**SPRING TESTING.**

**Exercise 1: Basic Unit Test for a Service Method Task:**

Write a unit test for a service method that adds two numbers.

Service:

@Service

public class CalculatorService {

public int add(int a, int b) {

return a + b;

}

}

**SOLUTION:**

**TEST:**

@SpringBootTest

public class CalculatorServiceTest {

@Autowired

private CalculatorService calculatorService;

@Test

public void testAdd() {

int result = calculatorService.add(2, 3);

assertEquals(5, result);

}

}

**Exercise 2: Mocking a Repository in a Service Test**

Task:

Test a service that uses a repository to fetch data.

Entity:

@Entity

public class User {

@Id

private Long id;

String name;

// getters and setters

}

Repository:

public interface UserRepository extends JpaRepository {

}

Service:

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public User getUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

}

**SOLUTION:**

**TEST:**

@ExtendWith(MockitoExtension.class)

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

public void testGetUserById() {

User user = new User();

user.setId(1L);

user.setName("John");

when(userRepository.findById(1L)).thenReturn(Optional.of(user));

User result = userService.getUserById(1L);

assertNotNull(result);

assertEquals("John", result.getName());

}

}

**Exercise 3: Testing a REST Controller with MockMvc**

Task: Test a controller endpoint that returns a user.

Controller:

@RestController

@RequestMapping("/users")

public class UserController {

@Autowired private UserService userService;

@GetMapping("/{id}")

public ResponseEntity getUser(@PathVariable Long id) {

return ResponseEntity.ok(userService.getUserById(id));

}

}

**SOLUTION:**

**TEST:**

@WebMvcTest(UserController.class)

public class UserControllerTest {

@Autowired

private MockMvc mockMvc;

@MockBean

private UserService userService;

@Test

public void testGetUser() throws Exception {

User user = new User();

user.setId(1L);

user.setName("Jane");

when(userService.getUserById(1L)).thenReturn(user);

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

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Jane"));

}

}

**Exercise 4: Integration Test with Spring Boot**

Task: Write an integration test that tests the full flow from controller to database.

**SOLUTION:**

**TEST:**

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

@AutoConfigureMockMvc

public class UserIntegrationTest {

@Autowired

private MockMvc mockMvc;

@Autowired

private UserRepository userRepository;

@BeforeEach

public void setup() {

userRepository.deleteAll();

User user = new User();

user.setId(1L);

user.setName("Test User");

userRepository.save(user);

}

@Test

public void testGetUserFromDb() throws Exception {

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

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("Test User"));

}

}

**Exercise 5: Test Controller POST Endpoint**

Task: Test a POST endpoint that creates a user.

Controller:

@PostMapping

public ResponseEntity createUser(@RequestBody User user) {

return ResponseEntity.ok(userService.saveUser(user)); }

**SOLUTION:**

**TEST:**

@Test

public void testCreateUser() throws Exception {

User user = new User();

user.setId(1L);

user.setName("New User");

when(userService.saveUser(any(User.class))).thenReturn(user);

mockMvc.perform(post("/users")

.contentType(MediaType.APPLICATION\_JSON)

.content("{\"id\":1,\"name\":\"New User\"}"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.name").value("New User"));

}

**Exercise 6: Test Service Exception Handling**

Task: Test how a service handles a missing user.

**SOLUTION:**

**TEST:**

@Test

public void testMissingUser() {

when(userRepository.findById(99L)).thenReturn(Optional.empty());

User result = userService.getUserById(99L);

assertNull(result);

}

**Exercise 7: Test Custom Repository Query**

Task: Add and test a custom query method.

**SOLUTION:**

**TEST:**

@DataJpaTest

public class UserRepositoryTest {

@Autowired

private UserRepository userRepository;

@Test

public void testFindByName() {

User user = new User();

user.setName("Alice");

userRepository.save(user);

List<User> users = userRepository.findByName("Alice");

assertFalse(users.isEmpty());

assertEquals("Alice", users.get(0).getName());

}

}

**Exercise 8: Test Controller Exception Handling**

Task: Add and test a @ControllerAdvice for handling exceptions.

**SOLUTION:**

**TEST:**

@Test

public void testUserNotFound() throws Exception {

when(userService.getUserById(42L)).thenThrow(new NoSuchElementException());

mockMvc.perform(get("/users/42"))

.andExpect(status().isNotFound())

.andExpect(content().string("User not found"));

}

**Exercise 9: Parameterized Test with JUnit**

Task: Use @ParameterizedTest to test multiple inputs.

**SOLUTION:**

**TEST:**

@ParameterizedTest

@CsvSource({

"1, 2, 3",

"2, 3, 5",

"5, 7, 12"

})

public void testAddParameterized(int a, int b, int expected) {

CalculatorService calculatorService = new CalculatorService();

assertEquals(expected, calculatorService.add(a, b));

}