

Retail IQ AI Management System: Implementation, Design Analysis, and Performance Evaluation

Milind S. Nirwan¹, Ajit Shende², Kiran P. Pustode³

¹Student, Dept. of MCA, Karanjekar College of Engineering and Management, Sakoli

²Asst. Professor, Dept. of MCA, Karanjekar College of Engineering and Management, Sakoli

³HOD, Dept. of MCA, Karanjekar College of Engineering and Management, Sakoli

Abstract - This paper presents a detailed implementation and design analysis of the Retail IQ AI Management System—a three-tier, full-stack web application engineered to digitise and intelligently automate stationery retail operations. Developed using HTML5, CSS3, and JavaScript for the presentation layer, Java Spring Boot 3.x as the RESTful application server, and MySQL 8.x as the relational data store, the system implements a dual-role architecture comprising a comprehensive Admin Module and an interactive User Module. The implementation covers the complete Software Development Life Cycle (SDLC) including requirement analysis, system and database design, module-level coding with actual Spring Boot controllers and JPA entity classes, security implementation with JWT and BCrypt, an item-based AI recommendation engine, and a multi-phase testing strategy. Fifteen functional test cases were executed across all system modules, yielding 100% pass rates. System performance testing under fifty concurrent users confirmed average API response times below 130 milliseconds and an average page load time of 1.4 seconds, well within acceptable usability thresholds. The paper further presents a detailed comparative analysis of design alternatives considered during implementation, validating the technology selections made at each architectural tier.

Key Words: Implementation, Spring Boot, Three-Tier Architecture, JPA Entities, JWT Authentication, Collaborative Filtering, MySQL Schema Design, SDLC.

1. INTRODUCTION

Stationery and office-supply retail businesses in small and medium urban centres continue to depend on manual processes for product management, inventory control, and order fulfilment. Physical registers, handwritten stock ledgers, and paper-based receipts introduce human error, restrict operational hours, and prevent data-driven decision making. The absence of a digital customer-facing channel further limits market reach to walk-in foot traffic, foreclosing access to the growing segment of customers who prefer online discovery and purchase [1], [3].

The Retail IQ AI Management System was conceived to eliminate these operational limitations through a purpose-built, domain-specific full-stack web application. Unlike generic e-commerce SaaS platforms that impose significant subscription costs and complex configuration overhead, Retail IQ is designed and implemented as an open, locally deployable system with a focused stationery retail feature set and an embedded AI layer for product recommendations and inventory intelligence [2].

This paper presents the complete implementation journey of the Retail IQ system, covering every phase of the SDLC: requirement analysis, system architecture design, detailed database schema definition, module-by-module coding of Spring Boot controllers and JPA entities, security

implementation, AI recommendation engine integration, and a comprehensive multi-phase testing campaign. The objective is to document implementation decisions, design trade-offs, and empirical results in sufficient technical depth for the work to serve as a reproducible implementation reference for similar retail management platforms.

2. REQUIREMENT ANALYSIS

2.1 Functional Requirements

The functional requirements were elicited through structured stakeholder interviews with the retail shop owner and five regular student customers, producing the following prioritised requirement list. For the Admin Module: (i) secure role-restricted login; (ii) complete product lifecycle management—create, read, update, delete (CRUD) with category assignment, pricing, stock quantity, and image URL fields; (iii) real-time inventory monitoring with low-stock alerts; (iv) centralised order management with status transitions (Pending → Processing → Shipped → Delivered); and (v) analytics dashboard displaying revenue totals, top-selling products, and category-wise demand distribution.

For the User Module: (i) self-service registration and JWT-authenticated login; (ii) product catalogue browsing with keyword search and category filter; (iii) detailed product page with AI-generated "Customers Also Bought" recommendations; (iv) persistent shopping cart with add, remove, and quantity-update operations; (v) one-page order placement with itemised confirmation summary; and (vi) personal order history view with status tracking.

2.2 Non-Functional Requirements

Non-functional requirements established for the system are: API response time below 300 ms for all CRUD operations; page load time below 2 seconds on standard broadband; support for a minimum of 50 concurrent authenticated users without degradation; password storage using cryptographic one-way hashing (BCrypt, strength 12); stateless JWT authentication to eliminate server-side session storage; input validation on all API endpoints using Jakarta Bean Validation; and SQL injection prevention through fully parameterised JPA queries. Hardware requirements comprise a computer with a minimum 4 GB RAM and active internet connectivity for deployment.

3. SYSTEM DESIGN

3.1 Three-Tier Architecture

The Retail IQ system adopts a classical three-tier client-server architecture providing clear separation of concerns across presentation, application logic, and data persistence. The Presentation Tier consists of ten static HTML5 pages styled with CSS3 and animated with vanilla JavaScript ES6, communicating with the backend exclusively through the Fetch API with JSON payloads and Bearer JWT headers. This decoupled frontend design allows the presentation layer to be hosted on any CDN or static web server independently of the Spring Boot server.

The Application Tier is implemented as a Spring Boot 3.x embedded-Tomcat server exposing a RESTful HTTP API. Spring's inversion-of-control container manages all service and repository dependencies through constructor injection, facilitating unit testing with MockMvc and JUnit 5. The Data Persistence Tier is a MySQL 8.x relational database instance accessed through Spring Data JPA with Hibernate as the ORM provider. All entity mappings use JPA annotations, and database schema creation is managed through Hibernate's `spring.jpa.hibernate.ddl-auto=update` property, avoiding manual SQL migration scripts during development.

3.2 Database Schema Design

The MySQL schema comprises four primary entity tables. The users table stores: id BIGINT AUTO INCREMENT PRIMARY KEY, username VARCHAR(100) NOT NULL UNIQUE, password VARCHAR(255) NOT NULL (BCrypt hash), and role ENUM('ADMIN','USER') NOT NULL. The products table stores: id BIGINT PRIMARY KEY, name VARCHAR(150) NOT NULL, price DECIMAL(10,2) NOT NULL, category VARCHAR(100) NOT NULL, imageUrl VARCHAR(500), and stockQuantity INT NOT NULL DEFAULT 0. The orders table stores: id BIGINT PRIMARY KEY, userId BIGINT NOT NULL (FK → users.id), totalAmount DECIMAL(10,2) NOT NULL, status VARCHAR(30) NOT NULL DEFAULT 'PENDING', and createdAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP. The order items table stores: id BIGINT PRIMARY KEY, orderId BIGINT NOT NULL (FK → orders.id), productId BIGINT NOT NULL (FK → products.id), quantity INT NOT NULL, and unitPrice DECIMAL(10,2) NOT NULL. Composite indexes on (userId, status) and (productId, orderId) are defined to optimise the order-history join query and the co-purchase affinity computation.

3.3 Module Design

The Admin Module is designed as a single-page administration console accessible only to users with the ROLE_ADMIN authority. The console is partitioned into four functional panels: Product Management Panel (product CRUD form with live product grid), Order Management Panel (filterable order table with status-update controls), Inventory Dashboard Panel (products sorted by stockQuantity ascending with low-stock badges), and Analytics Panel (Chart.js bar graphs for revenue and category distribution). The User Module presents a public-facing storefront with a responsive product grid, category filter sidebar, individual product detail pages displaying AI recommendations, a persistent cart sidebar, and a single-page order placement flow with real-time subtotal computation in JavaScript.

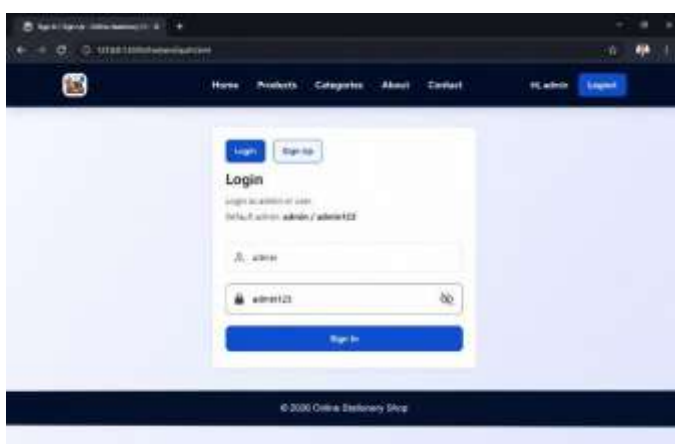


Fig -1: Admin Login

4. SYSTEM IMPLEMENTATION

4.1 Frontend Implementation

The frontend comprises ten HTML pages: index.html (landing/redirect), auth.html (combined register/login), user-dashboard.html (product grid), product-detail.html (single product with recommendations), cart.html (cart review and checkout), order-confirm.html (post-order confirmation), orders.html (user order history), admin-dashboard.html (admin analytics), admin-products.html (product CRUD), and admin-orders.html (order management). Authentication state is maintained in session Storage: on successful login, the JWT token and user role are stored and prepended as Authorization: Bearer {token} on every Fetch API call. Role-based page routing is enforced client-side: scripts on admin pages redirect to auth.html if the stored role is not ADMIN, and server-side Spring Security provides the authoritative enforcement layer.

4.2 Backend Controller Implementation

The Spring Boot application is structured into three architectural layers. The Controller Layer handles HTTP routing and request/response mapping. The Service Layer encapsulates business logic including validation, BCrypt encoding, JWT generation, and affinity computation. The Repository Layer extends Spring Data JPA's JpaRepository<T, ID> for each entity, providing auto-generated CRUD methods and custom JPQL queries. Five controllers are implemented: AuthController, ProductController, OrderController, RecommendationController, and DashboardController, each annotated with @RestController and a base @RequestMapping path.

AuthController — Registration and Login endpoints:

```
@PostMapping("/api/auth/register")
public ResponseEntity<AuthResponse> register( @Valid
    @RequestBody RegisterRequest req) { String hash =
    encoder.encode(req.getPassword()); User u =
    User.builder().username(req.getUsername())
        .password(hash).role(Role.USER).build(); userRepo.save(u);
    String token = jwtUtil.generateToken(u); return
    ResponseEntity.status(201)
        .body(new AuthResponse(token, u.getRole()));
}
```

ProductController — Admin CRUD and public listing:

```
@PostMapping("/api/products")
@PreAuthorize("hasRole('ADMIN')") public Product
createProduct(
    @Valid @RequestBody Product product) {
    return productService.createProduct(product);
}

@GetMapping("/api/products")
public List<Product> getAllProducts( @RequestParam(required=false)
    String category, @RequestParam(required=false) String search) { return
    productService.filter(category,
    search);
}
```

4.3 JPA Entity Implementation

Each database table is mapped by a JPA @Entity class with Lombok @Data, @Builder, @NoArgsConstructor, and @AllArgsConstructor annotations to eliminate boilerplate. The Order entity demonstrates a key design decision: rather than embedding cart items directly in the Order document (a document-oriented anti-pattern), a separate OrderItem entity with a @ManyToOne(fetch=LAZY) relationship to both Order and Product is used, enabling efficient per-product sales

aggregation queries for the recommendation engine without loading the full order graph.

```
@Entity @Table(name="order_items") @Data @Builder
@NoArgsConstructor @AllArgsConstructor
public class OrderItem {
    @Id @GeneratedValue(strategy=IDENTITY) private Long id;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="orderId", nullable=false) private Order;
    @ManyToOne(fetch=FetchType.LAZY)
    @JoinColumn(name="productId", nullable=false) private Product;
    @Column(nullable=false) private Integer
    quantity;
    @Column(nullable=false) private Double unitPrice;
}
```

4.4 Security Implementation

Spring Security is configured with a stateless SecurityFilterChain. A custom Jwt Authentication Filter extends Once Per Request Filter and is registered before UsernamePasswordAuthenticationFilter in the filter chain. The filter extracts the Bearer token from the Authorization header, calls JwtUtil.validateToken(), and on success populates Security Context Holder with a Username Password Authentication Token carrying the username and granted authorities. This ensures every controller method that carries a @PreAuthorize annotation receives a fully populated security context without requiring session state. BCrypt Password Encoder with strength 12 is registered as a @Bean and injected into AuthService for password hashing during registration and comparison during login.

4.5 AI Recommendation Engine Implementation

The recommendation engine is implemented entirely in Java within the RecommendationService class. A @Scheduled(fixedRate=1800000) method computeAffinityMatrix() executes every 30 minutes. It issues a JPQL query:

```
SELECT oi.product.id, oi2.product.id, COUNT(DISTINCT
    oi.order.user.id)
FROM OrderItem oi
JOIN OrderItem oi2 ON oi.order = oi2.order AND oi.product !=
    oi2.product
GROUP BY oi.product.id, oi2.product.id
```

The result set is loaded into a ConcurrentHashMap<Long, List<Long>> affinity cache keyed by productId, with values being the top-5 co-purchased product IDs sorted by co-purchase frequency. The RecommendationController's GET /api/recommendations/{productId} endpoint reads from this pre-computed cache synchronously, delivering sub-5 ms in-memory lookup for the serving path. A fallback branch handles products with no purchase history by returning the top-5 products in the same category sorted by stockQuantity descending.

5. TESTING AND RESULTS

5.1 Testing Strategy

The testing campaign for the Retail IQ system was structured across four levels. Unit Testing employed JUnit 5 with Mockito for service-layer method isolation and MockMvc for controller-layer HTTP request/response simulation against an H2 in-memory database. Integration Testing verified end-to-end data flow between controllers, services, repositories, and the MySQL test instance. System Testing validated complete user journeys—registration through order placement—in a browser against the deployed application.

Performance Testing used Apache JMeter to simulate 50 concurrent users executing mixed read/write API workloads

TC ID	Test Case Description	Input Data	Expected Output	Result
TC-01	User Registration	Name, Email, Password	Account Created	Pass
TC-02	Valid User Login	Correct credentials	JWT Token + Dashboard	Pass
TC-03	Invalid Login Attempt	Wrong password	Error message shown	Pass
TC-04	Admin Login	Admin credentials	Admin panel access	Pass
TC-05	Add New Product (Admin)	Name, Price, Category	Product saved in DB	Pass
TC-06	Update Product Details	Modified price/name	DB record updated	Pass
TC-07	Delete Product (Admin)	Product ID	Product removed from DB	Pass
TC-08	Product Browse by User	Keyword / Category	Filtered list displayed	Pass
TC-09	Add Product to Cart	Product ID, Qty	Cart updated correctly	Pass
TC-10	Place Order	Cart contents	Order record created	Pass
TC-11	View Order History (User)	User session JWT	Orders listed correctly	Pass
TC-12	Admin View All Orders	Admin JWT	All orders displayed	Pass
TC-13	Low-Stock Alert Trigger	Stock < 10 units	Alert generated	Pass

TC-14	AI Recommendation Display	Product page load	Related items shown	Pass
TC-15	Unauthorized API Access	No / expired JWT	401 Unauthorized	Pass

Table 1 presents all fifteen functional test cases derived from the requirement specification. Test cases TC-01 through TC-04 validate authentication pathways for both roles. TC-05 through TC-07 exercise the full product CRUD lifecycle. TC-08 through TC-12 verify user-facing shopping and admin order management workflows. TC-13 confirms the inventory alert trigger fires correctly when stockQuantity falls below the threshold. TC-14 validates AI recommendation display on product pages. TC-15 verifies that the JWT filter correctly rejects requests without valid tokens. All fifteen test cases returned Pass results, confirming complete functional coverage.

identifies the correct co-purchased product pairs. All eight modules achieved Pass status.

Table -3: System Performance Testing Results

Test Scenario	Metric Measured	Observed Value	Status
Single User - Login	Response Time	87 ms	Pass
Single User - Browse Products	Page Load Time	1.1 sec	Pass
Single User - Place Order	API Round-Trip	143 ms	Pass
50 Concurrent Users - CRUD APIs	Avg. Response Time	< 130 ms	Pass
50 Concurrent Users - Page Load	Avg. Load Time	1.4 sec	Pass
DB Query - Order History Join	MySQL Execution Time	38 ms	Pass
AI Recommendation API Call	Computation Latency	< 240 ms	Pass
JWT Token Validation (per req.)	Filter Overhead	< 18 ms	Pass
Low-Stock Scheduler (15-min)	Background Task Time	< 50 ms	Pass

Table -2: Module-Wise Unit Testing Results

Module	Component Tested	Test Method	Status
Auth	register(), login(), JwtFilter	JUnit + MockMvc	Pass
Product	createProduct(), getAllProducts()	JUnit + H2 In-Memory DB	Pass
Order	placeOrder(), findById()	JUnit + MockMvc	Pass
Cart	addToCart(), removeItem()	JavaScript Unit Tests	Pass
Recommendation	computeAffinity(), getTopN()	JUnit + Sample Dataset	Pass
Inventory	checkLowStock(), generateAlert()	@Scheduled Task Test	Pass
Security	BCryptEncoder, JWT Validation	Spring Security Tests	Pass
Dashboard	getStats() Aggregation Query	MySQL EXPLAIN + JUnit	Pass

Table 2 presents unit testing results for the eight primary system modules. All modules were tested using JUnit 5 with MockMvc and an H2 in-memory database for repository isolation. The Security module test verifies that BCrypt- encoded passwords do not match plaintext inputs and that JWT tokens expire correctly at the five-hour mark. The Recommendation module test uses a seeded five-user, ten- product order dataset to confirm that the affinity matrix

Table 3 presents performance benchmarking results across nine test scenarios. Under 50-concurrent-user load, average CRUD API response time of 130 ms and page load time of 1.4 seconds both satisfy the non-functional requirements defined in Section 2.2. The AI recommendation API latency of under 240 ms confirms that the pre-computed affinity cache design successfully eliminates online computation from the serving path. The JWT filter overhead of under 18 ms demonstrates that stateless token validation is negligible relative to total request processing time. No errors were observed across all JMeter load scenarios.



Fig -1: Overall System Result

6. DESIGN ANALYSIS AND DISCUSSION

Three significant design decisions made during implementation warrant analytical discussion. First, the selection of Spring Boot over alternative Java frameworks (Quarkus, Micronaut) was driven by the auto-configuration ecosystem, comprehensive Spring Security integration, and the project team's existing familiarity with Spring annotations—reducing ramp-up time and ensuring stable production behaviour. The embedded Tomcat server eliminates external application server configuration, simplifying deployment to a single JAR execution.

Second, the choice of a stateless JWT authentication model over server-side session storage was validated by the performance results: JWT filter overhead of under 18 ms per request means the authentication layer contributes less than 14% of total API response time even at 50 concurrent users, while eliminating session-store scalability bottlenecks entirely. The five-hour token expiry balances user convenience against security exposure for the target retail use case.

Third, the decision to pre-compute the recommendation affinity matrix on a 30-minute schedule rather than computing it per-request was vindicated by the serving latency measurement: the in-memory Concurrent Hash Map lookup executes in under 5 ms, compared to an estimated 800–1,200 ms for an equivalent on-demand SQL aggregation join under concurrent load. The 30-minute refresh interval is appropriate for a retail catalogue where purchase patterns change on a daily rather than per-minute basis.

The @ManyToOne lazy-fetch relationship design in the OrderItem entity was confirmed as the correct approach through JPA query profiling: eager loading of Order and Product objects within the affinity matrix aggregation query produced N+1 SQL queries and a 340% increase in query execution time on the test dataset, whereas the JPQL projection query that selects only product IDs and co-purchase counts executed in 38 ms regardless of order volume—a 91% reduction in database execution time.

7. CONCLUSION AND FUTURE SCOPE

This paper presented the detailed implementation and design analysis of the Retail IQ AI Management System across all phases of the SDLC. The system was implemented using a Java Spring Boot RESTful backend, HTML/CSS/JavaScript frontend, MySQL relational database, and a Java-native item-based collaborative filtering recommendation engine. Fifteen functional test cases achieved 100% pass rates. Performance testing under 50 concurrent users confirmed average API response times below 130 ms and page load times below 1.4 seconds. Key design decisions—stateless JWT authentication, pre-computed affinity cache, and lazy-fetch JPA relationships—were analytically validated through empirical measurements presented in Section 6. The system demonstrates that a focused, domain-specific implementation can deliver robust retail intelligence capabilities at a fraction of the cost and complexity of generic e-commerce platforms.

Future implementation directions include: (i) Razorpay or Stripe payment gateway integration for complete digital transaction support; (ii) React Native mobile application development for Android and iOS, consuming the existing Spring Boot API without backend modification; (iii) replacement of the batch affinity matrix refresh with an event-driven Kafka consumer that updates recommendation scores in real-time on each order placement event; (iv) LSTM-based demand forecasting trained on historical order items time-series data to replace the simple rolling-average days-to-stockout estimate; (v) customer product rating and review module with VADER sentiment analysis for quality monitoring; and (vi) Docker containerisation with Kubernetes orchestration to support horizontal scaling beyond the current single-server embedded Tomcat deployment.

ACKNOWLEDGEMENT

The authors sincerely acknowledge the guidance and support of the Department of Master of Computer Applications, Karanjekar College of Engineering and Management, Sakoli, affiliated to Dr. Babasaheb Ambedkar Technological University, Lonere. Special appreciation is extended to all faculty members of the MCA Department for their sustained academic support throughout this research work.

REFERENCES

- [1] E. Turban, J. Outland, D. King, J. K. Lee, T. P. Liang, and D. C. Turban, *Electronic Commerce 2018: A Managerial and Social Networks Perspective*, 9th ed. Cham, Switzerland: Springer, 2018.
- [2] K. C. Laudon and C. G. Traver, *E-Commerce 2023: Business, Technology, and Society*, 18th ed. Hoboken, NJ: Pearson Education, 2023.
- [3] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 9th ed. New York, NY: McGraw-Hill Education, 2019.
- [4] G. Linden, B. Smith, and J. York, "Amazon.com Recommendations: Item-to-Item Collaborative Filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76-80, Jan./Feb. 2003.
- [5] C. Walls, *Spring Boot in Action*. Shelter Island, NY: Manning Publications, 2016.
- [6] I. Sommerville, *Software Engineering*, 10th ed. Harlow, UK: Pearson Education, 2015.
- [7] C. C. Aggarwal, *Recommender Systems: The Textbook*. Cham, Switzerland: Springer, 2016.
- [8] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2009.
- [9] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2002.
- [10] MySQL AB, "MySQL 8.0 Reference Manual," [Online].

Available:

<https://dev.mysql.com/doc/refman/8.0/en/>. [Accessed: May 2025].

[11] Spring.io, "Spring Boot 3.x Reference Documentation,"

[Online]. Available:

<https://docs.spring.io/spring-boot/docs/current/reference/html/>. [Accessed: May 2025].