Lab 5 - Advanced JPA, Logging and Mockito

SENG301 Software Engineering II

Sam Burtenshaw Neville Churcher Morgan English Fabian Gilson
Jack Patterson

26th March 2022

Learning objectives

This week's lab aims to introduce you to *Mockito*[1] mocking framework. By the end of this lab, you should be able to:

- Understand the purpose of mocking and stubbing when writing tests;
- Understand the basic functionality of *Mockito* testing framework;
- Implement a test class using *Mockito*.

Last, this lab will illustrate how to use loggers (with *Apache log4j2*[2]):

- Show how to set up multiple types of loggers for multiple purposes;
- How to use loggers inside the code at different levels (e.g., to print debug or error traces).

1 Introduction

When developing larger scale software systems, full usage scenarios may require the interactions of different components or classes, such as REST APIs. Mock objects are useful to write automated (user acceptance) tests where some involved parties can be substituted by default implementations (i.e. stubs) that are *mocking* an actual behaviour. This enables to isolate some behaviour of specific classes within test scenarios and therefore create self-contained tests (cfr. Lecture 8).

A concrete example is when creating unit/acceptance tests, you do not want to rely on external resources, e.g., REST API, that may or may not reply or that may require API keys to be set up (and sometimes paid for). This is even more critical in an automated context (i.e. Continuous Integration) when tests may be run thousands of times. So, you usually want to avoid calling third party APIs within unit/acceptance tests where failures to fetch data from these APIs would make your pipeline fail unexpectedly.

Mocking can also be used in a Test-Driven Development loop while developing a feature where part of the behaviour of a succession of methods can be mocked and therefore tested without having the full implementation ready yet.

Mockito, is one of the available mocking tools. There are a number of other available frameworks, such as *jMock*[3]. We have chosen to use *Mockito* due to its wide community support and compatibility with *JUnit*, *Cucumber* and *Spring*.

[1] See `https://site.mockito.org/`
[2] See `https://logging.apache.org/log4j/2.x/`
[3] See `http://jmock.org`

2 Domain, story and acceptance criteria

This Lab will build on the example case used in *Lab 3* and *Lab 4*. Please refer to these handouts if need be. We reproduce an updated domain model in Figure 2 where we introduce new entities for fetching words from an api, the `Dictionary...` classes.

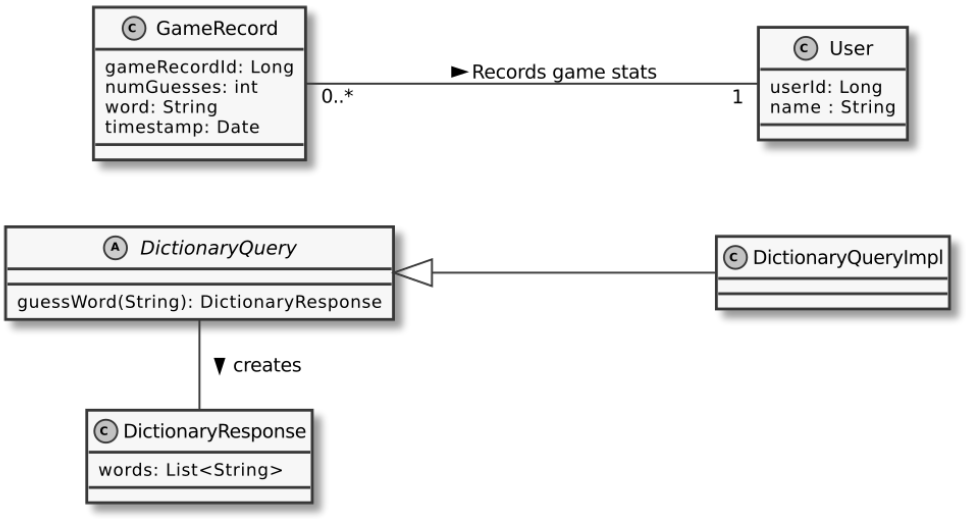

Figure 1: Domain model of a Wordle Clone App (updated from lab 4)

2.1 User stories

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

U2 As a player, I want to retrieve possible guesses so that I can see possible options.
- AC.1 I can get a list of possible words given the letters I know.
- AC.2 I can get a list of all possible words.
- AC.3 If no words match given the letters I know none will be returned.

3 Set up *Mockito* with *Gradle* (and other dependencies for this lab)

Next to this handout, you will find a `.zip` archive with an updated version of the expected result from lab 4 (including the test of U1-AC2). In a nutshell, the project is structured as follows:

`main` contains the sources of your project
- **`java`** java source files including the `model`, `accessor`, `util` and `location` packages (API calls)
- **`resources`** application configuration files (i.e. `hibernate.cfg.xml` with *sqlite* and `log4j2.xml`)

`test` contains all test sources
- **`java`** java test sources (including the `feature` and class files from *Lab 4*)
- **`resources`** test configuration files (i.e. `cucumber.properties` and `hibernate.cfg.xml` with *h2*)

`build.gradle` the dependencies configuration file

3.1 Initial `gradle` file

```
plugins {
```

```
    // Apply the application plugin to add support for building a CLI application in Java.
    id 'application'
}

repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
    // Use JUnit Jupiter for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.1'

    // Use JUnit Jupiter Engine for testing.
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine'

    // This dependency is used by the application.
    implementation 'com.google.guava:guava:30.1.1-jre'

    // use hibernate to persist (java) domain entities for us, aka JPA implementation
    implementation 'org.hibernate:hibernate-core:5.6.5.Final'

    // Sqlite as persistent DB (SENG301 lab 3)
    implementation 'org.xerial:sqlite-jdbc:3.36.0.3'
    implementation 'com.zsoltfabok:sqlite-dialect:1.0'

    // use a in-memory database to store entities (can be substituted with any database)
    implementation 'com.h2database:h2:2.1.210'

    //Cucumber dependencies
    testImplementation 'io.cucumber:cucumber-java:7.2.3'

    // Logging facility (SENG301 lecture 9 and lab 5)
    implementation 'org.apache.logging.log4j:log4j-core:2.17.0'

    // JSON deserialisation (for external REST API, lab 5)
    implementation group: 'com.fasterxml.jackson.core', name: 'jackson-core', version: '2.12.2'
    implementation group: 'com.fasterxml.jackson.core', name: 'jackson-annotations', version: '2.12.2'
    implementation group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: '2.12.2'
}

run {
    // this is needed if you want to start your main class with gradle run
    // you may pass the --console=PLAIN option to disable the green progress bar
    // e.g., gradlew run --console=PLAIN
    standardInput = System.in
}

application {
    // Define the main class for the application.
    mainClass = 'uc.seng301.wordleapp.lab5.App'
}

configurations {
    cucumberRuntime {
        extendsFrom testImplementation
    }
}

sourceSets {
    test {
        resources.srcDirs = ['src/test/resources']
        java.srcDirs = ['src/test/java', 'src/main/java']
        runtimeClasspath = project.sourceSets.main.compileClasspath +
                project.sourceSets.test.compileClasspath +
                fileTree("src/test/resources/test") +
                project.sourceSets.test.output + project.sourceSets.main.output
```

```
69      }
70  }
71
72  task cucumber() {
73      dependsOn assemble, testClasses
74      doLast {
75          javaexec {
76              main = "io.cucumber.core.cli.Main"
77              classpath = configurations.cucumberRuntime  + sourceSets.test.output
78              args = ['--plugin', 'pretty', '--glue', 'gradle.cucumber', 'src/test/resources']
79          }
80      }
81  }
82
83  tasks.named('test') {
84
85      // Use JUnit Platform for unit tests.
86      useJUnitPlatform()
87      // force the test task to run Cucumber too
88      finalizedBy cucumber
89  }
```

Listing 1: Initial `build.gradle` file with dependencies from *Lab 3-4*, logging feature and *Jackson* JSON handling for REST API.

From Listing 1, you can identify *Hibernate* (line 22), *sqlite* (lines 25-26), *h2* (line 29) and *Cucumber* (line 32) from previous labs. In this lab, we also introduced *Log4j2* (line 35) and JSON handling for the REST API we will use in this lab (lines 38-40).

3.2 Using an API

For this lab we will be using an API `https://seng301.csse.canterbury.ac.nz` to fetch possible worlde guesses based on letters we know. We can simply make a request to the `/solver` endpoint with a string of characters where any character we do not know is replaced by a period `'.'`. For example we can make the request `/solver/as...` and the server will respond with the words "crane" and "crank". We can also use this api to fetch all the words it has by simply providing an all unknown request to the endpoint `/solver/.....`. Feel free to try this out in your browser.

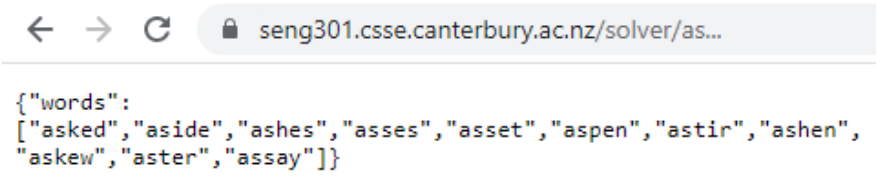

Figure 2: Example API request in browser

An alternative way is to use `curl`, a lightweight command line tool to transfer data over HTTP. See listing 2 for an example.

```
1  $ curl https://seng301.csse.canterbury.ac.nz/solver/as...
2  {"words":["asked","aside","ashes","asses","asset","aspen","astir","ashen","askew","aster","assay"]}
```

Listing 2: `GET` guesses through the wordle API with `curl`.

3.3 Running the application

How the application runs has been changed compared to the previous labs, now instead of simply running and receiving an ouptut you will have to interact with the application and 'play' the game yourself. To start, build (`$`

`./gradlew build`) and run (`$ ./gradlew run --console=plain`) the project to see what this programme does. Note that we have added `--console=plain` to the command as parameters. This removes the normal gradle logs so that we can more easily interact with our application through the command line. A similar trace as Listing 3 should be visible in your terminal. Have a little play around with the application to understand how it works.

```
1  ######################################################
2  Welcome to Wordle Clone App
3  ######################################################
4  Please enter your name, or type !q to exit:
5  Morgan
6  Please enter your guess (must be 5 letters), or use "help cran." to view possible guesses:
7  crane
8  crane
9  Please enter your guess (must be 5 letters), or use "help cran." to view possible guesses:
10 light
11 light
12 Please enter your guess (must be 5 letters), or use "help cran." to view possible guesses:
13 gully
14 gully
15 Solved in 3 guesses
16 High Scores:
17 1: GameRecord{gameRecordId=2, user=User{userId=1, userName='Morgan', gameRecordIds=[2]}, word='gully', numGuesses=3,
       timestamp=2022-03-22 16:08:52.17}
18 Please enter your name, or type !q to exit:
```

Listing 3: Execution trace of `App.java`.

You may notice repeated words (e.g., crane on lines 7 and 8). Line 7 is an input with line 8 printing out the word colour-coded following Wordle rules. If you do not see this colour-coding you may be in a terminal that does not support ANSI colour encoding, so try another such as the one built into IntelliJ.

For a refresher, the colour coding rules are as follows:

- A letter printed as green means this letter is in the word, and is in the correct place.
- A letter printed in yellow/orange means this letter is in the word, but not in the correct place.
- A letter printed in gray means this letter is not in the word.

4 Logging

Logging is an important aspect of software, that is often over-looked by inexperienced developers. Logging allows us to "log" or record messages during operation to a file. Up until this point you may have simply used print statements, however these come with drawbacks:

- It often isn't as time efficient to add print statements when compared to debugging with IDE integrated tools.
- Printing a lot of information that is not easily identifiable and doesn't follow strict patterns can be hard to understand, lack much needed information for debugging purposes, and be hard to search efficiently.
- When delivering a product to the user we don't want them to be overwhelmed with messages printing to the console (or similar) that they do not understand.

So how does logging help, you may ask?

- We can easily specify more information, such as a time stamp, the current class or method, a level of criticality, and more.
- The use of a file (with a specific format such as json) means that it is easier to search for or filter specific messages (often an important ability is to filter by criticality).
- Files also make it easier to find issues in production. It is much easier to retrieve files from your production server or instance. You may have experienced this in practice where upon an application crash a window asks you to (or automatically) selects a log file to send to the developers so they can better understand the bug and what caused it.
- Logs may also be sent to multiple outputs including sockets, ensuring that developers can have access to the

logs for a corrupted or shutdown machine.

In this project we make use of Log4J2, which you may have heard of over recent security flaw[4], however this issue has since been patched in newer versions. To set up our logger we define a configuration file in our `main/resources` folder, for this example we will use xml but there are several ways to do this.

The configuration file will not be discussed in detail but there are many comments, including commented out configurations so that you should take the time to look into it. However it is important that we know where the output is going.

Within the `log` folder several files will be produced after you run the application

- `app.log` - This records the logging statements we add in our code (linked to the `applog` *RollingFile*[5] appender, i.e. logger instance).
- `other.log` - This records all other log statements, often filled by logs from other libraries (linked to the `allother` *RollingFile* appender, i.e. logger instance).
- `sql.log` - This records all of the sql related actions done by hibernate. Note that this is very bad practice in production, as it may log sensitive data[6] (linked to the `sqllog` *RollingFile* appender, i.e. logger instance).

Note: To help your work on the codebase we have included a log statement that records the word which is randomly selected so you don't actually have to play the game each time. Try open `app.log` and see if you can find these log statements.

5 Using *Mockito* to mock external APIs

5.1 Update `build.gradle` file

Add below dependency under the `dependencies` clause of your `build.gradle` file.

```
implementation 'org.mockito:mockito-core:4.3.1'
```

Listing 4: *Mockito* dependency.

5.2 Create a new `feature` file for the new ACs

As you did in *Lab 4*, create a new `guesses.feature` file that will contain the acceptance criteria reproduced in Listing 5 for the story from Section 2.1. Place this file under `src/test/resources/gradle/cucumber`.

```
Feature: U2 - As a player, I want to retrieve possible guesses so that I can see possible options
  Scenario: AC1 - I can get a list of possible words given the letters I know
    Given I know the letters "th..." of the word
    When I get a list of possible words
    Then The available options are returned with only those that match

  Scenario: AC2 - I can get a list of all possible words
    Given I know the letters "" of the word
    When I get a list of possible words
    Then All available words are returned

  Scenario: AC3 - If there are no words that match none are returned
    Given I know the letters "aaaaa" of the word
    When I get a list of possible words
    Then No words are returned
```

Listing 5: Acceptance criteria for U2 in *Gherkin* (Cucumber) syntax

[4] See https://theconversation.com/what-is-log4j-a-cybersecurity-expert-explains-the-latest-internet-vulnerability-how-bad-it-is-and-whats-at-stake-173896

[5] See https://howtodoinjava.com/log4j2/log4j2-rollingfileappender-example/

[6] See https://twitter.com/TwitterSupport/status/992132808192634881

You can run the *Cucumber* task (`$ ./gradlew cucumber`) to generate the methods' skeletons. You will notice that because we declared two similar steps, two identical methods will be created. You will need to remove one manually as we only need to implement it once (i.e. *Cucumber* allows to reuse execution steps across different `Scenario`, making your tests modular). Copy this code into a new test class under `src/test/java/gradle/cucumber`, similarly as you did in *Lab 4*,

5.3 Implement AC1

The implementation of AC1 should be rather easy as it is very similar to *Lab 4*. Give it a go before copying below code (only the steps code is reproduced, but you need a similar `setup()` method annotated with `@Before`, as in *Lab 4*). You also need to declare the right fields (e.g., `dictionaryQuery`) and import the right classes e.g., `io.cucumber.java.en.When`)

```
public class GetGuessesFeature {
    private DictionaryQuery mockDictionaryQuery;
    private String known;
    private List<String> words;
    private final List<String> allWords = new ArrayList<>(Arrays.asList("which", "there", "their", "about", "would", "these",
          "other", "words", "could", "write"));
    @Before
    public void setup() {
        mockDictionaryQuery = Mockito.mock(DictionaryQuery.class);
        words = null;
    }

    @Given("I know the letters {string} of the word")
    public void i_know_the_letters_of_the_word(String known) {
        Pattern pattern = Pattern.compile(known);
        DictionaryResponse dictionaryResponse = new DictionaryResponse();
        dictionaryResponse.setWords(allWords.stream().filter(pattern.asPredicate()).collect(Collectors.toList()));
        Mockito.when(mockDictionaryQuery.guessWord(known)).thenReturn(dictionaryResponse);
        this.known = known;
    }

    @When("I get a list of possible words")
    public void i_get_a_list_of_possible_words() {
        words = mockDictionaryQuery.guessWord(known).getWords();
        Assertions.assertNotNull(words);
    }

    @Then("The available options are returned with only those that match")
    public void the_available_options_are_returned_with_only_those_that_match() {
        words = mockDictionaryQuery.guessWord(known).getWords();
        Assertions.assertNotNull(words);
        Assertions.assertEquals(words, allWords.stream().filter(s -> s.matches(known)).collect(Collectors.toList()));
    }
}
```

Listing 6: Acceptance testing code for AC1.

5.4 Using *Mockito* to fake API calls

We implemented a simple HTTP client in `DictionaryQueryImpl` that will suggest words for you (i.e. the service behind the `help` command). Take a few minutes to look into that class.

Mockito can supply default result values for any methods that it is *"stubbing"*, i.e. faking the implementation for. This is particularly useful in our example if you don't want to call the API each time in your tests. To this end, you need to:

- have a java `interface` for the class you want to mock (as we do for `DictionaryQuery`)
- declare an instance of this `interface` using *Mockito.mock()* special feature

In Listing 7, we show how to declare a mocked `DictionaryQuery` implementation. Below code should be placed in the `setup()` method of your test class. Do not forget to actually declare the `DictionaryQuery` instance as a class field to call it from the related scenario step.

```
1 DictionaryQuery dictionaryQuery = Mockito.mock(DictionaryQuery.class);
```

Listing 7: *"Stub"* a class with *Mockito*.

The second step is to define our return object that our Mocked class will return. In this example we use patterns to reduce a list of words to only those that match.

```
1 Pattern pattern = Pattern.compile(known);
2 DictionaryResponse dictionaryResponse = new DictionaryResponse();
3 dictionaryResponse.setWords(allWords.stream().filter(pattern.asPredicate()).collect(Collectors.toList()));
```

Listing 8: A simple return object to be used in mocking

The final step is to tell *Mockito* what methods you ask it to send fake replies for and what reply to give, as shown in Listing 8 and Listing 9. As for Listing 7, you should declare these *"stubbing"* statements in the `setup()` method of your test class. You can define as many fake replies as you like for as many methods as you need.

```
1 Mockito
2   .when(dictionaryQuery.guessWord(known))
3   .thenReturn(dictionaryResponse);
```

Listing 9: Supply default result values for methods with *Mockito*.

As can be seen from line 2 above, *Mockito* even allows to specify what input values it needs to supply fake replies for. In our case, the scenario in the `feature` file described in Listing 5 will look for words matching "cran." as a location (line 3), so we supply a fake response for that input value. Note that other input values would return `null` as no fake replies have been set up with *Mockito*.

In your test code, you will be able to call the mocked method as you would normally do in your code. Note that you need to capture the result of the mock `DictionaryQuery` call into the dedicated `DictionaryResult` object and keep these values in class fields to test their equality with what is expected.

5.5 Implement the remaining steps

Your last task is to implement the remaining steps from AC2 to make the full acceptance test pass.

6 To go further

As an interesting additional resource to go further with *Mockito* with depenpendency injection, for mocking REST calls in Spring (SENG302), you are encouraged to look at:

- `https://www.baeldung.com/mockito-annotations`
- `https://www.baeldung.com/spring-mock-rest-template`
- `https://github.com/Jet-C/spring-demo`