# 6 Lab: BDD with Cucumber framework

v2025-11-20



## 6.0 Introduction

**Learning objectives**

- Write descriptive test scenarios with Gerking DSL to capture requirements in BDD style.
- Automate test scenarios using Cucumber framework for Java and JUnit.

**Key Points**

- Story-driven development plays a central role in modern software development, in which the "story" acts as a unit of planning, monitoring and delivery. Being able to **lower the semantic gap between user stories and tests** will enhance traceability and testability of requirements.
- The Cucumber framework enables the concept of "executable specifications": with Cucumber we use **examples** to specify what we want the software to do (as we do to define stories' acceptance criteria). The **scenarios are written before the production code**. This is at the heart of BDD (behavior-driven development)
- Cucumber executes features (test scenarios) written with the Gherkin language (readable by non-programmers too).
- The steps included in the feature description (scenario) must be mapped into Java test code by annotating test methods with matching "expressions". Expressions can be (traditional) regular expressions or the (new) Cucumber expressions.
- BDD plays well with user stories: the user story can be used as a unit of specification, work assignment, acceptance, and delivery. In this sense, we can have better traceability from business requirements to code.

**Explore**

- Cucumber school: guided exercises, video lessons… Especially: Cumber for Java developers.

## 6.1 Getting started (calculator)

Consider that you are implementing a Reverse Polish Notation (RPN) calculator module and you want to supply some tests[1]. You operate the RPN calculator by pushing operands and operators (one at a time).

a) Start a (regular) Java Maven project to run Cucumber tests.

[1] This example is discussed in "Cucumber in a Nutshell" section, in B. Garcia's book; solution code available, if needed.

Maven can use project templates (a.k.a. archetypes) to create new projects, with a certain amount of configuration already included. We haven't used it before, so this is a good opportunity to do it now[2]:

Option 1: create the project using **Maven archetypes**

- Open your IDE and start the dialog to create new Maven project; at this point, look for the option to use an archetype to create the project. Be sure to select *io.cucumber:**cucumber-archetype***

Option 2: create an empty Maven project

- Create the project as usual. Them confirm that you set the POM dependencies (with dependencies management, based on official "skeleton" for JVM)

b) Create a .feature file (a regular text file with .feature extension) to describe the intended use of the calculator (related example). Note that the *.feature* file must go into the tests resources folder, e.g.:

./src/**test/resources**/*mypackage*/calculator.*feature*

c) Be sure to implement the required test steps.

You will need two test files: one (generic) to activate Cucumber engine and another with the effective test steps implementation (related example).

Note: in the example from Boni García book, step matching uses regular expressions. The best practice, however, is to use "cucumber expressions". **Be sure to "upgrade"** the sample code. E.g.:

| (**old**) regular expressions style: | (**better**) Cucumber expressions style: |
|---|---|
| @When("^I add (\\d+) and (\\d+)$") | @When("I add {int} and {int}") |

d) Run the tests.

Confirm that the elements in your feature are used to trigger and "feed" the tests.

e) Add a few more test scenarios in your feature (e.g.: multiply,…)

Now, if you try to build the project, the tests will not pass (insufficient tests for the new scenarios).

Note the "clues" in the output, giving suggestions to implement the missing steps (i.e., test methods).

Implement the required test steps to have your feature specification satisfied.

## 6.2 Passing data to tests

Consider a Library service class that manages a collection of Books.

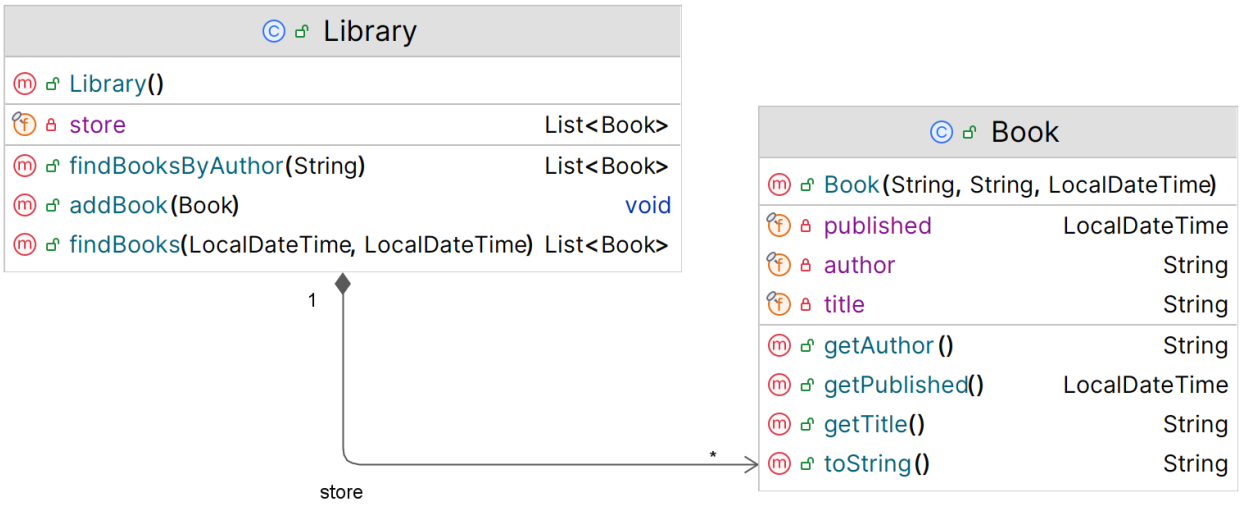
a) Write a feature to verify representative scenarios related to the **book search** user story. Consider a few search options (by author, by category, no results found, etc) that you may infer from the class contract (diagram).

You may use an AI tool to kickstart some useful Cucumber scenarios (i.e., the feature file).

[2] When using Maven archetypes, make sure you use an updated and maintained one. Often, archetypes are "old" and include deprecated modules.

Notes:

- Write the features before the test steps. Steps can be partially generated from features.
- Prefer the "cucumber expressions" (instead of regular expressions) to write the steps definitions.
- Start with a couple of basic scenarios, than refine.




b) Include the requirement to filter by dates. In this case, you will have to specify dates in the feature file. To handle dates matching, consider using a ParameterType configuration. This defines a new custom parameter type to use in the matching expressions. [ adapt from → partial snippet].

*@When("the customer searches for books published between {iso8601Date} and {iso8601Date}")*

The dates in the feature description need also to match the date mask defined in the ParameterType (**aaaa-mm-dd**). E.g.:

*Given a book with the title 'One good book', written by 'Anonymous', published in 2013-03-12*

*And another book with the title 'Some other book', written by 'Tim Tomson', published in 2020-08-23*

c) In this problem, it would be useful to load a "database" of books upfront. Gherkin supports the concept of "Data Tables". Consider using a DataTable mapping and access your data as a *list of maps* (use headings in the data).

## 6.3 Web automation (online library)

Cucumber is often used with web automation to write expressive web automation tests.

a) Consider the example from the previous lab, related to the "fake" online library (section 5.3).

Develop some scenarios to test the book search story and relevant scenarios (and write the "feature" file).

b) Implement the test automation using Cucumber, Jupiter and browser automation.

Option 1: using Selenium [related example].

Option 2: using Playwright integration [Playwright integration is similar to previous lab]