TQS Lab activities

v2021-04-12

Lab 1: Unit testing with JUnit 5

Learning objectives

- Identify relevant unit tests to verify the contract of a module.
- Write and execute unit tests using the JUnit framework.
- Link the unit tests results with further analysis tools (e.g.: code coverage)

Key points

- Unit testing is when you (as a programmer) write test code to verify units of (production) code. A unit represents a small subset of a much larger solution. A true “unit” does not have depend of the behavior of other (collaborating) components.
- Unit tests help the developers to (i) understand the module contract (what to construct); (ii) document the intended use of a component; (iii) prevent regression errors; (iv) increase confidence on the code.
- When following a TDD approach, typically you go through a cycle of Red-Green-Refactor. You’ll run a test, see it fail (go red), implement the simplest code to make the test pass (go green), and then refactor the code so your test stays green and your code is sufficiently clean.
- JUnit and TestNG are popular frameworks for unit testing in Java.

JUnit best practices: unit test one object at a time

A vital aspect of unit tests is that they're finely grained. A unit test independently examines each object you create, so that you can isolate problems as soon as they occur. If you put more than one object under test, you can't predict how the objects will interact when changes occur to one or the other. When an object interacts with other complex objects, you can surround the object under test with predictable test objects. Another form of software test, integration testing, examines how working objects interact with each other. See chapter 4 for more about other types

Lab activities

Be sure that your developer environment meets the following requirements:

- Java development environment (JDK; v11 suggested). Note that you should install it into a path without spaces or special characters (e.g.: avoid \Users\José Conceição\Java).
- Maven configured to run in the command line. Note: some projects include the Maven wrapper utility (*mvnw*); in these cases, Maven wrapper will download maven as needed.
- Java capable IDE, such as IntelliJ IDEA (version "Ultimate" suggested).

1.1

In this exercise, you will implement a stack data structure (TqsStack) with appropriate unit tests. Be sure to adopt a **write-the-tests-first** workflow:

a) Create a new **Maven-based,** Java standard application.

b) Add the required dependencies to run Junit tests[1]. Here are some examples:
   - JUnit documentation
   - starter project for Maven[2].

c) Create the required classes definition (**just the "skeleton"**, do not implement the methods body yet; you may need to add dummy return values). The **code should compile**, though the **implementation is incomplete** yet.

d) Write the unit tests that will verify the TqsStack contract.

   You may use the IDE features to generate the testing class; note that the IDE support will vary. Be sure to use JUnit 5.x.

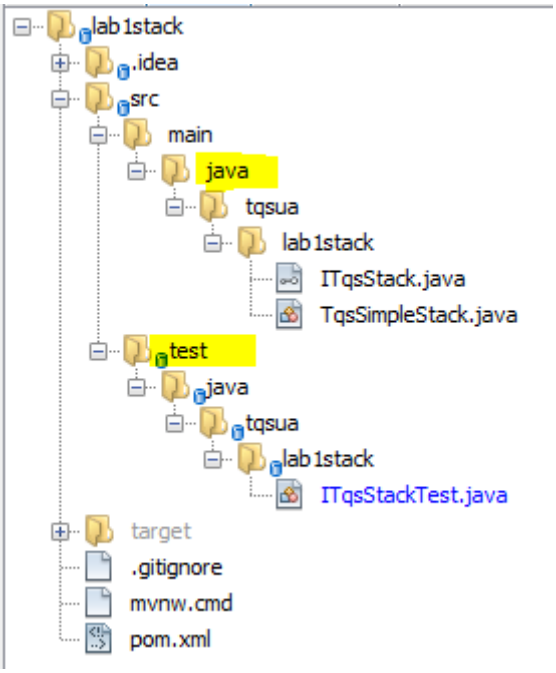
   Your tests will verify several assertions that should evaluate to true for the test to pass. See some examples.

e) Run the tests and prove that TqsStack implementation is not valid yet (the tests should fail for now, the first step in Red-Green-Refactor).

f) Correct/add the missing implementation to the TqsStack;

g) Run the unit tests.

h) Iterate from steps d) to f) and confirm that all tests pass.



Suggested stack contract:

- push(x): add an item on the top
- pop: remove the item at the top
- peek: return the item at the top (without removing it)

[1] If using IntelliJ: you may skip this step and ask, later, the IDE to fix JUnit imports.
[2] Delete the "pom-JITPACK.xml" and "pom-SNAPSHOT.xml", specially before importing into an IDE.

- size: return the number of items in the stack
- isEmpty: return whether the stack has no items

What to test[3]:

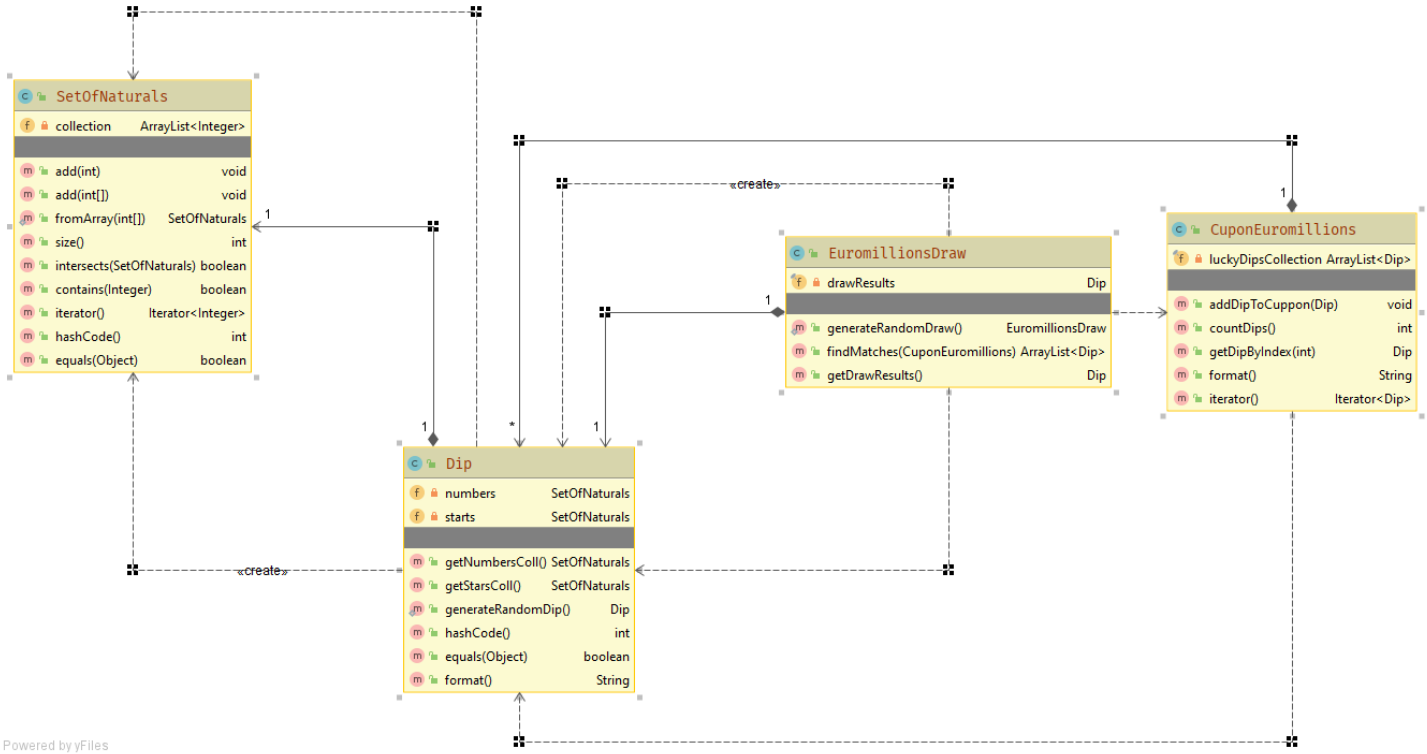
a) A stack is empty on construction.
b) A stack has size 0 on construction.
c) After n pushes to an empty stack, n > 0, the stack is not empty and its size is n
d) If one pushes x then pops, the value popped is x.
e) If one pushes x then peeks, the value returned is x, but the size stays the same
f) If the size is n, then after n pops, the stack is empty and has a size 0
g) Popping from an empty stack does throw a NoSuchElementException [You should test for the Exception occurrence]
h) Peeking into an empty stack does throw a NoSuchElementException
i) For bounded stacks only, pushing onto a full stack does throw an IllegalStateException

2a/ Pull the "euromillions-play" project and correct the code (or the tests themselves, if needed) to have the existing unit tests passing.

For the (failing) test:	**You should:**
testFormat	Correct the implementation of Dip#format so the tests pass.
testConstructorFromBadArrays	Implement new test logic to confirm that an exception will be raised if the arrays have invalid numbers (wrong count of numbers of starts)

Note: you may suspend temporary a test with the @Disabled tag (useful while debugging the tests themselves).

[3] Adapted from http://cs.lmu.edu/~ray/notes/stacks/



2b/ The class SetOfNaturals represents a set (no duplicates should be allowed) of integers, in the range [1, +∞]. Some basic operations are available (add element, find the intersection…).
What kind of unit test are worth writing for the entity SetOfNaturals?
Complete the project, adding the new tests you identified.

2c/ Note that the code provided includes "magic numbers" (2 for the number of stars, 50 for the max range,…). Refactor the code to extract constants and eliminate the "magic numbers".

2d/ Assess the coverage level in project "Euromillions-play".
Configure the maven project to run Jacoco analysis.
Run the maven "test" goal and then "jacoco:report" goal. You should get an HTML report under target/jacoco.
Interpret the results accordingly. Which classes/methods offer less coverage? Are all possible decision branches being covered?

Note: IntelliJ has an integrated option to run the tests with the coverage checks (without setting the Jacoco plugin in POM). But if you do it at Maven level, you can use this feature in multiple tools.

Troubleshooting some frequent errors

➔ "Test are run from the IDE but not from command line."
Be sure to configure the Surefire plug-in in Maven (example).

Explore

- JetBrains Blog on Writing JUnit 5 tests (with vídeo).
- JUnit 5 cheat sheet.
- Book: JUnit in Action. Note that you can access it from the OReilly on-line library, using your University of Aveiro's user account.

- Vogel's tutorial on JUnit. Useful to compare between JUnit 4 and JUnit 5.
- Working effectively with unit testing (podcast).

Lab 2: Mocking dependencies (for unit testing)

Learning objectives

— Prepare a project to run unit tests (JUnit 5) and mocks (Mockito 3.x), with mocks injection (@Mock).

— Write and execute unit tests with mocked dependencies.

— Experiment with mock behaviors: strict/lenient verifications, advanced verifications, etc.

Lab activities

Get familiar with sections 1 to 3 in the Mockito (Javadoc) documentation.

1a/
Consider the example in Figure 1: the StocksPortfolio holds a collection of Stocks; the current value of the *portfolio* depends on the current condition of the *Stocksmarket*. **StockPortfolio#getTotalValue()** method calculates the value of the portfolio by summing the current value (looked up in the stock market) of the owned stocks.

Implement (at least) one test to verify the implementation of **StockPortfolio#getTotalValue().** Given that test should have predictable results, you need to address the problem of having non-deterministic answers from the Stockmarket interface.

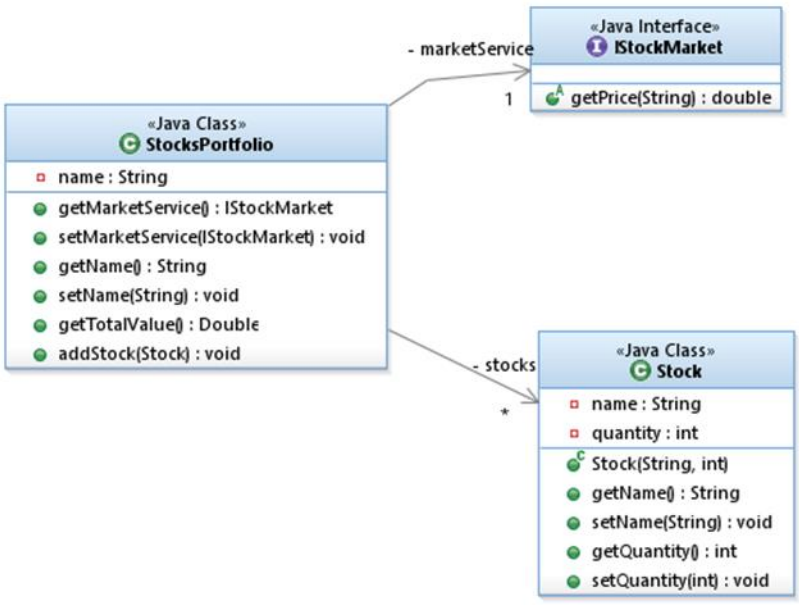


Figure 1: Classes for the StocksPortfolio use case.

a) Create the classes. You may write the implementation of the services before or after the tests.

b) Create the test for the getTotalValue(). As a guideline, you may adopt this outline:

```
1. Prepare a mock to substitute the remote service (@Mock annotation)
2. Create an instance of the subject under test (SuT) and use the mock to set the
   (remote) service instance.
```

```
3. Load the mock with the proper expectations (when...thenReturn)
4. Execute the test (use the service in the SuT)
5. Verify the result (assert) and the use of the mock (verify)
```

Notes:

- Consider use these Maven dependencies for your POM (JUnit5, Mockito).
- Mind the JUnit version. For JUnit 5, you should use the @ExtendWith annotation to integrate the Mockito framework.

```
@ExtendWith(MockitoExtension.class)
class StocksPortfolioTest { … }
```

- See a quick reference of Mockito syntax and operations.

1b/ Instead of the JUnit core asserts, you may use the Hamcrest library to create more human-readable assertions. Replace the "Assert" statements in the previous example, to use Hamcrest constructs. E.g.:

```
assertThat(result, is(14.0));
```

2/ Consider an application that needs to perform reverse geocoding to find a zip code for a given set of GPS coordinates. This service can found in public API (e.g.: using the MapQuest API).

a) Create the objects represented in Figure 1. TqsHttpClient represents a service to initiate HTTP requests to remote servers. At this point, **you do not need to implement TqsHttpBasic**; in fact, you should provide a substitute for it.

b) Consider that we want to verify the AddressResolver#findAddressForLocation, which invokes a remote geocoding service, available in a REST interface, passing the site coordinates.

Which is the service to *mock*?

c) To create a test for findAddressForLocation, you will need to know **the exact response of the geocoding service for a sample request**. Assume that we will use the MapQuest API. Use the browser or an HTTP client to try some samples so you know what to test for (example 1).

d) Implement a test for AddressResolver#findAddressForLocation using a mock.

e) Besides de "success" case, consider also testing for alternatives (e.g.: no address found; bad coordinates should not be accepted;…). This may affect the TqsHttpClient or other classes.

This getting started project [gs-mockForHttpClient] can be used in your implementation.

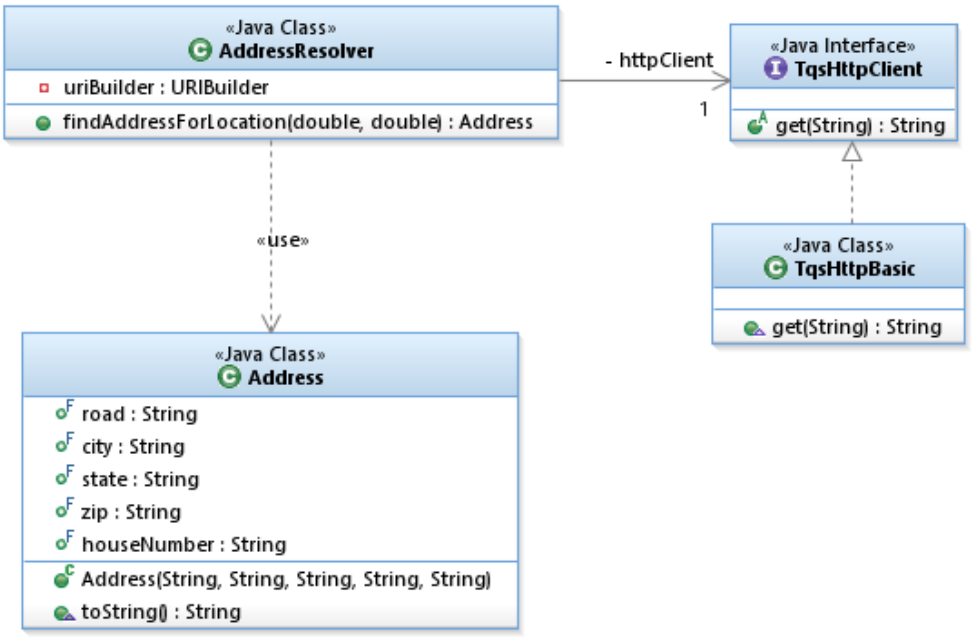


Figure 2: Classes for the geocoding use case.

3/ Consider you are implementing an integration test, and, in this case, you would use the real implementation of the module, not the mocks, in the test.
(This section can be included with the previous, continuing the same project.)

Create new test class (or reuse the existing AddressResolver**IT**), in a separate package, and be sure its name end with "**IT**".
Copy the tests from the previous exercise into this new test class, but remove any support for mocking (no Mockito imports in this test).
Correct the test implementation so it uses the real HttpClient implementation.
Run your test (and confirm that the remote API is invoked in the test execution).

If the "failsafe" maven plugin is configured, you should get different results with:
$ mvn test
$ mvn install failsafe:integration-test

Explore

- There is a recent book on JUnit and Mockito available from OReilly. The lessons are available as short videos too.

Lab 3: Acceptance testing with web automation

Learning objetives

- use web automation interactively with Selenium IDE.
- run web-based acceptance tests written for the WebDriver API, using the JUnit engine.
- apply the Page Objects Pattern to increase tests readability.

Key Points

- Acceptance tests (or functional test) exercise the user interface of the system, as if a real user was using the application. The system is treated as a black box.
- Browser automation (control the browser interaction from a script) is an essential step to implement acceptance tests on web applications. There are several frameworks for browser automation (e.g.: Puppeteer); for Java, the most used framework is the WebDriver API, provided by Selenium (that can be used with JUnit or TestNG engines).
- Selenium is an umbrella project for a range of tools and libraries that enable and support the automation of web browsers.
- The test script can easily get "messy" and hard to read. To improve the code (and its maintainability) we could apply the Page Objects Pattern.
- Web browser automation is also very handy to implement “smoke tests”.

Lab

Selenium works with multiple browsers but, for sake of simplicity, the samples will be discussed with respect to Firefox; you may use Chrome/Chromium as well.

Suggested setup:

a) Download the GeckoDriver for Firefox (ChromeDriver for Chrome/Chromium) and make sure it is available in the system PATH.

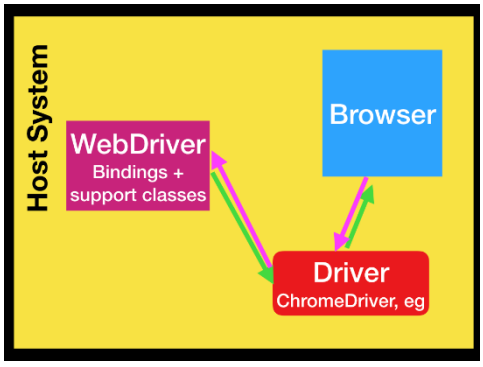
b) Install the “Selenium IDE” browser plugin. (or, alternatively, the Katalon Recorder).

In this lab:

1. Run web automation tests with JUnit 5 + Selenium 3
2. Create a web automation with Selenium IDE recorder
3. Run the test as a Java project (JUnit 5 + Selenium)
4. Refactor using the Web Page Object pattern

1/ Run web automation tests with WebDriver (locally)

Selenium WebDriver offers a concise programming interface (i.e., API) to drive a (web) browser, as if a real user is operating the browser.



This example presents a minimal scenario for running a WebDriver based test[4].

Note that you need:
- the browser installed locally;
- the web driver implementation for your browser copied into your filesystem(e.g.: GeckoDriver , ChromeDriver)

1a/
Run the example, adapting for your IDE.
Note: while the example explicitly defines the location of the driver property, it is possible to omit it if the driver implementation in in the system PATH.

1b/

[4] The tutorial uses Eclipse. You do not need to use Eclipse!

Refactor to use explicit init and clean up test methods to open and close the browser:

- @BeforeEach : init the driver.
- @AfterEach: close the browser.

While automating the interaction, you will often need to:

- Navigate (go to a page, go back, quit,…)
- Find elements in a page (e.g.: move to a text box)

2/ Create a web automation with Selenium IDE recorder

Usually, you can take advantage of the Selenium IDE to prepare/record your tests interactively and to explore the "locators" (e.g.: id for a given web element).

2a/ Record the test interactively
Install the Selenium IDE plug-in/add-on for your browser.

Using the https://blazedemo.com/ dummy travel agency web app, record a test in which you select a and buy a trip.
Be sure to add relevant "asserts" or verification to your test.
Replay the test (for success) but also experiment to break the test (by explicitly editing the test step parameters).

Add a new step, at the end, to confirm that the confirmation page contains the title "BlazeDemo Confirmation". Enter this assertion "manually" (in the editor, but not recording).

Be sure to save you Selenium IDE test project.

2b/ Export and run the test (Webdriver)
Export the test from Selenium IDE into a Java test class and include it in the previous project.
Refactor the generated code to be compliant with JUnit 5. [Note: adapt from exercise 1]

Run the test (programmatically, as a unit test).

3/ Refactor to use Selenium extensions for JUnit 5

JUnit 5 allows the use of extensions which may provide annotation for dependency injection (as seen previously for Mockito). This is usually a more compact and convenient approach.

The Selenium-Jupiter extension provides convenient defaults and dependencies resolution to run Selenium tests (WebDriver) on JUnit 5 engine.

Note that this library will ensure several tasks:

- transitively import the required Selenium dependencies. [you may just add the selenium-jupiter dependency in POM]
- enable dependency injection with respect to the WebDriver implementation (automates the use of WebDriverManager to resolve the specific browser implementation).
- if using dependency injection, it will also ensure that the WebDriver is initialized and closed.
- you do not need to pre-install the WebDriver binaries; they are retrieved on demand.

3a/
Browse the “Quick reference” and the “Local browsers” sections from the documentation. Implement the sample under the appropriate (local) browser for your case.

3b/
Create and additional test to use a “headless browser” (e.g.: HTMLUnit, PhantomJS).

3c/
Copy the example from exercise 2 and refactor to use the Selenium-Jupiter extension.

3d/
[Optional, if you have Docker in your system:] Consider also using a browser that is not installed in your system. You may resort to a Docker image very easily (see Docker browsers section).
Note that, in this case, the WebDriver will connect to a remote browser (no longer direct communication).

4/ Separation of concerns with Page Object pattern

Consider the example discussed here.
Implement the “Page object pattern” for a cleaner and more readable test, as suggested.

Notes:
The target web site implementation has changed from the time the article was written and the example requires some adaptations, including:

- instead of three pages, consider only two (“HomePage” and “DeveloperApplyPage”; no need for the intermediate step “DeveloperPortalPage”).
- Use Selenium IDE to interactively record the test case and “confirm” the web elements identifiers.

You may skip/comment the final “step” (application submission), as it may depend on a captcha, and you are not required to address it in this introductory-level…

Explore

- DZone RefCard for WebDriver API
- Puppeteer - a Node library which provides a high-level API to control headless Chrome/Chromium.
- Another Page Object Model example.
- Criticism on the Page Object Pattern for modern web apps (and alternatives).

Lab 4: Multi-layer application testing (with Spring Boot)

Prepare

This lab is based on Spring Boot. Most of students already used the Spring Boot framework (in IES course).
If you are new to Spring Boot, then you need to develop a basic understanding or collaborate with a colleague. Learning resources are available at the Spring site.

Key Points

- Isolate the functionality to be tested by limiting the context of loaded frameworks/components. For some use cases, you can even test with just standard unit testing.
- @SpringBootTest annotation loads whole application context, but it is better (faster) to limit application contexts only to a set of Spring components that participate in test scenario.
- @DataJpaTest only loads @Repository spring components, and will greatly improve performance by not loading @Service, @Controller, etc.
- Use @WebMvcTest to test Rest APIs exposed through Controllers. Beans used by controller need to be mocked.

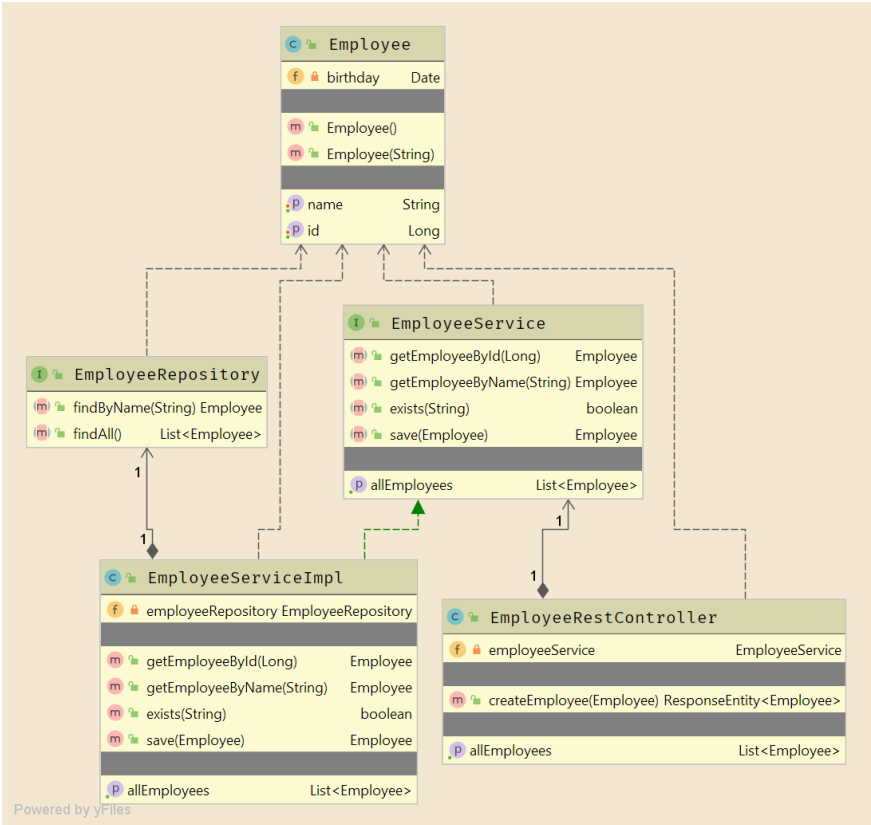
Lab

1/

Study the example available concerning a simplified Employee management application (gs-employee-manager).

This application follows commons practices to implement a Spring Boot solution:

- Employee: entity (@Entity) representing a domain concept.
- EmployeeRepository: the interface (@Repository) defining the data access methods on the target entity, based on the framework JpaRepository. “Standard” requests can be inferred and automatically supported by the framework (no additional implementation required).
- EmployeeService and EmployeeServiceImpl: define the interface and its implementation (@Service) of a service related to the “business logic” of the application. Elaborated decisions/algorithms, for example, would be implemented in this component.
- EmployeeRestController: the component that implements the REST-endpoint/boundary (@RestController): handles the HTTP requests and delegates to the EmployeeService.



The project contains a set of tests. Take note of the following test scenarios:

Purpose	Strategy	Notes
A/ Verify the data access services provided by the repository component. [*EmployeeRepositoryTest*]	Slice the test context to limit to the data instrumentation (@DataJpaTest) Inject a TestEntityManager to access the database; use this object directly to write to the database (no caches).	@DataJpaTest includes the @AutoConfigureTestDatabase. If a dependency to an embedded database is available, an in-memory database is set up. Be sure to include H2 in the POM.
B/ Verify the business logic associated with the services implementation. [*EmployeeService_UnitTest*]	Can be achieved with a unit tests; we can mock the repository behavior. Rely on Mockito to control the test and to set expectations and verifications.	Relying only in JUnit + Mockito makes the test a unit test, much faster that using a full SpringBootTest. No database involved.
C/ Verify the boundary components (controllers). No need to test the real HTTP-REST framework; just the controller behavior. [*EmployeeController_ WithMockServiceIT*]	Run the tests in a simplified light environment, simulating the behavior of an application server, by using @WebMvcTest mode. Get a reference to the server context with @MockMvc. To make the test more localized to the controller, you may mock the dependencies on the service (@MockBean); the repository component will not be involved.	MockMvc provides an entry point to server-side testing. Despite the name, is not related to Mockito. MockMvc provides an expressive API, in which methods chaining is expected. In principle, no database involved.
D/ Verify the boundary components (controllers). Test the REST API on the server-side; no API client involved. [*EmployeeRestControllerIT*]	Start the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal SpringBoot context. Use the entry point for server-side Spring MVC test support (MockMvc).	This would be a typical integration test in which several components will participate (the REST endpoint, the service implementation, the repository and the database).
E/ Verify the boundary components (controllers). Test the REST API with explicit HTTP client involved. [*EmployeeRestControllerTemplateIT*]	Start the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal SpringBoot context. Use a REST client to create realistic requests (TestRestTemplate)	Similar to the previous case, but instead of assessing a server entry point for tests, start a API client (so request and response un/marshaling will be involved).

Review questions: [answer in a **readme.md** file, in the appropriate folder]

a) Identify a couple of examples on the use of AssertJ expressive methods chaining.

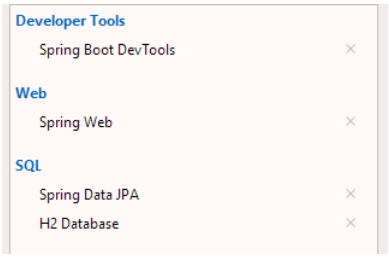
b) Identify an example in which you mock the behavior of the repository (and avoid involving a database).

c) What is the difference between standard @Mock and @MockBean?

d) What is the role of the file “application-integrationtest.properties”? In which conditions will it be used?

2/

Consider the case in which you will develop an API for a car information system.
Implement this scenario, as a Spring Boot application.
Consider using the Spring Boot Initializr to create the new project (integrated in IntelliJ or online);
Add the dependencies (*starters*) for: Developer Tools, Spring Web, Spring Data JPA and H2 Database.



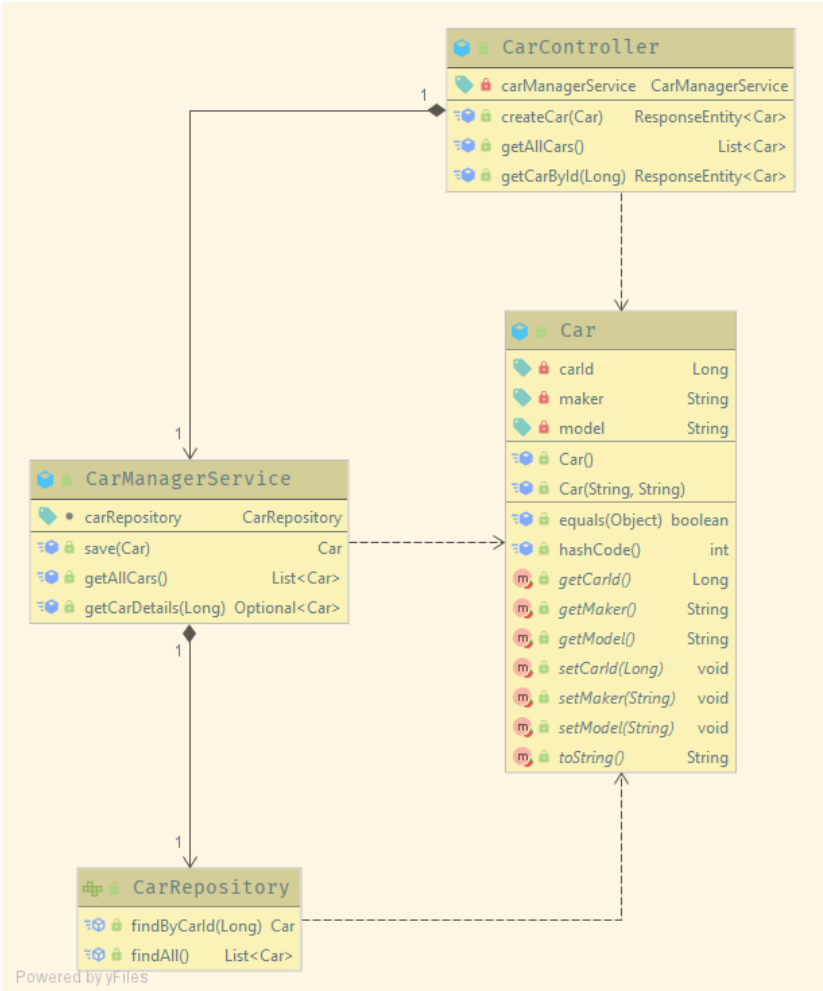
Take the structure modeled in the class diagram as a (minimal) reference.
In this exercise, **try to force a TDD approach**: write the test first; make sure the project can compile without errors; defer the actual implementation of production code as much as possible.
This approach will be encouraged if we try to write the tests in a top-down approach: **start from the controller, then the service, then the repository**.

a) Create a test to verify the Car[Rest]Controller (and mock the CarService bean). Run the test.

b) Create a test to verify the CarService (and mock the CarRepository). This can be a standard unit test with mocks.

c) Create a test to verify the CarRepository persistence. Be sure to include an in-memory database dependency in the POM (e.g.: H2).

d) Having all the previous tests passing, implement an integration test to verify the API. Suggestion: use the approach "E/" discussed in the previous project (Employees).



3/

e) Adapt the integration test to use a real database. E.g.:
   - Run a mysql instance and be sure you can connect (for example, using a Docker container)
   - Change the POM to include a dependency to mysql
   - Add the connection properties file in the resources or the “test” part of the project (see the application-integrationtest.properties in the sample project)
   - Use the @TestPropertySource and deactivate the @AutoConfigureTestDatabase.

Explore

- AssertJ library to create expressive assertions in tests: https://assertj.github.io/doc/
- Vídeo sobre testes em Spring Boot: https://www.youtube.com/watch?v=Wpz6b8ZEgcU