ShopHub: Java E-Commerce System

- CS5004 Final Project - Fall 2025
- Patrick - Northeastern University

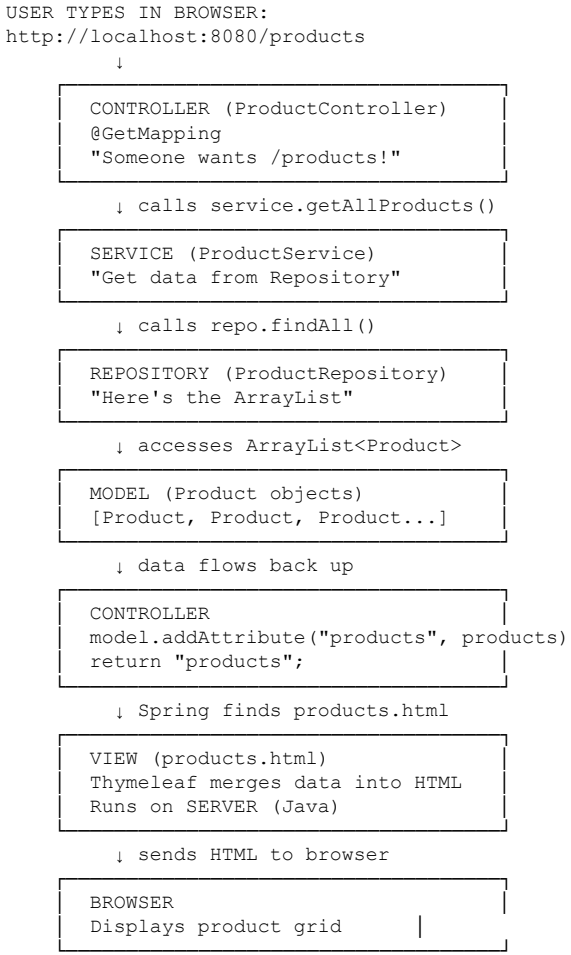
Project Goals:

- Build production-quality e-commerce system using Java
- Demonstrate OOP principles and design patterns
- Implement comprehensive testing and clean architecture
- Create intuitive, professional user interface


$1 FOR 2
CORN FLAKES
EKONO
WHITE RICE
$1 for 2
BICKIES
CHIEF
NEW FLAVOURED BLENDS
DANGER: SMOKING IS HARMFUL TO HEALTH
HYGIENIX

Key Tools and Methods

- **Technology Stack:**
-
- Language: Java 17
- Framework: Spring Boot 3.3.2
- Template Engine: Thymeleaf (Server-side Java)
- Testing: JUnit 5 with 36+ test methods
- Build Tool: Maven
- UI: HTML5 + CSS3



Code Walkthrough: Auto-ID Generation

- Problem: Users manually entered IDs → Duplicate errors
- Solution: System auto-generates unique, sequential IDs

```
// Counter maintains next available ID
private int nextId = 1;


private int getNextId() {
  return nextId++;  // Return current, then increment
}


public Product save(String name, double price, ...) {
  int id = getNextId();  // Auto-generate ID
  Product p = new Product(id, name, price, ...);
  products.add(p);
```

Object-Oriented Design Evidence

1. MVC Architecture

Clear separation:

Model-View-Controller

2. Stream API & Lambdas

Modern Java functional programming

3. DRY Principle

Don't Repeat Yourself

Reusable methods

Test Coverage:

36+ JUnit tests

```
// MVC Pattern
@Controller  // Web layer
public class ProductController {
  private final ProductService service;
}

// Stream API & Lambda
public Product findById(int id) {
  return products.stream()
    .filter(p -> p.getId() == id)
    .findFirst()
    .orElse(null);
}

// DRY - Reusable search
public List<Product> searchByName(
    String query) {
  return products.stream()
    .filter(p -> p.getName()
      .contains(query))
    .collect(Collectors.toList());
}
```

Lessons Learned

What Was Hard:

- Stream API mastery - learning to chain operations effectively
- Layered architecture - understanding when logic belongs where
- Thymeleaf syntax - learning server-side templating

What Was "Less Difficult":

- Spring Boot setup - starter dependencies handled configuration
- Testing with JUnit - clear patterns emerged
- Dependency Injection - Spring's approach is intuitive

Key Insight: Proper architecture makes code easy to test and modify

Limitations & Future Extensions

- **Current Limitations:**
-
- No data persistence (in-memory only)
- No user authentication
- No payment processing

Future Extensions:

- Add database
- Implement shopping cart system
- Add Spring Security authentication
- Integrate payment gateway (Stripe)
- Deploy to cloud (AWS/Azure/Heroku)

Resources and Citations

- Spring Boot Documentation: spring.io/projects/spring-boot
- Thymeleaf Official Docs: thymeleaf.org/documentation.html
- JUnit 5 User Guide: junit.org/junit5/docs
- Java SE 17 Documentation: docs.oracle.com/en/java/
- Baeldung Spring Boot Tutorials: baeldung.com/spring-boot
- Maven Documentation: maven.apache.org/guides/

AI Assistance:

- Tool: Claude by Anthropic (Claude 3.5 Sonnet)
- Usage: Architecture advice, code review, testing patterns
- Percentage: <20% of code lines
- *Note: All core business logic written by student*

Appendix

ShopHub:
Model = Product.java, Category.java: What IS a product? (data structure)
View = templates: How products LOOK on screen
***Service** = ProductService.java (Business logic between Controller and data)*
Controller = ProductController.java: Handles user clicks and requests

Step 1: Browser → Controller
User presses Enter in browser
Browser sends: GET request to /products

```
@Controller
@GetMapping  // ← This catches the request!
public String listProducts(Model model) {
  // Controller receives the request
```

➔ Controller says: "Someone wants to see products!"

Step 2: Controller → Service
Controller asks Service for data

```
  List<Product> products = service.getAllProducts();
  // ↑ Controller calls Service
```

➔ Controller says: "Service, get me all products please!"

Step 3: Service → Repository
Service asks Repository for data

```
@Service
public List<Product> getAllProducts() {
  return repo.findAll();
  // ↑ Service calls Repository
}
```

➔ Service says: "Repository, give me all products from storage!"

Step 4: Repository → Model
Repository accesses the ArrayList

```
@Repository
public List<Product> findAll() {
  return products;  // ArrayList of Product objects
  // ↑ Returns actual Product objects (Model!)
}
```

➔ Repository says: "Here are 4 Product objects from my ArrayList!"