OptiRun

<Additional title pending>

Universitetet
i Stavanger

altibox

Janicke Falch

June 2016

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

Abstract

TODO

Acknowledgements

TODO

Contents

1
Introduction

Software systems, ranging from business applications to consumer products, have grown to form a fundamental element in our society. People are becoming increasingly dependent on computers, and wish to be in control of the digital content they consume. As a consequence of digital content being increasingly accessible, the importance of, and demand for, high-quality software has become substantial. Enhanced pressure on software vendors to deliver frequent releases of high-quality software requires efficiency in every stage of the software development process.

Software testing constitutes a central aspect to the software development process. It plays a critical role in quality assurance and defect detection, and is an important means to ensure that the software in question behaves as expected and according to specifications.

1.1 Origin

The subsidiary company of the Lyse Group, Altibox, is a telecommunication service provider. They offer a selection of services including broadband, *Internet Protocol Television* (IPTV) and *Voice over Internet Protocol*, (VoIP). Since its origin, the company has grown a great deal, and at a considerably higher pace than originally anticipated. Because the company had trouble keeping up with its rapid development, they neglected to apply a number of established business practices in several different areas, including the field of software testing.

IPTV has long been amongst Altibox' most prominent services, and has previously been available primarily through the use of TV decoders. In the later years, however, an important focus area for Altibox has been to expand the availability of their TV and *Video On Demand* (VOD) content by applying *Over-The-Top* (OTT) technology, which means that the content is made accessible over the Internet. This is accomplished through the development of *TV Overalt*, which is available both as a web application for desktops and as mobile applications for portable devices.

1.2 Motivation

All software testing of Altibox' products are currently performed manually. Altibox has wished to incorporate test automation in the test process of the web-based version on TV Overalt for some time, but have failed to make it a priority. Automated tests for web-based systems on the user's level generally run slowly, and Altibox therefore desired a system that would let them execute such tests efficiently. They also wanted the tests to be executed in a controlled environment.

This thesis presents OptiRun: a tool that will help Altibox incorporate test automation according to their needs. The tool is not limited to TV Overalt, however. It can, with little effort, be applied as a testing platform for a broad range of web applications.

1.3 Purpose

The aim of this project was to design and create a tool that would substantiate test automation with a distinguished intent of optimizing the time of test runs as well as provide control and feedback of test results. The tool is intended to be utilized during system or acceptance testing by technical test analysts.

The main area of use is execution of Selenium test scripts for web applications. Tests should be able to be executed remotely in a controlled distributed environment. An algorithm for optimally allocating test cases on the available test machines in the distributed environment should be provided. Further, an interface for uploading, managing, executing and scheduling test scripts should be included, and logs from results of previous test runs should be available. here. Upon a failed test, an option to report this in the issue tracking system should be provided.

1.4 Outline

The remainder of this thesis is organized as follows:

Chapter 2 provides a theoretical basis for the thesis by discussing some background information relevant to this thesis. Software testing, constrained optimization and some related work will be presented in this chapter.

Chapter 3 introduces the most essential tools and technology used in this project. The Python programming language, Selenium and Django will be presented here.

Chapter 4 presents the design and architecture of the OptiRun. The web service will be presented from the user's perspective.

Chapter 5 describes the implementation of the system. Some features are explained in detail.

Chapter ?? presents some experimental results.

Chapter ?? discusses and evaluates the results described in the previous chapter. A set of research questions will also be covered in this chapter.

Chapter ?? concludes this thesis and presents some suggestions for future work.

2
Background

Before moving onto the technical substance, it will be beneficial to acquire a theoretical basis on the subject of this thesis. This chapter introduces software testing and explains some essential concepts of relevance. Further, it presents constrained optimization as well as some previous work related to this thesis.

2.1 Software Testing

A fundamental understanding of software testing is a useful prerequisite in order to fully understand the context of this thesis. Consequently, this section provides a short introduction to software testing. Since this is a large field, only a small selection of relevant concepts and ideas will be presented.

Software testing is the process of evaluating the quality of the application, system or component being tested, commonly referred to as the test object or the system under test. Software testing may involve any action oriented toward assessing the software with the goal of determining whether or not it meets the required results. [?]

Software is developed by human beings who can make errors. These errors may cause defects in the source code. Executing defected code can lead to failure in the program [?]. One of the purposes of software testing is to examine the test object with the intent of revealing such defects. Other objectives include to measure and ensure quality, and to provide confidence in the product [6, pp. 8]

2.1.1 The V-Model

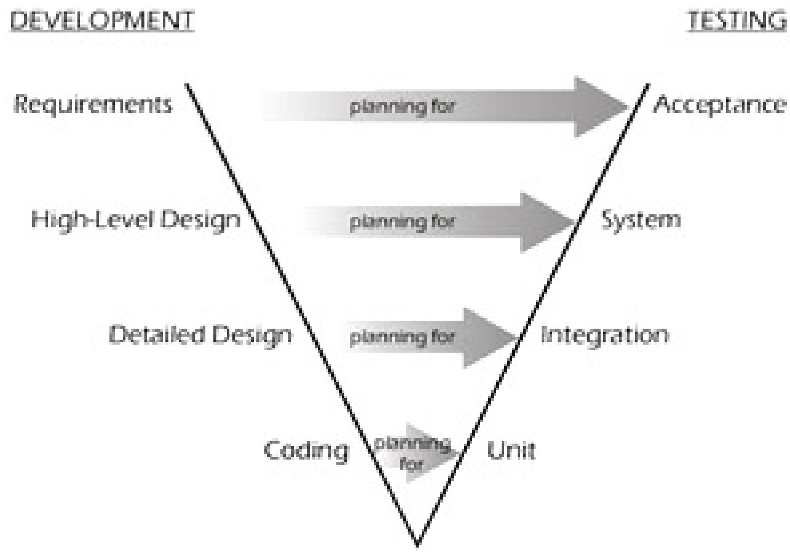

Figure 1: V-Model (NTS: Find a better version!)

The *V-model* is an important asset in software testing, behind which the central idea is to illustrate how each development task in a software development process have a corresponding testing task of equal importance. This is symbolized by the two branches of the letter "V" in the model. The development process is represented by the left branch which shows the system being gradually designed. The integration and testing process is represented by the right branch, which shows how elements are being assembled to form progressively larger subsystems, and then tested [6].

Depending on the literary source, the V-model covers a varying number of levels. The V-model shown in Figure 1 is obtained from [1], and covers four levels. The highest development level covers the gathering, specification and approvement of requirements. Acceptance testing correspondingly checks if these requirements are met. Further, high-level design covers the functional design of the system, and corresponds to system testing, which aims to verify if the system as a whole meets the required results. Detailed design covers technical system design and component specification, and integration testing correspondingly verifies that the different components work together as specified. The lowest level is the coding in which the specified components (modules, units and classes) are implemented. It corresponds to unit and component testing. The tests at this level aim to verify that system components perform as specified, by testing them in isolation. [6]

OptiRun is a tool for high-level test execution, and can be applied to the two highest test levels of the V-model; system testing and acceptance testing.

2.1.2 Black-Box and White-Box Testing

Software testing can be divided into *black-box* and *white-box* testing. White-box testing is based on analysis of the internal system structure, and requires knowledge of the source code. In black-box testing, the test object is seen as black box, whose behavior is watched from the outside. The inner structure of the system is either unknown or unconsidered. Test cases are determined from the specifications of the system. Black-box testing is predominantly used for higher levels of testing. The test object is accessed through the user interface (UI). Tests are performed from the perspective of an end user, and aim to mimic a human being interacting with the system. Black-box testing includes functional testing, which is used to validate a particular feature for correctness according to the requirements specifications. [6]

OptiRun performs tests by using Selenium to automate web browsers. It executes functional UI tests on full system builds, and run without access or consideration to the source code and the internal system structure. The test method covered by OptiRun is therefore black-box testing.

2.1.3 Test Automation

As opposed to humans performing software tests manually, test automation is the practice of writing scripts that conduct tests when executed. Automation can be applied to tests at any level. Low level tests, such as unit and component tests, generally run fast and are durable since they are isolated from changes in other parts of the system. These tests may be executed frequently. As we move up to the integration level, the tests does not run as fast, and become less durable since they are dependant on multiple components working together in subsystems. This thesis considers the tests at the two highest levels; system and acceptance tests. These tests are is concerned with testing the system at the perspective of the end user.

These tests require the system to work as a whole, and thus generally run slower and are more brittle. [9] Because of this, functional UI tests are oftest the fewest.

There are a number of strengths to automated tests. They are superior at verifying logical functionality. They can be executed any number of times, and they run more quickly than a human interacting with the system, thus saving time and reducing effort. Time saved on manual testing can be used to increase the test coverage, which can provide reduced risk. Tests and tasks that would be error prone if done manually and tests that are repeatedly performed are typical subjects for automation [4]. This includes regression testing, which is used to verify that defects have not been introduced in a new version of previously tested software [6].

Manual human testing has a number of benefits too. Human testers can identify corner cases and check how the system responds when it is used in manners it is not designed to be used for. They can also evaluate aesthetics and design of the UI as well as the overall user experience. In most cases, automated tests should not be a complete replacement for manual human testing. They should rather complement each other.

2.1.4 Continuous Integration

Automated tests can be applied to a continuous integration (CI) environment, which automatically builds and tests code contained in specified repositories. It can provide frequent and rapid feedback on the code in uploaded commits, and can thus be of excellent value for software developers [7].

Jenkins is a commonly used CI tool, which offers a plugin for running Selenium tests. This could be used to apply CI to OptiRun, but has been omitted for two main reasons.

Firstly, Selenium tests run on real browsers. They perform real browser operations, often having to wait for HTTP responses. The browsers also need to open prior to a test execution and close afterward. All of these tasks are a great deal more time-consuming than tests on lower levels. Including Selenium tests in a CI process as part of the build would render each build hugely time-consuming.

Secondly, in Altibox' case, software development and software testing are performed separately by different teams. The test team, to which OptiRun is aimed, does not have access to the actual code, and would not gain the same benefits from .

2.2 Constrained Optimization

TODO

2.3 Related Work

Some work has previously been done on the subject of test case scheduling and automation of functional UI tests for web applications. This section presents the scheduling method *TC-Sched* and the automated testing platform *Sauce Labs*.

2.3.1 TC-Sched

TC-Sched is a time-aware method for minimizing the execution time of test cases within a distributed system with resource constraints. The method was proposed by Mossige, Gotlieb, Meling, and Carlsson in [13]. The authors define the *optimal test case scheduling* (OTS) as the optimization problem of finding an execution ordering and assignment of all test cases to machines, in such a way that each test case is executed once, no global resource is used by two test cases at the same time, and the overall test execution time is minimized.

TC-Sched addresses situations in which some operations may require exclusive access to one or more *global resources*, such that only one test case can access each resource at a given time. To include the aspect of global resources in this project would have been redundant since there are no such global resources involved. Apart from that, TC-Sched is a closely related scheduling method that has been of inspiration. The implementation of the scheduling algorithm used in this project will be described in Chapter 5.

2.3.2 Sauce Labs

Cloud testing services use cloud computing environments as a means of simulating real-life user traffic. Sauce Labs is a cloud-hosted platform for automated testing of web and mobile applications. One of the founders, Jason Huggins, were also the original creator of Selenium [11], which will be introduced in Section 3.2.

Sauce Labs bear some similarities and some dissimilarities with the design of OptiRun. Both platforms are frameworks for automated functional UI testing of web applications through executing Selenium test scripts.

They both execute tests remotely in distributed environments through the use of Selenium Grid. However, whereas Sauce Labs perform test execution on virtual machines (VMs) that are created prior to each test run and destroyed immediately after [10], OptiRun requires the use of either real, physical machines, or customized VMs, that remain intact after use. Because of the number of machines in OptiRun being limited to the machines being connected to the distributed system, an algorithm for optimizing the overall execution time of test runs is also needed here. While Sauce Labs executes tests in their own data centers in a remote location and their own environments, OptiRun performs tests on the test object locally and in an environment intended for testing of the given test object, in this case TV Overalt. Further, Sauce Labs does not automatically create issues in the issue tracker. Sauce Labs is a paid service for which customers pay to gain access to using a limited number of concurrent virtual machines for a limited amount of time each month. The optimization algorithm in this project also distinguishes the two projects even further.

3
Technology

A number of distinctive tools and technologies have been applied throughout the work on this project. This chapter briefly introduces the most essential of these tools and technologies.

3.1 The Python Programming Language

The Python programming language has largely been used in the conduction of this project. Python has efficient high-level data structures, and is known for being simple, yet powerful. Since Python programs are executed by an interpreter, it is considered an interpreted language [12].

Building this project on Python was a natural choice for several reasons. Python is compatible with Selenium, which is introduced in Section 3.2. Other reasons include offering a large set of libraries and having high-quality documentation. Additionally, Python supports multithreading and is suitable for running a program as a service. Details of the implementation will be discussed in Chapter 5. Version 2.7.11 of Python has been used in this project.

3.2 Selenium

Selenium is an umbrella project for automation of web browsers. Several different tools and libraries are included in the framework, all with the common goal of supporting browser automation. Selenium can be used to automate different types of browser jobs such as web-based administration tasks, but is primarily used for test automation. The project is released under the Apache 2.0 license and is thus free and open-sourced. Selenium has language bindings for several different programming languages, including Python. The Selenium tools used in this project will be introduced below.

3.2.1 Selenium WebDriver

Selenium WebDriver consists of a set of libraries that helps with automation of tests for web applications, and a set of executable WebDriver files, one for each browser, that perform the actual automation specified in the WebDriver scripts. Selenium WebDriver interacts directly with the browser by sending calls using the native support for automation for each browser [5]. Selenium WebDriver runs on Windows, Linux and Macintosh, and supports most conventional web browsers, including Firefox, Chrome, Internet Explorer, Opera and Safari.

The WebDriver is used to create a new instance of the requested web browser. It is used to fetch a given web page, and then locates UI elements. When an element is located, the WebDriver can be used to perform an action on the element, such as clicking a button, checking a checkbox, or populating a text field.

Listing 1: Selenium WebDriver Example

```python
from selenium import webdriver

# Create a new instance of the Chrome WebDriver
driverdriver = webdriver.Chrome("<file path>/chromedriver.exe")

# Go to the Altibox TV Overalt start page
driver.get("https://tvoveraltstg.altibox.no/")

# Locate and click the Login button
login_button = driver.find_element_by_class_name('btn-login')
login_button.click()
```

Listing 1 shows a simple example of how Selenium WebDriver can be used with Python to open the Chrome web browser, navigate to the Altibox TV Overalt start page, locate the *Login* button, and click it. OptiRun revolves around executing scripts that uses the Selenium 2.0 libraries and WebDrivers to execute automated web browser tests.

3.2.2 Selenium Grid

Selenium Grid is a tool for executing Selenium tests on remote machines in a distributed environment, and thus allowing for parallel execution. This also opens up for running tests on different operating systems.

Reasons for wanting to incorporate Selenium Grid include being able to run tests against multiple browsers, browser versions and browsers running on different operating systems, and to reduce the execution time of the tests. In practice, a grid is made up of one Selenium Grid Server; a *hub*, and one or more slave machines; *nodes*, all running a Selenium Standalone Server. The nodes use Selenium WebDriver to communicate with the hub through a JSON wire protocol [2].

Selenium Grid 2.0 was used in this project.

3.3 Django

Django is a high-level web development framework that is implemented in the Python programming language. It encourages rapid development and enables efficiently maintainable web applications of high quality. The framework is a free and open-sourced project maintained by the non-profit organization *the Django Software Foundation*, who describe the framework as fast, secure and scalable [3]. Comparable to Selenium, Django is also essentially a collection of Python libraries [8]. The Django labraries can be imported and used to implement web applications. Some additional HTML, CSS and JavaScript code has been applied along with the Python code.

Aside from allowing rapid progression of development, one of the main reasons for choosing Django rather than building the dashboard from scratch or using a different web framework, is its powerful administrator site. An administrator site was exactly what was needed to build the dashboard of OptiRun. Another contributing factor was to provide consistency and the ability to communicate seamlessly with the remaining parts of the system, since Django builds on the same programming language as the rest of the system. The dashboard will be presented in Chapter 4. Details concerning the implementation will be explained in Chapter 5. The Django version used in this project is version 1.9.

4 System Overview/Design

This chapter aims to provide an overview of how OptiRun is assembled by describing the design and architecture of the system. System setup will be described thereafter, followed by a presentation of the web-based user interface of the tool.

4.1 Architecture

OptiRun is operated through a web-based dashboard. This web service works as a content management system for test scripts. It allows for sending execution requests for the available test scripts, or for scheduling test runs ahead of time.

Figure 2 shows the structural architecture of the system. In this case, the web server, the controller, the database, the file storage and the Selenium Grid Hub are all located on the same machine, hence the dashed border around these elements in the figure.

The web server uploads test scripts to a file location on the server. Meta data about the test cases as well as other information such as test groups, planned test executions, test results and user authentication are also stored in the database. The web server can report defects in the issue tracker system, which in the case of TV Overalt is JIRA.

The controller listens to test execution requests triggered through the dashboard. Upon an execution, it fetches the given test script from the file

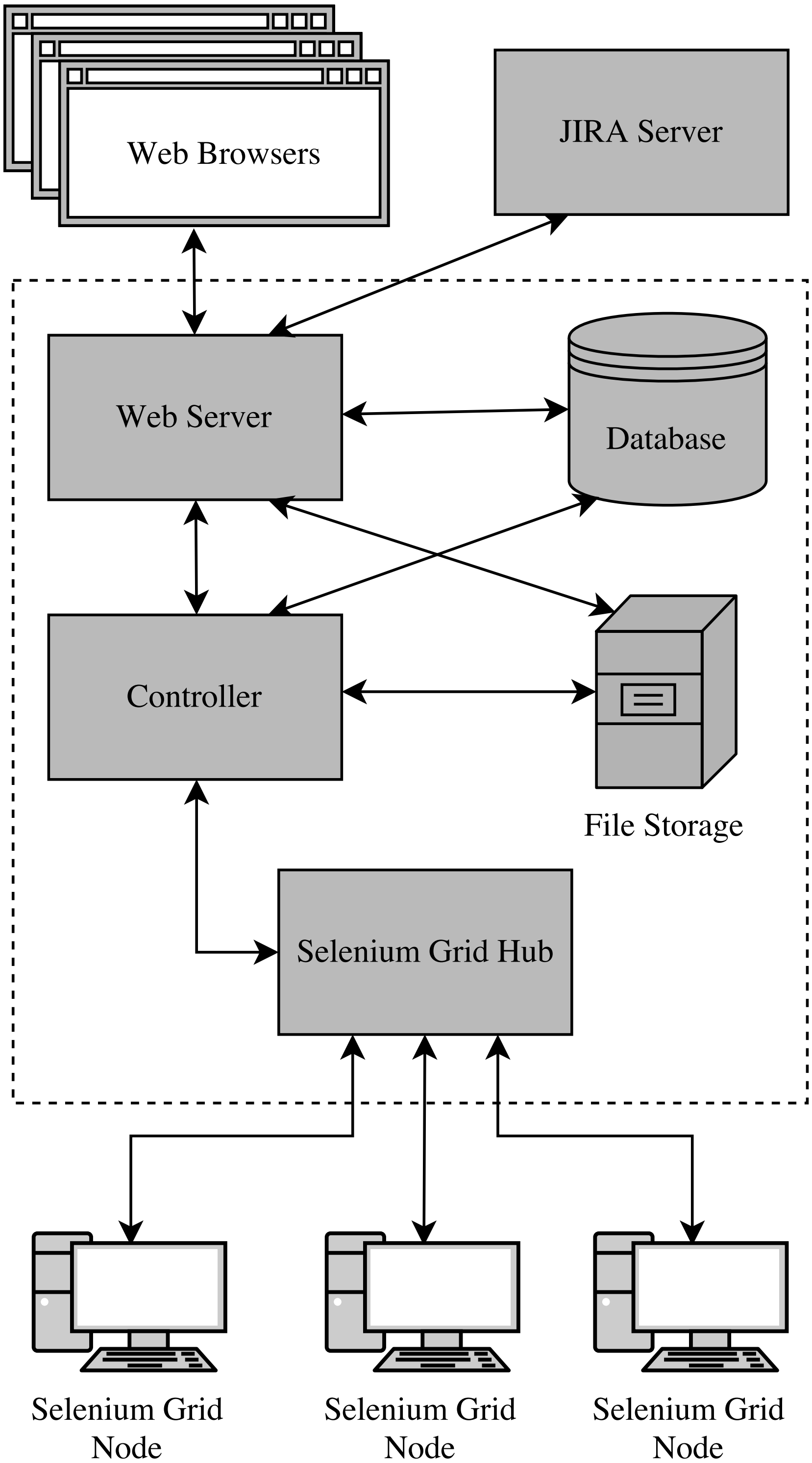

Figure 2: System Architecture

storage. Test scripts are sent to the current instance of the Selenium Grid Hub with specifications of which node in the distributed Selenium Grid that the test script should be executed on. The hub then triggers the given node to execute the test. After the test has finished, the controller stores the result in the database.

4.2 Setup

TODO

4.3 Dashboard

The graphical user interface of OptiRun is web-based, and is built on the high-level Web framework Django (3.3). The framework provides a powerful automatic administrator interface, which has been used as a foundation for the development of the dashboard. Working with Django thus allowed for rapid development while still retaining control of the content and functionality of the website. This was one of the main reasons why this particular framework was chosen for OptiRun.

This subsection will present the dashboard from the user's perspective. An in-depth presentation of how some of the functionality has been implemented can be found in Chapter 5.

4.3.1 Home Screen

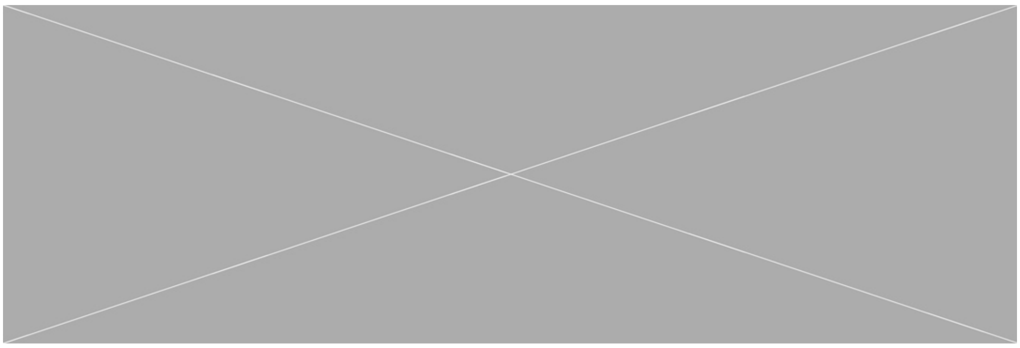

Figure 3: Home Screen

After logging in, the home screen is the first thing that meets the eye of the user. This screen provides navigation to all of the modules and pages described in the remaining subsections of this section. In order to give the user an indication of status, some graphs that show the results of recent test runs, as well as key numbers and other important information is displayed on this page.

4.3.2 Authentication and Authorization

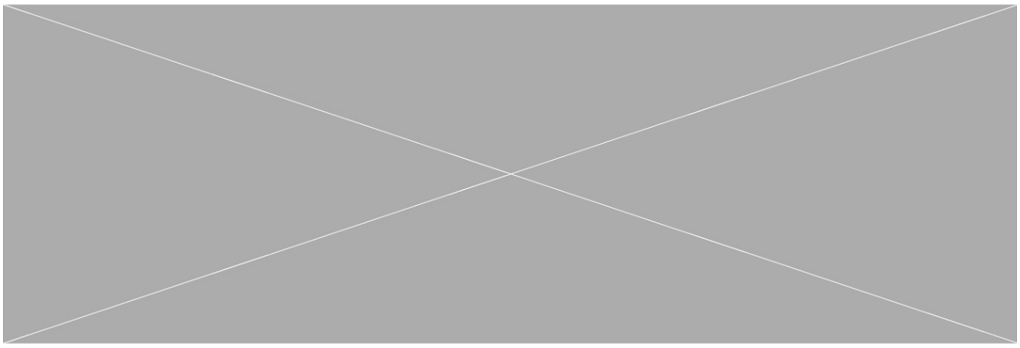

Figure 4: Authentication and Authorization Module

The Django administrator interface provides some built-in functionality, including a module for authentication and authorization of users. Superusers can list, edit and create new user accounts. They can also change user info and the permissions of other user accounts, add them to user groups, and specify the permissions of user accounts.

OptiRun is a tool that is meant to be part of a business process in which only trusted peers should be allowed to take part, and by extension OptiRun should only be operated by trusted associates. For this reason, no user accounts can be created by anyone other than a superuser. Thus, any person who wants and account must request this by someone with superuser privileges.

By default, all passwords are encrypted before being stored in the database.

4.3.3 Settings/Configuration

TODO

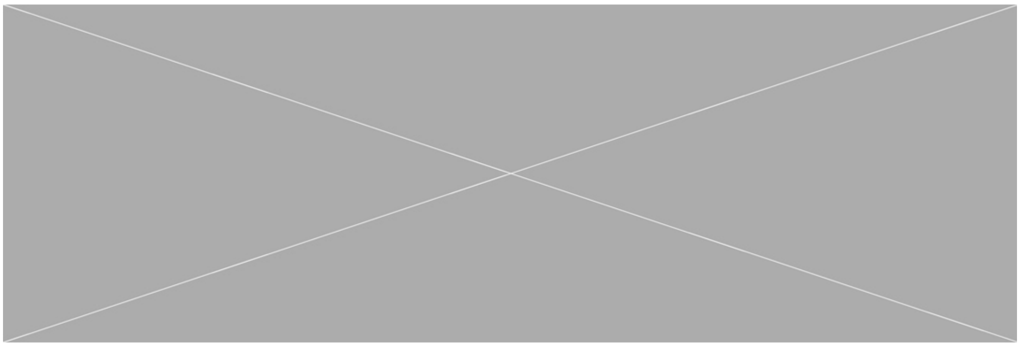

Figure 5: Test Case Module

4.3.4 Test Case Module

All registered test cases are listed in this module. The list contains meta data about the test cases as well as some values that are retrieved or calculated using information from other tables in the database. This includes the number of times the test case has been executed, the average execution time, the result of the previous execution and when the test case was last executed. It is possible to search and filter the list to narrow down the elements listed, and to sort the list by clicking on the attribute names in the table header.

New test cases can be created by clicking the button labeled *Add Test Case* in the upper right of the view. This opens a new page with a form in which the test script can be uploaded and corresponding details about the test case can be filled in. Which groups the test case should be a member of can also be specified here. Test cases can be members of multiple groups, or none. After a test case has been created, it can be edited by clicking on the title of the test case in the test case list.

Actions can be made to selected test cases. The selection of actions are located in a dropdown list containing two options; *Delete selected test cases* and *Execute selected test cases*. The former is a built-in Django function which, after the user confirms deletion in and intermediate page, removes all meta data regarding the selected items from the database. The latter action is implemented for OptiRun. The action takes the user to an intermediate page in which they can specify the platform and browsers the test cases should be executed on.

When an action has been performed, or attempted performed, a feedback message telling if the action was successful or not is displayed on the screen.

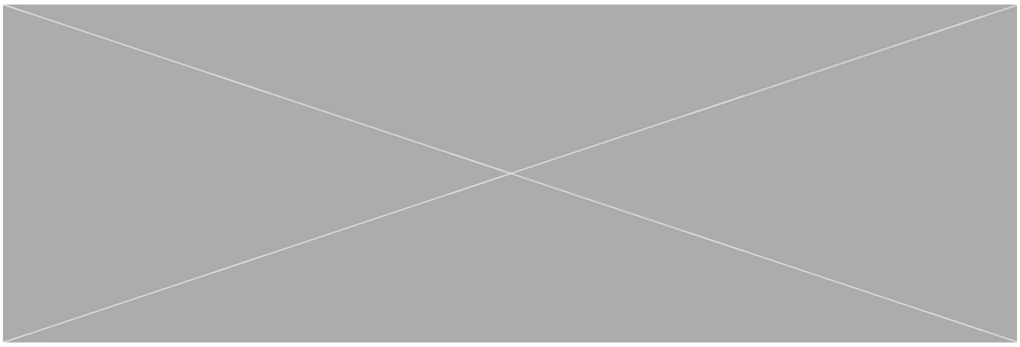

Figure 6: Intermediate Page for Immediate Test Execution Requests

4.3.5 Test Group Module

Test groups are listed similar to individual test cases as described in the previous subsection. As with the test cases, group items can also be searched, filtered and sorted. Also, new groups can be created, and existing groups can be edited.

However, the group model is simpler than the test case model. The only editable information about the groups are title, description and which test cases are members of each group. The purpose of this model is to group together test cases which should preferably be executed in the same test run.

4.3.6 Scheduling Module

The scheduling module provides an interface for planning test runs in the future. As with the test cases and groups, schedules are also displayed in a list can be searched, filtered and sorted. Schedule objects can be created and changed. Additionally, the schedule list enables schedule objects to be activated and deactivated from the action menu. Only activated schedule objects will start test runs according to their specified time of execution. This is the default status, and deactivation of schedule objects must be appointed explicitly through the action menu.

Upon creating or editing a schedule object, an execution time can be specified along with any desired recurrence pattern along with the option of providing an end time. Test groups or individual test cases that should be included in the schedule object are also specified here.

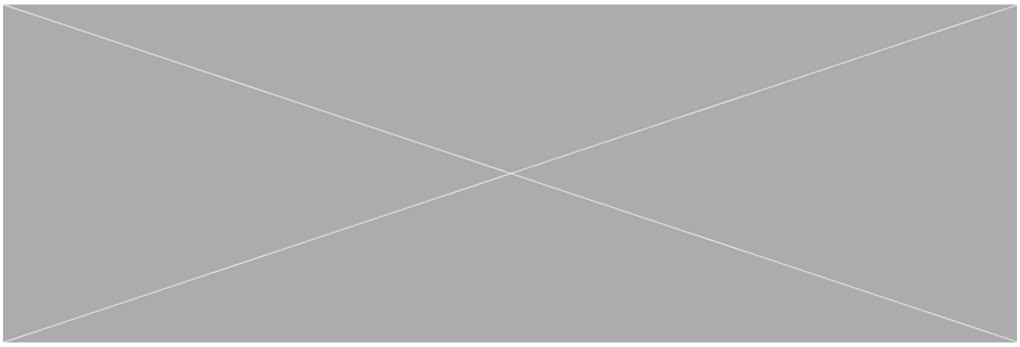

Figure 7: Schedule Creation Form

4.3.7 Execution Log

The layout of the log is the same as the remaining list views, but the functionality differs in that log items are read-only. They can not be created or deleted. Log items are created and inserted into the database based on the result from test executions. Each log object corresponds to an execution of an individual test script. The information listed in the logs include a number of attributes such as the duration of the test execution, the IP address or hostname of the machine on which it was executed, as well as information such as browser, platform, output and of course the result.

4.3.8 Other

The Help page contains miscellaneous information regarding how OptiRun works and how to use it. The page includes a quick walk-through of the tool, how-to's, and setup guide. This page can be of help to technical testers who wants to use the tool, and otherwise to those who wants insight of how it works.

The About page provides a short presentation what OptiRun is and how it came to be. Any necessary licenses are also included in this page.

5 Implementation

TODO: Write chapter intro

5.1 Data Flow

TODO

5.2 Selenium Grid Integration

TODO

5.3 Controller

TODO: Write section intro

5.3.1 Listening for Immediate Execution Requests

TODO

5.3.2 Checking Planned Execution Schedule

TODO

5.3.3 Queue

As test executions are requested by the controller, each individual test case is placed in an execution queue. The queuing system is made up of four distinct queues with corresponding descending priorities; one for urgent execution, one for immediate execution, one for planned execution, and one for trial execution. Each test case is placed in a queue according to which action triggered their request to be executed. If it was identified by the listener for immediate execution, it is placed in the corresponding queue. If a test case was identified by the schedule listener, the test case is placed in the queue for planned execution. If the test case have just been uploaded or updated, it will be placed in the queue for trial execution. The urgent queue is where test cases that were not executed because the test node they were allocated to crashed.

Sometimes a planned execution and an immediate execution of the same test case with the same browser and platform specifications, can be requested at the same time. A similar situation could occur when two schedule objects that are due at the same time includes the same test case with the same specifications. Scenarios such as these could potentially lead to time and resources being wasted on performing the same job twice. To avoid incidents in which two identical test cases are executed at the same time, a duplicate check is performed each time a test case is added to one of the queues. If there are duplicates, the test case is removed from the queue with the lowest priority.

The test case executor always checks the immediate queue first, then the planned queue, and lastly the trial queue. If the test executor finds test cases in any of the queues, it empties the queue and handles the containing test cases as a group.

5.3.4 Resource Allocation Algorithm

TODO

5.3.5 Test Execution

After test cases have been retrieved from the queue, allocated amongst the pool of available test machines, and sorted, it is time for execution.

A thread is started for each of the test machines. In these threads, each test case is started as a subprocesses using a shell command in which information regarding the desired test node and browser of the test execution are passed to the test script as arguments.

Immediately before a test case is executed, the test node is pinged to see if the connection is still up. This check is also done immediately after any failed test. If the node has crashed or otherwise failed before or during the execution, the test cases are moved to the queue for urgent executions, and attempted executed once the current test run has finished.

If there no test machines match the required browser/platform specification of a given test case, it is not executed. It will appear in the execution log, but will be marked as not executed.

5.4 Database

The relational database management system (RDBMS) SQLite can be regarded as a light-weight substitute to other SQL based database engines. Benefits to SQLite compared to these other database systems include its being self-contained and serverless, and the database is contained in a single disk file [**?**]. For convenience in regard to submission, SQLite was the preferred choice for this project. However, this can easily be replaced by another SQL database engine with little effort, if needed.

As explained in Chapter 3, Django data models define the database layout, and each model typically maps to an individual table in the database. SQL code for creating the database itself and the tables within it are all auto-generated by Django, based on the implementations of the models. If a m-to-n relation between two different models are defined in the model implementations, a separate relationship table is created in the database to cover this. The auto-generated SQL files covering creations and changes to the tables related to test automation are all placed under *testautomation* → *migrations* by default. When a change as been done to one of the models, a new migration can be created by performing a few shell commands [**?**].

The database can be accessed either by writing raw SQL queries, or through Django's API for database abstraction. The former approach was first implemented in this system, but as the latter proved to be cleaner and more consistent with the implementation of the web service and thus the project as a whole. It was also interesting to use a different procedure of database communications than the more commonly used raw SQL queries. Listing 2 shows a simplified example of how an INSERT statement is conducted using this approach. In this example, the implementation of the

Log model is imported, and then an object of this type is created with pseudo values for a small set of attributes. Line 5 in the listing represents the transaction execution and commit. Note that implementation of such object creations includes several more attributes than what is shown in this example.

Listing 2: Database Communication Using Abstraction API

```
1  from testautomation.models import Log
2  from datetime import datetime
3
4  l = Log(title='test1', result=1, execution_time=datetime.utcnow())
5  l.save()
```

In addition to providing excellent readability, the abstraction greatly decreases the required number of code lines needed to achieve the same result compared to executing raw SQL queries. This is because the database location does not need to be stated, connection with the database does not need to be programmatically established and then closed when the transaction is finished, and so forth. The abstraction takes care of all of this. The running time of the two approaches also proved to be approximately the same after testing both methods. Other types of queries such as Select, Update and Delete are also supported with this API.

Information about test cases, test groups, schedules, logs as well as user information is all stored in various tables of the database. Figure 8 shows the Entity-Relationship (ER) diagram of the database.

All timestamps stored in the database are in the Coordinated Universal Time (UTC) standard, which, as the name suggests, is universal, and therefore independent of timezones. The timezone used in the web service is set to 'Europe/Oslo' in the Django settings file. Whenever a timestamp is shown on screen, it is first converted to the specified timezone using Django's timezone library. If the user is located in a different timezone than the one specified in the settings, a label explaining that the computer is *xx* hours ahead or behind of server time is displayed next to any datetime picker, such as in the schedule creation form.

Storing timestamps according to the UTC standard rather than the current timezone can be considered good practice for multiple reasons. Firstly there can be no ambiguity. Confusion and misunderstandings related to conversion across different timezones will be avoided, which also means that timestamp calculations are simple. Further, there can be no invalid dates

Figure 8: ER Diagram of Database

linked to daylight savings time. Moreover, if the server were to be moved to a different timezone, timestamps would have to be converted.

5.5 Dashboard

TODO: Write section intro

5.5.1 Models

With Django, data models provides the foundations on which database tables are created and maintained. A model implementation can be seen as a Python equivalent to an SQL CREATE TABLE statement. Model implementations are translated to SQL and executed by Django. Each model, which is a subclass of *django.db.models.Model*, represents a table in the database, and each model field represents a database field. Similar to SQL, the datatype of each field along with any other specifications such as the maximum length of a textfield, default values and help text can be passed as *field option* parameters.

The models are located in *testautomation* → *models.py*. In this file, model specifications of test cases, groups, schedules and logs are implemented.

Additionally, models can contain elements that are not linked to the database, such as a field of which a value is based on a function that is performed each time the value is requested displayed. These functions can return primitive datatypes, but they can also contain HTML tags, so the content can be customized. Such functions are used in several of the models. For instance in the test case module, there are one function that query the LOG table in the database, and counts number of instances linked to the particular test case. Average duration and previous execution date are retrieved in a similar manner.

Listing 3: Model Implementation

```
from django.db import models

class TestCase(models.Model):
  title = models.CharField(max_length=80)
  script = models.FileField(upload_to='scripts')

  def times_run(self):
     tr = # perform query
```

```
        return tr
```

Listing 3 shows a largely reduced version of how the test case model has been implemented. This model contains two model fields and a function. The *script* field is of the type *model.FileField*, and the directory that the files should be uploaded to is passed as a parameter. The file upload itself is taken care of by Django.

5.5.2 Admin

Listing 4: Implementation of Model in Administrator Interface

```
1  from django.contrib import admin
2  from .models import TestCase
3
4  class TestCaseAdmin(admin.ModelAdmin):
5    list_display = ('title', 'times_run', 'avg_dur')
6    search_fields = ['title',]
7    fields = ('title', 'script')
8
9  admin.site.register(TestCase, TestCaseAdmin)
```

Listing 4 shows a very reduced interpretation of how the test case model is represented in the administrator interface. This adaption builds on the model implementation from Listing 3. Model administrator representations are subclasses of *django.contrib.admin.ModelAdmin*, and specifies how the model should be represented in the administrator interface. This interpretation specifies values of three of the many ModelAdmin options; which fields of the model should be displayed in the overview list of the test case module ('list_display'), which fields should be searchable ('search_fields') and which which fields should be present in the creation/change form ('fields'). Line 9 registers the model to the administrator interface with the specifications stated in the TestCaseAdmin class.

In the actual implementations of the model admin representations, a number of additional fields and specifications are also included. Custom forms with particular validation functionality can be integrated with the creation/change form. This has been done with test case objects to ensure that only test scripts that fulfill certain criteria can bine uploaded. Admin actions are also implemented here.

TODO: List and describe the implemented admin actions

5.5.3 Event Recurrence

Rrule is a part of to a Python library called *dateutil*, which provides an extension to the datetime module of Python. Rrule stands for recurrence rule, and is a small and fast library used in OptiRun to specify if, when and how a test run should be repeated. Instances of rrule can be implemented multiple ways. In this project, it is achieved through passing a string with a specific format, containing information about the desired recurrence constraints. This string is called an rrulestr. This string is stored in the database, and can be used at any point to create an rrule object and inquire when the next event should occur. An example of such a string, how an rrule object is instantiated in this project, and how the next occurrence is retrieved can be seen in Listing 5.

Listing 5: Recursion Rule

```
from dateutil.rrule import rrulestr
from datetime import datetime

rule_string = "DTSTART:20160615T160000\nRRULE:FREQ=WEEKLY"
rule = rrulestr(rule_string)

print rule.after(datetime.now())

>>> 2016-06-15 16:00:00
```

The string in the listing above is used to create an rrule instance in which the first occurrence is set to the 15^{th} of June 2016 at 4 pm, and repeats weekly. In addition to the recurrence properties shown in the listing above, the library provides an extensive number of recurrence options, including end date, occurrence count and interval.

The recurrence rule strings in this project are built dynamically when a schedule object is created or updated. In the test case, group and schedule modules of OptiRun, recurrence rule strings are used to calculate the time of the next planned execution for each particular test case, group and schedule. Rrule instances are also created by the controller to check when the next test run is scheduled.

Figures

Listings

References

[1] Craig, R. D. and Jaskiel, S. P., 2002, *Systematic Software Testing*. URL: http://flylib.com/books/en/2.174.1.1/1/. [ONLINE], Accessed: 2016-03-14.

[2] *Distributed Testing with Selenium Grid*. URL: https://www.packtpub.com/sites/default/files/downloads/Distributed_Testing_with_Selenium_Grid.pdf. [ONLINE], Accessed: 2016-03-13.

[3] *Django: The Web Framework for Perfectionists with Deadlines*. URL: https://www.djangoproject.com. [ONLINE], Accessed: 2016-01-31.

[4] Mitchell, J. L. and Black, R., 2015, *Advanced Software Testing - Vol. 3, 2nd Edition: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*.

[5] *Selenium WebDriver — Selenium Documentation*. URL: http://www.seleniumhq.org/docs/03_webdriver.jsp. [ONLINE], Accessed: 2016-02-02.

[6] Spillner, A., Linz, T. and Schaefer, H., 2011, *Software Testing Foundations, 3rd Edition*.

[7] Fowler, M., *Continuous Integration*. URL: http://martinfowler.com/articles/continuousIntegration.html. [ONLINE], Accessed: 2016-03-13.

[8] Holovaty, A. and Kaplan-Moss, J., 2007, *The Definitive Guide to Django: Web Development Done Right*. URL: http://www.djangobook.com/en/2.0/index.html.

[9] Roberts, J., 2013, *Automated Testing: End to End*, Pluralsight. URL: https://app.pluralsight.com/library/courses/automated-testing-end-to-end/table-of-contents. [ONLINE], Accessed: 2016-02-11.

[10] Sauce Labs, *Sauce Labs Features*. URL: https://saucelabs.com/features. [ONLINE], Accessed: 2016-03-14.

[11] Sauce Labs, *Sauce Labs Press Coverage*. URL: https://saucelabs.com/press-room/press-coverage/news-2. [ONLINE], Accessed: 2016-03-14.

[12] Swaroop, C. H., 2004, *A Byte of Python.* URL: http://python.swaroopch.com/. [ONLINE], Accessed: 2016-02-12.

[13] M. Mossige, A. Gotlieb, H. Meling, and M. Carlsson. *Optimal Test Execution Scheduling on Multiple Machines with Resource Constraints.* Technical report.