Junit Testing

Introduction

JUnit is a popular and widely used open-source testing framework for Java. It's specifically designed to help developers write and run unit tests effortlessly. Some key points about JUnit include:

- **Purpose**
  - **Unit Testing**: JUnit is primarily used for writing and running automated tests at the unit level, focusing on individual components or modules of code in isolation.
- **Features**
  - **Annotations**: JUnit leverages annotations (`@Test`, `@Before`, `@After`, etc.) to define test methods, setup, and teardown actions.
  - **Assertions**: Provides a set of assertion methods (e.g., `assertEquals`, `assertTrue`, `assertNotNull`) to verify expected outcomes.
  - **Test Runners**: Offers different test runners for running tests (`BlockJUnit4ClassRunner`, `Parameterized`, etc.).
  - **Parameterized Tests**: Allows running the same test method with different sets of parameters.
  - **Assertions Library**: Offers a rich set of assertion methods for validating conditions.
- **Test Lifecycle**
  - **Setup and Teardown**: Methods annotated with `@Before` are executed before each test method, while those annotated with `@After` are executed after each test method.

JUnit simplifies the process of writing, organizing, and executing automated tests in Java, promoting good software development practices by encouraging Test-Driven Development (TDD) and ensuring code reliability through continuous testing.

Annotations in Junit

Here's a brief explanation of several commonly used annotations in JUnit 5 testing:

Core Annotations:

`@Test`:

- Marks a method as a test method.
- Used to identify methods that should be executed as tests.

`@DisplayName`:

- Provides a custom display name for test classes or test methods in test reports.
- Improves readability and understanding of test cases.

`@Order`:

- Defines the order of test method execution within a test class.
- Helps in specifying a fixed execution order when needed.

`@BeforeEach`:

- Marks a method to run before each test method.
- Used to set up common test fixtures or perform initialization tasks.

`@AfterEach`:

- Marks a method to run after each test method.
- Used for cleaning up resources or performing teardown tasks after each test.

`@BeforeAll`:

- Indicates a method to be executed once before all test methods in the test class.
- Typically used for one-time setup tasks.

`@AfterAll`:

- Indicates a method to be executed once after all test methods in the test class have been executed.
- Used for one-time cleanup tasks.

Spring Testing Annotations:

- **`@SpringBootTest`**: Loads the complete Spring application context for integration testing.Useful for testing the entire application context.

- `**@WebMvcTest**`: Loads only the Spring MVC components for testing controllers.Useful for testing MVC components without loading the complete application context.

- `**@DataJpaTest**`: Loads only JPA-related components for testing JPA repositories.Optimized for testing JPA-specific functionality.

- `**@TestContainer**`: Launches containers (such as databases, Kafka, etc.) for integration testing.Provides isolation for tests that require external services.

- `**@ActiveProfiles**`: Specifies the active profiles to be used during testing. Enables testing with specific profiles (e.g., "test", "dev", etc.).

These annotations play vital roles in structuring, organizing, and executing tests in JUnit 5, providing flexibility, customization, and ease in writing different types of tests in various testing scenarios.

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.

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.

TestContainers

Below is a code snippet from our buddy application where we are testing the StudentsDal using TestContainers.

```
@DataJpaTest
@Testcontainers
@AutoConfigureTestDatabase(replace =
AutoConfigureTestDatabase.Replace.NONE)
public class StudentsDalTest {

	private static final MySQLContainer MY_SQL_CONTAINER = new
MySQLContainer("mysql:latest")
			.withDatabaseName("student-test-db")
			.withUsername("testUser")
			.withPassword("password");

	static {
		MY_SQL_CONTAINER.start();
	}

	@DynamicPropertySource
	static void registerDatabaseProperties(DynamicPropertyRegistry
registry) {
		registry.add("spring.datasource.url",
MY_SQL_CONTAINER::getJdbcUrl);
		registry.add("spring.datasource.username",
MY_SQL_CONTAINER::getUsername);
		registry.add("spring.datasource.password",
MY_SQL_CONTAINER::getPassword);
	}

	@Autowired
	StudentsDal studentsDal;

	@Test
	@Order(1)
	public void shouldSaveStudent() {
		Student student = new Student(1,
			"Rakesh",
			19,
			"JUnit",
			"Aryabhatta Hostels",
			"rakesh@gmail.com",
			"09874562134");
```

```
            Student resultStudent = studentsDal.save(student);
            assertThat(resultStudent)
                  .usingRecursiveComparison()
                  .ignoringFields("sid")
                  .isEqualTo(student);
      }
}
```

Explanation:

- This code is an example of a JUnit test class for testing a StudentsDal (Student Data Access Layer) repository in a Spring Boot application using JPA repositories and Testcontainers for an in-memory MySQL database.

- **@DataJpaTest**: This indicates that this is a test for JPA repositories. It configures a slice of the application context suitable for testing JPA-related components.

- **@Testcontainers**: Marks the usage of Testcontainers for integration testing.

- **@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE):** Specifies that the default test database configuration should not be replaced. It disables the automatic replacement of the configured database, allowing you to configure your database setup.

- **MySQLContainer**: Initializes a MySQL container from Testcontainers with specific configurations (database name, username, password).

- **Static Block**: Starts the MySQL container when the class is loaded.

- **DynamicPropertySource**: Registers dynamic properties for the Spring application context using Testcontainers. It sets the Spring datasource properties dynamically to use the MySQL container's URL, username, and password.

- **Autowired StudentsDal**: Injects the StudentsDal component to be tested.

- **@Test**: Denotes a test method.

- **shouldSaveStudent():** Method that tests the functionality of saving a Student.

- **Test Logic:**

    - Creates a Student object with specific details.
    - Invokes the save() method of studentsDal to save the student.
    - Uses AssertJ's assertThat to compare the saved resultStudent with the input student, ignoring the 'sid' field.

Summary:

- **Step 1**
  - configures test annotations for JPA and Test containers.
- **Step 2**
  - initializes and configures a MySQL container for testing.
- **Step 3**
  - tests the saving functionality of the StudentsDal repository by creating a Student, saving it, and asserting that the saved result matches the input, disregarding the 'sid' field.

@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.

References:

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