45426: Teste e Qualidade de Software

Integration testing and Spring Boot test support

Ilídio Oliveira

v2025-02-25

universidade de aveiro
departamento de eletrónica,
telecomunicações e informática
deti

Learning objectives

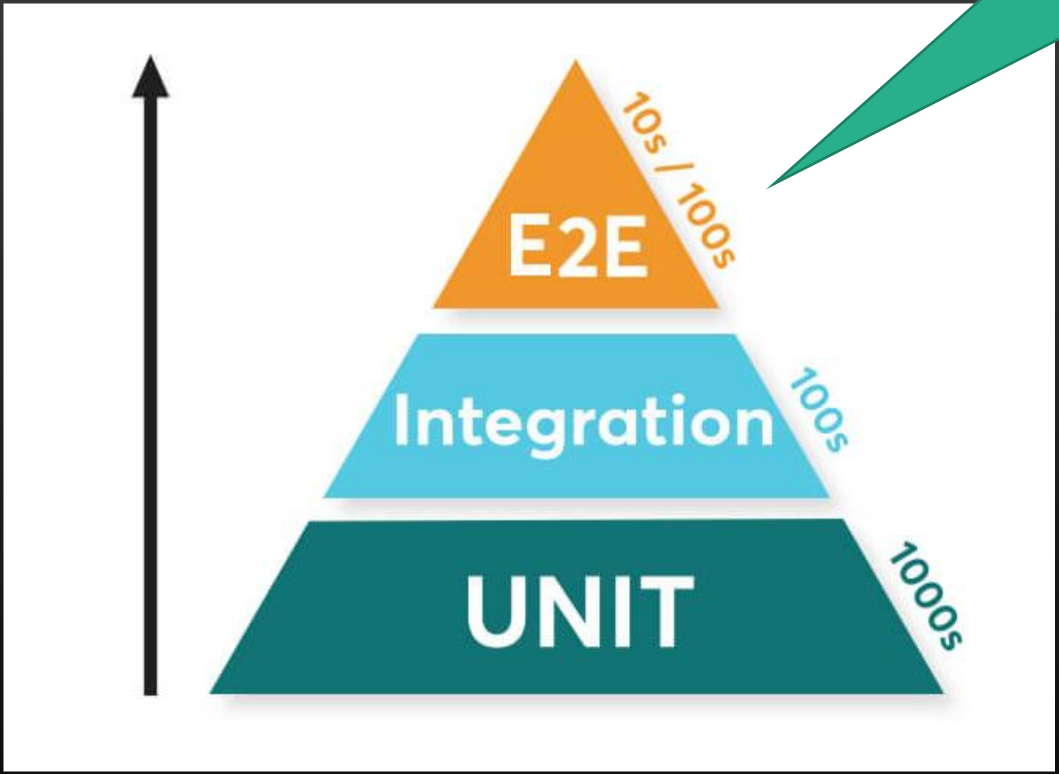
Relate the test of API with the right level of testing in the "pyramid of tests"

Justify the need for "slicing" test scopes.

Discuss diferente strategies to test layered applicationsin Spring Boot.

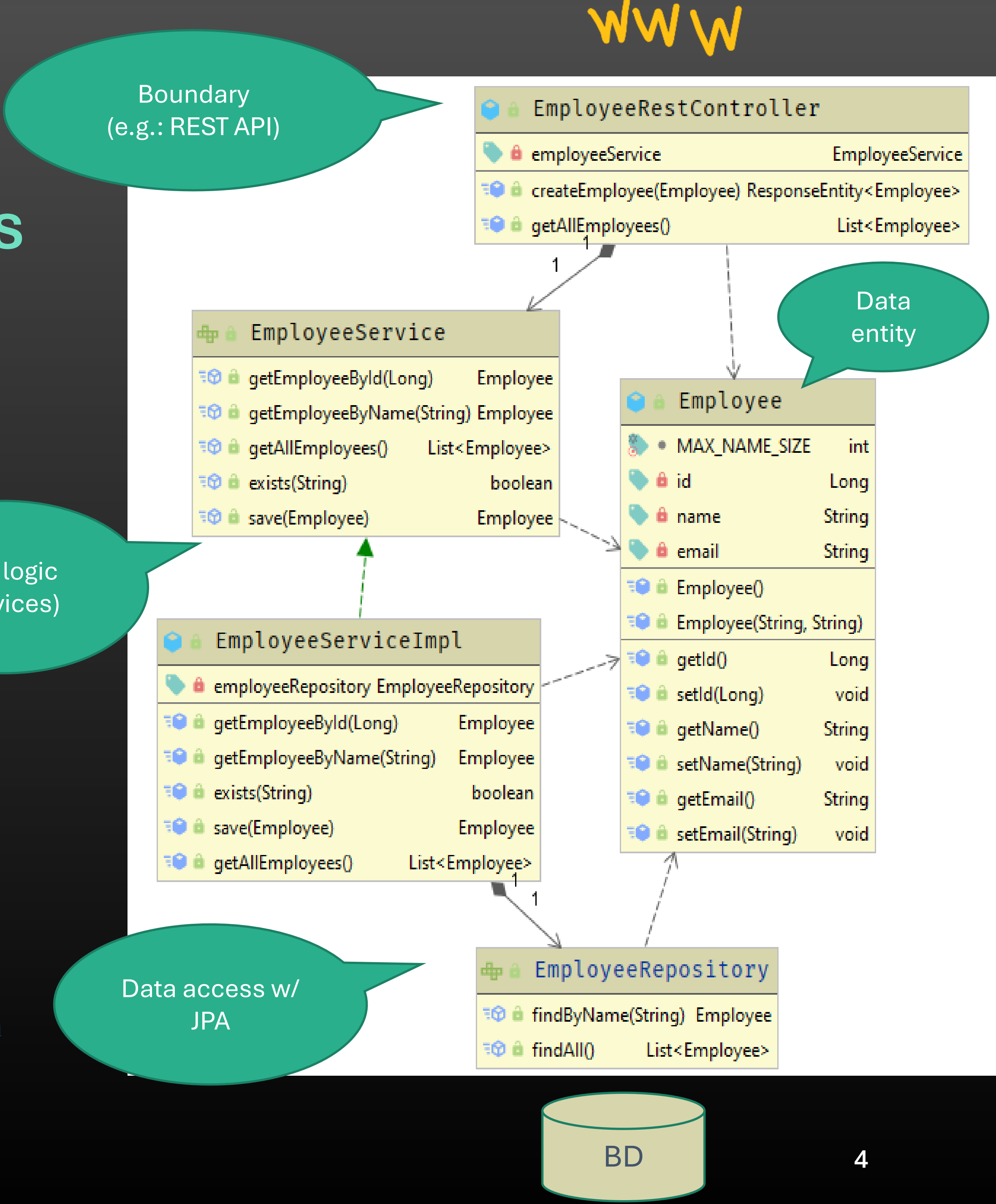
Read SpringBoot tests with mocking of dependencies.

Recall UAT scope



What can we conclude about the root case of an error, when an acceptance test fails?

SB typical layers



EmployeeRestController

employeeService	EmployeeService
createEmployee(Employee)	ResponseEntity<Employee>
getAllEmployees()	List<Employee>

EmployeeService

getEmployeeById(Long)	Employee
getEmployeeByName(String)	Employee
getAllEmployees()	List<Employee>
exists(String)	boolean
save(Employee)	Employee

EmployeeServiceImpl

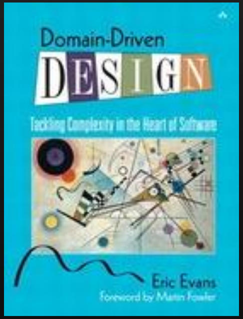
employeeRepository	EmployeeRepository
getEmployeeById(Long)	Employee
getEmployeeByName(String)	Employee
exists(String)	boolean
save(Employee)	Employee
getAllEmployees()	List<Employee>

Employee

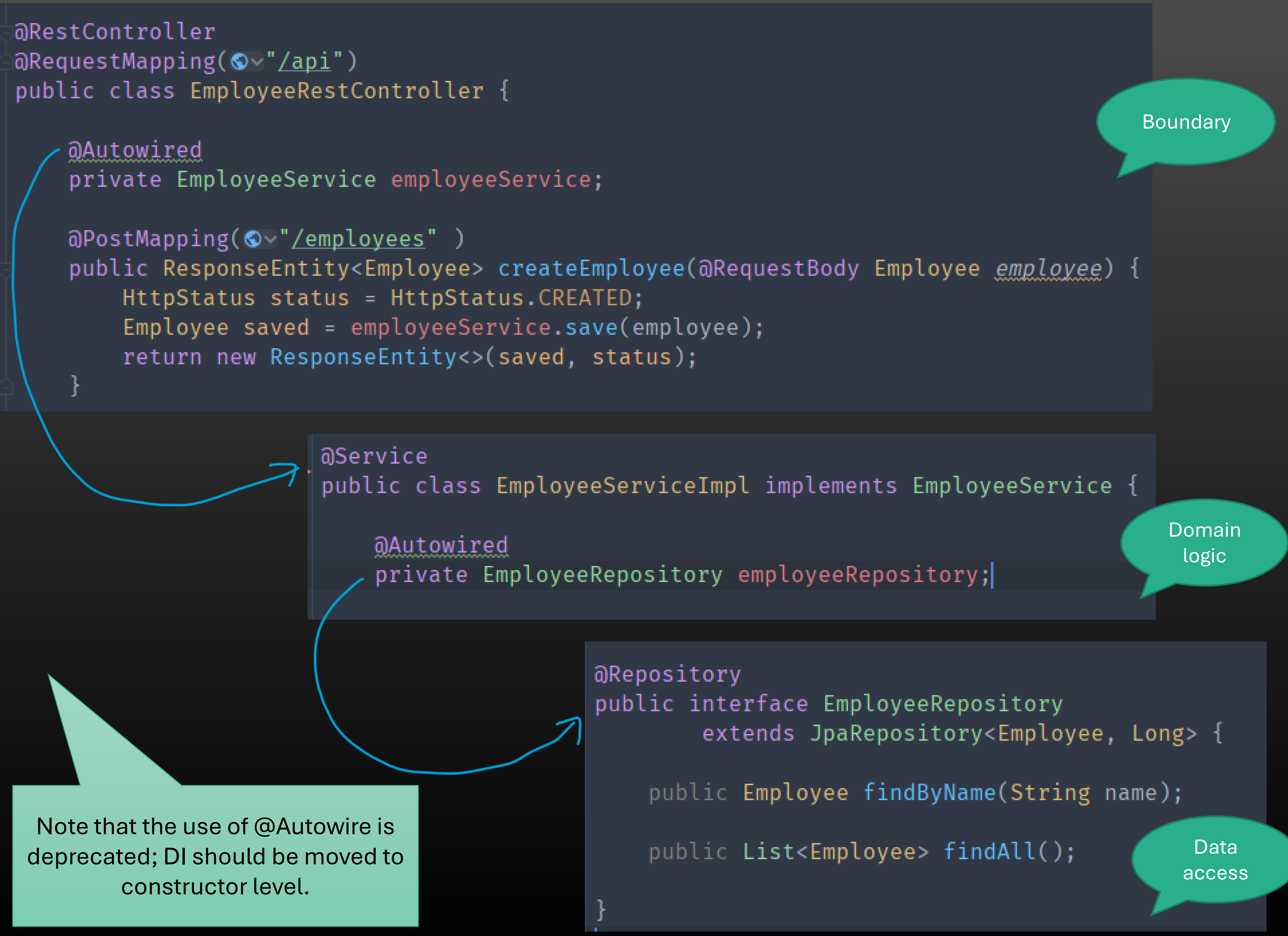
MAX_NAME_SIZE	int
id	Long
name	String
email	String
Employee()	
Employee(String, String)	
getId()	Long
setId(Long)	void
getName()	String
setName(String)	void
getEmail()	String
setEmail(String)	void

EmployeeRepository

findByName(String)	Employee
findAll()	List<Employee>


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

https://learning.oreilly.com/library/view/domain-driven-design-tackling/0321125215/



```java
@RestController
@RequestMapping("/api")
public class EmployeeRestController {

    @Autowired
    private EmployeeService employeeService;

    @PostMapping("/employees" )
    public ResponseEntity<Employee> createEmployee(@RequestBody Employee employee) {
        HttpStatus status = HttpStatus.CREATED;
        Employee saved = employeeService.save(employee);
        return new ResponseEntity<>(saved, status);
    }
```

```java
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;
```

```java
@Repository
public interface EmployeeRepository
        extends JpaRepository<Employee, Long> {

    public Employee findByName(String name);

    public List<Employee> findAll();

}
```

Note that the use of @Autowire is deprecated; DI should be moved to constructor level.

Spring Boot components

Components registration

In each layer, we have various components.

Simply put, to detect them automatically, Spring uses classpath scanning annotations.

Then, it registers each component in the ApplicationContext.

A few of these annotations:

@Component: generic stereotype for any Spring-managed component

@Service: “components” meant to be used at the service layer

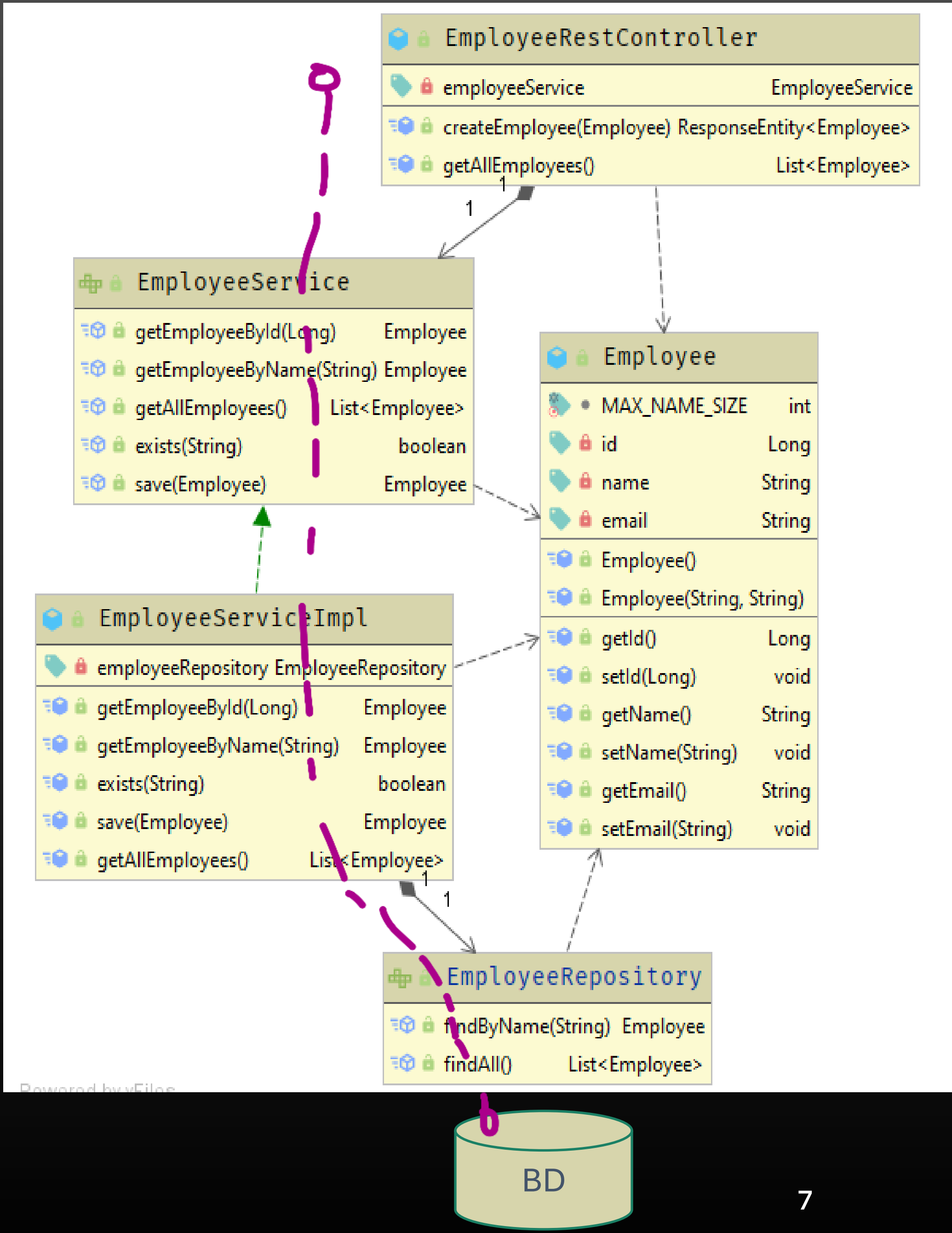
@Repository: classes at the persistence layer, which will act as a database repository

@Service and *@Repository* are special cases of *@Component*.

Test scope #1

Scenario:

- call the REST-endpoint and verify behavior
- full-scope integration test



Spring Boot testing

A helper framework used to simplify the creation of Spring Framework apps

Provides:

- Curated dependencies
- “Starter” configurations (data, web, testing,…)
- “Opinionated” auto-configuration of many components
- Can run most tests without starting an external web container

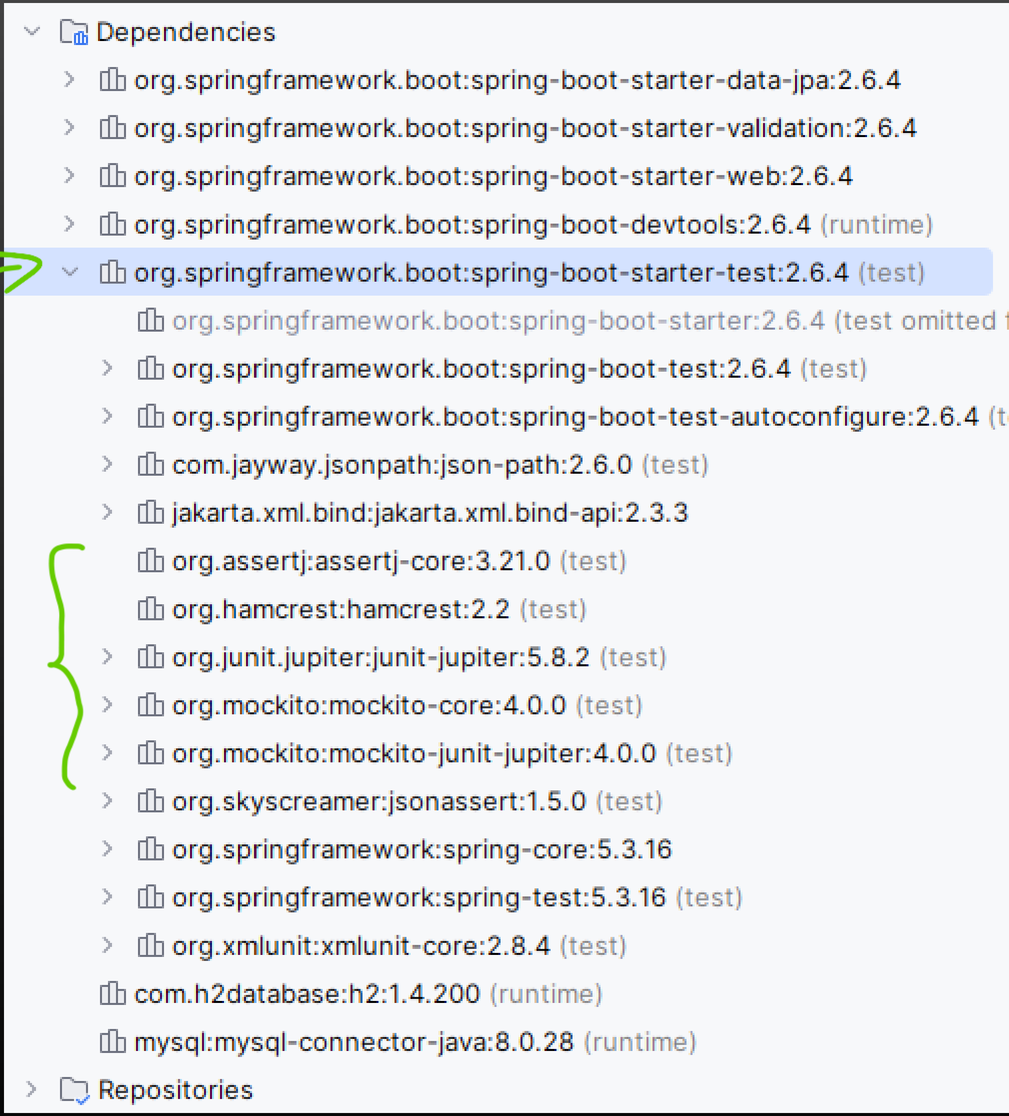
Extending SB philosophy to testing

Test features enabled with

- **spring-boot-starter-test**

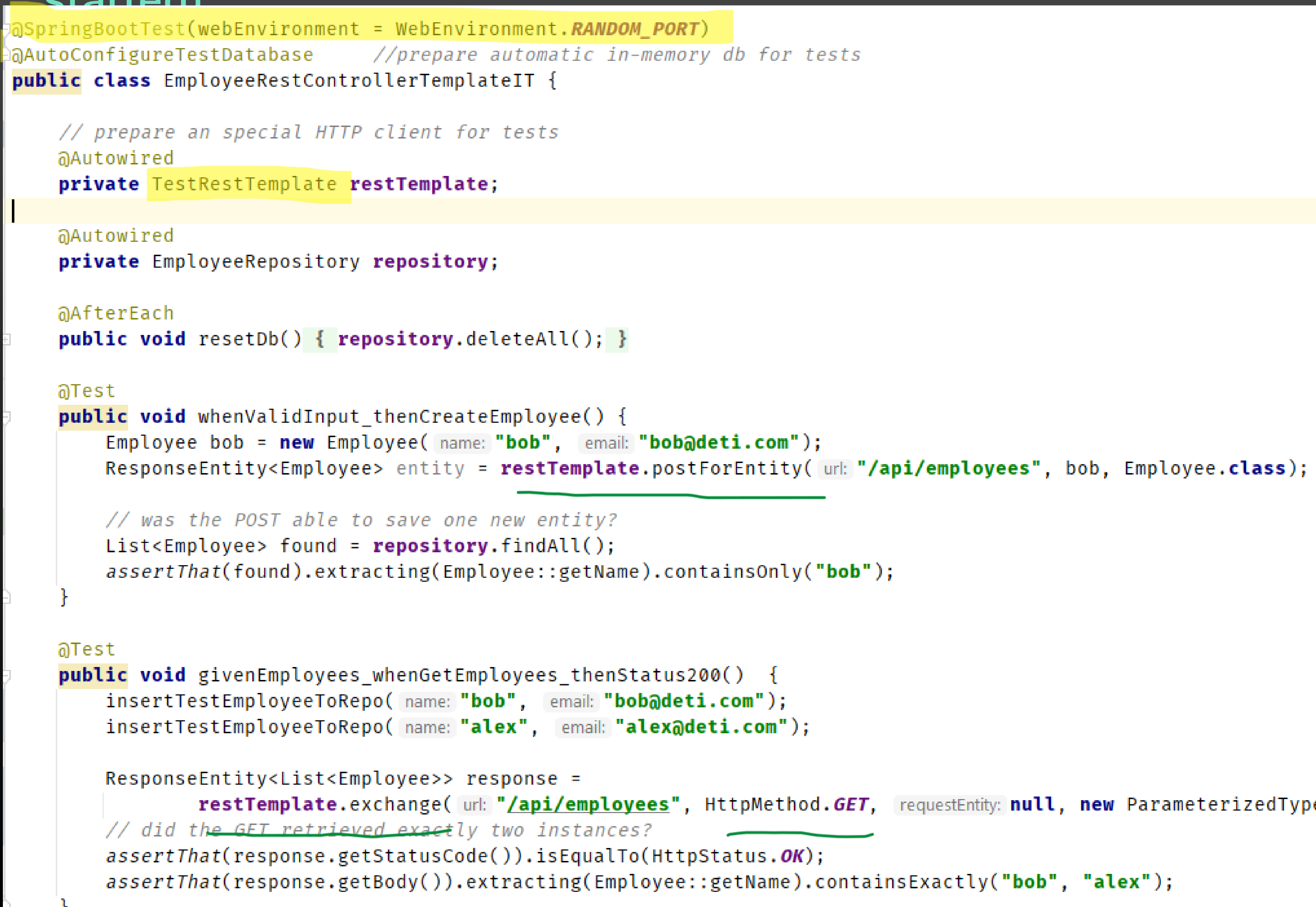
Starter provides:

- Convenient testing dependencies
- Testing auto-config

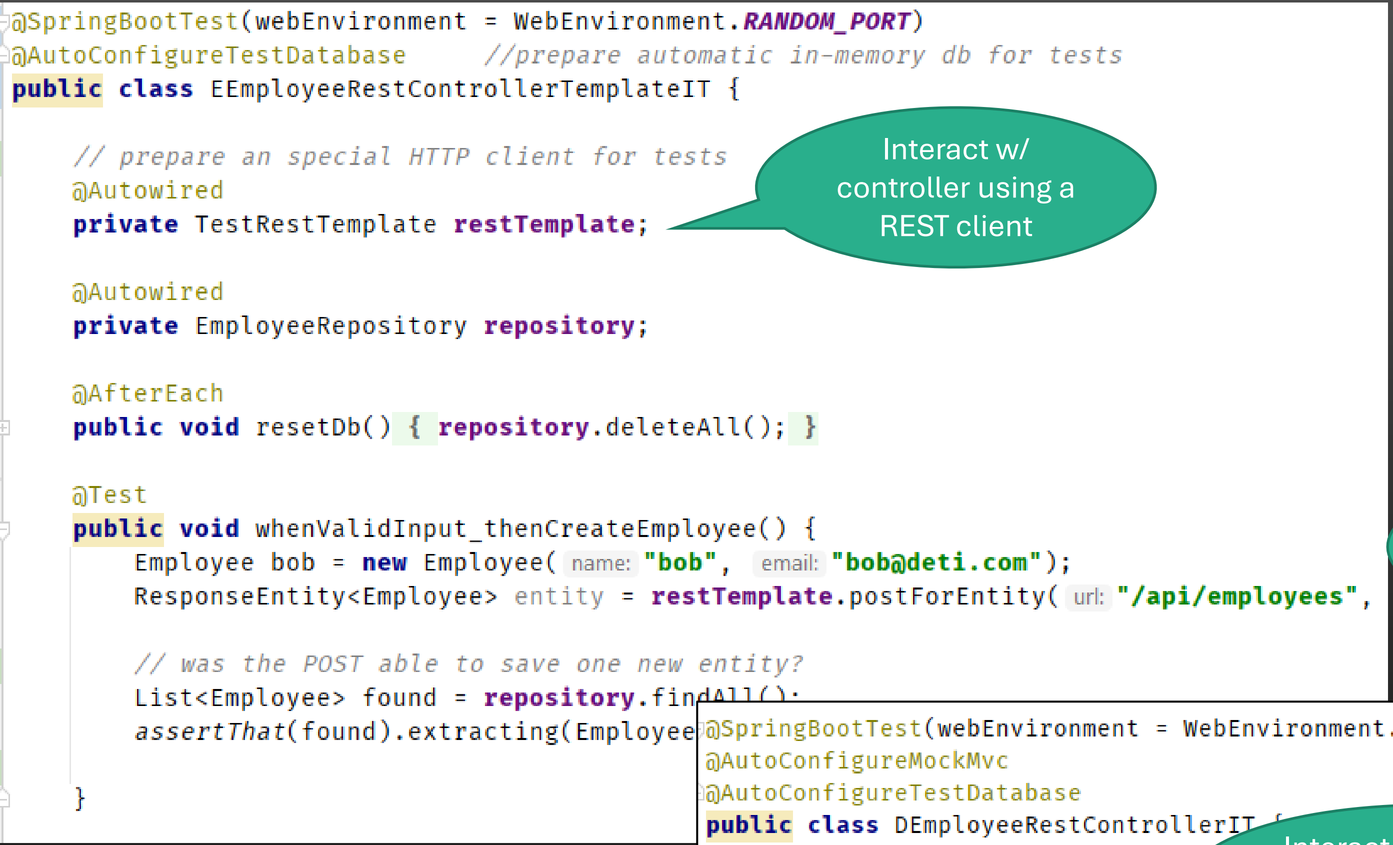


```
Dependencies
  org.springframework.boot:spring-boot-starter-data-jpa:2.6.4
  org.springframework.boot:spring-boot-starter-validation:2.6.4
  org.springframework.boot:spring-boot-starter-web:2.6.4
  org.springframework.boot:spring-boot-devtools:2.6.4 (runtime)
  org.springframework.boot:spring-boot-starter-test:2.6.4 (test)
    org.springframework.boot:spring-boot-starter:2.6.4 (test omitted f
    org.springframework.boot:spring-boot-test:2.6.4 (test)
    org.springframework.boot:spring-boot-test-autoconfigure:2.6.4 (t
    com.jayway.jsonpath:json-path:2.6.0 (test)
    jakarta.xml.bind:jakarta.xml.bind-api:2.3.3
    org.assertj:assertj-core:3.21.0 (test)
    org.hamcrest:hamcrest:2.2 (test)
    org.junit.jupiter:junit-jupiter:5.8.2 (test)
    org.mockito:mockito-core:4.0.0 (test)
    org.mockito:mockito-junit-jupiter:4.0.0 (test)
    org.skyscreamer:jsonassert:1.5.0 (test)
    org.springframework:spring-core:5.3.16
    org.springframework:spring-test:5.3.16 (test)
    org.xmlunit:xmlunit-core:2.8.4 (test)
  com.h2database:h2:1.4.200 (runtime)
  mysql:mysql-connector-java:8.0.28 (runtime)
Repositories
```

Testing the REST controller (full stack, web server started)



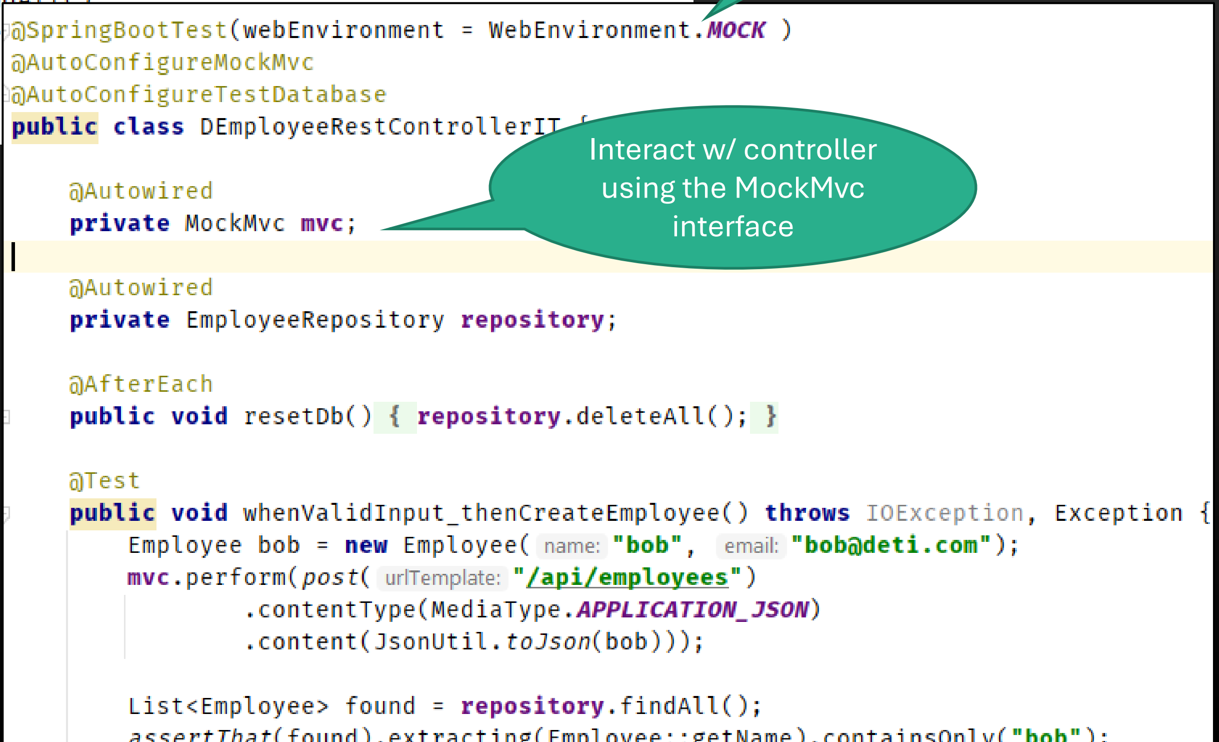
```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureTestDatabase     //prepare automatic in-memory db for tests
public class EmployeeRestControllerTemplateIT {

    // prepare an special HTTP client for tests
    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private EmployeeRepository repository;

    @AfterEach
    public void resetDb() { repository.deleteAll(); }

    @Test
    public void whenValidInput_thenCreateEmployee() {
        Employee bob = new Employee( name: "bob",  email: "bob@deti.com");
        ResponseEntity<Employee> entity = restTemplate.postForEntity( url: "/api/employees", bob, Employee.class);

        // was the POST able to save one new entity?
        List<Employee> found = repository.findAll();
        assertThat(found).extracting(Employee::getName).containsOnly("bob");
    }

    @Test
    public void givenEmployees_whenGetEmployees_thenStatus200()  {
        insertTestEmployeeToRepo( name: "bob",  email: "bob@deti.com");
        insertTestEmployeeToRepo( name: "alex",  email: "alex@deti.com");

        ResponseEntity<List<Employee>> response =
                restTemplate.exchange( url: "/api/employees", HttpMethod.GET,  requestEntity: null, new ParameterizedType
        // did the GET retrieved exactly two instances?
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(response.getBody()).extracting(Employee::getName).containsExactly("bob", "alex");
    }
```



```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureTestDatabase     //prepare automatic in-memory db for tests
public class EEmployeeRestControllerTemplateIT {

    // prepare an special HTTP client for tests
    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private EmployeeRepository repository;

    @AfterEach
    public void resetDb() { repository.deleteAll(); }

    @Test
    public void whenValidInput_thenCreateEmployee() {
        Employee bob = new Employee( name: "bob",  email: "bob@deti.com");
        ResponseEntity<Employee> entity = restTemplate.postForEntity( url: "/api/employees",

        // was the POST able to save one new entity?
        List<Employee> found = repository.findAll();
        assertThat(found).extracting(Employee

    }
```

Interact w/ controller using a REST client

Optionally use a mocked servlet environment



```java
@SpringBootTest(webEnvironment = WebEnvironment.MOCK )
@AutoConfigureMockMvc
@AutoConfigureTestDatabase
public class DEmployeeRestControllerIT {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private EmployeeRepository repository;

    @AfterEach
    public void resetDb() { repository.deleteAll(); }

    @Test
    public void whenValidInput_thenCreateEmployee() throws IOException, Exception {
        Employee bob = new Employee( name: "bob",  email: "bob@deti.com");
        mvc.perform(post( urlTemplate: "/api/employees")
                .contentType(MediaType.APPLICATION_JSON)
                .content(JsonUtil.toJson(bob)));

        List<Employee> found = repository.findAll();
        assertThat(found).extracting(Employee::getName).containsOnly("bob");
```

Interact w/ controller using the MockMvc interface

Another useful approach is to not start the server at all but to test only the layer below that, where Spring handles the incoming HTTP request and hands it off to your controller. (Most of the stack is used; your code will be called in the same way; removes the cost of starting the server.)

@SpringBootTest

@SpringBootTest annotation

Enable FULL context, using all available auto configurations

Heavy!

better to limit Application Context to a set of spring components that participate in test scenario, by listing them (with annotations)

Slicing the test context

Only load slices of functionality when testing spring boot

@xxxxxTest at class level, e.g.: @DataJpaTest, @DataMongoTest, @JsonTest, @WebMvcTest,…

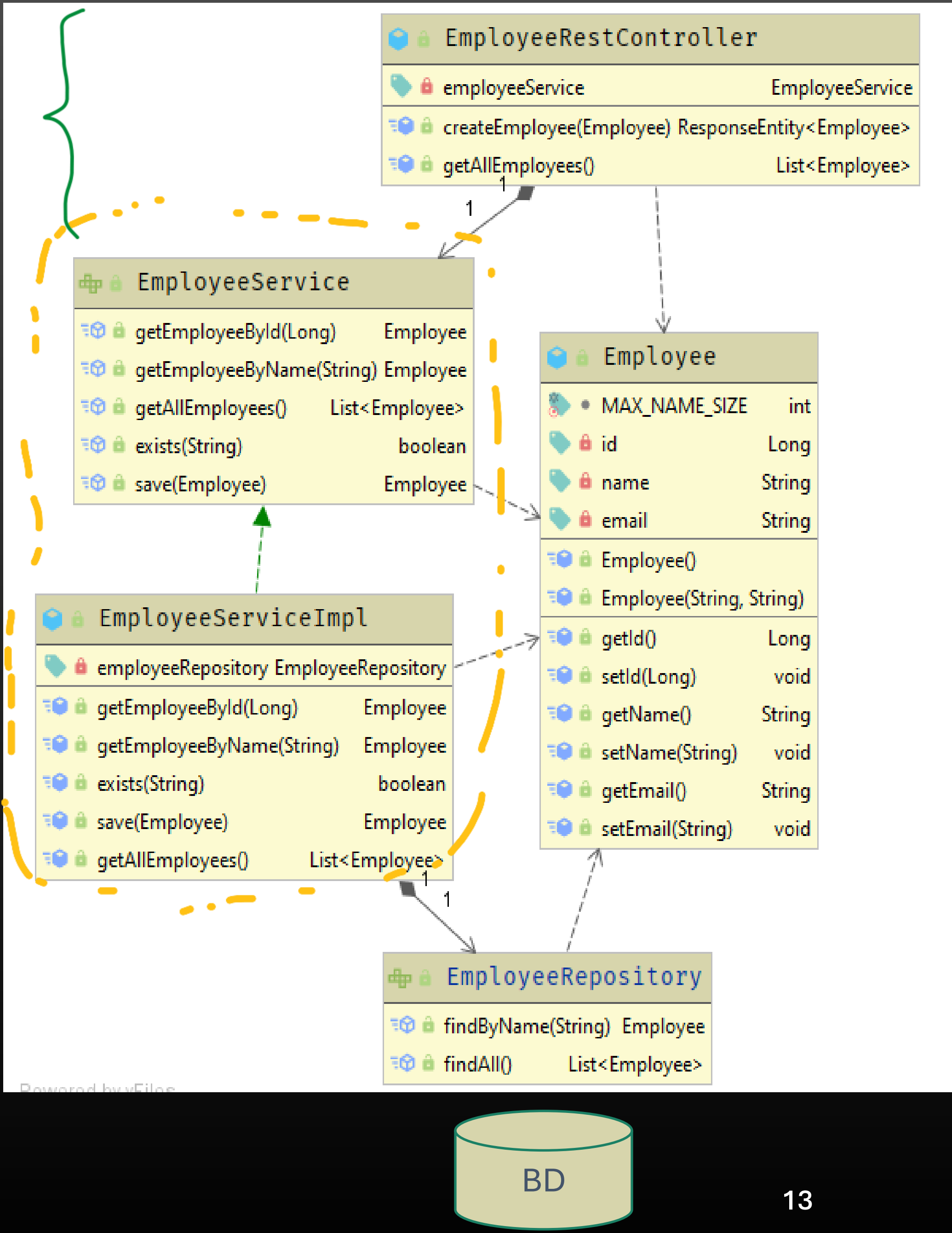
Test scope #2: controller

Scenario:

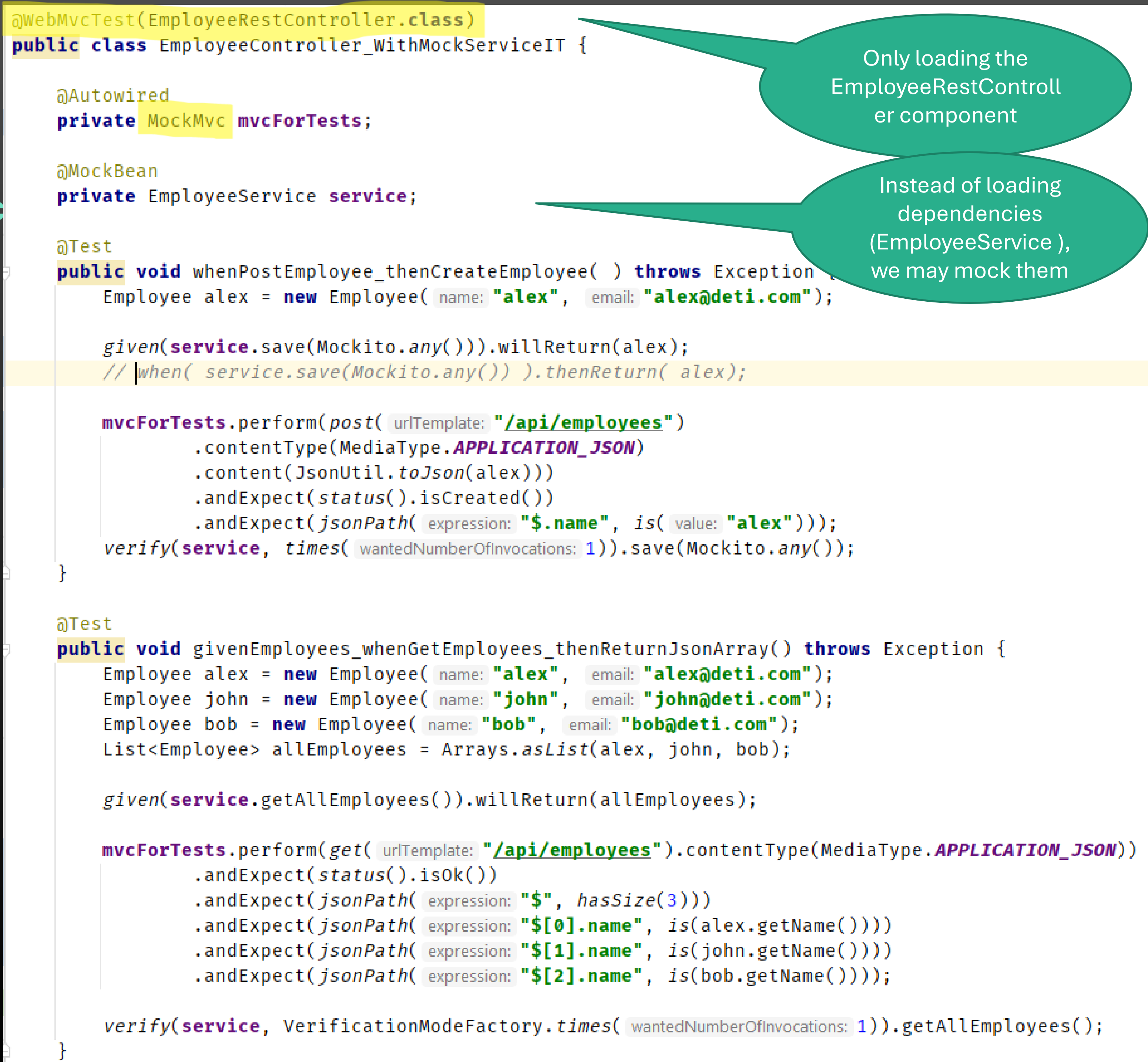
- Test the boundary (Controller)
- Focus on REST contract: path expressions, parameters, JSON,...

Strategy:

- Enable web MVC
- Mock Service behavior [note: we are mocking a SB component, not a regular class...]



MockMvc



```java
@WebMvcTest(EmployeeRestController.class)
public class EmployeeController_WithMockServiceIT {

    @Autowired
    private MockMvc mvcForTests;

    @MockBean
    private EmployeeService service;

    @Test
    public void whenPostEmployee_thenCreateEmployee( ) throws Exception
        Employee alex = new Employee( name: "alex",  email: "alex@deti.com");

        given(service.save(Mockito.any())).willReturn(alex);
        // when( service.save(Mockito.any()) ).thenReturn( alex);

        mvcForTests.perform(post( urlTemplate: "/api/employees")
                .contentType(MediaType.APPLICATION_JSON)
                .content(JsonUtil.toJson(alex)))
                .andExpect(status().isCreated())
                .andExpect(jsonPath( expression: "$.name", is( value: "alex")));
        verify(service, times( wantedNumberOfInvocations: 1)).save(Mockito.any());
    }

    @Test
    public void givenEmployees_whenGetEmployees_thenReturnJsonArray() throws Exception {
        Employee alex = new Employee( name: "alex",  email: "alex@deti.com");
        Employee john = new Employee( name: "john",  email: "john@deti.com");
        Employee bob = new Employee( name: "bob",  email: "bob@deti.com");
        List<Employee> allEmployees = Arrays.asList(alex, john, bob);

        given(service.getAllEmployees()).willReturn(allEmployees);

        mvcForTests.perform(get( urlTemplate: "/api/employees").contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath( expression: "$", hasSize(3)))
                .andExpect(jsonPath( expression: "$[0].name", is(alex.getName())))
                .andExpect(jsonPath( expression: "$[1].name", is(john.getName())))
                .andExpect(jsonPath( expression: "$[2].name", is(bob.getName())));

        verify(service, VerificationModeFactory.times( wantedNumberOfInvocations: 1)).getAllEmployees();
    }
```

Only loading the EmployeeRestController component

Instead of loading dependencies (EmployeeService), we may mock them

I Oliveira

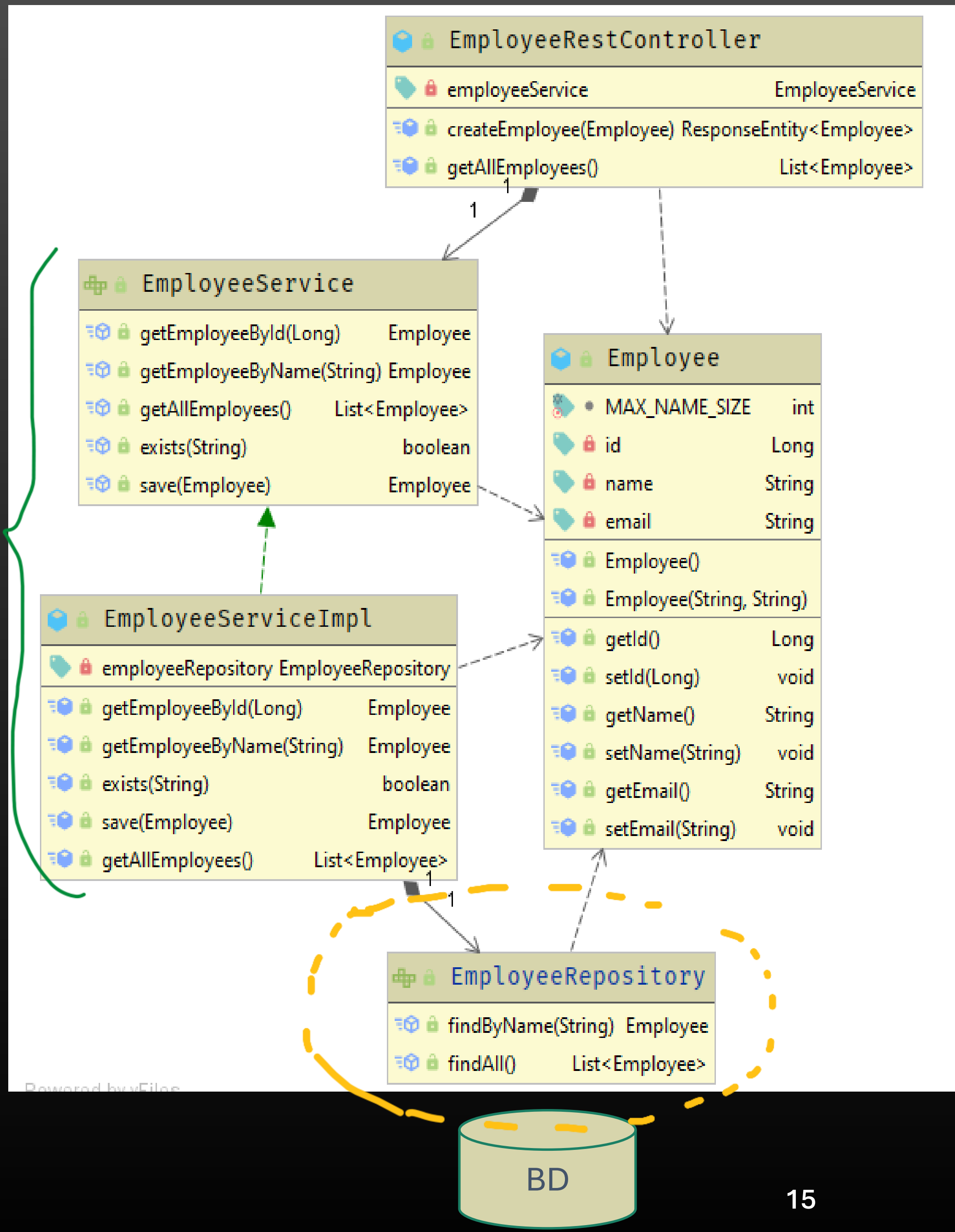
Test scope #3: service/domain logic

Scenario:

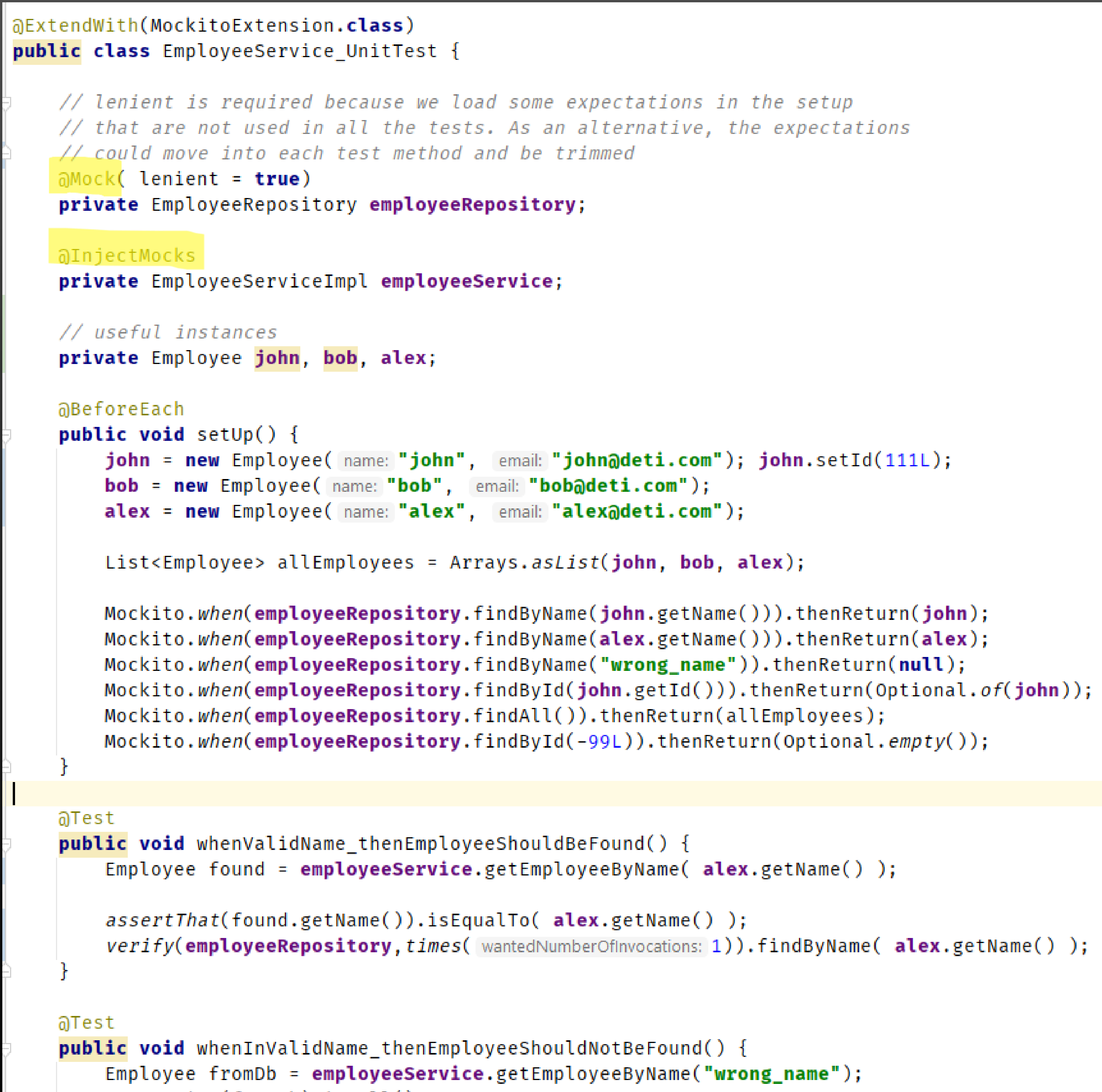
- Test the Service
- Focus on bizz logic and "high level" data use

Strategy:

- Make the test a standard JUnit test
- Mock dependencies on the data source provider (Repository)



Mock repository access



```java
@ExtendWith(MockitoExtension.class)
public class EmployeeService_UnitTest {

    // lenient is required because we load some expectations in the setup
    // that are not used in all the tests. As an alternative, the expectations
    // could move into each test method and be trimmed
    @Mock( lenient = true)
    private EmployeeRepository employeeRepository;

    @InjectMocks
    private EmployeeServiceImpl employeeService;

    // useful instances
    private Employee john, bob, alex;

    @BeforeEach
    public void setUp() {
        john = new Employee( name: "john",  email: "john@deti.com"); john.setId(111L);
        bob = new Employee( name: "bob",  email: "bob@deti.com");
        alex = new Employee( name: "alex",  email: "alex@deti.com");

        List<Employee> allEmployees = Arrays.asList(john, bob, alex);

        Mockito.when(employeeRepository.findByName(john.getName())).thenReturn(john);
        Mockito.when(employeeRepository.findByName(alex.getName())).thenReturn(alex);
        Mockito.when(employeeRepository.findByName("wrong_name")).thenReturn(null);
        Mockito.when(employeeRepository.findById(john.getId())).thenReturn(Optional.of(john));
        Mockito.when(employeeRepository.findAll()).thenReturn(allEmployees);
        Mockito.when(employeeRepository.findById(-99L)).thenReturn(Optional.empty());
    }

    @Test
    public void whenValidName_thenEmployeeShouldBeFound() {
        Employee found = employeeService.getEmployeeByName( alex.getName() );

        assertThat(found.getName()).isEqualTo( alex.getName() );
        verify(employeeRepository,times( wantedNumberOfInvocations: 1)).findByName( alex.getName() );
    }

    @Test
    public void whenInValidName_thenEmployeeShouldNotBeFound() {
        Employee fromDb = employeeService.getEmployeeByName("wrong_name");
```

I Oliveira

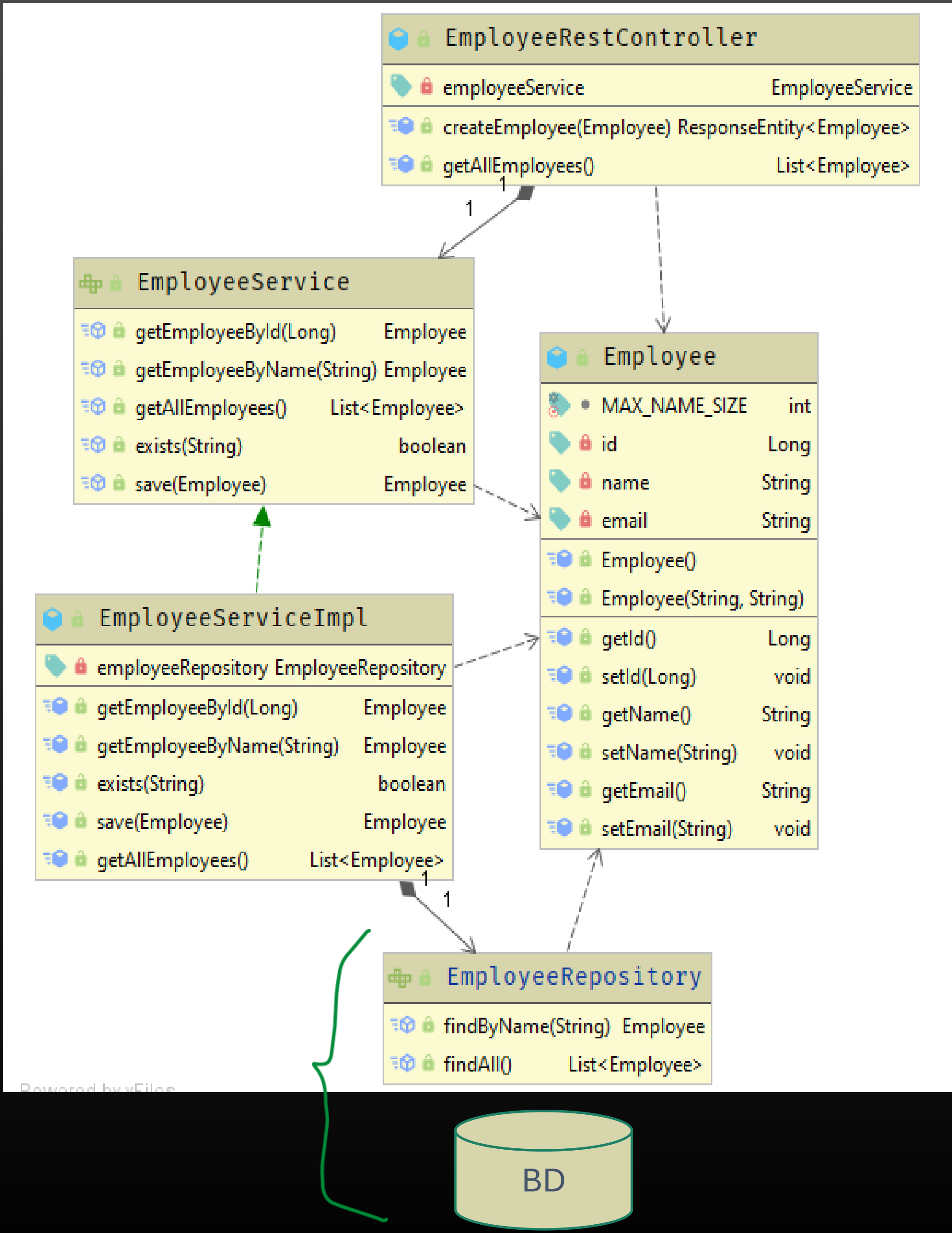
Test scope #4: data layer (JPA)

Scenario:

- Test the data access interface
- Focus on complex queries

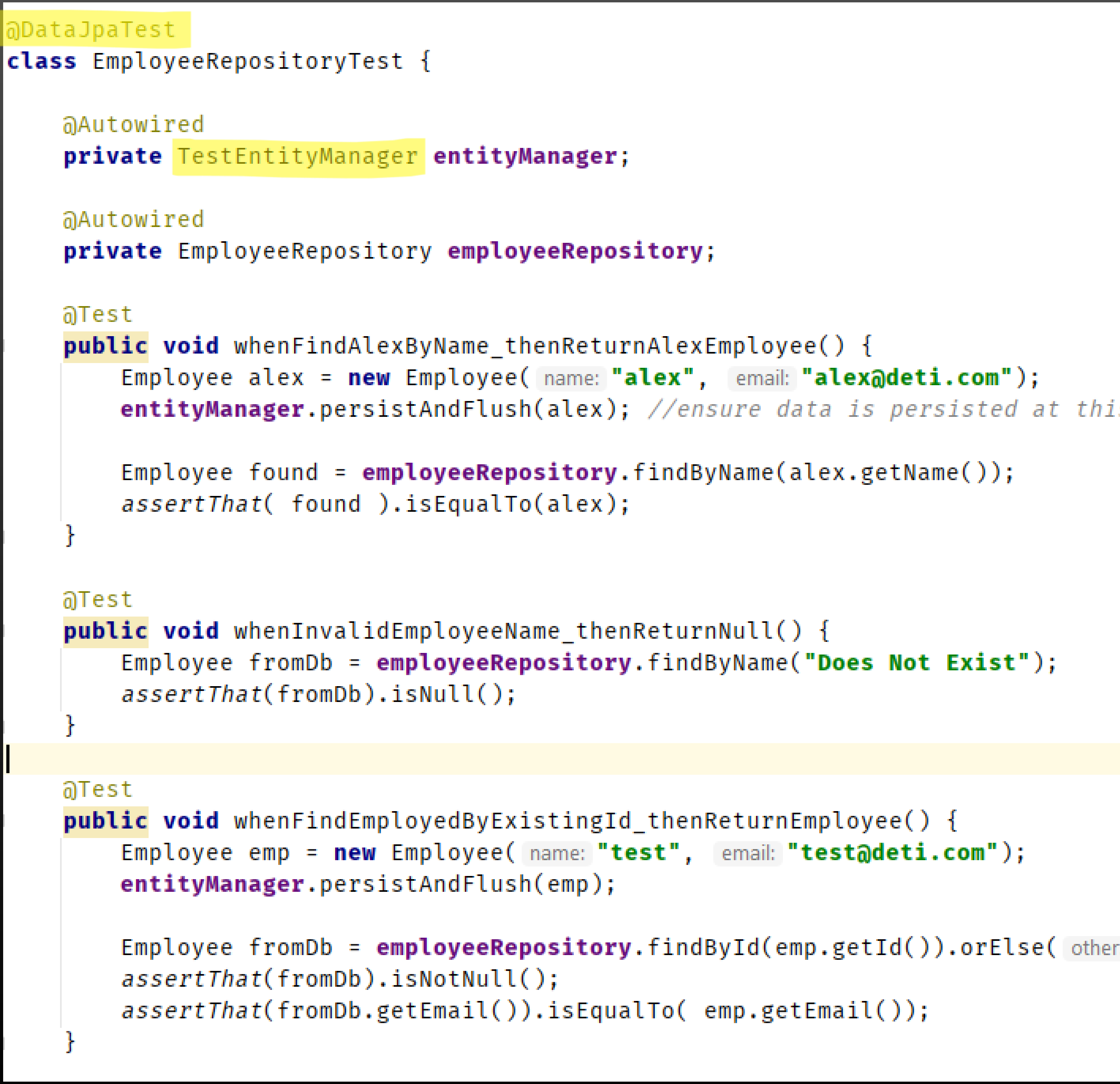
Strategy:

- Load only JPA-related instrumentation
- Use TestEntityManager



I Oliveira

Focus on JPA data access methods



```
@DataJpaTest
class EmployeeRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository employeeRepository;

    @Test
    public void whenFindAlexByName_thenReturnAlexEmployee() {
        Employee alex = new Employee( name: "alex",  email: "alex@deti.com");
        entityManager.persistAndFlush(alex); //ensure data is persisted at thi

        Employee found = employeeRepository.findByName(alex.getName());
        assertThat( found ).isEqualTo(alex);
    }

    @Test
    public void whenInvalidEmployeeName_thenReturnNull() {
        Employee fromDb = employeeRepository.findByName("Does Not Exist");
        assertThat(fromDb).isNull();
    }

    @Test
    public void whenFindEmployedByExistingId_thenReturnEmployee() {
        Employee emp = new Employee( name: "test",  email: "test@deti.com");
        entityManager.persistAndFlush(emp);

        Employee fromDb = employeeRepository.findById(emp.getId()).orElse( other
        assertThat(fromDb).isNotNull();
        assertThat(fromDb.getEmail()).isEqualTo( emp.getEmail());
    }
```

References

Spring.io docs

Testing the web layer

Eugen Paraschiv’s tutorials

Testing in Spring Boot