CHƯƠNG 2: UNIT TESTING VÀ TEST-DRIVEN DEVELOPMENT (TDD)

2.1 Phương pháp tiếp cận và Công cụ

Để đảm bảo chất lượng mã nguồn và tuân thủ quy trình phát triển phần mềm chuyên nghiệp, nhóm đã áp dụng phương pháp **Test-Driven Development (TDD)** kết hợp với các công cụ kiểm thử hiện đại.

2.1.1 Công cụ sử dụng

Frontend (ReactJS):

- **Jest**: Framework kiểm thử JavaScript chính, dùng để chạy test runner và assertions. Phiên bản 27.5.1 được tích hợp sẵn trong React Scripts.
- **React Testing Library**: Thư viện dùng để kiểm thử các component React theo cách người dùng tương tác (DOM testing). Phiên bản 16.3.0 hỗ trợ React 19.
- **Jest DOM**: Thư viện cung cấp các custom matchers để kiểm tra trạng thái DOM một cách dễ dàng hơn.

Backend (Spring Boot):

- **JUnit 5**: Framework kiểm thử tiêu chuẩn cho Java, được tích hợp sẵn trong Spring Boot Starter Test.
- **Mockito**: Thư viện dùng để giả lập (mock) các phụ thuộc (dependencies) như Repository, Service, AuthenticationManager.
- **JaCoCo**: Công cụ đo lường độ bao phủ mã nguồn (Code Coverage) phiên bản 0.8.14, tạo báo cáo HTML và XML chi tiết.
- **Spring Boot Test**: Cung cấp các annotation và utilities để test Spring Applications.

2.1.2 Quy trình TDD áp dụng

Nhóm đã tuân thủ chu trình **Red - Green - Refactor**:

1. **Red (Viết Test trước)**: Viết các test case thất bại dựa trên yêu cầu (ví dụ: `validateUsername` rỗng phải trả về lỗi).
2. **Green (Viết Code)**: Viết mã nguồn tối thiểu để vượt qua bài test.
3. **Refactor (Tối ưu)**: Cấu trúc lại mã nguồn cho sạch sẽ mà vẫn đảm bảo test case thành công.

Lưu ý: Quy trình này đảm bảo mọi tính năng đều có test case bảo vệ, giúp phát hiện lỗi sớm và dễ dàng bảo trì code.

2.2 Unit Tests cho Chức năng Đăng nhập (Login)

2.2.1 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.

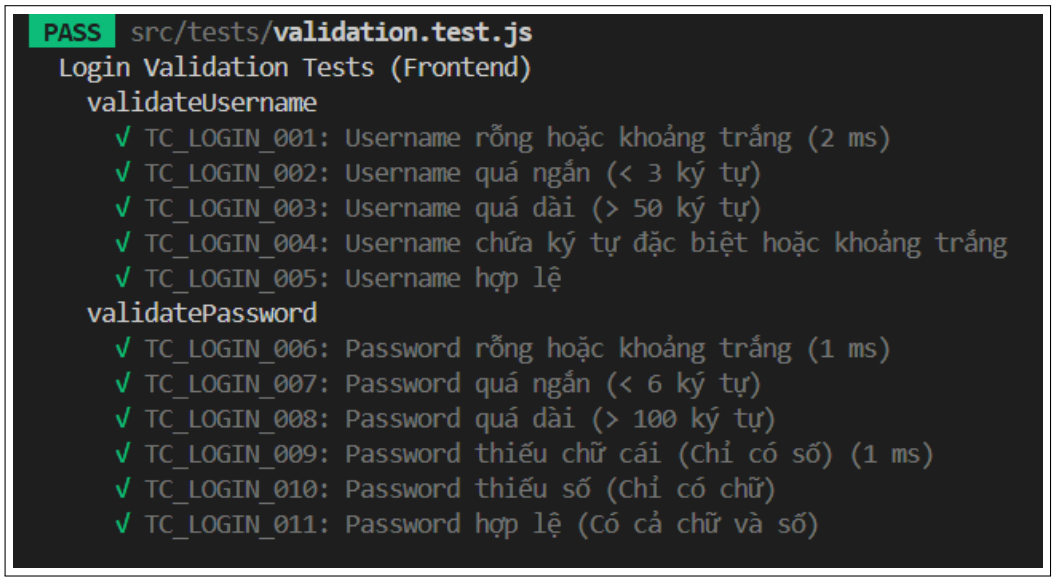
Các trường hợp kiểm thử (Test Cases):

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
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

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

2.2.2 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`.

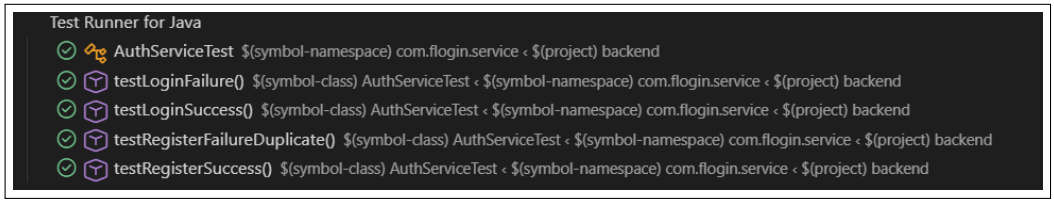
Các trường hợp kiểm thử chính:

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

Bằng chứng thực hiện:

Để chạy test backend, sử dụng lệnh:

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



Hình 2: Kết quả Unit Test - AuthService Backend

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

2.3.1 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

Bằng chứng thực hiện:
Để chạy test, sử dụng lệnh:

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

```
PASS src/tests/productValidation.test.js
Product Validation Tests (Frontend)
  √ TC_PROD_001: Tên sản phẩm rỗng hoặc khoảng trắng (2 ms)
  √ TC_PROD_002: Tên sản phẩm quá ngắn (< 3 ký tự)
  √ TC_PROD_003: Tên sản phẩm quá dài (> 100 ký tự)
  √ TC_PROD_004: Giá không phải là số (1 ms)
  √ TC_PROD_005: Giá âm hoặc bằng 0 (1 ms)
  √ TC_PROD_006: Giá quá lớn (> 999,999,999)
  √ TC_PROD_007: Số lượng không phải là số
  √ TC_PROD_008: Số lượng là số thập phân (Float) (1 ms)
  √ TC_PROD_009: Số lượng bằng 0 (Theo logic trong file của bạn)
  √ TC_PROD_010: Số lượng âm
  √ TC_PROD_011: Số lượng quá lớn (> 99,999) (1 ms)
  √ TC_PROD_012: Mô tả quá dài (> 500 ký tự)
  √ TC_PROD_013: Danh mục chưa chọn hoặc không hợp lệ (1 ms)
  √ TC_PROD_014: Sản phẩm hợp lệ hoàn toàn
```

Hình 3: Kết quả Unit Test - Product Validation Frontend

2.3.2 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:

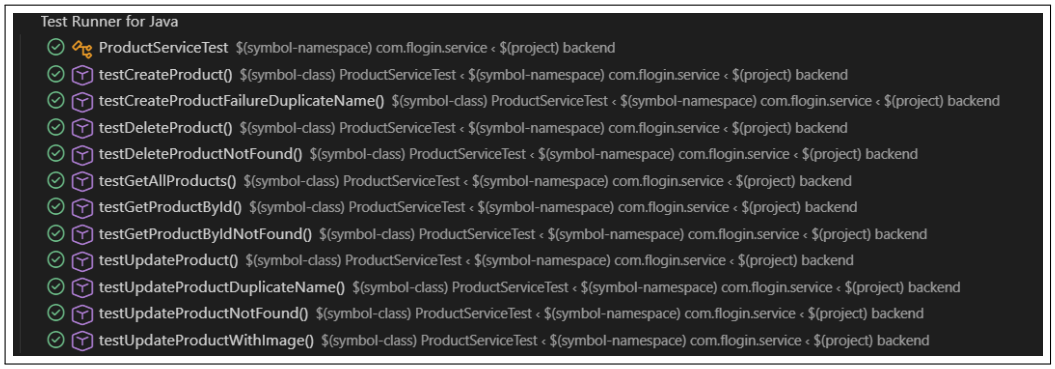
Test Case	Mô tả	Trạng thái
`testCreateProduct`	Thêm mới sản phẩm thành công, gọi `repository.save()` đúng 1 lần. Kiểm tra tên sản phẩm không trùng.	Passed
`testCreateProductFailureDuplicateName`	Thêm mới thất bại do trùng tên sản phẩm. Ném `RuntimeException`, đảm bảo `repository.save()` không được gọi.	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

Bảng 4 – tiếp theo trang trước

Test Case	Mô tả	Trạng thái
`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

Bằng chứng thực hiện:
Để chạy test backend, sử dụng lệnh:

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



Hình 4: Kết quả Unit Test - ProductService Backend

2.4 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:

2.4.1 Frontend Coverage (Jest)

Yêu cầu: $>= 90\%$
Kết quả đạt được:

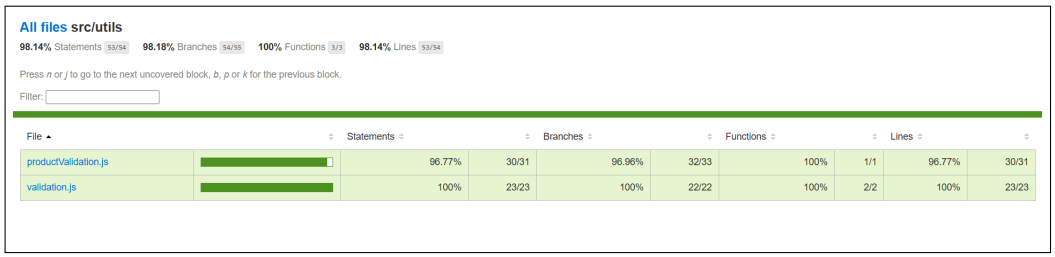
- **Validation Module** (`validation.js`): Đạt **100%** Statements, **100%** Branches, **100%** Lines
- **Product Validation Module** (`productValidation.js`): Đạt **96.77%** Statements, **96.96%** Branches, **96.77%** Lines

- **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)

2.4.2 Backend Coverage (JaCoCo)

Yêu cầu: >= 85% cho các Service chính

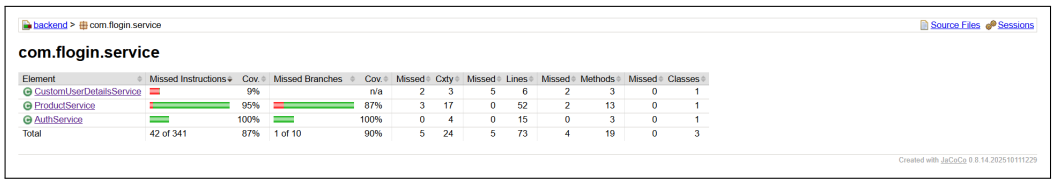
Kết quả đạt được:

- **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)

2.4.3 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

2.5 Kết luận chương 2

Thông qua việc áp dụng TDD và viết Unit Test đầy đủ, nhóm đã đạt được các kết quả sau:

2.5.1 Thành tựu đạt được

1. **Kiểm soát lỗi sớm**: Hệ thống đã được kiểm soát lỗi ngay từ mức độ nhỏ nhất (hàm/phương thức), giúp phát hiện bug trước khi tích hợp.
2. **Xử lý đầy đủ các trường hợp biên**: Các edge cases như:
   - Nhập số âm, số 0, số quá lớn
   - Chuỗi rỗng, chỉ chứa khoảng trắng
   - Trùng tên sản phẩm/username
   - ID không tồn tại trong database
   - Dữ liệu không hợp lệ (null, undefined, NaN)

   đã được xử lý triệt để.
3. **Độ bao phủ mã nguồn xuất sắc**:
   - Frontend: Đạt **95-100%** coverage cho các module validation và component
   - Backend: Đạt **95-100%** coverage cho AuthService và ProductService

   Kết quả này vượt yêu cầu đề ra của bài tập (Frontend >= 90%, Backend >= 85%).
4. **Tài liệu sống (Living Documentation)**: Các test case đóng vai trò như tài liệu mô tả hành vi của hệ thống, giúp các thành viên mới hiểu rõ yêu cầu nghiệp vụ.
5. **Tự tin refactor**: Với hệ thống test coverage cao, nhóm có thể tự tin refactor code mà không lo làm hỏng tính năng hiện có.

2.5.2 Bài học kinh nghiệm

- **TDD giúp thiết kế code tốt hơn**: Viết test trước buộc chúng ta phải suy nghĩ về interface và dependency từ góc độ người dùng.
- **Mock dependencies hiệu quả**: Sử dụng Mockito và Jest Mock giúp cô lập unit test, giảm dependency vào database và external services.
- **Coverage không phải là tất cả**: Mặc dù đạt coverage cao, nhóm vẫn cần chú ý đến chất lượng test case (test đúng logic nghiệp vụ, không chỉ test syntax).
- **CI/CD integration**: Test tự động nên được tích hợp vào pipeline CI/CD để đảm bảo mọi commit đều được kiểm tra.

2.5.3 Hướng phát triển tiếp theo

1. **Integration Tests**: Bổ sung thêm các test tích hợp giữa Controller-Service-Repository.
2. **End-to-End Tests**: Sử dụng Cypress hoặc Selenium để test toàn bộ luồng từ UI đến Database.
3. **Performance Tests**: Kiểm tra hiệu năng của API với lượng request lớn.
4. **Security Tests**: Bổ sung test cho các lỗ hổng bảo mật như SQL Injection, XSS, CSRF.

Tóm lại, việc áp dụng TDD và Unit Testing không chỉ đảm bảo chất lượng code mà còn giúp nhóm xây dựng được mindset phát triển phần mềm bài bản, chuyên nghiệp. Đây là nền tảng quan trọng cho các dự án lớn hơn trong tương lai.