4 Lab: testing multi-layer applications (Spring Boot)

v2025-10-14

4 Lab: testing multi-layer applications (Spring Boot) 1
4.1 Employee management (getting started example) 1
4.2 Meals booking API (test slicing)....... 4\
4.3 Testing practices....... 4
4.4 Rest Assured library....... 4

Learning objectives

- Identify different test objectives in a layered application.
- Implement tests using the Spring Boot instrumentation, segmenting test scopes to enhance tests specificity and performance.

Key Points

- The Spring Boot framework provides a rich framework to develop enterprise applications with Java but also provides test instrumentation that enables you to run segmented tests.
- @SpringBootTest annotation loads whole application context. If possible, limit application contexts only to a subset of Spring components that participate in test scenario.
- @DataJpaTest 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.
- For some scenarios, in which you do not depend on Spring Boot managed beans, you can test with standard unit testing (no Spring Boot application configuration required).

Explore

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

4.1 Employee management (getting started example)

Tests scope and Spring Boot:

Scope	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). Great for business logic too.
Integration Tests	Validate the interaction between multiple architecture layers and/or with the infrastructure services.	Test involving multiple layers (e.g., repository + service + controller). Remember including some tests that 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[1]). This application follows the Spring Boot architecture style to structure the solution (Figure 1):

- 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.

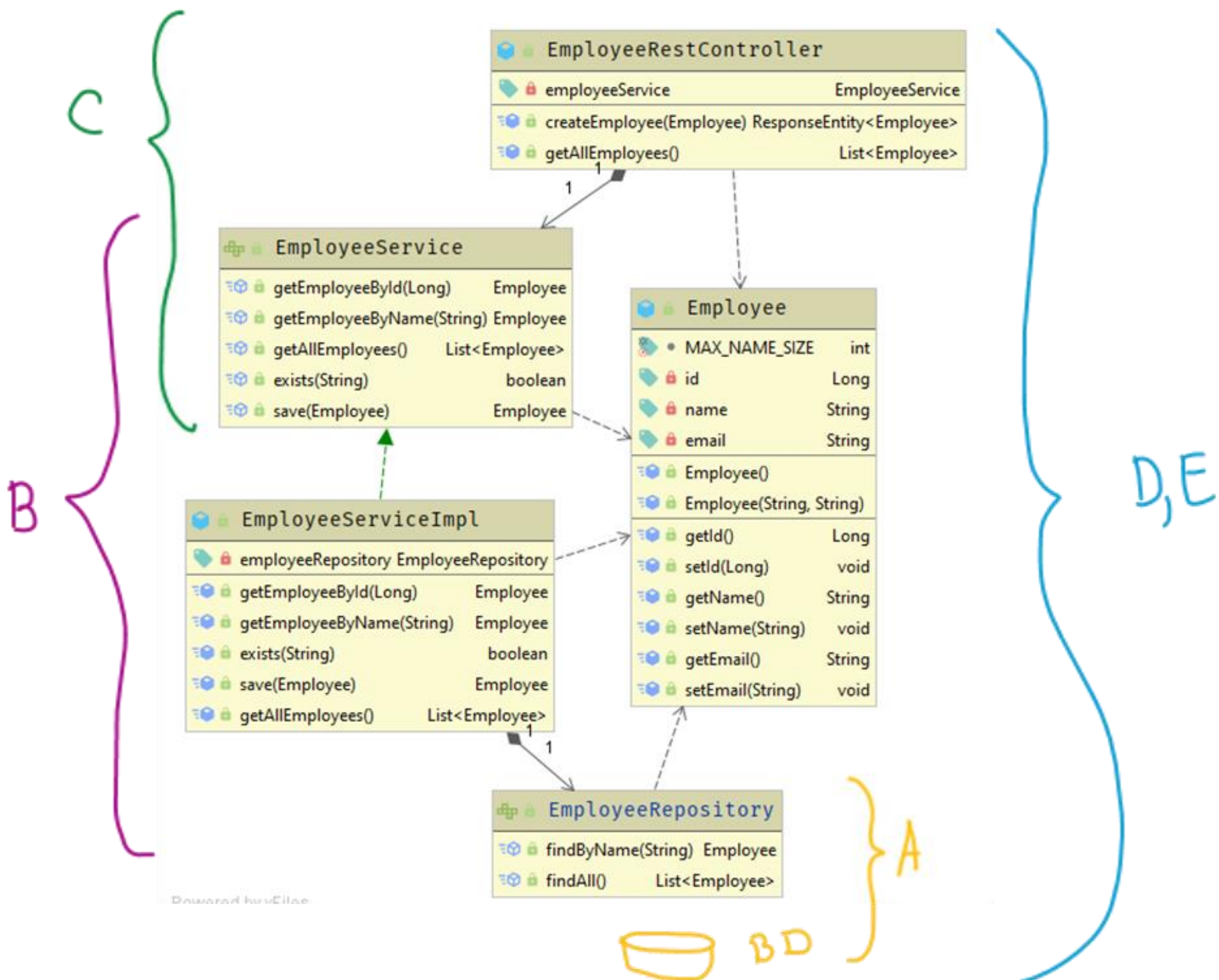

Figure 1: structure of the provided Spring Boot application. Letters annotate different test scopes.

[1] Remember: don't clone external Git repositories directly into your own TQS repository, to avoid having "git inside git".

Explore the code and study 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 Maven dependency for 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. [*EmployeeServiceUnitTest*]	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 external API client is 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, 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/ 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*
```

Answer in your lab notes:

- What is the difference between standard @Mock and @MockBean?
- What is the role of the file “application-integrationtest.properties”? In which conditions will it be used?

4.2 Meals booking API (test slicing)

Let's return to the meals booking application from last week (activity 3.4).

Make a copy of the existing implementation, to which you will now add tests.

a) We will follow a top-down (or outward in) approach.

   Create a test to verify the [Rest]Controller (and mock the "service" bean), as "resource efficient" as possible. Run the test.

b) Include tests to verify the meals booking logic. In fact, you should be able to adapt/reuse content from Lab 1 (activity 1.3), as you are likely to need standard unit test with mocks.

c) Create a test to verify the repository persistence contract. Be sure to include an in-memory database dependency in the POM (e.g.: H2).

   Be sure to include a custom query method in your repository (using *@Query*).

d) Your tests should be passing, but, if you did use the "mocking" approach to slice the tests scope, you haven't tested the whole suite (the integrated stack, from boundary to the repository).

   Implement an **integration test** to verify interactions from the boundary (API) to the repository, without fake/mocked elements. Suggestion: use the scenario "**E/**" discussed in the first section (Employees example).

   Adapt the integration test to use a real database. Suggestion: use the same approach as in the previous lab (activity 3.4).

   - Make sure you have included the required database driver as a dependency.
   - Add the connection properties file in the **test resources** part of the project (see the application-integrationtest.properties in the sample project)
   - In this test, use the @TestPropertySource and deactivate the @AutoConfigureTestDatabase.
   - You should be able to test your app without further changes, reusing the same test logic.

4.3 Testing practices

a) In the previous example, you were asked to write tests in a top-down approach. Can this approach help with a TDD practice, in which you try to defer the implementation (of the production code) as much as possible? Elaborate.

b) Consider you are implementing a backend application for a car rental company. It is critical for the business to manage clients, cars, booking and rentals. Among others, you need to address the following requirement: "find a car that provides a *suitable replacement* for some given car, for example, to be used as a courtesy car for a client". Offering a "similar" car involves selecting a car in the same segment, motor type, etc., and that is available.

   Describe the **core tests** you recommend and how would they be implemented in a Spring Boot application.

4.4 Rest Assured library

Spring Boot provides a robust testing infrastructure. With minimal configuration/annotations, Spring Boot recognizes the "intention" of the tests and does its best to fire up the appropriate environment and interaction objects. (Note that, otherwise, you would need to ensure the proper deployment of artifacts in an application container and rely mainly on logs to audit problems…)

Testing a REST API is a common task and there are alternatives to the Spring Boot own instrumentation.

REST Assured can be used as a REST-client test library for Java, e.g.:

```
given().
        param("x", "y").
when().
        get("/lotto").
then().
        statusCode(400).body("lotto.lottoId", equalTo(6));
```

a) Replicate the integration tests you may have, that verify your API for the meals booking project, now using the Rest Assured library.

   Remember Rest Assured is an “external client” to your API; you need to run the application so these (integration) tests can connect.

Note: you may explore related examples from documentation or the tutorial materials from Baeldung.