

Department of Veterans Affairs

Open Source Electronic Health Record (EHR) Services

Web Application Automated Testing Framework (WAATF) Software Design Document (SDD)



Version 1.0

September 2013

Revision History

Date	Version	Description	Author
09/23/2013	0.1	Initial Draft	Andal FeQuiere
09/24/2013	0.2	Peer Review	Meredith Watkins
09/24/2013	0.3	Updates per Peer Review	Andal FeQuiere
09/24/2013	0.4	Formal Review	Paul Bradley
09/25/2013	0.5	Updates per Formal Review	Andal FeQuiere
09/25/2013	1.0	Final Review	Paul Bradley

Table of Contents

1.	Introduction.....	3
1.1.	Purpose	3
1.2.	Scope and Audience.....	3
1.3.	Acronyms and Definitions	4
2.	Tools.....	5
3.	Tool Application.....	6
4.	Framework Architecture.....	6
4.1.	High level Architecture.....	6
4.2.	AutomatedTestingSuite Object.....	7
4.3.	Selenium Tests.....	7
4.4.	Logging System	10
5.	Installation and Setup.....	13
5.1.	Clone Automated Functional Testing Source Code	13
5.2.	Test Structure and Setup	13
5.3.	Testing Execution	15

1. Introduction

The Department of Veterans Affairs (VA) has contributed the latest U.S. Department of State Freedom of Information Act (FOIA) release of the Veterans Health Information Systems and Technology Architecture (VistA) codebase to Open Source Electronic Health Record Agent (OSEHRA), the custodial agent that serves as the central governing body of a new open source community. The VA has begun to make VistA more modular and easier for developers to work with, and has also begun the deployment of the open source version of VistA inside the VA. In order to accelerate this work, the VA is in need of a comprehensive testing framework to verify refactored code, enhancements, and bug fixes.

The Web Application Automated Testing Framework (WAATF) was developed, in response to a need, to provide automated testing support for web applications. The WAATF uses cutting edge build tools and architecture to provide ease in use and development of test scripts. The WAATF also provides scripts to perform functional testing of any web application on remote driven web browsers.

1.1. Purpose

The purpose of this document is to define the tools, concepts, assumptions, and structure used within the automated testing framework for web application automated testing.

The approach described below is a hybrid framework using both data-driven and modularized code techniques. This framework is not restricted to only the code being produced by this specific project; eventually these concepts will be integrated into the automated testing framework currently under development by OSEHRA, incorporating the testing for all code received by OSEHRA.

This plan aims at outlining an automated testing infrastructure that supports the creation, management, and execution of a rich set of functional tests constructed for a variety of purposes including functional verification, regression testing, and certification of the abovementioned code.

1.2. Scope and Audience

The following text describes the automated testing framework used for functional verification of the MyHealtheVet (MHV) application. The WAATF can be used to test any web application. MHV is provided as a reference. The framework described here can be used to test functionally of any feature within the MHV application.

The test framework is the set of tools, concepts, assumptions, software, and structure that enables automated software testing. This framework has the overarching objective of simplifying test creation, configuration, and modification.

1.3. Acronyms and Definitions

Acronym	Term
AUT	Automated Unit Test
EE	Enterprise Edition
EHR	Electronic Health Record
FOIA	Freedom of Information Act
IDE	Integrated Development Environment
JDK	Java Development Kit
ME	Micro Edition
MHV	My Health e Vet
MVC	Model View Controller
OSEHRA	Open Source Electronic Health Record Agent
SDD	Software Design Document
SDK	Software Development Kit
SE	Standard Edition
VA	Department of Veterans Affairs
VistA	Veterans Health Information Systems and Technology Architecture
WAATF	Web Application Automated Testing Framework

2. Tools

Name	Version	Description	Information
CMake	2.8.7	CMake is a family of tools designed to build, test, and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the Compiler environment of your choice.	http://www.cmake.org/ http://www.cmake.org/cmake/resources/software.html
Java Development Kit (JDK)	6	The JDK is an implementation of either one of the Java Standard Edition (SE), Java Enterprise Edition (EE), or Java Micro Edition (ME) platforms released by Oracle Corporation in the form of a binary product aimed at Java developers on Solaris, Linux, Mac OS X or Windows. Since the introduction of Java platform, it has been by far the most widely used Software Development Kit (SDK).	http://www.oracle.com/technetwork/java/index.html http://www.oracle.com/technetwork/java/javase/downloads/index.html
GitBash	1.8.0	Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It's comprised of two parts that include: <ul style="list-style-type: none">– Git: A version control tool– Bash: A shell that runs commands once you type the name of a command and press 'Enter' on your keyboard	http://git-scm.com/downloads
Eclipse (optional Integrated Development Environment [IDE])	Indigo	Eclipse is a community for individuals and organizations who wish to collaborate on commercially-friendly open source software. Its projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software	http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/indigosr2

		across the lifecycle. The Eclipse Foundation is a not-for-profit, member supported corporation that hosts the Eclipse projects and helps cultivate both an open source community and an ecosystem of complementary products and services.	
EGit		Git Eclipse Plug-in	Git Plug-in http://www.eclipse.org/egit/

3. Tool Application

- CMake – Used to build the automated test script
- Java – Create and test scripting language
- GitBash – Bash shell used for test execution and Git operations
- Eclipse – Optional development IDE that includes Git plug-in (EGit)
- Selenium – Used to record and replay test scripts in a web browser

4. Framework Architecture

4.1. High level Architecture

The WAATF project Architecture is composed of four main packages:

- Controller – AllTestSuites.java used for launching all tests
- Utils – Contains files to assist with logging and other utility functions
- Templates – Templates which represent a subset of data in the properties file; modeled for a specific functionality
- Suites – Consists of a library of individual test case scripts to be executed

Figure 1 below displays a high level diagram of how the Controller, Templates, and Suite objects are related. The architecture for this framework greatly relates to that of Model View Controller (MVC), which is a software architecture pattern which separates the representation of information from the user's interaction. The Controller object is AllTestSuite class, which sends commands to its associated view and controls the flow of the application. The Controller object launches each individual Suite object. The View consists of each Suite objects requests for information from the model that it needs for generating an output representation to the user. Each View Suite object correlates to a specific test case and functionality to be executed. Although the tests themselves are unique, there are similar functions such as login, logout, and future functions that are shared amongst all test cases. To allow for code reusability, each major action throughout the test framework has been modularized such that it can be referenced in any test. The Model consists of Template objects and properties files that notify its associated views and controllers when there has been a change in its state. The Template object is used to

model data from the corresponding properties file and represent a major function for the particular test case. Currently there exists only two templates: Login and Profile.

Each Suite Object also has a properties file which is used to provide data to the templates needed for that particular test. The naming convention of the properties file is the name of the Suite Object followed with a .properties extension. The properties file in conjunction with the templates provides a layer of abstraction such that the tester can make changes to the data passed to the application without having to modify any code. Thus, allowing the test framework to operate as data driven while also implementing MVC architecture.

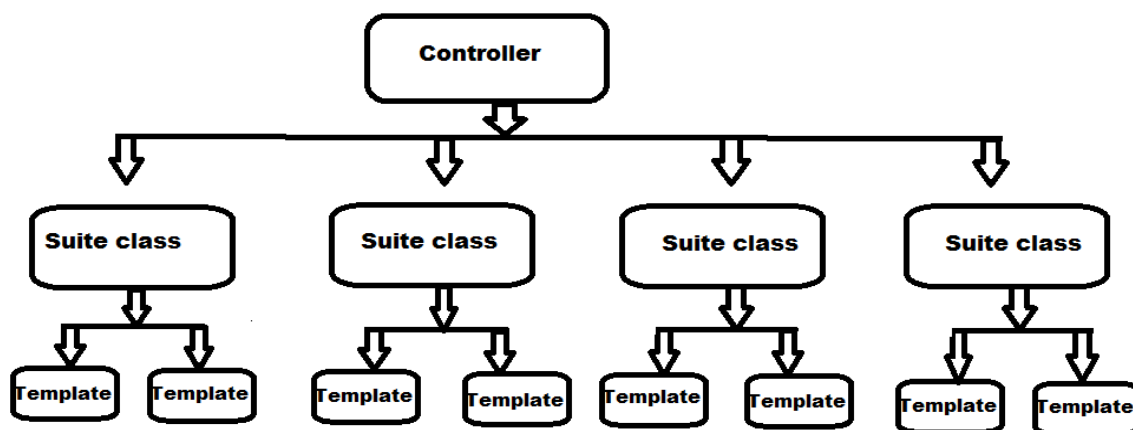


Figure 1: Architecture Diagram

4.2. AutomatedTestingSuite Object

The AutomatedTestingSuite class is an abstract class which can be extended by any test suite class. This class was created as a means to provide an additional layer of abstraction for building tests and also to be used as an object which holds reusable functions. The AutomatedTestingSuite class manages logging, Selenium web drivers, test properties file management, and test verification through verification statements. Utilizing the AutomatedTestingSuite class allows extended test classes to simply test case creation as well as reduced the complexity and number of lines of code generated.

4.3. Selenium Tests

The example automated tests, that are included in the framework, are for the MHV application and uses Selenium to emulate user interaction. Selenium is an application tool, primarily used for automating web applications for testing purposes. To display Selenium functionality, two tests were developed, LoginLogout and Profile tests. Displayed below is an image of the file system structure with the tests included in the org.automatedtesting.suites package:

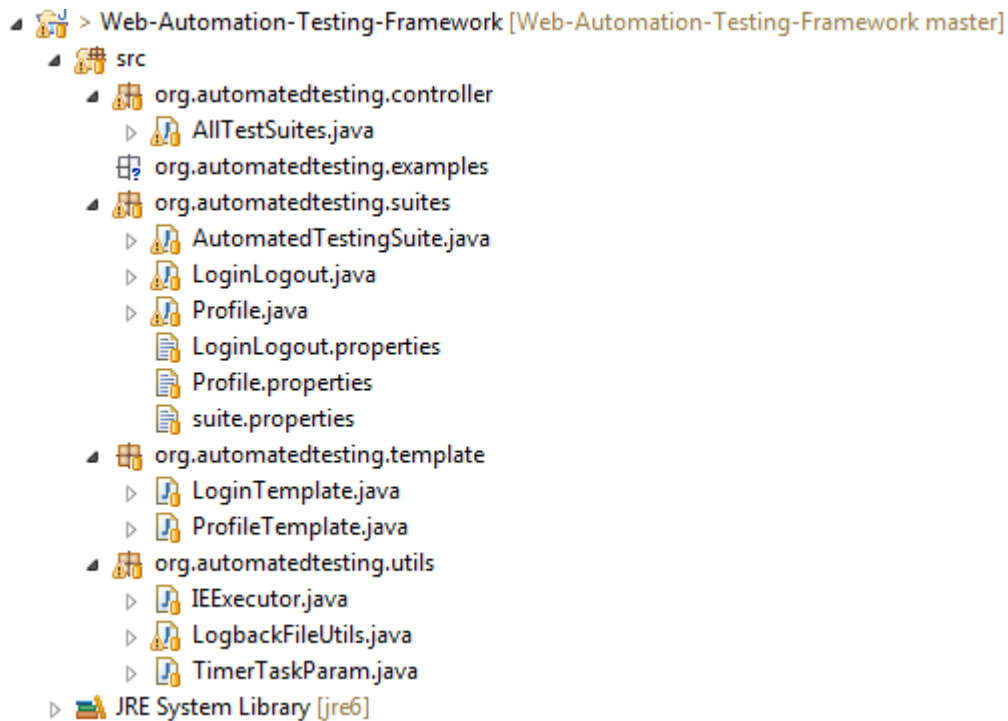


Figure 2: Test Framework packages

The LoginLogout test, performs logging in to the MHV application, verifies that the user is logged in and then logs the user out. Using Selenium the system can emulate a user typing their username and password credentials and logging in the application followed by logging out. Displayed below is an example of the test code:

```

public class LoginLogout extends AutomatedTestingSuite {
    //private static StringBuffer verificationErrors = new StringBuffer();
    //public Properties propertyFile = getTestProperties("vet1");
    LoginTemplate loginTemp = new LoginTemplate(propertyFile);

    @Test
    public HashMap LoginLogout() {

        try{
            strTime = System.currentTimeMillis();
            LogbackFileUtils.start(this.getClass());
            logger.info(":: In Start of LoginLogout() method");
            //r.delay(5000);
            logger.info("*****Start open application*****");
            driver.get(getProperties().getProperty("URL"));
            logger.info("::getting the URL from the properties file :: The URL is ::"+getProperties().getProperty("URL"));
            r.delay(15000);
            //Login
            login(loginTemp);
            logger.info("::Logged in successful::");
            r.delay(10000);

            //Logout
            logger.info("::End of LoginLogout() method ::");
            logout();
            LogbackFileUtils.stop();

            return returnObj;
        }catch(Exception e){
            logger.error("Test Failure: " + e.toString());
            closeDriver();
            endTime = System.currentTimeMillis();
            buildTestStatusObj(true);
            LogbackFileUtils.stop();
            return returnObj;
        }
    }

    public static void login(LoginTemplate loginTemp){
        //Verification for login page
        verify("Test -- Verify the Member Login header text",By.xpath("//div[@class='bea-portal-login-window-content']/div/h2"),loginTemp.getLoginHeaderLbl());

        String username = loginTemp.getLoginUserName().trim();
        String password = loginTemp.getLoginPassword().trim();
        driver.findElement(By.name("loginPortlet_homepage{actionForm.userName}")).clear();
        driver.findElement(By.name("loginPortlet_homepage{actionForm.userName}")).sendKeys(username);
        driver.findElement(By.name("loginPortlet_homepage{actionForm.password}")).clear();
        driver.findElement(By.name("loginPortlet_homepage{actionForm.password}")).sendKeys(password);
        driver.findElement(By.cssSelector("input.mhv-input-button")).click();
    }

    public static void logout(){
        //Logout and verify if test pass or failed
        //driver.switchTo().defaultContent();
        driver.findElement(By.linkText("Logout")).click();
        logger.info("::The Logout button was clicked::");
        closeDriver();

        //Build returnObject for Test
        endTime = System.currentTimeMillis();
        buildTestStatusObj(false);
    }

    public static WebElement findLogout(){
        return driver.findElement(By.linkText("Logout"));
    }

}
//end of class.

```

Figure 3: LoginLogout Test Script

The Profile test, performs logging in to the MHV application, verifies that the user is logged in, goes to My Profile, updates the following fields: street, city, state, and zip code, saves changes and then logs the user out. Using Selenium the system can emulate a user logging in and updating their Profile information as well as other actions. Displayed below is an example of the test code:

```

public class Profile extends AutomatedTestingSuite {
    //private static StringBuffer verificationErrors = new StringBuffer();
    //public Properties propertyFile = getTestProperties("vet1");
    LoginTemplate loginTemp = new LoginTemplate(propertyFile);
    ProfileTemplate profileTemp = new ProfileTemplate(propertyFile);

    @Test
    public HashMap RunProfile() {
        try{
            strTime = System.currentTimeMillis();
            LogbackFileUtils.start(this.getClass());
            logger.info(":: In Start of RunProfile() method");
            //r.delay(5000);
            logger.info("*****Start open application*****");
            driver.get(getProperties().getProperty("URL"));
            logger.info("::getting the URL from the properties file :: The URL is ::"+getProperties().getProperty("URL"));
            r.delay(5000);

            //Login SEP
            LoginLogout.login(loginTemp);
            r.delay(5000);

            Profile.updateAddress(profileTemp);
            r.delay(5000);

            //Logout
            LoginLogout.logout();
            LogbackFileUtils.stop();

            return returnObj;
        }catch(Throwable e){
            logger.error("Test Failure: " + e.toString());
            closeDriver();
            endTime = System.currentTimeMillis();
            buildTestStatusObj(true);
            LogbackFileUtils.stop();
            return returnObj;
        }
    }

    public static void updateAddress(ProfileTemplate profileTemp){
        driver.findElement(By.linkText("PERSONAL INFORMATION")).click();
        r.delay(5000);
        driver.findElement(By.linkText("My Profile")).click();
        r.delay(5000);
        verify("Test -- Verify the Update Profile header label text",By.xpath("//div[@class='mhv-portlet-title']/h2"), profileTemp.getProfileHeaderLbl());

        driver.findElement(By.id("street1")).clear();
        driver.findElement(By.id("street1")).sendKeys(profileTemp.getProfileStreet1());
        driver.findElement(By.id("city")).clear();
        driver.findElement(By.id("city")).sendKeys(profileTemp.getProfileCity());
        driver.findElement(By.id("postalCode")).clear();
        driver.findElement(By.id("postalCode")).sendKeys(profileTemp.getProfilePostalCode());
        new Select(driver.findElement(By.id("state"))).selectByVisibleText(profileTemp.getProfileState());
        driver.findElement(By.cssSelector("input.mhv-input-button")).click();
        r.delay(5000);

        if(driver.findElement(By.id("middleName")).getText().trim().isEmpty())
            driver.findElement(By.name("manageUserProfile_profilesactionOverride:saveNoMiddleNameAction")).click();
        r.delay(3000);
        verify("Test -- Verify update success",By.xpath("//div[@id='mhv-error-message-informational']"), profileTemp.getProfileUpdateSuccess());
    }
}
//end of class.

```

Figure 4: Profile Test Script

4.4. Logging System

The results of each test will be maintained in a result file which will display Pass/Fail indications for each test. If a test failure occurs, a brief error message may be included in the result file. In addition to the result file, one or more log files will be created for each test to record the

interaction between the test harness and the automated unit test (AUT). Detailed error messages will be located in the log files.

The Logging system included the usage of SLF4J and standard Java logging utilities. Currently Selenium utilizes standard Java logging utilities, but with technology advancements come logging advancements that currently include SLF4J. SLF4J is a Java library which serves as an abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time without any code redesign.

The following Java jar files were added to the project to allow the application to view logs that Selenium generated:

- SLF4J-API-1.7.5.jar
- JUL-to-SLF4J-1.7.5.jar
- Logback-core-1.0.7.jar
- Logback-classic-1.0.7.jar

Logback is an implementation of SLF4J, which was used to assist in generation of individual log files for each test suite. The configuration was implemented in a LogbackFileUtils.java file and as well as the AllTestSuites object to bridge the gap in Java standard logging and SLF4J.

The following code displayed below made these enhancements possible:

```
// Optionally remove existing handlers attached to j.u.l root logger
SLF4JBridgeHandler.removeHandlersForRootLogger(); // (since SLF4J 1.6.5)
SLF4JBridgeHandler.install();
LogManager.getLogManager().getLogger("").setLevel(java.util.logging.Level.FINEST);

logger = LoggerFactory.getLogger(mainClass.getClass());
LogbackFileUtils.start(mainClass.getClass());

logger.info("----- Web Automation Test Suites -----");
printResults(new LoginLogout().LoginLogout());
Thread.sleep(5000);
printResults(new Profile().RunProfile());

LogbackFileUtils.stop();
```

Figure 5: JUL to SLF4J and Logback

```

public class LogbackFileUtils {
    private static FileAppender<ILoggingEvent> fileAppender;
    private static FileAppender<ILoggingEvent> fileAppenderSelenium;
    private static boolean initialized = false;

    public static void init(Class classname) {
        LoggerContext loggerContext = (LoggerContext) LoggerFactory.getILoggerFactory();
        Logger myLogger = loggerContext.getLogger(classname);

        PatternLayoutEncoder encoder = new PatternLayoutEncoder();
        encoder.setContext(loggerContext);
        encoder.setPattern("%d{HH:mm:ss.SSS} [%-5level] %msg %n");
        encoder.start();

        fileAppender = new FileAppender<ILoggingEvent>();
        fileAppender.setContext(loggerContext);
        fileAppender.setName(classname.getSimpleName());
        fileAppender.setAppend(false);
        fileAppender.setEncoder(encoder);
        myLogger.addAppender(fileAppender);
        myLogger.setLevel(Level.DEBUG);

        //Selenium file appender
        if (isAutomatedTestingSuiteClass(classname)) {
            Logger myLoggerSelenium = loggerContext.getLogger(RemoteWebDriver.class);
            fileAppenderSelenium = new FileAppender<ILoggingEvent>();
            fileAppenderSelenium.setContext(loggerContext);
            fileAppenderSelenium.setName("Selenium_RemoteWebDriver");
            fileAppenderSelenium.setAppend(false);
            fileAppenderSelenium.setEncoder(encoder);
            myLoggerSelenium.addAppender(fileAppenderSelenium);
            myLoggerSelenium.setLevel(Level.DEBUG);

            initialized = true;
        }
        else
            initialized = false;
        StatusPrinter.print(loggerContext);
    }
}

```

Figure 6: LogbackFileUtils Class Part 1

```

public static void start(Class classname) {
    init(classname);
    stop();
    String fileSetter = "logs/" + classname.getSimpleName() + new SimpleDateFormat("yyyy-MM-dd--HH_mm_ss").format(new Date()) + ".log";
    fileAppender.setFile(fileSetter);
    fileAppender.start();
    if(initialized){
        fileAppenderSelenium.setFile(fileSetter);
        fileAppenderSelenium.start();
    }
}

public static void stop() {
    if (fileAppender.isStarted())
        fileAppender.stop();
    if (initialized && fileAppenderSelenium.isStarted())
        fileAppenderSelenium.stop();
}

public static boolean isAutomatedTestingSuiteClass(Class cls){
    //need to check that cls is a class which extends AutomatedTestingSuite
    return AutomatedTestingSuite.class.isAssignableFrom(cls);
}
}

```

Figure 7: LogbackFileUtils Class Part 2

5. Installation and Setup

5.1. Clone Automated Functional Testing Source Code

1. Setup Git

<https://help.github.com/articles/set-up-git>

2. Get ATF Source Code

a. Source Code Option

Git clone [git@github.com:afequ871/Web-Automation-Testing-Framework.git](https://github.com/afequ871/Web-Automation-Testing-Framework.git)

5.2. Test Structure and Setup

The test harness utilizes a test structure that organizes the build, configuration, and execution methods. This structure also defines the file organization.

As shown in Figure 8, the CMAKE and CTEST scripts are used to control the build, configuration, and execution of tests. The first step in the testing process is to build the platform specific test scripts. This is accomplished with the CMAKE-GUI (see Figure 9), a configuration and generation tool. Currently this configuration and generation step is manual but will be automated in the future. After producing the platform specific test scripts, CMAKE is used to

create and configure the AUT and then populate it with test data. After the AUT is populated with test data, each test is invoked using CTEST. Test results are then captured in log files.

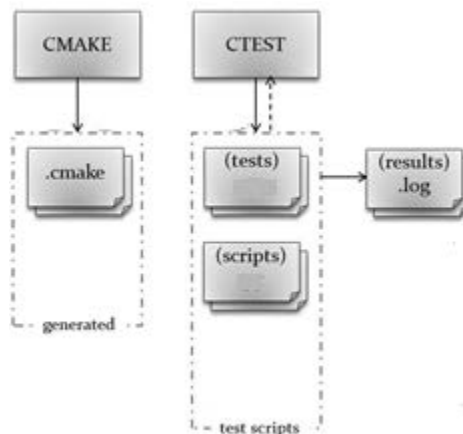


Figure 8: Build and Test Summary

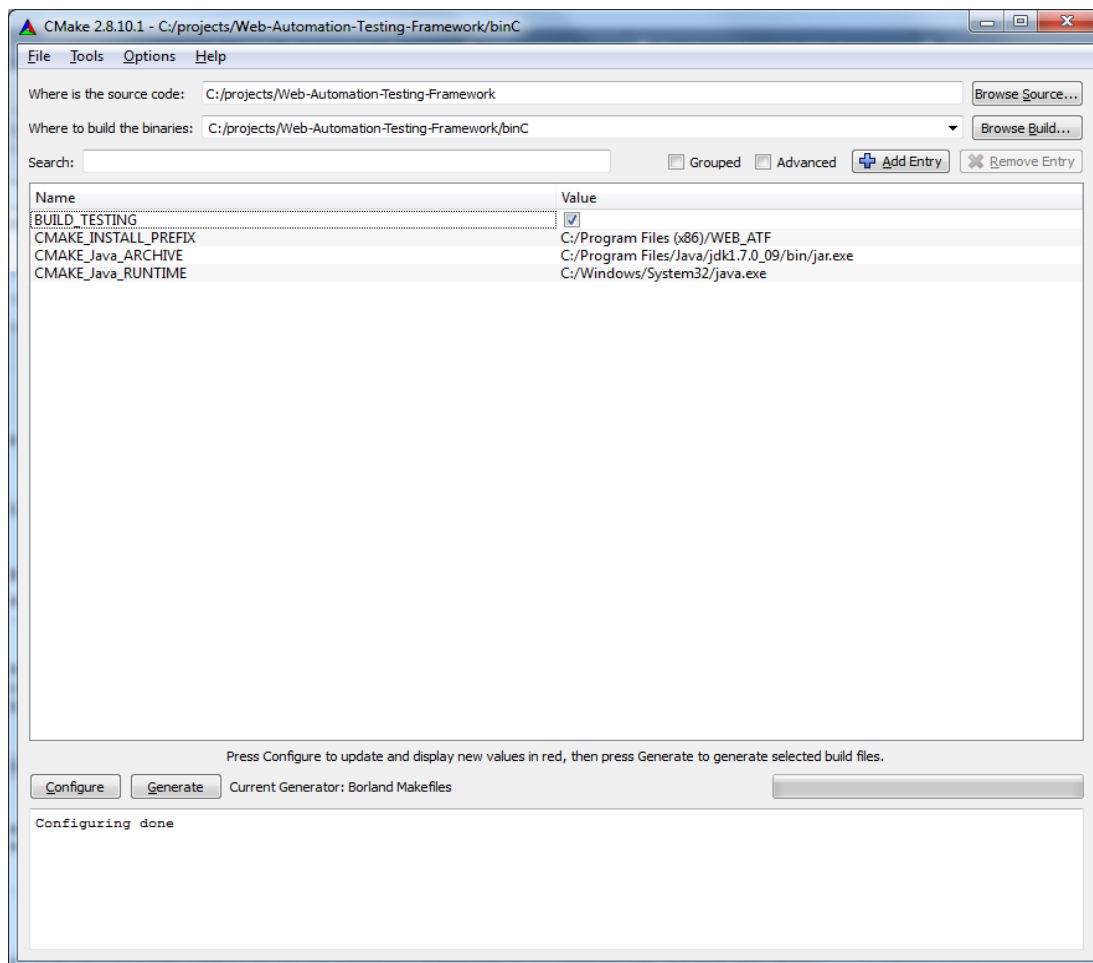


Figure 9: CMAKE-GUI Screenshot

5.3. Testing Execution

To run the automation tests perform the following steps:

- a) Run CMake-GUI to configure and generate CMake files making a /binC or /build folder
- b) From the /binC (/build) folder do the following:
 - a. CMake -P BuildProject.cmake
 - b. CTest -R WEB_ (Test)

Please note the following:

- The CMake -P BuildProject.CMake must be run before every CTest -R WEB due to the fact that after every code change the application needs to generate the appropriate binary class files.
- Test result logs are located in the /binC/Testing/execution/logs
- New test suites can be added by placing the "AllTestSuites main" .java file in the /src/org/automatedtesting/controller directory and related test classes should be placed in /src/org/automatedtesting/suites directory.