More on Junit Testing

Introduction

JUnit is a widely used testing framework for Java applications. It provides a robust platform for developers to create and execute unit tests, ensuring the reliability and correctness of their code.

1. **Unit Testing**: Unit testing involves testing individual units or components of code to ensure they perform as expected. A unit can be a method, class, or a small functional code.

2. **Key Features of JUnit:**

   ★ **Annotations:** JUnit uses annotations to identify methods that specify test cases, setup, teardown, etc. (*@Test, @Before, @After*)
   ★ **Assertions:** Assertion methods validate expected outcomes against actual results (*assertEquals, assertTrue, assertNull*).
   ★ **Exception Handling:** Testing code behaviour when specific exceptions are thrown.

3. **Integration with Frameworks:** JUnit is widely integrated with various frameworks like Spring Boot, allowing developers to perform testing in different contexts like web applications, microservices, etc.

4. **Benefits of JUnit Testing**:

   ★ Facilitates early bug detection and debugging.
   ★ Ensures code quality and maintainability.
   ★ Supports test-driven development (TDD) practices.
   ★ Provides a standardized approach to testing.

Overall, JUnit plays a vital role in maintaining code quality by allowing developers to write and execute tests effectively, contributing to the robustness and reliability of Java applications.

@Order

In JUnit 5, the **@Order** annotation allows you to specify the execution order of test methods within a test class or the order of test classes in test suites. It's advantageous when you need deterministic test execution sequences, especially in cases where tests have dependencies or when specific setup and teardown procedures are necessary.

Here's an example of how **@Order** can be used in **JUnit 5**:

Ordering Test Methods within a Test Class:

```java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.MethodOrderer;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.TestMethodOrder;

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)   //   Specifying   the
order of test methods
public class MyTestClass {

    @Test
    @Order(1) // Specifies the order of execution for this test method
    public void firstTest() {
        // Test logic
    }

    @Test
    @Order(2)
    public void secondTest() {
        // Test logic
    }
}
```

Key Points:

★ **@Order** annotation can be applied at the method level to define the execution order for test methods within a test class.

★ For specifying the order of test classes within a test suite, **@Order** is used at the class level.

★ It accepts an integer value to determine the execution sequence, with lower values indicating earlier execution.

★ Test methods or classes with the same order value are executed arbitrarily.

★ The **@TestMethodOrde**r annotation with **MethodOrderer.OrderAnnotation.class** is used to define the order of test methods.

★ It's part of the JUnit Jupiter API in JUnit 5 and allows for deterministic test execution sequences.

By using **@Order** in JUnit 5, you can ensure the execution order of tests, providing more control and predictability in the test execution process.

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:

```java
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 request 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.

Note:

When using version **2.7.17** / **2.7.16** of Spring and due to multiple dependencies in the projects, such as **Spring Security using JWT**, the autowire approach for using mockMvc must be modified. Otherwise, it fails to mock all the dependencies and returns an erroneous state.

```
@WebMvcTest(controllers = UserController.class)
@DisplayName("Testing UserController")
public class UserControllerTests {
    @MockBean
    private UserService userService;
    @MockBean
    private JwtAuthenticationHelper jwtAuthenticationHelper;

    private MockMvc mockMvc;
    @Autowired
    private WebApplicationContext webApplicationContext;

    @BeforeEach
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }

    @Test
    @DisplayName("Testing API: 'GET /user/all'")
    @WithMockUser(username = "admin", roles = "ADMIN")
    void shouldTestGetAllUsers() throws Exception {
        User user = new User();
        user.setId(1L);
        user.setEmail("user1");
        Mockito.when(userService.getAllUsers()).thenReturn(List.of(user));
        mockMvc.perform(get("/user/all")
                        .contentType("application/json"))
                .andExpect(status().isOk());
    }
}
```

- **MockMvcBuilders.webAppContextSetup() Context**: This sets up a more extensive web application context, potentially including more beans than the limited context loaded by **@WebMvcTest**.

- The demonstrated approach can be extended to test various HTTP request methods like **POST**, **GET**, **PUT**, and **DELETE** for endpoints within a Spring application.

References:

1. Official Website
2. MockMvc
3. Baeldung