**Serenity BDD:**

* Is an open source ***library*** that aims to make a idea of ***living document***.
* **Helps** to ***write*** ***cleaner*** and more ***maintainable automated acceptance*** and ***regression test***.
* It also ***uses test results*** to***produce illustrated***, ***narrative reports*** that ***document*** and ***describe*** ***what the application does*** and ***how it works***.
* It not only tells ***what tests have been executed*** but also ***what requirements*** have been ***tested***.
* It ***provides*** strong ***support*** for ***automated web tests*** using Selenium and also works effectively for ***non-web tests*** that exercise ***web services*** or ***call application code directly***.
* It ***works*** with ***BDD tools*** like ***Cucumber*** or ***JBehave*** or simple use ***Junit***.
* It also ***integrates*** with ***requirements stored*** in an ***external source*** (such as ***JIRA*** or ***other test management tools***) or simply ***directory- based approach*** to ***organize requirements***.

Serenity BDD is ***commonly used*** for ***automated Regression Tests***. Whereas ***BDD acceptance tests*** are **defined very early on** in the ***piece***, ***before developments starts***. ***Regression Tests*** ***involve an existing system***. Other than that, the steps involved in defining and automating the tests very similar.

**What serenity BDD does?**

* Makes it easy to write, execute and report on automated acceptance tests in terms like. (BA, Dev, QA can relate to).
* Structures automated acceptance tests into steps and sub-steps and this makes the tests cleaner, more flexible and ease to maintain.
* When the tests are executed, Serenity produces illustrated, narrative-style reports.

**Detailed description of *aggregation reports*:**

Serenity BDD ***aggregation report*** can be ***organized by using features, stories, steps, scenarios tests***.

When we use different frameworks, with Serenity BDD it is possible that the same things will have different definitions.

Example:

**Scenario** in JBehave/cucumber is same as **test** in Junit.

**Examples** in JBehave/cucumber is same as **test data** in Junit.

[ **test** is a synonym of **acceptance criteria**]

The aggregate report **contains test results** of **all executed scenarios**, and consists of the next tab:

* **Overall Test Results**: **General info** about **provided features/ stories** in the test and statistic of **passed/ignored/skipped/failed test**.
* **Requirements**: Detailed info based on **Features, Stories** and **Acceptance Criteria**.
* **Features**: Summary table of **all Features**.
* **Stories:** Summary table with **statistics of stories**.

[ target/site/serenity directory - - > the reports will be produced and placed]

\*\*\* Serenity aggregate report are generated from JSON tests results produced when we run the serenity BDD tests.

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

**Serenity with Cucumber:**

***Cucumber*** is a popular BDD ***test automation tool***.

***Cucumber-JVM*** is the ***Java implementation*** of ***Cucumber***.

In Cucumber, we express acceptance criteria in a natural, human-readable form. **Given-When-Then**

This ***format*** is known as ***Gherkin***, and is widely used in Cucumber and other Cucumber-based BDD tools such as SpecFlow( for .NET) and Behave ( for Python).

Gherkin is a flexible, highly readable format that can be written collaboratively with product owners.

The loosely structured Given-When-Then format helps people to focus on what they are trying to achieve, and how they will know when they get it.

Sometimes tables can be used to summarize several different examples of same scenario. In Gherkin, we use example tables to do this.

Writing executable specifications

In Cucumber, scenarios are stored in a Feature file, which contain an overall description of a feature as well as number of scenarios.

Feature files are organized in resources/features directory.

(can be located anywhere but good idea to have in one location).

Organizing the feature files in sub-directories that reflect the higher-level requirements or have feature definitions for several higher-level features. Serenity generates reports based on them.

Or we can specify the **thusydides.requirements.dir** property in the Serenity.properties file to point to the root requirement directory.

Thucydides.requirements.dir=src/test/resources/com/example/next/features

**The Scenario Runner:**

***Cucumber runs the feature files*** *via* ***Junit***, and ***needs a dedicated test runner class*** to ***actually run*** the ***feature file***.

When we run tests with Serenity, we use **CucumberWithSerenity** test runner.

If the ***feature files*** are ***not in the same package*** as the ***test runner class***, we ***need to use @CucumberOptions class*** to ***provide*** the ***rood directory*** where the ***feature files can be found***.

**Step Definitions:**

In Cucumber, ***each line of the Gherkin scenario maps*** to a ***method in a Java class***, known as ***Step -Definition.***These use annotations like @Given, @When and @Then match lines in scenario to Java methods and define simple regular expressions to indicate parameter that will be passed into the methods.

Theses step definitions use Serenity to organize the step definition code into reusable components.

The @Step annotation tells Serenity that this variable is a Step Library. In Serenity we use Step libraries to add a layer of abstraction between the “What” and the “how” of our acceptance tests.

The Cucumber step definitions describe “what” the acceptance test is doing, in fairly implementation - neutral, business-friendly terms. So we say “searches for items containing ‘nike’”, not “enters ‘nike’ into the search field and clicks on search button”. This layered approach make the tests both easier to understand and to maintain, and helps build up a great library of reusable business steps that we can use in other tests.

Step definition files need to go in or underneath the package containing the scenario runners.

@@@@@@@@@@@@

Configuring Serenity: just need to add few dependencies:

Net.serenity-bdd:serenity-core:1.1.1

Net.serenity-bdd:serenity-junit:1.1.1

Junit:4.12

**Building Serenity projects in Maven:**

***Serenity BDD integrates*** ***with Maven*** via the **serenity-maven-plugin**

**Concepts of the required dependencies and plugin in pom:**

Steps 1 - - > We need to add Serenity – Core dependency.

<!-- https://mvnrepository.com/artifact/net.serenity-bdd/serenity-cucumber -->

<dependency>

<groupId>net.serenity-bdd</groupId>

<artifactId>serenity-cucumber</artifactId>

<version>1.9.45</version>

</dependency>

Step 2 - - > we need to some plugins to build serenity report with Maven.

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>8</source>

<target>8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.0</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.0</version>

<configuration>

<testFailureIgnore>true</testFailureIgnore>

</configuration>

</plugin>

<plugin>

<artifactId>maven-failsafe-plugin</artifactId>

<version>2.18</version>

<configuration>

<includes>

<include>\*\*/features/\*\*/When\*.java</include>

</includes>

<systemProperties>

<webdriver.driver>${webdriver.driver}</webdriver.driver>

</systemProperties>

</configuration>

</plugin>

<plugin>

<groupId>net.serenity-bdd.maven.plugins</groupId>

<artifactId>serenity-maven-plugin</artifactId>

<version>${serenity.maven.version}</version>

<executions>

<execution>

<id>serenity-reports</id>

<phase>post-integration-test</phase>

<goals>

<goal>aggregate</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

Step 3 - - > In order to set default config for serenity, we need create a Serenity.config or Serenity.properties file:

webdriver {

base.url = "https://www.google.com "

driver = chrome

}

headless.mode=false

serenity {

project.name = "Serenity Guidelines"

tag.failures = "true"

linked.tags = "issue"

restart.browser.for.each = scenario

take.screenshots = AFTER\_EACH\_STEP

console.headings = minimal

browser.maximized = true

}

jira {

url = "https://jira.tcbs.com.vn"

project = Auto

username = username

password = password

}

drivers {

windows {

webdriver.chrome.driver = src/main/resources/webdriver/windows/chromedriver.exe

}

mac {

webdriver.chrome.driver = src/main/resources/chromedriver

}

linux {

webdriver.chrome.driver = src/main/resources/webdriver/linux/chromedriver

}

}

Another dependency supported library which is Cucumber or Junit or JBehave.

Typically, we want Serenity tests run as integration tests (during integration phase of the maven build) rather than unit tests.

And also want to the build to immediately fail when a test fails, but continue until it has generated the Serenity aggregate reports before failing at the end of the build. To do this we use

**maven-failsafe-plugin –** this plug in runs our integration test in the integration-test phase without immediately failing the build when a test fails. Build failure is triggered later in the lifecycle, during the verify phase. Also it is good idea turn off failing build if some test was failed -just to allow maven execute all tests.

Normal Junit tests run from Maven need o start or end with Test. But for acceptance tests, a more flexible strategy is better, as it makes it easier to name test cases after scenarios or stories. In pom.xml file we configure maven-failsafe-plugin to run all of the tests in the junit directory, regardless of how they are name.

Next we need to add and configure the **serenity-maven-plugin**. A useful technique is to build the aggregate goal plugin to the post-integration-test phase. To run tests and to generate the reports we use **mvn verify**.

Like the surefire plugim the maven-failsafe-plugin starts a new JVM instance to run the tests.

@@@@@@@@@@@@@@@@@@@@@@

**Managing Screenshots:**

By default, Serenity saves screenshot for every step executed during the tests, Serenity can be configured when screenshots are stored.

The property **serenity.take.screenshots** can be set to configure how often the screenshots are taken with properties values.

FOR\_EACH\_ACTION - - > saves a screenshot at every web element action.

BEFORE\_AND\_AFTER\_EACH\_STEP - - > saves screenshot before and after every step.

AFTER\_EACH\_STEP - - > saves a screenshot after every step

FOR\_FAILURES - - > saves screenshots only for failing steps.

DISABLED - - > doesn’t save screenshot for any steps.

**Using annotation to control screenshot:** can annotate any test or step method

**@Screenshots(onlyOnFailures=true)** - - > only on failures

@Screenshots(forEachStep=true) - - > for each step in test.

@Screenshots(forEachAction=true) - - > for each action in the test.

@Screenshots(disabled=true) - - > doesn’t take screenshot for any action.

**Taking screenshots at any arbitrary point during a step:**

**Serenity.takeScreenshot()**  - - > calling this method will capture screenshot in step methods.

**Serenity BDD features and annotations:**

**@Steps** - - > marks a Serenity step library. When we annotated a member of the class with the @Steps annotation, Serenity BDD will automatically instantiate it for us.

**@step** - - > marks a method that will be recorded and will appear in the test report. Step methods are annotated with the @Step annotation.

@Step annotation can take a String value to override the default step name.

By default, the name of a step is derived from the method name, if we want something more readable, we add a String parameter to the @Step annotation.

If we put references to variable in the String ({0}, {1} etc). Serenity will inject the method parameter into the String.

**@Manual** - - > defined on scenario level or all scenarios in story.

**@manual** - - > defined on story level

[ @ manual is not designed to be defined for individual step within a test]

**@Managed** - - > define useful parameter. [ @Managed(driver=”chrome”) ]

Can also get Serenity to open browser at start of tests [ @Manager(uniqueSession=true) ]

Can also used to clear cookies [ @Manager(clearCookies=BeforeEachTest) ]

**@Pending** - - > skipping the tests.

**@Concurrent** - - > define in class level after @RunWith for parallel execution

[ @Concurrent(threads =”4”) uses 4 threads per CPU]

**@TestData** - - > to pass parameter to the tests

**Storing data between steps:**

Approach 1: by setting member variables in the step definition classes member variable and used in steps

Approach 2: member variable is set in the @Step method and then reused in the subsequent steps.

Approach 3: Serenity provides test session, which is essentially a hasp map where we can store variables for the duration of a single test. Variables in the Serenity test session are available in any step definition class.

Step 1 : store the value in a session variable.

Serenity.setSessionVariable(“variablename”).to(parameter passed in method);

Step 2: retrieve the session variable into another step.

String item-Serenity.sessionVariableCalled(“variablename”).toString();

[Note: the step methods can take parameter, the parameters that are passed into step will be recorded and reported in the Serenity report.]

**Setting up project and organizing the directory structure.**

Serenity tries to simplify the process by using a connection-over-configuration approach, which significantly reduces amount the work needed to get started with the acceptance tests.

By default, Serenity supports a simple directory-based convention for organizing the requirements. The standard structures uses three levels, capabilities, features and stories.

As a +user+ -- > will be provided in the report as coverage.

\*I want to see all list\* - - > will be provided in the report as narrative text.

**Comments in Scenario:**

Serenity will ignore a commented condition, but it will be displayed in report.

PageObjects -- > Serenity provides PageObjects class to interact with the actual web application.

**Running test against a Selenium Grid server:**

To run Serenity test on Selenium Grid, we need to provide the URL of the Slenium Hub using

webdriver.remote.url property.

To provide specific driver

webdriver.remote.driver property

To provide specific o/s

webdriver.remote.os property

To provide specific version

webdriver.remote.version property

To we are running Selenium Hub locally on default port 4444:

mvn verify -Dwebdriver.remote.url=http://localhost:4444/wd/hub -Dwebdriver.remote.driver=chrome -Dwebdriver.remote.os=WINDOWS

Serenity provides: ( issues for Cucumber @issues: ISSUE-123,ISSUE-789 in feature files).

WithTag( name=”name” value=”value ” - - > to organize the test executions

@ssue - - > used for linking single issue

@Issues - - > used for multiple linking

@Title - - > used for providing readable name of test case

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

**Testing REST with Serenity BDD:**

Serenity BDD can help to create tests for REST services, with all advantages that Sereinity BDD introduce to Web Test and even more.

**\*All we need is use in import instead of RestAssured - SerenityRest.**

**Reports crated when Rest is tested:**

If we use SerenityRest then all request/response will be included in generated reports.

**Command line exection:**

> mvn clean verify -Dtest=path\_to\_the\_AcceptanceTest

To generate the Serenity report:

>mvn clean verify -Dtest=path\_to\_the\_AcceptanceTest serenity:aggregate