```
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

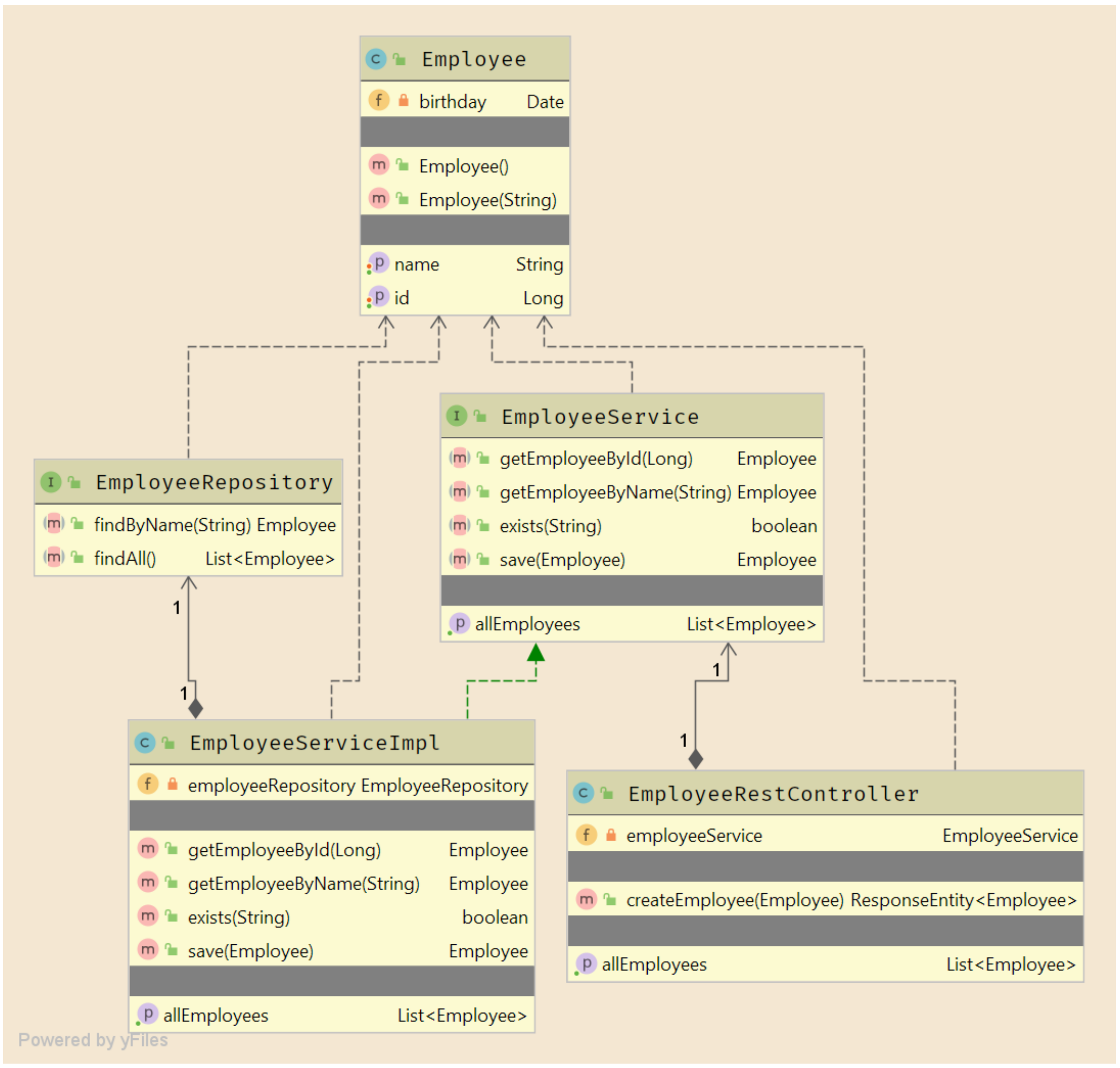
1/
Study the example available concerning a simplified Employee management application (gs-employee-manager).
This application follows commons practices to implement a Spring Boot solution:

— 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.


Employee
birthday Date
Employee()
Employee(String)
name String
id Long
EmployeeService
getEmployeeById(Long) Employee
getEmployeeByName(String) Employee
exists(String) boolean
save(Employee) Employee
allEmployees List<Employee>
EmployeeRepository
findByName(String) Employee
findAll() List<Employee>
1
1
EmployeeServiceImpl
employeeRepository EmployeeRepository
getEmployeeById(Long) Employee
getEmployeeByName(String) Employee
exists(String) boolean
save(Employee) Employee
allEmployees List<Employee>
1
1
EmployeeRestController
employeeService EmployeeService
createEmployee(Employee) ResponseEntity<Employee>
allEmployees List<Employee>
Powered by yFiles

The project contains a set of tests. Take note of the following test scenarios:

Purpose	**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 directly this object to write to the database.	@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 services implementation. [EmployeeServiceImplUnitTest]	Can be achieved with a unit test, if we mock the repository behavior. 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). No need to test the real HTTP-REST framework; just the controller behavior. [EmployeeControllerIT]	Run the tests in a simplified light 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	MockMvc provides an entry point to server-side testing. Despite the name, is not related to Mockito. MockMvc provides an expressive API, in which methods chaining is expected.

	(@MockBean); the repository component will not be involved.	
D/ Verify the boundary components (controllers). Test the REST API on the server-side; no 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/ Verify the boundary components (controllers). Test the REST API with explicit HTTP client involved. [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 server entry point for tests, start a API client (so request and response un/marshaling will be involved).

Review questions:

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

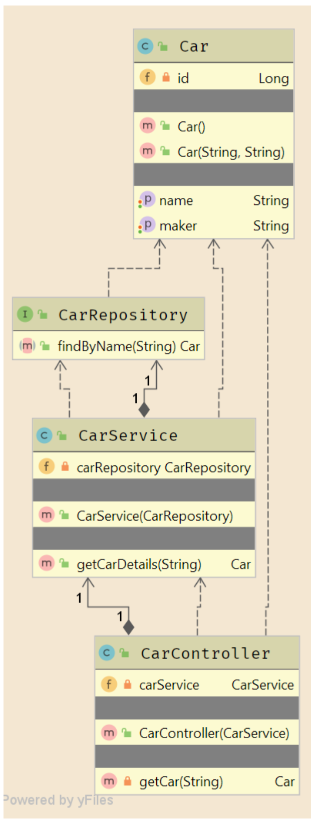
b) Identify an example in which you mock the behaviour of the repository (and avoid involvind a database).

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

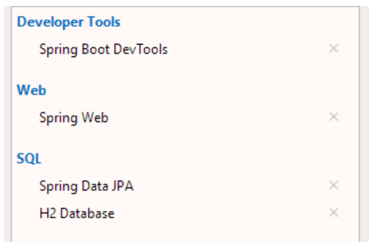
d) What is the role of the file “application-integrationtest.properties”? In which conditions will it be used?

2/

Consider the case in which you will develop an API for a car information system.



Implement this scenario, as a Spring Boot application. Consider using the [Spring Boot Initializr]() to create the new project (integrated in IntelliJ or online); add the “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 ×

Take the structure modeled in the classes 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 will be forced if we try to write the tests in a top-down approach: start from the controller, than the service, than the repository.

a) Create a test to verify the CarController (and mock the CarService bean). 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 a in-memory database dependency in the POM (e.g.: H2).

d) Having all the previous tests passing, implement an integration test to verify the API. Suggestion: use the E/ approach discussed for the previous (Employees) example.

e) 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;
- Add the connection properties file in the resources or the “test” part of the project (see the application-integrationtest.properties in the sample project)
- Use the @TestPropertySource and deactivate the @AutoConfigureTestDatabase.

Lab 6: Static Code analysis with Sonar Qube

Prepare

You will find a lot of demos in Youtube under for static code analysis in Java with SonarQuebe.
It is not required but you may choose to have a look.

Key Points

Static code quality can be inspected to obtain quality metrics on the code base. These metrics are based on the occurrence of known weaknesses, a.k.a. “code smells”. The code is not executed (thus static analysis).
Key measures include the occurrence of problems likely to produce errors, vulnerabilities (security/reliability concerns) and code smells (bad/poor practice or coding style); coverage (ratio tested/total); and code complexity assessment.
The estimated effort to correct the vulnerabilities is called the **technical debt**. Every software quality engineer must use specialized tools to obtain realistic technical debt information.

Lab

In this lab:
Task 1: Analyze an existing project (maven-based)
Task 2: Include tests and coverage
Task 3: Define and apply quality gates

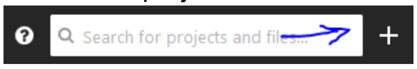
Task 1: Analyze an existing project (maven-based)

a/ Install the SonarQube server either as a local service, or by using the docker image (see instructions in the link).
For the purpose of this lab, you don’t need to configure a production database (the embedded H2 database is used by default). If, however, you want a production-like setup, you should consider using a persistent database (for example, like this configuration; not required for the class).

b/ Configure the environment for Sonar Scanner for Maven. Be sure to adapt the <sonar.host.url> for your setup (e.g.: http://127.0.0.1:9000)

c/ Select a Maven-based, Java application project to use. You may reuse one from previous labs, for example, the Euromillions from Lab 1 (part 2b), with tests passing.
Configure the POM to integrate the Sonar scanner plugin (section #2, in this guide)

d/ Create a project in the Sonar Qube dashboard (default : http://127.0.0.1:9000).



Be sure to define the **project key equal to the maven** GroupId:ArtifactId elements.

Then, generate a token for the project and take note for later use.