Advanced JUnit Topics

Introduction

Advanced JUnit topics encompass a deeper exploration and understanding of sophisticated testing methodologies and techniques within the JUnit 5 framework. These topics are geared towards enhancing testing strategies and leveraging advanced features provided by JUnit.

Here's an overview of what Advanced JUnit Topics may cover:

- **Annotation Deep Dive**: A comprehensive exploration of various JUnit 5 annotations and their functionalities, Understanding their nuances, uses, and the scenarios they best cater to.

- **Mocking**: Deep dive into mocking frameworks (e.g. Mockito) for creating mock objects.

- **Test Coverage and Optimization**: Understanding code coverage metrics (statement, branch, path coverage) and their significance in testing.Strategies to improve code coverage through effective testing practices.

- **Integration Testing Strategies**: Exploring different approaches to integration testing using JUnit 5, such as testing with Spring Context, RESTful APIs, databases, etc. Understanding the balance between unit and integration tests for robust test suites.

- **Refactoring and Code Improvement**: Leveraging testing strategies to refactor code and improve code quality. Identifying code smells and optimizing code based on test outcomes.

- **Best Practices and Patterns**: Emphasizing best practices in testing, including naming conventions, structuring tests, and maintaining clean, readable test code. Understanding testing patterns for common scenarios like error handling, boundary conditions, etc.

Delving into these advanced topics aims to equip developers with a deeper understanding of JUnit 5 functionalities and advanced testing strategies. By mastering these concepts, developers can create more effective, maintainable, and reliable test suites, thus enhancing the overall software quality and development process.

Junit Annotations Simplified

@SpringBootTest vs @WebMvcTest

@SpringBootTest and *@WebMvcTest* are both annotations used in Spring Boot applications for testing, but they serve different purposes and have distinct scopes:

@SpringBootTest

- **Purpose:**
  - @SpringBootTest` is used for integration testing of the entire Spring context or a significant portion of it.
  - It loads the Spring application context, including all beans, configurations, and auto-configurations.
  - This annotation is suitable for comprehensive end-to-end testing, allowing tests to interact with the application's entire context, including repositories, services, controllers, etc.
- **Usage:**
  - It's often applied at the class level to test the entire Spring application or specific configurations.
  - Example: `@SpringBootTest(classes = YourApplication.class).
- **Scope:**
  - Broad scope: Loads the complete Spring context.
  - Slower in execution due to loading the entire application context.

@WebMvcTest

- **Purpose:**
  - @WebMvcTest` is specifically for testing the web layer of the application, focusing only on the MVC components
  - It loads a narrower slice of the application context, limited to the web layer beans such as controllers, advice, and related components.
  - It doesn't load the entire application context, making tests more focused and faster.
- **Usage:**
  - Applied at the class level, targeting specific controller classes for testing.
  - Example: `@WebMvcTest(YourController.class).

❖ **Scope:**

➢ Narrow scope: Limits the context to MVC components only (controllers, advices, etc.).
➢ Faster in execution due to loading a smaller portion of the application context.

When to Use Each

@SpringBootTest:

- Use for comprehensive integration testing of the entire application or substantial parts of it.
- Suitable for end-to-end testing, including interactions between different layers and components of the application.

@WebMvcTest:

- Use for focused testing of the web layer (controllers, MVC components).
- Ideal for testing controller endpoints and MVC-related behaviour without loading the entire application context, making tests faster and more specific.

In summary, *@SpringBootTest* is used for broader integration testing of the entire application. At the same time, *@WebMvcTest* is tailored explicitly for testing the web layer, providing a more focused and faster testing approach for MVC components.

@SpringBootTest with @AutoConfigureMockMvc (Integration testing)

Let's dive deeper into the details of integration testing using @SpringBootTest with @AutoConfigureMockMvc in Spring Boot with JUnit 5.

@SpringBootTest

@SpringBootTest is an annotation used in Spring Boot for integration testing. It loads the complete Spring application context for the test. Here's what it does:

1. **Loads Application Context**: It starts up the Spring application context by scanning your main application configuration (including all components, beans, and configurations) and sets up an environment similar to the production environment but tailored for testing purposes.

2. **Configures Test Environment**: This annotation initializes embedded databases, mocks certain services or beans if necessary, and sets up the environment to test Spring Boot applications without running a separate server.

@AutoConfigureMockMvc

@AutoConfigureMockMvc is used for web-layer testing in Spring Boot applications. It configures and auto-creates a MockMvc instance, enabling you to simulate HTTP requests and test your controllers without starting a real server. Here's what it does:

1. **Creates MockMvc Instance**: It auto-configures and provides a `MockMvc` instance, a simulation of the web environment, allowing you to perform HTTP requests and test your controllers as if they were handling actual requests.

2. **Simulates HTTP Requests**: You can use this `MockMvc` instance to simulate HTTP requests (GET, POST, PUT, DELETE, etc.) to your controllers and verify the responses, status codes, data returned, or how controllers interact with services.

Integration Testing with *@SpringBootTest* and *@AutoConfigureMockMvc*:

When used together, **@SpringBootTest** and **@AutoConfigureMockMvc** set up an integration test environment for your Spring Boot application, specifically focusing on testing the interaction between different layers (e.g., controllers, services, repositories) without the need to start a real server.

Here's an example scenario:

Suppose you have a Spring Boot application with *RESTful* endpoints handled by controllers interacting with services that communicate with a database. An integration test using these annotations allows you to:

1. **Simulate HTTP Requests**: Use the MockMvc instance to simulate HTTP requests to your RESTful endpoints defined in controllers.

2. **Validate Controller Behavior**: Test if the controllers handle requests properly, including input validation, handling of request parameters, and returning the expected responses.

3. **Mock Service Layer**: Potentially mock or provide test implementations of service classes to isolate the controller's behaviour and test the interaction between controllers and services.

4. **Verify End-to-End Functionality**: Ensure that the entire flow from the HTTP request to the service layer and, potentially, the database (if used) works as expected.

Integration testing with *@SpringBootTest* and *@AutoConfigureMockMvc* helps verify the collaboration of different parts of your application as a whole, ensuring that they integrate and function correctly.

@DataJpaTest

In Spring Boot, *@DataJpaTest* is an annotation for testing JPA repositories in an isolated environment. It's specifically designed to test the data layer of your Spring application by setting up a test slice that focuses on JPA entities and repositories.

@DataJpaTest

1. **Isolation of the Data Layer**: When you annotate your test class or test methods with *@DataJpaTest*, it sets up a specific slice of your Spring context that focuses only on the JPA-related components (entities, repositories, and their configurations).

2. **Loads JPA Configuration**: This annotation auto-configures an in-memory embedded database (such as H2) and initializes only the necessary Spring components for JPA, excluding other components like controllers, services, or other beans not directly related to JPA.

3. **EntityManager and Repository Initialization**: *@DataJpaTest* ensures that EntityManager is appropriately set up and initializes the JPA repositories, allowing you to write tests specifically targeting repository methods.

Here's an example demonstrating the usage of `@DataJpaTest` for testing a simple JPA repository in a Spring Boot application.

Suppose you have an entity called `User` and its corresponding repository `UserRepository`.

1. **User Entity**

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String email;

    // Constructors, getters, setters...
}
```

2. **UserRepository**

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
    // Additional custom queries or methods can be defined here
}
```

Now, let's create a test class that uses *@DataJpaTest* to test the functionality of the **UserRepository**.

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest
public class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    public void whenFindByUsername_thenReturnUser() {
        // Given
        User user = new User();
        user.setUsername("john_doe");
        user.setEmail("john@example.com");
        userRepository.save(user);

        // When
        User found = userRepository.findByUsername("john_doe");

        // Then
        assertThat(found).isNotNull();
        assertThat(found.getUsername()).isEqualTo(user.getUsername());
        assertThat(found.getEmail()).isEqualTo(user.getEmail());
    }

    // Other tests for UserRepository methods can be written similarly
}
```

Explanation of the test

- The *@DataJpaTest* annotation is used at the class level to set up the test slice specifically for JPA-related components.
- *@Autowired* is used to inject the UserRepository into the test class.
- The `*whenFindByUsername_thenReturnUser*` test method checks if the `UserRepository` correctly retrieves a user by username. It creates a user, saves it using the repository, and then verifies if the retrieval by username is working as expected.
- This test ensures that the `UserRepository` methods, especially the custom `findByUsername` method, function properly without the need to start up the entire Spring application context, focusing solely on the JPA-related components needed for testing

In summary, *@DataJpaTest* is a focused annotation in Spring Boot that creates an environment tailored for testing JPA entities and repositories. It's an excellent choice for writing efficient and targeted tests specifically for your data access layer without loading unnecessary components, allowing you to ensure that your repository methods behave as expected without involving other parts of your application.

Code Coverage

Code coverage refers to measuring how much of your source code is executed by your tests. It helps to determine which parts of your codebase are tested and which are not. It's a crucial metric in software testing to assess the effectiveness and thoroughness of your test suite.

Code Coverage in General:

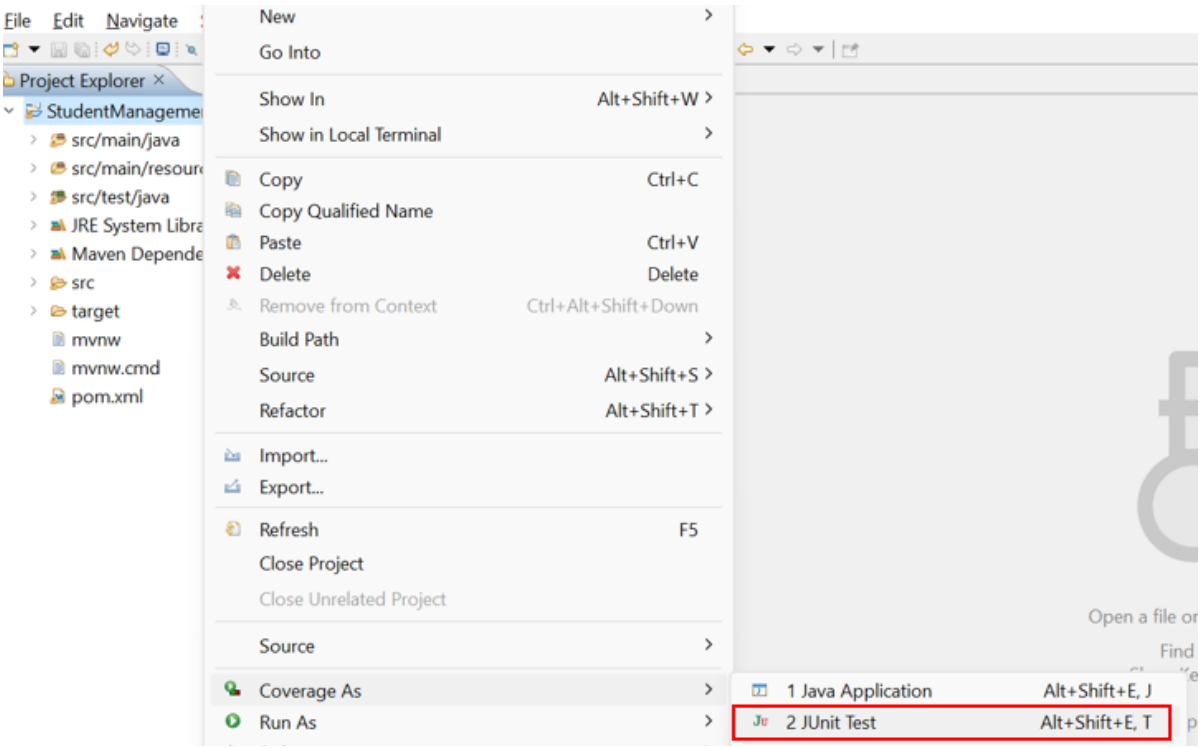
- **Statement Coverage**: It measures the number of executable statements in your code executed during testing. It's the most basic form of code coverage.
- **Branch Coverage**: It measures whether all possible branches in the code (like if-else conditions or switch statements) are executed at least once during testing.
- **Path Coverage**: This is the most comprehensive form of coverage, which aims to test every possible path through the code, including all possible combinations of branches.

Code coverage tools generate reports showing which lines, functions, branches, or paths have been exercised by your tests. It helps identify areas of the code that require additional testing and ensures a higher confidence in the reliability of your code.

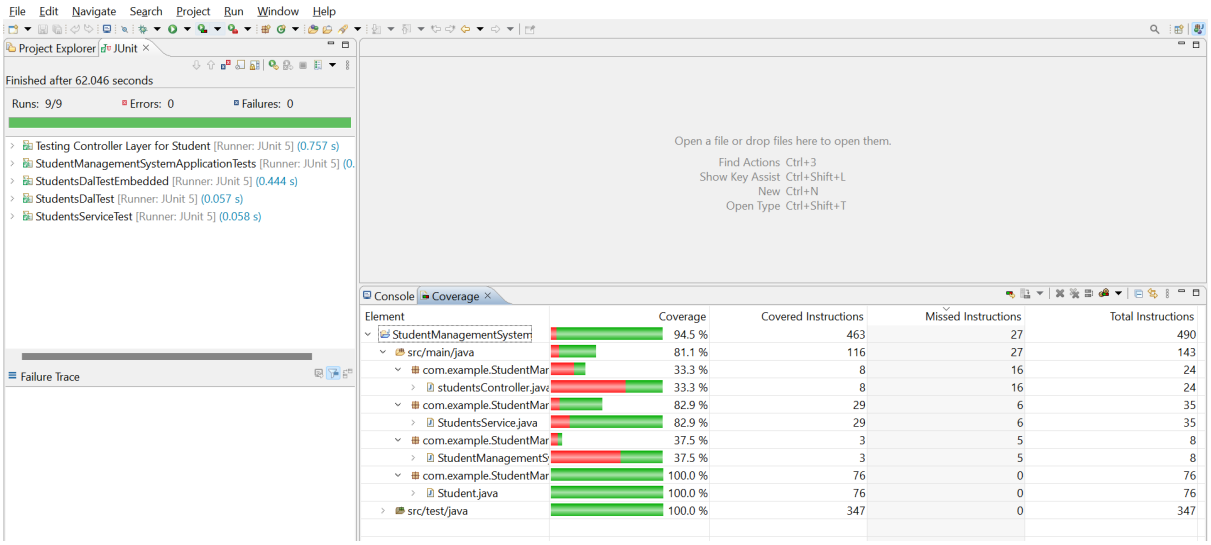
Code Coverage in Eclipse:

Let's examine our buddy application by utilizing the "Coverage As" feature in Eclipse.

- Proceed to your buddy application, right-click on it, and select "Coverage As JUnit Test" to execute the test cases while assessing coverage.

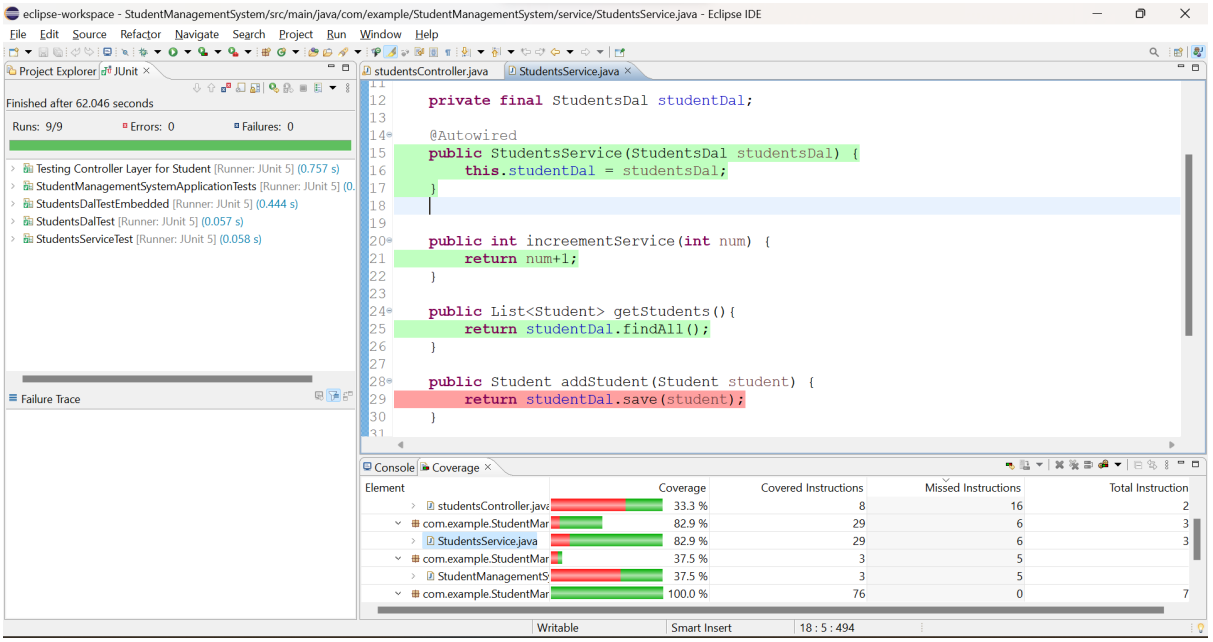


- After executing the test cases using the Coverage tool, you'll likely observe an output similar to the following:



❖ The coverage tool highlights your source code in different colours to represent different coverage levels:

➢ Green: Lines that your tests have fully executed.
➢ Yellow: Lines that have been partially executed.
➢ Red: Lines that haven't been executed at all.



❖ The tested code is often highlighted in green, indicating that it has been checked and proven to work correctly. On the other hand, untested code is highlighted in red, signalling that it hasn't been verified through testing and may have potential issues or errors.

❖ This visual distinction helps identify what parts of the codebase have been thoroughly examined for functionality and what areas might still need testing to ensure reliability and correctness.

❖ **Code Coverage**: It is a metric used in testing to measure the extent to which the source code of a program is tested by its associated test suite. It quantifies the proportion of code lines, branches, or paths executed during the testing process compared to the total lines, branches, or paths available in the codebase.

➢ Having a low coverage means that there are parts of the codebase that have not been tested or exercised by the test cases.

➢ A significant portion of the code has not been verified during testing, which could potentially lead to undetected bugs or errors within those untested sections.

- **Covered Instructions**: These represent the lines of code or instructions executed or accessed during the test execution. Covered instructions indicate the parts of the code exercised by the test cases.

- **Missed Instructions** refer to the lines of code or instructions not executed or accessed during the test execution. Missed instructions indicate areas of the code not tested or validated by the test cases.

- **Total Instructions**: This refers to the entire set of lines of code or instructions present in the codebase that could be executed. It represents the total scope of the code under consideration for testing.

References:

1. Official Website
2. Eclipse
3. Baeldung

ObjectMapper

In a Spring Boot application, when testing code involving JSON serialization or deserialization, you might use the ObjectMapper provided by Jackson, a widely used library for working with JSON in Java.

When testing with JUnit in a Spring Boot environment, you can configure and use ObjectMapper instances to test JSON serialization and deserialization within your tests.

Here's an example of how you might use **ObjectMapper** in a JUnit test in the Buddy Spring Boot application:

```
package com.example.StudentManagementSystem.controller;

import com.example.StudentManagementSystem.model.Student;
import com.example.StudentManagementSystem.service.StudentsService;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
```

```
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

@WebMvcTest(controllers = studentsController.class)
@DisplayName("Testing Controller Layer for Student")
public class studentControllerTest {
	@Autowired
	private MockMvc mockMvc;
	@Autowired
	ObjectMapper objectMapper;
	@MockBean
	StudentsService studentsService;

	@Test
	@DisplayName("/students/addStudent")
	void shouldAddStudent() throws Exception{

		Student student = new Student(1,
				"Rakesh",
				19,
				"JUnit",
				"Aryabhatta Hostels",
				"rakesh@gmail.com",
				"09874562134");


Mockito.when(studentsService.addStudent(student)).thenReturn(student);
		String json = objectMapper.writeValueAsString(student);


mockMvc.perform(MockMvcRequestBuilders.post("/students/addStudent")
		.contentType("application/json").content(json))
		.andExpect(MockMvcResultMatchers.status().isOk());

	}

}
```

In the provided JUnit test, the primary focus is testing the *addStudent()* method in the **studentsController** class. The test focuses on verifying whether the controller correctly handles adding a new student by sending a **POST** request with JSON data.

Here's an explanation with a primary focus on the usage of **ObjectMapper** and the test code:

Test Setup Explanation:

- ★ **@WebMvcTest**: Indicates that this test focuses only on the web layer by restricting the configuration to the studentsController class.

- ★ **@MockBean StudentsService**: Mocks the StudentsService dependency, allowing controlled behaviour during testing.

- ★ **MockMvc**: Represents the Spring MVC infrastructure for simulating HTTP requests and responses.

- ★ **ObjectMapper:** Converts Java objects to JSON and vice versa for request and response handling.

Test Execution Explanation:

1. **Student Creation**
   A Student` object is created with mock data.

2. **Mocking Service Behavior**
   Mockito.when(studentsService.addStudent(student)).thenReturn(student); sets up the mock behaviour. When the *addStudent* method from the *StudentsService* is called with a student object, it returns the same student object.

3. **Object to JSON Conversion**:
   String json = objectMapper.writeValueAsString(student); converts the *Student* object into a JSON string using the **ObjectMapper**.

4. **Performing Mock HTTP Request**:
   mockMvc.perform(MockMvcRequestBuilders.post("/students/addStudent") initiates a POST request to the endpoint "*/students/addStudent*" using the *MockMvc*.

5. **Request Configuration**:
   .contentType("application/json").content(json))` configures the request content type as JSON and sets the JSON string as the requested content.

6. **Expectation**:
   .andExpect(MockMvcResultMatchers.status().isOk())` ensures that the expected result of the request is an HTTP 200 OK status.

Overall: The primary goal of this test is to simulate a POST request to the `/students/addStudent` endpoint with JSON data representing a student object. It then verifies that the controller returns an HTTP status code of 200 (OK) for the successful addition of the student. The usage of **ObjectMapper** aids in converting the `Student` object into a JSON string for the requested content, allowing the test to validate the controller's behaviour in handling incoming JSON data for adding a student.

Mockito

Mockito is a popular Java framework for mocking and stubbing in unit tests. It allows you to simulate the behaviour of dependencies and focus on testing specific parts of your code in isolation.

Basic Concepts

- **Mocking**: Creating simulated objects that mimic the behaviour of real objects. Mockito provides methods to create mocks of interfaces and classes.

- **Stubbing**: Configuring mock objects to return specific values or behave in certain ways when their methods are called.

Example of Mocking and Stubbing:

Let's say you have a `*Calculator*` interface and a `*CalculatorService*` class that depends on this interface:

```
public interface Calculator {
    int add(int a, int b);
    int subtract(int a, int b);
}


public class CalculatorService {
    private Calculator calculator;

    public CalculatorService(Calculator calculator) {
        this.calculator = calculator;
    }

    public int addTwoNumbers(int a, int b) {
        return calculator.add(a, b);
    }

    public int subtractTwoNumbers(int a, int b) {
        return calculator.subtract(a, b);
    }
}
```

Mock Creation and Stubbing:

Here's how you can use Mockito to create a mock of the `*Calculator*` interface and stub its behaviour in a test:

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

public class CalculatorServiceTest {

    @Test
    public void testAddTwoNumbers() {
        // Create a mock of the Calculator interface
        Calculator calculatorMock = mock(Calculator.class);

        // Stubbing: Define behavior for the add method
        when(calculatorMock.add(2, 3)).thenReturn(5);

        // Create a CalculatorService instance with the mock
                            CalculatorService    calculatorService    =    new
CalculatorService(calculatorMock);

        // Test the CalculatorService method
        int result = calculatorService.addTwoNumbers(2, 3);

          // Verify that the add method of the mock was called with specific
arguments
        verify(calculatorMock).add(2, 3);

        // Check if the result matches the expected value
        assertEquals(5, result);
    }
}
```

In this example:

- **mock(Calculator.class)** creates a mock instance of the `*Calculator*` interface.
- **when(calculatorMock.add(2, 3)).thenReturn(5)** stubs the `*add*` method of the mock to return 5 when called with arguments 2 and 3.
- **verify(calculatorMock).add(2, 3)** verifies that the `*add*` method of the mock was called with specific arguments.

This test demonstrates how to create a mock, stub its behaviour, call a method using the mock, and verify that the method was invoked as expected.

when().thenReturn Vs Mockito.when().thenReturn

The difference between `when().thenReturn` and `Mockito.when().thenReturn()` lies in how you access the `when()` method.

when().thenReturn

- **Usage**: Involves importing the `when()` method statically from `**org.mockito.Mockito**`.
- **Example**

```
import static org.mockito.Mockito.*;
 // Usage
 when(mockedList.size()).thenReturn(10);
```

- **Explanation**: Allows direct usage of `when()` without explicitly referencing `Mockito`.

Mockito.when().thenReturn()

- **Usage**: It Involves explicitly calling the `when()` method from `**org.mockito.Mockito**`.
- **Example:**

```
import org.mockito.Mockito;
   // Usage
   Mockito.when(mockedList.size()).thenReturn(10);
```

- **Explanation: It** requires explicit invocation of methods from the `Mockito` class.

The core functionality, setting up behaviour for mocked methods using `thenReturn()`, remains the same in both cases. The difference lies in how you access the `when()` method—either via a static import or by explicitly mentioning the `Mockito` class. It's a matter of coding preference and style in your test code.

Mockito.when().then Vs Mockito.when().thenReturn

In Mockito, both `when().then` and `when().thenReturn()` are used for stubbing method behaviours in mock objects, but they have slight differences in their usage.

when().then

- The `when().then` structure in Mockito is a more generic approach that allows you to define behaviour for methods where you need to execute custom logic or perform actions but do not explicitly return a value.

For example:

```
Mockito.when(mockedList.size()).then(invocation -> {
    // Custom logic before returning a value
    System.out.println("Calculating size...");
    return 10;
});
```

Here, instead of simply returning a value, you can execute custom code before returning the result.

when().thenReturn()

- The `when().thenReturn()` is specifically used to stub method invocations when you want to define a return value for that method.

For example:

```
Mockito.when(mockedList.size()).thenReturn(10);
```

This method is used when you directly want to set a specific return value without executing any additional custom logic. It's typically used for methods that return a value.

Both methods are useful in different scenarios. Use `then` when you need to perform custom logic, and `thenReturn` when you only need to define the return value without additional operations.

Testing Spring Security with JWT

Here we are using the **Gym III** Application from the lecture User Persistence and JWT Authentication to demonstrate testing for JWT generation and validation.

```
@SpringBootTest
@AutoConfigureMockMvc
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
@DisplayName("Testing GymSecurityConfig")
class GymSecurityConfigTests {
    @Autowired
    MockMvc mockMvc;
    @Autowired
```

```
    UserService userService;
    @Autowired
    ObjectMapper objectMapper;
    String jwtTokenAdmin = null;
    UserRequest admin = new UserRequest();

    @BeforeAll
    void setup() {
        admin.setEmail("admin@gmail.com");
        admin.setGender("MALE");
        admin.setPassword("admin12");
        admin.setUserType("admin");
        admin.setAge(22);
        userService.createUser(admin);
    }
    @Test
    @DisplayName("Generating 'json web token' for user with role ADMIN")
    @Order(1)
    void testLoginAdmin() throws Exception{
        JwtRequest adminLoginRequest = JwtRequest.builder()
                .username(admin.getEmail())
                .password(admin.getPassword())
                .build();

                                        MvcResult     loginAdminResult     =
mockMvc.perform(MockMvcRequestBuilders.post("/auth/login")
                        .contentType("application/json")

.content(objectMapper.writeValueAsString(adminLoginRequest)))
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andReturn();

        jwtTokenAdmin = extractJwtToken(loginAdminResult);
        Assertions.assertNotNull(jwtTokenAdmin);
    }

    @Test
    @Order(2)
    @DisplayName("Testing 'json web token' for user with role ADMIN")
    void testValidAccessAdmin() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/gym/all")
                        .header(HttpHeaders.AUTHORIZATION,  "Bearer  "  +
jwtTokenAdmin)

.contentType("application/json")).andExpect(MockMvcResultMatchers.status
().isOk());
```

```
    }

        private String extractJwtToken(MvcResult loginResult) throws
Exception {
                                        String responseJson =
loginResult.getResponse().getContentAsString();
          JwtResponse jwtResponse = objectMapper.readValue(responseJson,
JwtResponse.class);
        return jwtResponse.getJwtToken();
    }
```

Explanation:

Let's discuss the above code and understand its components:

Test Class Overview: This test class is intended to perform integration testing for the security configurations and endpoints of a Spring Boot application related to gym functionalities.

Annotations Used:

- `**@SpringBootTest**`: Indicates that this is a Spring Boot test and loads the complete application context for testing purposes.
- `**@AutoConfigureMockMvc**`: Automatically configures the `MockMvc` instance, which is used to simulate HTTP requests and interactions.
- `**@TestInstance(TestInstance.Lifecycle.PER_CLASS)**`: Specifies that test instances should be created once per test class.
- `**@TestMethodOrder(MethodOrderer.OrderAnnotation.class)**`: Orders the test methods based on the value provided by the `@Order` annotation.
- `**@DisplayName**`: Provides a custom display name for the test class or test method to be shown in test reports.

Dependency Injection

- `**@Autowired MockMvc mockMvc**`: Injects the `MockMvc` instance for simulating HTTP requests.
- `**@Autowired UserService userService**`: Injects the `UserService` for performing user-related operations.
- `**@Autowired ObjectMapper objectMapper**`: Injects the `ObjectMapper` to serialize and deserialize Java objects to/from JSON.

Test Setup (`setup()` method)

- `**@BeforeAll void setup()**`: Annotated method to run once before all test methods.

- Creates an `admin` user using `UserService`.

Test Methods

- **`testLoginAdmin()`**

    - **Purpose**: Tests the login functionality for a user with the role `**ADMIN**`.
    - **Workflow**

        - Builds a login request (`JwtRequest`) for the `admin` user.
        - Performs a POST request to the `/auth/login` endpoint with the admin's credentials.
        - Verifies that the response status is successful (HTTP 200).
        - Extracts the JSON Web Token (`jwtTokenAdmin`) from the response.
        - Asserts that the `jwtTokenAdmin` is not null.

- **`testValidAccessAdmin()`**

    - **Purpose**: Tests if a user with the role `**ADMIN**` can access a specific endpoint.
    - **Workflow**:
        - Performs a GET request to the `/gym/all` endpoint.
        - Adds the `jwtTokenAdmin` to the authorization header.
        - Verifies that the response status is successful (HTTP 200).

- **Utility Method:** `private String extractJwtToken(MvcResult loginResult) throws Exception`: Private method used to extract the JWT token from the response of the login request (`/auth/login` endpoint).

Test Flow

- **Setup**: Before the tests, the `setup()` method creates an `admin` user in the database using `UserService`.

- `**testLoginAdmin()**`: Checks if the admin login works, retrieves a valid JSON Web Token for the admin user and asserts its existence.

- `**testValidAccessAdmin()**`: Verifies if an admin user with a valid token can access the `/gym/all` endpoint successfully.

Summary

- These tests focus on validating the authentication and authorization aspects for an admin user within the application.

- The `**MockMvc**` instance allows the simulation of **HTTP** requests and interaction with endpoints.
- The `**UserService**` is utilized to create a test user.
- **ObjectMapper** assists in handling JSON serialization and deserialization.
- Assertions **verify** the expected behaviour and responses from the tested endpoints.

References:

1. Integration Testing
2. Official Website
3. Baeldung
4. Junit5