Comprehensive REST API Testing Lab

Unit, Integration, and Automated Testing

with Spring Boot, MySQL, Mockito, and Postman

Student Lab Manual

Duration: 2-3 Hours

Target Audience: Students

Contents

Lab Overview

This comprehensive lab teaches professional testing practices for REST API development using Spring Boot. You will master three essential testing paradigms: unit testing to verify individual components in isolation, integration testing to validate database interactions, and automated API testing using Postman. Throughout this lab, you'll learn industry-standard practices including mocking external dependencies, managing test databases with Docker, and measuring code coverage with JaCoCo.

Learning Outcomes: By the end of this lab, you will be able to write production-grade tests, mock external services, verify database interactions, and automate API validation—skills essential for any professional software engineer.

Part I
Part 1: Conceptual Foundation

1 Testing Pyramid and Test Types

Professional software development follows the **testing pyramid**, a concept that defines the distribution and purpose of different test types.

1.1 The Testing Pyramid

Test Type	Characteristics	Purpose
Unit Tests (Base: 70%)	Fast, isolated, test single methods or functions. Mock all external dependencies.	Verify individual component logic and catch bugs early.
Integration Tests (Middle: 20%)	Slower, test multiple components together. Test with real or containerized databases.	Verify that components work together correctly and database interactions are valid.
End-to-End Tests (Top: 10%)	Slowest, test the entire application flow. Test from user perspective.	Verify complete user workflows work as expected in production-like environments.

1.2 Why Each Test Type Matters

1.2.1 Unit Tests

Unit tests verify the **logic of individual components** in isolation. They should:

- Execute in milliseconds (very fast feedback)
- Test a single **method or function** with multiple scenarios
- Mock all external dependencies (database, APIs, file systems)
- Be deterministic (always pass or always fail, never flaky)

Tip

Good unit tests are the first line of defense against bugs. They're fast enough to run hundreds of times during development, catching logic errors before they reach integration tests.

1.2.2 Integration Tests

Integration tests verify that **multiple components work together**. They should:

- Test real interactions (e.g., service + database)
- Use containerized or in-memory databases (not mocks)
- Execute in seconds (slower than unit tests, but still reasonably fast)
- Validate database queries, transaction handling, and data persistence

Tip

Integration tests catch issues that unit tests cannot find: incorrect SQL queries, missing database mappings, transactional bugs, and race conditions.

1.2.3 Automated API Tests (Postman)

API tests verify that **endpoints respond correctly**. They should:

- Test complete request-response cycles
- Validate HTTP status codes and response bodies
- Test edge cases and error scenarios
- Be automatable and repeatable

2 Mocking and Test Isolation

One of the most critical skills in testing is understanding **when and how to mock dependencies**.

2.1 What is Mocking?

A **mock** is a fake object that simulates the behavior of a real object. Instead of calling a real database or external API during a test, you use a mock that returns predetermined responses.

2.2 Why Mock?

Speed Real databases and APIs are slow. Mocks respond instantly.

Isolation By mocking external services, you test only your code's logic, not the external service's behavior.

Reliability External services might be unavailable or slow during testing. Mocks ensure consistent test behavior.

Cost Mocking prevents charges from APIs with usage costs or rate limits.

2.3 Mockito Framework

Mockito is a Java mocking library that makes it easy to create mocks and verify interactions.

Important

Key Mockito concepts:

- `@Mock`: Creates a mock object (fake dependency)
- `@InjectMocks`: Injects mocks into the class being tested
- `when(...).thenReturn(...)`: Sets up what a mock should return
- `verify()`: Asserts that a mock method was called

3 Database Testing Strategies

3.1 In-Memory vs. Docker vs. TestContainers

Approach	Setup	Realism	Speed
H2 In-Memory	Very fast, no setup.	Low; doesn't match production MySQL.	Very fast.
Docker Compose	Requires Docker installed.	High; real MySQL instance.	Fast; startup takes seconds.
TestContainers	Requires Docker; automatic provisioning.	High; spins up a real database.	Fast; container lifecycle automated.

For this lab, we'll use **Docker Compose** to spin up a real MySQL database specifically for testing.

3.2 Test Profile Configuration

Spring Boot supports multiple profiles (dev, test, prod). We'll create a `test` profile so tests use their own MySQL database, separate from development.

Part II
Part 2: Environment Setup

4 Step 1: Prerequisites and Installation

4.1 Required Software

Before starting, ensure you have installed:

- **Java Development Kit (JDK) 17+** - `https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html`
- **IntelliJ IDEA Community Edition** - `https://www.jetbrains.com/idea/download/`
- **Docker Desktop** - `https://www.docker.com/products/docker-desktop/`
- **Postman** - `https://www.postman.com/downloads/`
- **MySQL Workbench (optional)** - `https://dev.mysql.com/downloads/workbench/`

4.2 Verify Installation

```
# Check Java
java -version

# Check Docker
docker --version
docker-compose --version

# Verify Docker daemon is running
docker ps
```

Listing 1: Verify all tools are installed

5 Step 2: Project Setup

5.1 Option A: Download Starter Project

- Visit: `https://github.com/Braniacsl/CS4297-Lab`
- Click the green "Code" button and select "Download ZIP"
- Extract the ZIP file
- Open the project in IntelliJ IDEA

5.2 Option B: Create from Scratch

Use Spring Initializr (`https://start.spring.io/`) with these dependencies:

- Spring Web
- Spring Data JPA
- MySQL Driver
- JUnit Jupiter (for testing)
- Mockito (for mocking)

6 Step 3: Configure Docker Compose for Testing

Create a `docker-compose.yml` file in your project root:

```yaml
version: '3.8'
services:
  mysql-test:
    image: mysql:8.0
    container_name: mysql-test-db
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: testdb
      MYSQL_USER: testuser
      MYSQL_PASSWORD: testpass
    ports:
      - "3306:3306"
    volumes:
      - mysql-test-data:/var/lib/mysql
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      timeout: 5s
      retries: 10

volumes:
  mysql-test-data:
```

Listing 2: docker-compose.yml - MySQL Database Setup

6.1 Database Initialization Script

Create `init.sql` in your project root:

```sql
-- Create Users table
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(100) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- Create sample index for performance
CREATE INDEX idx_email ON users(email);
```

Listing 3: init.sql - Database Schema

6.2 Start the Test Database

```bash
# From your project root directory
docker-compose up -d

# Verify the container is running
docker ps

# Verify you can connect to MySQL
docker exec mysql-test-db mysql -u testuser -ptestpass testdb -e "SELECT 1;"
```

Listing 4: Start Docker Compose

Tip

The `-d` flag starts the container in detached mode (runs in background). Use `docker-compose logs -f` to view logs, and `docker-compose down` to stop when finished.

7 Step 4: Configure Spring Boot Application Properties

Create `src/main/resources/application-test.properties`:

```properties
# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=testuser
spring.datasource.password=testpass
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Hibernate Configuration
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true

# Logging
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE

# Server Port for Integration Tests
server.port=8081
```

Listing 5: application-test.properties - Test Profile Configuration

Part III
Part 3: Unit Testing

8 Step 1: Build Service Layer with Dependencies

8.1 Create User Entity

```java
package com.cs4297.lab.model;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import java.time.LocalDateTime;

@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

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

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @PrePersist
    protected void onCreate() {
        createdAt = LocalDateTime.now();
        updatedAt = LocalDateTime.now();
    }

    @PreUpdate
    protected void onUpdate() {
        updatedAt = LocalDateTime.now();
    }
}
```

Listing 6: User.java - JPA Entity

8.2 Create Repository Interface

```java
package com.cs4297.lab.repository;

import com.cs4297.lab.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
    Optional<User> findByUsername(String username);
    boolean existsByEmail(String email);
    boolean existsByUsername(String username);
}
```

Listing 7: UserRepository.java - Spring Data JPA Repository

8.3 Create Service Class

```java
package com.cs4297.lab.service;

import com.cs4297.lab.model.User;
import com.cs4297.lab.repository.UserRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;
import java.util.Optional;
import java.util.regex.Pattern;

@Service
@Transactional
public class UserService {

    private final UserRepository userRepository;
    private static final String EMAIL_REGEX =
        "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\\.[A-Z|a-z]{2,}$";
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile(EMAIL_REGEX);

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User createUser(String username, String email, String password) {
        // Validate inputs
        if (!isValidEmail(email)) {
            throw new IllegalArgumentException("Invalid email format");
        }

        if (userRepository.existsByEmail(email)) {
            throw new IllegalArgumentException("Email already exists");
        }

        if (userRepository.existsByUsername(username)) {
            throw new IllegalArgumentException("Username already exists");
        }

        if (password == null || password.length() < 8) {
            throw new IllegalArgumentException(
                "Password must be at least 8 characters");
        }

        User user = new User();
        user.setUsername(username);
        user.setEmail(email);
        user.setPassword(password); // In production, hash this!

        return userRepository.save(user);
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public Optional<User> updateUser(Long id, String email, String password) {
        return userRepository.findById(id).map(user -> {
            if (email != null && !email.equals(user.getEmail())) {
                if (!isValidEmail(email)) {
                    throw new IllegalArgumentException("Invalid email");
                }
                user.setEmail(email);
            }
            if (password != null && password.length() >= 8) {
                user.setPassword(password);
            }
            return userRepository.save(user);
        });
    }

    public boolean deleteUser(Long id) {
        if (userRepository.existsById(id)) {
            userRepository.deleteById(id);
            return true;
        }
        return false;
    }

    private boolean isValidEmail(String email) {
        return email != null && EMAIL_PATTERN.matcher(email).matches();
    }
}
```

Listing 8: UserService.java - Business Logic Layer

9 Step 2: Write Unit Tests for Service

```java
package com.cs4297.lab.service;

import com.cs4297.lab.model.User;
import com.cs4297.lab.repository.UserRepository;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    private User testUser;

    @BeforeEach
    void setUp() {
        testUser = new User();
        testUser.setId(1L);
        testUser.setUsername("testuser");
        testUser.setEmail("test@example.com");
        testUser.setPassword("password123");
    }

    @Test
    void testCreateUser_Success() {
        // Arrange
        when(userRepository.existsByEmail("test@example.com"))
            .thenReturn(false);
        when(userRepository.existsByUsername("testuser"))
            .thenReturn(false);
        when(userRepository.save(any(User.class)))
            .thenReturn(testUser);

        // Act
        User created = userService.createUser(
            "testuser", "test@example.com", "password123");

        // Assert
        assertNotNull(created);
        assertEquals("testuser", created.getUsername());
        assertEquals("test@example.com", created.getEmail());
        verify(userRepository, times(1)).save(any(User.class));
    }

    @Test
    void testCreateUser_InvalidEmail() {
        // Act & Assert
        assertThrows(IllegalArgumentException.class, () ->
            userService.createUser("testuser", "invalid-email",
                "password123")
        );

        // Verify save was never called
        verify(userRepository, never()).save(any());
    }

    @Test
    void testCreateUser_DuplicateEmail() {
        // Arrange
        when(userRepository.existsByEmail("test@example.com"))
            .thenReturn(true);

        // Act & Assert
        assertThrows(IllegalArgumentException.class, () ->
            userService.createUser("testuser", "test@example.com",
                "password123")
        );

        verify(userRepository, never()).save(any());
    }

    @Test
    void testCreateUser_WeakPassword() {
        // Act & Assert
        assertThrows(IllegalArgumentException.class, () ->
            userService.createUser("testuser", "test@example.com",
                "weak")
        );

        verify(userRepository, never()).save(any());
    }

    @Test
    void testGetUserById_Found() {
        // Arrange
        when(userRepository.findById(1L))
            .thenReturn(Optional.of(testUser));

        // Act
        Optional<User> found = userService.getUserById(1L);

        // Assert
        assertTrue(found.isPresent());
        assertEquals("testuser", found.get().getUsername());
    }

    @Test
    void testGetUserById_NotFound() {
        // Arrange
        when(userRepository.findById(99L))
            .thenReturn(Optional.empty());

        // Act
        Optional<User> found = userService.getUserById(99L);

        // Assert
        assertFalse(found.isPresent());
    }

    @Test
    void testDeleteUser_Success() {
        // Arrange
        when(userRepository.existsById(1L)).thenReturn(true);

        // Act
        boolean deleted = userService.deleteUser(1L);

        // Assert
        assertTrue(deleted);
        verify(userRepository, times(1)).deleteById(1L);
    }

    @Test
    void testDeleteUser_NotFound() {
        // Arrange
        when(userRepository.existsById(99L)).thenReturn(false);

        // Act
        boolean deleted = userService.deleteUser(99L);

        // Assert
        assertFalse(deleted);
        verify(userRepository, never()).deleteById(any());
    }
}
```

Listing 9: UserServiceTest.java - Unit Tests with Mockito

Important

Notice the pattern in these tests:

- **Arrange**: Set up mocks with `when(...).thenReturn(...)`
- **Act**: Call the method being tested
- **Assert**: Verify the results with assertions
- **Verify**: Check that mocks were called correctly

Part IV
Part 4: Controller Testing with MockMvc

10 Step 1: Create REST Controller

```java
package com.cs4297.lab.controller;

import com.cs4297.lab.model.User;
import com.cs4297.lab.service.UserService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        List<User> users = userService.getAllUsers();
        return ResponseEntity.ok(users);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        return userService.getUserById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public ResponseEntity<User> createUser(
            @RequestParam String username,
            @RequestParam String email,
            @RequestParam String password) {
        try {
            User user = userService.createUser(username, email, password);
            return ResponseEntity.status(HttpStatus.CREATED).body(user);
        } catch (IllegalArgumentException e) {
            return ResponseEntity.badRequest().build();
        }
    }

    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(
            @PathVariable Long id,
            @RequestParam(required = false) String email,
            @RequestParam(required = false) String password) {
        try {
            return userService.updateUser(id, email, password)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
        } catch (IllegalArgumentException e) {
            return ResponseEntity.badRequest().build();
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        if (userService.deleteUser(id)) {
            return ResponseEntity.noContent().build();
        }
        return ResponseEntity.notFound().build();
    }
}
```

Listing 10: UserController.java - REST API Endpoints

11 Step 2: Test Controller with MockMvc

```java
package com.cs4297.lab.controller;

import com.cs4297.lab.model.User;
import com.cs4297.lab.service.UserService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
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.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Arrays;
import java.util.Optional;

import static org.mockito.ArgumentMatchers.*;
import static org.mockito.Mockito.*;
import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
```

```java
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.hamcrest.Matchers.is;

@WebMvcTest(UserController.class)
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    private User user1;
    private User user2;

    @BeforeEach
    void setUp() {
        user1 = new User();
        user1.setId(1L);
        user1.setUsername("alice");
        user1.setEmail("alice@example.com");
        user1.setPassword("password123");

        user2 = new User();
        user2.setId(2L);
        user2.setUsername("bob");
        user2.setEmail("bob@example.com");
        user2.setPassword("password456");
    }

    @Test
    void testGetAllUsers_Success() throws Exception {
        // Arrange
        when(userService.getAllUsers())
            .thenReturn(Arrays.asList(user1, user2));

        // Act & Assert
        mockMvc.perform(get("/api/users")
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.length()", is(2)))
            .andExpect(jsonPath("$[0].username", is("alice")))
            .andExpect(jsonPath("$[1].username", is("bob")));
    }

    @Test
    void testGetUserById_Success() throws Exception {
        // Arrange
        when(userService.getUserById(1L))
            .thenReturn(Optional.of(user1));

        // Act & Assert
        mockMvc.perform(get("/api/users/1")
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.username", is("alice")))
            .andExpect(jsonPath("$.email", is("alice@example.com")));
    }

    @Test
    void testGetUserById_NotFound() throws Exception {
        // Arrange
        when(userService.getUserById(99L))
            .thenReturn(Optional.empty());

        // Act & Assert
        mockMvc.perform(get("/api/users/99")
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isNotFound());
    }

    @Test
    void testCreateUser_Success() throws Exception {
        // Arrange
        when(userService.createUser("alice", "alice@example.com",
            "password123"))
            .thenReturn(user1);

        // Act & Assert
        mockMvc.perform(post("/api/users")
                .param("username", "alice")
                .param("email", "alice@example.com")
                .param("password", "password123")
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.username", is("alice")));
    }

    @Test
    void testCreateUser_InvalidEmail() throws Exception {
        // Arrange
        when(userService.createUser("alice", "invalid", "password123"))
            .thenThrow(new IllegalArgumentException("Invalid email"));

        // Act & Assert
        mockMvc.perform(post("/api/users")
                .param("username", "alice")
                .param("email", "invalid")
                .param("password", "password123")
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isBadRequest());
```

```java
    }

    @Test
    void testDeleteUser_Success() throws Exception {
        // Arrange
        when(userService.deleteUser(1L)).thenReturn(true);

        // Act & Assert
        mockMvc.perform(delete("/api/users/1")
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isNoContent());
    }

    @Test
    void testDeleteUser_NotFound() throws Exception {
        // Arrange
        when(userService.deleteUser(99L)).thenReturn(false);

        // Act & Assert
        mockMvc.perform(delete("/api/users/99")
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isNotFound());
    }
}
```

Listing 11: UserControllerTest.java - Controller Layer Tests

Tip

`@WebMvcTest` creates a lightweight Spring context with only MVC components, making tests fast and focused. `MockMvc` simulates HTTP requests without starting a real server.

Part V
Part 5: Integration Testing with Real Database

12 Step 1: Repository Integration Tests

```java
package com.cs4297.lab.repository;

import com.cs4297.lab.model.User;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.TestPropertySource;

import java.util.Optional;

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

@DataJpaTest
@TestPropertySource(locations = "classpath:application-test.properties")
class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    private User testUser;

    @BeforeEach
    void setUp() {
        testUser = new User();
        testUser.setUsername("testuser");
        testUser.setEmail("test@example.com");
        testUser.setPassword("password123");
    }

    @Test
    void testSaveUser() {
        // Act
        User saved = userRepository.save(testUser);

        // Assert
        assertNotNull(saved.getId());
        assertEquals("testuser", saved.getUsername());
    }

    @Test
    void testFindByEmail() {
        // Arrange
        userRepository.save(testUser);

        // Act
        Optional<User> found = userRepository.findByEmail("test@example.com");

        // Assert
        assertTrue(found.isPresent());
        assertEquals("testuser", found.get().getUsername());
    }

    @Test
    void testFindByUsername() {
        // Arrange
        userRepository.save(testUser);

        // Act
        Optional<User> found = userRepository.findByUsername("testuser");

        // Assert
        assertTrue(found.isPresent());
        assertEquals("test@example.com", found.get().getEmail());
    }

    @Test
    void testExistsByEmail_True() {
        // Arrange
        userRepository.save(testUser);

        // Act
        boolean exists = userRepository.existsByEmail("test@example.com");

        // Assert
        assertTrue(exists);
    }

    @Test
    void testExistsByEmail_False() {
        // Act
        boolean exists =
            userRepository.existsByEmail("nonexistent@example.com");

        // Assert
        assertFalse(exists);
    }

    @Test
    void testUpdateUser() {
        // Arrange
        User saved = userRepository.save(testUser);
        Long userId = saved.getId();
```

```java

        // Act
        saved.setEmail("newemail@example.com");
        userRepository.save(saved);
        Optional<User> updated = userRepository.findById(userId);

        // Assert
        assertTrue(updated.isPresent());
        assertEquals("newemail@example.com", updated.get().getEmail());
    }

    @Test
    void testDeleteUser() {
        // Arrange
        User saved = userRepository.save(testUser);
        Long userId = saved.getId();

        // Act
        userRepository.deleteById(userId);
        Optional<User> deleted = userRepository.findById(userId);

        // Assert
        assertFalse(deleted.isPresent());
    }
}
```

Listing 12: UserRepositoryTest.java - Database Integration Tests

Tip

`@DataJpaTest` automatically configures a test database and enables transaction management. Each test runs in a transaction that's rolled back afterward, keeping tests isolated.

13 Step 2: Full Stack Integration Tests

```java
package com.cs4297.lab;

import com.cs4297.lab.model.User;
import com.cs4297.lab.repository.UserRepository;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
    org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.TestPropertySource;
import org.springframework.test.web.servlet.MockMvc;

import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.hamcrest.Matchers.is;

@SpringBootTest
@AutoConfigureMockMvc
@TestPropertySource(locations = "classpath:application-test.properties")
class UserControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private UserRepository userRepository;

    @BeforeEach
    void setUp() {
        // Clean up database before each test
        userRepository.deleteAll();
    }

    @Test
    void testCreateAndRetrieveUser() throws Exception {
        // Create user
        mockMvc.perform(post("/api/users")
                .param("username", "alice")
                .param("email", "alice@example.com")
                .param("password", "password123"))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.username", is("alice")));

        // Retrieve all users
        mockMvc.perform(get("/api/users"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.length()", is(1)))
            .andExpect(jsonPath("$[0].email", is("alice@example.com")));
    }

    @Test
    void testUpdateUserThroughAPI() throws Exception {
        // Create user
        mockMvc.perform(post("/api/users")
                .param("username", "alice")
                .param("email", "alice@example.com")
                .param("password", "password123"))
            .andExpect(status().isCreated());

        // Get the user ID from database
        User user = userRepository.findByUsername("alice").orElse(null);
        assert user != null;

        // Update user
        mockMvc.perform(put("/api/users/" + user.getId())
                .param("email", "newemail@example.com"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.email", is("newemail@example.com")));
    }

    @Test
    void testCreateDuplicateUser() throws Exception {
        // Create first user
        mockMvc.perform(post("/api/users")
                .param("username", "alice")
                .param("email", "alice@example.com")
                .param("password", "password123"))
            .andExpect(status().isCreated());

        // Try to create duplicate
        mockMvc.perform(post("/api/users")
                .param("username", "alice2")
                .param("email", "alice@example.com")
                .param("password", "password123"))
            .andExpect(status().isBadRequest());
    }
}
```

Listing 13: UserControllerIntegrationTest.java - Full Stack Tests

Part VI

Part 6: Automated API Testing with Postman

14 Step 1: Start Your Application

```
# From IntelliJ IDE:
# 1. Right-click on the main application class
# 2. Select "Run 'Application'" or press Ctrl+Shift+F10

# Or from command line:
./mvnw spring-boot:run
```

Listing 14: Start Spring Boot Application

15 Step 2: Create Postman Collection

1. Open Postman
2. Click "Create" → "Collection"
3. Name it "User API Tests"
4. Click "Create"

16 Step 3: Add Environment Variable

1. Click "Environments" in the left sidebar
2. Click "Create Environment"
3. Name it "Test"
4. Add variable:
    - **Key:** `baseUrl`
    - **Value:** `http://localhost:8080`
5. Click "Save"
6. Select "Test" environment in the dropdown

17 Step 4: Create Test Requests

17.1 GET All Users

1. Add request to collection: **GET** `{{baseUrl}}/api/users`
2. Go to ”Tests” tab
3. Add test script:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response is an array", function () {
    pm.expect(pm.response.json()).to.be.an('array');
});

pm.test("Response time is less than 1000ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(1000);
});
```

Listing 15: Postman Test - GET All Users

17.2 Create User (POST)

1. Add request: **POST** `{{baseUrl}}/api/users`
2. Body (form-data):
   - `username`: `alice`
   - `email`: `alice@example.com`
   - `password`: `password123`
3. Tests tab:

```
pm.test("Status code is 201", function () {
    pm.response.to.have.status(201);
});

pm.test("User ID is returned", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property("id");
    pm.expect(jsonData.id).to.be.above(0);
});

pm.test("Email is correct", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.email).to.equal("alice@example.com");
});

```

```
// Save user ID for next requests
pm.environment.set("userId", pm.response.json().id);
```

Listing 16: Postman Test - Create User

17.3 Get User by ID

1. Add request: **GET** `{{baseUrl}}/api/users/{{userId}}`
2. Tests tab:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Correct user is returned", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.username).to.equal("alice");
});
```

Listing 17: Postman Test - Get User by ID

17.4 Update User (PUT)

1. Add request: **PUT** `{{baseUrl}}/api/users/{{userId}}`
2. Body (form-data):
   - `email`: `newemail@example.com`
3. Tests tab:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Email was updated", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.email).to.equal("newemail@example.com");
});
```

Listing 18: Postman Test - Update User

17.5 Delete User

1. Add request: **DELETE** `{{baseUrl}}/api/users/{{userId}}`
2. Tests tab:

```
pm.test("Status code is 204", function () {
    pm.response.to.have.status(204);
});

// Clear the userId variable after successful delete
pm.environment.unset("userId");
```

Listing 19: Postman Test - Delete User

18 Step 5: Run Collection with Runner

1. Click "Runner" in top menu
2. Select your "User API Tests" collection
3. Select "Test" environment
4. Click "Run User API Tests"
5. Observe all tests passing (green checkmarks)

Tip

The runner executes all requests in order. Notice how the POST request saves the user ID, and subsequent requests use that ID. This demonstrates data flow between requests.

Part VII
Part 7: Code Coverage Analysis with JaCoCo

19 Step 1: Add JaCoCo Plugin to POM

Add to `pom.xml`:

```xml
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.8</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Listing 20: pom.xml - JaCoCo Configuration

20 Step 2: Run Tests with Coverage

```bash
# Run Maven tests and generate JaCoCo report
./mvnw clean test

# Open the HTML report
# Location: target/site/jacoco/index.html
open target/site/jacoco/index.html
```

Listing 21: Generate Coverage Report

21 Step 3: Interpret Coverage Report

The JaCoCo report shows four metrics:

Line Coverage Percentage of executable code lines executed by tests

Branch Coverage Percentage of code branches (if/else) tested

Cyclomatic Complexity Number of different paths through code

Method Coverage Percentage of methods tested

Important

Industry standard target: **80% code coverage**. Aim for high coverage on business logic; lower coverage on generated code is acceptable.

Part VIII
Part 8: Advanced Testing Scenarios

22 Mocking External API Calls

When your service calls an external API, mock it in tests to avoid:

- Network latency
- External service unavailability
- API rate limits
- Unexpected responses

22.1 Example: Payment API Integration

```java
package com.cs4297.lab.service;

public interface PaymentClient {
    PaymentResponse processPayment(String userId, double amount);
}

public class PaymentService {
    private final PaymentClient paymentClient;

    public PaymentService(PaymentClient paymentClient) {
        this.paymentClient = paymentClient;
    }

    public String chargeUser(String userId, double amount) {
        try {
            PaymentResponse response = paymentClient.processPayment(
                userId, amount);
            if (response.isSuccessful()) {
                return "Payment of $" + amount + " successful. " +
                    "Transaction ID: " + response.getTransactionId();
            }
        } catch (PaymentException e) {
            return "Payment failed: " + e.getMessage();
        }
        return "Payment failed: unknown error";
    }
}
```

Listing 22: PaymentService with External API Call

22.2 Unit Tests with Mocked API

```java
@ExtendWith(MockitoExtension.class)
class PaymentServiceTest {

    @Mock
    private PaymentClient paymentClient;

    @InjectMocks
    private PaymentService paymentService;

    @Test
    void testChargeUser_Success() {
        // Arrange
        PaymentResponse mockResponse = new PaymentResponse(true, "TXN123");
        when(paymentClient.processPayment("user1", 100.0))
            .thenReturn(mockResponse);

        // Act
        String result = paymentService.chargeUser("user1", 100.0);

        // Assert
        assertTrue(result.contains("successful"));
        assertTrue(result.contains("TXN123"));
    }

    @Test
    void testChargeUser_PaymentFailure() {
        // Arrange
        PaymentResponse mockResponse = new PaymentResponse(false, null);
        when(paymentClient.processPayment("user1", 100.0))
            .thenReturn(mockResponse);

        // Act
        String result = paymentService.chargeUser("user1", 100.0);

        // Assert
        assertTrue(result.contains("failed"));
    }

    @Test
    void testChargeUser_ExceptionHandling() {
        // Arrange
        when(paymentClient.processPayment("user1", 100.0))
            .thenThrow(new PaymentException("Network timeout"));

        // Act
        String result = paymentService.chargeUser("user1", 100.0);

        // Assert
        assertTrue(result.contains("Payment failed"));
    }
}
```

Listing 23: PaymentServiceTest - Testing with Mock API

Part IX
Part 9: Lab Demonstration and Evaluation

23 What Students Must Demonstrate

Component	Evidence	Evaluation Focus
Unit Tests	All unit tests pass. Show test output. Explain what each test verifies.	Understanding of test isolation and mocking.
Integration Tests	Integration tests run against real database. Show `@DataJpaTest` and `@SpringBootTest` tests passing.	Database interaction and transaction handling.
Controller Tests	MockMvc tests pass. Show JSON response validation.	HTTP request/response testing.
Postman Collection	Run entire collection in Postman runner. All tests green. Show variable persistence across requests.	Automated API testing workflow.
Code Coverage	Show JaCoCo report. Explain line/branch coverage. Target 80%+ on service layer.	Understanding testing metrics and coverage goals.

24 Sample Interview Questions

1. **Q: Why should you mock external dependencies in unit tests?**

   A: Mocks isolate the code being tested, make tests faster, and prevent failures due to external service unavailability. They allow testing edge cases without relying on real services.

2. **Q: What's the difference between `@WebMvcTest` and `@SpringBootTest`?**

   A: `@WebMvcTest` creates a lightweight context with only MVC components. `@SpringBootTest` loads the full application context including database. WebMvcTest is faster; SpringBootTest is more realistic.

3. **Q: When should you use integration tests vs. unit tests?**

   A: Unit tests verify individual component logic (faster, more isolated). Integration tests verify components work together (slower, more realistic). Use both: mostly unit tests, some integration tests.

4. **Q: How does Mockito's `@InjectMocks` work?**

A: It creates an instance of the class under test and injects all `@Mock` fields into it. This eliminates boilerplate constructor code.

5. **Q: What does 80% code coverage mean?**

   A: 80% of the executable code lines are executed by tests. However, coverage doesn't guarantee correctness—tests must verify correct behavior, not just execute code.

A Troubleshooting Guide

A.1 Docker Issues

A.1.1 Error: "Cannot connect to Docker daemon"

Warning

Solution:

1. Ensure Docker Desktop is running (check system tray)
2. On Mac/Linux: `docker --version` should work
3. Restart Docker Desktop

A.1.2 Error: "Port 3306 already in use"

Warning

Solution:

1. Check what's using the port: `lsof -i :3306`
2. Stop the container: `docker-compose down`
3. Or use different port in docker-compose.yml: `- "3307:3306"`

A.2 Test Failures

A.2.1 Error: "NullPointerException on @Mock field"

Warning

Solution:

1. Verify `@ExtendWith(MockitoExtension.class)` is present on test class
2. Ensure `@InjectMocks` is on the class under test, not `@Autowired`

A.2.2 Error: "Cannot find test database"

Warning

Solution:

1. Verify `application-test.properties` exists in `src/test/resources/`
2. Verify docker-compose is running: `docker ps`
3. Check database connection URL matches your setup

A.3 Postman Issues

A.3.1 Problem: Environment variable not persisting between requests

Warning

Solution:

1. Use `pm.environment.set("key", value);` in test script
2. Verify you've selected the correct environment
3. Check the environment dropdown shows your environment name

B Best Practices Summary

- **Test pyramid:** 70% unit, 20% integration, 10% end-to-end
- **Isolation:** Mock external dependencies in unit tests
- **Realistic:** Use real databases in integration tests
- **Coverage:** Target 80% code coverage on business logic
- **Naming:** Use clear test names like `testCreateUser_Success`
- **Organization:** Group related tests with `@Nested` classes
- **Cleanup:** Always clean up test data in `@BeforeEach` or `@AfterEach`

C Quick Reference: Testing Annotations

Annotation	Purpose
`@Test`	Marks method as a test case
`@BeforeEach`	Runs before each test (setup)
`@AfterEach`	Runs after each test (cleanup)
`@Mock`	Creates a mock object
`@InjectMocks`	Injects mocks into class under test
`@ExtendWith(MockitoExtension.class)`	Enables Mockito in test class
`@WebMvcTest`	Tests MVC layer only (lightweight)
`@SpringBootTest`	Tests entire application (full context)
`@DataJpaTest`	Tests JPA repositories with database
`@TestPropertySource`	Provides test-specific properties