DAY - 11

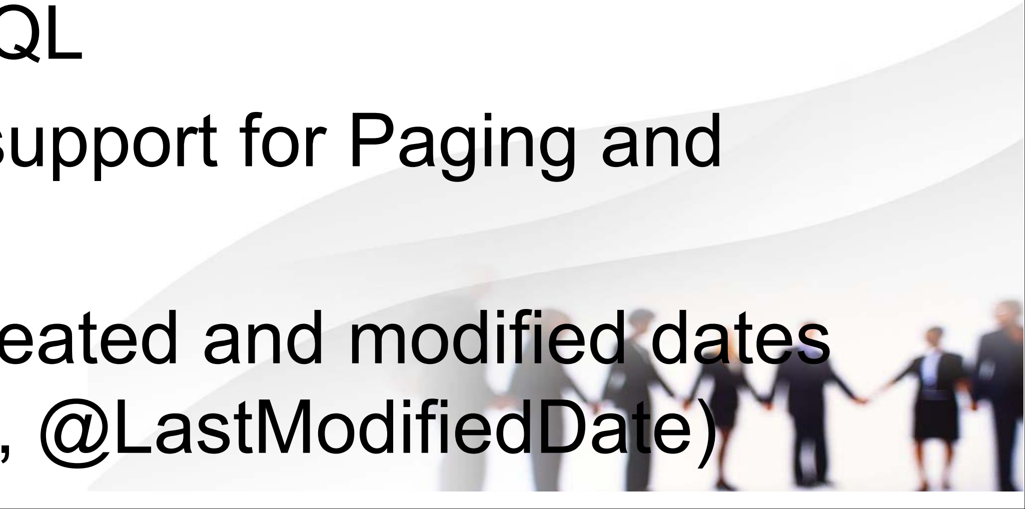
SPRING DATA JPA, TESTING, SWAGGER

SPRING DATA JPA

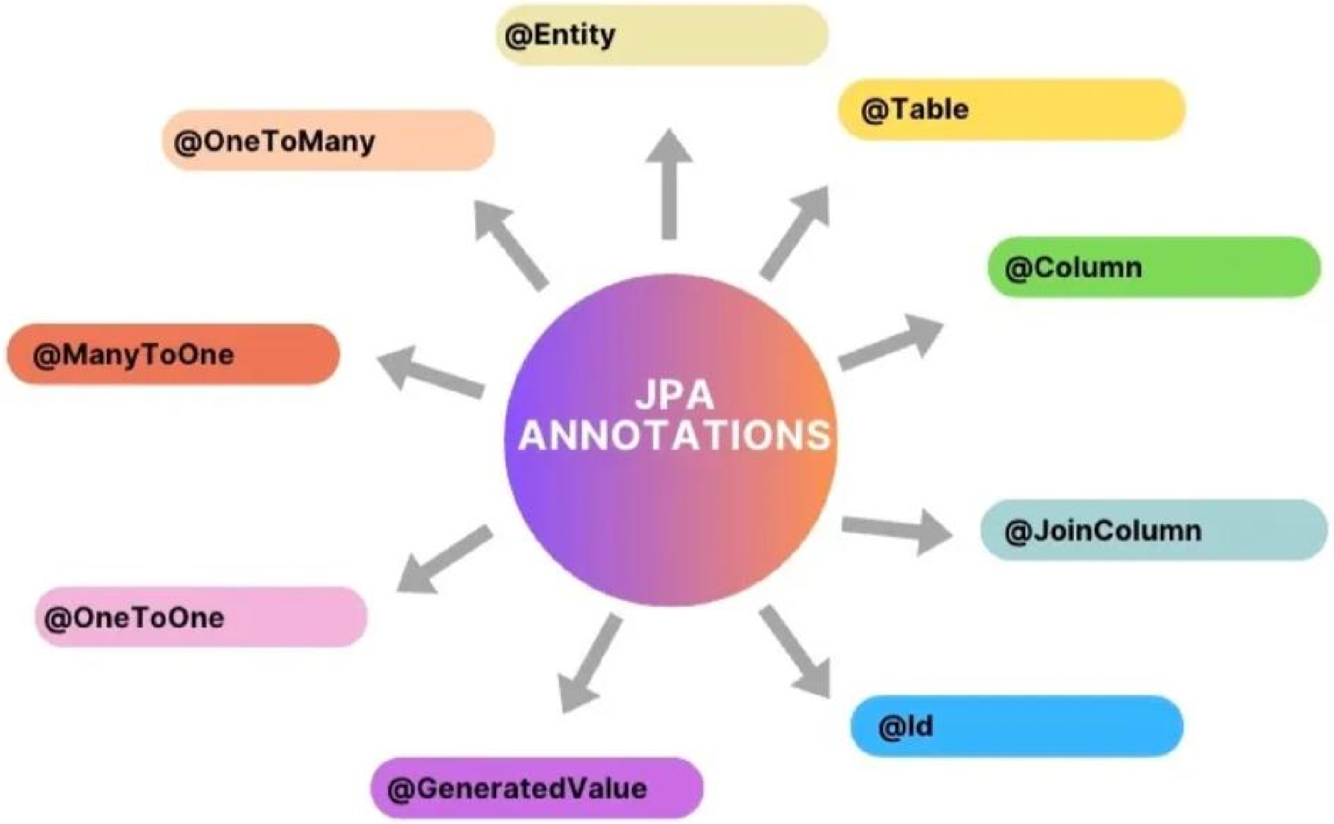
- part of Spring Framework
- JPA - Java Persistence API - Java specification for managing, persisting and accessing the data in Java applications
- simplify data access for Spring Applications
- provides high level abstraction over JPA std, making easier to work with the databases in Spring applns
- eleminates lot duplicate(boilerplate) code typically associated with common data access with CRUD operations

KEY FEATURES

- 1. Repository interfaces - create interface that define CRUD operations without actual implementations, Spring DATA JPA - dynamically generates the implementations at runtime.
- 2. Query methods - you can derive query methods from repository interface,Spring DATA JPA - dynamically generates the implementations at runtime.
- 3. Custom queries - JPQL , HQL , SQL
- 4. Pagination and Sorting - builtin - support for Paging and sorting
- 5. Auditing - Automatic tracking of created and modified dates for entities(@CreatedBy, @createdUser, @LastModifiedDate)

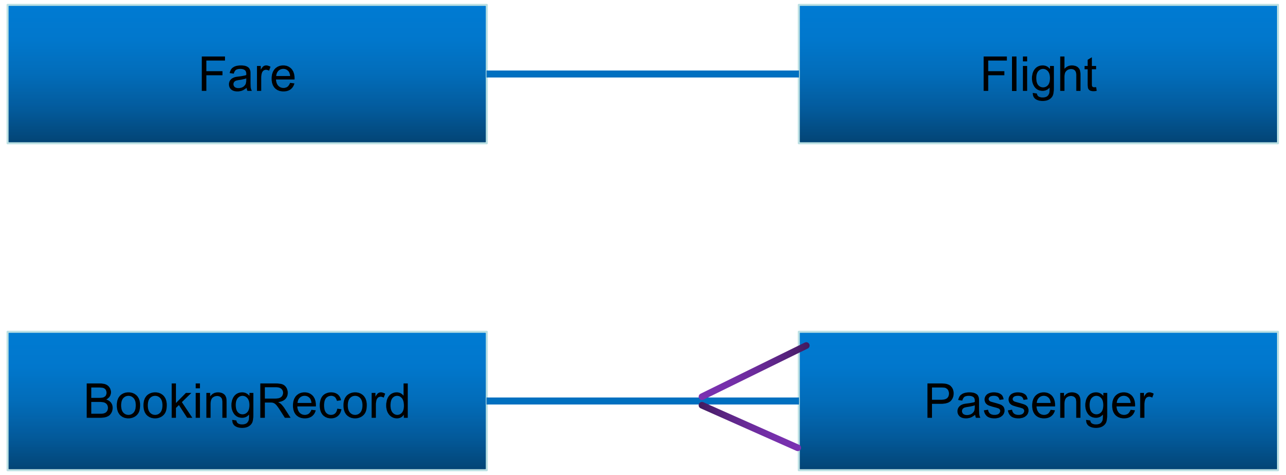


ANNOTATIONS



Example

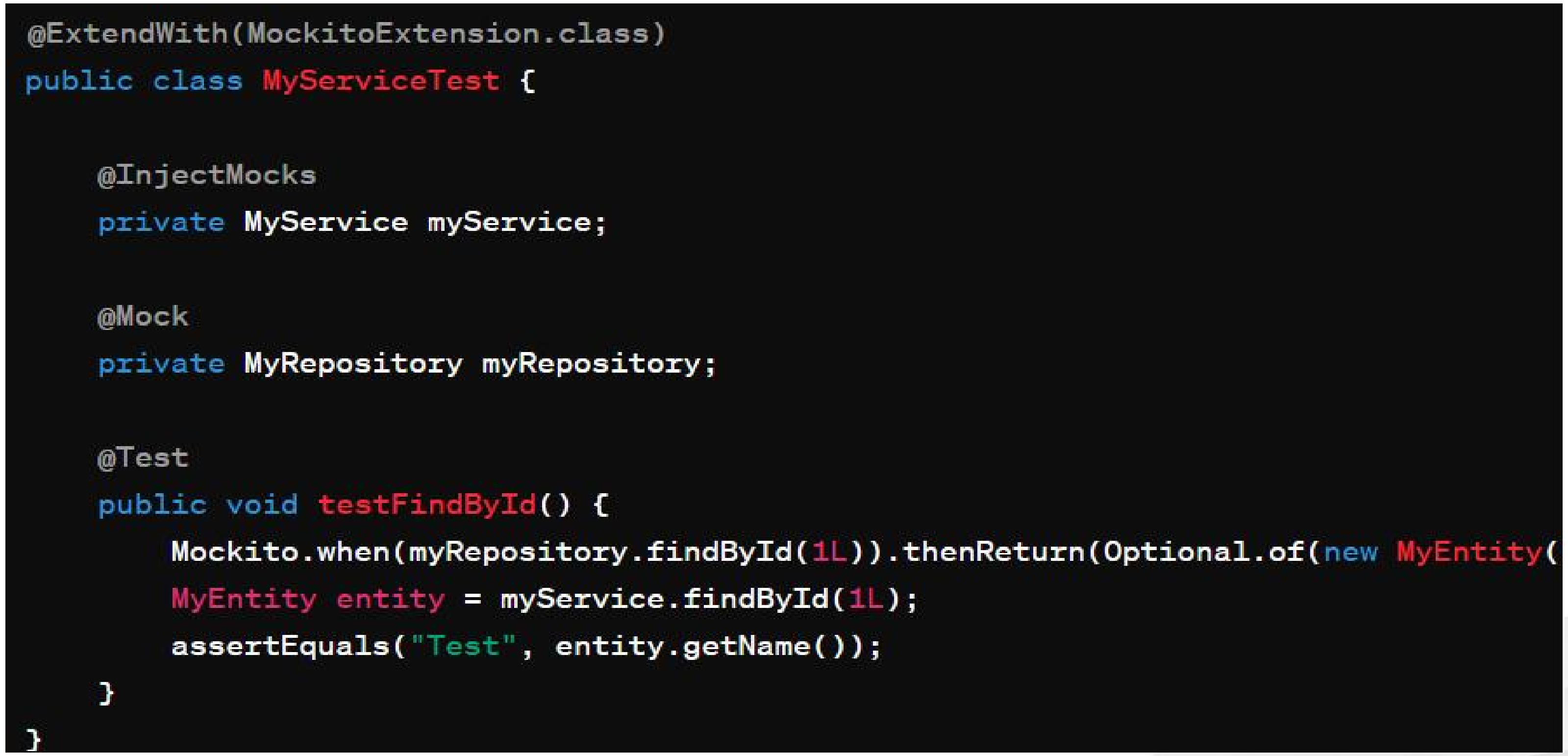
- one - to - one & one to many



SPRING BOOT TESTING

- Testing in Spring Boot is crucial for ensuring the correctness and reliability of your applications. There are several types of tests you can write in a Spring Boot application, including unit tests, integration tests, and end-to-end tests. Here's a brief overview of how to perform testing in Spring Boot:
- Unit Testing:
- Unit tests focus on testing individual components or units of your application in isolation. In Spring Boot, you can use JUnit and Mockito for writing unit tests.

unit testing

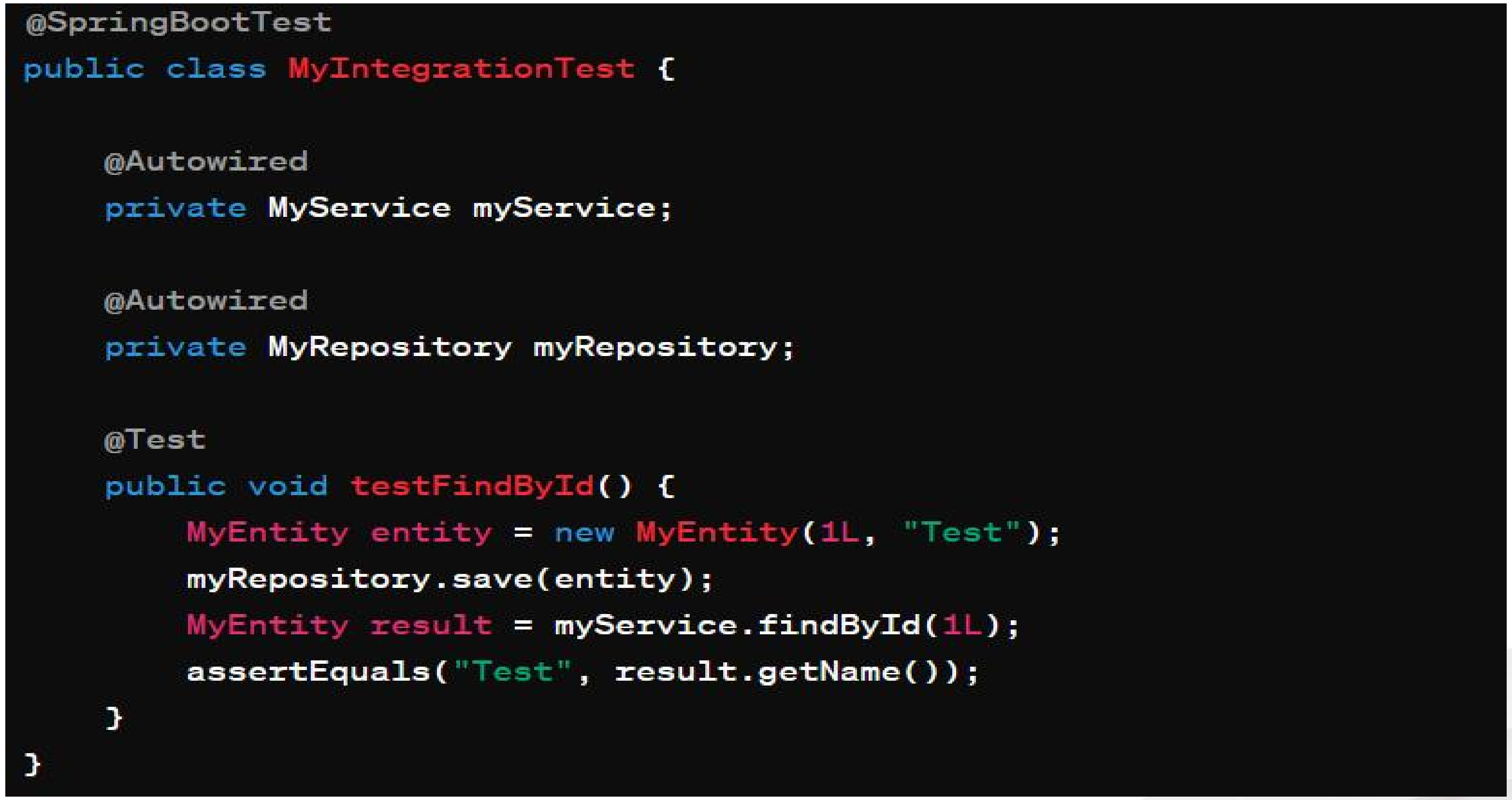


```java
@ExtendWith(MockitoExtension.class)
public class MyServiceTest {

    @InjectMocks
    private MyService myService;

    @Mock
    private MyRepository myRepository;

    @Test
    public void testFindById() {
        Mockito.when(myRepository.findById(1L)).thenReturn(Optional.of(new MyEntity(
        MyEntity entity = myService.findById(1L);
        assertEquals("Test", entity.getName());
    }
}
```

Integration Testing

- Integration tests verify the interactions between different components or layers of your application. Spring Boot provides @SpringBootTest annotation for creating integration tests. You can use @Autowired to inject dependencies.
- Example:

Integration testing



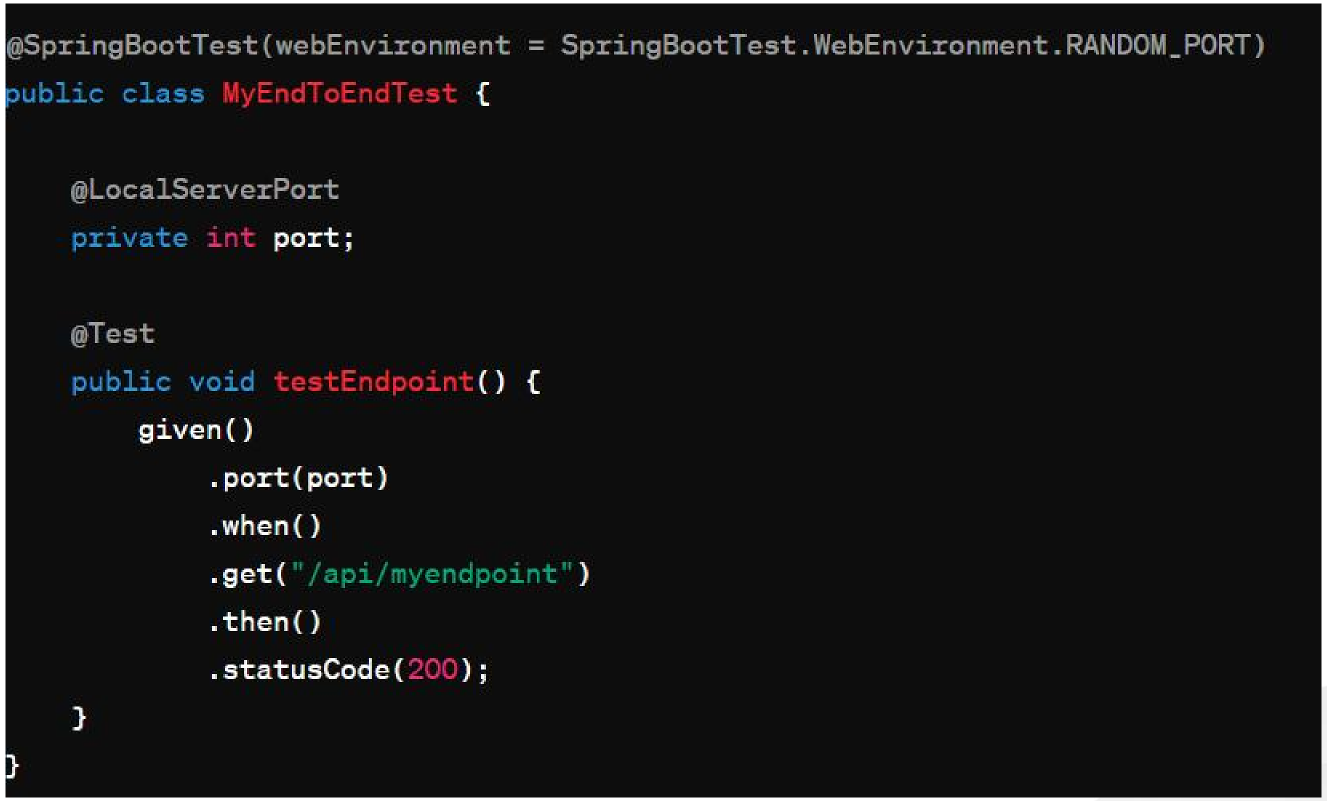
```java
@SpringBootTest
public class MyIntegrationTest {

    @Autowired
    private MyService myService;

    @Autowired
    private MyRepository myRepository;

    @Test
    public void testFindById() {
        MyEntity entity = new MyEntity(1L, "Test");
        myRepository.save(entity);
        MyEntity result = myService.findById(1L);
        assertEquals("Test", result.getName());
    }
}
```

End-to-End Testing:

- End-to-end tests validate the behavior of your entire application, including external dependencies. Tools like Selenium or REST Assured are commonly used for end-to-end testing in Spring Boot.

- Example (with REST Assured):

end-to-end testing



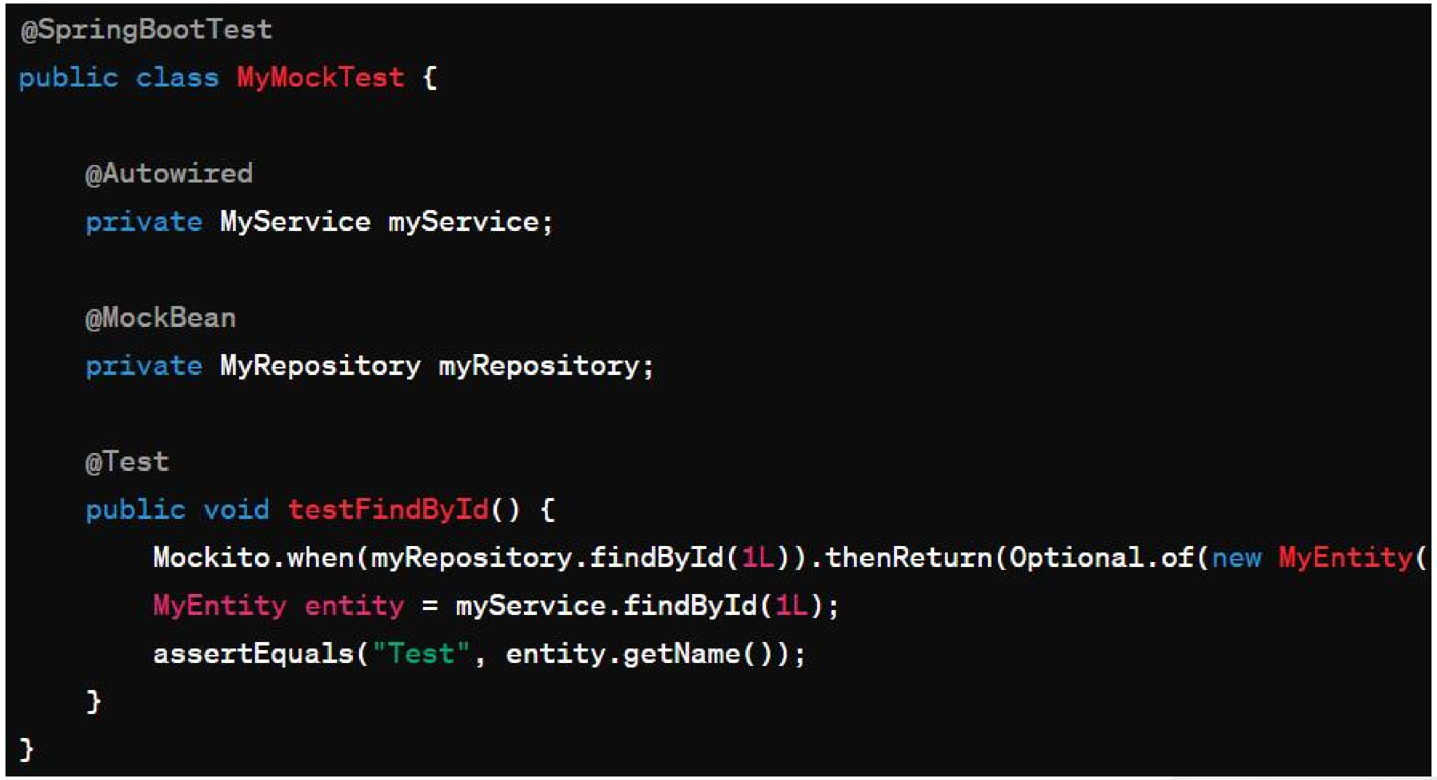
```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class MyEndToEndTest {

    @LocalServerPort
    private int port;

    @Test
    public void testEndpoint() {
        given()
            .port(port)
            .when()
            .get("/api/myendpoint")
            .then()
            .statusCode(200);
    }
}
```

Mocking Dependencies

- Mockito is commonly used for mocking dependencies in Spring Boot tests. You can use @MockBean annotation to replace beans with mocks in the Spring application context.

- Example:

mocking dependencies



```java
@SpringBootTest
public class MyMockTest {

    @Autowired
    private MyService myService;

    @MockBean
    private MyRepository myRepository;

    @Test
    public void testFindById() {
        Mockito.when(myRepository.findById(1L)).thenReturn(Optional.of(new MyEntity(
        MyEntity entity = myService.findById(1L);
        assertEquals("Test", entity.getName());
    }
}
```

mockito fundamentals

```
import static org.mockito.Mockito.*;

        //mock creation
        LinkedList mockedList = mock(LinkedList.class);

        //using mock object
        mockedList.add("one");
        mockedList.clear();

        //verification
        verify(mockedList).add("one");
        verify(mockedList).clear();
```

Repository Testing

- Testing repositories in Spring Boot typically involves testing CRUD (Create, Read, Update, Delete) operations along with custom query methods.
- You can use Spring Data JPA's @DataJpaTest annotation to focus your tests on JPA components only, which will speed up the test execution by loading only the relevant parts of the Spring context.

Repository Testing - steps

- We use @DataJpaTest annotation to configure the test for JPA components only. It will set up an in-memory database and only load components relevant for JPA testing. (@AutoConfigureTestDatabase) - class level annotations
- We autowire the XXXXRepository to perform CRUD operations.
- In each test method, we perform a specific operation (e.g., save a user, find a user by username) and then assert the expected outcomes.

Service Testing

- Testing Spring Boot services typically involves unit testing the individual methods within the service class. (integration testing)
- Annotate with @SpringBootTest,
- You can use JUnit along with Mockito to mock dependencies and verify the behavior of your service methods. Here's an example of how to write unit tests for a Spring Boot service:

Service Testing - steps

- @SpringBootTest - ServiceTest
- We're using @Mock to mock the XXXXRepository dependency.
- @InjectMocks is used to inject the mocked XXXRepository into the XXXXService.
- In each test method, we set up behavior for the mocked repository using Mockito.when() - stubbing.
- We then call methods on the XXXXService and assert on the expected behavior.

Controller/API Testing

– MockMvc for Integration Testing:

- Use Spring’s MockMvc to test your controllers in isolation. Write tests to verify the correct handling of HTTP requests and responses.

– Request and Response Validation:

- Check if the request parameters are correctly mapped and if the response is as expected.

MockMvc

- In Spring Boot testing with MockMvc, several frequently used methods are available to perform various actions and assertions on HTTP requests and responses. Here's a list of some commonly used methods in MockMvc:

MockMvc - Performing HTTP Requests

- perform(MockHttpServletRequestBuilder requestBuilder): Performs an HTTP request and returns a ResultActions object for further assertions.
  - get(String urlTemplate),
  - post(String urlTemplate),
  - put(String urlTemplate),
  - delete(String urlTemplate): Convenience methods to create GET, POST, PUT, and DELETE requests, respectively.
- contentType(MediaType mediaType): Sets the content type of the request.

MockMvc - Asserting Response Status

- andExpect(status().isOk()): Asserts that the HTTP response status is 200 (OK).
- andExpect(status().isNotFound()): Asserts that the HTTP response status is 404 (Not Found).
- andExpect(status().isCreated()): Asserts that the HTTP response status is 201 (Created).
- andExpect(status().isBadRequest()): Asserts that the HTTP response status is 400 (Bad Request).

MockMvc - Asserting Response Content

- andExpect(content().json(String jsonContent)): Asserts that the response body matches the provided JSON content.
- andExpect(content().contentType(MediaType mediaType)): Asserts that the content type of the response is the specified media type.

MockMvc - Asserting Response Body with JSONPath

- andExpect(jsonPath(String expression, Matcher<?> matcher)): Asserts that the specified JSONPath expression matches the expected value using a Hamcrest matcher.

MockMvc - Others

- Setting Request Content:
  - – content(String content): Sets the content of the request.
- Setting Request Parameters:
  - – param(String name, String... values): Sets request parameters.
- Setting Request Headers:
  - – header(String name, String... values): Sets request headers.

SWAGGER 3

- The OpenAPI specification defines the industry-standard specification for designing REST APIs, while Swagger provides a range of tools (Swagger Editor, Swagger UI, Swagger Codegen…) to support the development, testing, and documentation of these APIs. So we can think about Swagger 3 as OpenAPI 3 specification implementation.

springdoc-openapi

- Springdoc-openapi is a library that integrates with the Spring Boot framework to automatically generate OpenAPI documentation for REST APIs. It allows developers to describe their API endpoints and models using annotations and generates an OpenAPI specification in either JSON or YAML format.
- It also supports various features of the OpenAPI 3 specification, such as security definitions, schema validation, and JSON Web Token (JWT) authentication.
- Additionally, it integrates with other Spring Boot libraries, such as Spring WebMvc/WebFlux,Spring Data Rest, Spring Security and Spring Cloud Function Web, to automatically generate documentation for these components as well.

CONFIGURATION

- For Spring Boot 3:
- To use Swagger 3 in your Maven project, you need to add the springdoc-openapi-starter-webmvc-ui + security-starter dependency to your project’s pom.xml file:

- <dependency>
-   <groupId>org.springdoc</groupId>
-   <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
-   <version>2.0.3</version>
- </dependency>

Steps

- Spring boot version below 3:-
  - <dependency>
  -     <groupId>io.springfox</groupId>
  -     <artifactId>springfox-swagger2</artifactId>
  -     <version>3.0.0</version>
  - </dependency>

  - <dependency>
  -     <groupId>io.springfox</groupId>
  -     <artifactId>springfox-boot-starter</artifactId>
  -     <version>3.0.0</version>
  - </dependency>

  - Enabling the UI
    <dependency>
  -     <groupId>io.springfox</groupId>
  -     <artifactId>springfox-swagger-ui</artifactId>
  -     <version>3.0.0</version>
  - </dependency>

Steps

- SecurityConfig with
- @Configuration
- @EnableWebSecurity,
- public class SecurityConfig{
-         @Bean
-     public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
-         http.cors().disable().csrf().disable() .httpBasic();
-         return http.build();
-     }
- }
- OpenAPIConfig is optional one
- http://localhost:8089/swagger-ui/index.html

application.properties

- springdoc.swagger-ui.operationsSorter=method
- springdoc.swagger-ui.tagsSorter=alpha
- springdoc.swagger-ui.filter=true

- techacademy.openapi.dev-url=http://localhost:8089
- techacademy.openapi.prod-url=https://xxx-api.com

OpenAPIConfig

```
•    import java.util.List;
•    import org.springframework.beans.factory.annotation.Value;
•    import org.springframework.context.annotation.Bean;
•    import org.springframework.context.annotation.Configuration;
•    import io.swagger.v3.oas.models.OpenAPI;
•    import io.swagger.v3.oas.models.info.Contact;
•    import io.swagger.v3.oas.models.info.Info;
•    import io.swagger.v3.oas.models.info.License;
•    import io.swagger.v3.oas.models.servers.Server;
•    @Configuration
•    public class OpenAPIConfig {
•      @Value("${techacademy.openapi.dev-url}")
•      private String devUrl;
•      @Value("${techacademy.openapi.prod-url}")
•      private String prodUrl;
```

OpenAPIConfig

```
•       @Bean
•       public OpenAPI myOpenAPI() {
•         Server devServer = new Server();
•         devServer.setUrl(devUrl);
•         devServer.setDescription("Server URL in Development environment");

•         Server prodServer = new Server();
•         prodServer.setUrl(prodUrl);
•         prodServer.setDescription("Server URL in Production environment");

•         Contact contact = new Contact();
•         contact.setEmail("XXXX@gmail.com");
•         contact.setName("XXXx");
•         contact.setUrl("https://www.XXXX.com");

•
```

OpenAPIConfig

- License mitLicense = new License().name("MIT License").url("https://choosealicense.com/licenses/mit/");

- Info info = new Info()
- .title("Flight Ticket Booking API")
- .version("1.0")
- .contact(contact)
- .description("This API exposes endpoints to manage flight ticket bookings.").termsOfService("https://www.XXXX.com/terms")
- .license(mitLicense);

- return new OpenAPI().info(info).servers(List.of(devServer, prodServer));
- }
- }