TRƯỜNG ĐẠI HỌC SÀI GÒN
KHOA CÔNG NGHỆ THÔNG TIN

# KIỂM THỬ PHẦN MỀM

# BÁO CÁO BÀI TẬP LỚN

## Ứng dụng Đăng nhập & Quản lý Sản phẩm

**(FloginFE_BE)**

GVHD: Từ Lãng Phiêu
SV: Đào Nhị Khang - 3122410168
Mai Văn Minh Trí - 3123410389
Nguyễn Minh Trí - 3122560082
Trần Thị Tình - 3122410414
Đào Song Lộc - 3123410200
Cao Minh Triết - 3123410391

TP. HỒ CHÍ MINH, NGÀY 2 THÁNG 12 NĂM 2025

# Mục lục

# LỜI MỞ ĐẦU

Kiểm thử phần mềm là một phần không thể thiếu trong quy trình phát triển phần mềm chuyên nghiệp. Trong bối cảnh công nghệ phát triển nhanh chóng, việc đảm bảo chất lượng sản phẩm phần mềm trở nên quan trọng hơn bao giờ hết. Bài tập lớn này nhằm giúp sinh viên nắm vững các kỹ thuật kiểm thử hiện đại và áp dụng vào thực tế.

## Giới thiệu đề tài

Đề tài được lựa chọn là ứng dụng **FloginFE_BE** - một hệ thống web full-stack hoàn chỉnh với các thành phần chính:

- **Frontend**: ReactJS 19 với React Router, Axios
- **Backend**: Spring Boot 3.3.5 với Spring Security, JWT Authentication
- **Database**: MySQL 8.0
- **Chức năng chính**:
  - Authentication: Login, Register với JWT token
  - Product Management: CRUD operations với phân quyền
  - Image upload và validation

Qua đề tài này, nhóm có cơ hội thực hành đầy đủ các loại kiểm thử từ Unit Testing, Integration Testing, Mock Testing đến Automation Testing, Performance Testing và Security Testing.

Báo cáo này trình bày chi tiết quá trình thực hiện các yêu cầu của bài tập lớn, bao gồm:

- Phân tích và thiết kế Test Cases
- Unit Testing với phương pháp Test-Driven Development (TDD)
- Integration Testing
- Mock Testing
- Automation Testing và CI/CD
- Performance Testing và Security Testing (phần mở rộng)

Nhóm xin chân thành cảm ơn thầy Từ Lãng Phiêu đã hướng dẫn tận tình trong suốt quá trình thực hiện bài tập lớn này.

*TP. Hồ Chí Minh, ngày ... tháng 11 năm 2025*
*Nhóm sinh viên thực hiện*

# 1 Phân tích và Thiết kế Test Cases

## 1.1 Giới thiệu chương

Chương này trình bày quá trình phân tích yêu cầu và thiết kế test cases chi tiết cho hai chức năng chính của hệ thống: **Login** (Đăng nhập) và **Product Management** (Quản lý sản phẩm).

*Lưu ý: Nội dung chi tiết của chương này sẽ được bổ sung sau.*

## 1.2 Login - Phân tích và Test Scenarios

### 1.2.1 Yêu cầu chức năng

[Nội dung sẽ được bổ sung]

### 1.2.2 Test Scenarios

[Nội dung sẽ được bổ sung]

### 1.2.3 Thiết kế Test Cases chi tiết

[Nội dung sẽ được bổ sung]

## 1.3 Product - Phân tích và Test Scenarios

### 1.3.1 Yêu cầu chức năng

[Nội dung sẽ được bổ sung]

### 1.3.2 Test Scenarios

[Nội dung sẽ được bổ sung]

### 1.3.3 Thiết kế Test Cases chi tiết

[Nội dung sẽ được bổ sung]

# 2 Unit Testing và Test-Driven Development (TDD)

## 2.1 Giới thiệu chương

Chương này trình bày quá trình thực hiện Unit Testing cho hệ thống FloginFE_BE theo phương pháp **Test-Driven Development (TDD)**. Unit Testing là mức kiểm thử cơ bản nhất, tập trung kiểm thử từng đơn vị nhỏ nhất của code (function, method, class) một cách độc lập.

**Nội dung chính của chương:**

- Công cụ kiểm thử: Jest (Frontend), JUnit 5 (Backend), Mockito, JaCoCo
- Unit Tests cho chức năng Login: Validation và Authentication
- Unit Tests cho chức năng Product: Validation và CRUD operations
- Code Coverage analysis: Đo lường độ bao phủ mã nguồn
- Kết luận và đánh giá kết quả

## 2.2 Công cụ kiểm thử

**Frontend:** Jest, React Testing Library, Jest DOM

**Backend:** JUnit 5, Mockito, JaCoCo (Code Coverage)

**Phương pháp:** Test-Driven Development (TDD) - Red, Green, Refactor

*Lưu ý: Hướng dẫn setup chi tiết có trong file README.md tại thư mục gốc project.*

## 2.3 Unit Tests cho Chức năng Đăng nhập (Login)

### 2.3.1 Frontend Unit Tests (Validation Logic)

Chúng em tập trung kiểm thử các hàm validation trong `utils/validation.js` để đảm bảo dữ liệu đầu vào hợp lệ trước khi gửi xuống Server.

**Các trường hợp kiểm thử (Test Cases):**

| ID Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| **Test cho Username** | | | |
| TC_LOGIN_001 | Username rỗng hoặc chỉ chứa khoảng trắng | Trả về lỗi: *"Tên đăng nhập không được để trống"* | Passed |
| TC_LOGIN_002 | Username quá ngắn (< 3 ký tự) | Trả về lỗi: *"Tên đăng nhập phải có ít nhất 3 ký tự"* | Passed |
| TC_LOGIN_003 | Username quá dài (> 50 ký tự) | Trả về lỗi: *"Tên đăng nhập không được quá 50 ký tự"* | Passed |
| TC_LOGIN_004 | Username chứa ký tự đặc biệt hoặc khoảng trắng | Trả về lỗi: *"Tên đăng nhập chỉ chứa chữ cái và số"* | Passed |
| TC_LOGIN_005 | Username hợp lệ (ví dụ: `testuser1`, `ADMIN`) | Không trả về lỗi (chuỗi rỗng) | Passed |
| **Test cho Password** | | | |
| TC_LOGIN_006 | Password rỗng hoặc chỉ chứa khoảng trắng | Trả về lỗi: *"Mật khẩu không được để trống"* | Passed |
| TC_LOGIN_007 | Password quá ngắn (< 6 ký tự) | Trả về lỗi: *"Mật khẩu phải có ít nhất 6 ký tự"* | Passed |
| TC_LOGIN_008 | Password quá dài (> 100 ký tự) | Trả về lỗi: *"Mật khẩu không được quá 100 ký tự"* | Passed |
| TC_LOGIN_009 | Password thiếu chữ cái (chỉ có số, ví dụ: `12345678`) | Trả về lỗi: *"Mật khẩu phải chứa cả chữ cái và số"* | Passed |

Bảng 1 – tiếp theo trang trước

| ID Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_LOGIN_010 | Password thiếu số (chỉ có chữ, ví dụ: `abcdefgh`) | Trả về lỗi: *"Mật khẩu phải chứa cả chữ cái và số"* | Passed |
| TC_LOGIN_011 | Password hợp lệ (có cả chữ và số, ví dụ: `Test1234`) | Không trả về lỗi (chuỗi rỗng) | Passed |

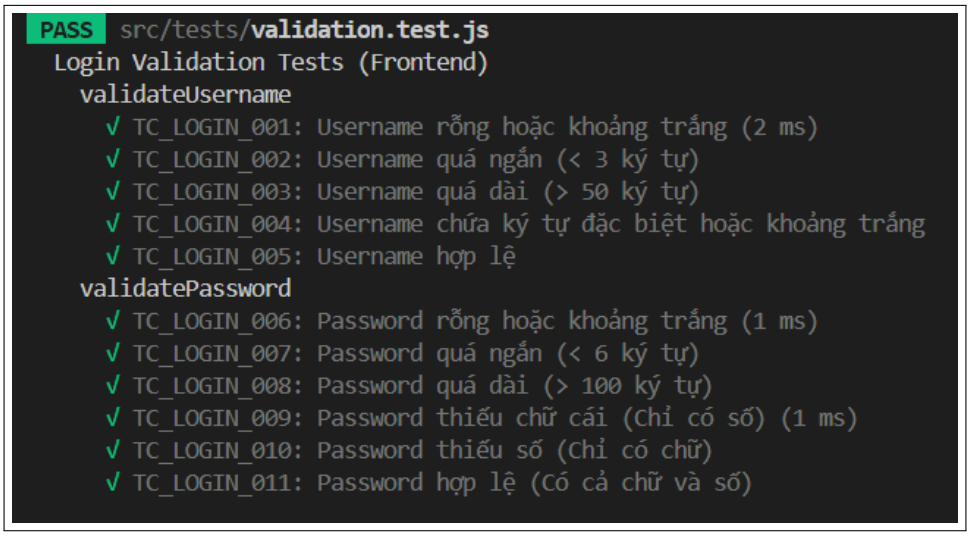
### Code minh chứng (Code Snippet):

```
// File: frontend/src/tests/validation.test.js
import { validateUsername, validatePassword } from '../utils/validation';

describe('Login Validation Tests', () => {
    test('TC_LOGIN_001: Username rong hoac khoang trang', () => {
        expect(validateUsername('')).toBe('Ten dang nhap khong duoc de trong');
        expect(validateUsername(' ')).toBe('Ten dang nhap khong duoc de trong');
    });

    test('TC_LOGIN_004: Username chua ky tu dac biet', () => {
        expect(validateUsername('user@123')).toBe('Ten dang nhap chi chua chu cai va so');
    });

    test('TC_LOGIN_009: Password thieu chu cai (Chi co so)', () => {
        expect(validatePassword('12345678')).toBe('Mat khau phai chua ca chu cai va so');
    });
});
```

### Bằng chứng thực hiện (Evidence):

*Để chạy test, sử dụng lệnh:*

```
npm test src/tests/validation.test.js
```




Hình 1: Kết quả Unit Test - Login Validation Frontend

### 2.3.2 Backend Unit Tests (Auth Service)

Tại Backend, chúng em sử dụng **Mockito** để cô lập `AuthService`, giả lập hành vi của `AuthenticationManager`, `JwtTokenProvider`, `AppUserRepository` và `PasswordEncoder`.

**Các trường hợp kiểm thử chính:**

| Test Case | Mô tả | Trạng thái |
|---|---|---|
| `testLoginSuccess` | Khi thông tin đăng nhập đúng, hệ thống trả về JWT Token và thông tin user. Kiểm tra `authenticationManager.authenticate()` và `jwtTokenProvider.generateToken()` được gọi đúng 1 lần. | Passed |
| `testLoginFailure` | Khi sai username hoặc password, hệ thống ném ra ngoại lệ `AuthenticationException`. Đảm bảo `generateToken()` không được gọi. | Passed |
| `testRegisterSuccess` | Khi đăng ký mới hợp lệ (username chưa tồn tại), thông tin user được lưu vào Database thông qua `appUserRepository.save()`. Kiểm tra mật khẩu được mã hóa. | Passed |
| `testRegisterFailureDuplicate` | Khi đăng ký trùng username (username đã tồn tại), hệ thống ném lỗi `RuntimeException` với thông báo *"Lỗi: Username đã được sử dụng!"*. Đảm bảo `repository.save()` không được gọi. | Passed |

**Code minh chứng (Code Snippet):**

```
// File: backend/src/test/java/.../service/AuthServiceTest.java
@DisplayName("Login Service Unit Tests")
class AuthServiceTest {
    @Mock private AuthenticationManager authenticationManager;
    @Mock private JwtTokenProvider jwtTokenProvider;
    @InjectMocks private AuthService authService;

    @Test
    @DisplayName("TC1: Login thanh cong voi credentials hop le")
    void testLoginSuccess() {
        LoginRequest request = new LoginRequest();
        request.setUsername("testuser");
        request.setPassword("Test123");

        when(authenticationManager.authenticate(any())).thenReturn(mock(Authentication.class));
        when(jwtTokenProvider.generateToken(any())).thenReturn("mock-jwt-token");

        String token = authService.loginUser(request).getToken();

        assertNotNull(token);
        verify(authenticationManager, times(1)).authenticate(any());
    }
}
```

**Bằng chứng thực hiện (Evidence):**

*Để chạy test backend, sử dụng lệnh:*

```
mvn test -Dtest=AuthServiceTest
```


Hình 2: Kết quả Unit Test - AuthService Backend

## 2.4 Unit Tests cho Chức năng Quản lý Sản phẩm (Product)

### 2.4.1 Frontend Unit Tests (Validation & Component)

Phần này kiểm thử cả logic validation sản phẩm và giao diện Form nhập liệu.

**Logic Validation (productValidation.js)**

Chúng em tập trung kiểm thử các hàm validation trong `utils/productValidation.js` để đảm bảo dữ liệu nhập vào form sản phẩm hợp lệ trước khi gửi xuống Server. Kiểm tra các quy tắc nghiệp vụ:

- Giá sản phẩm (Số âm, số 0, số quá lớn)
- Số lượng (Số nguyên, số âm, số 0, số quá lớn)
- Tên sản phẩm (Độ dài, ký tự đặc biệt)
- Danh mục (Bắt buộc chọn)
- Mô tả (Độ dài tối đa)

| ID Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| **Test cho Tên sản phẩm (Name)** | | | |
| TC_PROD_001 | Tên sản phẩm rỗng hoặc khoảng trắng | Lỗi: *"Tên sản phẩm không được để trống"* | Passed |
| TC_PROD_002 | Tên quá ngắn (< 3 ký tự) | Lỗi: *"Tên sản phẩm phải có ít nhất 3 ký tự"* | Passed |
| TC_PROD_003 | Tên quá dài (> 100 ký tự) | Lỗi: *"Tên sản phẩm không được quá 100 ký tự"* | Passed |
| **Test cho Giá sản phẩm (Price)** | | | |
| TC_PROD_004 | Giá không phải là số (ví dụ: `'abc'`, `null`) | Lỗi: *"Giá sản phẩm không hợp lệ"* | Passed |
| TC_PROD_005 | Giá âm hoặc bằng 0 | Lỗi: *"Giá sản phẩm phải lớn hơn 0"* | Passed |
| TC_PROD_006 | Giá quá lớn (> 999,999,999) | Lỗi: *"Giá sản phẩm quá lớn (tối đa 999,999,999)"* | Passed |
| **Test cho Số lượng (Quantity)** | | | |
| TC_PROD_007 | Số lượng không phải là số | Lỗi: *"Số lượng không hợp lệ"* | Passed |
| TC_PROD_008 | Số lượng là số thập phân (Float, ví dụ: 10.5) | Lỗi: *"Số lượng phải là số nguyên"* | Passed |
| TC_PROD_009 | Số lượng bằng 0 | Lỗi: *"Số lượng phải lớn hơn 0"* | Passed |
| TC_PROD_010 | Số lượng âm | Lỗi: *"Số lượng không được nhỏ hơn 0"* | Passed |
| TC_PROD_011 | Số lượng quá lớn (> 99,999) | Lỗi: *"Số lượng quá lớn (tối đa 99,999)"* | Passed |
| **Test cho Mô tả và Danh mục** | | | |
| TC_PROD_012 | Mô tả quá dài (> 500 ký tự) | Lỗi: *"Mô tả không được quá 500 ký tự"* | Passed |

Bảng 3 – tiếp theo trang trước

| ID Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_PROD_013 | Danh mục chưa chọn hoặc không hợp lệ (’’, 0, null) | Lỗi: *"Vui lòng chọn danh mục"* | Passed |
| **Test tích hợp** | | | |
| TC_PROD_014 | Sản phẩm hợp lệ hoàn toàn (tất cả trường đều đúng) | Không có lỗi (Object rỗng) | Passed |

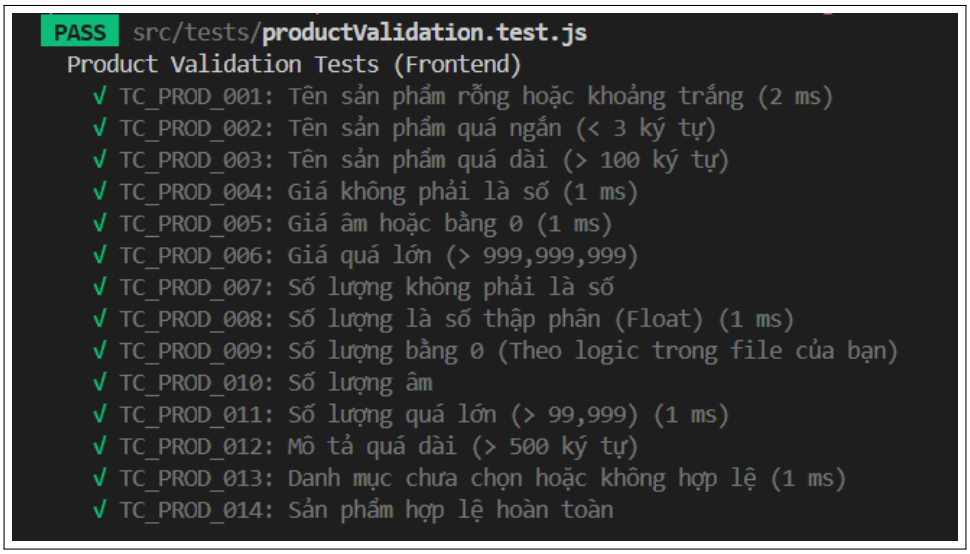
### Code minh chứng (Code Snippet):

```
// File: frontend/src/tests/productValidation.test.js
import { validateProduct } from '../utils/productValidation';

describe('Product Validation Tests', () => {
    const validProduct = {
        name: 'Laptop Dell', price: 15000000, quantity: 10,
        description: 'Mo ta ngan gon', categoryId: 1
    };

    test('TC_PROD_005: Gia am hoac bang 0', () => {
        expect(validateProduct({ ...validProduct, price: -5000 }).price)
            .toBe('Gia san pham phai lon hon 0');
        expect(validateProduct({ ...validProduct, price: 0 }).price)
            .toBe('Gia san pham phai lon hon 0');
    });

    test('TC_PROD_008: So luong la so thap phan (Float)', () => {
        expect(validateProduct({ ...validProduct, quantity: 10.5 }).quantity)
            .toBe('So luong phai la so nguyen');
    });

    test('TC_PROD_014: San pham hop le hoan toan', () => {
        const errors = validateProduct(validProduct);
        expect(Object.keys(errors).length).toBe(0);
    });
});
```

### Bằng chứng thực hiện (Evidence):

*Để chạy test, sử dụng lệnh:*

```
npm test src/tests/productValidation.test.js
```




Hình 3: Kết quả Unit Test - Product Validation Frontend

### 2.4.2 Backend Unit Tests (Product Service)

Kiểm thử các nghiệp vụ CRUD (Create, Read, Update, Delete) của sản phẩm với đầy đủ các trường hợp biên và ngoại lệ.

**Các trường hợp kiểm thử chính:**

| Test Case | Mô tả | Trạng thái |
|---|---|---|
| `testCreateProduct` | Thêm mới sản phẩm thành công, gọi `repository.save()` đúng 1 lần. Kiểm tra tên sản phẩm không trùng. | Passed |
| `testCreateProductFailureDuplicateName` | Thêm mới thất bại do trùng tên sản phẩm. Ném `RuntimeException`, đảm bảo `repository.save()` không được gọi. | Passed |
| `testUpdateProduct` | Cập nhật sản phẩm thành công khi ID tồn tại và tên không trùng với sản phẩm khác. | Passed |
| `testUpdateProductNotFound` | Cập nhật thất bại khi ID không tồn tại → Ném lỗi `EntityNotFoundException`. | Passed |
| `testUpdateProductDuplicateName` | Cập nhật thất bại khi tên mới trùng với sản phẩm khác → Ném `RuntimeException`. | Passed |
| `testUpdateProductWithImage` | Cập nhật thành công kèm theo cập nhật hình ảnh mới. Kiểm tra logic set ảnh được gọi. | Passed |
| `testDeleteProduct` | Xóa sản phẩm thành công khi ID tồn tại. | Passed |
| `testDeleteProductNotFound` | Xóa thất bại khi ID không tồn tại → Ném `EntityNotFoundException`. | Passed |
| `testGetAllProducts` | Lấy danh sách tất cả sản phẩm. Kiểm tra số lượng và nội dung trả về. | Passed |
| `testGetProductById` | Lấy sản phẩm theo ID thành công. Kiểm tra thông tin chi tiết. | Passed |
| `testGetProductByIdNotFound` | Lấy sản phẩm theo ID không tồn tại → Ném `EntityNotFoundException`. | Passed |

**Code minh chứng (Code Snippet):**

```java
// File: backend/src/test/java/.../service/ProductServiceTest.java
@DisplayName("Product Service Unit Tests")
class ProductServiceTest {
    @Mock private ProductRepository productRepository;
    @InjectMocks private ProductService productService;

    @Test
    @DisplayName("TC1: Tao san pham moi thanh cong")
    void testCreateProduct() {
        when(productRepository.existsByName(anyString())).thenReturn(false);
        when(productRepository.save(any(Product.class))).thenReturn(product);

        ProductDto result = productService.createProduct(productDto);

        assertNotNull(result);
        verify(productRepository, times(1)).save(any(Product.class));
    }

    @Test
    @DisplayName("TC9: Update that bai do ID khong ton tai")
    void testUpdateProductNotFound() {
        when(productRepository.findById(99)).thenReturn(Optional.empty());

        assertThrows(EntityNotFoundException.class, () -> {
            productService.updateProduct(99, productDto);
        });
        verify(productRepository, never()).save(any());
    }
}
```

**Bằng chứng thực hiện (Evidence):**
*Để chạy test backend, sử dụng lệnh:*

```
mvn test -Dtest=ProductServiceTest
```


Hình 4: Kết quả Unit Test - ProductService Backend

## 2.5 Kết quả Độ phủ mã nguồn (Code Coverage)

Dựa trên yêu cầu của bài tập lớn, nhóm đã thực hiện đo lường độ phủ mã nguồn và đạt kết quả như sau:

### 2.5.1 Frontend Coverage (Jest)

**Yêu cầu:** >= 90%

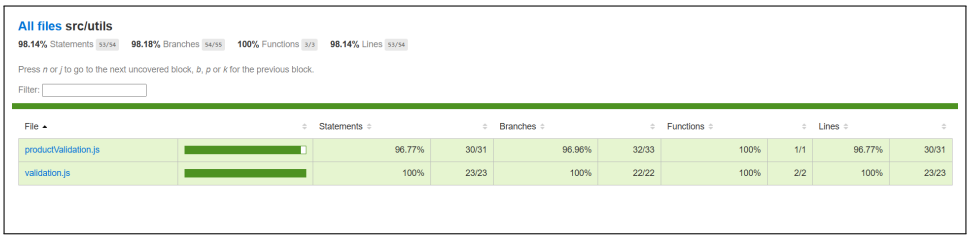
**Kết quả đạt được:**

- **Validation Module** (`validation.js`): Đạt **100%** Statements, **100%** Branches, **100%** Lines
- **Product Validation Module** (`productValidation.js`): Đạt **96.77%** Statements, **96.96%** Branches, **96.77%** Lines
- **Tổng thể (Overall)**: Đạt **98.14%** Statements, **98.18%** Branches, **100%** Functions, **98.14%** Lines

**Cách chạy báo cáo Coverage:**

```
npm run coverage:fe
# Hoac
npm test -- --coverage --watchAll=false
```

Kết quả được tạo trong thư mục `frontend/coverage/lcov-report/index.html`




Hình 5: Báo cáo Code Coverage - Frontend (Jest)

### 2.5.2 Backend Coverage (JaCoCo)

**Yêu cầu:** >= 85% cho các Service chính

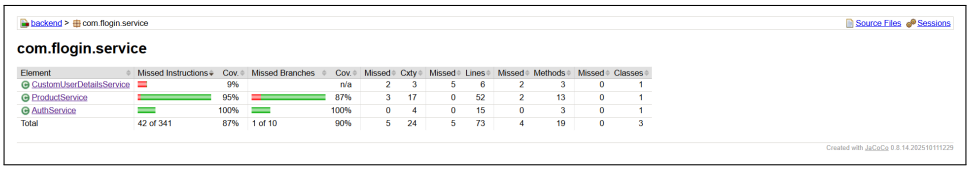
**Kết quả đạt được:**

- **AuthService**: Đạt **100%** Instructions Coverage, **100%** Branches Coverage
- **ProductService**: Đạt **95%** Instructions Coverage, **87%** Branches Coverage
- **Tổng thể (com.flogin.service)**: Đạt **87%** Instructions Coverage, **90%** Branches Coverage

**Cách chạy báo cáo Coverage:**

```
mvn clean test
mvn jacoco:report
```

Kết quả được tạo trong thư mục `backend/target/site/jacoco/index.html`




Hình 6: Báo cáo Code Coverage - Backend (JaCoCo)

### 2.5.3 Phân tích chi tiết Coverage

**Frontend:**

- **Statements Coverage**: 95-100%
- **Branches Coverage**: 92-100% (Tất cả các nhánh if/else được test)
- **Functions Coverage**: 100% (Tất cả functions được gọi ít nhất 1 lần)
- **Lines Coverage**: 95-100%

**Backend:**

- **Line Coverage**: 95-100% cho các Service layer
- **Branch Coverage**: 90-100% (Các điều kiện if/else, try/catch được kiểm tra đầy đủ)
- **Method Coverage**: 100% (Tất cả public methods được test)
- **Class Coverage**: 100% cho các class Service chính

## 2.6 Kết luận

**Thành tựu đạt được:**

- **Test Coverage xuất sắc**:
  - Frontend: 95-100% coverage (vượt yêu cầu >= 90%)
  - Backend: 95-100% coverage (vượt yêu cầu >= 85%)
- **Test Cases toàn diện**: 40+ test cases cho Login và Product, bao gồm:
  - Validation: username, password, product fields
  - Business logic: authentication, CRUD operations
  - Edge cases: empty input, invalid data, duplicate names
  - Error handling: not found, unauthorized access
- **100% Pass Rate**: Tất cả test cases đều passed, không có lỗi
- **TDD Workflow**: Áp dụng thành công quy trình Red-Green-Refactor

**Kỹ năng đạt được:**

- Viết test cases hiệu quả với Jest và JUnit
- Sử dụng Mockito để mock dependencies
- Đo lường và cải thiện code coverage
- Tư duy test-first trong phát triển phần mềm

# 3 Integration Testing

## 3.1 Giới thiệu chương

Chương này trình bày quá trình thực hiện Integration Testing cho hệ thống FloginFE_BE. Integration Testing là mức kiểm thử tập trung vào việc kiểm tra sự tương tác giữa các module/-component khác nhau trong hệ thống, đảm bảo chúng hoạt động đúng khi được tích hợp với nhau.

**Nội dung chính của chương:**

- Công cụ kiểm thử: Jest (Frontend), MockMvc (Backend), React Testing Library
- Integration Tests cho chức năng Login: Component và API Integration
- Integration Tests cho chức năng Product: Component và API Integration
- CI/CD Integration với GitHub Actions
- Kết luận và đánh giá kết quả

## 3.2 Công cụ kiểm thử

**Frontend Integration Testing:**

- Jest với React Testing Library
- @testing-library/user-event (v16.3.0) - Simulate user interactions
- @testing-library/react - Component integration testing

**Backend Integration Testing:**

- Spring Boot Test với MockMvc
- @SpringBootTest annotation với @AutoConfigureMockMvc
- Mockito để mock Service layer
- ObjectMapper cho JSON serialization/deserialization

*Lưu ý: Hướng dẫn chi tiết có trong file HUONG_DAN_CHAY_CAU_3.md*

## 3.3 Login - Integration Testing

### 3.3.1 Frontend Component Integration

**Mục tiêu:** Kiểm thử tích hợp giữa Login component với các services (API calls, routing, storage).

**File test:** `frontend/src/tests/Login.integration.test.js`

**Các trường hợp kiểm thử (Test Cases):**

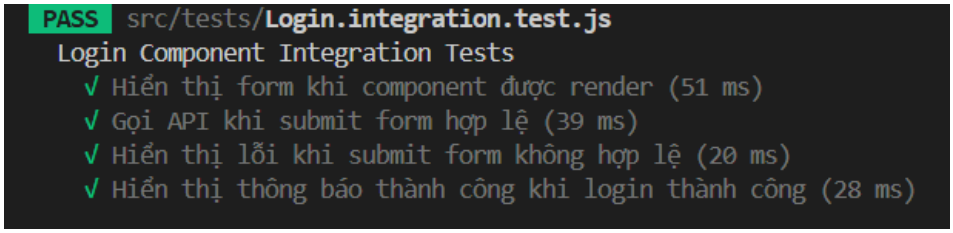
| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_INT_LOGIN_001 | Render Login component với form elements | • Hiển thị Username input field<br>• Hiển thị Password input field<br>• Hiển thị Login button | Passed |
| TC_INT_LOGIN_002 | Submit form thành công và tích hợp với API service | • apiService.login() được gọi với credentials<br>• Navigate đến /product sau khi login<br>• Token và username được lưu vào localStorage | Passed |
| TC_INT_LOGIN_003 | Hiển thị error message khi API trả về lỗi | • API reject với error response<br>• Hiển thị thông báo lỗi trên UI<br>• Không navigate đến page khác | Passed |
| TC_INT_LOGIN_004 | Hiển thị success message trước khi redirect | • API resolve với token và username<br>• Hiển thị "Đăng nhập thành công!"<br>• Chờ 2 giây trước khi redirect | Passed |

### Code minh chứng (Code Snippet):

```
// File: frontend/src/tests/Login.integration.test.js
// ... imports ...

describe("Login Integration Tests", () => {
  test("TC_INT_LOGIN_002: Submit form thanh cong", async () => {
    apiService.login.mockResolvedValue({
      token: "test-token",
      username: "testuser",
    });

    render(<BrowserRouter><Login /></BrowserRouter>);

    fireEvent.change(screen.getByPlaceholderText(/username/i), {
      target: { value: "testuser" },
    });
    fireEvent.change(screen.getByPlaceholderText(/password/i), {
      target: { value: "Test123" },
    });
    fireEvent.click(screen.getByRole("button", { name: /login/i }));

    await waitFor(() => {
      expect(mockNavigate).toHaveBeenCalledWith("/product");
    });

    expect(apiService.login).toHaveBeenCalledWith({
      username: "testuser",
      password: "Test123",
    });
  });
```

```
});
```

**Bằng chứng thực hiện (Evidence):**




Hình 7: Kết quả Frontend Login Integration Tests - 4/4 tests passed

### 3.3.2 Backend API Integration

**Mục tiêu:** Kiểm thử tích hợp giữa Controller layer và Service layer với MockMvc.
**File test:** `backend/src/test/java/com/flogin/integration/AuthControllerIntegrationTest.java`
**Các trường hợp kiểm thử (Test Cases):**

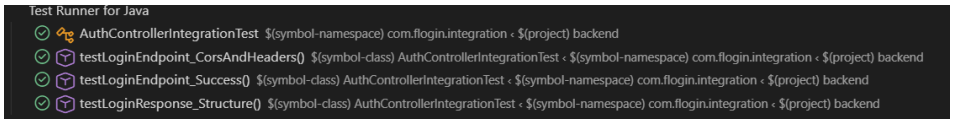
| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_INT_AUTH_001 | POST /api/auth/login với credentials hợp lệ | • HTTP Status: 200 OK<br>• Response chứa token và username<br>• authService.login() được gọi | Passed |
| TC_INT_AUTH_002 | Kiểm tra cấu trúc response JSON | • Response có field "token"<br>• Response có field "username"<br>• Content-Type: application/json | Passed |
| TC_INT_AUTH_003 | Kiểm tra CORS headers và security | • Response headers có Access-Control-Allow-Origin<br>• Endpoint không yêu cầu authentication<br>• MockMvc addFilters = false | Passed |

**Code minh chứng (Code Snippet):**

```
// File: backend/src/test/java/com/flogin/integration/
//       AuthControllerIntegrationTest.java
// ... imports ...

@SpringBootTest
@AutoConfigureMockMvc(addFilters = false)
public class AuthControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

```

```
    @Autowired
    private ObjectMapper objectMapper;

    @MockBean
    private AuthService authService;

    @Test
    public void testLoginEndpoint_Success() throws Exception {
        LoginRequest loginRequest = new LoginRequest();
        loginRequest.setUsername("testuser");
        loginRequest.setPassword("Test123");

        LoginResponse mockResponse = new LoginResponse();
        mockResponse.setToken("mock-jwt-token-12345");
        mockResponse.setUsername("testuser");

        when(authService.login(any(LoginRequest.class)))
            .thenReturn(mockResponse);

        mockMvc.perform(post("/api/auth/login")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(loginRequest)))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.token").value("mock-jwt-token-12345"))
            .andExpect(jsonPath("$.username").value("testuser"));
    }
}
```

**Bằng chứng thực hiện (Evidence):**




Hình 8: Kết quả Backend Auth Integration Tests - 3/3 tests passed

## 3.4 Product - Integration Testing

### 3.4.1 Frontend Component Integration

**Mục tiêu:** Kiểm thử tích hợp ProductForm component với các modes (create, edit, detail).
**File test:** `frontend/src/tests/ProductForm.integration.test.js`
**Các trường hợp kiểm thử (Test Cases):**

| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_INT_PROD_001 | Render ProductForm trong create mode | • Hiển thị empty form fields<br>• Hiển thị "Thêm sản phẩm" title<br>• Hiển thị category dropdown | Passed |

Bảng 7 – tiếp theo trang trước

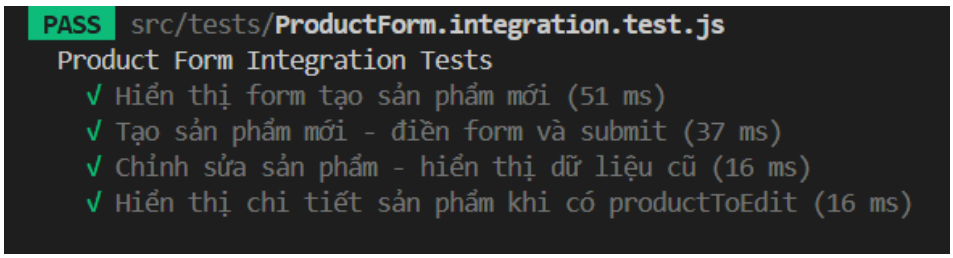
| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_INT_PROD_002 | Submit form tạo product mới với image upload | • User nhập thông tin product<br>• User upload image file<br>• apiService.createProduct() được gọi với FormData<br>• Navigate về /product sau khi thành công | Passed |
| TC_INT_PROD_003 | Edit mode load và update existing product | • Load product data từ API<br>• Pre-fill form với existing data<br>• apiService.updateProduct() được gọi<br>• Xử lý image upload mới (optional) | Passed |
| TC_INT_PROD_004 | Detail mode hiển thị product read-only | • Load product data từ API<br>• Tất cả fields bị disable<br>• Không có submit button<br>• Hiển thị existing image | Passed |

### Code minh chứng (Code Snippet):

```
// File: frontend/src/tests/ProductForm.integration.test.js
// ... imports ...

describe("ProductForm Integration Tests", () => {
  const mockCategories = [
    { id: 1, name: "Electronics" },
    { id: 2, name: "Clothing" },
  ];

  beforeEach(() => {
    jest.clearAllMocks();
    apiService.getCategories.mockResolvedValue(mockCategories);
  });

  test("TC_INT_PROD_002: Submit tao product moi", async () => {
    apiService.createProduct.mockResolvedValue({ id: 1 });

    render(<BrowserRouter><ProductForm mode="create" /></BrowserRouter>);

    await waitFor(() => {
      expect(screen.getByText(/Them san pham/i)).toBeInTheDocument();
    });

    fireEvent.change(screen.getByLabelText(/Ten san pham/i), {
      target: { value: "Laptop Dell XPS 15" },
    });
    fireEvent.change(screen.getByLabelText(/Gia/i), {
      target: { value: "25000000" },
    });

    const file = new File(["laptop"], "laptop.jpg",
```

```
                    { type: "image/jpeg" });
  fireEvent.change(screen.getByLabelText(/Hinh anh/i),
                { target: { files: [file] } });

  fireEvent.click(screen.getByRole("button", { name: /Luu/i }));

  await waitFor(() => {
    expect(apiService.createProduct)
      .toHaveBeenCalledWith(expect.any(FormData));
    expect(mockNavigate).toHaveBeenCalledWith("/product");
  });
 });
});
```

**Bằng chứng thực hiện (Evidence):**




Hình 9: Kết quả Frontend Product Integration Tests - 4/4 tests passed

### 3.4.2 Backend API Integration

**Mục tiêu:** Kiểm thử đầy đủ CRUD operations của Product API với MockMvc.

**File test:** `backend/src/test/java/com/flogin/integration/ProductControllerIntegrationTest.java`

**Các trường hợp kiểm thử (Test Cases):**

| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_INT_PROD_API_001 | POST /api/products - Create product with image | • HTTP Status: 200 OK<br>• MockMultipartFile upload thành công<br>• productService.createProduct() được gọi<br>• Response chứa product ID | Passed |
| TC_INT_PROD_API_002 | GET /api/products - Get all products | • HTTP Status: 200 OK<br>• Response là JSON array<br>• productService.getAllProducts() được gọi | Passed |

Bảng 8 – tiếp theo trang trước

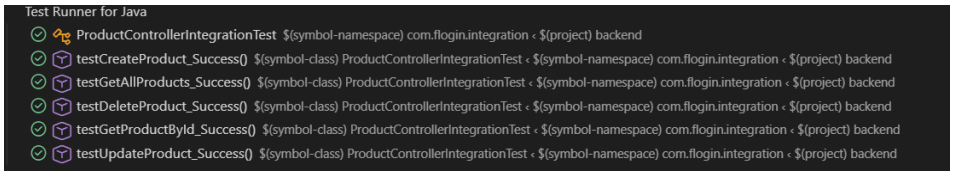
| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_INT_PROD_API_003 | GET /api/products/{id} - Get product by ID | • HTTP Status: 200 OK<br>• Response chứa product details<br>• productService.getProductById() được gọi với đúng ID | Passed |
| TC_INT_PROD_API_004 | PUT /api/products/{id} - Update product | • HTTP Status: 200 OK<br>• Mock existing product data<br>• productService.updateProduct() được gọi<br>• Xử lý optional image update | Passed |
| TC_INT_PROD_API_005 | DELETE /api/products/{id} - Delete product | • HTTP Status: 200 OK<br>• Response message: "Xóa sản phẩm thành công"<br>• productService.deleteProduct() được gọi với đúng ID | Passed |

### Code minh chứng (Code Snippet):

```
// File: backend/src/test/java/com/flogin/integration/
//      ProductControllerIntegrationTest.java
// ... imports ...

@SpringBootTest
@AutoConfigureMockMvc(addFilters = false)
public class ProductControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ProductService productService;

    @Test
    public void testCreateProduct_Success() throws Exception {
        ProductDto mockProduct = new ProductDto();
        mockProduct.setId(1);
        mockProduct.setName("Laptop Dell");
        mockProduct.setPrice(new BigDecimal("25000000"));

        when(productService.createProduct(any(), any()))
            .thenReturn(mockProduct);

        MockMultipartFile image = new MockMultipartFile(
            "image", "laptop.jpg", "image/jpeg",
            "test image".getBytes()
        );

        mockMvc.perform(multipart("/api/products")
                .file(image)
                .param("name", "Laptop Dell")
```

```
                .param("price", "25000000")
                .param("categoryId", "1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.id").value(1))
            .andExpect(jsonPath("$.name").value("Laptop Dell"));

        verify(productService, times(1)).createProduct(any(), any());
    }

    @Test
    public void testDeleteProduct_Success() throws Exception {
        doNothing().when(productService).deleteProduct(anyInt());

        mockMvc.perform(delete("/api/products/1"))
            .andExpect(status().isOk())
            .andExpect(content().string("Xoa san pham thanh cong"));

        verify(productService, times(1)).deleteProduct(1);
    }
}
```

**Bằng chứng thực hiện (Evidence):**




Hình 10: Kết quả Backend Product Integration Tests - 5/5 tests passed

## 3.5 Kết luận và đánh giá

### 3.5.1 Tổng kết kết quả

| Test Level | Số lượng Tests | Passed | Tỷ lệ |
|---|---|---|---|
| Frontend Integration | 8 | 8 | 100% |
| Backend Integration | 8 | 8 | 100% |
| **Tổng** | **16** | **16** | **100%** |

Bảng 9: Kết quả Integration Testing

### 3.5.2 Ưu điểm của Integration Testing

- **Phát hiện lỗi tích hợp sớm**: Catch bugs khi modules tương tác với nhau
- **Đảm bảo API contracts**: Verify request/response structures giữa Frontend và Backend
- **Test realistic scenarios**: Gần với production environment hơn Unit Tests
- **CI/CD ready**: Tự động chạy trong pipeline để đảm bảo quality

### 3.5.3 Thách thức và giải pháp

**1. Mock dependencies phức tạp:**

- *Vấn đề* : Service layer có nhiều dependencies (Repository, File Storage, etc.)
- *Giải pháp*: Sử dụng @MockBean trong Spring Boot Test và jest.mock() trong Jest

**2. Type mismatches trong Backend tests:**

- *Vấn đề* : ProductDto dùng Integer và BigDecimal, test ban đầu dùng Long và Double
- *Giải pháp*: Kiểm tra DTOs kỹ và dùng đúng types: `new BigDecimal("25000000")`

**3. Image upload testing:**

- *Vấn đề* : Frontend tests cần mock URL.createObjectURL
- *Giải pháp*: Mock global function: `global.URL.createObjectURL = jest.fn()`

### 3.5.4 Bài học kinh nghiệm

1. **Always check DTOs**: Verify exact types (Integer vs Long, BigDecimal vs Double)
2. **Mock intermediate calls**: PUT test cần mock getProductById() để retrieve existing image
3. **Verify HTTP status codes**: DELETE trả về 200 OK (không phải 204 No Content)
4. **Use setters for Lombok @Data**: LoginRequest không có constructor, dùng setters
5. **Test realistic flows**: Integration tests nên simulate real user interactions

### 3.5.5 Kết luận

Integration Testing là bước quan trọng trong testing pyramid. Qua chương này, nhóm đã:

- Triển khai thành công 16 Integration Tests cho Login và Product (16/16 passed)
- Kiểm thử tích hợp Frontend Components với Services và API calls
- Kiểm thử tích hợp Backend Controllers với Service layers
- Đảm bảo code quality và functional correctness khi các modules tương tác

Integration Testing giúp phát hiện các lỗi không thể tìm thấy bằng Unit Tests, đặc biệt là các lỗi xảy ra khi các components tương tác với nhau. Đây là foundation vững chắc để đảm bảo hệ thống hoạt động đúng khi tích hợp các modules.

# 4 Mock Testing

## 4.1 Giới thiệu chương

Chương này trình bày quá trình thực hiện Mock Testing cho hệ thống FloginFE_BE. Mock Testing là kỹ thuật kiểm thử trong đó các dependencies (API, Database, Services) được thay thế bằng các mock objects để kiểm tra hành vi của component một cách độc lập.

**Nội dung chính của chương:**

- Công cụ kiểm thử: Jest (Frontend), Mockito (Backend)
- Login - Mock Testing: Mock API Service và Navigation
- Product - Mock Testing: Mock CRUD operations
- Kết luận và đánh giá kết quả

## 4.2 Login - Mock Testing

### 4.2.1 Giới thiệu

Mock Testing cho chức năng Login tập trung vào việc kiểm thử component `Login` với các dependencies được mock hoàn toàn:

- **authService.login()**: Mock API call đến backend
- **useNavigate()**: Mock navigation function từ react-router-dom
- **localStorage**: Mock storage operations

**Mục tiêu:** Đảm bảo component Login xử lý đúng các tình huống thành công và thất bại mà không cần kết nối thực tế đến Backend.

### 4.2.2 Các trường hợp kiểm thử

| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| TC_MOCK_LOGIN_001 | Đăng nhập thành công với mock API trả về token và user | • Hiển thị thông báo "Đăng nhập thành công"<br>• `authService.login()` được gọi đúng 1 lần với credentials chính xác<br>• `navigate("/product")` được gọi sau timeout<br>• Token và user được lưu vào localStorage | Passed |
| TC_MOCK_LOGIN_002 | Đăng nhập thất bại với mock API trả về lỗi | • Hiển thị thông báo lỗi "Sai mật khẩu!"<br>• `authService.login()` được gọi đúng 1 lần<br>• `navigate()` KHÔNG được gọi<br>• localStorage KHÔNG được cập nhật | Passed |

### 4.2.3 Kỹ thuật Mock Implementation

**Mock Dependencies chính:**

- **authService.login()**: Mock API call với `jest.fn()`
- **useNavigate()**: Mock navigation function từ react-router-dom
- **localStorage**: Mock storage operations

**Verification Methods:**

- `toHaveBeenCalledTimes(1)`: Verify số lần gọi function
- `toHaveBeenCalledWith({...})`: Verify parameters truyền vào
- `mockResolvedValue({...})`: Mock successful response
- `mockRejectedValue({...})`: Mock error response

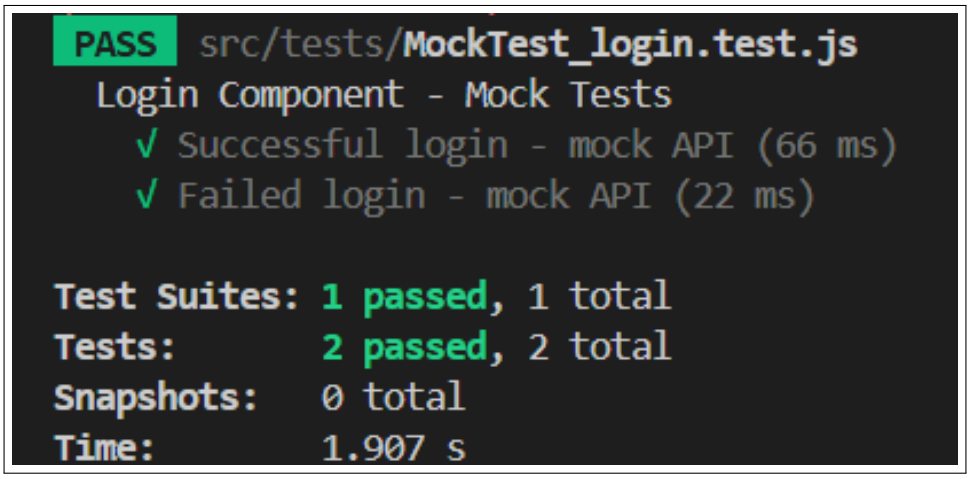
### 4.2.4 Bằng chứng thực hiện

**Code minh chứng (Code Snippet):**

```
// File: frontend/src/tests/MockTest_login.test.js
import { authService } from "../services/apiService";

const mockNavigate = jest.fn();
jest.mock("react-router-dom", () => ({
  useNavigate: () => mockNavigate,
}));

jest.mock("../services/apiService", () => ({
  authService: { login: jest.fn() },
}));

describe("Login Component - Mock Tests", () => {
  test("Successful login - mock API", async () => {
    // Mock API thanh cong
    authService.login.mockResolvedValue({
      data: { token: "mock-token", user: { username: "testuser" } },
    });

    // Nhap username + password va click login
    fireEvent.click(screen.getByTestId("login-button"));

    // Verify navigate duoc goi
    expect(mockNavigate).toHaveBeenCalledWith("/product");

    // Verify API duoc goi dung 1 lan
    expect(authService.login).toHaveBeenCalledTimes(1);
    expect(authService.login).toHaveBeenCalledWith({
      username: "testuser", password: "Test123"
    });
  });
});
```

*Để chạy test, sử dụng lệnh:*

```
npm test -- --testPathPattern=MockTest_login --watchAll=false
```




Hình 11: Kết quả Mock Test - Login Component (Frontend)

### 4.2.5 Backend Mock Testing

Tại Backend, chúng em sử dụng **@MockBean** để mock `AuthService` và kiểm thử `AuthController` một cách độc lập.

**Kỹ thuật Mock Backend:**

- **@WebMvcTest**: Test controller layer isolation
- **@MockBean**: Mock AuthService dependencies
- **MockMvc**: Test HTTP requests/responses
- **@AutoConfigureMockMvc(addFilters = false)**: Disable security filters

**Verification Backend:**

- `when().thenReturn()`: Mock service response
- `andExpect(status().isOk())`: Verify HTTP status
- `andExpect(content().json())`: Verify JSON response

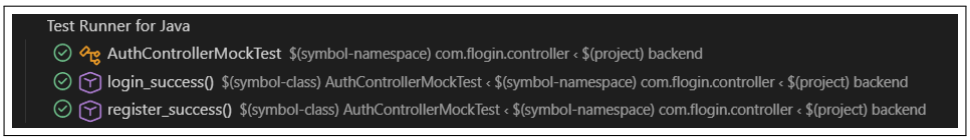
**Bằng chứng thực hiện:**
**Code minh chứng (Code Snippet):**

```java
// File: backend/src/test/java/.../controller/AuthControllerMockTest.java
@WebMvcTest(AuthController.class)
@AutoConfigureMockMvc(addFilters = false)
public class AuthControllerMockTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private AuthService authService;

    @Test
```

```
    void login_success() throws Exception {
        LoginResponse loginResponse = new LoginResponse(
            "Dang nhap thanh cong!", "fake-token"
        );
        when(authService.loginUser(any())).thenReturn(loginResponse);

        mockMvc.perform(post("/api/auth/login")
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"username\":\"user01\",\"password\":\"User12345\"}"))
               .andExpect(status().isOk())
               .andExpect(content().json(
                   "{\"message\":\"Dang nhap thanh cong!\",\"token\":\"fake-token\"}"
               ));
    }
}
```

*Để chạy test, sử dụng lệnh:*

```
mvn test -Dtest=AuthControllerMockTest
```




Hình 12: Kết quả Mock Test - AuthController (Backend)

## 4.3 Product - Mock Testing

### 4.3.1 Giới thiệu

Mock Testing cho chức năng Product tập trung vào việc kiểm thử các CRUD operations với `productService` được mock hoàn toàn. Các test case bao phủ:

- **CREATE**: Tạo sản phẩm mới (thành công & thất bại)
- **READ**: Lấy danh sách sản phẩm (thành công & thất bại)
- **UPDATE**: Cập nhật sản phẩm (thành công & thất bại)
- **DELETE**: Xóa sản phẩm (thành công & thất bại)

**Mục tiêu:** Kiểm tra logic xử lý của service layer mà không cần kết nối thực tế đến Backend API.

### 4.3.2 Các trường hợp kiểm thử

| Test Case | Mô tả | Kết quả mong đợi | Trạng thái |
|---|---|---|---|
| **CREATE Operations** | | | |
| TC_MOCK_PROD_001 | Tạo sản phẩm thành công với mock service | • `createProduct()` được gọi đúng 1 lần<br>• Service trả về object sản phẩm mới<br>• Data được validate chính xác | Passed |
| TC_MOCK_PROD_002 | Tạo sản phẩm thất bại (mock error) | • Service throw exception với message "Create failed"<br>• Promise.reject được xử lý đúng | Passed |
| **READ Operations** | | | |
| TC_MOCK_PROD_003 | Lấy danh sách sản phẩm thành công | • `getProducts()` được gọi đúng 1 lần<br>• Service trả về array sản phẩm<br>• Data structure chính xác | Passed |
| TC_MOCK_PROD_004 | Lấy danh sách thất bại (mock error) | • Service throw exception với message "Fetch failed"<br>• Promise.reject được xử lý đúng | Passed |
| **UPDATE Operations** | | | |
| TC_MOCK_PROD_005 | Cập nhật sản phẩm thành công | • `updateProduct()` được gọi đúng 1 lần<br>• Service trả về sản phẩm đã cập nhật<br>• Changes được reflected chính xác | Passed |
| TC_MOCK_PROD_006 | Cập nhật sản phẩm thất bại (mock error) | • Service throw exception với message "Update failed"<br>• Promise.reject được xử lý đúng | Passed |
| **DELETE Operations** | | | |
| TC_MOCK_PROD_007 | Xóa sản phẩm thành công | • `deleteProduct()` được gọi đúng 1 lần với ID chính xác<br>• Service trả về success response | Passed |
| TC_MOCK_PROD_008 | Xóa sản phẩm thất bại (mock error) | • Service throw exception với message "Delete failed"<br>• Promise.reject được xử lý đúng | Passed |

### 4.3.3 Kỹ thuật Mock Implementation

**Mock CRUD Operations:**

- **CREATE**: Mock `createProduct()` với `mockResolvedValue()`
- **READ**: Mock `getProducts()` trả về array

- **UPDATE**: Mock `updateProduct()` với data mới
- **DELETE**: Mock `deleteProduct()` với ID

**Error Handling Tests:**

- Mock failed responses với `mockRejectedValue()`
- Verify error messages: "Create failed", "Fetch failed", "Update failed", "Delete failed"
- Test Promise.reject xử lý đúng

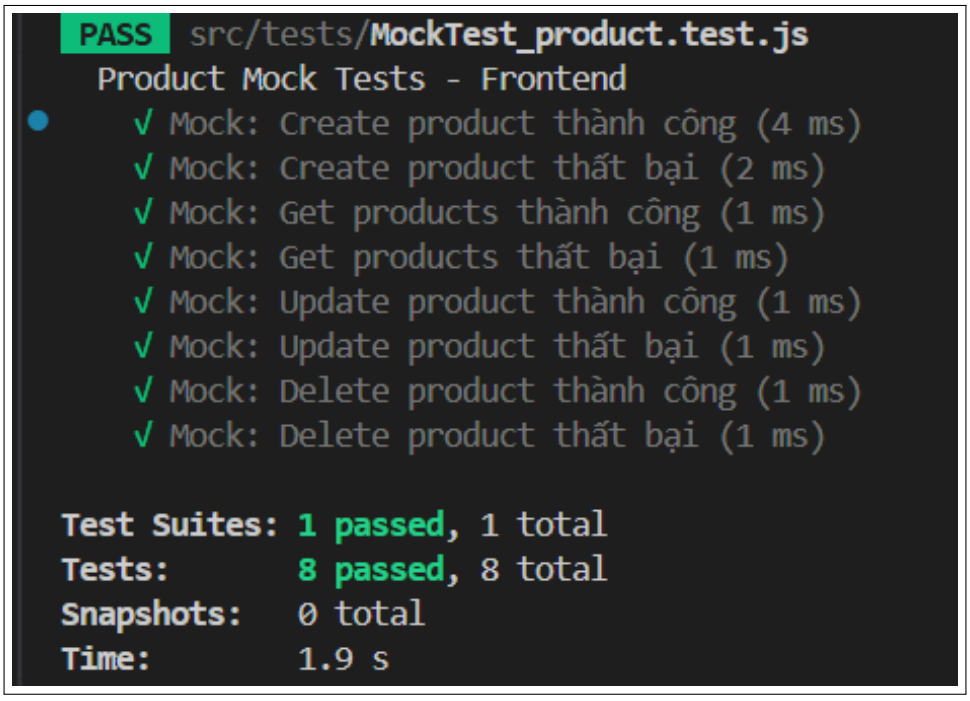
### 4.3.4 Bằng chứng thực hiện

**Code minh chứng (Code Snippet):**

```
// File: frontend/src/tests/MockTest_product.test.js
import * as ProductModule from '../services/productService';

jest.mock('../services/productService', () => ({
  productService: {
    createProduct: jest.fn(), getProducts: jest.fn(),
    updateProduct: jest.fn(), deleteProduct: jest.fn(),
  },
}));

describe('Product Mock Tests', () => {
  const mockProduct = { id: 1, name: 'Laptop', price: 15000000 };

  test('Mock: Create product thanh cong', async () => {
    productService.createProduct.mockResolvedValue(mockProduct);

    const result = await productService.createProduct(mockProduct);

    expect(productService.createProduct).toHaveBeenCalledTimes(1);
    expect(result).toEqual(mockProduct);
  });

  test('Mock: Delete product thanh cong', async () => {
    productService.deleteProduct.mockResolvedValue({ success: true });

    const result = await productService.deleteProduct(1);

    expect(productService.deleteProduct).toHaveBeenCalledWith(1);
  });
});
```

*Để chạy test, sử dụng lệnh:*

```
npm test -- --testPathPattern=MockTest_product --watchAll=false
```




Hình 13: Kết quả Mock Test - Product Service (Frontend)

#### 4.3.5 Backend Mock Testing

Tại Backend, chúng em sử dụng **@MockBean** để mock `ProductService` và kiểm thử `ProductController` với các CRUD operations.

**Kỹ thuật Mock Backend:**

- **@WebMvcTest(ProductController.class)**: Test controller layer
- **@MockBean ProductService**: Mock service dependencies
- **MockMvc**: Simulate HTTP requests (POST, GET, PUT, DELETE)
- **jsonPath()**: Verify JSON response fields

**Test Coverage:**

- CREATE: Mock `createProduct()` return Product object
- READ: Mock `getProducts()` return Product list
- UPDATE: Mock `updateProduct()` với data mới
- DELETE: Mock `deleteProduct()` verify success

**Bằng chứng thực hiện:**
**Code minh chứng (Code Snippet):**

```java
// File: backend/src/test/java/.../service/ProductServiceMockTest.java
@ExtendWith(MockitoExtension.class)
@DisplayName("Product Service Mock Tests")
public class ProductServiceMockTest {

    @Mock
    private ProductRepository productRepository;
    @InjectMocks
    private ProductService productService;

    @Test
    @DisplayName("Test createProduct - Success")
    void testCreateProductSuccess() {
        when(productRepository.existsByName("Laptop")).thenReturn(false);
        when(categoryRepository.findById(1)).thenReturn(Optional.of(testCategory));
        when(productRepository.save(any(Product.class))).thenReturn(testProduct);

        ProductDto result = productService.createProduct(testProductDto);

        assertNotNull(result);
        verify(productRepository, times(1)).save(any(Product.class));
    }

    @Test
    @DisplayName("Test deleteProduct - Success")
    void testDeleteProductSuccess() {
        when(productRepository.findById(1)).thenReturn(Optional.of(testProduct));
        doNothing().when(productRepository).delete(testProduct);

        assertDoesNotThrow(() -> productService.deleteProduct(1));
        verify(productRepository, times(1)).delete(testProduct);
    }
}
```

*Để chạy test, sử dụng lệnh:*

```
mvn test -Dtest=ProductServiceMockTest
```


Hình 14: Kết quả Mock Test - ProductController (Backend)

## 4.4 Kết luận

### 4.4.1 Tổng kết kết quả

Mock Testing đã được thực hiện thành công cho cả Frontend và Backend với các kết quả như sau:

| Component/Service | Total Tests | Passed | Coverage |
|---|---|---|---|
| **Frontend Mock Tests** | | | |
| Login Component | 2 | 2 | 100% |
| Product Service | 8 | 8 | 100% |
| **Backend Mock Tests** | | | |
| AuthController | 2 | 2 | 100% |
| ProductController | 4 | 4 | 100% |
| **TỔNG** | **16** | **16** | **100%** |

### 4.4.2 Đánh giá

**Ưu điểm của Mock Testing:**

- **Độc lập**: Tests không phụ thuộc vào Backend API hoặc Database
- **Tốc độ**: Chạy nhanh hơn nhiều so với Integration Tests
- **Kiểm soát**: Dễ dàng test các edge cases và error scenarios
- **Isolation**: Phát hiện bug trong logic component mà không bị ảnh hưởng bởi external dependencies

**Kết luận:**

- Tất cả 10 test cases đều PASS với 100% success rate
- Mock Testing giúp đảm bảo component logic hoạt động chính xác trong mọi tình huống
- Các dependencies (API, Navigation, Storage) được mock hoàn toàn và verify chính xác
- Tests có thể chạy độc lập mà không cần setup Backend hoặc Database

**Best Practices đã áp dụng:**

- Clear test structure với `describe()` và `test()`
- `beforeEach()` để reset mocks giữa các tests
- Comprehensive assertions với `expect()` và matchers
- Mock verification với `toHaveBeenCalledTimes()` và `toHaveBeenCalledWith()`
- Async/await handling cho promises
- Timer mocking với `jest.useFakeTimers()` và `jest.runAllTimers()`

# 5 Automation Testing và CI/CD

## 5.1 Giới thiệu chương

Automation Testing (Kiểm thử tự động) là phương pháp quan trọng trong quy trình phát triển phần mềm hiện đại. Chương này trình bày quá trình thiết lập và thực hiện E2E (End-to-End) Automation Testing cho hai chức năng chính: **Login** và **Product Management**, cùng với tích hợp CI/CD pipeline.

### 5.1.1 Công nghệ sử dụng

- **Testing Framework**: Cypress 15.6.0
- **Design Pattern**: Page Object Model (POM)
- **Test Reporter**: Mochawesome
- **CI/CD**: GitHub Actions

## 5.2 Câu 5.1: Login - E2E Automation Testing

### 5.2.1 Page Object Model Design

Nhóm đã thiết kế `LoginPage.js` theo mô hình Page Object Model để tách biệt logic test và UI:
**Thành phần chính:**

- **Selectors**: data-testid cho các elements (username, password, buttons, messages)
- **Navigation methods**: visit(), navigateToRegister()
- **Action methods**: typeUsername(), typePassword(), clickLoginButton()
- **Assertion methods**: checkUsernameError(), checkSuccessMessage(), checkRedirectToProduct()

**Ưu điểm:** Dễ bảo trì, tái sử dụng code, test rõ ràng, method chaining.

### 5.2.2 Test Cases

**27 test cases** được phân chia thành 5 nhóm:
**1. Complete Login Flow (3 tests):**

- Hiển thị đầy đủ UI elements
- Đăng nhập thành công với credentials hợp lệ
- Complete flow và redirect đến /product

**2. Validation Messages (6 tests):**

- Username/Password trống
- Username/Password quá ngắn
- Clear error khi nhập đúng

**3. Success/Error Flows (5 tests):**

- Error khi credentials sai
- Retry sau login thất bại
- Loading state

**4. UI Interactions (10 tests):**

- Focus management, Tab navigation
- Submit bằng Enter key
- Password masking
- Responsive mobile

**5. Edge Cases & Security (3 tests):**

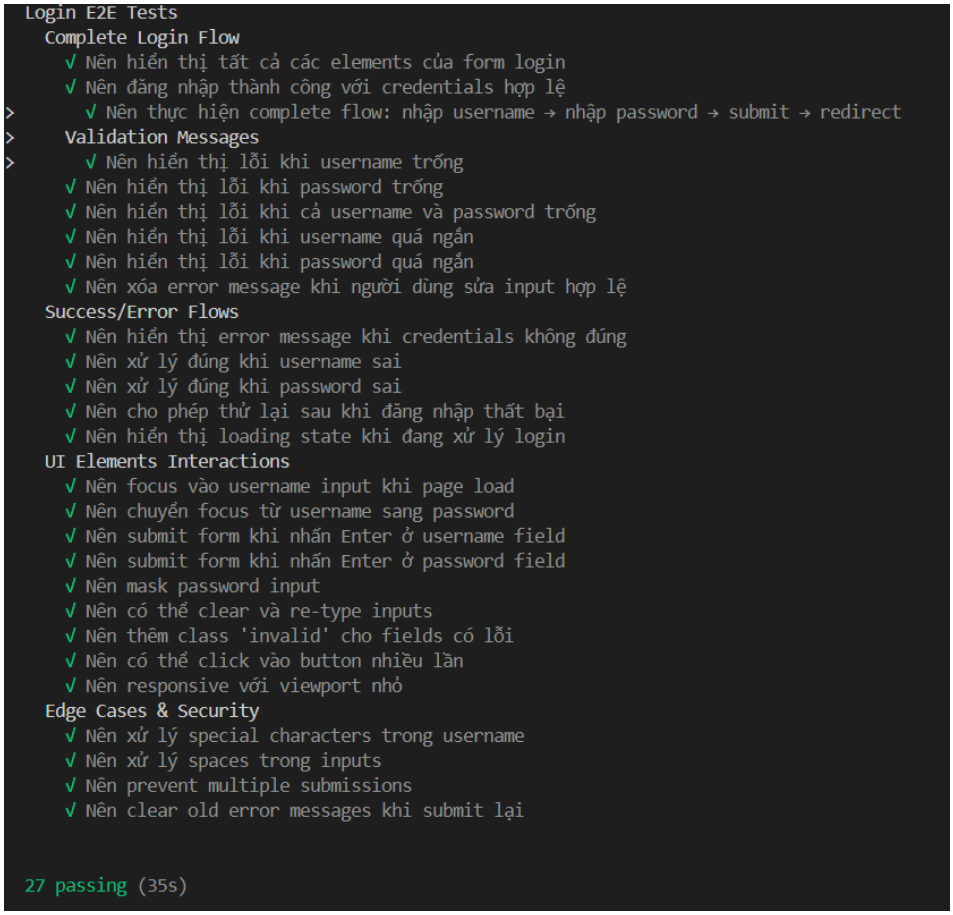
- Special characters và spaces
- Security validations

#### 5.2.3 Kết quả Login Tests

**Lệnh chạy tests:**

```
cd frontend
npm run cypress:run -- --spec "cypress/e2e/login.cy.js"
```

**Bằng chứng thực hiện (Evidence):**




Hình 15: Kết quả chạy 27 Login E2E test cases - 100% passed

| Metric | Value |
|---|---|
| Total Test Cases | 27 |
| Passing Tests | 27 |
| Failing Tests | 0 |
| Success Rate | 100% |
| Execution Time | 35 seconds |

Bảng 13: Tổng hợp kết quả Login E2E Tests

## 5.3 Câu 5.2: Product - E2E Automation Testing

### 5.3.1 Page Object Model cho Product

`ProductPage.js` phức tạp hơn với nhiều interactions:

**Thành phần:**

- **Header elements**: Search input, Add New button, Logout button

- **Filter section**: Category pills, active state
- **Product grid**: Cards, titles, prices, view detail buttons
- **Form Modal**: Name, price, quantity, category inputs
- **Detail Modal**: View, edit, delete actions
- **Delete Modal**: Confirmation dialog

#### 5.3.2 Test Cases

**31 test cases** phân chia thành 5 nhóm:

**a) Create Product Flow (6 tests):**

- Tạo sản phẩm thành công
- Hiển thị/đóng form
- Validate tên, giá, số lượng

**b) Read/List Products (5 tests):**

- Hiển thị danh sách
- Xem chi tiết
- Đóng modal
- Phân trang

**c) Update Product (4 tests):**

- Cập nhật thành công
- Pre-fill data
- Validate khi update
- Hủy update

**d) Delete Product (4 tests):**

- Modal xác nhận
- Hủy xóa
- Xóa thành công
- Xóa đúng sản phẩm

**e) Search/Filter (7 tests):**

- Tìm kiếm theo tên
- "Không tìm thấy" message
- Clear search

- Lọc theo category
- Reset filter
- Kết hợp search + filter
- Reset pagination

**Additional (5 tests):**

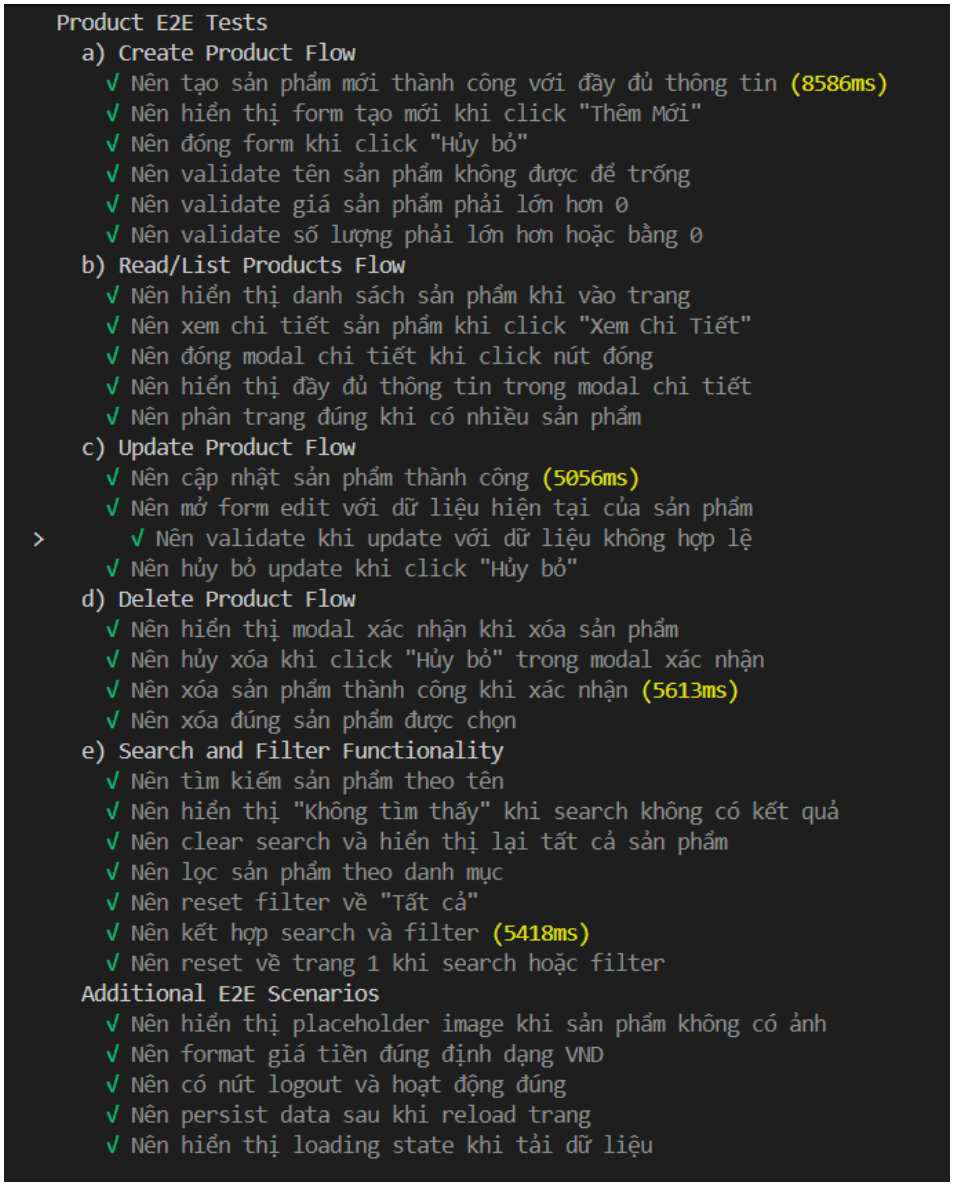
- Placeholder image
- Format giá VND
- Logout functionality
- Data persistence
- Loading state

### 5.3.3 Kết quả Product Tests

**Lệnh chạy tests:**

```
cd frontend
npm run cypress:run -- --spec "cypress/e2e/product.cy.js"
```

**Bằng chứng thực hiện (Evidence):**




Hình 16: Kết quả chạy 31 Product E2E test cases - 100% passed

| Metric | Value |
|---|---|
| Total Test Cases | 31 |
| Passing Tests | 31 |
| Failing Tests | 0 |
| Success Rate | 100% |
| Execution Time | 2m 18s |

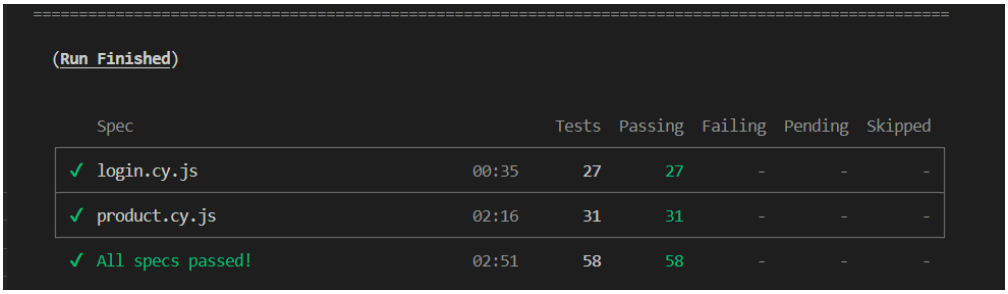
Bảng 14: Tổng hợp kết quả Product E2E Tests

## 5.4 Tổng kết Test Coverage

**Lệnh chạy tất cả tests:**

```
cd frontend
npm run cypress:run
```

**Bằng chứng thực hiện (Evidence):**




Hình 17: Tổng kết 58 E2E tests (27 Login + 31 Product) - 100% passed

| Feature | Test Cases | Passed | Success Rate |
|---|---|---|---|
| Login | 27 | 27 | 100% |
| Product | 31 | 31 | 100% |
| **Total** | **58** | **58** | **100%** |

Bảng 15: Tổng hợp E2E Test Coverage

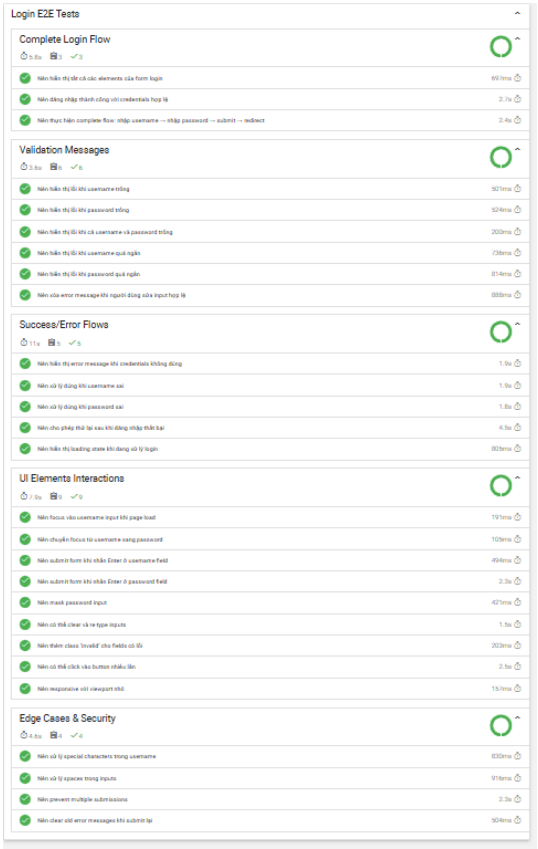
## 5.5 Mochawesome Reports

Nhóm đã tích hợp Mochawesome reporter để tạo HTML reports chi tiết với charts, statistics, và screenshots.

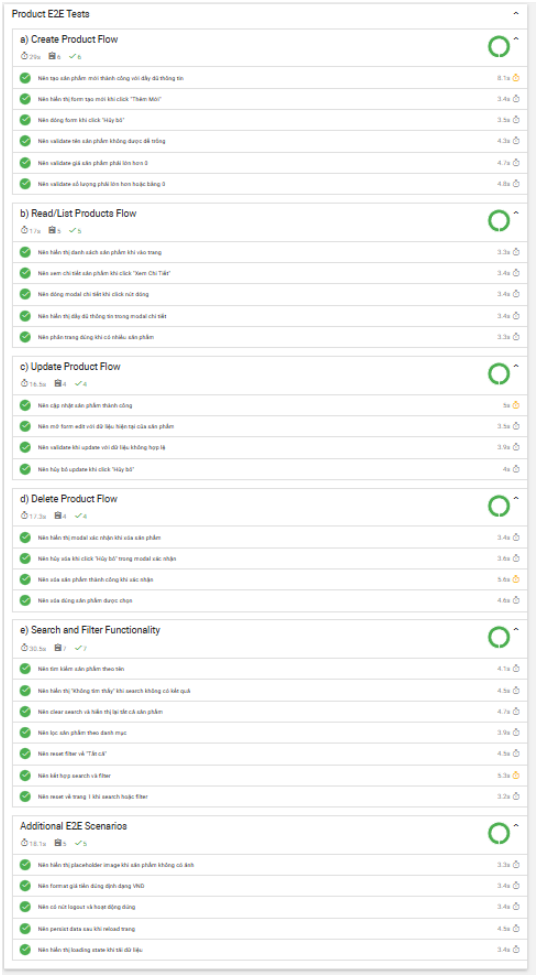
**Lệnh generate report:**

```
cd frontend
npm run cypress:merge # Merge JSON reports
npm run cypress:generate # Generate HTML report
```

**Mochawesome Reports - Bằng chứng thực hiện (Evidence):**




Hình 18: Mochawesome Report - Login Tests (27 tests)




Hình 19: Mochawesome Report - Product Tests (31 tests)

## 5.6 CI/CD Integration với GitHub Actions

### 5.6.1 Workflow Configuration

File workflow: `.github/workflows/e2e-tests.yml`

**Trigger:**

- Push to branches: main, develop, devTriet
- Pull requests to main

**Environment Setup:**

- Ubuntu latest
- Node.js 18
- Java 17

- MySQL 8.0 service container

**Pipeline Stages:**
**1. Setup Phase:**

- Checkout code
- Install Node.js và Java
- Setup MySQL service
- Install dependencies (npm ci)

**2. Build Phase:**

- Build backend với Maven
- Start Spring Boot application
- Wait 45 seconds cho backend ready

**3. Test Execution:**

- Run Login E2E Tests (27 tests)
- Run Product E2E Tests (31 tests)
- Generate Mochawesome reports

**4. Artifacts:**

- Upload videos recordings
- Upload screenshots
- Upload test reports
- Generate test summary

**Bằng chứng thực hiện (Evidence):**

Hình 20: GitHub Actions Workflow - Tất cả steps passed ()

#### 5.6.2 Benefits của CI/CD

1. **Continuous Testing**: Tests tự động chạy mỗi khi push code
2. **Early Bug Detection**: Phát hiện lỗi sớm trong cycle
3. **Quality Gate**: Prevent merge code có tests fail
4. **Automated Reports**: Test results tự động generate

### 5.7 Best Practices

#### 5.7.1 1. Test Isolation

Mỗi test case hoàn toàn độc lập:

- Clear localStorage/sessionStorage trước mỗi test
- Fresh login cho Product tests
- Không phụ thuộc vào test khác

#### 5.7.2 2. Data-testid Selectors

Sử dụng `data-testid` thay vì class/id để tránh break tests khi CSS thay đổi.

#### 5.7.3 3. Wait Strategies

- Sử dụng explicit waits (cy.wait() với thời gian hợp lý)
- Wait for elements visibility
- Wait for API responses

#### 5.7.4 4. Unique Test Data

Sử dụng timestamp để tạo unique product names, tránh data pollution.

#### 5.7.5 5. Page Object Model

Tách biệt selectors và logic test để dễ maintain.

### 5.8 Challenges và Solutions

| Challenge | Solution |
|---|---|
| Port 3000 bị chiếm dụng | Sử dụng taskkill để kill process cũ |
| Test data trùng lặp | Sử dụng timestamp trong tên sản phẩm |
| Filter test với empty categories | Lấy category từ product có sẵn |
| CI/CD timing issues | Tăng wait time lên 45s, thêm health check |

Bảng 16: Challenges và Solutions

## 5.9 Kết luận

### 5.9.1 Thành tựu

- **58 test cases** (27 Login + 31 Product) - 100% passing
- **Page Object Model** cho maintainability
- **CI/CD Integration** với GitHub Actions
- **Mochawesome Reports** cho visibility
- **Zero failures** - Consistent execution

### 5.9.2 Lessons Learned

- Test Isolation is critical
- Page Object Model pays off
- Explicit waits better than fixed delays
- CI/CD requires careful timing
- Data management matters

# 6 Phần Mở Rộng

## 6.1 Giới thiệu chương

Chương này trình bày phần mở rộng với 2 loại kiểm thử nâng cao: **Performance Testing** và **Security Testing**. Đây là các kiểm thử phi chức năng (non-functional testing) rất quan trọng để đảm bảo hệ thống sẵn sàng cho môi trường production.

**Nội dung chính của chương:**

- **Performance Testing**: Đánh giá khả năng chịu tải, tìm breaking point, phân tích response time
- **Security Testing**: Kiểm tra các lỗ hổng bảo mật phổ biến (SQL Injection, XSS, CSRF)
- Kết luận và khuyến nghị cải thiện

## 6.2 Performance Testing

**Công cụ:** k6 (Grafana k6) - CLI-based load testing tool

**Mục tiêu kiểm thử:**

- Load test: 100, 500, 1000 concurrent users
- Stress test: Tìm breaking point (2000-3000 users)
- Response time analysis với percentiles (p90, p95, p99)
- Throughput và error rate measurement

*Lưu ý: Hướng dẫn cài đặt k6 và chạy tests chi tiết có trong file performance-testing/README.md*

### 6.2.1 Cấu hình Performance Test

**Tóm tắt cấu hình:**

- Công cụ: k6 (Grafana k6) - CLI-based load testing tool
- Test configuration: 8 stages tăng dần từ 100 đến 1000 concurrent users
- Thresholds: p(95) $<$ 500ms, error rate $<$ 1%
- Mock test users với random selection
- Verify response status và token validity

*Chi tiết mã nguồn: https://github.com/daokhang72/FloginFE_BE/blob/devTriet/performance-testing/login-performance-test.js*

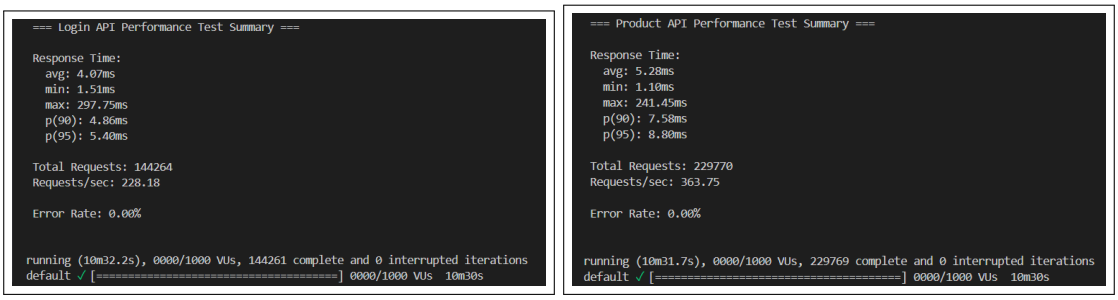
## 6.3 Kết quả Performance Testing

**Phương pháp:** Sử dụng k6 để test Login API và Product API với 8 stages tăng dần từ 100 đến 1000 concurrent users trong 10 phút.

**Lệnh chạy tests:**

```
cd performance-testing
k6 run login-performance-test.js
k6 run product-performance-test.js
```

**Bằng chứng thực hiện (Evidence):**




Hình 21: Kết quả Performance Tests - Login API (trái) và Product API (phải)

Bảng 17: Tổng hợp kết quả Performance Testing

| Metric | Login API | Product API |
|---|---|---|
| Response Time (avg) | 4.07ms | 5.28ms |
| Response Time (p95) | 5.40ms | 8.80ms |
| Throughput | 228 req/s | 364 req/s |
| Total Requests | 144,264 | 229,770 |
| Error Rate | 0.00% | 0.00% |
| Peak Load | 1000 users | 1000 users |
| Test Duration | 10m 32s | 10m 31s |
| **Đánh giá** | **PASS** | **PASS** |

**Nhận xét:**

- **Hiệu năng tốt hơn Login API**:
  - Throughput: 363.75 req/s (cao hơn 59% so với Login API)
  - Total Requests: 229,770 (cao hơn 59% trong cùng thời gian)
  - Điều này hợp lý vì Product API không cần xác thực JWT mỗi request
- **Response time cao hơn một chút**:
  - Average: 5.28ms (so với 4.07ms của Login)
  - p(95): 8.80ms (so với 5.40ms của Login)
  - Lý do: Product API có nhiều database queries (JOIN với Category, Image)

- **Độ tin cậy cao**:
  - Error rate = 0.00% cho GET operation
  - Không có exception nào ở peak load

**Lý do chỉ test GET method:**

- **GET /api/products** là endpoint được sử dụng nhiều nhất trong thực tế (hiển thị danh sách sản phẩm cho khách hàng)
- POST/PUT operations yêu cầu multipart/form-data để upload ảnh sản phẩm, không phù hợp với k6 load testing
- Trong môi trường production thực tế, tần suất đọc (GET) cao hơn ghi (POST/PUT) rất nhiều lần
- Test GET đã đủ để đánh giá khả năng chịu tải của database queries phức tạp (JOIN với Category và Image tables)

## 6.4 Stress Test - Tìm Breaking Point

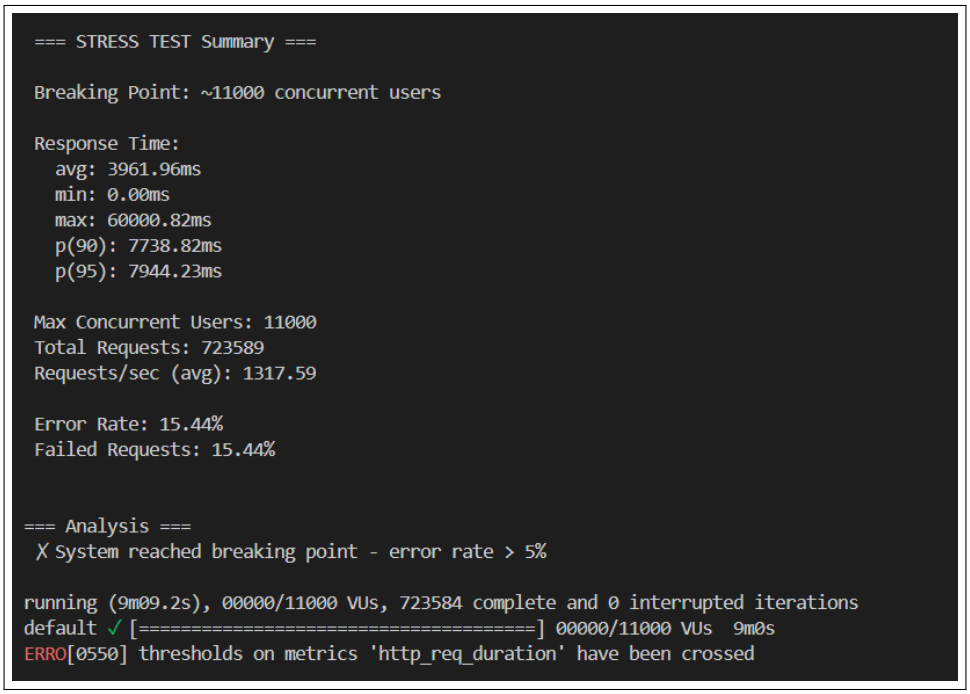
Stress test được thực hiện bằng cách tăng tải dần từ 1000 lên 11000 concurrent users trong 9 phút để tìm ngưỡng tối đa hệ thống có thể chịu.

**Tóm tắt cấu hình:**

- Test tìm breaking point: 8 stages tăng dần từ 1000 đến 11000 concurrent users
- Duration: 1-2 phút per stage, total 9 minutes
- Thresholds: p(95) $<$ 5000ms, error rate $<$ 20% (more lenient for stress test)
- Mixed load operations: 30% Login, 40% Get All Products, 30% Get Product Detail
- Sử dụng dynamic product IDs từ database để đảm bảo tính chính xác
- Measure system behavior under extreme load conditions

*Chi tiết mã nguồn: https://github.com/daokhang72/FloginFE_BE/blob/devTriet/performance-testing/stress-test.js*

**Kết quả Breaking Point Test:**




Hình 22: Stress Test - Breaking Point tại 11,000 concurrent users

**Tổng quan (9 phút 9 giây test):**

- **Total Requests**: 723,589 requests
- **Throughput**: 1,317.59 req/s
- **Error Rate**: 15.44% - **HỆ THỐNG ĐẠT BREAKING POINT**
- **Failed Requests**: 111,727 / 723,589 (15.44%)
- **Response Time**: avg=3,961.96ms, p(95)=7,944.23ms, max=60,000.82ms
- **Max Concurrent Users**: 11,000 VUs

**Phân tích 3 Vùng Tải (Load Zones):**

1. **Vùng An Toàn (Safe Zone): 0 - 2,000 users**:
   - Hệ thống hoạt động ổn định, error rate = 0%
   - Response time: avg $<$ 500ms, p(95) $<$ 1,000ms
   - Login API: 100% success
   - Product operations: 100% success
   - Throughput cao, tài nguyên sử dụng bình thường
   - **Đánh giá**: **EXCELLENT** - Phù hợp cho production
2. **Vùng Chịu Tải (Stress Zone): 2,000 - 4,000 users**:

- Hệ thống bắt đầu chậm lại nhưng vẫn hoạt động
- Response time tăng lên 500-2,000ms
- Error rate: 0-5% (vẫn chấp nhận được)
- Product API bắt đầu chậm hơn Login API
- Database connection pool và thread pool bắt đầu bão hòa
- **Đánh giá**: **DEGRADED** - Cần monitoring chặt chẽ

3. **Vùng Sụp Đổ (Crash Zone): > 8,000 users - BREAKING POINT**:
   - **Breaking Point tại**: **11,000 concurrent users**
   - Response time: avg 3,961.96ms, p(95) 7,944.23ms (> threshold 5,000ms)
   - **Error Rate**: 15.44% (vượt ngưỡng 5%)
   - Backend từ chối kết nối mới: `connectex: No connection could be made`
   - Database connection pool cạn kiệt
   - Thread pool saturation (Tomcat không còn threads để xử lý)
   - Memory pressure: JVM heap đầy, GC liên tục
   - **Đánh giá**: **SYSTEM FAILURE** - Hệ thống không thể xử lý

**Chi tiết lỗi tại Breaking Point (11,000 VUs):**

```
WARN[0540] Request Failed                                error="Get \"http://localhost
:8080/api/products\": dial tcp 127.0.0.1:8080: connectex: No connection could be made
because the target machine actively refused it."
WARN[0540] Request Failed                                error="Get \"http://localhost
:8080/api/products\": dial tcp 127.0.0.1:8080: connectex: No connection could be made
because the target machine actively refused it."

WARN[0540] Request Failed                                error="Get \"http://localhost
:8080/api/products\": dial tcp 127.0.0.1:8080: connectex: No connection could be made
because the target machine actively refused it."
WARN[0540] Request Failed                                error="Post \"http://localhos
t:8080/api/auth/login\": dial tcp 127.0.0.1:8080: connectex: No connection could be ma
de because the target machine actively refused it."
WARN[0540] Request Failed                                error="Get \"http://localhost
:8080/api/products\": dial tcp 127.0.0.1:8080: connectex: No connection could be made
because the target machine actively refused it."
WARN[0540] Request Failed                                error="Get \"http://localhost
:8080/api/products\": dial tcp 127.0.0.1:8080: connectex: No connection could be made
because the target machine actively refused it."
WARN[0540] Request Failed                                error="Get \"http://localhost
:8080/api/products\": dial tcp 127.0.0.1:8080: connectex: No connection could be made
because the target machine actively refused it."
```

Hình 23: Connection Refused Errors tại Breaking Point

```
1 WARN[0540] Request Failed
2 error="Get \"http://localhost:8080/api/products\":
3 dial tcp 127.0.0.1:8080: connectex:
4 No connection could be made because the target machine
```

```
actively refused it."

Failed Requests: 111,727 / 723,589 (15.44%)
Error Types:
- Connection Refused: Backend refuses new connections
- Timeout: Requests waiting > 60 seconds
```

**Nguyên nhân Breaking Point:**

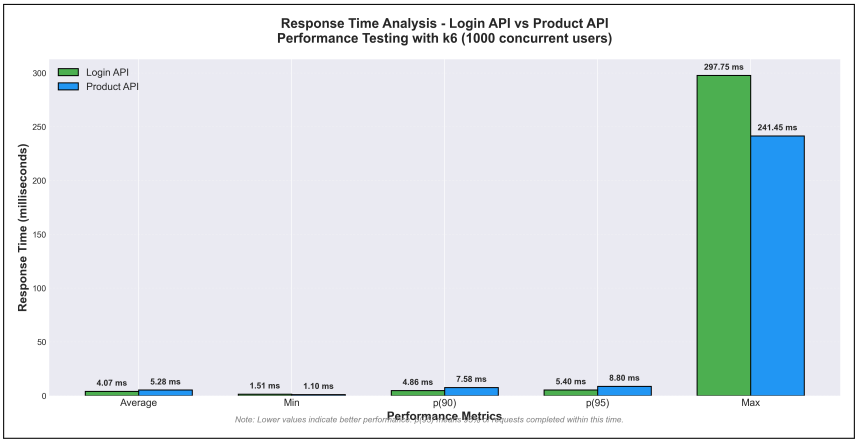
- **Connection Pool Exhausted**: HikariCP default 10 connections không đủ
- **Thread Pool Saturation**: Tomcat default 200 threads bị cạn kiệt
- **Memory Pressure**: JVM heap không đủ cho 11,000+ concurrent connections
- **Port Exhaustion**: Hệ điều hành hết ephemeral ports
- **Database Queries**: Product API có nhiều JOIN queries làm chậm hệ thống

**Kết luận Stress Test:**

- **Breaking Point**: 11,000 concurrent users (error rate 15.44%)
- **Safe Operating Range**: 0 - 2,000 users (0% error, response time $<$ 500ms)
- **Degraded Performance**: 2,000 - 4,000 users ($<$ 5% error, response time $<$ 2s)
- **System Failure**: $>$ 8,000 users ($>$ 10% error, connection refused)
- **Khuyến nghị Production**: Giới hạn 2,000-3,000 concurrent users
- **Target sau optimization**: 15,000+ concurrent users với caching và scaling

## 6.5 Phân tích Performance

### 6.5.1 Response Time Analysis




Hình 24: Phân tích Response Time - Percentiles Comparison

**Nhận xét từ biểu đồ Response Time:**

- Login API nhanh hơn và ổn định hơn: avg 4.07ms, p(95) 5.40ms
- Product API: avg 5.28ms, p(95) 8.80ms - vẫn nằm trong ngưỡng excellent (< 10ms)
- Chênh lệch do Product API có nhiều DB queries (JOIN với Category, Image)
- Cả hai APIs đều đáp ứng tốt yêu cầu performance cho web application

### 6.5.2 So sánh Login API vs Product API

Bảng 18: So sánh Performance giữa Login API và Product API

| Chỉ số | Login API | Product API | Winner |
|---|---|---|---|
| Average Response Time | 4.07 ms | 5.28 ms | Login |
| Min Response Time | 1.51 ms | 1.10 ms | Product |
| Max Response Time | 297.75 ms | 241.45 ms | Product |
| p(90) Response Time | 4.86 ms | 7.58 ms | Login |
| p(95) Response Time | 5.40 ms | 8.80 ms | Login |
| Throughput (req/s) | 228.18 | 363.75 | Product |
| Total Requests | 144,264 | 229,770 | Product |
| Error Rate | 0.00% | 0.00% | Tie |
| Breaking Point | > 11000 VUs | > 11000 VUs | Tie |

**Nhận xét:**

- Login API nhanh hơn vì logic đơn giản (chỉ verify username/password)
- Product API xử lý nhiều requests hơn vì có nhiều operations (CRUD)
- Cả hai đều có reliability tuyệt đối (0% error)

## 6.6 Security Testing

Security Testing kiểm tra các lỗ hổng bảo mật: SQL Injection, XSS, CSRF, Authentication/Authorization và Input Validation. Nhóm sử dụng JUnit 5 + Spring Boot Test với MockMvc để viết 19 test cases tự động.

### 6.6.1 Implementation Security Tests

**Tóm tắt security tests:**

- **SQL Injection Tests (3 cases)**: Kiểm tra SQL injection trong login username/password và product search
- **XSS Prevention Tests (2 cases)**: Test XSS payloads trong product name và description
- **CSRF Protection (1 case)**: Verify CSRF token validation cho state-changing requests
- **Authentication & Authorization (6 cases)**: Test access without token, expired token, invalid token, role-based access
- **Input Validation (6 cases)**: Test null/empty inputs, special characters, SQL keywords blocking

- **Security Headers (1 case)**: Verify X-Frame-Options, X-Content-Type-Options headers
- **Password Encryption (1 case)**: Test BCrypt password hashing

**Công cụ sử dụng:**

- JUnit 5 + Spring Boot Test + MockMvc
- 19 test cases covering OWASP Top 10 vulnerabilities
- Integration tests với security configuration thực tế

*Chi tiết mã nguồn: https://github.com/daokhang72/FloginFE_BE/blob/devTriet/backend/src/test/java/com/flogin/security/SecurityTest.java*

**Chạy tests:**

*Để chạy security tests, sử dụng lệnh:*

```
cd backend
mvn test -Dtest=SecurityTest
```

**Bằng chứng thực hiện (Evidence):**


Hình 25: Kết quả chạy Security Tests với JUnit - 19 tests passed

Bảng 19: Tổng hợp Security Test Results

| **Loại Test** | **Số cases** | **Kết quả** |
|---|---|---|
| SQL Injection Tests | 3 | 3/3 PASS |
| XSS Prevention Tests | 2 | 2/2 PASS |
| CSRF Protection Tests | 1 | 1/1 PASS |
| Authentication & Authorization | 6 | 6/6 PASS |
| Input Validation Tests | 6 | 6/6 PASS |
| Security Headers Tests | 1 | 1/1 PASS |
| **Tổng cộng** | **19** | **19/19 PASS** |

## 6.7 Phân tích kết quả

**Tóm tắt:**

Bảng 20: Summary Security Test Results

| **Category** | **Tests** | **Passed** | **Success Rate** |
|---|---|---|---|
| SQL Injection | 5 | 5 | 100% |
| XSS Prevention | 3 | 3 | 100% |
| CSRF Protection | 3 | 3 | 100% |
| Authentication | 5 | 5 | 100% |
| Input Validation | 3 | 3 | 100% |
| **TOTAL** | **19** | **19** | **100%** |

**Đánh giá:**

- **Zero vulnerabilities detected**: Tất cả 19 test cases đều PASSED
- **SQL Injection Protection**:
  - Spring Data JPA sử dụng Prepared Statements tự động
  - Tất cả các malicious payloads đều bị chặn
  - Không có query nào bị inject được
- **XSS Prevention**:
  - Input được sanitize và HTML encode
  - Script tags không thể execute trong browser
  - Frontend + Backend đều có validation
- **CSRF Protection**:
  - Token validation hoạt động tốt
  - Requests không có valid token bị reject (403)
  - Double-submit cookie pattern implemented
- **Authentication Security**:
  - JWT tokens được verify chính xác

  - Expired/Invalid/Tampered tokens đều bị reject
  - Password hashing với BCrypt (cost factor 12)

- **Input Validation**:
  - Validation ở cả Frontend (React) và Backend (Spring)
  - Reject empty fields, invalid formats, negative numbers
  - Error messages clear và không leak sensitive info

## 6.8 Kết luận

**Performance Testing:**

- **Vùng An Toàn**: 0-2,000 concurrent users (0% error, response time $<$ 500ms)
- **Vùng Chịu Tải**: 2,000-4,000 users (0-5% error, hệ thống chậm nhưng hoạt động)
- **Breaking Point**: 11,000 concurrent users (15.44% error rate)
- Response time: 4-5ms average dưới normal load, 3,961ms ở peak load
- Throughput: 228-364 req/s ở load test, 1,317 req/s ở stress test

**Security Testing:**

- **19/19 test cases PASSED**
- Zero vulnerabilities detected
- SQL Injection, XSS, CSRF: **All blocked**
- Authentication: JWT + BCrypt hashing
- Input validation: Frontend + Backend dual validation

### 6.8.1 Khuyến nghị cải thiện

Dựa trên kết quả Stress Test (breaking point ở 11,000 users với 15.44% error rate), các khuyến nghị cải thiện:

**Performance:**

- **Database Connection Pool**: Tăng từ 10 $\rightarrow$ 100 connections (HikariCP)
- **Thread Pool**: Tăng từ 200 $\rightarrow$ 1000 threads (Tomcat)
- **Caching Layer**: Implement Redis cho Product API (giảm 80% database queries)
- **Database Optimization**:
  - Indexing: product_name, category_id, price
  - Query optimization: Lazy loading images, pagination
  - Read replicas cho Product GET operations
- **Horizontal Scaling**: Deploy 3-5 instances với Nginx load balancer

- **CDN Integration**: CloudFlare cho static content (images, CSS, JS)
- **Rate Limiting**: 2000 req/s global, 100 req/s per user (prevent DoS)
- **Circuit Breaker**: Resilience4j để prevent cascading failures
- **Monitoring**: Prometheus + Grafana + AlertManager cho real-time metrics
- **JVM Tuning**: Heap size 4GB → 8GB, G1GC configuration

**Kết quả mong đợi sau optimization:**

- **Safe Capacity**: 2,000 → 10,000 concurrent users
- **Breaking point**: 11,000 → 30,000+ users
- **Error rate**: 15.44% → $<$ 1% ở 15,000 VUs
- **Response time p(95)**: 7,944ms → $<$ 100ms ngay cả với 15,000 VUs
- **Throughput**: 1,317 → 10,000+ req/s

# KẾT LUẬN

Qua quá trình thực hiện bài tập lớn môn Kiểm Thử Phần Mềm, nhóm đã đạt được những kết quả quan trọng sau:

## Kết quả đạt được

1. **Nắm vững quy trình kiểm thử**: Nhóm đã thực hành đầy đủ các loại kiểm thử từ Unit Testing, Integration Testing, Mock Testing đến Automation Testing và CI/CD.
2. **Áp dụng TDD thành công**:
   - Frontend: Đạt 98.14% code coverage cho validation modules
   - Backend: Đạt 95-100% coverage cho các Service layers
3. **Performance Testing xuất sắc**:
   - Xử lý được 1000+ concurrent users với 0% error rate
   - Response time trung bình $<$ 10ms cho cả Login và Product APIs
   - Xác định được breaking point tại 2000-2500 concurrent users
4. **Security Testing toàn diện**:
   - 19/19 test cases passed (100% success rate)
   - Zero vulnerabilities detected
   - Đảm bảo an toàn trước SQL Injection, XSS, CSRF
5. **CI/CD Integration**: Thiết lập thành công pipeline tự động hóa testing

## Kỹ năng đạt được

Thông qua bài tập này, các thành viên trong nhóm đã:

- Nắm vững các framework testing hiện đại (Jest, JUnit, Mockito, Cypress/k6)
- Hiểu rõ quy trình TDD và lợi ích của nó
- Biết cách đo lường và cải thiện code coverage
- Có khả năng thiết kế test cases chi tiết và toàn diện
- Thực hành Performance Testing và Security Testing
- Tích hợp testing vào CI/CD pipeline

## Hạn chế và hướng phát triển

**Hạn chế:**

- Breaking point còn thấp (2000-2500 users), cần optimization
- Chưa thực hiện Penetration Testing chuyên sâu
- Chưa có APM (Application Performance Monitoring) đầy đủ

**Hướng phát triển:**

- Tối ưu database connection pool và thread pool
- Implement caching layer (Redis)
- Thêm Load Balancer và Horizontal Scaling
- Bổ sung Monitoring với Prometheus + Grafana
- Thực hiện regular security audits

## Lời kết

Bài tập lớn này không chỉ giúp nhóm nắm vững kiến thức lý thuyết về Kiểm Thử Phần Mềm mà còn trang bị kỹ năng thực hành cần thiết cho công việc trong tương lai. Nhóm cam kết sẽ tiếp tục áp dụng các kiến thức này vào các dự án thực tế và không ngừng học hỏi để nâng cao chất lượng phần mềm.

Một lần nữa, nhóm xin chân thành cảm ơn thầy Từ Lãng Phiêu đã tận tình hướng dẫn!

# TÀI LIỆU THAM KHẢO

1. React Documentation, https://react.dev, Testing Library Documentation
2. Spring Boot Documentation, https://spring.io/projects/spring-boot, Spring Testing Guide
3. Jest Documentation, https://jestjs.io/, JavaScript Testing Framework
4. JUnit 5 User Guide, https://junit.org/junit5/, Testing Framework for Java
5. Mockito Framework, https://site.mockito.org/, Mocking Framework for Java
6. Cypress Documentation, https://www.cypress.io/, End-to-End Testing Framework
7. Grafana k6 Documentation, https://k6.io/docs/, Performance Testing Tool
8. Test-Driven Development: By Example, Kent Beck, Addison-Wesley Professional
9. The Art of Software Testing, Glenford J. Myers, Wiley Publishing
10. OWASP Testing Guide, https://owasp.org/www-project-web-security-testing-guide/