SMOKE TESTING BENCH FOR FREENEST SERVICE

Teemu Ojala

Bachelor's Thesis
May 2012

Degree Programme in Information Technology
Technology, ICT

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) OJALA, Teemu	Julkaisun laji Opinnäytetyö	Päivämäärä 02.05.2012
	Sivumäärä 61	Julkaisun kieli Englanti
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi
SMOKE TESTING BENCH FOR FREENEST SERVICE

Koulutusohjelma
Tietotekniikka

Työn ohjaaja(t)
LEINO, Janne

Toimeksiantaja(t)
RINTAMÄKI, Marko

Tiivistelmä

Opinnäytetyön tarkoituksena oli luoda kolmenkymmenen automaattitestin sarja FreeNEST-ohjelmiston eri komponenttien testaamiseen. FreeNEST on Jyväskylän ammattikorkeakoulun sisällä toimivan SkyNEST-projektin kehittämä selainpohjainen projektitöissä käytettävä ohjelmisto, joka koostuu useista eri avoimen lähdekoodin työkaluista. Työssä luotu testisarja pyrkii korvaamaan useasti toistettavien testien manuaalisen suorittamisen automatisoiduilla testeillä, jotka voidaan ajaa käyttäen Robot Framework nimistä testiautomaatiotyökalua.

Testit on tarkoitus ajaa käyttäen hyödyksi SkyNEST-projektin omaa OpenStack-pohjaista pilvipalvelua, jota myös kehitetään tällä hetkellä projektin sisällä. Työn teoriaosassa on vertailtu automaattitestauksen ja manuaalitestauksen etuja ja rajoituksia sekä esitelty eri testauskäytäntöjä ja testien luomisessa ja ajamisessa käytetyt työkalut ja ohjelmat.

Työssä on myös esitelty yhden testitapauksen valmistuminen askelittain käyttäen apuna RIDE-editoria, joka on Robot Frameworkin oma testienluontityökalu.

Testien luonti onnistui ajallaan ja niiden ajaminen OpenStack-pilveen luoduilla virtuaalikoneilla voitiin suorittaa. Testien kattavuus kuitenkin jäi tiettyjen testien suorittajaan liittyvien rajoitusten takia puutteelliseksi ja täten kaikille FreeNEST-ohjelman perustoiminnoille ei saatu vielä luotua testisarjoja.

Avainsanat (asiasanat)
automaatiotestaus, Robot Framework, RIDE, FreeNEST, SkyNEST, OpenStack

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
OJALA, Teemu	Bachelor´s / Master's Thesis	02.05.2012
	Pages 61	Language English
	Confidential ( ) Until	Permission for web publication (X)

Title
SMOKE TESTING BENCH FOR FREENEST SERVICE

Degree Programme
Information Technology

Tutor(s)
LEINO, Janne

Assigned by
RINTAMÄKI, Marko

Abstract

The main objective of this study was to create a set of thirty automated tests. These tests verify some of the basic functionality of different components included in FreeNEST.

FreeNEST is a web based project management tool developed in the SkyNEST project, which is hosted inside the JAMK University of Applied Sciences. FreeNEST consist of several different open source based software tools. The set of tests created in this study aims at replacing manual tests that have to be performed repeatedly with automatic tests executable by using a test automation framework called Robot Framework.

The tests are meant to be executed using the OpenStack cloud developed and maintained by SkyNEST project. In the theory part of this study the benefits and restrictions of automated and manual testing are compared and different methods used in testing are presented.

Additionally, the tools and software used in the creation of the tests are introduced in the text and the creation of a single test is explained step by step using the RIDE editor, which is the test case editor of Robot Framework.

The test cases were finished on time and executing them using the virtual machines created in the OpenStack cloud was successful. The total coverage of the created tests however remained slightly incomplete, due to certain restrictions in the current test execution system.

Keywords
test automation, Robot Framework, RIDE, FreeNEST, SkyNEST, OpenStack

Miscellaneous

CONTENTS

FIGURES

TABLES

TERMINOLOGY

FreeNEST	A project management tool developed in the SkyNEST project.
SkyNEST	Name of the project hosted inside JAMK University of Applied Sciences.
Ad-hoc	Software testing that is performed at random without any planning or documentation.
Selenium	Software testing framework used for web applications.

Cloud computing	Cloud computing is a general term used when delivering any locally hosted services over the Internet for external clients.
TestLink	Web based test management program.
Robot Framework	A generic test automation framework, which development is supported by the Nokia Siemens Network.
HTML	Hypertext Markup Language is a language mainly used to create documents and pages for the World Wide Web.
Python	Programming language with a design that emphasizes on code readability.
Jython	Implementation of the Python language written in Java, which is another programming language.
OpenStack	Open source cloud computing platform for creating public and private clouds.
Junkcloud	Unofficial name of the OpenStack cloud developed and maintained by the SkyNEST project.
Firebug	A plug-in for Firefox to get more information about web sites. Used as versatile web development tool.
Git	Open source version control system.
PHP	Programming and scripting language usually used inside HTML pages.
RIDE	Tool for creating test cases for Robot Framework.

Functional testing	Functional tests validate the proper behavior of a certain process in precisely described scenarios.
Non-Functional testing	Used for estimating the overall performance of a system under testing.

1 INTRODUCTION

Testing has always been a major part of software development and that is a fact that likely will not change in the future either. Software testing is process a designed to ensure that a computer code or program does what it is supposed to do and to make sure that the code does not do anything unintended either. Testing is carried out to make software reliable and predictable; however, it can never guarantee the total absence of errors. Creating a clear test plan that includes complete and thought-out test cases will help in developing software by detecting as high amount of errors as possible during the production. (Myers 2004, 1-2, 43)

Selecting the amount of testing that should be performed to the software under development is challenging to estimate. System testing in particular can be performed until the time or money of the project ends. There should always be some acceptance criteria concerning testing and after they are reached, it should end the testing cycle. (Haikala & Märijärvi 2006, 293)

2 BASIS OF THE WORK

2.1 Assigner of thesis

The thesis was assigned by JAMK University of Applied Sciences, more precisely a cloud software project called SkyNEST that is hosted inside the School of Technology at JAMK. SkyNEST is a part of TIVIT Cloud Software program, which aims to improve the competitiveness of Finnish software development.

> *According to the 2009 survey most significant factors of competitiveness are:* ***operational efficiency, user experience, web software, open systems, security engineering and sustainable development****. Cloud software ties these factors together as software increasingly moves to the web. Cloud Software program especially aims to pioneer in building*

> *new cloud business models, lean software enterprise model and open cloud software infrastructure.*(Järvinen 2012.)

2.2 Background for thesis

FreeNEST is developed by using agile software development method and therefore many builds are made frequently. FreeNEST is a project platform containing many different open source applications, and they all have to be tested as well individually as also regarding their performance between each others. Doing hundreds of tests manually after each build would be time consuming and a waste of personnel resources. Giving the option for testers to run series of tests automatically and many times faster than by manual execution, enables the project team to use these previously wasted personnel resources for developing testing methods even further.

2.3 Objectives of the study

The main objective the thesis is to create an automatic testing environment for the FreeNEST web service instance and a compact testing plan to execute smoke testing for its future releases. The smoke test contains thirty relatively quick tests that confirm the functionality of some of the critical functions of FreeNEST. These tests are meant to be run as easily and simply as possible by any tester and all the results from the tests will be reported automatically into a web-based test management program.

3 THEORY

3.1 Test Automation

The reason for running tests on any software in development is to locate and eliminate possible errors so they would not appear in the final product. Even if software development groups are trying their best to test their products, there will

always be defects in the delivered software. Testing is performed either manually or it is automated by using automated software testing tools. By using this automated method, effectiveness, efficiency and coverage of the testing can be improved and therefore the number of defects that end up in the final product is minimized. (Smartbear 2012.)

When the testing is performed manually, a tester simply uses the program normally with a computer. It includes going through different screens of the application, trying out versatile sets of usage and input combinations, and then comparing all the received results to the expected behavior of the software. In addition, the testers have to record their observations based on the output they get from the program in different situations. (Smartbear 2012.)

Automated testing tools can perform tests that have been created in advance. They are able to perform specified actions and compare results of these actions to the expected behavior of the software. After the tests have been run, the automated testing tool can report the success of the test or possible errors to a test engineer. Once created, these automated tests can be repeated easily and can perform actions faster than any human person ever could. (Smartbear 2012.)

3.1.1 Benefits of automated testing

The usage of some form of fast paced agile development method is very common in organizations today. Test automation is becoming more of a necessity rather than an option in the software development, since many upgrades are frequently made to the product under development.

Between the development cycles of software, many tests have to be repeated often to ensure quality. Every time the source code is altered, these tests should be performed on the software. If the software supports different operating systems and hardware configurations, the amount of tests to be run increases even more. Running these tests manually would be costly and time consuming; on the other

hand, automated tests have to be created only once, and after that can be run over and over again whenever needed. (Smartbear 2012.)

Some tests are nearly impossible to be run manually. They might be too long or might require thousands of simultaneous users to test. With test automation, tests can be run simultaneously on different computers and can simulate as many virtual users interacting with the software as needed. Automated testing tools are able to monitor memory contents, data tables, file contents and internal program states quickly and accurately. Using all this gathered info, they are able to determine if the product really works as intended. (Smartbear 2012.)

Some tests might require a tester to repeat same lines of commands hundreds of times. Automated tests can be configured to run the same tests many times continuously, as many times as needed. Automation will also remove the possibility of human errors, since automated tests will follow the configured steps precisely every time and will always record detailed results. It is also exhausting for the employee in the organization to keep repeating the same tests frequently. (Smartbear 2012.)

3.1.2 Restrictions in automation

Even though test automation has very noticeable benefits over manual testing, it is not always a viable option to use it. In certain occasions, it might be too hard or even impossible to create automated tests for the product. Some parts of the software may change considerably and therefore the automated tests have to be rewritten, and if this happens frequently, a great deal of time and effort has to be used constantly to keep them updated. Creating working test automation will always take some time, and if there is not enough of it and the project has a very strict deadline, it is better to stick with manual testing. (Selenium 2012.)

Some tests might require somewhat different user interaction than a machine can provide and therefore are not possible to be run automatically. For example

automated tests might not be able to give feedback on the visual errors in the program. If the tests do not need to be run very often, it is also better to use manual testing. Manual testing allows the tester to perform more random testing (ad-hoc) on the program, which can help discovering errors that have never been considered even to exist.

For the employees in the organization to use these automated testing tools, it will always require some knowledge of them. If there is not enough experience currently available, some time has to be invested in researching these tools. The last thing to consider is to calculate costs and compare them to possible benefits of the automation. Setting up an automated tester is always more expensive than using manual methods in testing, however, in the long term using the automated tester will be more cost efficient than doing the tests manually over and over again. (Outsourcebazaar 2006.)

3.2 Testing Techniques

Finding bugs is not the only reason for software testing, since the system in whole also has to work as intended. Overall quality and the reliability of the developed software has to be measured, and this is achieved by using a few different types of testing techniques and they all have their own purpose in the testing process. These four types of techniques are called Correctness Testing, Performance Testing, Reliability Testing and Security Testing. (Khan 2010.)

Testing is also divided into three different types of testing depending on their target part of the software. Unit testing is used to verify the functions of independent components of the product before they are combined with other components into a larger unit. In integration testing, the functionality between components is tested. Units are tested in groups, and their interaction between each other is monitored. System testing is carried out when the full complexity of the product is assembled. This is usually the most formal part of testing and it is used to ensure that the

product responds correctly in normal situations and handles possible exceptions in a correct way. (Krishna Training 2011.)

3.2.1 Correctness Testing

The minimum requirement of any produced software is that it behaves as it should. Correctness testing is used to make sure that the software works as intended and to discover and remove the incorrect behavior of the system. Correctness testing is divided into three different types of testing, depending on the type of person executing the tests and his/her personal knowledge of the software in question. These three types of testing are white box testing, black box testing and their combination, grey box testing. (Khan 2010.)

White box testing

White box testing requires the tester to have full knowledge of the software and its source code under testing. It consists of accurately driven tests, where input is given to the system and then its effects are inspected inside the system process. After the system gives the required output, the tester has all the information about the steps leading from the input to the output. White box testing is used in all types of testing (integration, unit and system testing). White box testing is the most reliable method to make sure that all the steps in the test are properly executed. (Khan 2010.)

Black box testing

Black box testing is another type of correctness testing; however, it is not limited to only be used in it. In Black box testing, the tester has no knowledge of the test product's internal structure. The test designer simply selects valid and invalid inputs and determines what should be the expected output, when the input is given. This method of testing is also usable on all types of testing and it is much simpler than white box testing, but it does not give as comprehensive results as white box. (OneStopTesting 2012.)

Gray box testing

The combination of white box testing and black box testing is called gray box testing. In this method, the tester has some knowledge of the internal structure of the product, but the testing itself is still done at black box level. To create test cases for gray box testing, the test engineer has access to the internal data structures and algorithms; however, the resulted tests are still not as accurate as in white box testing. (Khan 2010.)

3.2.2 Performance Testing

Performance tests are executed to determine the speed and effectiveness of a system. The performed tests can calculate and monitor the resource usage, throughput and response time of the system under test. Performance testing can be executed on any applications to measure their specifications; however, the most usual targets are web applications, since they require low latency on the website, high throughput and low utilization. Performance testing can be divided into load testing and stress testing. (Khan 2010.)

To ensure that a service can handle a set number of users, load tests should be executed. These tests can be carried out by using virtual users, since they are easy to administrate. Load testing is also used to observe the performance of applications in heavy load situations, thus it is easier for web services to find out how many users the service can hold until the systems start failing or their performance decreases. When the load placed on the system is intentionally raised above its normal amounts of usage, the tests are considered as stress tests. Load and stress tests can be done either manually or automatically; however, manual testing is not nearly as practical as automated testing, since with automated testing, tests can be easily repeated as many times as needed. (Khan 2010.)

3.2.3 Reliability testing

Reliability testing is used to discover failures in the system, so they can be fixed before deployment of the system in question. With the help of reliability testing, an estimation model is created, which is used to estimate the overall reliability of the system in present and in the future. Developers can then use all the gathered information from the tests results and this estimation model to decide if the software is ready for releasing. Stress tests can actually be considered as a variation of reliability tests, where system reliability is tested in extreme conditions. (Khan 2010.)

3.2.4 Security testing

To make a program not accessible for unauthorized users, security testing should be performed to the developed system. Finding all the major weaknesses of the system helps preventing the access of unauthorized users, thus preventing any harm purposely done to the system. It is also important to secure the information of authorized users from outsiders. Security testing will not only help against attacks, but also helps the system to run longer without any major problems. (Khan 2010.)

3.3 Cloud Computing

Cloud computing is a service that consists of a group of computing resources, such as networks, servers, storages, applications and services. These resources are then delivered over the web to the customers and typically these services can be divided into three groups, which are called software as a service (SaaS), platform as a Service (PaaS) and Infrastructure as a Service (IaaS). These three different types of service are illustrated in figure 1. Cloud computing can also be divided by its means of use into four groups, private cloud, community cloud, public cloud and hybrid cloud. (NIST 2012.)

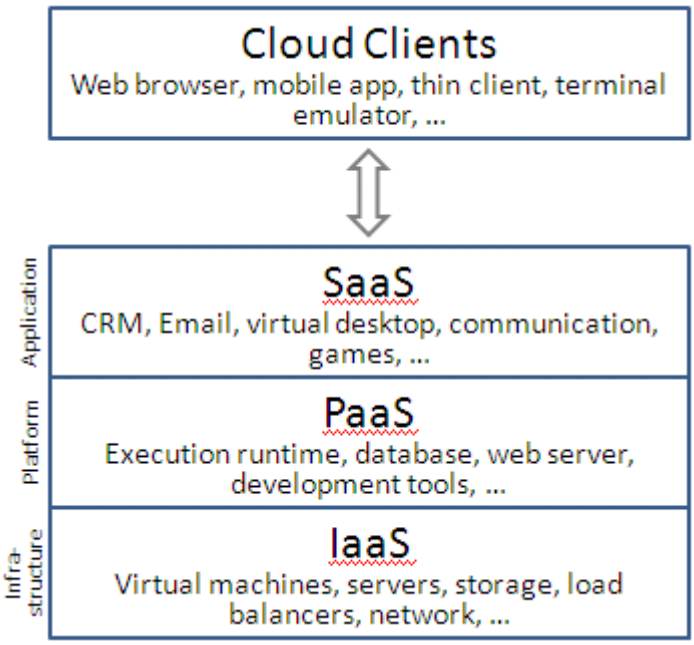


FIGURE 1. The three cloud service models (Wikipedia 2012).

There are many benefits for the clients of cloud compute services, the most noticeable being the scalability of the service. The client organization can add or subtract the amount of resources they need, making the service very flexible. These services are very easy to implement, since the clients do not have to buy their own hardware, software or implementation services, all is handled by the cloud compute service provider. Since all the storaging and server managing is outsourced, a company can lessen the burden on its own IT team, thus the in-house IT department can focus more on the business-critical tasks rather than increasing manpower with costly training. Lastly, the provider's quality of service offers continuous support, and in cases of emergency situations, an instant response to the fault at question. (Waxer 2012.)

Software as a service (SaaS)

Software as a service allows client organizations to access the provider's applications running on a cloud infrastructure. These applications can be anything the client needs and wants to use. This will remove the need for organizations to handle the installation, set-up and maintenance of the required applications by themselves; everything is handled by the service provider. The applications are then accessible

through either a client interface, such as a web browser, or a designated program interface. (Webopedia 2012.)

Platform as a Service (PaaS)

Platform as a service is another service model in cloud computing. It is an environment that provides the possibility to develop, deploy, manage and integrate applications inside the cloud using the efficiency, flexibility and speed offered by the cloud service. SaaS is a service application that is ready to be used, while PaaS is a platform allowing the client to create their own differentiating and unique applications for their exact needs. (IBM 2012.)

Infrastructure as a Service (IaaS)

By requesting Infrastructure as a Service, the client has control over the software environment inside the cloud, but is not required to maintain any of the equipment it is running on. The client does not have to invest in their own hardware to run the software on, but can request virtual machines from the IaaS provider. These virtual machines can then be used to contain any software the client wants and if more storage or other resources are required, the service provider can easily add more of them with the help of virtualization technologies. (Caruso 2011.)

4 TECHNOLOGIES

4.1 General

Many programs used to make testing easier on different aspects are being developed all the time in the web. Some of them are components or complete installments that allow users to perform automated tests for different software that are in development. Other programs may be used to help the testers themselves to plan new tests and keep track on tests that have already been performed for different

builds of the target software and even to create vast test plans for the development project.

All different programs and components used in this study are open source based so there are no costs using any of them as long as the products made with them are not used in any commercial way. TestLink is a web based test management system that is currently included within the default installment of FreeNEST web service platform. The author was already familiar with TestLink as he had used it previously; therefore it was an easy choice to select it as the management part for this assembly. Robot Framework is a test automation framework used to execute the desired tests. Selenium RC is the component that is required by Robot Framework to understand and command web-based applications like FreeNEST therefore making automated tests possible with the assembly used in this work.

4.2 TestLink

TestLink is a free to use web based test management program that helps its users to create and keep track of tests that have to be performed on the target software currently being developed. Creating a new project is the first step when using TestLink. Test Plans are then created along with many singular Test Cases that can later be assigned into these Test Plans. The same Test Case can be assigned into any number of Test Plans and one project can consist of many different Test Plans. This kind of hierarchical system makes it so much easier for the users to manage different areas of testing in a project. Figure 2 displays the front page of TestLink. (TestLink 2012.)

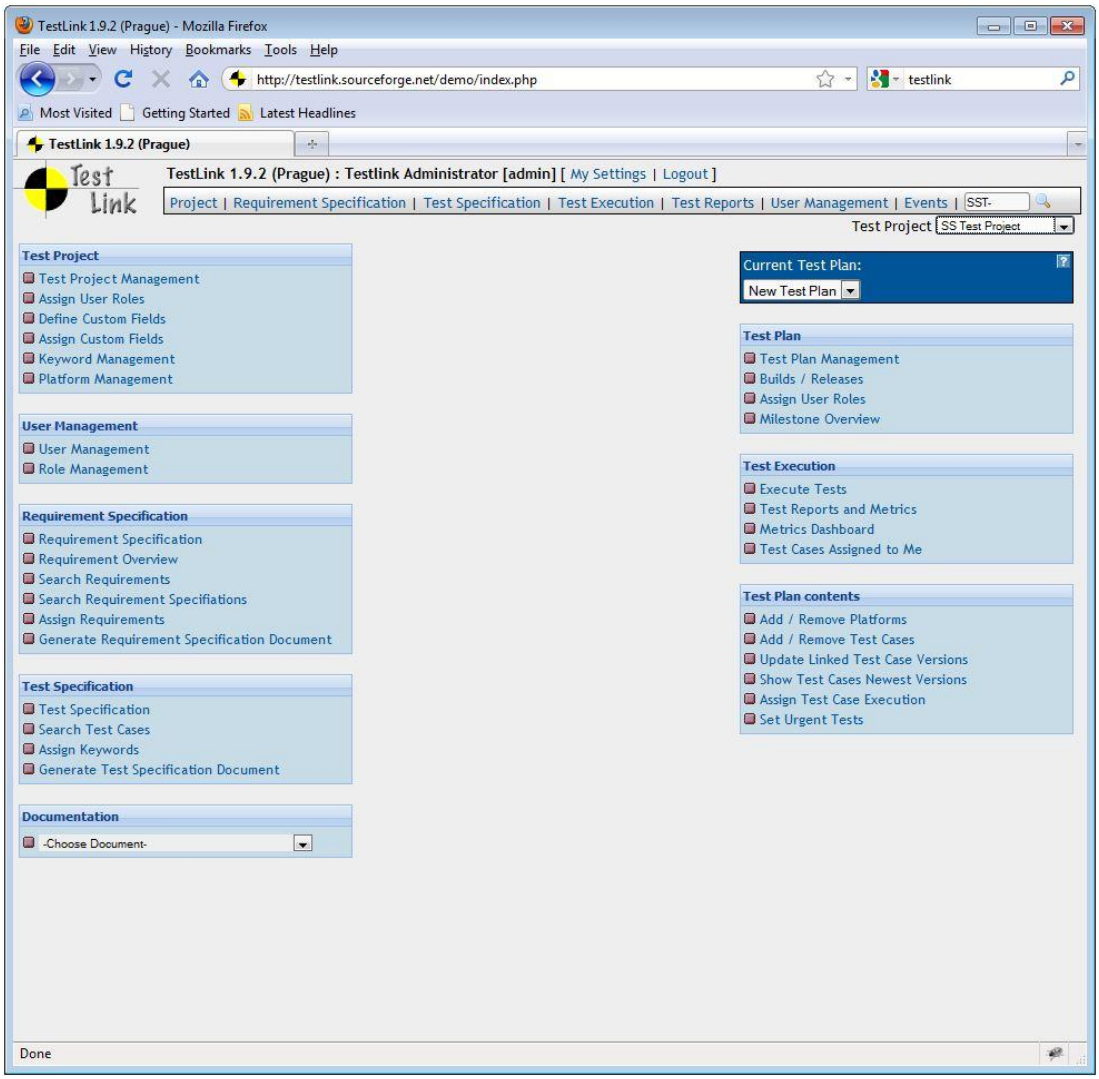


FIGURE 2. A Screenshot of TestLink demo.

TestLink Projects

When creating a new project with TestLink, the user is asked to name the new project and specify a unique ID for it. This ID is automatically added to the names of any Test Cases created within the project in question. The user can also write a description for the project that can explain different aspects in slight detail, such as the aim of the project or its target group. There are also few settings that can be turned on for this project. The most important one considering this thesis is to enable the usage of API keys. Without them the project will not allow users to execute any automated tests on the project. (TestLink 2012.)

Test Cases

After a new project has been created, users can immediately start creating Test Cases for the project. Test Cases are put inside Test Suites much like files are put inside folders in a computer, therefore it is recommended to create a number of Test Suites to help keeping the possibly hundreds of Test Cases inside the project clearly sorted. One good example is to sort Test Cases by their method of execution, either manual or automated. (TestLink 2012.)

The first thing to do when creating a Test Case is to give it a title, which can for example be a short description of the case. There is also a description box to write more detailed information about the test. All the steps can be written on a textbox reserved for them and there is also space to write down expected results of the test. If the settings of the project allow it, test importance and the execution type can also be selected. Execution type can be either manual or automated. Keywords can be assigned to the Test Case to help categorizing or filtering different Test Cases in some other areas of TestLink. (TestLink 2012.)

Test Plans

Before any tests can be executed, at least one Test Plan has to be created for the project. Additionally the Test Plan itself requires at least one build. A build is a specific release or a version of software. Each project is usually made up of many different builds and in TestLink test execution is based on Builds and Test Cases. If the project has no builds created for it, test execution is disabled. Additionally, test Cases have to be assigned into Test Plans to enable executing the tests. (TestLink 2012.)

Test Execution

As mentioned above, two methods of execution are available for the user, which are manual and automated. When executing the tests manually, the user will simply read the description of the written test and follow the steps as accurately as possible.

After all the steps have been performed, the status of the Test Case can be changed by comparing the perceived results by the tester to the expected results written on the Test Case itself. The new given status can be failed, passed or blocked.
When TestLink is configured correctly to allow automated test execution, a button that launches the test run can be found in the Test Case. TestLink itself cannot perform any automated testing; however, it can send a request into an external testing framework to perform the test and then report the results back to TestLink. (TestLink 2012.)

4.3 Robot Framework

Robot Framework is described as a generic test automation framework. It is used for acceptance testing and acceptance test-driven development (ATDD). It uses simple keywords that act like commands. By implementing new test libraries, many more keywords become available to use which allows more complex tests and makes it possible to run tests on many different kinds of target software. (Robot Framework 2012.)

Tests for the Robot Framework can be made in four different file formats. They can be written using the hypertext markup language (HTML), tab-separated values (TSV), plain text or reStructuredText (rest) file format. All of these are somewhat similar to each other, but have some minor differences and benefits over each other. (Robot Framework 2012.)

HTML Format

Recognized extensions of files for test cases created in HTML format are .html, .htm and .xhtml. In these files, all the test data is written inside many separate text tables as shown in figure 3. The first cell in each test data table is used by Robot Framework to identify different types of tables inside the test file. If it is not one of the recognized tables, it will be ignored completely. The tables can be created using any kind of editor; however, using a graphical editor is more favorable as you can see the tables while you edit them. (Robot Framework 2012.)

Setting	Value	Value	Value
Library	OperatingSystem		

Variable	Value	Value	Value
${MESSAGE}	Hello, world!		

Test Case	Action	Argument	Argument
My Test	[Documentation]	Example test	
	Log	${MESSAGE}	
	My Keyword	/tmp	
Another Test	Should Be Equal	${MESSAGE}	Hello, world!

Keyword	Action	Argument	Argument
My Keyword	[Arguments]	${path}	
	Directory Should Exist	${path}	

FIGURE 3. Test case in HTML-format (Robot Framework 2012).

TSV Format

TSV files are edited very much like files in HTML format. The main difference is that all the test data is written in one uniform table unlike in the HTML files, where the different test data tables are separated by completely different tables. The test data tables in TSV files are separated by inserting one or more asterisks in the cells of the table and any text that has been written before the first asterisks will be completely ignored. Figure 4 illustrates this method of dividing the tables inside a TSV file. These TSV files can be created and edited with different spreadsheet programs, the most common one being Microsoft Excel. (Robot Framework 2012.)

Setting	*Value*	*Value*	*Value*
Library	OperatingSystem		
Variable	*Value*	*Value*	*Value*
${MESSAGE}	Hello, world!		
Test Case	*Action*	*Argument*	*Argument*
My Test	[Documentation]	Example test	
	Log	${MESSAGE}	
	My Keyword	/tmp	
Another Test	Should Be Equal	${MESSAGE}	Hello, world!
Keyword	*Action*	*Argument*	*Argument*
My Keyword	[Arguments]	${path}	
	Directory Should Exist	${path}	

FIGURE 4. Test case in TSV-format (Robot Framework 2012).

Plain Text Format

Plain Text Format uses simple .txt files to create tests for Robot Framework which is the biggest benefit when comparing this format to the ones mentioned before since the text editors are very easy to use. When doing tests on plain test format, the separator between cells is either two or more empty spaces or a pipe character that is surrounded with spaces. Both of these different ways of cell separation are displayed in figure 5. Just like in TSV format, asterisks are used to recognize the test data tables in the text file and all text that is written before the first asterisks will be ignored. (Robot Framework 2012.)

```
*** Settings ***
Library     OperatingSystem

*** Variables ***
${MESSAGE}  Hello, world!

*** Test Cases ***
My Test  [Documentation]  Example test
    Log          ${MESSAGE}
    My Keyword  /tmp

Another Test
    Should Be Equal  ${MESSAGE}  Hello, world!

*** Keywords ***
My Keyword  [Arguments]  ${path}
    Directory Should Exist  ${path}
```

```
| *Setting*  |     *Value*     |
| Library    | OperatingSystem |

| *Variable* |     *Value*     |
| ${MESSAGE} | Hello, world!   |

| *Test Case*  | *Action*         | *Argument*   |
| My Test      | [Documentation]  | Example test |
|              | Log              | ${MESSAGE}   |
|              | My Keyword       | /tmp         |
| Another Test | Should Be Equal  | ${MESSAGE}   | Hello, world!

| *Keyword*  |
| My Keyword | [Arguments] | ${path}
|            | Directory Should Exist | ${path}
```

FIGURE 5. Test case in plain text; space and pipe formats (Robot Framework 2012).

ReStructuredText Format

ReStructuredText (reST) also uses plain text as its method of creating test data tables. This format is commonly used for documentation of Python projects, and is very similar to the HTML format, since the data is defined in tables in the same way as in HTML. All the test data and these tables are still edited in a sententious text format and can be edited using any text editor. The test data tables are identified based on the text in the first cell, at the same time ignoring all the data that is outside these recognized tables. An example of a test case in reST format can be seen in figure 6. (Robot Framework 2012.)

```
============  ================  =======  =======
  Setting          Value          Value    Value
============  ================  =======  =======
Library       OperatingSystem
============  ================  =======  =======


============  ================  =======  =======
  Variable         Value          Value    Value
============  ================  =======  =======
${MESSAGE}    Hello, world!
============  ================  =======  =======


============  ==================  ============  =============
 Test Case          Action           Argument      Argument
============  ==================  ============  =============
My Test       [Documentation]     Example test
\             Log                 ${MESSAGE}
\             My Keyword          /tmp
\
Another Test  Should Be Equal     ${MESSAGE}    Hello, world!
============  ==================  ============  =============


============  ====================  ============  ==========
 Keyword            Action             Argument     Argument
============  ====================  ============  ==========
My Keyword    [Arguments]           ${path}
\             Directory Should Exist  ${path}
============  ====================  ============  ==========
```

FIGURE 6. Test case in reST-format (Robot Framework 2012).

4.4 RIDE

Robot Framework IDE (RIDE) can be used to create and alternate automated tests in a more graphical and distinct environment. It uses files that are in HTML, TSV or Plain Text format. In RIDE, keywords can be given to the test case inside a grid regardless of the file type used. All the usable keywords from the available libraries are listed inside the search function of RIDE. If more libraries are added later, all the keywords from those newly added libraries are also found within this search function. RIDE can also be used to execute the created tests, and overall makes easier to manage test cases for Robot Framework. (RIDE 2012.)

4.5 Selenium

Selenium is a set of tools that can be used to automate browsers. These tools are primarily intended for testing web applications via automation, although their usage is not limited in anyway and can be used in any other tasks as well. Selenium can be used individually, but it can also be controlled by other testing frameworks. It is supported by multiple browser platforms and all its operations are very flexible, since many options are given in locating an element in the user interface of the system under test. There is a number of different ways to compare test results with the expected results of any driven tests, thus accuracy of these tests is not a problem either. (SeleniumHQ 2012.)

Selenium RC and WebDriver

There are four different software tools in Selenium. This suite of tools allows testers to perform automated tests to almost any web application with a vast amount of testing functions. Selenium Remote Control (Selenium RC) was the first main project of Selenium and it is still widely supported; however, mainly it has been replaced with Selenium WebDriver (Selenium 2). Both of these tools are meant for controlling browsers to perform the tests on web applications. Even if falling into the shadow of Selenium WebDriver, Selenium RC is still being widely used, having support for many different languages and internet browsers that the Selenium WebDriver is not compatible with yet. (Selenium 2012.)

Selenium IDE

Selenium Integrated Development Environment (Selenium IDE) is a Firefox plugin, which can be used to record user actions on a browser. All the actions are automatically converted into test script and can, and should be, manually altered by the user, since the recorder is not capable of adding any conditional statements on the test script. Selenium IDE is mainly intended just to create a prototype of a test

case and it is recommended to use it with either Selenium RC or Selenium WebDriver. (Selenium 2012.)

Selenium Grid

The last tool in the set is called Selenium-Grid. When tests are normally executed with the standard Selenium tools, controlling more than six simultaneous browsers with the Selenium is not recommended, as the Remote Control becomes too unstable. It is possible to target the tests to run on different Remote Controls at the same time; this however makes it hard to run specific tests in parallelized fashion. A set of Remote Controls created in this way is also very hard to manage and alter later. Figure 7 below illustrates the operation of a single Selenium Remote Control. (Selenium Grid 2012.)

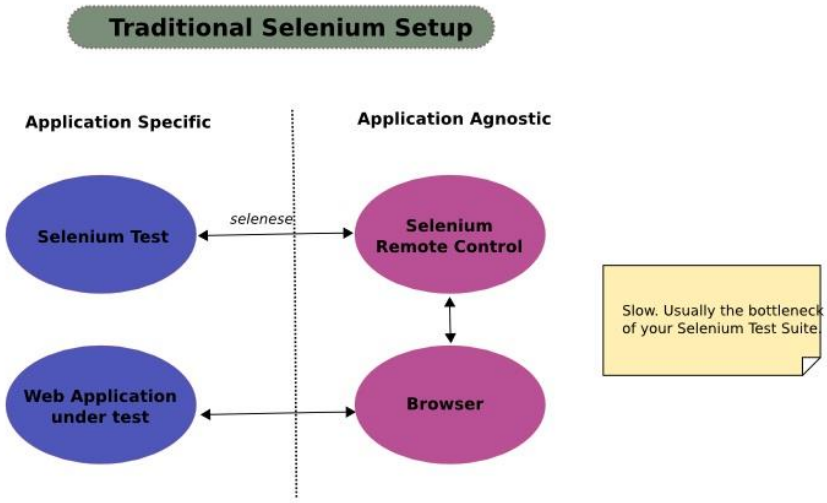


FIGURE 7. Normal Selenium Setup (Selenium Grid 2012).

Selenium Grid uses a new component that will help to manage a number Remote Controls at once. The component is called Selenium Hub (illustrated in figure 8). It can allocate one Selenium Control to execute a certain test, but does not bind the Selenium Control and the test to each other, which means next time the test is run again, it can be done by some other Remote Control selected by the Selenium Hub. This way the Hub can divide the tests that are wanted to be run simultaneously equally between the Remote Controls that are available to use and more Remote

Controls can be assigned for the Hub whenever needed, which makes it largely scalable. Selenium Hub can also run several Selenium Controls on one host machine. (Selenium Grid 2012.)

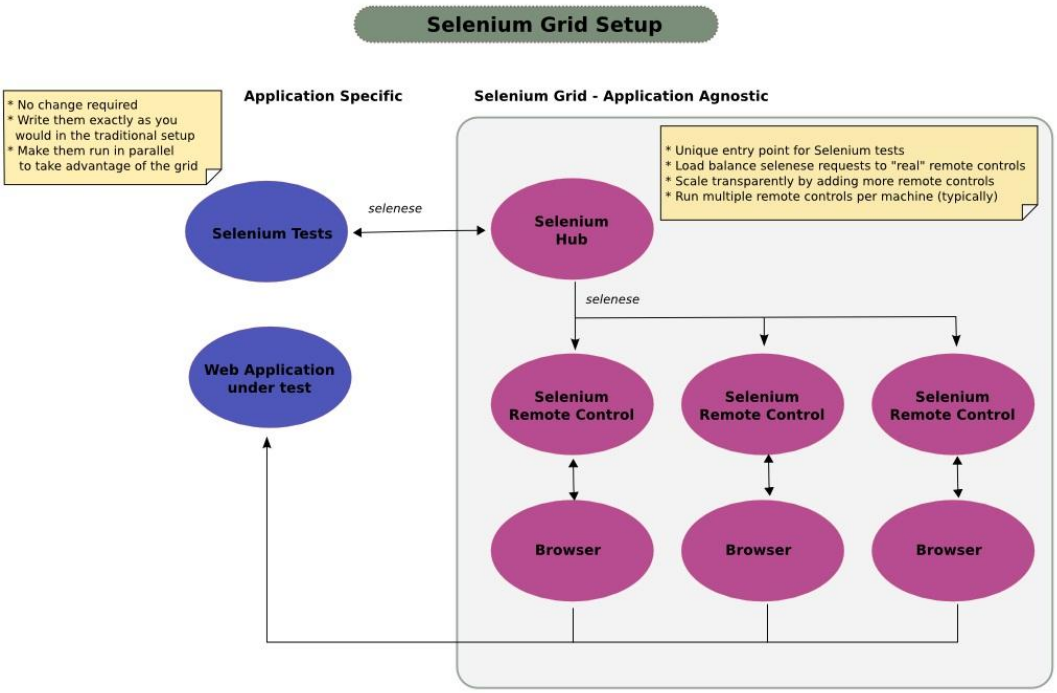


FIGURE 8. Selenium Grid Setup (Selenium Grid 2012).

4.6 OpenStack

OpenStack is an open source cloud computing platform for creating public and private clouds. It gives the possibility for any organization to create and offer their own cloud computing services since OpenStack can be implemented on any standard hardware. The three major components of the OpenStack project are Compute, Object Storage and Imaging Service. Compute is the controller used to start up virtual instances and configure the networking. Object Storage is used to store objects in a system, which is enormously scalable and large by its capacity. The Imaging service is a system used to lookup and retrieve the virtual machine images inside the OpenStack cloud. (OpenStack 2012.)

There are also two new noticeable components, and the first one is called Identity, that provides coherent authentication through all OpenStack projects and is fully

capable to be integrated with all existing authentication systems. The second new component is Dashboard, which allows the users of OpenStack to access and provision resources in the cloud through a self-service portal. (OpenStack 2012.)

4.7 Firebug

Firebug is a plug-in application for the Mozilla Firefox web browser used as a web development tool. It can also be used on different web browsers; however, it will have limited functionality on any other browser than Firefox. It has a wide range of different features that will benefit any web designer or developer and it is designed to reduce the amount of guesswork in web page debugging and can also be used to inspect accurate information about the web page, such as page layout and source code. (McBlain 2011.)

5 TEST EXECUTION

5.1 Automated tester

The SkyNEST project was in a need of test automation framework to increase the testing efficiency done to the FreeNEST project platform. Previously all the testing done after each released build of FreeNEST in development had to be tested manually, because the project did not yet have enough time or vacant resources to be used on creating an automated test bench that could be used in the software testing. The project was moving on fast and the product was tested manually constantly.

After many members of the project had to leave and the development of the FreeNEST was decelerated considerably, a good opportunity to plan and design more efficient and accurate testing method was granted for the project team. When the development of the FreeNEST starts speeding up again, this automated test bench

will allow the future testers to have more time to spend on improving the testing methods and the test cases, when less time has to be used on manual testing.

SkyNEST project is also developing and maintaining their own OpenStack cloud inside the project. By setting up the automated tester inside this cloud, many benefits are gained for the process of test execution, such as easier and faster management of virtual instances that are needed in the testing. It will also grant higher resources to perform the tests and more accurate monitoring of the performance of the instances when they are located inside a cloud.

Junkcloud

Junkcloud is the unofficial name of the project's OpenStack cloud. It is still a relatively new part of the project; however, it has attracted a great deal of interest from many different people outside the project. Currently Junkcloud works as an IaaS platform (see chapter 3.3) for the test execution system developed in the FreeNEST project. To enable test execution in the cloud, three instances, or virtual machines, were created inside it. These three machines are called Team Server, Master Tester and Deep Forest. The first two are the ones that actually host the tools used in the automated testing. Deep Forest only hosts the target applications of the tests and it is not always the same instance, as many ”Deep Forest” instances can be created as needed.

Team Server is the machine hosting the FreeNEST project platform, which in this case is used for test management and execution. TestLink is one of the applications installed by default on the FreeNEST, and is also the selected test management program used in the FreeNEST project to create test cases and execute them to perform software testing for the development of FreeNEST. TestLink has the option to execute tests automatically; however, it requires an external automation framework to do it.

Master tester has all the required tools to execute tests requested by the Team Server. Robot Framework was selected to function as the automated tester, but in order to perform automated tests on web based applications such as FreeNEST, it needs the help of Selenium server. Robot Framework also needs commands written inside test files that it can understand, and these files are also located inside the Master Tester. The test cases created in this thesis will be added as part of those files that the Robot Framework will use.

5.2 Operating the tester

When a test has to be run with this set-up, the tester will use the TestLink located inside FreeNEST in the Team Server. After the desired test, that is configured correctly to run automatically, is selected, simply a button has to be pressed and the process starts. TestLink sends information of the test that has to be run into a PHP script located in the Master Tester. This PHP script then sends the information forward into another script that is written in Python. The PHP script is working as a translator for TestLink, since TestLink doesn't understand Python at all. The Python script is the one sending the request for Robot Framework to perform the test.

With the information the Robot Framework receives, it is able to select a right test case file from all the files that are created in advance and stored inside the Master Tester. These test files contain all the commands the Robot Framework then sends for the Selenium server that is also running inside the Master Tester. Selenium performs the commands it receives from Robot Framework to the target application hosted in the Deep Forest machine. After the test has been run, or if any step of it has failed, Robot Framework creates a log file, and sends the result of the test back to the scripts that forwards them to the TestLink. The result of the test is then automatically updated to the TestLink database about the test in question. The flow of the process is shown in figure 9; however, the GIT Version control Repository shown in the figure is not implemented yet.

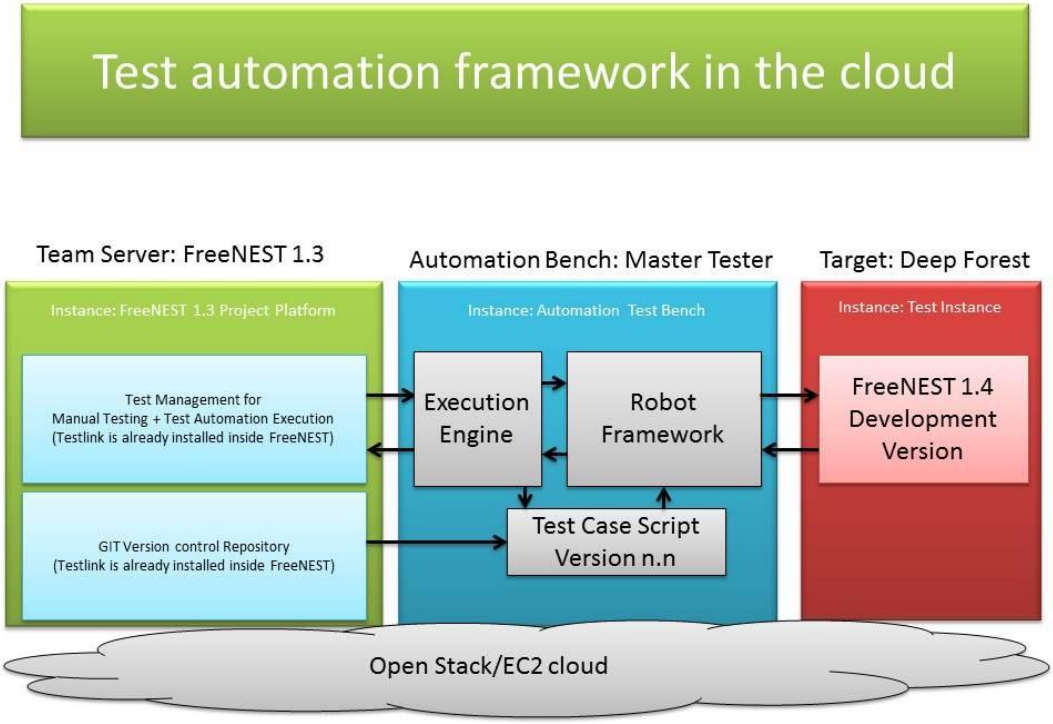


FIGURE 9. The test automation framework created inside the Junkcloud.

There are still some current problems concerning this automated tester. Since the Junkcloud is a very low-budget cloud, it is not as powerful as desired. There are also some parts inside the cloud that work as a bottleneck and slow down the testing process considerably. This set-up is very demanding to maintain and configure compared to the tests run manually. Still, since this is only the first version of working test automation in this project and is still heavily under development, all of the problems should be fixed in the future if wanted and many more features and improvements are most likely added to it.

6 CREATION OF THE SMOKE TEST PLAN

6.1 Planning

The main object of this thesis was to select and write a set of thirty tests that verify some basic functionality of the FreeNEST service that can then be executed automatically with the test automation framework built inside junkcloud. The amount of tests to create was specified when this study was assigned and it was also decided that the tests should perform functional testing on the system and leave the non-functional testing outside the assignment. The author had previously been part of the development of the FreeNEST project as a tester so he already had some experience about the tests used to verify the functionality of some of its components. When deciding which tests are suitable to be run with the current test executioner, many aspects had to be considered to see what tests currently fits into the smoke test plan.

The first thing to consider is to make sure the test is capable of running automatically and getting correct results. Creating a test that inspects visual side of the user interface is not possible with the tools used in the test executioner. Some tests might have needed deeper knowledge about programming languages that the author did not have, thus making them impossible for him to create. The length and contents of each test had to be precisely thought out as well. Creating too long tests that check the functionality of too many components at one run had to be avoided, because the test execution is interrupted immediately when an error has occurred. If the test is used to test multiple components and there is error on the first one, none of the other components will be tested.

On the other hand, tests should not be too short either. For example, when testing the user's accessibility into administration tools, the test should be run with both admin user account and normal user account, since one of those two is allowed to access those tools and the other one should not be. Of course this test could be

divided into two separate tests, which however would lower the efficiency of the driven test to half.

One last thing had to be considered while making these tests was the restrictions of the current test automation framework. None of the tests are allowed to make permanent changes to the system under test and should return the target application to the original state that it was in before running the test if possible. The reason for this is that the current set-up is not yet efficient enough to replace the target instance with a fresh one instantly whenever needed.

6.2 Creation of a Robot Framework Test Case

This chapter describes the creation of one of the thirty test cases that were added to the smoke test plan for FreeNEST. This example only describes a small fraction of all the possible keywords that can be used with Robot Framework. More example test cases can be seen in the appendices of this thesis. Libraries that the author had in use were the Robot Frameworks own library called BuiltIn which has basic commands that can be used in any tests and SeleniumLibrary that enables keywords which can be used by the test to give commands for web browsers to make certain actions.

The minimum requirement for creating test cases for Robot Framework is any kind of text editor, but working without other tools is not recommended. The author used RIDE editor to create the test case files, and saved them in .txt format. Additionally, the help of Firefox plug-in called firebug was used, which makes it much easier to find certain identifiers from the source code of the web application. To verify that the commands are correctly written in the test file, Robot Framework was used to execute the tests after every set of several commands. To use the keywords from SeleniumLibrary, Selenium Server has to be running or the test will not work. Robot Framework uses Python, Jython and IronPython programming languages and requires atleast one of them installed on the operating system. The one selected to

execute the tests here is Python. All these tools were used on Windows 7 operating system.

To start creating the test file, RIDE has to be launched first. This is done by writing ride.py into the Windows command console. The instructions are given below as follows. The user is instructed to create a new project and it can be simply selected from the dropdown list under “file” or by pressing key combination ctrl+N. This will cause a new window, presented in figure 10, to open. User is then asked to give a name for the project. Since the author is making a project that will just have one test in it, it is named after the test. One project can contain multiple test cases, and they can all be run one after another with a click of a button, so if for example you wanted to create a series of test cases that will test the login instead of just one, you could give it a name such as “Login Tests”.

This test will verify the login to Webmin, an administrative application that is one of the applications installed with FreeNEST package. You can also select the location where to save the test file and the format it will be created with. The user decided to use the plain text format.

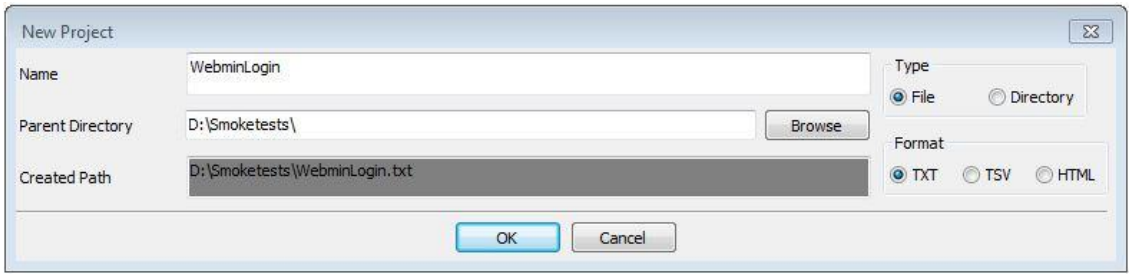


FIGURE 10. Creating a new project with RIDE.

After the new project is created, SeleniumLibrary has to be added to the used Libraries list by using the “Add Library” button located on the right side of the user interface while the newly created project is selected. The library has to be named exactly as SeleniumLibrary for it to be recognized correctly inside the test file. There are also options to insert arguments, an alias and comments for the Library; however, they are not unnecessary for this case, so they are left blank. These options can be seen in figure 11.

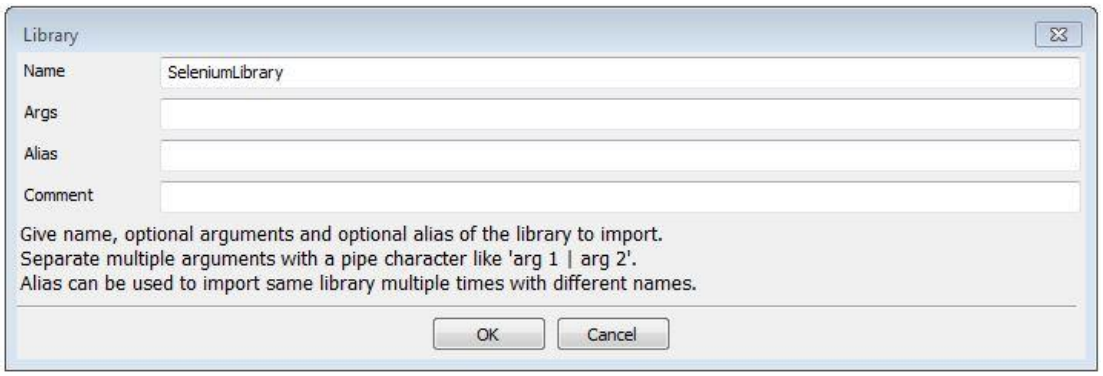


FIGURE 11. Adding a library to a test project.

To add a test case for the project, simply right clicking the WebminLogin project found at the left side of the user interface, where all the components of the project are listed, opens up a list with an option to create one or by using key combination ctrl+shift+N. The program will ask a name for the new test case (illustrated in figure 12) and again it will be WebminLogin.

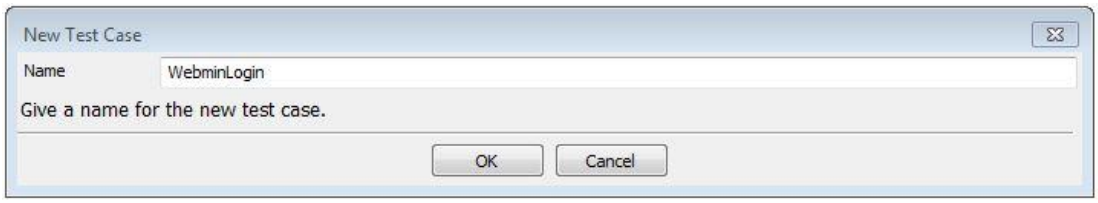


FIGURE 12. Creating a test case for a test project in RIDE.

Keywords to the grid found inside the WebminLogin test case can now be added. There are also some extra settings on top of the grid as shown in figure 13, but filling them is not required so they are left empty for now. By clicking the “settings” button, they can be minimized.

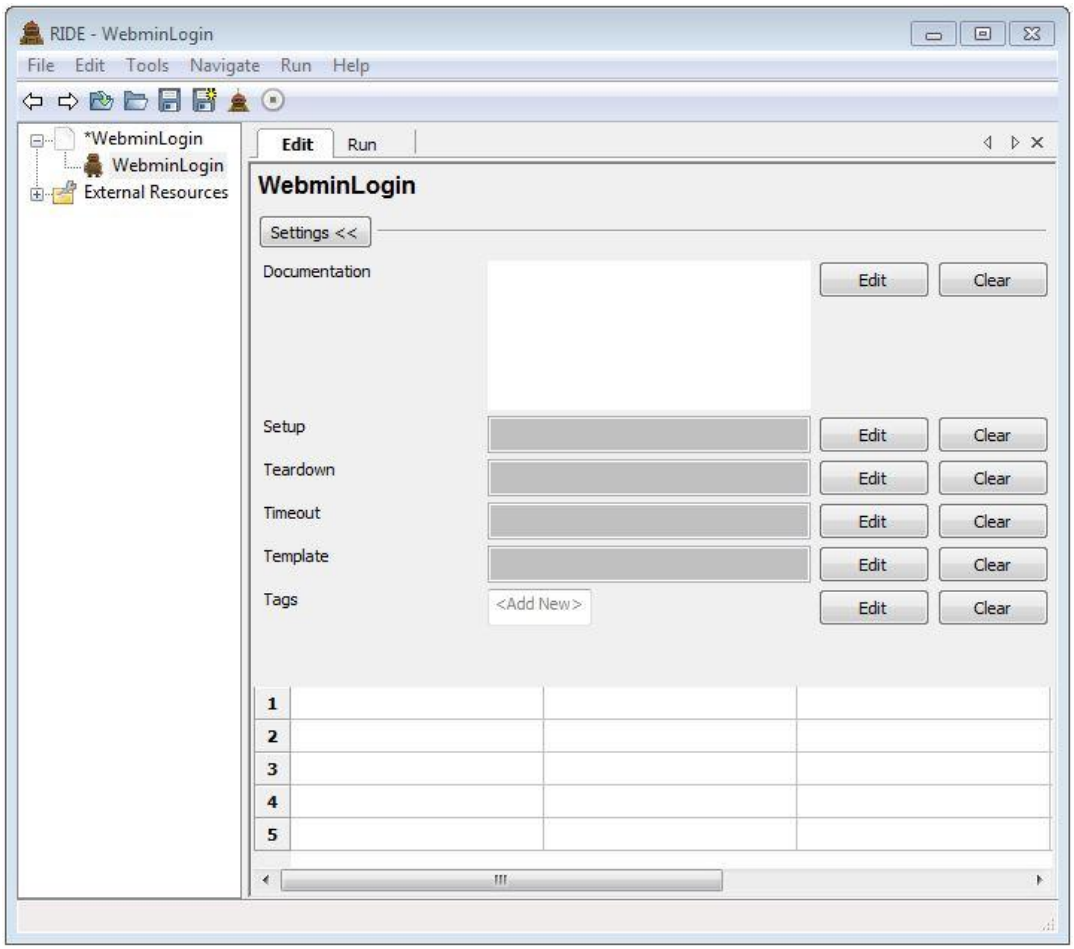


FIGURE 13. Empty test case in RIDE.

Keywords are added to the leftmost cells of the rows. Usually only one keyword is in one row, but there are some exceptions, such as when using keywords that add certain conditions to another keyword. For example a keyword “Run Keyword and Expect Error” can be inserted in front of a normal keyword such as “Click Link”. The list of all the keywords and their explanation can be found in the internet sites of the Libraries or by opening the Search Keywords window (visualized in figure 14) from the dropdown lists located on the top of RIDE.

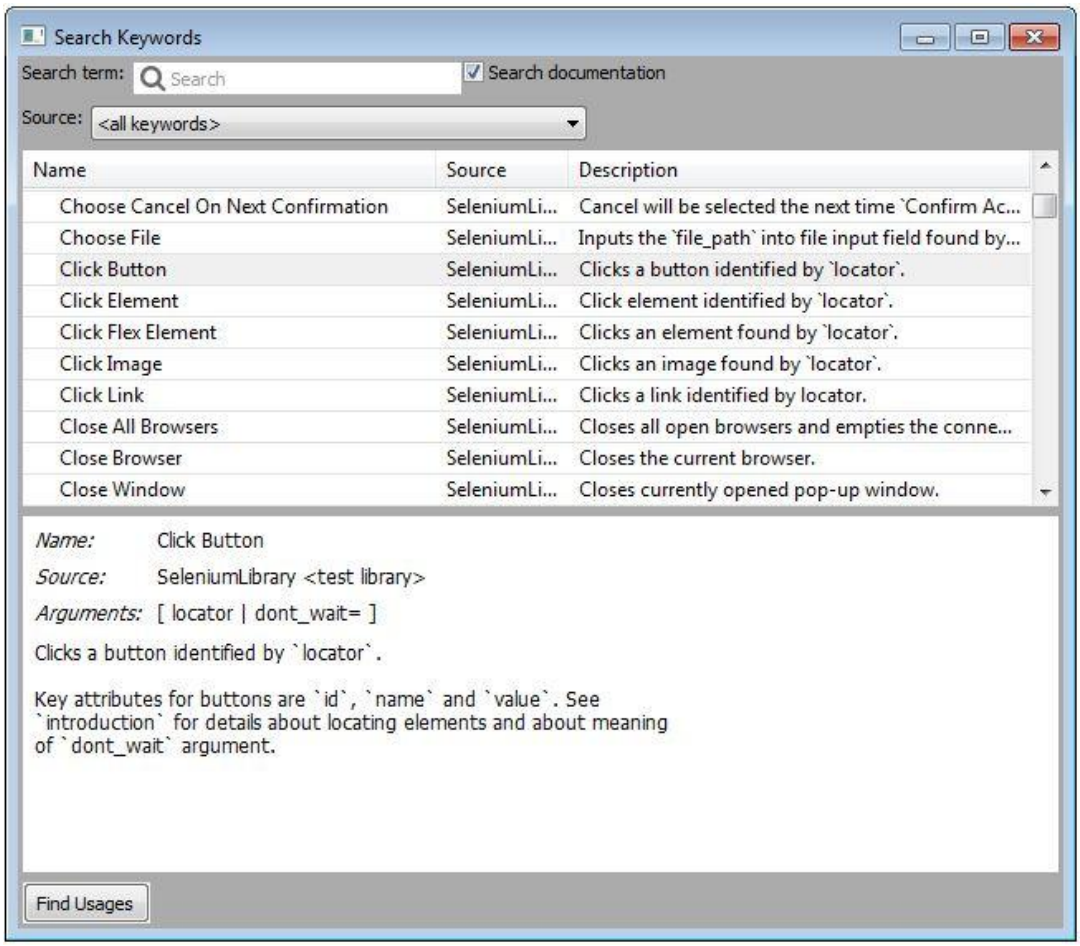


FIGURE 14. Search Keywords window in RIDE.

The first command to perform is to open up an internet browser. This is done by using the keyword “Open Browser”, so it has to be inserted into the first row of the grid. After the keyword is written the cell next to it will turn in color red (see FIGURE 15). This means that the cell requires some information inside it to make the keyword work. By moving the mouse cursor over the red cell, it can be seen that the cell is missing an argument. In this case URL (Uniform Recource Locator) has to be given that is wanted the browser to open into.

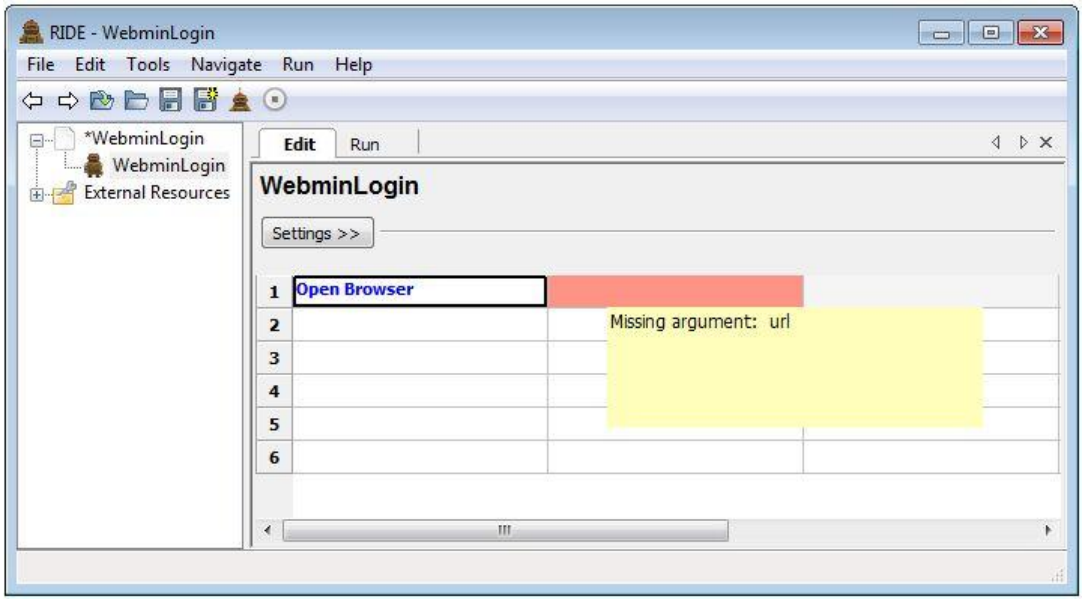


FIGURE 15. Missing an argument in keyword.

This test was meant to verify the login in Webmin, but first the browser has to be navigated there through FreeNEST. FreeNEST is installed with default settings so its address will be the URL used in this cell. It can also be specified in the cell next to the red cell, which internet browser we want the test to be run in. By writing “ff” in it, Robot Framework will understand to execute the test in Firefox browser. The missing arguments have been inserted for the test case in figure 16.

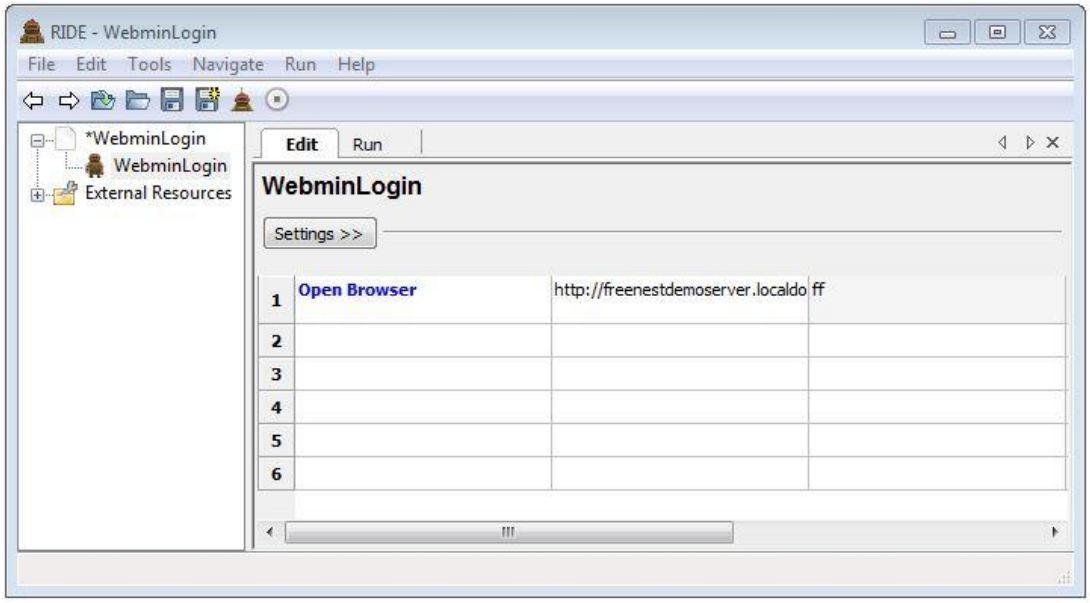


FIGURE 16. Filling the required arguments to a keyword.

If run the test were now run, a browser would open into the FreeNEST login screen. Next the test should insert the login credentials of the FreeNEST admin into this login screen. For that the keywords that are called “Input Text” and “Input Password” are used here. These keywords require an argument called locator inserted to the cell to the right of them. First, however, the IDs of the text boxes in the login screen have to be known and this can be done by inspecting the source code of the page. One way is to just right click anywhere on the page and from the list that opens, select the “View Page Source”. Using this method is not recommended, since the source codes of some sites are very complicated and finding certain information from them can be difficult this way. By installing a plug-in for the Firefox called firebug and activating it, another option appears in the list when right clicking the page. This option is “Inspect Element” and should be found at the bottom of the list, which is shown in figure 17.

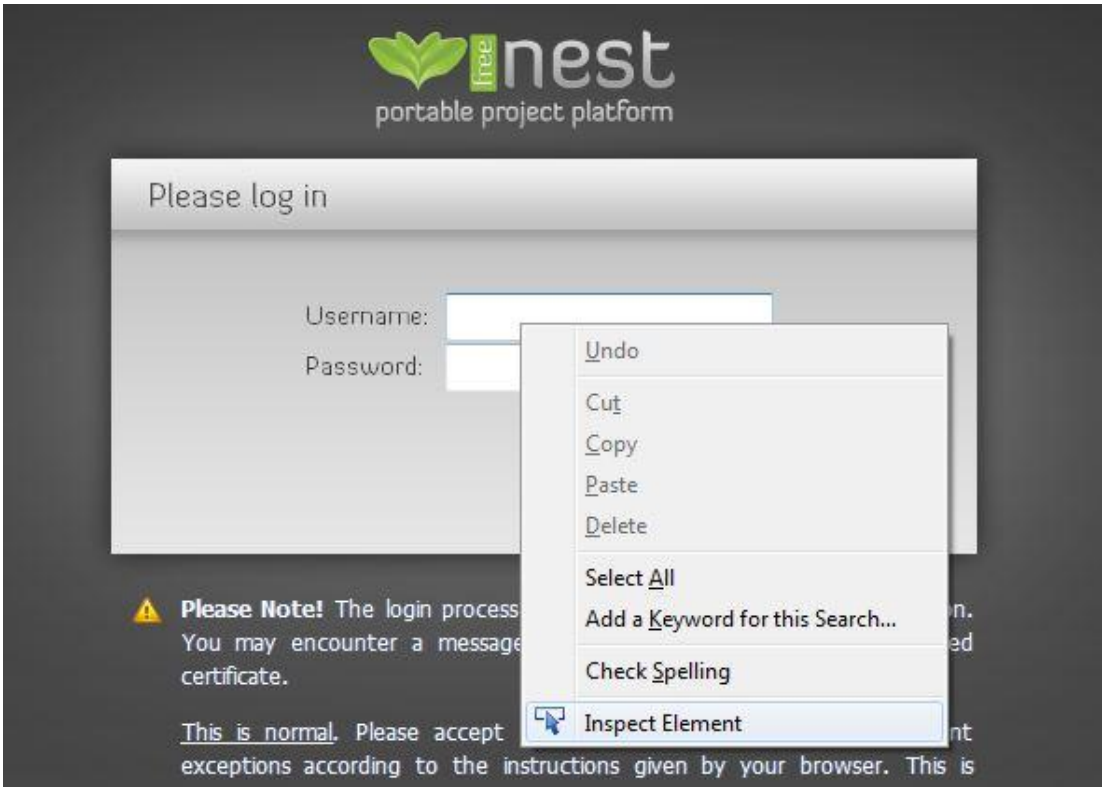


FIGURE 17. FreeNEST login screen.

By directly right clicking on the textbox wanted to be filled and selecting the “Inspect Element” option, the firebug will automatically show information from the source

code related to the textbox as shown in figure 18. A text field for the firebug should be found on the bottom of the internet browser.

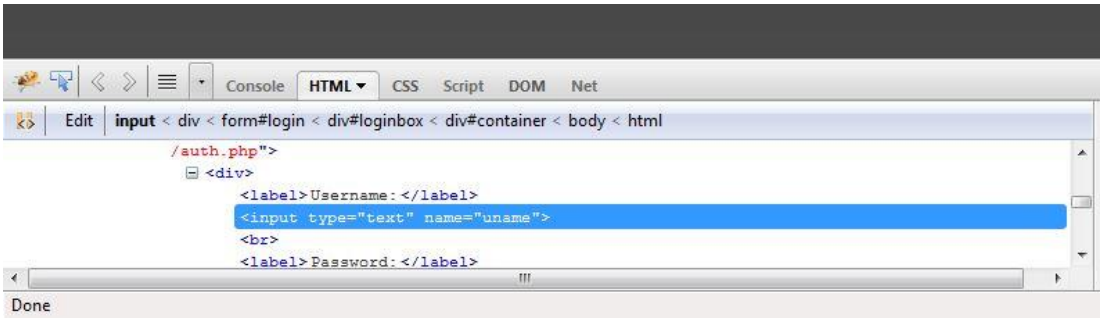


FIGURE 18. Firebug showing source code of a textbox.

From the source code can be seen that the name of the username textbox is “uname”. Since a password needs to be inserted also, the name of that textbox is also required and by inspecting the source code again, can be seen that the name of it is “password”. Now that the names of those two textboxes are known, the “Input Text” and “Input Password” can be instructed to fill them with login credentials. Default username and password for the admin in FreeNEST is AdminUser, so they can be added to the next cell from the locator cell. In figure 19, the required user information has been added to the grid.

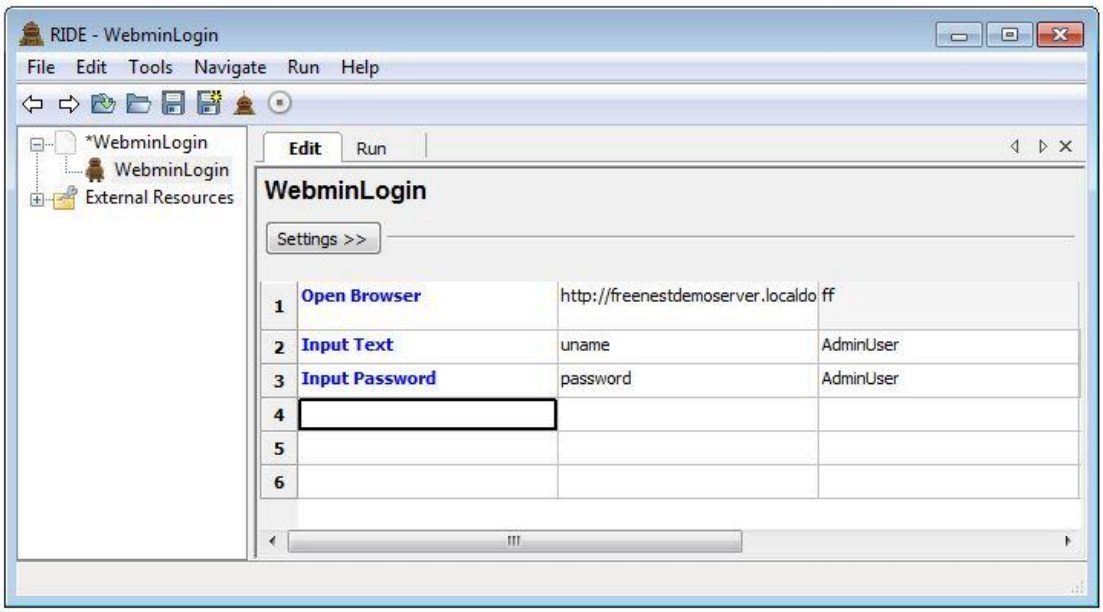


FIGURE 19. Inserting login credentials for test case.

In order to get the user logged in, login button has to be pressed. The “Click Button” keyword can be used to do it, but again the ID of the button is needed. Right clicking on that button and selecting the inspect element again will show the part in the source code needed to be inspected. Key attributes for buttons are id, name and value. Since the value is the only one given here (see figure 20), it can be used with the “Click Button” keyword to carry out the login. Figure 21 demonstrates the usage of the keyword in this test case.

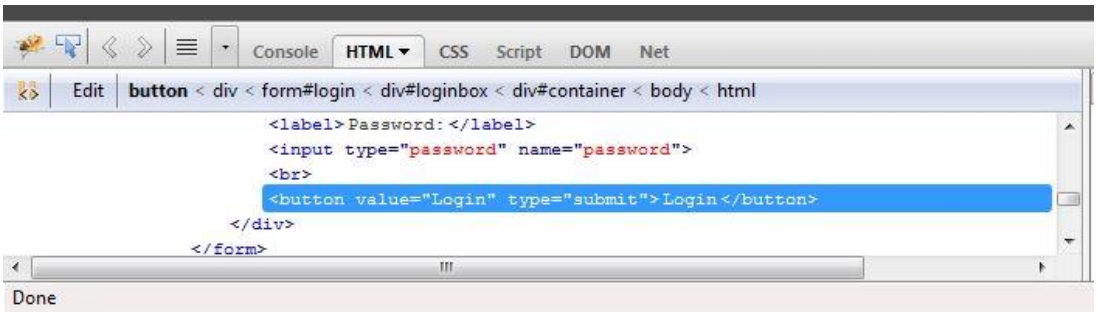


FIGURE 20. Source code of a log in button.

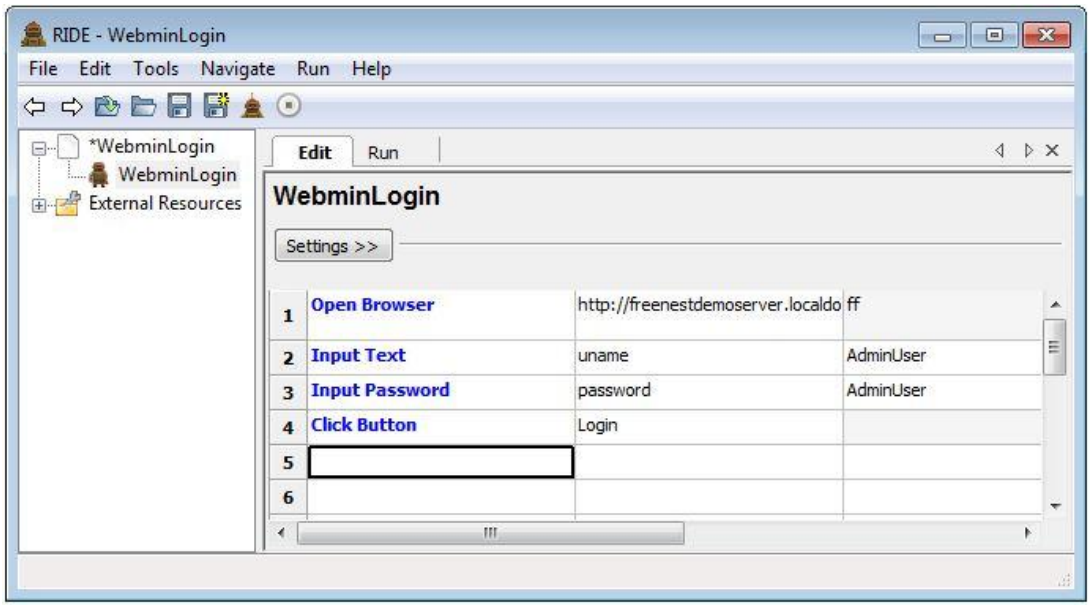


FIGURE 21. Usage of the keyword “Click Button” in RIDE.

After the login, the FreeNEST will open up on the front page (see figure 22). To get into the Webmin login, the browser has to be navigated through the top bar located

in the upper part of the screen. Webmin can be found under the “Administration” tab and the link to it is called “Nest Server Administration”.

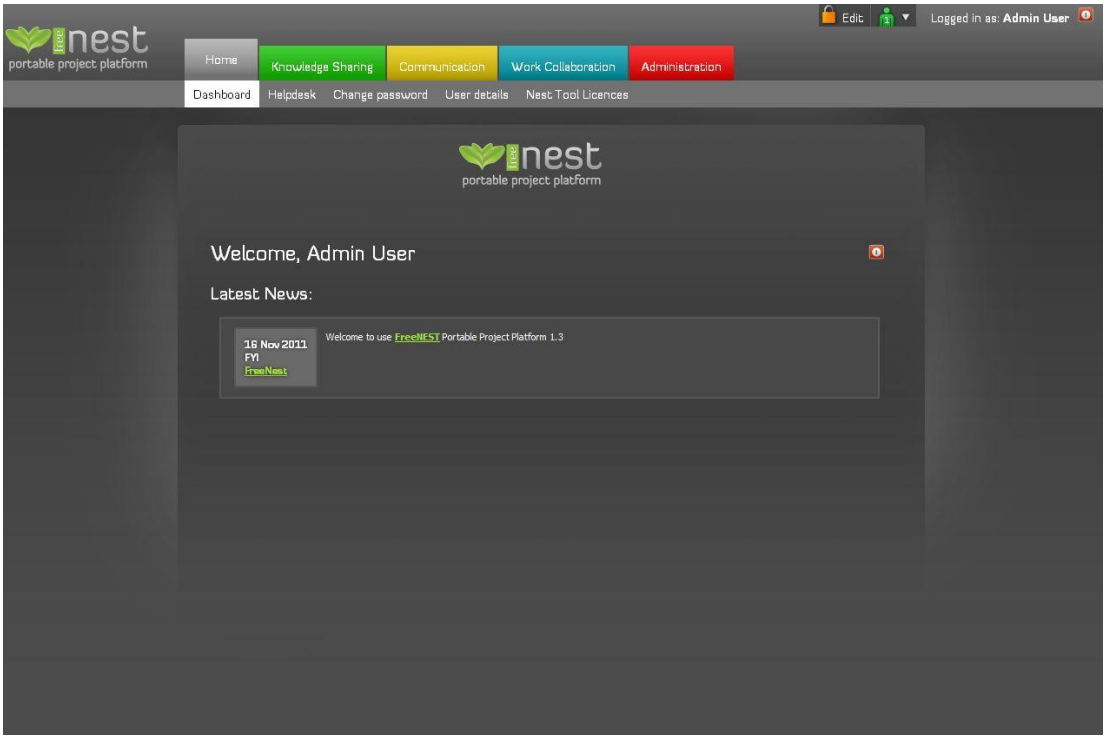


FIGURE 22. The front page of FreeNEST.

In order to navigate through these two links, the keyword “Click Link” can be used. This keyword also requires a locator, which can be found in the source code and is marked with the ID, name or href of the link. Figure 23 illustrates the required part of the source code. In this case the links are identified with href and they are both named as they are seen in the top bar. As shown in figure 24, the first click of a link requires an extra argument called “dont_wait” after the locator. Inserting arguments will add additional rules to the keywords that are driven. Whenever a click link keyword is run, the test will wait until current page has loaded in the browser. Since clicking the upper tabs of this top bar in FreeNEST does not result in a page load, this “dont_wait” argument has to be inserted or the test will wait for the page load forever and will fail in a timeout.

FIGURE 23. Identifying the link locator.

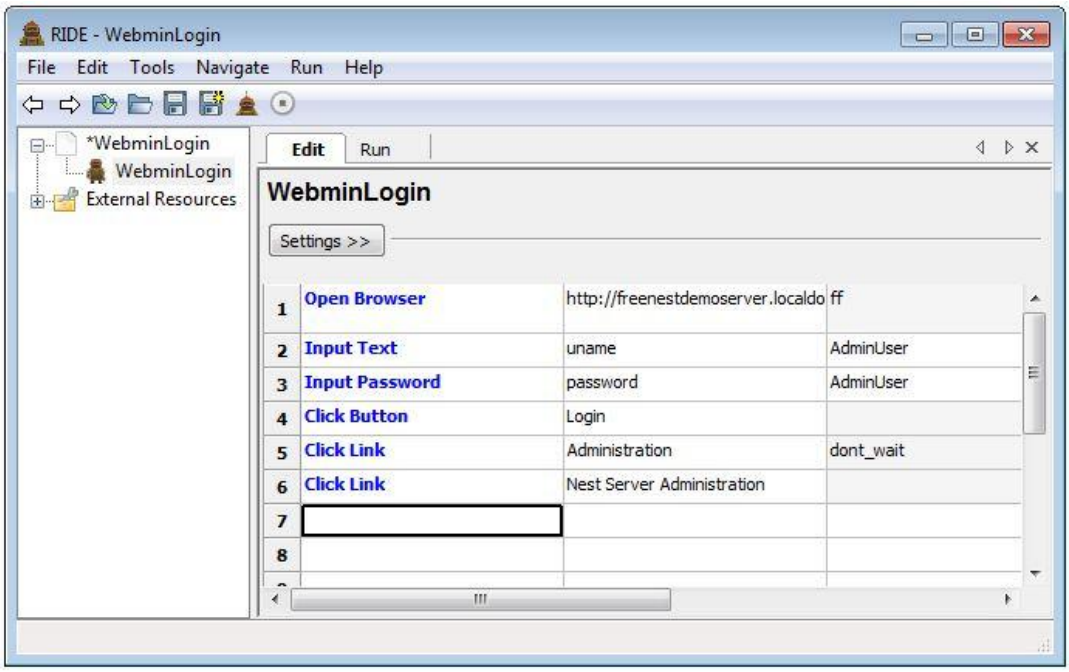


FIGURE 24. Usage of keyword “Click Link” in RIDE.

After these navigation steps, the test should now end in the login screen of Webmin, which presented in figure 25. The same method used before to log into FreeNEST can be used here. By inspecting the source code with firebug can be seen that the locator for username field is “user” and for the password field it is “pass”. The default username and password for Webmin administrator in FreeNEST is “admin”, and like before, the login button has to be pressed again. The name for it is “Login” in this case. The required keywords and their arguments have been added to the grid in figure 26.

FIGURE 25. Login screen of Webmin.

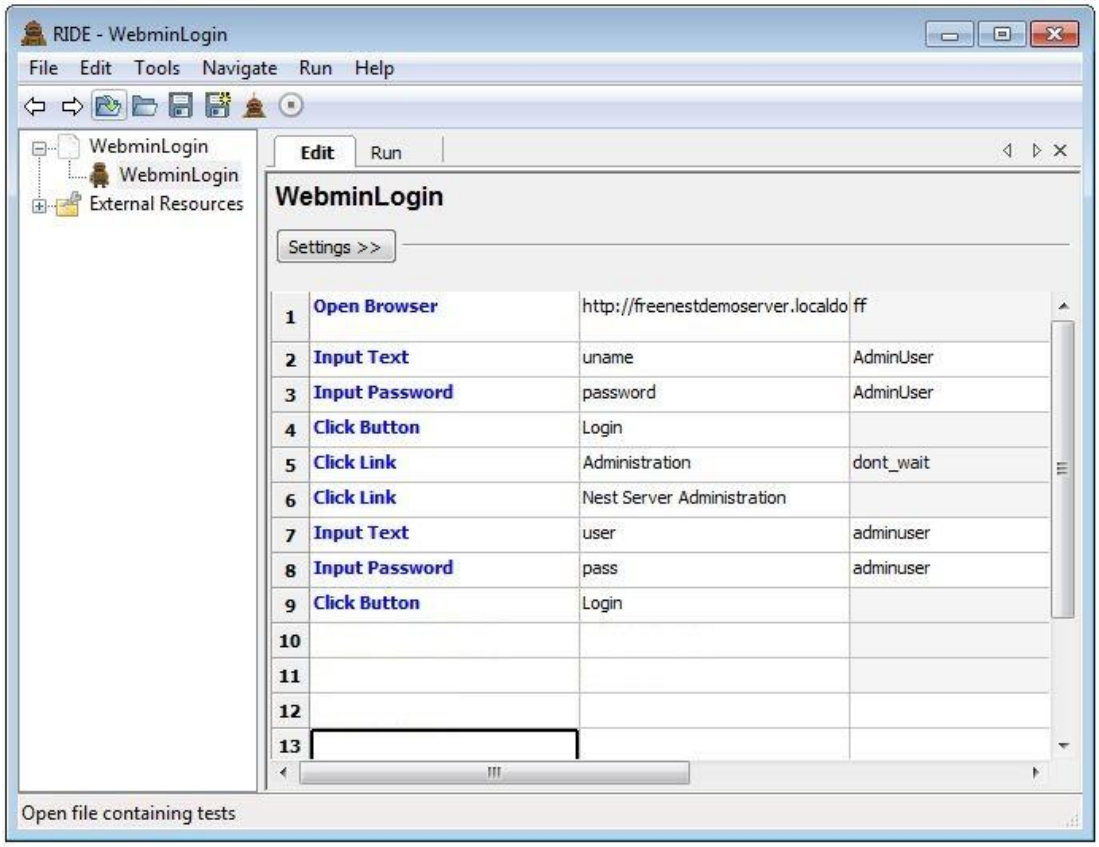


FIGURE 26. Login information for Webmin in RIDE.

Next it should be verified that the browser is currently in the front page of Webmin (see figure 27) to confirm if the test is successful or if it has failed. There are several methods of doing this and one of them is to make the test check if certain text is found on the page. Other ways are such as checking the source code for certain elements or verifying that the current address of the internet browser is in the right place. These all can be used as subsequent set of keywords; however, for this case, verifying some text from webmin should be sufficient.

FIGURE 27. Webmin main menu inside FreeNEST.

The keyword to check if certain text is found is called “Page Should Contain”. In the cell next to this keyword, a string of text has to be inserted, and this is the exact text the test will try to find on the current page. The text selected should be something that is not too common and should be found only on that certain page or the text might give a false pass status if it finds the text even if it is in a completely different site. Of course, this is a good example why multiple checks should be performed when not being sure if one is sufficient. For this text check the author has selected the “Webmin version” text string found in the main page of Webmin and it has been added to the test grid as seen in figure 28.

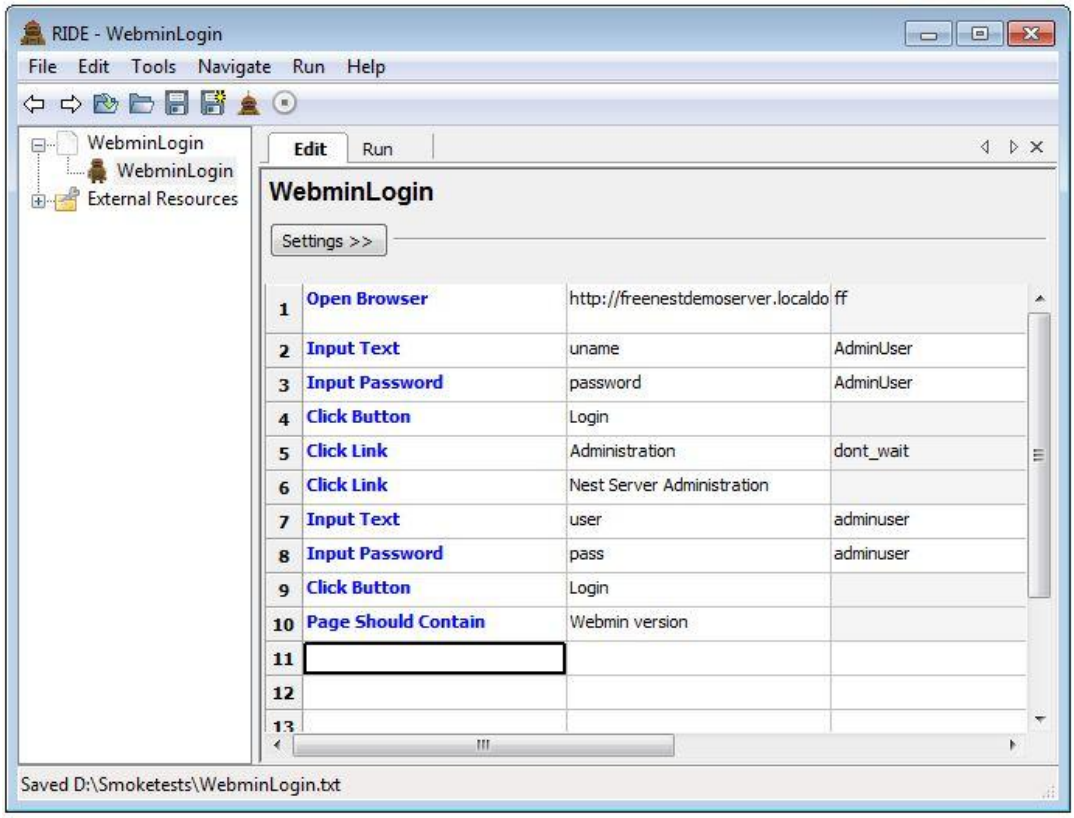


FIGURE 28. Text check in RIDE.

The current test technically ends in here, but few more keywords should be inserted in to make it more complete. The first step that should be added is to log out the user from FreeNEST to make sure it does not affect other tests run after this one. This can be done by using the log out button located in the upper right corner of FreeNEST. Again the firebug can be used to get information about the button.

FIGURE 29. Source code of log out button in FreeNEST.

As seen in figure 29, the button is not actually classified as “button”, therefore using the keyword “Click Button” cannot be used this time. Instead another one called

“Click Element” has to be used, which works very much like the “Click Button” keyword. The ID of the button can be seen in the source code and it can be added as the locator for the keyword. The last keyword to add into the test is “Close Browser”. This is not unnecessary to do, but running hundreds of tests and not closing any of the browsers will cause complications on the machine doing the test, such as decrease in performance. The finished test case can be seen in figure 30.

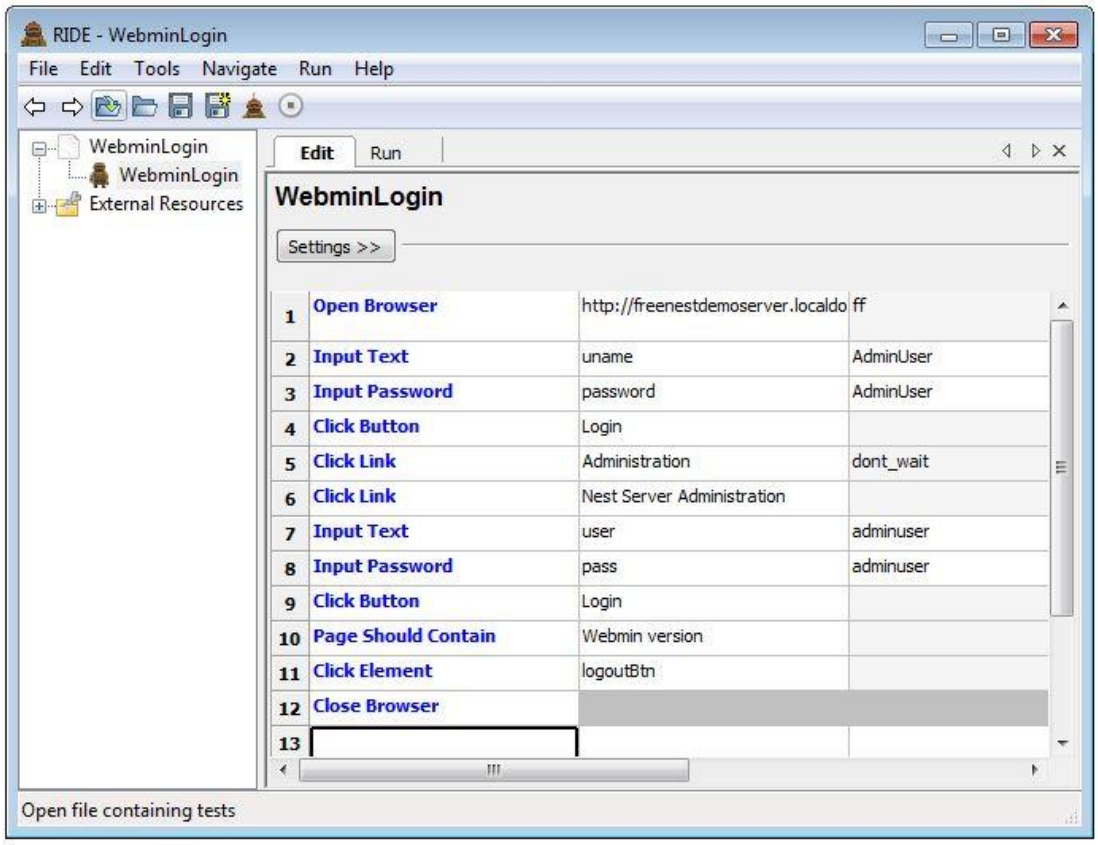


FIGURE 30. Completed test case in RIDE.

Now that the test is completed, it can be executed to make sure there are no errors found in the steps. It is also recommendable to run the test after adding few keywords while still designing it. The test can be executed by selecting on the “run” tab above the test grid and clicking on the “start” button, which is shown in figure 31. Before running the test, it needs to be made sure the execution profile is correct depending on what programming language is used. In this case it is Python so “pybot” is selected from the dropdown list.

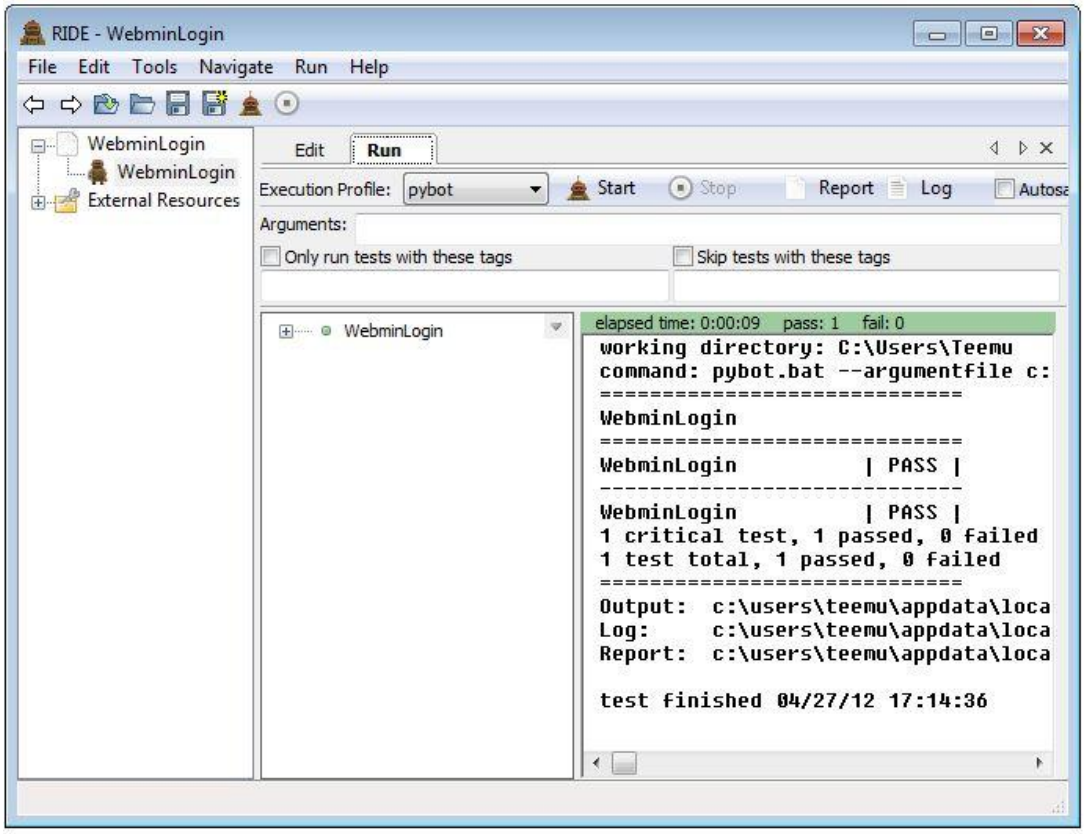
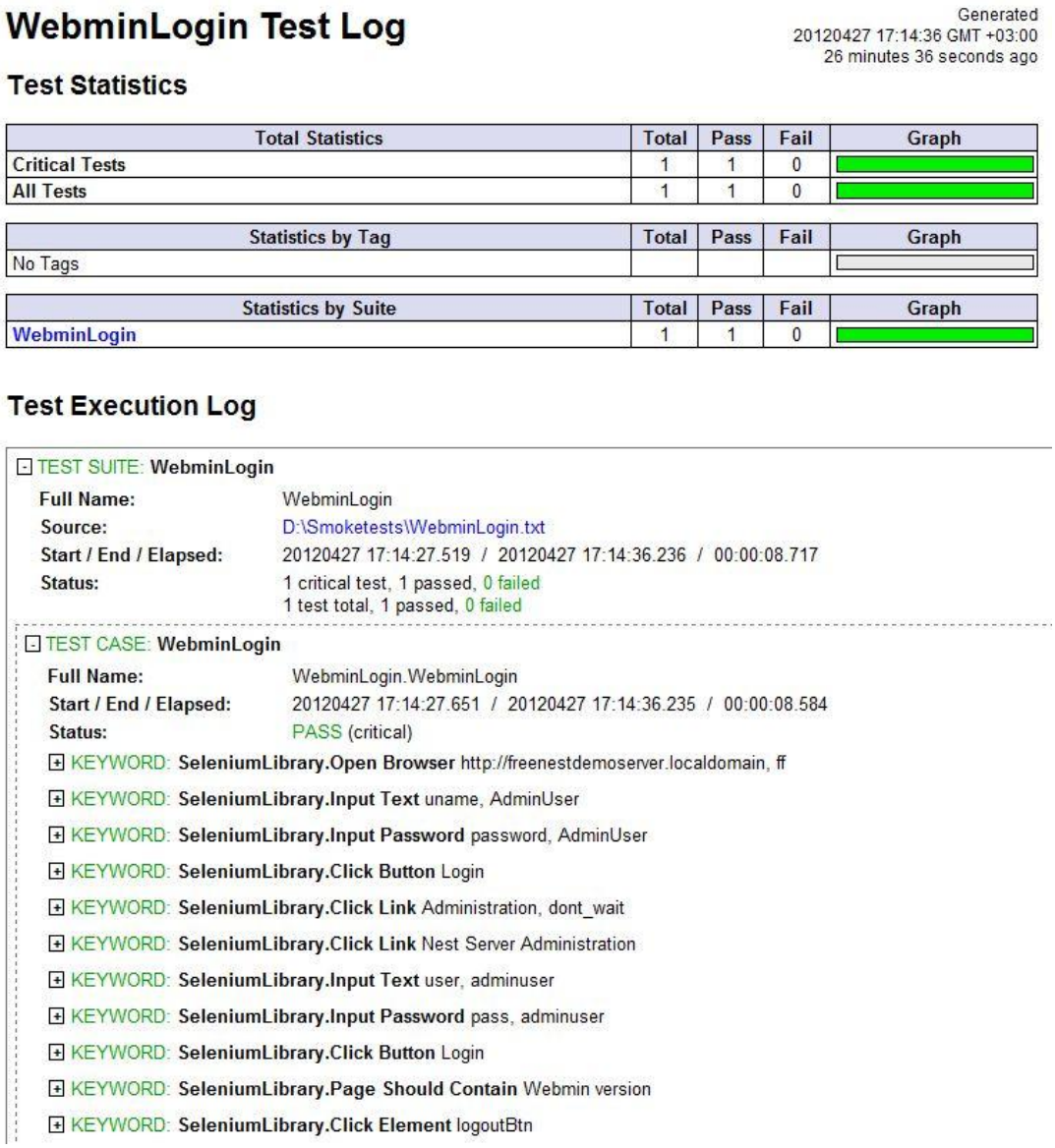


FIGURE 31. Executing a test in RIDE.

After the test has been executed, the result is immediately shown in the run screen. If everything is done right while creating the test and the target application is not defective either, the result should be “pass”. More information about the test run can be found by clicking on the “Report” and “log” buttons found next to the “Start” button. These logs can give more precise information about the test and if the test failed, it will show where and why it failed. The generated log file from this test case can be seen in figure 32.



WebminLogin Test Log

Generated
20120427 17:14:36 GMT +03:00
26 minutes 36 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Graph
Critical Tests	1	1	0	
All Tests	1	1	0	

Statistics by Tag	Total	Pass	Fail	Graph
No Tags				

Statistics by Suite	Total	Pass	Fail	Graph
WebminLogin	1	1	0	

Test Execution Log

TEST SUITE: **WebminLogin**

Full Name: WebminLogin
Source: D:\Smoketests\WebminLogin.txt
Start / End / Elapsed: 20120427 17:14:27.519 / 20120427 17:14:36.236 / 00:00:08.717
Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed

TEST CASE: **WebminLogin**

Full Name: WebminLogin.WebminLogin
Start / End / Elapsed: 20120427 17:14:27.651 / 20120427 17:14:36.235 / 00:00:08.584
Status: PASS (critical)

KEYWORD: **SeleniumLibrary.Open Browser** http://freenestdemoserver.localdomain, ff
KEYWORD: **SeleniumLibrary.Input Text** uname, AdminUser
KEYWORD: **SeleniumLibrary.Input Password** password, AdminUser
KEYWORD: **SeleniumLibrary.Click Button** Login
KEYWORD: **SeleniumLibrary.Click Link** Administration, dont_wait
KEYWORD: **SeleniumLibrary.Click Link** Nest Server Administration
KEYWORD: **SeleniumLibrary.Input Text** user, adminuser
KEYWORD: **SeleniumLibrary.Input Password** pass, adminuser
KEYWORD: **SeleniumLibrary.Click Button** Login
KEYWORD: **SeleniumLibrary.Page Should Contain** Webmin version
KEYWORD: **SeleniumLibrary.Click Element** logoutBtn

FIGURE 32. Log file of a passed test run by Robot Framework.

Normally, when one step in a test fails, the steps after that are not executed. This will cause the “Close Browser” keyword to be ignored too, thus adding this keyword to the teardown arguments in the test case settings mentioned before, will cause the browser to shut down even after the test fails. Another issue to note is that this test was designed to run on a test machine that has Selenium server running all the time. To make sure it runs in other environments too, adding the keyword “Start Selenium Server” to the first row of the test grid will launch the server every time the test is run. Additionally inserting the keyword “Stop Selenium Server” at the very last row will ensure that the server is not left running when it is not needed.

7 RESULTS

7.1 Discussion on results

The creation of the whole set of thirty smoke tests was finished in time for the completion of this thesis and for the most part, it fulfills the expectations given at the beginning of this project and all the tests are done as variably as possible for the current test execution engine. There were still some factors that dropped the possible amount of different types of tests and therefore the overall coverage of the written smoke tests and leave some areas of the different testing techniques almost completely untouched that should be used in smoke testing every new version of the FreeNEST.

Most of the tests that were included in the smoke test plan, are either navigational tests that verify that it is possible to access all the different applications and services provided by FreeNEST, or they are tests to perform authorization tests in the forms of login tests or verifying that only authorized users have access into administrative tools found in several places inside FreeNEST project platform. The minor part of the tests consists of search tests, simple integration tests and tests that edit user information. Full list of all the created test cases can be seen in tables 1 and 2.

TABLE 1. List of created test cases (part 1/2).

Test Case name	Test usage	Test type/Notes
BugzillaAdminRights	Verifies that only FreeNEST admin has access to administrative tools in Bugzilla.	Security testing, correctness testing.
BugzillaLogin	Verifies that users logged in FreeNEST are also logged into Bugzilla.	Correctness testing.
BugzillaSearch	Performs a search action in Bugzilla and verifies that results are found.	Correctness testing.
CactiLogin	Log in test for Cacti tool inside FreeNEST with correct user credentials.	Correctness testing.
ChangePassword	Changes the password of a user, confirms the change and then changes it back to the default.	Correctness testing. See appendix 2 for more information.
ChangezillaAdminRights	Verifies that only FreeNEST admin has access to admin tools in Changezilla.	Security testing, correctness testing.
ChangezillaLogin	Verifies that users logged in FreeNEST are also logged into Changezilla.	Correctness testing.
ChangezillaSearch	Performs a search action in Changezilla and verifies that results are found.	Correctness testing.
CheckTabs	Navigates through all the front pages of every component installed within FreeNEST.	Correctness testing.
CreateProfilePage	Creates a profile page in Foswiki for a user and removes it afterwards.	Correctness testing.
CreateWikiTopic	Creates a normal topic in Foswiki and removes it afterwards.	Correctness testing.
EditUser	Edits user information, confirms the change and restores the user to default state.	Correctness testing.
ForumAdminRights	Verifies that only admins have access to administrative tools in the forum.	Security testing, correctness testing.
HelpdeskAdminRights	Verifies that only FreeNEST admin access to administrative tools in Helpdesk.	Security testing, correctness testing. See appendix 1 for more information.
IrcLogin	Logs in user to IRC chat and confirms that the user can connect to the chat server.	Correctness testing. See appendix 5 for more information.

TABLE 2. List of created test cases (part 2/2).

Test Case name	Test usage	Test type/Notes
LoginAdminUser	Normally logs in the administrator of FreeNEST.	Correctness testing.
LoginLowercase	Tries to log in the user by using lowercase letters only and verifies that the log in will fail.	Security testing, correctness testing.
LoginRecentlyLoggedOutUserNoPW	Tries to log in recently logged out user without password and verifies that the log in will fail.	Security testing, correctness testing.
LoginRecentlyLoggedOutUserNoPWSameBrowser	Same as above, but without restarting the browser.	Security testing, correctness testing.
LoginWrongPassword	Tries to log in user to FreeNEST with incorrect password and verifies that the log in will fail.	Security testing, correctness testing.
TestLinkNavigation	Navigates trough links found in different pages of TestLink.	Correctness testing.
TracAdminRights	Verifies that only FreeNEST admin has access to administrative tools in Trac.	Security testing, correctness testing.
TracNavigation	Navigates trough links found in different pages of Trac.	Correctness testing. See appendix 4 for more information.
UserControlAdminRights	Verifies that only FreeNEST admin has rights to manage other users of FreeNEST.	Security testing, correctness testing.
Web2ProjectLogin	Log in test for Web2Project inside FreeNEST with correct user credentials.	Correctness testing.
WebminLogin	Log in test for Webmin inside FreeNEST with correct user credentials.	Correctness testing. Creation of the test case explained in chapter 6.2
WikiAdminRights	Verifies that only FreeNEST admin has access to administrative tools in Foswiki.	Security testing, correctness testing.
WikiNavigation	Navigates trough links found in different pages of Foswiki.	Correctness testing.
WikiSearch	Performs a search action in Foswiki and verifies that results are found.	Correctness testing. See appendix 3 for more information.
WikiwordDashboard	Confirms the Foswiki links created in front page of FreeNEST are working.	Correctness testing

The factors that prevented the creation of more perfect smoke test plan were the restrictions in the current test execution engine and my personal knowledge regarding automated testing. For example creating tests that repeat endlessly in a loop would have required some more know-how about programming, since the tests created for robot framework can be improved by using Python program language inside the test cases and to create new test libraries. In addition getting more precise and profound information about all the different ways Robot Framework is possible to create and perform tests would have required a great deal more time that I simply did not have anymore. In the end, what I managed to create was a guideline for the future automated tests that will be created in the future by new members in the project.

7.2 Future improvements

Since the current smoke tests mostly test the correctness and the security areas of the four different possible methods of testing mentioned in chapter 3.2 (Testing Techniques), including tests that concentrate on testing the performance and reliability of the target software would make the smoke test plan more complete. When the OpenStack cloud the project uses gets improved both technically and materially, the limits of the test performance can be minimized. Getting new hardware on the OpenStack cloud and optimizing it to its fullest will increase the overall performance of the cloud, giving the option to replace the target machine as quickly and as often as needed. This will allow the adding of tests that make permanent changes on the target machine, such as create thousands of users to the database to see the limits of the system.

Additionally implementing the Selenium Grid (see chapter 4.5) makes it possible to uphold multiple test executors at once. Each one of them can perform tests on different target machines at the same time, adding some flexibility and efficiency to the automated testing; however, this too requires more performance from the cloud it is executed on. On top of these, adding automated and more precise test log reporting back to the test management machine and implementing a way to handle

test case updates more efficiently when needed with the help of the Git version control system, the test automation engine will be in near perfect state. Some of these improvements are already being researched by other thesis workers and members of the project and will be implemented gradually in the future into the test execution machines.

SOURCES

Caruso, J. 2011. IaaS vs PaaS vs SaaS. Article about cloud computing in Network World. Referred on April 29, 2012. http://www.networkworld.com/news/2011/102511-tech-argument-iaas-paas-saas-252357.html

Haikala, I & Märijärvi, J. 2006. Ohjelmistotuotanto. Helsinki: Talentum.

IBM. n.d. Platform as a service (PaaS). Referred on April 28, 2012. http://www.ibm.com/cloud-computing/us/en/paas.html

Järvinen, J. n.d. Cloud Software Program. Referred on May 4, 2012. http://www.cloudsoftwareprogram.org/cloud-software-program

Khan, E. 2010. Different Forms of Software Testing Techniques for Finding Errors. Referred on April 26, 2012. http://www.ijcsi.org/papers/7-3-1-11-16.pdf

Krishna Training. 2011. Unit, Integration and System testing. Referred on April 26, 2012. http://www.krishnatraining.com/unit-integration-and-system-testing/

McBlain, A. 2011. Firebug Guide for Web Designers. Referred on May 04, 2012. http://sixrevisions.com/tools/firebug-guide-web-designers/

Myers, G. 2004. The Art of Software Testing, Second Edition. Hoboken, N.J: John Wiley & Sons.

NIST. n.d. Cloud Computing Program. Article by National Institute of Standards and Technology. Referred on April 25, 2012. http://nist.gov/itl/cloud/

OneStopTesting. n.d. Black Box Testing. Referred on April 26, 2012. http://www.onestoptesting.com/blackbox-testing/

OpenStack. n.d. OpenStack: The Open Source Cloud Operating System. Referred on April 29, 2012. http://openstack.org/projects/

Outsourcebazaar. 2006. Automated vs Manual testing. Referred on April 17, 2012. http://www.outsourcebazaar.com/index_Article_AutomatedVsManualTesing.html

RIDE. n.d. How To guide for RIDE. Referred on May 04, 2012. https://github.com/robotframework/RIDE/wiki/How-To

Robot Framework. 2012. Robot Framework User Guide. Referred on April 11, 2012. http://robotframework.googlecode.com/hg/doc/userguide/RobotFrameworkUserGuide.html?r=2.7.1

Selenium. 2012. Introduction to Selenium. Referred on April 24, 2012. http://seleniumhq.org/docs/01_introducing_selenium.html

Selenium Grid. n.d. How it Works. Referred on April 30, 2012. http://selenium-grid.seleniumhq.org/how_it_works.html

SeleniumHQ. n.d. What is Selenium? Referred on April 24, 2012. http://seleniumhq.org

Smartbear. n.d. Why Automated Testing? Referred on April 25, 2012. http://support.smartbear.com/articles/testcomplete/manager-overview/

TestLink. 2012. TestLink User Manual. Referred on April 5, 2012. http://www.teamst.org/_tldoc/1.9/testlink_user_manual.pdf

Waxer, C. n.d. The Benefits of Cloud Computing. Referred on April 25, 2012. http://www.webhostingunleashed.com/features/cloud-computing-benefits/

Webopedia. n.d. Saas – Software as a Service, Storage as a Service. Referred on April 25, 2012. http://www.webopedia.com/TERM/S/SaaS.html

Wikipedia. n.d. Cloud Computing. Referred on April 28, 2012. http://en.wikipedia.org/wiki/Cloud_computing

APPENDIX 1. TESTCASE THAT VERIFIES ADMIN RIGHTS ON HELPDESK.

This test case will verify that only user with administrator rights will have access to the administrative tools in Helpdesk. AdminUser should be automatically recognized as administrator in Helpdesk immediately after logging into FreeNEST and DemoUser should have access only to the normal users section.

```
*** Settings ***
Library         SeleniumLibrary

*** Test Cases ***
HelpdeskAdminRights
    Open Browser  http://freenestdemoserver.localdomain/  ff
    Input Text  uname  AdminUser
    Input Password  password  AdminUser
    Click Button  Login
    Click Link  Helpdesk
    Page Should Contain  Welcome AdminUser Admin
    Click Link  Setup
    Page Should Contain  Select the category to configure
    Click Element  logoutBtn
    Close browser
    Open Browser  http://freenestdemoserver.localdomain/  ff
    Input Text  uname  DemoUser
    Input Password  password  DemoUser
    Click Button  Login
    Click Link  Helpdesk
    Page Should Contain  Describe the Problem/Action
    Page Should Not Contain  Welcome DemoUser Admin
    Click Element  logoutBtn
    Close All Browsers
```

APPENDIX 2. TESTCASE FOR TESTING PASSWORD CHANGE.

This test case will change the password of DemoUser, one of the default users in FreeNEST. It will then verify the change by logging in again with the new password and in the end will change the password back to the original, thus making the test executable multiple times.

```
*** Settings ***
Library         SeleniumLibrary

*** Test Cases ***
ChangePassword
    Open Browser  http://freenestdemoserver.localdomain/  ff
    Input Text  uname  DemoUser
    Input Password  password  DemoUser
    Click Button  Login
    Click Link  Change password
    Input Text  current_passwd  DemoUser
    Input Password  new_passwd  demo
    Input Password  verify_new_passwd  demo
    Click Button  btnChange  dont_wait
    Run Keyword And Ignore Error  Confirm Action
    Run Keyword And Ignore Error  Click Element  logoutBtn
    Close Browser
    Open Browser  http://freenestdemoserver.localdomain/  ff
    Input Text  uname  DemoUser
    Input Password  password  demo
    Click Button  Login
    Click Link  Change password
    Input Text  current_passwd  demo
    Input Password  new_passwd  DemoUser
    Input Password  verify_new_passwd  DemoUser
    Click Button  btnChange  dont_wait
    Click Element  logoutBtn
    Close Browser
```

APPENDIX 3. TESTCASE TO VERIFY THE FUNCTIONALITY OF SEARCH IN FOSWIKI.

This test case performs a simple search with the search function of FosWiki. The search function does not have a button to launch the search, therefore “Press Key” keyword has to be used to launch it. 13 is the id cumber for “Enter”.

```
*** Settings ***
Library         SeleniumLibrary

*** Test Cases ***
WikiSearch
    Open Browser  http://freenestdemoserver.localdomain/  ff
    Input Text  uname  AdminUser
    Input Password  password  AdminUser
    Click Button  Login
    Click Link  Knowledge Sharing  dont_wait
    Click Link  Wiki
    Input Text  quickSearchBox  Example
    Press Key  quickSearchBox  13
    Sleep  2 Seconds
    Page Should Contain  FaqExample
    Click Element  logoutBtn
    Close All Browsers
```

APPENDIX 4. TESTCASE THAT NAVIGATES THROUGH DIFFERENT PARTS OF TRAC.

This relatively long test case will navigate in Trac project management service and verify that all the links in the main menu are working.

```
*** Settings ***
Library         SeleniumLibrary

*** Test Cases ***
TracNavigation
    Open Browser  http://freenestdemoserver.localdomain/  ff
    Input Text  uname  AdminUser
    Input Password  password  AdminUser
    Click Button  Login
    Click Link  Work Collaboration  dont_wait
    Click Link  Tickets & Tasking
    Click Link  New Requirement
    Set Selenium Speed  0.2 seconds
    Page Should Contain  Create New Requirement
    Click Link  New Task
    Page Should Contain  Create New Task
    Click Link  New User Story
    Page Should Contain  Create New User Story
    Click Link  New Bug
    Page Should Contain  Create New Bug
    Click Link  Product Backlog
    Page Should Contain  Product Backlog
    Page Should Contain  Business Value
    Click Link  2
    Page Should Contain  This is example of traceable object
    Click Link  Active Tickets
    Page Should Contain  List all active tickets by priority.
    Click Link  Active Tickets by Version
    Page Should Contain  grouping results by version
```

```
    Click Link  Active Tickets by Milestone
    Page Should Contain  grouping results by milestone
    Click Link  Accepted, Active Tickets by Owner
    Page Should Contain  List accepted tickets
    Click Link  Accepted, Active Tickets by Owner (Full
Description)
    Page Should Contain  demonstrates the use of full-row
display
    Click Link  All Tickets By Milestone (Including closed)
    Page Should Contain  complex example to show how to make
advanced reports
    Click Link  My Tickets
    Page Should Contain  This report demonstrates the use of
the automatically set USER dynamic variable
    Click Link  Active Tickets, Mine first
    Page Should Contain  List all active tickets by priority
    Click Link  Index by Title
    Page Should Contain  Index by Title | Index by Date
    Click Link  WikiStart
    Page Should Contain  Welcome to FreeNEST TRAC
    Click Link  TracGuide
    Page Should Contain  The Trac User and Administration Guide
    Click Link  WikiPageNames
    Page Should Contain  Wiki Page Names
    Click Link  TracPermissions
    Page Should Contain  Trac Permissions
    Click Link  TracWiki
    Page Should Contain  The Trac Wiki Engine
    Click Element  logoutBtn
    Close All Browsers
```

APPENDIX 5. TESTCASE TO VERIFY THE IRC LOGIN.

This test case will log user into IRC chat and verify the connection is successful. Since the IRC opens up in a pop-up window, the active window has to be switched twice and a certain frame has to be selected in the IRC window. Furthermore, during the log in, it is recommendable to use the "sleep" keyword to pause the test to make sure that the user has enough time to connect to the IRC chat room before executing the "Current Frame Should Contain" keyword.

```
*** Settings ***
Library         SeleniumLibrary

*** Test Cases ***
IrcLogin
    Open Browser  http://freenestdemoserver.localdomain/  ff
    Input Text  uname  AdminUser
    Input Password  password  AdminUser
    Click Button  Login
    Click Link  Communication  dont_wait
    Click Link  IRC Chat  dont_wait
    Get Window Names
    Select Window  popUpProjectIRC
    Click Button  Login  dont_wait
    Wait Until Page Contains Element  fmain  timeout=20
    Select Frame  fmain
    Sleep     10 seconds
    current frame should contain  has joined
    Close Window
    Select Window  main
    Click Element  logoutBtn
    Close All Browsers
```