Serenity BDD

Behaviour-Driven Test Automation with Serenity BDD and Cucumber-JVM

Last updated: 10 July 2017

Contents

Introduction

This document is based on documentation written and owned by John Ferguson Smart, https://github.com/serenity-bdd/serenity-articles

http://www.serenity-bdd.net [Serenity BDD] (previously known as "Thucydides") is an open source reporting library that helps you write better structured, more maintainable automated acceptance criteria, and also produces rich meaningful test reports (or "living documentation") that not only report on the test results, but also what features have been tested. This document describes how Serenity BDD works with the popular Behaviour-Driven Development (BDD) tool http://cukes.info [Cucumber-JVM].

Behaviour-Driven Development (BDD) fundamentals

Behaviour-Driven Development (BDD) is a core concept underlying many of Serenity BDD's features. A team using Behaviour-Driven Development use conversations and collaboration around concrete examples to build up a shared understanding of the features they are supposed to build. Conversations about concrete examples, and counter-examples, are a great way to flush out any hidden assumptions or misunderstandings about what a feature needs to do.

Suppose you are building a web site where artists and craftspeople can sell their goods online. One important feature for such a site would be the search feature. You might express this feature using a story-card format commonly used in agile projects like this:

```
In order to find items that I would like to purchase

As a potential buyer

I want to be able to search for items containing certain words
```

To build up a shared understanding of this requirement, you could talk through a few concrete examples. The conversation might go something like this:

- "So give me an example of how a search might work."

- "Well, if I search for 'wool', then I should see only woollen products."

- "Sound's simple enough. Are there any other variations on the search feature that would produce different outcomes?"

- "Well, I could also filter the search results; for example, I could look for only handmade woollen products."

- "So you can filter by handmade items. Could you give me some examples of other product types you would want to filter by?"

And so on. In practice, many of the examples that get discussed become “acceptance criteria” for the features. And many of these acceptance criteria become automated acceptance tests. Automating acceptance tests provides valuable feedback to the whole team, as these tests, unlike unit and integration tests, are typically expressed in business terms, and can be easily understood by non-developers. The reports that are produced when these tests are executed give a clear picture of the state of the application.

Writing acceptance criteria with Cucumber-JVM

http://cukes.info [Cucumber] is a popular BDD test automation tool. Cucumber-JVM is the Java implementation of Cucumber, and is what we will be focusing on. In Cucumber, you express acceptance criteria in a natural, human-readable form. For example, we could write the “wool scarf” example mentioned before like this:

```
Given I want to buy a wool scarf

When I search for items containing ‘wool’

Then I should only see items related to ‘wool’
```

This format is known as “Gherkin”, and is widely used in Cucumber and other Cucumber-based BDD tools such as Specflow (for Microsoft .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 focus on what they are trying to achieve, and how they will know when they get it.

Sometimes tables can be used to summarise several different examples of the same scenario. In Gherkin, you can use example tables to do this. For instance, the following scenario illustrates how you can search for different types of products made of different materials:

```
Scenario Outline: Filter by different item types
  Given I have searched for items containing '<material>'
  When I filter results by type '<type>'
  Then I should only see items containing '<material>' of type '<type>'
Examples:
  | material | type           |
  | silk     | Handmade       |
  | bronze   | Vintage        |
  | wool     | Craft Supplies |
```

We will learn how to automate these scenarios using Cucumber and Serenity BDD.

Writing executable specifications with Cucumber and Serenity BDD

Writing the scenario

Let's start off with the first example discussed before. In Cucumber, scenarios are stored in "Feature Files", which contain an overall description of a feature as well as a number of scenarios. The Feature File for the example is called "`search_by_keyword.feature`", and looks something like this like this:

(`search_by_keyword.feature` file):

```
Feature: Searching by keyword

  In order to find items that I would like to purchase
  As a potential buyer
  I want to be able to search for items containing certain words

  Scenario: Should list items related to a specified keyword
    Given I want to buy a wool scarf
    When I search for items containing 'wool'
    Then I should only see items related to 'wool'

  Scenario: Should be able to filter search results by item type
    Given I have searched for items containing 'wool'
    When I filter results by type 'Handmade'
    Then I should only see items containing 'wool' of type 'Handmade'

  Scenario Outline: Filter by different item types
    Given I have searched for items containing '<material>'
    When I filter results by type '<type>'
    Then I should only see items containing 'foo' of type '<type>'
  Examples:
    | material | type           |
    | silk     | Handmade       |
    | bronze   | Vintage        |

  Scenario: Should be able to view details about a searched item
    Given I have searched for items containing 'yarn'
    When I select an item
    Then I should see the corresponding item details
```

These feature files can be placed in different locations, but you can reduce the amount of configuration you need to do with Serenity BDD if you put them in the "`src/test/resources/features`" directory.

You typically organise the feature files in sub-directories that reflect the higher-level requirements. In the following directory structure, for example, we have feature definitions for several higher-level features: "`search`" and "`shopping_cart`":

```
|----src
| |----test
| | |----resources
| | | |----features
| | | | |----search
| | | | | |----search_by_keyword.feature
| | | | |----shopping_cart
| | | | | |----adding_items_to_the_shopping_cart.feature
```

The Scenario Runner

Cucumber runs the feature files via JUnit, and needs a dedicated test runner class to actually run the feature files. When you run the tests with Serenity BDD, you use the "`CucumberWithSerenity`" test runner. If the feature files are not in the same package as the test runner class, you also need to use the "`@CucumberOptions`" class to provide the root directory where the feature files can be found. The test runner to run all of the feature files looks like this:

(`AcceptanceTests.java` file):

```
package net.serenity_bdd.samples.etsy.features;

import cucumber.api.CucumberOptions;
import net.serenitybdd.cucumber.CucumberWithSerenity;
import org.junit.runner.RunWith;

@RunWith(CucumberWithSerenity.class)
@CucumberOptions(features="src/test/resources/features")
public class AcceptanceTests {}
```

Step definitions

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

(`SearchByKeywordStepDefinitions.java` file):

```
package net.serenity_bdd.samples.etsy.features.steps;

import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
import net.serenity_bdd.samples.etsy.features.steps.serenity.BuyerSteps;
import net.thucydides.core.annotations.Steps;

public class SearchByKeywordStepDefinitions {
    @Steps
    BuyerSteps buyer;

    @Given("I want to buy (.*)")
    public void buyerWantsToBuy(String article) {
        buyer.opens_etsy_home_page();
    }

    @When("I search for items containing '(.*)'")
    public void searchByKeyword(String keyword) {
        buyer.searches_for_items_containing(keyword);
    }

    @Then("I should only see items related to '(.*)'")
    public void resultsForACategoryAndKeywordInARegion(String keyword) {
        buyer.should_see_items_related_to(keyword);
    }

    String searchTerm;

    @Given("I have searched for items containing '(.*)'")
    public void buyerHasSearchedFor(String keyword) {
        searchTerm = keyword;
        buyer.opens_etsy_home_page();
        buyer.searches_for_items_containing(keyword);
    }

    @When("I filter results by type '(.*)'")
```

```
    public void filterResultsBy(String type) {
        buyer.filters_results_by_type(type);
    }

    @Then("I should only see items containing '(.*)' of type '(.*)'")
    public void shouldSeeMatchingFilteredResults(String keyword, String type) {
        buyer.should_see_items_related_to(keyword);
        buyer.should_see_items_of_type(type);
    }

    @When("I select an item")
    public void selectsAnItem() {
        buyer.selects_item_number(1);
    }

    @Then("I should see the corresponding item details")
    public void shouldSeeCorrespondingDetails() {
        buyer.should_see_matching_details(searchTerm);
    }
}
```

Serenity BDD uses these step definitions to organise the step definition code into more reusable components. The "`@Steps`" annotation tells Serenity BDD that this variable is a Step Library. In Serenity BDD, 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 'wool'", not "enters 'wool' into the search field and clicks on the search button". This layered approach makes the tests both easier to understand and to maintain, and helps build up a great library of reusable business-level steps that we can use in other tests. Without this kind of layered approach, step definitions tend to become very technical very quickly, which limits reuse and makes them harder to understand and maintain.

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

```
|----src
| |----test
| | |----java
| | | |----net
| | | | |----serenity_bdd
| | | | | |----samples
| | | | | | |----etsy
| | | | | | | |----features                                    //<1>
| | | | | | | | |----AcceptanceTests.java                      //<2>
| | | | | | | | |----steps                                     //<3>
| | | | | | | | | |----SearchByKeywordStepDefinitions.java
| | | | | | | | | |----serenity                                  //<4>
| | | | | | | | | | |----BuyerSteps.java
```

<1> The scenario runner package

<2> A scenario runner

<3> Step definitions for the scenario runners

<4> Serenity Step Libraries are placed in a different sub-package

The Serenity BDD Step Libraries

A Serenity BDD Step Library is just an ordinary Java class, with methods annotated with the "`@Step`" annotation, as shown here:

(`BuyerSteps.java` file):

```
package net.serenity_bdd.samples.etsy.features.steps.serenity;

import com.google.common.base.Optional;
import net.serenity_bdd.samples.etsy.pages.HomePage;
import net.serenity_bdd.samples.etsy.pages.ItemDetailsPage;
import net.serenity_bdd.samples.etsy.pages.SearchResultsPage;
import net.thucydides.core.annotations.Step;
import org.hamcrest.Matcher;
import java.util.List;
import static org.assertj.core.api.Assertions.assertThat;

public class BuyerSteps {

    HomePage homePage;                                          // <1>
    SearchResultsPage searchResultsPage;

    @Step                                                       // <2>
    public void opens_etsy_home_page() {
        homePage.open();
    }

    @Step
    public void searches_for_items_containing(String keywords) {
        homePage.searchFor(keywords);
    }

    @Step
    public void should_see_items_related_to(String keywords) {
        List<String> resultTitles = searchResultsPage.getResultTitles();
        resultTitles.stream().forEach(title -> assertThat(title.contains(keywords)));
    }

    @Step
    public void filters_results_by_type(String type) {
        searchResultsPage.filterByType(type);
    }

    public int get_matching_item_count() {
        return searchResultsPage.getItemCount();
    }

    @Step
    public void should_see_item_count(Matcher<Integer> itemCountMatcher) {
        itemCountMatcher.matches(searchResultsPage.getItemCount());
    }

    ItemDetailsPage detailsPage;

    @Step
    public void selects_item_number(int number) {
        searchResultsPage.selectItem(number);
    }

    @Step
    public void should_see_matching_details(String searchTerm) {
        String itemName = detailsPage.getItemName();
        assertThat(itemName.toLowerCase()).contains(searchTerm);
    }

    @Step
    public void should_see_items_of_type(String type) {
        Optional<String> selectedType = searchResultsPage.getSelectedType();
        assertThat(selectedType.isPresent()).describedAs("No item type was selected").isTrue();
        assertThat(selectedType.get()).isEqualTo(type);
    }
}
```

<1> Step libraries often use Page Objects, which are automatically instantiated

<2> The "`@Step`" annotation indicates a method that will appear as a step in the test reports

These step definitions implement the "what" behind the "how" of the "Given-When-Then" steps. However, like any well-written code, step definitions should not be overly complex, and should focus on working at a single level of abstraction. Step definitions typically orchestrate calls to more technical layers such as web services, databases, or WebDriver Page Objects. For example, in automated web tests like this one, the step library methods do not call WebDriver directly, but rather they typically interact with "Page Objects".

The Page Objects

Page Objects encapsulate how a test interacts with a particular web page. They hide the WebDriver implementation details about how elements on a page are accessed and manipulated behind more business-friendly methods. Like steps, Page Objects are reusable components that make the tests easier to understand and to maintain.

Serenity BDD automatically instantiates Page Objects for you, and injects the current WebDriver instance. All you need to worry about is the WebDriver code that interacts with the page. And Serenity BDD provides a few shortcuts to make this easier as well. For example, here is the Page Object for the Home page:

(`HomePage.java` file):

```
package net.serenity_bdd.samples.etsy.pages;

import com.google.common.base.Function;
import com.google.common.collect.Lists;
import com.thoughtworks.selenium.webdriven.commands.WaitForCondition;
import net.thucydides.core.annotations.DefaultUrl;
import net.thucydides.core.pages.PageObject;
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.CacheLookup;
import org.openqa.selenium.support.FindAll;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.FluentWait;
import org.openqa.selenium.support.ui.Wait;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.util.List;
import java.util.NoSuchElementException;
import java.util.concurrent.TimeUnit;

@DefaultUrl("http://www.etsy.com")                      // <1>
public class HomePage extends PageObject {              // <2>

    @FindBy(css = "button[value='Search']")
    WebElement searchButton;

    public void searchFor(String keywords) {
        $("#search-query").sendKeys(keywords);          // <3>
        searchButton.click();                           // <4>
    }
}
```

<1> What URL should be used by default when we call the "`open()`" method

<2> A Serenity BDD Page Object must extend the "`PageObject`" class

<3> You can use the "`$`" method to access elements directly using CSS or XPath expressions

<4> Or you may use a member variable annotated with the "`@FindBy`" annotation

And here is the second Page Object we use:

(`SearchResultsPage.java` file):

```
package net.serenity_bdd.samples.etsy.pages;

import com.google.common.base.Optional;
import net.serenitybdd.core.pages.WebElementFacade;
import net.thucydides.core.pages.PageObject;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

import java.util.List;
import java.util.stream.Collectors;

public class SearchResultsPage extends PageObject {
    public static final String SCROLL_TO_FILTERS = "$('a.red')[0].scrollIntoView(false);";

    @FindBy(css=".listing-card")
    List<WebElement> listingCards;

    public List<String> getResultTitles() {
        return listingCards.stream()
                .map(element -> element.getText())
                .collect(Collectors.toList());
    }

    public void selectItem(int itemNumber) {
        listingCards.get(itemNumber - 1)
                .findElement(By.tagName("a")).click();
    }

    public void filterByType(String type) {
        showFilters();
        findBy("#search-filter-reset-form").then(By.partialLinkText(type)).click();
    }

    public int getItemCount() {
        String resultCount = $(".result-count").getText()
                .replace("We found ","")
                .replace(" item","")
                .replace("s","")
                .replace("!","")
                .replace(",","")
                ;
        return Integer.parseInt(resultCount);
    }

    public Optional<String> getSelectedType() {
        List<WebElementFacade> selectedTypes = findAll("#search-filter-reset-form a.radio-
label.strong");
        return (selectedTypes.isEmpty()) ? Optional.absent() :
Optional.of(selectedTypes.get(0).getText());
    }

    public void showFilters() {
        evaluateJavascript(SCROLL_TO_FILTERS);
    }
}
```

In both cases, we are hiding the WebDriver implementation of how we access the page elements inside the page object methods. This makes the code both easier to read and reduces the places you need to change if a page is modified.

Reporting and Living Documentation

Reporting is one of Serenity BDD's fortes. Serenity BDD not only reports on whether a test passes or fails, but documents what it did, in a step-by-step narrative format that includes test data and screenshots for web tests. For example, the following page illustrates the test results for our first acceptance criteria:

Test results reported in Serenity BDD

Test Outcome

Should List Items Related To A Specified Keyword

Feature: Search By Keyword

Searching By Keyword (feature)

Search (capability)

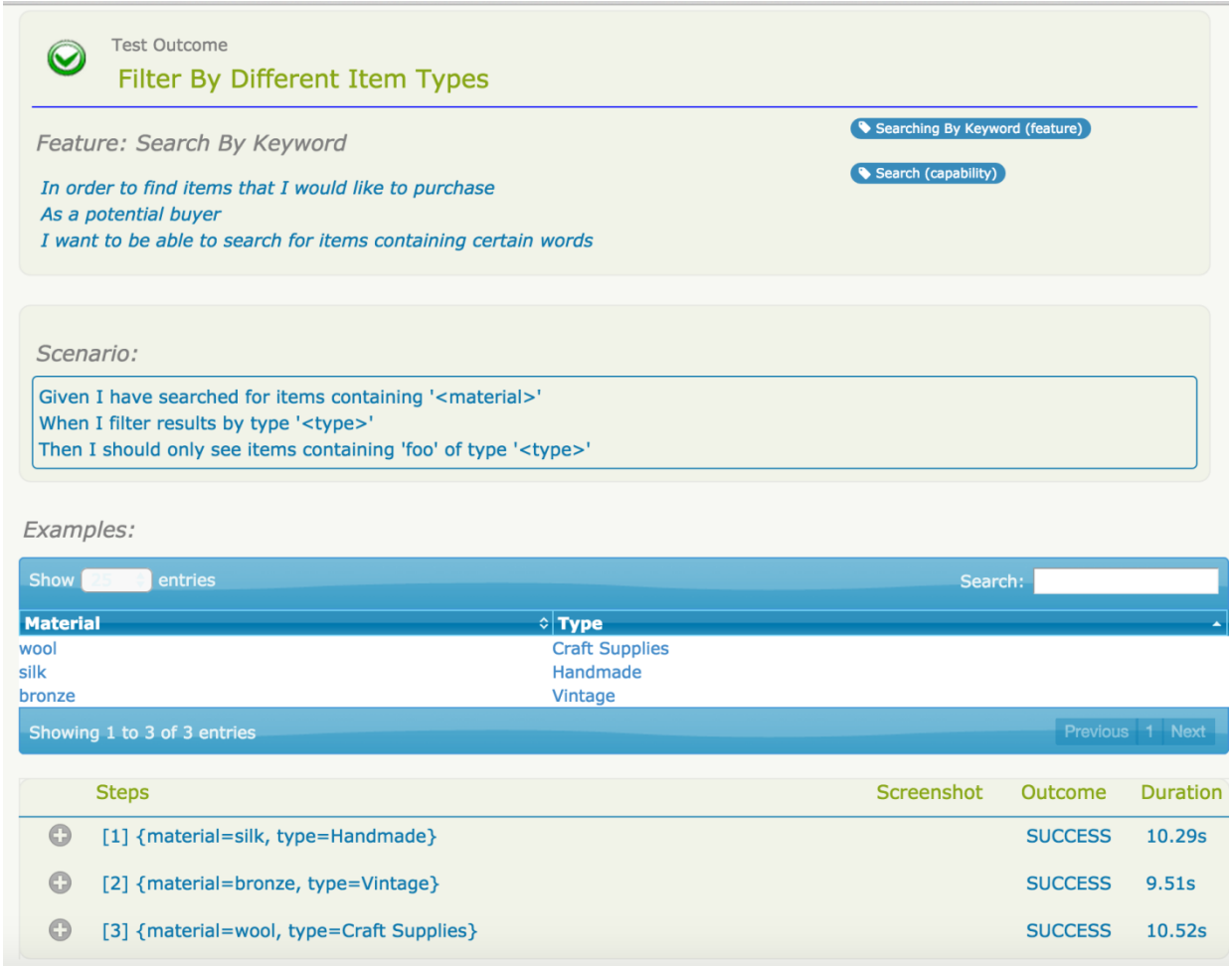
In order to find items that I would like to purchase
As a potential buyer
I want to be able to search for items containing certain words

Steps	Screenshot	Outcome	Duration
Given I want to buy a wool scarf		SUCCESS	7.48s
Opens etsy home page		*SUCCESS*	*6.84s*
When I search for items containing 'wool'		SUCCESS	2.58s
Searches for items containing: wool		*SUCCESS*	*2.06s*
Then I should only see items related to 'wool'		SUCCESS	3.6s
Should see items related to: wool		*SUCCESS*	*3.11s*

Notice how this report faithfully reproduces the example from the conversation with the business, and also gives the option of stepping into the "what", to see how a particular step has been implemented, and (in this case) what the corresponding screen shots look like.

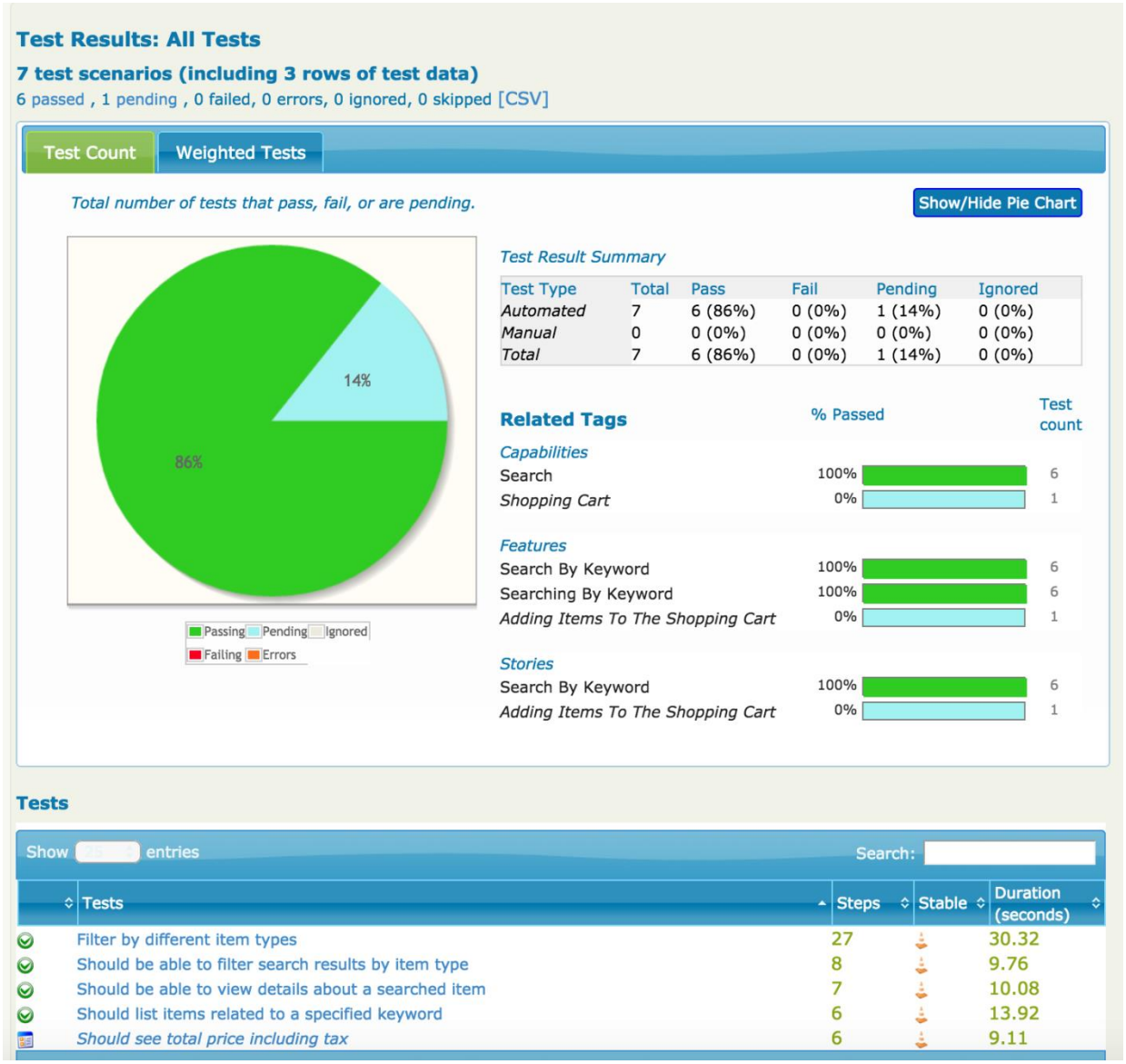
We saw previously how example tables can be a great way to summarize business logic; it is important for these tables to be reflected in the test results, as illustrated here:

Test results containing an examples table in Serenity BDD



But test outcomes are only part of the picture. It is also important to know what work has been done, and what is work in progress. When you are using Cucumber, any scenario that contains steps without matching step definition methods will appear in the reports as “Pending” (blue in the graphs).

Test result overview

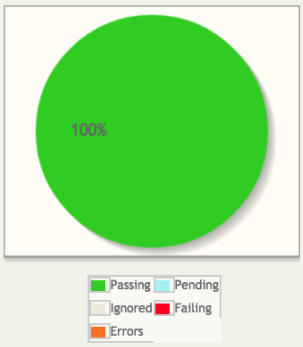


In the previous section, we saw how the feature files were organized into directories that represent higher-level features or capabilities. Serenity BDD will use this package structure to group and aggregate the test results for each feature. This way, Serenity BDD can report about how well each requirement has been tested, and will also tell you about the requirements that have "not" been tested:

Serenity BDD features report

Capability: Search

Search



Requirements Overview

Requirement Type	Total	Pass	Fail	Pending	Ignored	Untested
Features	1	1	0	0	0	0
Acceptance Criteria (tests)	6	6	0	0	0	

Test Result Summary

Test Type	Total	Pass	Fail	Pending	Ignored
Automated	6	6 (100%)	0 (0%)	0 (0%)	0 (0%)
Manual	0	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Total	6	6 (100%)	0 (0%)	0 (0%)	0 (0%)

Features (1)

Show 25 entries Search:

ID	Feature	Auto. Tests					Manual Tests					Coverage
	Searching by keyword *In order to find items that I would like to purchase* *As a potential buyer* *I want to be able to search for items containing certain words*	*6*	6	0	0	0	*0*	0	0	0	0	100%

Showing 1 to 1 of 1 entries Previous 1 Next

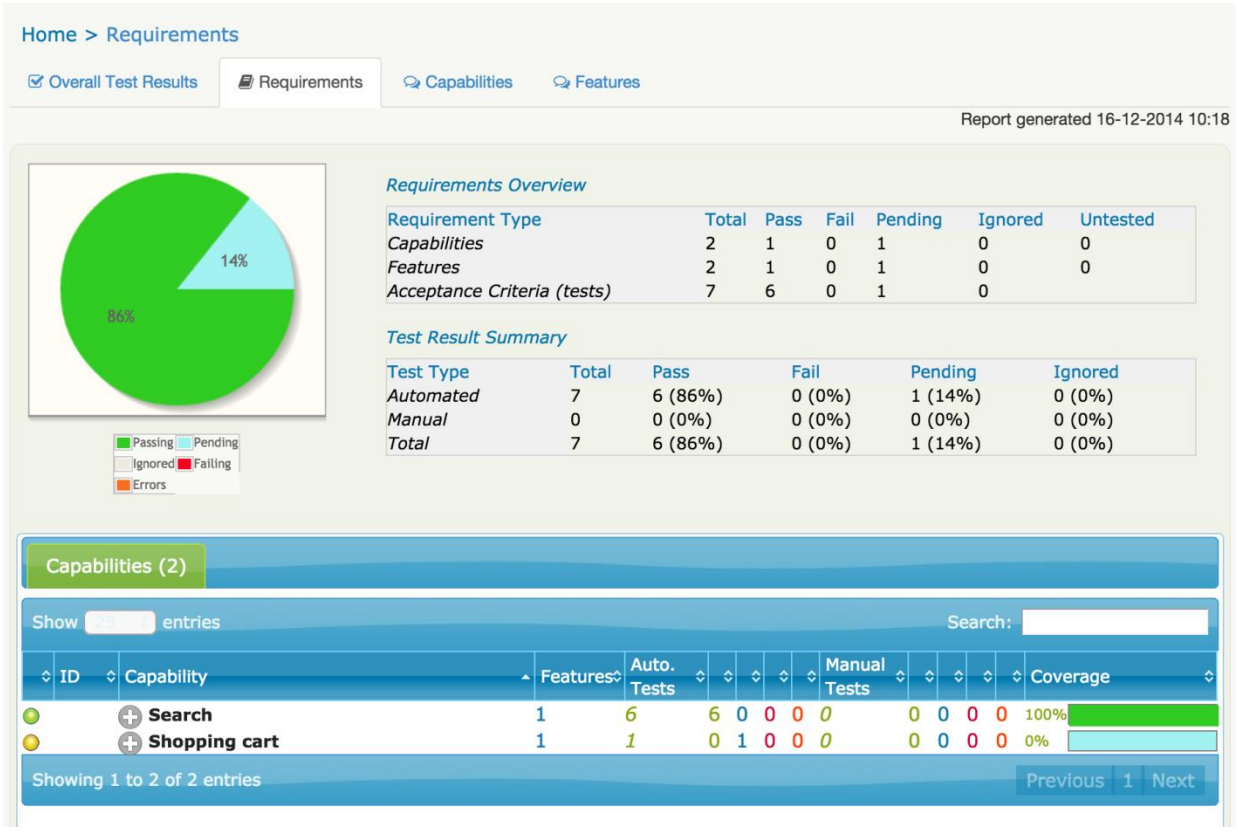
Acceptance Tests (6)

Show 25 entries Search:

Acceptance Tests	Steps	Stable	Duration (seconds)
Should list items related to a specified keyword	6		13.92
Should be able to view details about a searched item	7		10.08
Should be able to filter search results by item type	8		9.76
Filter by different item types	27		30.32

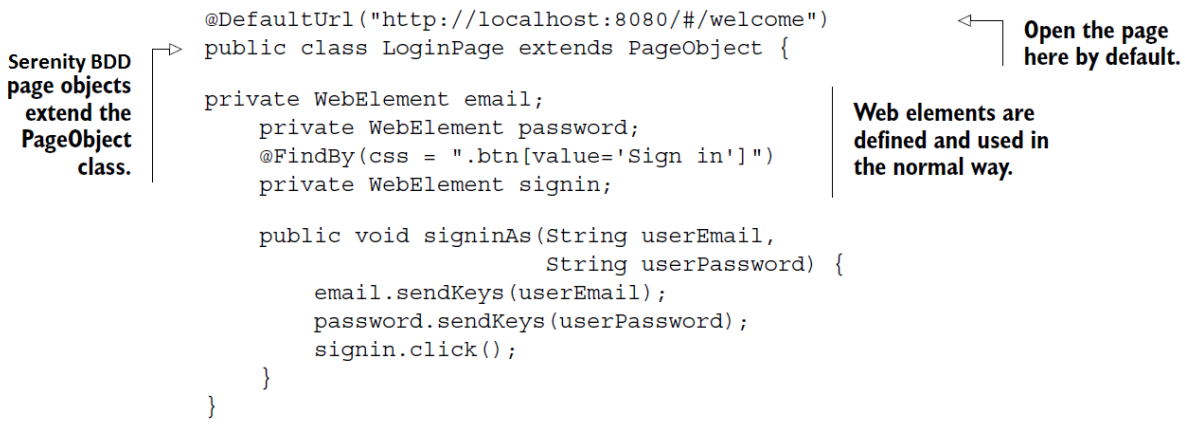
Showing 1 to 4 of 4 entries Previous 1 Next

Serenity BDD requirements report



Extending Selenium Page Objects with Serenity BDD

Serenity BDD takes care of instantiating Page Objects and managing the WebDriver instance and it provides a number of useful base methods. A simple Serenity BDD Page Object looks similar to the standard WebDriver:

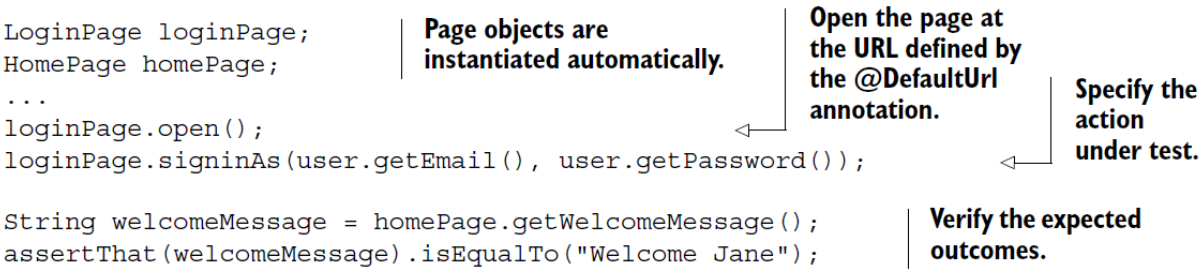



```
@DefaultUrl("http://localhost:8080/#/welcome")
public class LoginPage extends PageObject {

private WebElement email;
    private WebElement password;
    @FindBy(css = ".btn[value='Sign in']")
    private WebElement signin;

    public void signinAs(String userEmail,
                         String userPassword) {
        email.sendKeys(userEmail);
        password.sendKeys(userPassword);
        signin.click();
    }
}
```

Serenity BDD will also instantiate any page object instances in the test code, which simplifies the test code further:




```
LoginPage loginPage;
HomePage homePage;
...
loginPage.open();
loginPage.signinAs(user.getEmail(), user.getPassword());

String welcomeMessage = homePage.getWelcomeMessage();
assertThat(welcomeMessage).isEqualTo("Welcome Jane");
```

Example Maven POM for Serenity BDD

This Maven setup runs Serenity BDD tests as Integration tests using the Maven Failsafe plugin.

In Maven, Unit tests will be executed first, using the Maven Surefire plugin. Integration tests will only be executed after all Maven Surefire (Unit) tests pass.

However, the following setup skips Unit tests.

Serenity BDD (Integration) tests can be executed with this command from the command line:

```
mvn clean verify
```

This runs Serenity BDD (Integration) tests only and also generates an aggregate report in the "…`target/site/serenity`" directory. If you do also want to run the Unit tests, then please remove the section that blocks the Maven Surefire plugin (Unit tests) from executing.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.pepgo</groupId>
    <artifactId>serenity-bdd</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>Serenity BDD Integration Tests</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <serenity.version>1.0.47</serenity.version>
        <serenity.maven.version>1.0.47</serenity.maven.version>
        <serenity.cucumber.version>1.1.36</serenity.cucumber.version>
        <webdriver.driver>firefox</webdriver.driver>
        <tags></tags>
    </properties>

    <dependencies>
        <dependency>
            <groupId>net.serenity-bdd</groupId>
            <artifactId>core</artifactId>
            <version>${serenity.version}</version>
        </dependency>
        <dependency>
            <groupId>net.serenity-bdd</groupId>
            <artifactId>serenity-cucumber</artifactId>
            <version>${serenity.cucumber.version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.25</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.assertj</groupId>
```

```
            <artifactId>assertj-core</artifactId>
            <version>3.8.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <!-- Allow Java 8 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <!-- Don't run any tests during the unit test phase -->
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>
                    <skip>true</skip>
                </configuration>
            </plugin>

            <!-- Run all tests during the integration test phase -->
            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>2.20</version>
                <configuration>
                    <includes>
                        <include>net/serenity/samples/etsy/*.java</include>
                    </includes>
                    <systemProperties>
                        <webdriver.driver>${webdriver.driver}</webdriver.driver>
                        <tags>${tags}</tags>
                    </systemProperties>
                    <parallel>methods</parallel>
                    <threadCount>2</threadCount>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>integration-test</goal>
                            <goal>verify</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- Generate the test reports after the integration tests -->
            <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>
                <dependencies>
```

```
                    <dependency>
                        <groupId>net.serenity-bdd</groupId>
                        <artifactId>core</artifactId>
                        <version>1.0.47</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>
</project>
```