3 Lab: Unit tests with dependency mocking

v2025-02-26

3 Lab: Unit tests with dependency mocking 1

3.1 Employee management (getting started example) 1

3.2 Cars service (test slicing and TDD) 4

3.3 Integration test 5

Learning objectives

- Apply different test strategies for different test scopes in a multi-layer Spring Boot application.
- Distinguish integration tests from slicing aproaches.

Key Points

- @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, when possile.
- @DataJpaTest only loads Spring data components, and will greatly improve performance by not loading @Service, @Controller, etc.
- Use @WebMvcTest to test the web boundary layer, exposed through Controllers. Beans used by the controller can be mocked.
- Isolate the functionality to be tested by limiting the context of loaded frameworks/components. For some scenarios, you can even test with just standard unit testing (no Spring Boot application configuration required).

Explore

- Talk on Spring Boot tests (by Pivotal): https://www.youtube.com/watch?v=Wpz6b8ZEgcU

3.1 Employee management (getting started example)

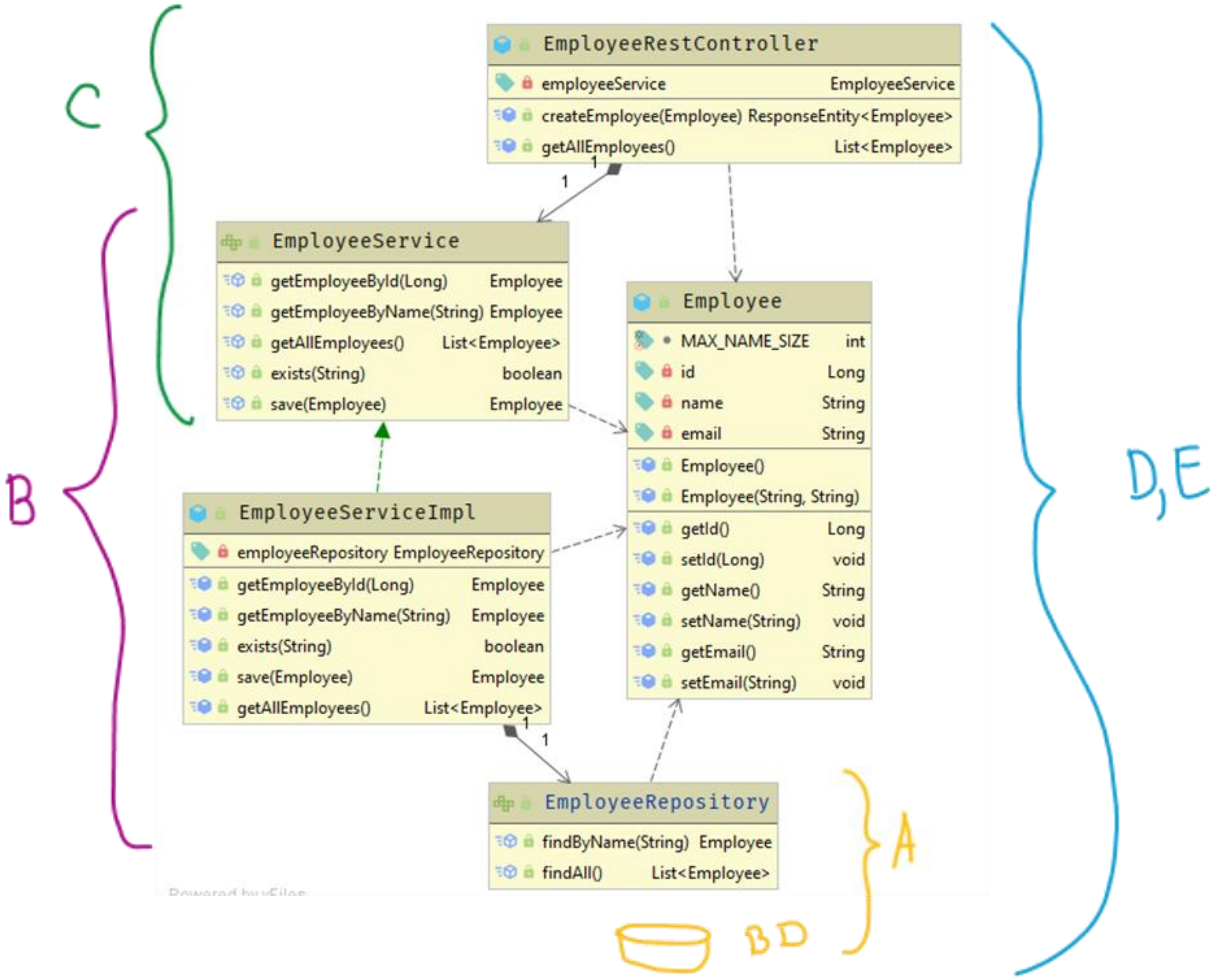
Types of tests and Spring Boot:

Type	**Purpose**	**Spring Boot examples**
Unit tests	Focused on correctness of logic in isolation from collaborators and infrastructure. Run fast.	Test a single class or method in isolation (e.g., a service or a utility class).
Integration Tests	Validate the interaction between multiple architecture layers and/or with the infrastructure services.	Test involving multiple layers (e.g., repository + service + controller). Sometimes use the real application context to confirm wiring and configuration.
End-to-End	Simulate the entire application flow for certain features.	Typically require a running instance of the application and related services. May include front end (user facing) or external programmatic client.

Study the example concerning a simplified Employee management application (project: gs-employee-manager). This application follows the Spring Boot architecture style to structure the solution (see class diagram):

- Employee: entity (@Entity) representing a domain concept.
- EmployeeRepository: the interface (@Repository) defining the data access methods on the target entity, based on JpaRepository. “Standard” requests can be inferred and automatically supported by the framework; custom queries can be declared, if needed.
- 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 boundary endpoint (@RestController): handles the HTTP requests and delegates to the EmployeeService.

The project already contains a set of tests.



Study in the code the examples for the following test scenarios:

Purpose/scope	**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 to write to the database directly (no caches involved).	@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 implementation of services. [*EmployeeService_UnitTest*]	Often it can be achieved with unit tests, if one mocks the repository. 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), limiting to the controller context. [*EmployeeController_ WithMockServiceTest*]	Run the tests in a simplified web 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).	MockMvc provides an entry point to server-side testing. Despite the name, it is not related to Mockito. MockMvc provides an expressive API, in which methods chaining is expected. Focused on the boundary layer in isolation.
D/ Integration test, from boundary to repo. Load the full Spring Boot application. No esternal 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/ Integration test, from boundary to repo. Load the full application. Test the REST API with explicit HTTP client. [*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 convenient servlet entry point for tests, uses an API client (so request and response un/marshaling will be involved).

Note 1: both D/ and E/ load the full Spring Boot Application (auto scan, etc...). The main difference is that in D/ one accesses the server context through a special testing servlet (MockMvc object), while in E/ the requester is a REST client (TestRestTemplate).

Note 2: you may run individual tests using maven command line options. E.g.:

```
$ mvn test -Dtest=EmployeeService*
```

Review questions: [answer in a **readme.md** file, in /lab3_1 folder]

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

b) Take note of transitive annotations included in @DataJpaTest.

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

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

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

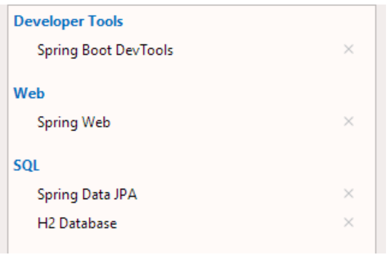
f) the sample project demonstrates three test strategies to assess an API (C, D and E) developed with SpringBoot. Which are the main/key differences?

3.2 Cars service (test slicing and TDD)

Consider the case in which you will develop an API for a **car information system** (as a Spring Boot application).

Consider using the Spring Boot Initializr to create the new project (either online or may be integrated in your IDE);

Add the dependencies (*starters*) for: Developer Tools, Spring Web, Spring Data JPA and H2 Database.



Developer Tools

Spring Boot DevTools

Web

Spring Web

SQL

Spring Data JPA

H2 Database

Use 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**.

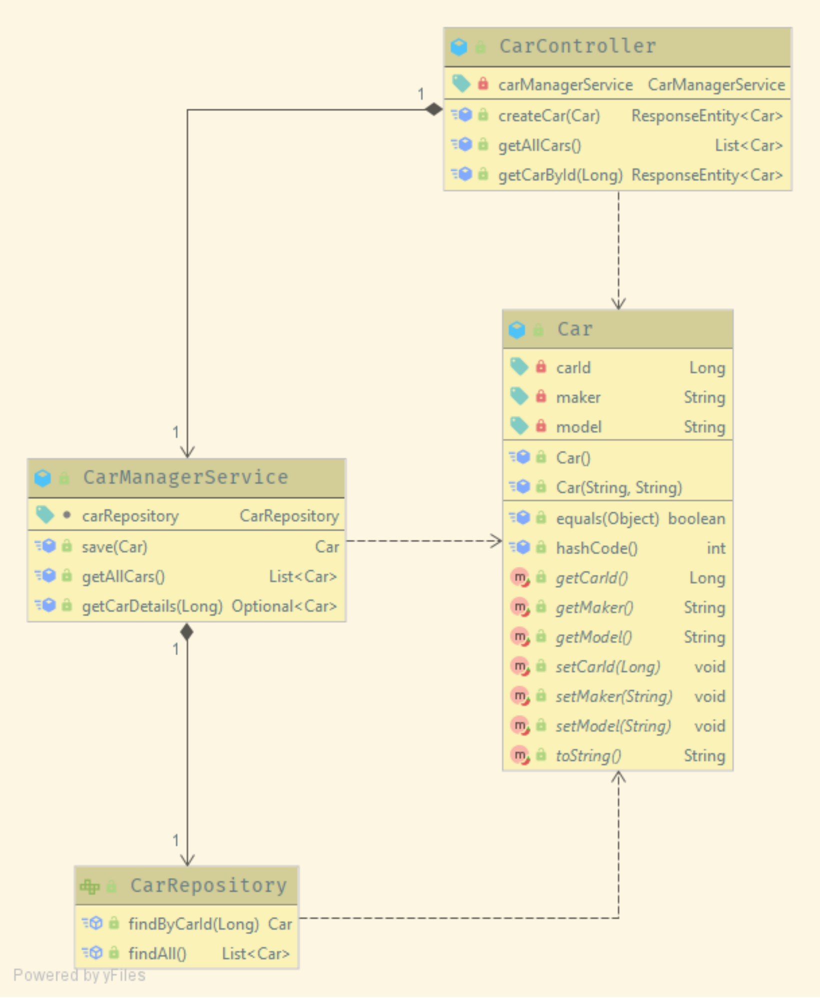
a) Create a test to verify the Car [Rest]Controller (and mock the CarService bean), as “resource efficient” as possible. Run the test.

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) Although the previous tasks are focused more in “wiring” that true test logic, in a larger application, we would expect to implement more complex, interesting, and mission-critical business logic.

   Implement the required functionality to answer the request: “find a car that provides a *suitable replacement* for some given car”, for example, to be used as a courtesy car for a client. Usually, this involves a car in the same segment, motor type, etc. You may consider a simple “matching” strategy for now. Test the business logic at the appropriate level.



3.3 Integration test

[Continue in the same project of the previous exercise.]

a) Having all the previous tests passing, implement an integration test to verify interactions from the boundary (API) to the repository. Suggestion: use the scenario "**E/**" discussed in the first project (Employees).

b) 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* [optionally remove H2].
- Add the connection properties file in the resources of the "test" part of the project (see the application-integrationtest.properties in the sample project)
- Use the @TestPropertySource and deactivate the @AutoConfigureTestDatabase.

c) What could be the advantages and disadvantages of using a real database connection in testing activities?