Software Architecture

THIRD PRACTICAL ACTIVITY – PRAC3

Presentation

In PRAC2 we have implemented three microservices of the **Photo&Film4You** case. In this PRAC3 we will test one of the three microservices and validate that our code follows a hexagonal architecture.

PRAC3 covers the contents of the books "Microservices Patterns" and "Fundamentals of Software Architecture" (see Resources section for the chapters to be studied).

Competencies

In this practice, the following competencies of the Degree in Computer Engineering are worked on:

- Know how to propose and evaluate different technological alternatives to solve a specific problem.
- Application of Software Engineering techniques to the different stages of the lifecycle of the software development project.

Objectives

The goals of the PRAC3 are:

- Recognize the factors of testing and quality of a specification/design.
- Evaluate formally the testing and the quality of an implementation.
- Identify the most important models and standards of quality of software.

Description of PRAC3

PRAC3 consists of 3 exercises. In each of them, it is necessary to describe **how the exercise has been done** and provide **instructions** for its correct execution.

Exercise 1 (5 points)

Starting from the published solution of PRAC2, which is available at: https://github.com/orgs/UOC-SA-SPRING-2025/repositories, you have to implement **unit tests**, using **JUnit**, **Spring Boot Test**, **Mockito,** and **AssertJ**.

Notes: You can use the following references to help you to develop the tests:

- https://github.com/anirban99/hexagonal-architecture/tree/master/src/test/java/com/example/hexagonal/architecture
- https://github.com/eugenp/tutorials/blob/master/spring-boot-modules/spring-boot-testing/src/test/java/com/baeldung/boot/testing/EmployeeServiceImplIntegrationTest.java
- https://spring.io/guides/gs/testing-web/
- https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing.spring-boot-applications.with-mock-environment

Tip: In the unit tests, we do not use any real infrastructure or real dependencies. We do not use HTTP communication, servers, databases, external services, or other microservices. Every aspect related to dependencies has to be simulated (with **Mockito**), except for domain classes like Product, etc.

WHAT TO DO

Develop the following classes and tests:

1. **DigitalItemUnitTest**. Include a test for the domain class DigitalItem that verifies that when creating a new DigitalItem, its status is AVAILABLE.

2. **DigitalSessionServiceUnitTest. Include tests that verify:**

    - **Test Method 1:** When calling the method *findDigitalSessionByUser* with an existing/valid userId, the list of digital sessions is correctly returned.

    - **Test Method 2:** When calling the method *findDigitalSessionByUser* with a non-existent userId, the exception indicating that the user was not found is thrown.

Notes:
Dependencies on both *RestTemplate* and *DigitalSessionRepository* must be mocked.

3. DigitalItemRESTControllerUnitTest. Include a test that verifies that when calling the *findDigitalItemBySession* endpoint, the items belonging to the session with the specified identifier are correctly returned.

   Notes:

- The server cannot be started to perform the test. The web infrastructure must be simulated to receive and handle the REST request, as well as the dependency on the DigitalItemService.

Exercise 2 (3 points)

Starting from the published solution of PRAC2, which is available at: https://github.com/orgs/UOC-SA-SPRING-2025/repositories, you have to implement **integration tests**, using **JUnit, Spring Boot Test and AssertJ.**

Notes: You can use the following reference to help you to develop the tests:

- https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot-testing/src/test/java/com/baeldung/boot/testing

Tip: The integration tests have to test that the service communicates with its real dependencies and infrastructure. Therefore, they do not use the library **Mockito** as in the unit tests.

WHAT TO DO

Develop the following class with a test:

- **DigitalItemRepositoryIntegrationTest**. Test that when adding a DigitalSession with some DigitalItems, we can correctly retrieve these items using the *findDigitalItemBySession* method.
  Notes:
  - In this case, it will be necessary to use a real database to properly test that the integration works correctly.

- **DigitalItemRESTControllerIntegrationTest**. Test that when adding a DigitalSession with some DigitalItems, we can correctly retrieve them through an HTTP request to

the route exposed by the DigitalItemRESTController to get the DigitalItems of a DigitalSession.

Notes: In this case, the controller method should NOT be called directly; instead, the HTTP request must be made directly to the real web service of the Spring application.

Exercise 3 (2 points)

Starting from the published solution of PRAC2, which is available at: https://github.com/orgs/UOC-SA-SPRING-2025/repositories, we want to verify that the architecture is correctly based on a hexagonal architecture and that we use good architectural rules.

Recommended reading:

- Chapter "Measuring and Governing Architecture Characteristics" of the book "Fundamentals of Software Architecture".
- ArchUnit Documentation: https://www.archunit.org/userguide/html/000_index.html#_junit_4_5_support
- ArchUnit examples with JUnit5: https://github.com/tng/archunit-examples/tree/main/example-junit5/src/test

WHAT TO DO

Develop the following 2 automated architecture verification tests on the *UserService* microservice:

- The hexagonal architecture (called *onion* in **ArchUnit**) is satisfied.
- Classes that are in the *package domain.service* and are annotated with *@Service* have their name ending in **ServiceImpl**.

Resources

Basic Resources

- Book "Microservices Patterns" (Chris Richardson): Chapter 9. "Testing microservices: Parte 1". Introduction.
- Book "Microservices Patterns" (Chris Richardson): Chapter 10. "Testing microservices: Parte 2".
- Book "Fundamentals of Software Architecture" (Richards & Ford).
    - Chapter 3. “Modularity".
    - Chapter 6. “Measuring and Governing Architecture Characteristics".

Evaluation criteria

- In this activity, **the use of artificial intelligence tools is not allowed,** and it must be completed **strictly individually**. If irregularities are detected, the activity will be penalized with a grade of D. This includes the reuse of solutions from this practice from previous semesters. In the teaching plan and on the UOC website about academic integrity and plagiarism, you will find information about what constitutes irregular conduct in evaluation and the consequences it may have.
- The weight of each exercise is indicated in the statement.
- The evaluation criteria for each exercise are indicated in the annex.
- It is necessary to justify the solution to each of the exercises. Both the correctness of the solution and the given justification will be evaluated.

Format and submission date

A single **PDF document** named LastNameFirstName_ISCSDPRAC3.pdf must be submitted, with a description of how the practice was done (justifying decisions and providing instructions on how the developed tests should be executed), and the **code of the tests implemented** must be submitted (in a ZIP file).

The PDF document and the ZIP file must be submitted to the “Submission of PRAC3” space section of the classroom **before 11:59 p.m. on June 9, 2025**. Late submissions will not be accepted.

Annex

Evaluation Criteria for PRAC3

	Excellent (10 puntos)	**Notable (7 puntos)**	**Sufficient (5 puntos)**	**Insufficient (2,5 puntos)**	**No answer / All wrong (0 puntos)**
Ex. 1	• 4 tests correctly implemented and justified in exercises 1.1, 1.2, and 1.3. • Correct use of JUnit in all 4 tests of exercises 1.1, 1.2, and 1.3. • Correct use of Mockito in 3 tests of exercises 1.2 and 1.3. • Separate the two tests of exercise 1.2 into 2 methods. • Overall implementation quality is perfect (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (4 points)	• 3 tests correctly implemented and justified in exercises 1.1, 1.2, and 1.3. • Correct use of JUnit in all 4 tests of exercises 1.1, 1.2, and 1.3. • Correct use of Mockito in 2 out of 3 tests of exercises 1.2 and 1.3. • Separate the two tests of exercise 1.2 into 2 methods. • Overall implementation quality is good (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (3 points)	• 2 tests correctly implemented and justified in exercises 1.1, 1.2, and 1.3. • Correct use of JUnit in all 4 tests of exercises 1.1, 1.2, and 1.3. • Correct use of Mockito in 2 out of 3 tests of exercises 1.2 and 1.3. • Do not separate the two tests of exercise 1.2 into 2 methods. • Overall implementation quality is sufficient (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (2 points)	• 1 test correctly implemented and justified in exercises 1.1, 1.2, and 1.3. • Incorrect use of JUnit in all 4 tests of exercises 1.1, 1.2, and 1.3. • Mockito was not used in the 3 tests of exercises 1.2 and 1.3. • Do not separate the two tests of exercise 1.2 into 2 methods. • Overall implementation quality is insufficient (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (1 point)	No answer, no justification, or all incorrect. (0 points)
Ex. 2	• 2 tests correctly implemented and justified in exercises 2.1 and 2.2, • Overall implementation quality is perfect (criteria: readability, format, no warnings, use of constants instead of direct coding in	• 2 tests correctly implemented, but justification is incorrect in exercises 2.1 and 2.2, • Overall implementation quality is good (criteria: readability, format, no warnings, use of constants instead of direct	• 1 test correctly implemented, but no justification in exercises 2.1 and 2.2, • Overall implementation quality is sufficient (criteria: readability, format, no warnings, use of constants	• 1 test with justification, but incorrectly implemented in exercises 2.1 and 2.2, • Overall implementation quality is insufficient (criteria: readability, format, no warnings, use of constants	No answer, no justification, or all incorrect. (0 points)

	method calls, etc.). (4 points)	coding in method calls, etc.). (3 points)	instead of direct coding in method calls, etc.). (2 points)	instead of direct coding in method calls, etc.). (1 point)	
Ex. 3	• 2 tests are correctly implemented and justified. • Overall implementation quality is perfect (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (2 points)	• 2 tests are implemented but the justification is incorrect. • Overall implementation quality is good (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (1.5 points)	• 1 test correctly implemented and justified. • Overall implementation quality is sufficient (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (1 point)	• Tests are poorly implemented and not justified. • Overall implementation quality is insufficient (criteria: readability, format, no warnings, use of constants instead of direct coding in method calls, etc.). (0.5 points)	No answer, no justification, or all incorrect. (0 points)