a) Open http://demo.redmine.org
b) Sign-in with your credentials
c) Assert that you have successfully logged in (by verifying the presence of the username)
d) Logout

... and Stop recording. Test your macro (replay).

Add a new step, at the end, to confirm that, after logout, the home shows the "Sign in" option present. Enter this assertion "manually" (in the editor, but not recording).

**2a/** Run the test as a Java project (JUnit 5, Selenium)

Prepare a (new) project to run JUnit tests and Selenium (sample POM.xml available; alternative site).

Take note of the information in this page under "quick reference"; then, in the section "Local browsers", pick the example that suites your setup and run the test (as you usually do with JUnit). You will have to deploy the WebDriver implementation (binary) for you browser [ → download browser driver]. Be sure to include in the system PATH.

**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 code that was generated to be compliant with JUnit 5 and the Selenium-Jupiter extension.

Run the test (programmatically).

**3/** Use the Web Page Object pattern

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

Notes:
Execute the suggested steps to interactively record the test case. Export it to Java and run the test with JUnit automation, refactoring for JUnit 5.
The text in the tutorial is somewhat old. You may **need to adapt to the current implementation** of the site under test (e.g.: IDs of page elements,...).
You should **stop** at "Increased readability" with Cucumber.

### Explore

- Puppeteer - a Node library which provides a high-level API to control headless Chrome/Chromium.
- Another, more recent, Page Object Model example.

## Lab 4: Behavior-driven development (Cucumber in Java)

### Key Points

— The Cucumber framework enables the concept of "executable specifications": with Cucumber we use concrete examples to specify what we want the software to do. **Scenarios are written before production code**.
— 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.

## Lab

Bear in mind that the integration of JUnit 5 and Cumcumber is still very recent and most of the samples available in the internet are still based on JUnit 4.

In this lab you will:
1/ Create a simple cumcumber-enabled project
2/ Book search example
3/ Integrate Cucumber with Selenium Webdriver

**1**/ Create a simple cucumber-enabled project

This example assumes that you have Maven available in the path and you can invoke mvn from the command line. Check with:

Go through the Cucumber getting started tutorial for Java.
The example uses the command line and the IntelliJ IDEA (you may use other IDE, but you will need to adapt the instructions).

Note that:
→ Start the project from the suggested Maven archetype io.cucumber:cucumber-archetype
→ You need a test class that activates the Cucumber runner; that is the purpose of the "RunCucumberTest" file automatically included in this archetype.
→ the features are stored under src/test/resources/ and the folder structure (under resources) must mirror the package hierarchy under src/test/java). In this sample: src/test/resources/hellocumber/ and src/test/java/hellocumcumber/
→ IntelliJ recognizes the .feature file type. You can even select "Run all features" to run the Cucumber tests.

**2/** Book search example

To get into the "spirit" of BDD, partner with a colleague, and jointly write a couple of features to verify a book search user story. Consider a few search options (by year, etc).

Take the approach discussed in this example, and write your own tests. Feel free to add different scenarios/features.

The sample uses Cucumber v2, but you should write the test steps using Cucumber 3x. Some changes are required:
In the "book search" story:

— You will need to change the parameters placeholder in the steps definition. Prefer the "cucumber expressions" (instead of regular expressions). [ → partial snippet]

— migrate the date formatter option, by creating a new datatype in a "Configurer" class, which should be placed in the same package as the test steps [ → possible solution snipet ]. This defines a new custom parameter type ("`date_iso_local_date_time`")

— The dates in the feature description need also to match the date mask used (aaaa-mm-dd).

In the "salary" story:

— Adapt the data table definition for the Salary use case [ → possible solution snipet]. This allows to extract a List<Employees> from the feature definition and use it as a parameter.

```
public void
the_salary_management_system_is_initialized_with_the_following_data(final
List<Employee> employees)
```

Notes:
→ you may build on the previous project or start with a new one, but stick with the base archetype (io.cucumber:cucumber-archetype).

**3/** Integrate Cucumber with Selenium Webdriver

Implement the brief sample using cucumber and selenium webdriver to declare expressive web automation tests.
You may now extend this strategy to revisit the example (Weather forecast) of lab 3.

## Lab 5: 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, but it is better to limit Application Context 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 “bizz 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.