**TDD USING JUNIT AND MOCIKTO**

**UNIT TESTING WITH JUNIT**

**Unit testing with JUnit** ensures that individual components of an application function correctly. JUnit is a **popular testing framework for Java**, widely used for writing and running unit tests.

**Steps to Unit Test with JUnit**

1. **Set up JUnit in your project** (JUnit 5 recommended).
2. **Create a test class** with the @Test annotation.
3. **Write test methods** to verify expected behavior.
4. **Use assertions** (e.g., assertEquals, assertTrue) to validate outcomes.

**Example: Unit Testing a Simple Service**

**1️.Business Logic Class (CalculatorService.java)**

java

public class CalculatorService {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

**2️.Unit Test Class (CalculatorServiceTest.java)**

java

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

public class CalculatorServiceTest {

private final CalculatorService calculator = new CalculatorService();

@Test

public void testAddition() {

assertEquals(10, calculator.add(7, 3));

}

@Test

public void testMultiplication() {

assertEquals(15, calculator.multiply(5, 3));

}

}

**Explanation**

* @Test marks each method as a test case.
* assertEquals(expected, actual) checks expected vs. actual values.
* **No dependencies on Spring Boot** (pure unit test).

**TDD**

**Test-Driven Development (TDD) in Spring Boot** is a methodology where tests are written *before* the actual code is implemented. This approach ensures that every piece of functionality is validated from the start.

**How TDD Works in Spring Boot:**

1. **Write a test** – Define expected behavior using JUnit and Spring Boot test annotations.
2. **Run the test (fail expectedly)** – Since the feature doesn’t exist yet, the test fails.
3. **Implement the code** – Write the necessary functionality to make the test pass.
4. **Run the test (pass expectedly)** – Ensure the implementation meets requirements.
5. **Refactor and optimize** – Clean up the code while keeping tests successful.

**1)TESTING THE REPOSITORY LAYER**

Saying in unit testing -Arrange Act Assert

**1️.Entity Class (User.java)**

Defines the **User** entity with fields id, name, and email.

java

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

private String name;

private String email;

// Constructors, Getters, and Setters

public User() {}

public User(String name, String email) {

this.name = name;

this.email = email;

}

public Integer getId() { return id; }

public String getName() { return name; }

public String getEmail() { return email; }

}

**2️.Repository Interface (UserRepository.java)**

Extends JpaRepository to enable built-in **CRUD operations**.

java

@Repository

public interface UserRepository extends JpaRepository<User, Integer> {

Optional<User> findByEmail(String email);

}

**3️.Test Class (UserRepositoryTest.java)**

Uses **JUnit + Spring Boot + AssertJ** to test **Create (Save), Read, Update, Delete (CRUD)** operations.

java

@SpringBootTest

@RunWith(SpringRunner.class)

@DataJpaTest

public class UserRepositoryTest {

@Autowired

private UserRepository userRepository;

@Test

public void testCreateUser() {

// Save user using repository.save()

User user = new User("Alice", "alice@example.com");

User savedUser = userRepository.save(user);

// Validate creation

assertThat(savedUser).isNotNull();

assertThat(savedUser.getId()).isNotNull();

assertThat(savedUser.getName()).isEqualTo("Alice");

}

@Test

public void testFindUserById() {

// Save user

User user = new User("Bob", "bob@example.com");

User savedUser = userRepository.save(user);

// Fetch from database

Optional<User> foundUser = userRepository.findById(savedUser.getId());

// Validate retrieval

assertThat(foundUser).isPresent();

assertThat(foundUser.get().getName()).isEqualTo("Bob");

}

@Test

public void testFindUserByEmail() {

// Save user

User user = new User("Charlie", "charlie@example.com");

userRepository.save(user);

// Execute repository method

Optional<User> foundUser = userRepository.findByEmail("charlie@example.com");

// Validate results

assertThat(foundUser).isPresent();

assertThat(foundUser.get().getName()).isEqualTo("Charlie");

}

@Test

public void testUpdateUser() {

// Save user

User user = new User("David", "david@example.com");

User savedUser = userRepository.save(user);

// Update user details

savedUser.setName("David Updated");

User updatedUser = userRepository.save(savedUser);

// Validate update

assertThat(updatedUser.getName()).isEqualTo("David Updated");

}

@Test

public void testDeleteUser() {

// Save user

User user = new User("Eve", "eve@example.com");

User savedUser = userRepository.save(user);

// Delete user

userRepository.deleteById(savedUser.getId());

// Validate deletion

Optional<User> deletedUser = userRepository.findById(savedUser.getId());

assertThat(deletedUser).isEmpty();

}

}

**Explanation**

* **Uses** userRepository.save(user) instead of persist().
* **CRUD operations tested**:
  + ✅ **Create** (save())
  + ✅ **Read** (findById(), findByEmail())
  + ✅ **Update** (save() with new values)
  + ✅ **Delete** (deleteById())
* assertThat(foundUser).isPresent(); ensures successful retrieval.
* assertThat(deletedUser).isEmpty(); confirms entity deletion.

**2)TESTING THE SERVICE LAYER**

* Entity Class (User.java) – Represents the database structure.
* DTO Class (UserDTO.java) – Used for transferring data.
* Repository Interface (UserRepository.java) – Extends JpaRepository.
* Service Layer (UserService.java) – Implements business logic.
* Test Class (UserServiceTest.java) – Uses Mockito for unit testing all CRUD operations.

1️. Entity Class (User.java)

Defines the User entity with fields id, name, and email.

java

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer id;

private String name;

private String email;

// Constructors, Getters, and Setters

public User() {}

public User(Integer id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

public Integer getId() { return id; }

public String getName() { return name; }

public String getEmail() { return email; }

}

2️.DTO Class (UserDTO.java)

This class is used to transfer data between layers without exposing the entity directly.

java

public class UserDTO {

private String name;

private String email;

// Constructors

public UserDTO() {}

public UserDTO(String name, String email) {

this.name = name;

this.email = email;

}

// Getters & Setters

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

}

3️.Repository Interface (UserRepository.java)

Extends JpaRepository, enabling built-in CRUD operations.

java

@Repository

public interface UserRepository extends JpaRepository<User, Integer> {

Optional<User> findByEmail(String email);

}

4️.Service Layer (UserService.java)

Implements CRUD operations using the repository.

java

@Service

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public UserDTO getUserById(Integer id) {

return userRepository.findById(id)

.map(user -> new UserDTO(user.getName(), user.getEmail()))

.orElse(null);

}

public UserDTO createUser(UserDTO userDTO) {

User user = new User(null, userDTO.getName(), userDTO.getEmail());

User savedUser = userRepository.save(user);

return new UserDTO(savedUser.getName(), savedUser.getEmail());

}

public UserDTO updateUser(Integer id, UserDTO userDTO) {

Optional<User> optionalUser = userRepository.findById(id);

if (optionalUser.isPresent()) {

User user = optionalUser.get();

user.setName(userDTO.getName());

user.setEmail(userDTO.getEmail());

User updatedUser = userRepository.save(user);

return new UserDTO(updatedUser.getName(), updatedUser.getEmail());

}

return null;

}

public void deleteUser(Integer id) {

userRepository.deleteById(id);

}

}

5️.Test Class (UserServiceTest.java)

Uses JUnit + Mockito to test all CRUD operations.

java

@SpringBootTest

@RunWith(MockitoJUnitRunner.class)

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

public void testCreateUser() {

UserDTO userDTO = new UserDTO("Alice", "alice@example.com");

User savedUser = new User(1, "Alice", "alice@example.com");

when(userRepository.save(any(User.class))).thenReturn(savedUser);

UserDTO result = userService.createUser(userDTO);

assertThat(result).isNotNull();

assertThat(result.getName()).isEqualTo("Alice");

}

@Test

public void testGetUserById() {

User mockUser = new User(1, "Bob", "bob@example.com");

when(userRepository.findById(1)).thenReturn(Optional.of(mockUser));

UserDTO user = userService.getUserById(1);

assertThat(user).isNotNull();

assertThat(user.getName()).isEqualTo("Bob");

}

@Test

public void testUpdateUser() {

User mockUser = new User(1, "Charlie", "charlie@example.com");

when(userRepository.findById(1)).thenReturn(Optional.of(mockUser));

when(userRepository.save(any(User.class))).thenReturn(new User(1, "Charlie Updated", "updated@example.com"));

UserDTO updatedUserDTO = new UserDTO("Charlie Updated", "updated@example.com");

UserDTO result = userService.updateUser(1, updatedUserDTO);

assertThat(result).isNotNull();

assertThat(result.getName()).isEqualTo("Charlie Updated");

}

@Test

public void testDeleteUser() {

doNothing().when(userRepository).deleteById(1);

userService.deleteUser(1);

verify(userRepository, times(1)).deleteById(1);

}

}

Explanation

* Mockito Mocks (@Mock) the repository, avoiding database dependency.
* @InjectMocks injects the mock repository into the service.
* when(...).thenReturn(...) simulates repository behavior.
* CRUD operations tested:
  + ✅ Create (createUser())
  + ✅ Read (getUserById())
  + ✅ Update (updateUser())
  + ✅ Delete (deleteUser())

**3)Test The Controller Layer**

1️.Controller Class (UserController.java)

Handles HTTP requests related to User operations.

java

@RestController

@RequestMapping("/users")

public class UserController {

private final UserService userService;

public UserController(UserService userService) {

this.userService = userService;

}

@GetMapping("/{id}")

public ResponseEntity<UserDTO> getUserById(@PathVariable Integer id) {

UserDTO user = userService.getUserById(id);

return user != null ? ResponseEntity.ok(user) : ResponseEntity.notFound().build();

}

@PostMapping

public ResponseEntity<UserDTO> createUser(@RequestBody UserDTO userDTO) {

UserDTO savedUser = userService.createUser(userDTO);

return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);

}

}

2️.Test Class (UserControllerTest.java)

Uses MockMvc and Mockito to verify controller behavior.

java

@SpringBootTest

@RunWith(MockitoJUnitRunner.class)

@AutoConfigureMockMvc

public class UserControllerTest {

@Autowired

private MockMvc mockMvc;

@Mock

private UserService userService;

@InjectMocks

private UserController userController;

@BeforeEach

public void setup() {

mockMvc = MockMvcBuilders.standaloneSetup(userController).build();

}

@Test

public void testGetUserById() throws Exception {

UserDTO userDTO = new UserDTO("Alice", "alice@example.com");

when(userService.getUserById(1)).thenReturn(userDTO);

mockMvc.perform(get("/users/1"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Alice"))

.andExpect(jsonPath("$.email").value("alice@example.com"));

}

@Test

public void testCreateUser() throws Exception {

UserDTO userDTO = new UserDTO("Bob", "bob@example.com");

when(userService.createUser(any(UserDTO.class))).thenReturn(userDTO);

mockMvc.perform(post("/users")

.contentType(MediaType.APPLICATION\_JSON)

.content("{\"name\":\"Bob\",\"email\":\"bob@example.com\"}"))

.andExpect(status().isCreated())

.andExpect(jsonPath("$.name").value("Bob"))

.andExpect(jsonPath("$.email").value("bob@example.com"));

}

}

Explanation

* MockMvc – Simulates HTTP requests to test the controller.
* Mockito Mocks (@Mock) the service layer to avoid dependencies.
* @InjectMocks injects the mocked service into the controller.
* mockMvc.perform() sends HTTP requests and verifies responses.
* jsonPath("$.name") checks JSON response values.