

2 Lab: Unit tests with dependency mocking

v2025-10-04

2	Lab: Unit tests with dependency mocking	1
2.1	Stocks portfolio	1
2.2	Conversion method behavior	2
2.3	Connect to remote resource and integration tests	4

Learning objectives

- Run unit tests ([JUnit 5](#)) and mocks ([Mockito](#)), with mocks injection (@Mock).
- Experiment with mock behaviors: strict/lenient verifications, advanced verifications, etc.

Useful resources

- B. Garcia's [chapter on Jupiter and Mockito integration](#).

2.1 Stocks portfolio

Consider the example in Figure 1: the `StocksPortfolio` holds a collection of `Stock`s; the current value of the *portfolio* depends on the current condition of the *Stock Market*. `StockPortfolio#totalValue()` method calculates the value of the portfolio (by summing the current value of owned stocks, looked up in the stock market service).

Implement (at least) one test to verify the implementation of `StockPortfolio#totalValue()`.

Given that test should have predictable results, you need to address the problem of having non-deterministic answers from the `IStockmarketService` interface.

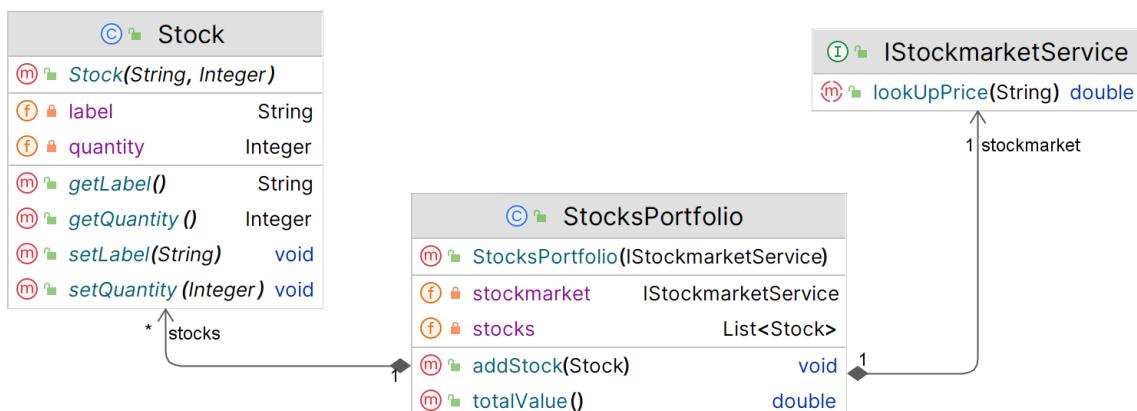


Figure 1: Classes for the StocksPortfolio use case.

- Create the classes. You may write the implementation of the services before or after the tests.
- Create the test for the `StocksPortfolio#totalValue()`. As a guideline, you may adopt this outline:
 - Prepare a mock to substitute the remote stockmarket service (@Mock annotation)
 - Create an instance of the subject under test (SuT) and use the mock to set the (remote) service instance.
 - Load the mock with the proper expectations (when...thenReturn)
 - Execute the test (use the service in the SuT)
 - 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.
 - Check this [example](#) for a quick view on main syntax options and operations.
 - When you run the tests using Mockito in Java 21+ you may get the console warning “Mockito is currently self-attaching to enable the inline-mock-maker. This will no longer work in future releases of the JDK...”. You may need to [explicitly setting up instrumentation for inline mocking](#) (add Mockito as an agent to Maven's surefire plugin).
- c) In the previous task, you have implemented a test with a few expectations declared to instruct the mock to provide a few canned returns.
Add more expectations; in fact, add extra “stocks” types in the expectations that those strictly needed in the test.
Run the test. Note the Mockito warnings.
- d) Instead of the default [JUnit \(Jupiter\) core assertions](#), use another assertions library.
Consider using the [Hamcrest](#) or [AssertJ](#) to create more human-readable assertions. Refactor your code.
Note: Compare the two styles in these [examples](#) (AssertJTest, HamcrestTest).
- e) Add a new method to StocksPortfolio to return the top n most valuable shares. For example, $n=3$ would return a list with the 3 most valuable stocks in the portfolio (considering their quantity times the market value). Consider using an AI tool to implement the method.
- ```
/**
 * @param topN the number of most valuable stocks to return
 * @return a list with the topN most valuable stocks in the portfolio
 */
public List<Stock> mostValuableStocks(int topN) { ... }
```
- f) ....but do not trust AI-generated code to be correct, this time. Write tests for the new code yourself.  
Did the AI offer a correct implementation? Are edge cases properly covered? (Prove with tests.)

## 2.2 Conversion method behavior

If you write code in a TDD<sup>1</sup> style, you’ll quickly run into the situation where your code references some (often third-party) dependency or subsystem. You’ll typically want to ensure that the code under test can be isolated from that dependency.

Consider an application that needs information from a product catalog provided by a B2B partner. The “partner” products API is documented at <https://fakestoreapi.com/docs>

You will consume the remote data through an API, using an HTTP client, but the team has not yet chosen the implementation or library to use for the “HTTP client” module.

---

<sup>1</sup> Test Driven Development (TDD) is a test intensive approach that involves first writing rigorous tests for correctness and only then making progress by writing code that passes those tests.



- a) Create the objects represented in Figure 2. Consider adopting a TDD approach and create just the classes “skeleton” (the code should compile but it is not expected to work, for now).
  - b) Your first test will focus on the behavior of ProductFinderService#findProductDetails(id)

This method takes a product *id number* and gets the product information from the [Products API](#). For that, ProductFinderService uses an HTTP Client to invoke an HTTP GET method.

Consider that:

- the team has not yet decided which HTTP Client to use;
  - each API call is paid and the team wants to minimize costs at development time.
  - the main aspect to be tested is whether `findProductDetails` can instantiate well-formed objects from the JSON obtained in the API response.

How would mocking be used in this scenario?

- c) Implement the following tests cases to verify the behavior of `findProductDetails`:

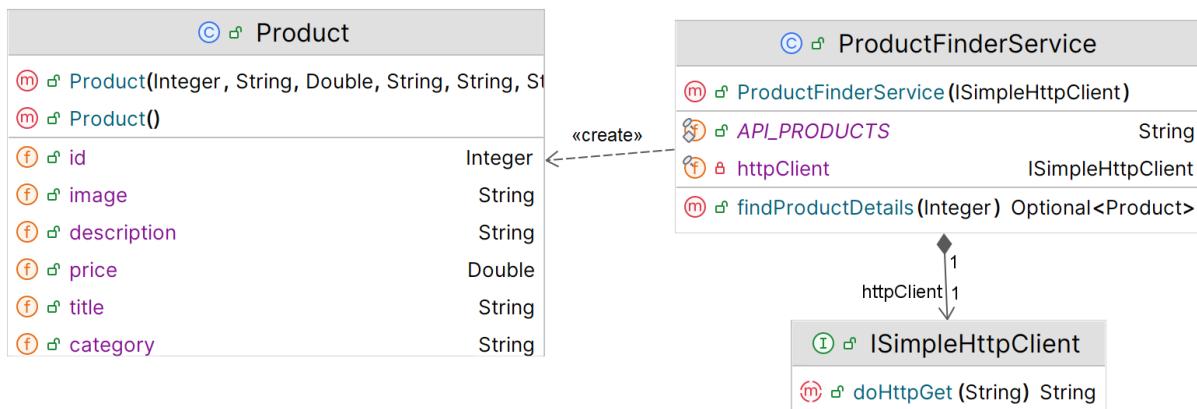
  - a. `findProductDetails(3)` returns a valid Product object, with id = 3 and title = "Mens Cotton Jacket".
  - b. `findProductDetails(300)` returns no Product (could return an empty Optional).

Note: you want to mock the response of `ISimpleHttpClient#doHttpGet()`.

- d) Implement the required code for the tests to pass.

You should implement the parsing logic inside `findProductsDetails` (parse JSON into POJO<sup>2</sup>).

Remember you don't have yet the implementation for the HTTP Client (and you are not expected to use a real one for now).



*Figure 2: Classes for the product catalog use case.*

<sup>2</sup> POJO means “plain old Java Objects” and is used when you want to refer to “traditional” object-oriented data structuring.

## 2.3 Connect to remote resource and integration tests

The mocked HTTP Client was useful, but, at some point, you also need to test against the real remote API Service, otherwise you will have the false impression that your code is robust (tests always passing) even in the case the remote API has changed.

Consider you are implementing an **integration test** for the previous example; in this case, you would use the real implementation of the modules in the test (no mocks).

- a) Implement a working version of an HTTP Client compatible with `ISimpleHttpClient`. You may base your code on [this sample](#).
- b) Be sure the “[failsafe](#)” maven plugin is configured. Use the same (up to date!) version for `surefire` and `failsafe` plugins.
- c) Create a test class (`ProductFinderServiceIT`) in a separate file (be sure the suffix is “`IT`”). Copy the tests from the previous exercise into this new test class and remove any support for mocking (**no Mockito imports** in this second test class).
- d) Correct/complete the test implementation so it uses the **real HttpClient** implementation.
- e) Run your test. Confirm that the remote API is being invoked in the test execution (e.g.: by connecting/disconnecting from the Internet).  
Confirm that your test is passing.

```
$ mvn failsafe:integration-test
```

- f) In the previous exercise we used a mock fake to provide the HTTP client behavior and, here, the real implementation. Comment, in your own words:
  - Was it easy to swap between the mock and the real implementation, in tests? How did it affect the design of the production code?
  - What would be the consequence of instantiating the dependency (the HTTP client) **inside** the service constructor (instead of being passed as a parameter)?
- g) Concerning test execution, take note of the differences between the following cases and provide an explanation.

```
$ mvn test
$ mvn package
$ mvn package -DskipTests=true
$ mvn failsafe:integration-test
$ mvn install
```