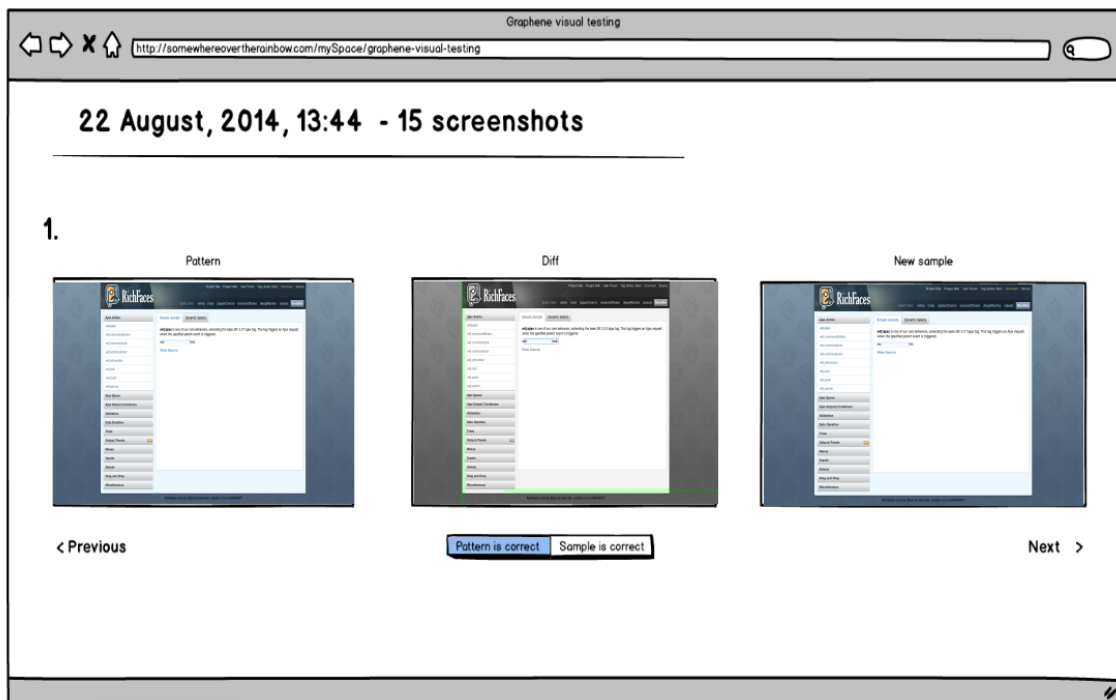


Figure A.3: GUI mockup for result viewer web application - actual comparison results.

B Appendix B - List of contributions to opensource projects

1. Graphene visual testing arquillian extension
2. Web Manager
3. ESTE DOKONCIT: RUSHEYE to Arquillian, not merged yet
4. RICHFACEs showcase test suite, not merged yet
5. Graphene screenshooter, an extension to Graphene, which enables taking of the screen-shots during functional testing. Available at: <https://github.com/arquillian/arquillian-graphene/tree/master/extension/screenshooter>.
6. 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

Screenshoty z web manazera.

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

Ako spustit samotne vizualne testovanie.

Bibliography