Lab 4 - Acceptance Testing with Cucumber

SENG301 Software Engineering II

Sam Burtenshaw Neville Churcher Morgan English Fabian Gilson
Jack Patterson

16th March 2022

Lab objectives

This lab aims to help students with their first use of acceptance test automation tools by using the *Cucumber* framework. Students will learn:

- why automated acceptance tests are important;
- how to set up the a `gradle` project with *Cucumber*;
- how to convert stories' acceptance criteria to code that will contribute to the success of a delivery.

1 Context

1.1 Acceptance testing

In Agile Software Development, acceptance criteria play a key role for the success of a product delivery. User stories are meant to be short and reflecting a *"promise for a conversation"*. Practical details are expressed in acceptance criteria representing the expected behaviour of a story under certain circumstances (e.g., normal or exceptional cases) or give more details about the exact attributes of concepts (i.e. domain entities) used in a story. This set of criteria is used by both the product owner and the developers as a sort of *"contract"* to agree on what is expected from a story to be successfully implemented (i.e. pass).

These tests are different from unit tests, which are aimed at developers and help them ensuring the code works as expected or even drive the code design (e.g., Test-Driven Development). In some ways, *"unit tests ensure you build the thing right, whereas acceptance tests ensure you build the right thing"*.

1.2 Acceptance Test-Driven development (ATDD)

Together with *Story Test-Driven Development* and *Specification by example*, *Acceptance Test-Driven development* (ATDD) was coined by Kent Beck in 2003, one of Agile's father[1]. Despite Beck dismissing the idea, ATDD is a powerful technique used within the *Behaviour-Driven Development* (BDD[2]) philosophy, an agile software development process that recommends conversation and collaboration across disciplines, that is, among the developers, product owner and other stakeholders. ATDD is sometines substituted with *Story Test-Driven Development* since acceptance criteria attached to stories are translated into a semi-formal language that enables automation testing from within the code itself. An example of such language is *Gherkin* using the general template `"Given, When, Then"` to describe user acceptance scenarios.

[1] See `https://www.agilealliance.org/glossary/atdd/`
[2] See `https://cucumber.io/docs/bdd/`

1.3 Acceptance testing and *Cucumber*

Cucumber[3] is a test automation tool that implements the BDD philosophy. Put in simple terms, *Cucumber* reads the acceptance tests written in the *Gherkin* language, a human-readable template-based language, and runs the acceptance tests. *Cucumber* is not synonymous of ATDD or BDD, but one of the available frameworks to automate your acceptance tests. Other ATDD tools exists, such as FitNesse[4] or to some degree, Selenium[5].

2 Domain, story, and acceptance criteria

This Lab will build on the example case used in *Lab 3 - Domain modelling and Java Persistence API*. Please refer to that handout if you need to. We reproduce an updated domain model in Figure 1.

Figure 1: Domain model of a Wordle App (updated from lab 3)

2.1 User stories

Let's assume your product owner came with the following user story and acceptance criteria. For the sake of simplicity, we identified only one class of users for the system, called *"player"*.

U1 As a player, I want to record the outcome of my wordle game so that I can keep track of my previous games.
- AC.1 I can create a game record with a user, a word, and a number of guesses that are not empty.
- AC.2 Two game records may exist with the same attributes but must have different timestamps
- AC.3 A game record can not have 0 or fewer guesses.
- AC.4 A timestamp cannot be in the future.

3 Set up *Cucumber* with *Gradle*

Next to this handout, you will find a `.zip` archive with an updated version of the expected result from lab 3. In a nutshell, the project is structured as follows:

`main` contains the sources of your project
- **`java`** java source files including the `model` and `accessor` packages
- **`resources`** application configuration files (i.e. `hibernate.cfg.xml` with *sqlite*)

`test` contains all test sources
- **`java`** java test sources
- **`resources`** test configuration files (i.e. `cucumber.properties` and `hibernate.cfg.xml` with *h2*)

`build.gradle` the dependencies configuration file

Compared to lab 3, two different `hibernate.cfg.xml` files are supplied to the application that you can uses interchangeably. One uses a *sqlite* database (under `main` package), the other uses an *h2* database (under `test`).

[3] See `https://cucumber.io/docs/cucumber/api/`
[4] See `http://www.fitnesse.org/`
[5] See `https://selenium.dev/`

3.1 Add *Cucumber* dependencies

Take a look at the `build.gradle` file we have set up for you. It is a complete version from lab 3 (with *Hibernate*, *h2* and *sqlite* dependencies filled in). You will need to add *Cucumber* dependencies (line 31-32) and related task (line 40-62), as shown in Listing 1 where we reproduce the expected full `build.gradle` configuration file.

```
1  plugins {
2      // Apply the application plugin to add support for building a CLI application in Java.
3      id 'application'
4  }
5
6  repositories {
7      // Use Maven Central for resolving dependencies.
8      mavenCentral()
9  }
10
11 dependencies {
12     // Use JUnit Jupiter for testing.
13     testImplementation 'org.junit.jupiter:junit-jupiter:5.8.1'
14
15     // Use JUnit Jupiter Engine for testing.
16     testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine'
17
18     // This dependency is used by the application.
19     implementation 'com.google.guava:guava:30.1.1-jre'
20
21     // use hibernate to persist (java) domain entities for us, aka JPA implementation
22     implementation 'org.hibernate:hibernate-core:5.6.5.Final'
23
24     // Sqlite as persistent DB (SENG301 lab 3)
25     implementation 'org.xerial:sqlite-jdbc:3.36.0.3'
26     implementation 'com.zsoltfabok:sqlite-dialect:1.0'
27
28     // use a in-memory database to store entities (can be substituted with any database)
29     implementation 'com.h2database:h2:2.1.210'
30
31     //Cucumber dependencies
32     testImplementation 'io.cucumber:cucumber-java:7.2.3'
33 }
34
35 application {
36     // Define the main class for the application.
37     mainClass = 'uc.seng301.wordleapp.lab4.App'
38 }
39
40 configurations {
41     cucumberRuntime {
42         extendsFrom testImplementation
43     }
44 }
45
46 sourceSets {
47     test {
48         resources.srcDirs = ['src/test/resources']
49         java.srcDirs = ['src/test/java', 'src/main/java']
50         runtimeClasspath = project.sourceSets.main.compileClasspath +
51                 project.sourceSets.test.compileClasspath +
52                 fileTree("src/test/resources/test") +
53                 project.sourceSets.test.output + project.sourceSets.main.output
54     }
55 }
56
57 task cucumber() {
58     dependsOn assemble, testClasses
59     doLast {
60         javaexec {
```

```
61            main = "io.cucumber.core.cli.Main"
62            classpath = configurations.cucumberRuntime  + sourceSets.test.output
63            args = ['--plugin', 'pretty', '--glue', 'gradle.cucumber', 'src/test/resources']
64        }
65    }
66 }
67
68 tasks.named('test') {
69    // Use JUnit Platform for unit tests.
70    useJUnitPlatform()
71    // force the test task to run Cucumber too
72    finalizedBy cucumber
73 }
```

Listing 1: Additional configuration settings in `build.gradle` file.

line 32 the cucumber dependency

lines 40-44 this `configurations` section declares the `cucumberRuntime` as a testing configuration.

line 46-55 this enables to use a separate `hibernate.cfg.xml` file either to run or test the application. Compare both configuration files, you will notice that one uses *sqlite*, the other `h2`. It is good practice to use a non-persisting database for tests.

lines 57-73 specify a new task named `cucumber` that can be invoked by typing `$ ./gradlew cucumber` and will run the *Cucumber* tests for you. Because that task is a java class, it uses the `javaexec` engine (i.e. `java` command) with the parameters defined in line 63. These parameters are specifying:

- a *"pretty"* output format to display the results of the *Cucumber* tests;
- the `gradle.cucumber` package where the acceptance tests will be placed;
- the `src/test/resources` folder where the `cucumber.properties` are.

lines 57-62 we extend the `test` task to always run the `cucumber` task too.

3.2 Further configurations for *Cucumber*

You can take a look into the file `app/src/test/resources/cucumber.properties`. It contains a unique line to ask *Cucumber* to not print additional messages regarding the online publishing of test results. Additional properties can be set in that file, if need be (e.g., defining object factories for custom type mapping if you want to make *Cucumber* aware of more advanced types as test case parameters).

3.3 Run *Cucumber*

To make sure you have set up your `build.gradle` file correctly, run the `cucumber` task. To this end, execute the task by running the command `$ ./gradlew cucumber`. In *IntelliJ*, you may need to ask to refresh the configuration for *IntelliJ* to pick up the configuration changes. Click on the small arrow icon visible inside the `build.gradle` file (close to the *elephant* icon somewhere at the top right of the opened file editor), or via the same button in the `gradle` tool window in *IntelliJ* (accessible via the *"gradle"* button on the extreme right, close to the top). You should get a similar output as Listing 2.

```
> Task :app:cucumber
Mar 14, 2022 11:21:57 AM io.cucumber.core.runtime.FeaturePathFeatureSupplier get
WARNING: No features found at file:/C:/Users/morga/IdeaProjects/uc-seng301-wordleapp/app/src/test/resources/

0 Scenarios
0 Steps
0m0.063s

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own
     scripts or plugins.
See https://docs.gradle.org/7.4/userguide/command_line_interface.html#sec:command_line_warnings
BUILD SUCCESSFUL in 2s
```

Listing 2: First execution of `cucumber` task with no scenario yet.

Important notes:

- If your build fails with a message like *"It is currently in use by another Gradle instance"*, you may need to run `$ find ~/.gradle -type f -name "*.lock" -delete` since the lab machines are currently facing some disk access issues. After deleting these `lock` files, you should be able to rerun the task successfully.
- if you receive an error like *"Error: Could not find or load main class org.gradle.wrapper.GradleWrapperMain Caused by: java.lang.ClassNotFoundException: org.gradle.wrapper.GradleWrapperMain"*, this means you miss `gradle`'s `jar` library. You can simply run `$ gradle wrapper` (not `./gradlew`) to fetch and setup the missing library.
- if you receive an error like *"bash: ./gradlew: Permission denied"*, you can also run `$ gradle wrapper` (to fetch again an executable version) or `$ chmod u+x gradlew` (to make the `gradlew` script executable).

4 Writing acceptance tests with *Cucumber*

You will now learn how to set up a first acceptance test scenario.

4.1 Set up your first `.feature` file

As we defined in Listing 1, *Cucumber* will look under the `gradle.cucumber` package to find *Cucumber* features and tests.

1. You need to create a folder called `gradle` under `app/src/test/resources/`.
2. Inside that `gradle` folder, create a `cucumber` folder.
3. Create a file named `gamerecord.feature` (as a new generic file), **filling the extension by yourself**.

4.2 Defining the first scenario

In the `feature` file you just created, you can add the following test description, reproduced in Listing 3.

```
1 Feature: U1 - Create a new game record
2
3   Scenario:  AC1 - Create a game record with player, and number of guesses
4     Given I have a player "John"
5     When I create a game record with the player, word "crane", and 4 guesses
6     Then The game record is created with the correct user, word, number of guesses, and the current timestamp
```

Listing 3: First acceptance test scenario (U1-AC1).

A `feature` file can contain many scenarios and each scenario is composed of a description and some *"steps"*.

line 1 A feature file starts with the name of the feature. **As a convention in SENG301/302**, we use the story number (i.e. U1) and a shortened version of the story description.

line 3 Each acceptance criteria (AC) has its corresponding `Scenario`. Similarly to `Features` referring to the user story number, **we prefix the scenario description with the id of the AC** followed by a dash.

line 4 The `Given` clause denotes a pre-condition to your acceptance test.

line 5 The `When` clause expresses what is executed, the actual action that you test in this scenario.

line 6 The `Then` clause shows what are the postconditions (or effect) of the execution of the action above.

4.3 Run the (non implemented yet) scenario

After saving your file, you can run the `$ ./gradlew cucumber` task and should be facing a similar output as presented in Listing 4.

```
$ ./gradlew cucumber

> Task :app:cucumber FAILED

Scenario: AC1 - Create a game record with player, and number of guesses                                    # src/test/
     resources/gradle/cucumber/gamerecord.feature:3
  Given I have a player "John"
  When I create a game record with the player, word "crane", and 4 guesses
  Then The game record is created with the correct user, word, number of guesses, and the current timestamp

Undefined scenarios:
file:///.../uc-seng301-wordleapp-lab4/app/src/test/resources/gradle/cucumber/gamerecord.feature:3 # AC1 - Create a game
     record with player, and number of guesses

1 Scenarios (1 undefined)
3 Steps (2 skipped, 1 undefined)
0m0.118s

You can implement missing steps with the snippets below:

@Given("I have a player {string}")
public void i_have_a_player(String string) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}

@When("I create a game record with the player, word {string}, and {int} guesses")
public void i_create_a_game_record_with_the_player_word_and_guesses(String string, Integer int1) {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}

@Then("The game record is created with the correct user, word, number of guesses, and the current timestamp")
public void the_game_record_is_created_with_the_correct_user_word_number_of_guesses_and_the_current_timestamp() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}

[ ... trace truncated ... ]
```

Listing 4: Execution trace of `cucumber` task with unimplemented features.

As you can see, *Cucumber* warns you that it found a Scenario (line 10 and following), but that it is undefined. It also prints skeleton code (lines 19-35) that you will copy-paste into a Java class to implement (i.e. simulate) this acceptance criteria.

For every step present in your *Cucumber* test (i.e. any of `@Given`, `@When` or `@Then`), a method is created with its textual description inside the annotation. You can also see that elements defined between quotes are mapped to methods parameters (e.g.,line 19 where *"John"* has been transformed to a `String` parameter). Similar behaviour can be seen for integer values on line 25.

4.4 Creating the `Feature` class

To prepare the java class that will receive the skeleton code, you need to:

- Create a folder named `gradle` under `app/src/test/java`.
- Create a folder named `cucumber` under that new `gradle` folder.
- Create a Java class named `CreateNewGameRecordFeature.java`

You can now copy-paste the skeleton code from the `cucumber` task (lines 19-35 in Listing 4) into the Java class you created in previous step and rerun the `cucumber` task. You will see that now, *Cucumber* tells you that the scenario is *"Pending"* and not *"Undefined"* with a similar (truncated) trace as Listing 5.

```
$ ./gradlew cucumber

[ ... trace truncated ... ]

Pending scenarios:
file:///.../uc-seng301-wordleapp-lab4/app/src/test/resources/gradle/cucumber/gamerecord.feature:3 # AC1 - Create a game
     record with player, and number of guesses

1 Scenarios (1 pending)
3 Steps (2 skipped, 1 pending)
0m0.219s

[ ... trace truncated ... ]
```

Listing 5: Execution trace of `cucumber` task with unimplemented features.

4.5 Implement the acceptance test for AC1

You can now implement this acceptance test in a similar fashion as Listing 6. Imports have been left out, but are composed of `io.cucumber.java` subpackages or classes, `org.hibernate` (see lab 3), `org.junit.jupiter.api.Assertions`, your `model` classes and the new `Accessor` classes (from the code base you received).

```java
import java.util.Date;

public class CreateNewGameRecordFeature {

    private GameRecordAccessor gameRecordAccessor;
    private UserAccessor userAccessor;
    private User player;
    private GameRecord firstGameRecord;
    private Date timestampBefore;

    String gameRecordWord;
    int gameRecordNumGuesses;

    @Before
    public void setup() {
        Configuration configuration = new Configuration();
        configuration.configure();
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        userAccessor = new UserAccessor(sessionFactory);
        gameRecordAccessor = new GameRecordAccessor(sessionFactory);
    }

    @Given("I have a player {string}")
    public void i_have_a_player(String name) {
        player = new User();
        player.setUserName(name);
        Long playerId = userAccessor.persistUser(player);
        Assertions.assertNotNull(player);
        Assertions.assertNotNull(playerId);
        Assertions.assertSame(player.getUserName(), name);
    }

    @When("I create a game record with the player, word {string}, and {int} guesses")
    public void i_create_a_game_record_with_the_player_word_and_guesses(String word, Integer numGuesses) {
        gameRecordWord = word;
        gameRecordNumGuesses = numGuesses;
        timestampBefore = new Date();
        firstGameRecord = new GameRecord();
        firstGameRecord.setUser(player);
        firstGameRecord.setWord(word);
        firstGameRecord.setNumGuesses(numGuesses);
        Long firstGameRecordId = gameRecordAccessor.persistGameRecord(firstGameRecord);
```

```
43          Assertions.assertNotNull(firstGameRecord);
44          Assertions.assertNotNull(firstGameRecordId);
45      }
46
47      @Then("The game record is created with the correct user, word, number of guesses, and the current timestamp")
48      public void the_game_record_is_created_with_the_correct_user_word_number_of_guesses_and_the_current_timestamp() {
49          Assertions.assertTrue(!firstGameRecord.getTimestamp().before(timestampBefore) &&
50                  !firstGameRecord.getTimestamp().after(new Date()));
51          Assertions.assertEquals(firstGameRecord.getUser(), player);
52          Assertions.assertEquals(firstGameRecord.getWord(), gameRecordWord);
53          Assertions.assertEquals(firstGameRecord.getNumGuesses(), gameRecordNumGuesses);
54      }
55  }
```

Listing 6: Code snippet of the implementation of the acceptance test for AC1 in *Cucumber*.

lines 14-21 you can declare some code that will be executed before running the `feature` class (i.e. the scenario). These are called hooks[6] and have a similar semantics to the ones from *JUnit*. In this part, we need to instantiate the *Hibernate* features. It's important to note that the `@Before` annotation is from the `io.cucumber` package.

line 28 *Cucumber* reuses the assertion mechanisms from *JUnit*, i.e. from `org.junit.jupiter.api`.

lines 24,34 Note that, for readability reasons, we have renamed the default names for parameters to meaningful names and **we expect you to keep this readability habit**.

4.6 Run the full scenario

By running the above scenario with `$ gradlew cucumber`, you should have a similar trace as Listing 7

```
[ ... trace with hibernate startup logs truncated ... ]
Given I have a player "John"                                                                    # gradle.cucumber.C
reateNewGameRecordFeature.i_have_a_player(java.lang.String)
  When I create a game record with the player, word "crane", and 4 guesses                        # gradle.cucumber
      .C
reateNewGameRecordFeature.i_create_a_game_record_with_the_player_word_and_guesses(java.lang.String,java.lang.Integer)
  Then The game record is created with the correct user, word, number of guesses, and the current timestamp # gradle.cucumber
      .C
reateNewGameRecordFeature.the_game_record_is_created_with_the_correct_user_word_number_of_guesses_and_the_current_timestamp()

1 Scenarios (1 passed)
3 Steps (3 passed)
0m1.321s

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts
     or plugins.
See https://docs.gradle.org/7.4/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 4s
9 actionable tasks: 9 executed
```

Listing 7: Output trace from *Cucumber* with passing scenario.

5 It's your turn now

5.1 Implement U1 AC2

You are required to write the acceptance test code followed by the implementation code for below scenario, reproduced in Listing 8 that you will add into your `gamerecord.feature` file.

[6] See `https://cucumber.io/docs/cucumber/api/#hooks`

```
Scenario:  AC2 - Create a second game record with player, and number of guesses
  Given I have a player "John"
  And I have already created a game record with the player, word "crane", and 4 guesses
  When I create a game record with the player, word "crane", and 4 guesses
  Then The game record is created with the correct user, word, number of guesses, and the current timestamp
  And there are now two game records with identical values except timestamp
```

Listing 8: Second acceptance test scenario (U1 - AC2).

Follow the same step-by-step approach as described from Section 4.3. Because we want you to work in a TDD-way (write tests first), you may need to refactor the existing code in `GameRecordAccessor` to make it modular and reuse part of the logic in there. We also introduce Gherkin's `And` to combine multiple steps.

5.2 Write acceptance tests for other U1 ACs

Now, you are required to write the scenarios for the other ACs and implement them.