Devonfw Testing Guide
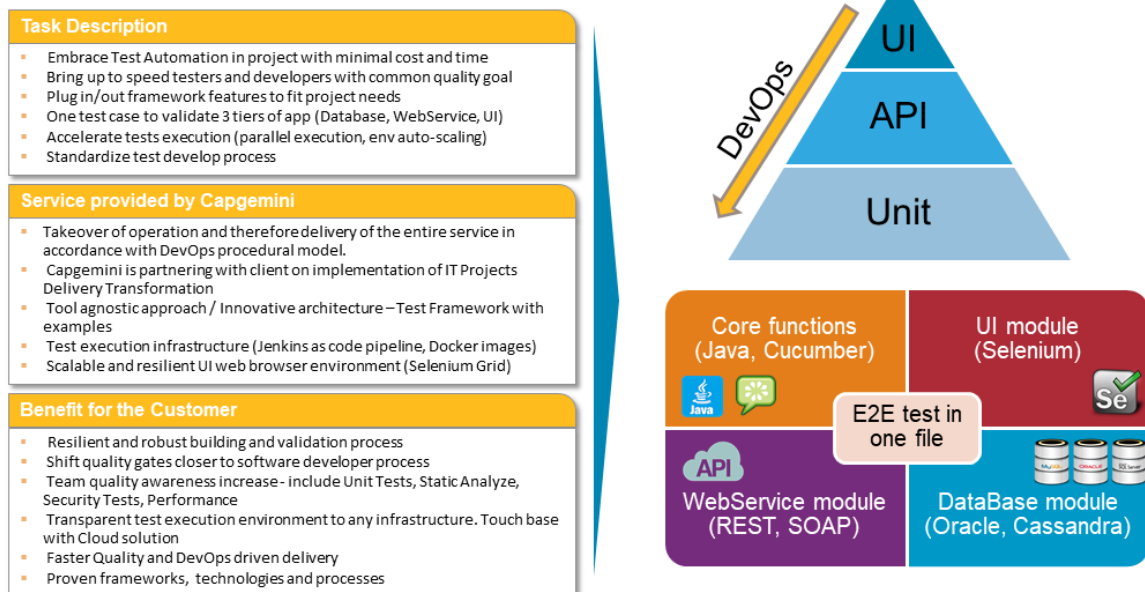
2018-02-05

Table of Contents

1. Basics

1.1 Home

1.1.1 What is E2E Allure Test Framework

End to end automation test framework written in Java.

E2E Test Framework for DevOps & Smart Automation

Task Description
- Embrace Test Automation in project with minimal cost and time
- Bring up to speed testers and developers with common quality goal
- Plug in/out framework features to fit project needs
- One test case to validate 3 tiers of app (Database, WebService, UI)
- Accelerate tests execution (parallel execution, env auto-scaling)
- Standardize test develop process

Service provided by Capgemini
- Takeover of operation and therefore delivery of the entire service in accordance with DevOps procedural model.
- Capgemini is partnering with client on implementation of IT Projects Delivery Transformation
- Tool agnostic approach / Innovative architecture – Test Framework with examples
- Test execution infrastructure (Jenkins as code pipeline, Docker images)
- Scalable and resilient UI web browser environment (Selenium Grid)

Benefit for the Customer
- Resilient and robust building and validation process
- Shift quality gates closer to software developer process
- Team quality awareness increase - include Unit Tests, Static Analyze, Security Tests, Performance
- Transparent test execution environment to any infrastructure. Touch base with Cloud solution
- Faster Quality and DevOps driven delivery
- Proven frameworks, technologies and processes

DevOps

UI
API
Unit

Core functions (Java, Cucumber)

UI module (Selenium)

E2E test in one file

WebService module (REST, SOAP)

DataBase module (Oracle, Cassandra)

E2E Test Framework modules

* - task not started, to be defined

1.1.2 Benefits to the project

Benefits to project

End to end project verification under one solution

Resilient and robust DevOps driven delivery

Team quality awareness

Shift left quality gates

Easy to enhance

1.1.3 Road map plan

1.1.4 Wiki Structure:

- How to install *Allure Test Framework*

- Allure Test Framework modules:

  - Core test module

  - Selenium test module

  - WebAPI test module

  - Security test module

  - DataBase test module

  - Mobile test module

  - Standalone test module

  - DevOps module

1.2 How to install

1.2.1 Install can be done in two scenarios:

• Out of the box install - Fast and easy

• Advanced installation - How to install all ingredients

Easy out of the box install

1. Java 1.8 JDK 64bit

   • Download and install Java download link

   • Windlows Local Environment how to set:

     • **Variable name**: JAVA_HOME | **Variable value**: c:\Where_You've_Installed_Java

     • **Variable name**: PATH | **Variable value**: %JAVA_HOME%/bin;%JAVA_HOME%\lib

- Verify in command line:

```
> java --version
```

2. Download package [Ready to use AllureTestEnvironment](#)

3. To folder C:\ unzip with [7z](#) downloaded Allure Test Framework

   Note: Please double check, place where you have unzipped Allure_Test_Framework

4. In unzipped folder (C:\Allure_Test_Framework\) run *start-eclipse.bat*

5. . Update project structure (*ALT + F5*)

Manual step by step install

1. Java 1.8 JDK 64bit

   - Download and install Java download link

Java SE Development Kit 8u131

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement   ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.87 MB	jdk-8u131-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.81 MB	jdk-8u131-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.66 MB	jdk-8u131-linux-i586.rpm
Linux x86	179.39 MB	jdk-8u131-linux-i586.tar.gz
Linux x64	162.11 MB	jdk-8u131-linux-x64.rpm
Linux x64	176.95 MB	jdk-8u131-linux-x64.tar.gz
Mac OS X	226.57 MB	jdk-8u131-macosx-x64.dmg
Solaris SPARC 64-bit	139.79 MB	jdk-8u131-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.13 MB	jdk-8u131-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u131-solaris-x64.tar.Z
Solaris x64	96.96 MB	jdk-8u131-solaris-x64.tar.gz
Windows x86	191.22 MB	jdk-8u131-windows-i586.exe
Windows x64	198.03 MB	jdk-8u131-windows-x64.exe

- Windows Local Environment how to set:

    - **Variable name**: JAVA_HOME | **Variable value**: c:\Where_You've_Installed_Java

    - **Variable name**: PATH | **Variable value**: %JAVA_HOME%/bin;%JAVA_HOME%\lib

- Verify in command line:

```
> java --version
```

2. Maven 3.5

- Download   Maven   http://www-eu.apache.org/dist/maven/maven-3/3.5.0/binaries/apache-maven-3.5.0-bin.zip

- Unzip, to C:\maven

- Windows Local Environment

  - **Variable name**: M2_HOME | **Variable value**: c:\maven

  - **Variable name**: PATH | **Variable value**: %M2_HOME%\bin
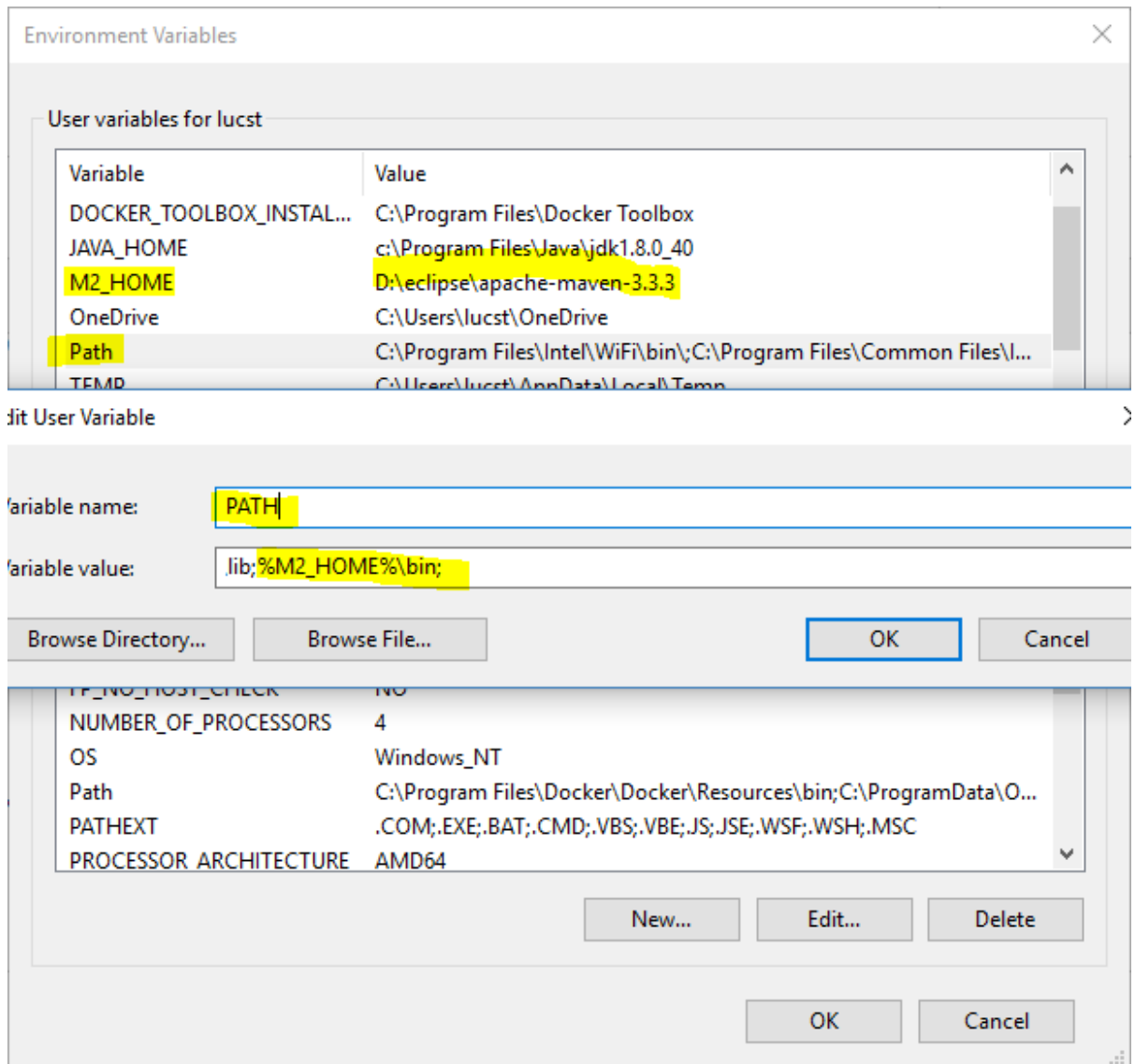
- Verify in command line:

```
> mvn --version
```

3. Eclipse IDE

    - Download and unzip [Eclipse](#)

4. Download Allure Test Framework [source code](#)

5. Import projects in Eclipse

    - Import:

- Projects from folders:

- Open already created projects

- Update project structure - *ALT + F5*

2. Modules

2.1 Core Test Module

2.1.1 What is Core Test Module

Core functionality ingredients

- Test report with logs and/or screenshots
- Test groups/tags
- Data Driven (inside test case, external file)
- Test case parallel execution
- BDD - Gherkin – Cucumber approach
- Run on independent OperatingSystem
- Externalize test environment (DEV, QA, PROD)

2.1.2 Core Test Module Functions

- Test reports with logs and/or screenshots

- Test groups/tags

- Data driven approach

- Test case parallel execution

- BDD - Gherkin - Cucumber approach

- Run on independent Operating Systems

- Externalize test environment (DEV, QA, SIT, PROD)

[[core-test-module_how-to-start?]] == How to start? Read: Framework Test Class

Test Class

Overview

The following image gives a general overview of the test class "lifecycle".

Before Class
Before Test
Test
AfterTest

Before Test
Test
After Test
After Class

More information on the methods and annotations used in this image can be found in the following chapter.

Methods and annotations

The actual tests that will be executed are loacted in so called *Test Classes*. Starting a new project, a new package should be created.

Source folder: allure-app-under-test/src/test/java

Name: com.example.selenium.tests.tests.YOUR_PROJECT

Test classes have to extend the **BaseTest** class.

```java
public class DemoTest extends BaseTest {

  @Override
  public void setUp() {

  }

  @Override
  public void tearDown() {

  }
}
```

BasePage method: setUp

This method will be executed before the test. It allows objects to be instantiated, e.g. Page objects.

```java
@Override
public void setUp() {
  someTestPage = new SomeTestPage();
}
```

BasePage method: tearDown

The tearDown methods executes after the test. It allows the clean up of the testing environment.

Annotations

The *@Test* annotation indicates that the following method is a test method.

Additionally, there are two annotations that can help preparing and disassembling the test class: *@BeforeClass* and *@AfterClass*.

@BeforeClass will execute the following method once at the beginning, before running any test method. Compared to the *setUp()* method provided by the BaseTest class, this annotation will only run once, instead of before every single test method. The advantage here: Things like login can be set up in *@BeforeClass*, as they can oftentimes be very time consuming. Loggin in on a webapplication once and afterwards running all the test methods is more efficient than loggin in before every test method, even though they are being executed on the same page.

@AfterClass will execute after the last test method. Just like *@BeforeClass* this method will only run once, in contrary to the *tearDown()* method.

Initialize a new test method by using the *@Test* annotation.

```
@Test
public void willResultBeShown() {

}
```

This method will interact with a page object in order to test it.

Sample setup

```
@BeforeClass
public static void setUpBeforeClass() throws Exception {
 BFLogger.logInfo("[Step1] Login as Account Administrator");
}

@AfterClass
public static void tearDownAfterClass() throws Exception {
 BFLogger.logInfo("[Step4] Logout");
}

@Override
public void setUp() {
 BFLogger.logInfo("Open home page before each test");
}

@Override
public void tearDown() {
 BFLogger.logInfo("Clean all data updated while executing each test");
}

@Test
public void test1() {
 BFLogger.logInfo("[Step2] Filter by \"Creation Date\" - Descending");
 BFLogger.logInfo("[Step3] Set $1 for first 10 Users in column \"Invoice to pay\"");

}

@Test
public void test2() {
 BFLogger.logInfo("[Step2] Filter by \"Invoice to pay\" - Ascending");
 BFLogger.logInfo("[Step3] Set $100 for first 10 Users in column \"Invoice to pay\"");
```

```
}
```

2.2 Selenium Test Module

2.2.1 What is Allure E2E Selenium Test Module

2.2.2 Selenium Structure

- What is Selenium

- What is WebDriver

- What is Page Object Model/Pattern

- List of web elements (Button, Dropdown, Checkbox, Alert Popup, etc.)

2.2.3 Framework Features

- Construction of Framework Page Class

  - Every Page class must extend BasePage

  - What is isLoaded(), load() and pageTitle() for

  - How to create selector variable - 'private static final By ButtonOkSelector = By.Css(…)'

  - How to prepare everlasting selector - documentation

- Method/action naming convention - [documentation](#)

- Why we should use findDynamicElement() and findElementQuietly() instead of classic Selenium findElement

- List of well-rounded groups of user friendly actions (ElementButton, ElementCheckbox, ElementInput, etc.)

- Verification points of well-defined Page classes and Test classes - [documentation](#)

- [Run on different browsers: Chrome, Firefox, IE, Safari, Edge](#)

- [Run with full range of resolution (mobile and desktop): Testing Response Design Webpage](#)

[[selenium-test-module_how-to-start?]] == How to start? Read: [My first Selenium Test](#)

Sample Walkthrough

This page will walk you through the process of creating a test case. We'll create a very simple test for the Google search engine.

Test Procedure

We would like to open the Google search engine, enter some search query and afterwards submit the form. We hope to see some results being listed, otherwise the test will fail. Summarized, the testing process would look like this.

1. Open google.com

2. Enter the string "Test" into the searchbox

3. Submit the form

4. Get the results and check if the result list is empty

Creating new packages

We will need two new packages, one for the new page classes, the other one for our test classes.

Creating package for test classes

Open Eclipse, use the "Project Explorer" on the left to navigate to

allure-app-under-test →ₐ src/test/java →ₐ com.example →ₐ selenium.tests →ₐ tests

Right click on "tests", click on "New" →ₐ New Package. We'll the new package "com.example.selenium.tests.googleSearch".

Creating package for page classes

Navigate to

allure-app-under-test $\rightarrow$ src/main/java $\rightarrow$ com.example $\rightarrow$ selenium $\rightarrow$ pages

Right click on "pages", click on "New" $\rightarrow$ New Package. The new package will be called "com.example.selenium.pages.googleSearch".

Creating the test class

The test class will contain the entire testing-routine. At first, we'll create a new class inside of our newly created "googleSearch" package (under src/test/java) and call it "GoogleSearchTest".

As "GoogleSearchTest" is a test class, it has to extend the *BaseTest* class. You may have to import some required packages and afterwards include a few required methods.

```java
public class GoogleSearchTest extends BaseTest {

@Override
public void setUp() {

}

@Override
public void tearDown() {

}
```

```
}
```

Now, we'll need a new Page object, which will represent the Google Search page. The page class will be named "GoogleSearchPage".

```java
private GoogleSearchPage googleSearchPage;

@Override
public void setUp() {
 googleSearchPage = new GoogleSearchPage();
}
```

Creating the GoogleSearchPage class

We created a new field for the GoogleSearchPage class and instantiated an object in the *setUp()* method. As this class doesn't exist yet, we'll have to create it inside of the googleSearch page class package.

We extend the GoogleSearchPage class with *BasePage*, import all necessary packages and include all required methods.

```java
public class GoogleSearchPage extends BasePage {

@Override
public boolean isLoaded() {
 return false;
}

@Override
public void load() {

}
```

```
 @Override
 public String pageTitle() {
  return "";
 }


}
```

As this page class represents the Google homepage, we have to set up selectors for web elements required in our test case. In our example we have to create a selector for the search bar which we'll interact with. The selector will be implemented as a field.

```
 private static final By selectorGoogleSearchInput = By.css(#lst-ib);
```

The input field's id *#lst-ib* was found by using the developer console in Google Chrome.

This selector can be used to create a WebElement object of said search bar. Therefore, we'll create a new method and call it "enterGoogleSearchInput".

```
 public GoogleResultPage enterGoogleSearchInput(String searchText) {
  WebElement googleSearchInput = getDriver().findDynamicElement(selectorGoogleSearchInput);
  googleSearchInput.sendKeys(searchText);
  googleSearchInput.submit();

  return new GoogleResultPage();
 }
```

As you can see, we return another page object that wasn't yet created. This step is required, as the results that we would like to check are on another Google Page. This means we'll have to create another page class, which will be shown later.

Finally, the empty methods inherited from the BasePage class have to be filled:

```
 @Override
 public boolean isLoaded() {
  if(getDriver().getTitle().equals(pageTitle())) {
   return true;
  }
  return false;
 }

 @Override
 public void load() {
  getDriver().get("http://google.com");
 }

 @Override
 public String pageTitle() {
  return "Google";
 }
```

The method *isLoaded()* checks if the page was loaded by comparing the actual title with the expected title provided by the method *pageTitle()*. The *load()* method simply loads a given URL, in this case *http://google.com*.

The completion of these methods finalizes our *GoogleSearchPage* class. Now we still have to create the *GoogleResultPage* class mentioned before. This page will deal with the elements on the Google search result page.

Creating the GoogleResultPage class

By right-clicking on the "pages" package, we'll navigate to "new" → "Class" to create a new class.
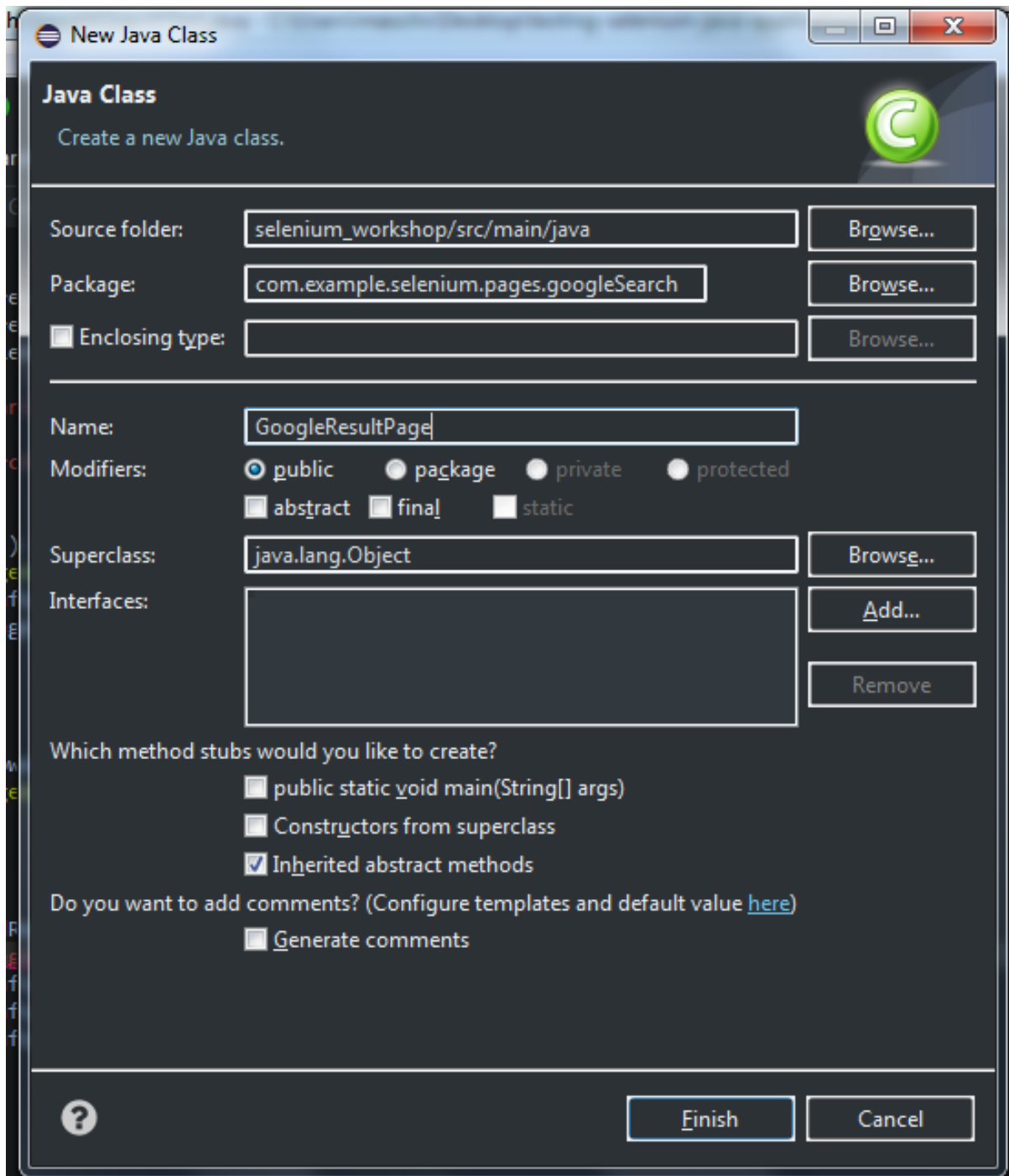
The *GoogleResultPage* class also has to extend *BasePage* and include all required methods. Next, a new selector for the result list will be created. By using the result list, we can finally check if the result count is bigger than zero and the search request therefore was successful.

```java
private static final By selectorResultList = By.cssSelector("#res");
```

We'll use this selector inside a new *getter*-method, which will return all ListElements.

```java
public ListElements getResultList() {
 return getDriver().elementList(selectorResultList);
}
```

This method will allow the testcase to simply get the result list and afterwards check if the list is empty or not.

Finally, we have to complete all inherited methods.

```java
@Override
public boolean isLoaded() {
 getDriver().waitForPageLoaded();
 if(getDriver().getCurrentUrl().contains("search")) {
  return true;
 }
 return false;
}

@Override
public void load() {
 BFLogger.logError("Google result page was not loaded.");
}

@Override
public String getTitle() {
 return "";
}
```

The method *isLoaded()* differs from the same method in *GoogleSearchPage*, because this site is being loaded as a result from a previous action. That's why we'll have to use the method *getDriver().waitForPageLoaded()* to be certain, that the page was loaded completely. Afterwards we check if the current URL contains the term "search", as it only occurs on the result page. This way we can check if we're on the right page.

Another result of this page being loaded by another object, we don't have to load any specific URL. We just add a BFLogger instance to print an error message if the page was not successfully loaded.

As we don't use the *getTitle()* method we simply return an empty String.

Finally, all required page classes are complete and we can finalize the test class.

Finalizing the test class

At this point, our GoogleSearchTest class looks like this:

```java
public class GoogleSearchTest {

 private GoogleSearchPage googleSearchPage;


 @Override
 public void setUp() {
  googleSearchPage = new GoogleSearchPage();
 }

 @Override
 public void tearDown() {

 }
}
```

Next up, we'll create the test method, let's call it *shouldResultReturn()*.

```java
@Test
public void shouldResultReturn() {
 GoogleResultPage googleResultPage = googleSearchPage.enterGoogleSearchInput("Test");
 ListElements results = googleResultPage.getResultList();
```

```
assertTrue("Number of results equals 0", results.getSize() > 0);
}
```

Code explanation: At first, we will run the *enterGoogleSearchInput()* method on the GoogleSearchPage with the parameter "Test" to search for this exact string on Google. As this method returns a GoogleResultPage object, we will store this in the local variable *googleResultPage*. Afterwards, we get the result list by utilizing the getter method that we created before. Finally, we create an assertion: We expect the list size to be bigger than zero, meaning that the google search query was successful as we received results. If this assertion is wrong, a message will be printed out, stating that the number of results equals zero.

We can run the test by right clicking on the test method → Run as → JUnit test.

After starting the test, you'll notice a browser window opening, resizing to given dimensions, opening Google, entering the query "Test" and submitting the form. After completing the test, you'll see the test results on the right side of Eclipse. A green color indicator means that the test was successful, red means the test failed.

This walkthrough should've provided you with the basic understanding on how the framework can be used to create test cases.

WebAPI Test Module

Security Test Module

[[security-test-module_what-is-security?]] == What is Security?

Application Security is concerned with **Integrity**, **Availability** and **Confidentiality** of data processed, stored and transferred by the application.

Application Security is a cross-cutting concern which touches every aspect of the Software Development Lifecycle. You can introduce some SQL injection flaws in your application and make it exploitable, but you can also expose your secrets due to poor secret management process (which will have nothing to do with code itself), and fail as well.

Because of this, and many other reasons, not every aspect of security can be automatically verified. Manual tests and audits will be still needed. Nevertheless, every security requirement which are automatically verified, will prevent code degeneration and misconfiguration in a continuous manner.

[[security-test-module_how-to-test-security?]] == How to test Security?

Security tests can be performed in many different ways like:

- **Static Code Analysis** - improves the security by (usually) automated code review. Good way to search after vulnerabilities, which are 'obvious' on the code level (like e.g. SQL injection). The downside is that the professional tools to perform such scans are very expensive and still produce many false positives.

- **Dynamic Code Analysis** - tests are run against a working environment. Good way to search after vulnerabilities, which require all client- and server-side components to be present and running (like e.g. Cross-Site Scripting). Tests are performed in a semi-automated manner and require a proxy tool (like e.g. OWASP ZAP)

- **Unit tests** - self written and maintained tests. They work usually on the HTTP/REST level (as this defines the trust boundary between the client and the server) and run against a working environment. Unit tests are best suited to verify requirements which involve business knowledge of the system or which assure secure configuration on the HTTP level.

In the current release of the Security Module the main focus will be **Unit Tests**.

Although the most common choice of environment for security tests to run on will be **integration** (as the environment offers the right stability and should mirror the production closely), it is not uncommon for some security tests to run on production as well. This is done for e.g. TLS configuration testing to ensure proper configuration of the most relevant environment in a continuous manner.

Scope definition

DataBase Test Module

Mobile Test Module

Standalone Test Module

2.3 DevOps Module

2.3.1 How we see DevOps

DevOps consists of a mix of three key components in a technical project: * People skills and mind set * Processes * Tools

In **E2E Allure Test Framework** we would like to address majority of these components.

2.3.2 QA Team Goal

As QA engineers we always take a lot of care about product code quality.

Therefore, we also have to understand, **test case is also a code which has to be validated** against quality gates. As a result, we must **test our developed test case** just as it was done during standard Software Delivery Life Cycle.

2.3.3 Well rounded test case production process

- How we define top notch test cases develop process in **E2E Allure Test Framework**
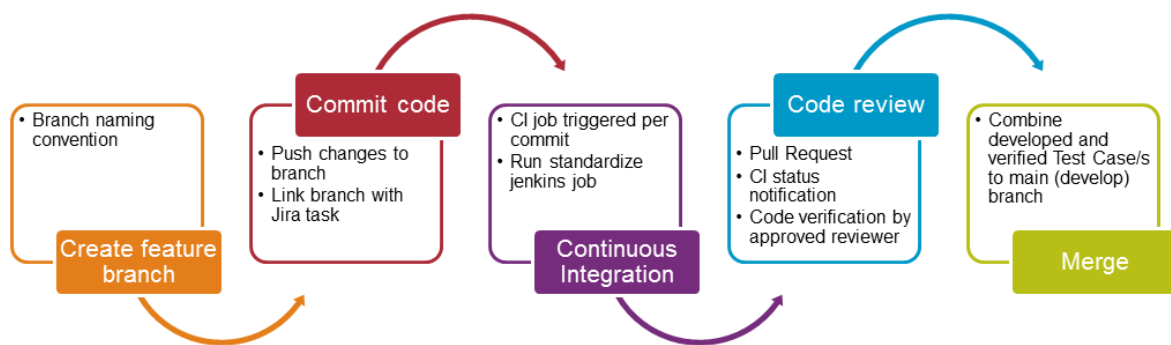
[[devops-module_continuous-integration-(ci)-and-continuous-delivery-(cd)]] == Continuous Integration (CI) and Continuous Delivery (CD)

- [Continuous Integration (CI)](#) - procedure where quality gates validate test case creation process

- [Continuous Delivery (CD)](#) - procedure where we include as smoke/regression/security created test cases, validated against CI
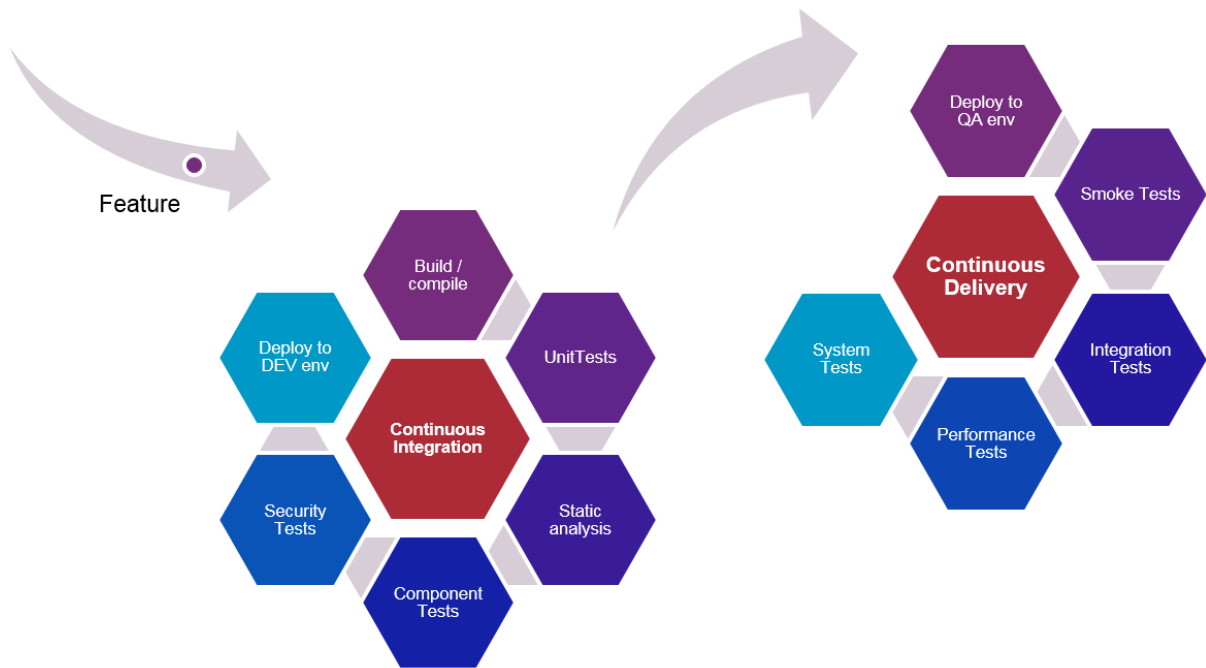
2.3.4 What will you receive in this DevOps module
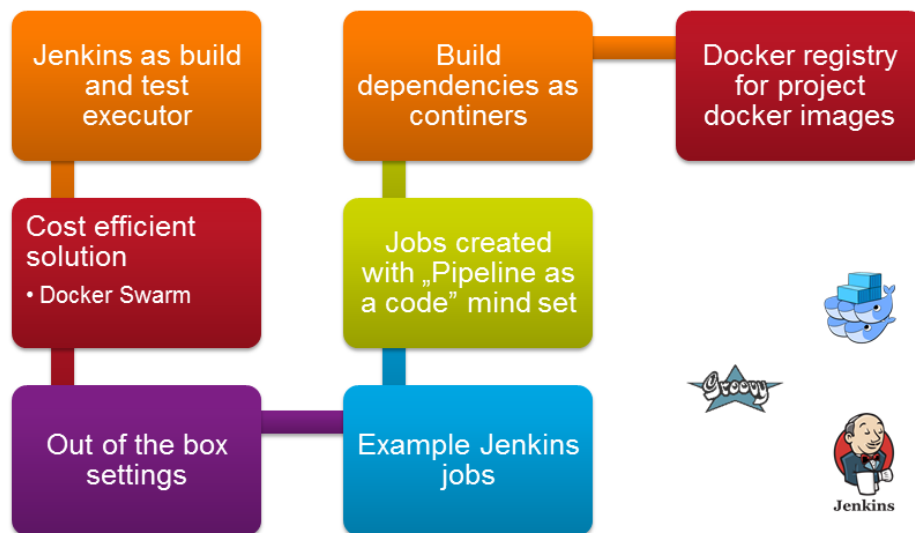
2.3.5 What will you gain with our DevOps module

The CI procedure has been divided into transparent modules. This solution makes configuration and maintenance very easy because you can manage versions and customize the configuration

independently for each module. A separate security module ensures protection of your credentials and assigned access roles regardless of changes in other modules.

Superior Continuous Integration ingredients
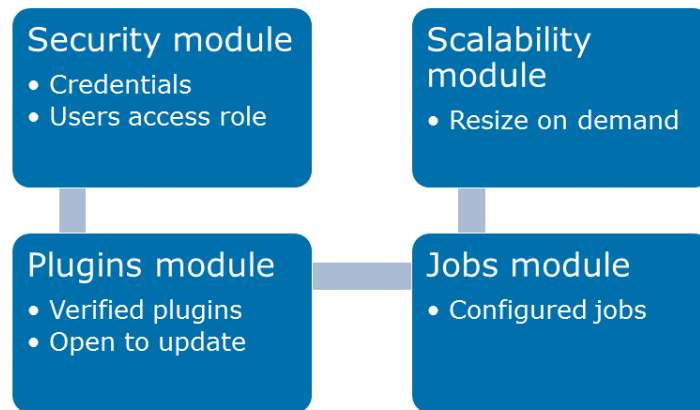
Your CI process will be matched to the current project. You can easily go back to the previous configuration, test a new one or move a selected one to other projects.

Setup your own proven CI modules
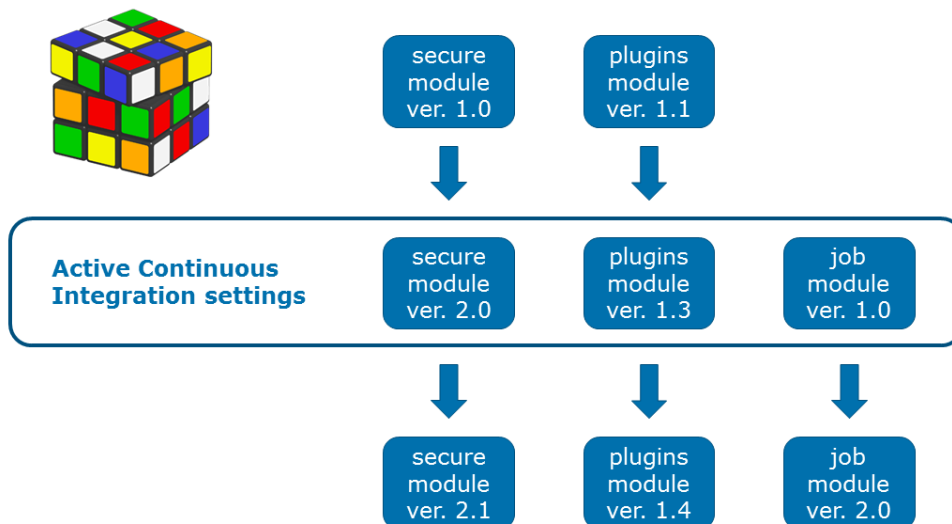
DevOps module supports delivery model in which executors are made available to the user as needed. It has advantages such as: * Save computing resources * Eliminate guessing on your infrastructure capacity needs * Stop spending time on running and maintaining additional executors

Classic executor architecture for Continuous Integration
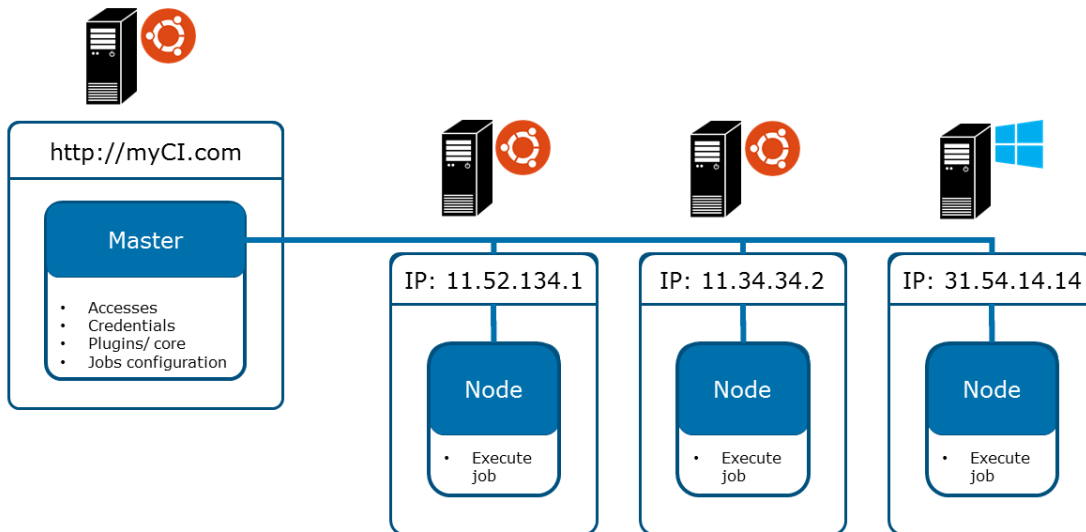
Innovative executor architecture for Continuous Integration
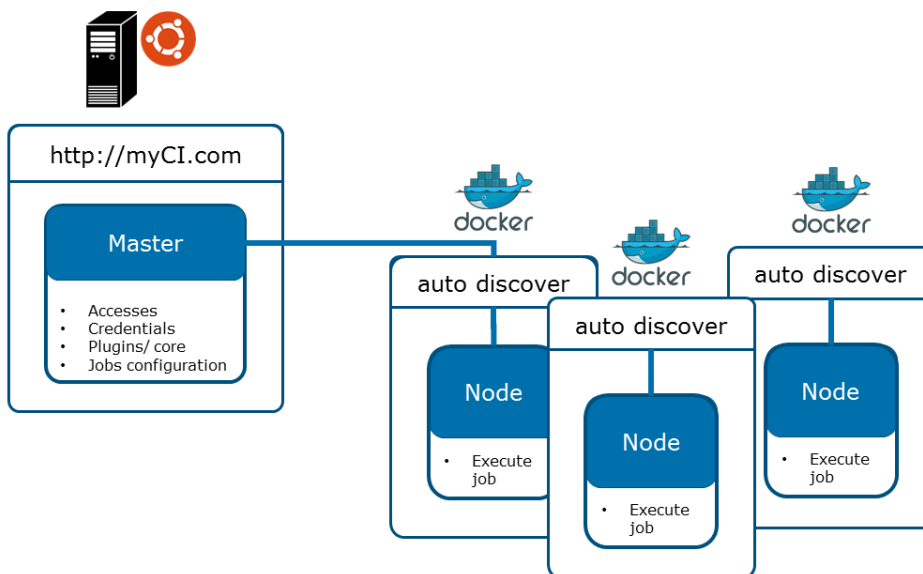
Benefits

Modularity
Build your own CI

Secure
Restricted access to credentials

Autoscaling and service discovery
Take as you need

2.3.6 How to build this DevOps module

If you want to install the module, please click the link below. Installation should not take more than a few minutes * DevOps module installation

Once you have implemented the module, you can learn more about:

- Building jobs & Running builds

- Docker commands

Continuous Integration

Embrace quality with Continuous Integration while you produce test case/s.

Overview

There are two ways to set up your Continuous Integration environment:

1. Create a Jenkins instance from scratch (e.g. by using the Jenkins Docker image)

   Using a clean Jenkins instance requires the installation of additional plugins. The plugins required and their versions can be found on this page.

2. Use thre pre-configured custom Docker image provided by us

   No more additional configurations is required (but optional) using this custom Docker image. Additionally, this Jenkins setup allows to be dynamically scaled across multiple machines and even the cloud (AWS, Azure, Google Cloud etc.).

Jenkins Overview
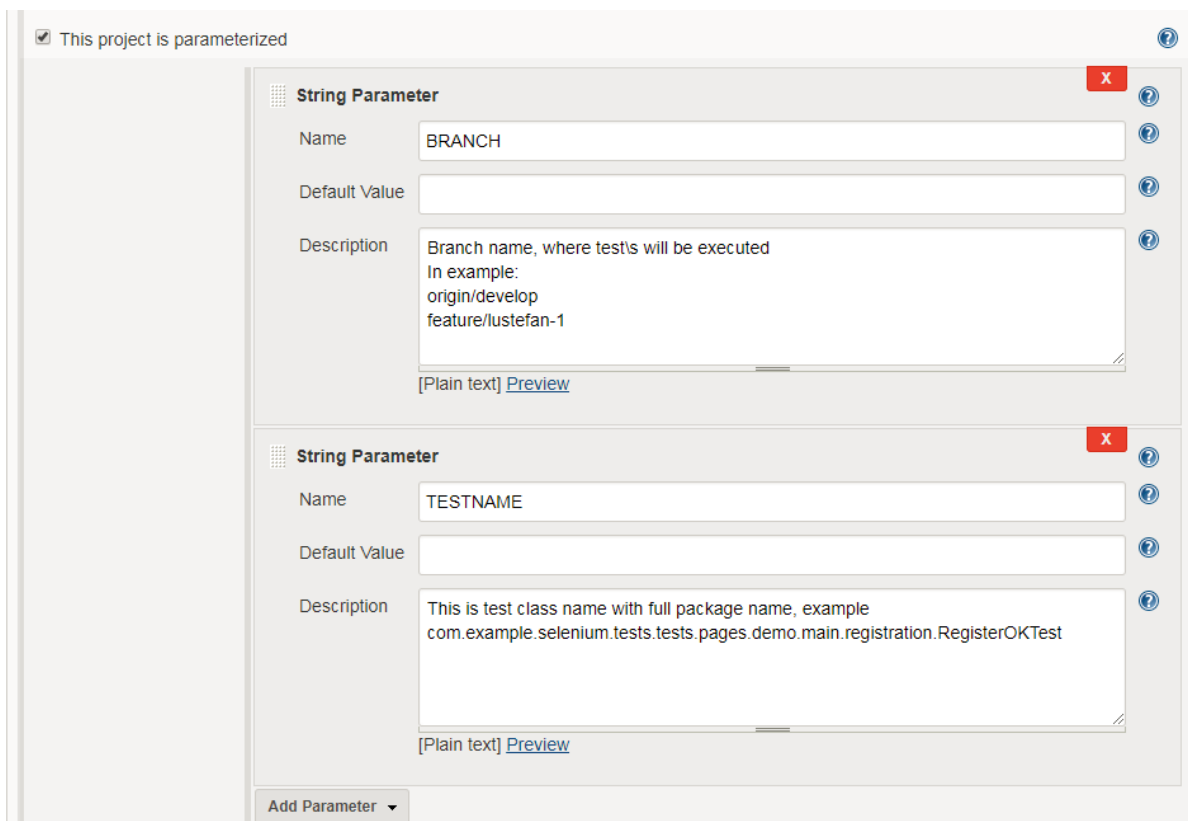
Jenkins is an Open Source Continuous Integration Tool. It allows the user to create automated build jobs which will run remotely on so called *Jenkins Slaves*. A build job can be triggered by several events, for example on new pull request on specified repositories or timed (e.g. at midnight).

Jenkins Configuration

Tests created by using the testing framework can be easily implemented on a Jenkins instance. The following chapter will describe such a job configuration. If you're running your own Jenkins instance you may have to install additional plugins listed on the page Jenkins Plugins for a trouble-free integration of your tests.

Initial Configuration
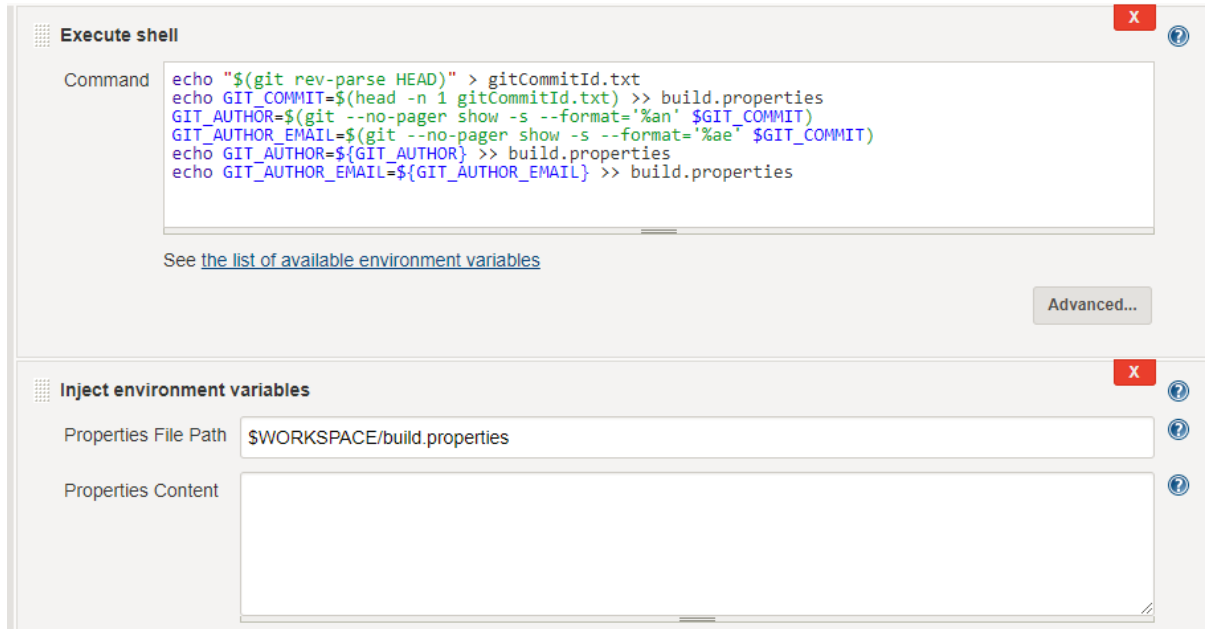
The test job is configured as a so-called *parametrized* job. This means, after starting the job, parameters can be specified, which will then be used in the build process. In this case, *branch* and *testname* will be expected when starting the job. These parameters specify which branch in the code repository should be checked out (possibly feature branch) and the name of the test, that should be executed.

Build Process Configuration

- The first step inside the build process configuration is to get the author of the commit that was made. The mail will be extracted and gets stored in a file called *build.properties*. This way, the author can be notified if the build fails.

- Next up, Maven will be used to check if the code can be compiled, without running any tests.

After making sure that the code can be compiled, the actual tests will be executed.

+ image::images/jenkins-build-3.png["Starting the actual tests", width="450", link="images/jenkins-build-3.png"]
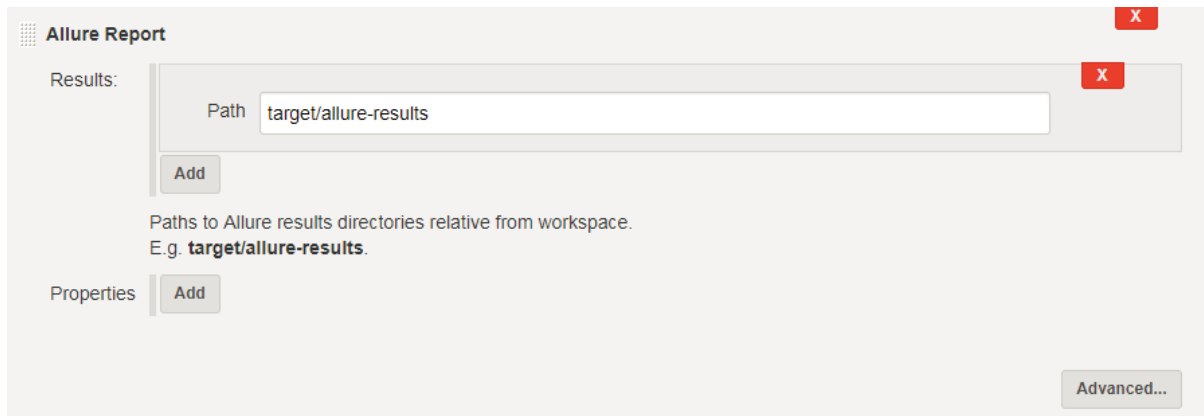
- Finally, reports will be generated.

Post Build Configuration

- At first, the results will be imported to the [Allure System](#)

Allure Report

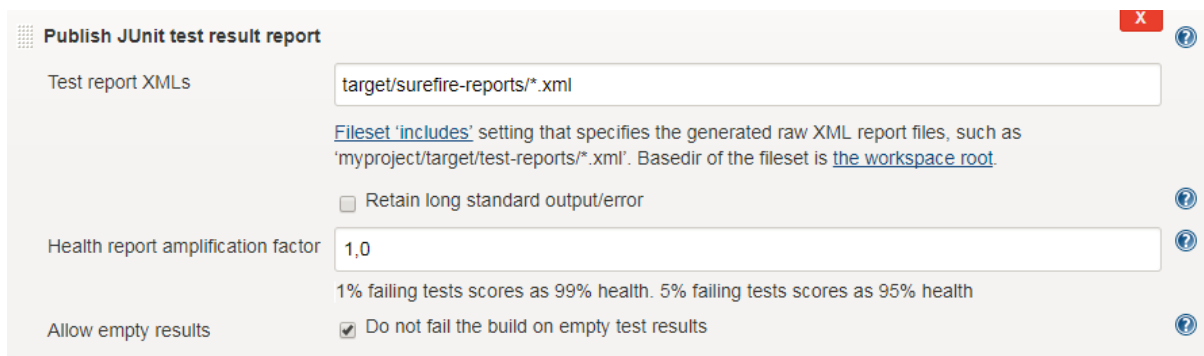Results:

Path   target/allure-results

Add

Paths to Allure results directories relative from workspace.
E.g. **target/allure-results**.

Properties   Add

Advanced...

- JUnit test results will be reported as well. Using this step, the test result trend graph will be displayed on the Jenkins job overview.

Publish JUnit test result report

Test report XMLs   target/surefire-reports/*.xml

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is the workspace root.
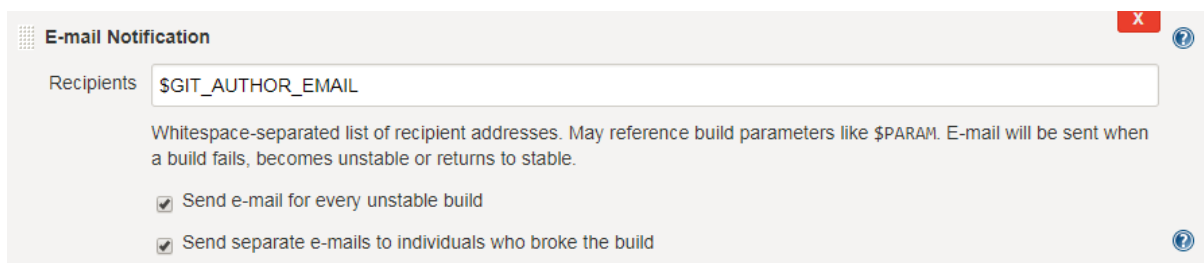
☐ Retain long standard output/error

Health report amplification factor   1,0

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

Allow empty results   ☑ Do not fail the build on empty test results

- Finally, an E-Mail will be sent to the previously extracted author of the commit.

E-mail Notification

Recipients $GIT_AUTHOR_EMAIL

Whitespace-separated list of recipient addresses. May reference build parameters like $PARAM. E-mail will be sent when a build fails, becomes unstable or returns to stable.

☑ Send e-mail for every unstable build

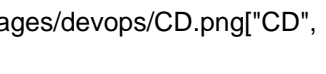☑ Send separate e-mails to individuals who broke the build

Using the Pre-Configured Custom Docker Image

If you are starting a new Jenkins instance for your tests, we'd suggest to use the pre-configured Docker image. This image already contains all configurations and additional features.

The configurations that are made are e.g. Plugins and Pre-Installed job setup samples. This way, you don't have to set up the entire CI-Environment from ground up.

The additional features from this docker image allow the dynamic creation and deletion of Jenkins slaves, by creating Docker containers. Also, Cloud Solutions can be implemented to allow wide-spread load balancing.

Continuous Delivery

Include quality with Continuous Delivery during product release. image::images/devops/CD.png["CD", width="450", link="image/devops/CD.png"]

Overview

CD from Jenkins point of view does not change a lot from Continuous Integration one.

Jenkins Overview

For Jenkins CD setup please use the same Jenkins settings as for CI link The only difference is:

• What type of test you we will execute. Before we have been picking test case(s), however now we will choose test suite(s)

• Who will will trigger given Smoke/Integration/Performance job

• What is the name of official branch. This branch ought to use be used always in every CD execution. It will be either **master**, or **develop**.

Jenkins for Smoke Tests

In point where we input test name - $TESTNAME (link), please input test suite which merge by tags -(link) how to all test cases need to run only smoke tests.

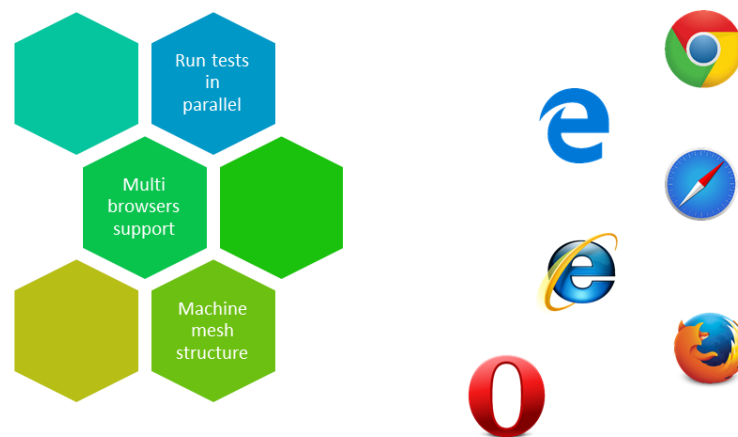**Jenkins for Performance Tests**

Under construction - added when WebAPI module is included.

Selenium Grid

What is Selenium Grid

## Selenium Grid – where we use it

Allows to run web/mobile browsers test cases to full fill bedrock factors, such as: * Independence infrastructure , similar to end users * Scalable infrastructure (\~50 simultaneous sessions at once) * Huge variety of web browsers (from mobile to desktop) * Continuous Integration and Continuous Delivery process * Support multi type programming languages (java, javascript, python, …)

In daily base test case execution/develop test automation engineer uses his local environments. However created browser test case must be able to run on any other infrastructure. Therefore Selenium Grid allows us this portability.

Selenium Grid Structure

Selenium Grid - structure
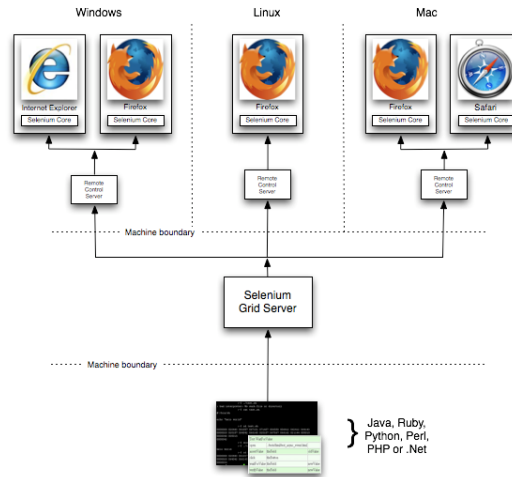
Full documentation for Selenium Grid can be found here and here.

As vanila flavor Selenium Grid is based on two, not to much complicated, ingredients:

1. **Selenium Hub** - as a one machine, accepts connections to grid from test cases executors. Moreover it plays managerial role in connection to/from Selenium Nodes

2. **Selenium Node** - from one to many machines, where on each machine is installed browser used during test case execution.

How to setup

There are two solutions:

- Classic, static solution - link

- Cloud, scalable solution - link

Pros and cons of both solutions:

Single Selenium Grid - comparison

	Cost to setup (20 instances/browsers)	Cost to scale up (down) new 20 instances/browsers	Team responsibility	Resilient and robust	Portability
Classic solution	• 200 $ /month (VMs)	• + 20 $ /month (VM) + 4 hour SeleniumGrid specialist	• Works as centralized solution used across all Test departments	• No replication. Manual actions needed to recover	• Manual infrastructure setup, for each browser (Chrome, Firefox, IE, Safari, Edge)
Cloud solution	• 10 $ /month (VMs)	• + 10 $ /month (VMs) + 0.25 hour junior team member	• Works as centralized per department or decentralized solution per team	• Auto recovery. Balance number of active instances/ browsers through swarm of active VMs	• Auto deploy script for Chrome and Firefox. Open interface to add manually IE, Safari, Edge

*) Classic solution – selenium grid run as a Java standalone application
*) Cloud solution – selenium grid run as a Docker container

VM – Virtual Machine

[[selenium-grid_how-to-use-it-with-**e2e-allure-test-frameworks**]] == How to use it with **E2E Allure Test Frameworks**

Run this command either in Eclipse or in Jenkins:

```
> mvn test -Dtest=com.capgemini.ntc.selenium.tests.samples.resolutions.ResolutionTest -
DseleniumGrid="http://10.40.232.61:4444/wd/hub" -Dos=LINUX -Dbrowser=chrome
```

As a result of this command:

• *-Dtest=com.capgemini.ntc.selenium.features.samples.resolutions.ResolutionTest* - name of test case to execute

• *-DseleniumGrid="http://10.40.232.61:4444/wd/hub"* - IP address of Selenium Hub

• *-Dos=LINUX* - what operating system must taken during test case execution

  • *-Dbrowser=chrome* - what type of browser will be used during test case execution