Executive Summary

The e-commerce application represents a sophisticated Spring Boot-based platform implementing enterprise-level architecture patterns and modern web technologies. The system's distinguishing feature is its dual-interface approach, providing both traditional web interfaces using Thymeleaf and a comprehensive RESTful API layer. This design supports various client requirements while maintaining security and scalability. The platform includes advanced features such as personalized product recommendations, real-time inventory management, and secure payment processing capabilities.

Table of Contents

1. Security Architecture and Implementation

Authentication Strategy Overview

The application implements a sophisticated dual-authentication strategy, carefully designed to address different client needs while maintaining robust security. This hybrid approach combines traditional session-based authentication for web interfaces with JWT-based authentication for API access, providing optimal security for each use case.

Session-Based Authentication Details

The web interface authentication system utilizes Spring Security's session management capabilities with several enhanced security features:

```
@Configuration
@EnableWebSecurity
public class ShopConfig {
    @Bean
    @Order(2)
    public SecurityFilterChain sessionSecurityFilterChain(HttpSecurity http) {
        http
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
                .maximumSessions(1)
                .maxSessionsPreventsLogin(false)
```

```
            )
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/", "/home", "/auth/**", "/error").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            );
        return http.build();
    }
}
```

This implementation includes:

1. **Session Management**:
   - Session fixation protection through automatic session ID regeneration
   - Concurrent session control limiting users to one active session
   - Session timeout handling with configurable duration
   - Secure session cookie configuration
2. **Authentication Flow**:
   - Custom `AuthenticationSuccessHandler` for role-based redirect logic
   - Remember-me functionality using secure token persistence
   - Failed login attempt tracking with temporary account locking
   - Secure password reset workflow
3. **CSRF Protection**:
   - Token-based CSRF protection for forms
   - CSRF token rotation on authentication
   - Custom CSRF token repository for distributed environments

JWT Authentication Implementation

The REST API implements JWT-based authentication with comprehensive security measures:

```
public class JwtUtils {
    private final String jwtSecret;
    private final int expirationTime;

    public String generateTokenForUser(Authentication authentication) {
        ShopUserDetails userPrincipal = (ShopUserDetails) authentication.getPrincipal();

        return Jwts.builder()
            .setSubject(userPrincipal.getEmail())
            .claim("id", userPrincipal.getId())
            .claim("roles", extractRoles(userPrincipal))
            .setIssuedAt(new Date())
            .setExpiration(calculateExpirationDate())
            .signWith(generateKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    private Key generateKey() {
        return Keys.hmacShaKeyFor(Decoders.BASE64.decode(jwtSecret));
    }

    private List<String> extractRoles(ShopUserDetails userDetails) {
```

```
        return userDetails.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.toList());
    }
}
```

Key security features include:

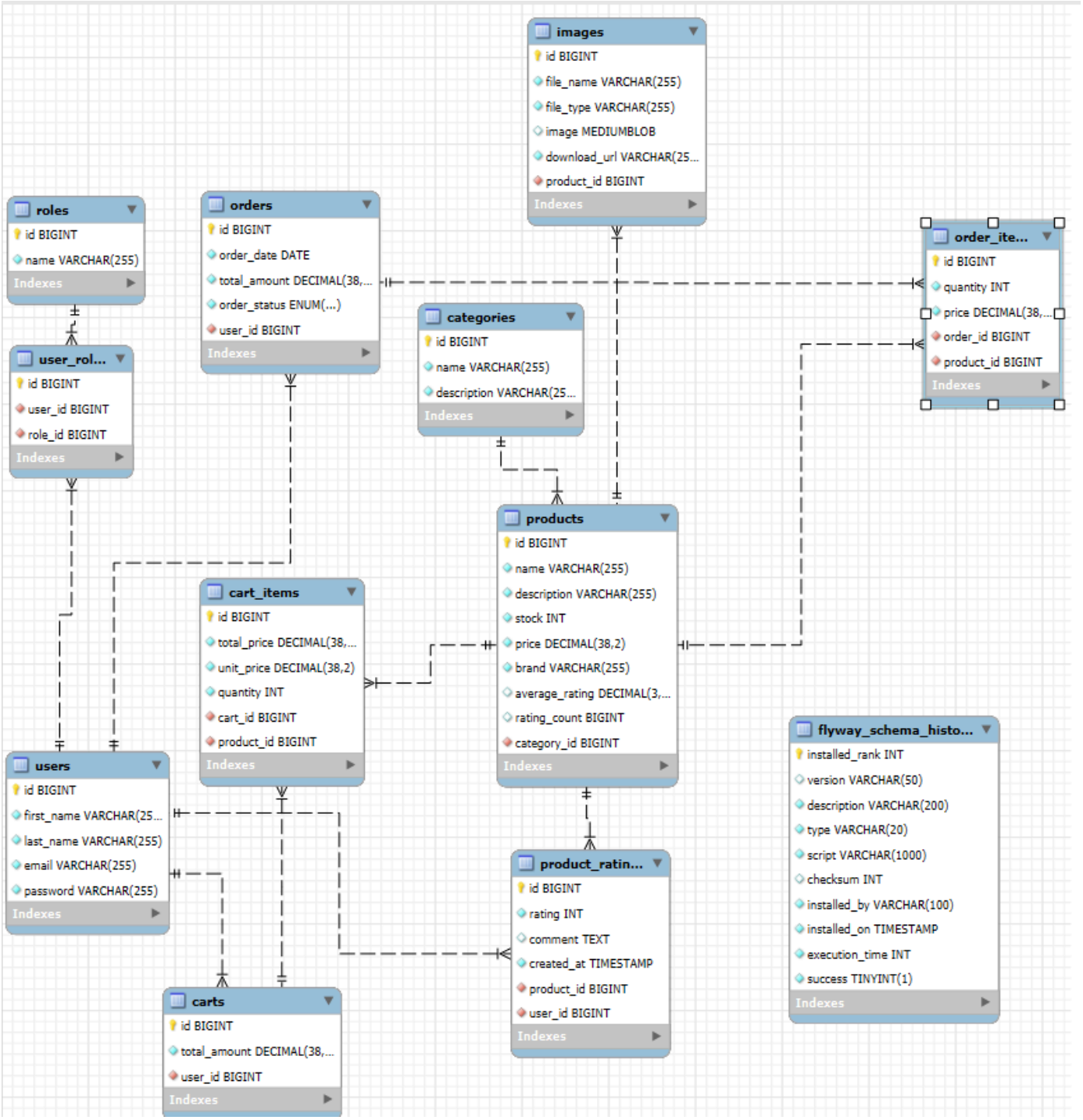
1. **Token Management**:
   - Secure token generation using HMAC-SHA256
   - Claims-based role and permission management
   - Token expiration and renewal handling
   - Blacklisting mechanism for revoked tokens
2. **Authorization Flow**:
   - Custom `AuthenticationTokenFilter` for token validation
   - Role-based access control using JWT claims
   - Fine-grained permission checking
   - Token refresh mechanism
3. **Security Headers**:
   - Implementation of security headers (X-Frame-Options, CSP, etc.)
   - XSS protection mechanisms
   - Content-Type enforcement

2. Database Design and Optimization

Schema Architecture

The database schema implements a fully normalized design with carefully considered relationships between entities. The design prioritizes data integrity while maintaining query performance through strategic denormalization where appropriate.

Entity-Relationship Diagram:



Core Entity Relationships:

```
CREATE TABLE products (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    stock INTEGER NOT NULL,
    price DECIMAL(38, 2) NOT NULL,
    brand VARCHAR(255) NOT NULL,
    average_rating DECIMAL(3, 2) DEFAULT 0.00,
    rating_count BIGINT DEFAULT 0,
    category_id BIGINT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
```

```
    FOREIGN KEY (category_id) REFERENCES categories (id)
);

CREATE TABLE orders (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    order_date DATE NOT NULL,
    total_amount DECIMAL(38, 2) NOT NULL,
    order_status ENUM ('PENDING','PROCESSING','SHIPPED','DELIVERED','CANCELLED') NOT NULL,
    user_id BIGINT NOT NULL,
    shipping_address_id BIGINT,
    payment_id BIGINT,
    FOREIGN KEY (user_id) REFERENCES users (id),
    FOREIGN KEY (shipping_address_id) REFERENCES addresses (id),
    FOREIGN KEY (payment_id) REFERENCES payments (id)
);
```

3. Recommendation System Architecture

Overview

The recommendation system implements multiple recommendation strategies through a flexible design pattern that allows dynamic strategy selection based on user behavior and data availability. The system is built around a core strategy interface with different implementations for various recommendation approaches.

Core Strategy Interface

```
public interface RecommendationStrategy {
    Page<ProductDto> getRecommendations(User user, Pageable pageable,
                                        boolean includeRatingWeight);
}
```

Strategy Implementations

1. Content-Based Filtering Strategy

```
@Component("contentBased")
@RequiredArgsConstructor
public class ContentBasedStrategy implements RecommendationStrategy {
    private final ProductRepository productRepository;
    private final ProductRatingRepository ratingRepository;
    private final IProductService productService;
}
```

Purpose:

- Used for new users with limited or no purchase history
- Relies on product attributes (category, brand, price) to make recommendations
- Effective when user preferences are known but interaction history is limited

Key Features:

- Calculates category preferences based on user browsing history
- Considers brand affinity from past interactions
- Weighs product similarity based on attributes
- Incorporates product ratings as a secondary factor

2. Item-Based Collaborative Filtering Strategy

```
@Component("itemBasedCF")
@RequiredArgsConstructor
public class ItemBasedCFStrategy implements RecommendationStrategy {
    private final OrderRepository orderRepository;
    private final ProductRepository productRepository;
    private final ProductRatingRepository ratingRepository;
    private final IProductService productService;
}
```

Purpose:

- Best for users with some purchase history
- Recommends products similar to ones the user has purchased or rated highly
- Effective for stable product catalogs

Key Features:

- Calculates product similarity based on:
  - Category matching (50% weight)
  - Brand similarity (30% weight)
  - Price range proximity (20% weight)
- Incorporates product ratings when available
- Uses purchase history to establish product relationships

3. Matrix Factorization Strategy

```
@Component("matrixFactorization")
@RequiredArgsConstructor
public class MatrixFactorizationStrategy implements RecommendationStrategy {
    private static final int LATENT_FACTORS = 10;
    private static final double LEARNING_RATE = 0.01;
    private static final double REGULARIZATION = 0.02;
    private static final int MAX_ITERATIONS = 100;
}
```

Purpose:

- Used for discovering latent features in user-product interactions
- Effective when there's sufficient rating data
- Good for finding non-obvious product relationships

Key Features:

- Implements Singular Value Decomposition (SVD)
- Uses gradient descent for optimization
- Considers both explicit (ratings) and implicit (purchases) feedback
- Balances between optimization and regularization

Orchestration Service

The `RecommendationService` class acts as the orchestrator for all recommendation strategies:

```
@Service
@RequiredArgsConstructor
public class RecommendationService implements IRecommendationService {
    private final Map<String, RecommendationStrategy> strategies;
    private final OrderRepository orderRepository;
    private final ProductRatingRepository ratingRepository;

    @Cacheable(value = "productRecommendations",
               key = "#user.id + '_' + #pageable.pageNumber + '_' + #pageable.pageSize")
    @Override
    public Page<ProductDto> getRecommendationsForUser(User user, Pageable pageable) {
        RecommendationType recommendationType = determineRecommendationType(user);
        return getRecommendations(user, recommendationType, pageable);
    }
}
```

Strategy Selection Logic

```
private RecommendationType determineRecommendationType(User user) {
    long orderCount = orderRepository.countByUserId(user.getId());
    boolean hasRatings = !ratingRepository.findByUserId(user.getId()).isEmpty();

    if (orderCount == 0) {
        return hasRatings ? RecommendationType.MATRIX_FACTORIZATION :
                            RecommendationType.CONTENT_BASED;
    } else if (orderCount < 5) {
        return hasRatings ? RecommendationType.ITEM_BASED :
                            RecommendationType.CONTENT_BASED;
    } else {
        return hasRatings ? RecommendationType.USER_BASED :
                            RecommendationType.ITEM_BASED;
    }
}
```

The strategy selection logic follows these rules:

1. New users (no orders):
    - If they have ratings: Use Matrix Factorization
    - If no ratings: Use Content-Based
2. Users with limited history (<5 orders):
    - If they have ratings: Use Item-Based CF

    - If no ratings: Use Content-Based
3. Users with substantial history (≥5 orders):
    - If they have ratings: Use User-Based CF
    - If no ratings: Use Item-Based CF

Performance Optimization

The system implements several performance optimizations:

1. **Caching:**

```
@Cacheable(value = "productRecommendations",
           key = "#user.id + '_' + #pageable.pageNumber + '_' + #pageable.pageSize")
```

- Recommendations are cached per user with pagination parameters
- Cache is invalidated when user behavior changes significantly

2. **Pagination:**

```
public Page<ProductDto> getRecommendations(User user,
    RecommendationType type, Pageable pageable) {
    String strategyName = switch (type) {
        case USER_BASED -> "userBasedCF";
        case ITEM_BASED -> "itemBasedCF";
        case CONTENT_BASED -> "contentBased";
        case MATRIX_FACTORIZATION -> "matrixFactorization";
    };

    return strategies.get(strategyName)
        .getRecommendations(user, pageable, true);
}
```

- All recommendation strategies support pagination
- Prevents memory overload with large result sets
- Allows for efficient client-side loading

The system's modular design allows for easy addition of new recommendation strategies while maintaining consistent interfaces and performance characteristics. Each strategy can be individually tuned and optimized based on specific use cases and data patterns.

4. REST API Design and Implementation

Architecture Overview

The REST API layer implements a comprehensive service-oriented architecture following REST principles and best practices. The API is versioned and follows a consistent URL structure under the `/order-api/v1/` prefix. Each endpoint is designed to be stateless, allowing for horizontal scalability and clear separation of concerns.

Response Handling and DTO Pattern

The application implements a consistent response pattern using Data Transfer Objects (DTOs) to manage data flow between the client and server. This approach provides several benefits: it decouples the internal domain model from the external API representation, allows for version control of the API contract, and provides a clean way to handle data validation.

```
public class ApiResponse {
    private String message;
    private Object data;
}

@RestController
@RequestMapping("${api.prefix}/products")
public class ProductRestController {
    @GetMapping("/{productId}")
    public ResponseEntity<ApiResponse> getProductById(@PathVariable Long productId) {
        ProductDto product = productService.convertToDto(
            productService.getProductById(productId));
        return ResponseEntity.ok(new ApiResponse("success", product));
    }
}
```

Error Handling Strategy

The application implements a centralized error handling mechanism through a global exception handler. This ensures consistent error responses across all endpoints and proper HTTP status code mapping. The system captures both expected business exceptions and unexpected runtime errors, providing appropriate feedback to API consumers.

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<?> handleResourceNotFoundException(
            ResourceNotFoundException ex, HttpServletRequest request) {
        logger.error("Resource not found: ", ex);
        if (isApiRequest(request)) {
            return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
        }
        return createModelAndView(HttpStatus.NOT_FOUND, "Resource Not Found",
                                  ex.getMessage());
    }
}
```

5. Caching Strategy

Multi-Level Caching Architecture

The application implements a sophisticated caching strategy using Spring Cache abstraction, providing a flexible and efficient caching solution. The caching system is designed to reduce database load and improve response times for frequently accessed data.

Cache Configuration and Management

The caching configuration is centralized in the CacheConfig class, which defines various cache regions for different types of data. The system uses a concurrent map cache implementation for development and can be easily switched to distributed caching solutions like Redis for production environments.

```
@Configuration
@EnableCaching
public class CacheConfig {
    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager cacheManager = new SimpleCacheManager();
        cacheManager.setCaches(Arrays.asList(
            new ConcurrentMapCache("products"),
            new ConcurrentMapCache("categories"),
            new ConcurrentMapCache("productRecommendations"),
            new ConcurrentMapCache("productRatings"),
            new ConcurrentMapCache("users")
        ));
        return cacheManager;
    }
}
```

Cache Usage in Services

The service layer implements caching using Spring's cache annotations. The caching strategy is carefully designed to balance memory usage with performance improvements. Cache eviction policies are implemented to ensure data consistency when underlying entities are modified.

```
@Service
public class ProductService {
    @Cacheable(value = "products", key = "#id")
    public Product getProductById(Long id) {
        return productRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Product not found"));
    }

    @CacheEvict(value = "products", allEntries = true)
    public Product updateProduct(UpdateProductRequest request, Long productId) {
        // Update logic
    }
}
```

6. Frontend Architecture

Thymeleaf Implementation

The frontend architecture leverages Thymeleaf's full capabilities through a sophisticated template hierarchy and component system:

Layout System

```
<!DOCTYPE html>
<html xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
    <title layout:title-pattern="$LAYOUT_TITLE: $CONTENT_TITLE">Order App</title>
    <!-- Common CSS/JS imports -->
    <link rel="stylesheet" th:href="@{/css/styles.css}">
    <th:block layout:fragment="headContent"/>
</head>
<body>
    <header th:replace="~{fragments/header :: header}"></header>
    <main layout:fragment="content">
        <!-- Page-specific content -->
    </main>
    <footer th:replace="~{fragments/footer :: footer}"></footer>
    <th:block th:replace="~{fragments/scripts :: scripts}"></th:block>
</body>
</html>
```

Component Organization

The application implements a modular component structure:

1. **Global Components**:

```
<!-- fragments/header.html -->
<header th:fragment="header">
    <nav class="navbar navbar-expand-lg navbar-dark">
        <!-- Navigation with dynamic authentication state -->
        <div th:if="${#authorization.expression('isAuthenticated()')}">
            <!-- Authenticated user menu -->
        </div>
    </nav>
</header>
```

2. **Product Components**:

```
<!-- fragments/products.html -->
<div th:fragment="products" class="container mb-5">
    <div class="row row-cols-1 row-cols-md-3 g-4">
        <div class="col" th:each="product : ${products}">
            <!-- Product card with rating system -->
            <div class="rating-container mb-2">
                <div class="stars">
                    <span th:each="i : ${#numbers.sequence(1, 5)}"
                          class="star"
                          th:classappend="${product.averageRating != null &&
                                           i <= product.averageRating} ?
                                           'filled' : ''"
                          th:text="'★'">
                    </span>
                </div>
            </div>
        </div>
```

```
        </div>
    </div>
```

3. **Recommendation Components**:

```
    <!-- fragments/recommendations.html -->
    <div th:fragment="recommendations"
         th:if="${not #lists.isEmpty(recommendations)}"
         sec:authorize="isAuthenticated() and hasRole('ROLE_USER')">
        <!-- Personalized product recommendations -->
    </div>
```

Security Integration in Templates

The frontend implements sophisticated security measures through Thymeleaf Security dialect:

```
<!-- Role-based content rendering -->
<div sec:authorize="hasRole('ROLE_ADMIN')">
    <!-- Admin-only content -->
    <a th:href="@{/admin/dashboard}" class="btn btn-primary">Dashboard</a>
</div>

<!-- Dynamic navigation based on authentication -->
<div sec:authorize="isAuthenticated()">
    <span th:text="${#authentication.name}"></span>
    <form th:action="@{/auth/logout}" method="post">
        <!-- Logout functionality -->
    </form>
</div>
```

7. Testing Framework

Comprehensive Testing Strategy

The application implements a multi-layered testing approach, ensuring code quality and functionality across all layers of the application. The testing strategy combines unit tests, integration tests, and end-to-end tests to provide comprehensive coverage of the codebase.

Unit Testing Implementation

Unit tests focus on isolated components, particularly the service layer where core business logic resides. The testing framework uses JUnit 5 with Mockito for dependency mocking, allowing for precise testing of business logic without external dependencies.

```
@ExtendWith(MockitoExtension.class)
class ProductServiceTests {
    @Mock
    private ProductRepository productRepository;
```

```
    @InjectMocks
    private ProductService productService;

    @Test
    void whenGetProductById_thenReturnProduct() {
        // Arrange
        Product mockProduct = new Product("Test Product", "Brand", 10,
            BigDecimal.TEN, "Description", new Category());
        when(productRepository.findById(1L)).thenReturn(Optional.of(mockProduct));

        // Act
        Product result = productService.getProductById(1L);

        // Assert
        assertNotNull(result);
        assertEquals("Test Product", result.getName());
        verify(productRepository).findById(1L);
    }
}
```

Integration Testing

Integration tests verify the interaction between different components of the application, focusing on database operations, cache behavior, and web layer functionality. These tests use Spring Boot's testing support to provide a realistic testing environment.

```
@SpringBootTest
@AutoConfigureMockMvc
class ProductControllerIntegrationTests {
    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ProductRepository productRepository;

    @Test
    @Transactional
    void whenCreateProduct_thenProductIsCreated() throws Exception {
        ProductDto productDto = new ProductDto();
        // Set up test data

        mockMvc.perform(post("/api/v1/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(productDto)))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.data.name")
                .value(productDto.getName()));
    }
}
```

8. Performance Optimization

Database Performance

The application implements several database optimization techniques to ensure efficient data access and manipulation. This includes careful index design, query optimization, and connection pooling configuration.

1. **Indexing Strategy**: The application implements a comprehensive indexing strategy based on query patterns:

```
-- Compound indexes for frequently filtered queries
CREATE INDEX idx_products_brand_name ON products(brand, name);

-- Indexes for foreign key relationships
CREATE INDEX idx_orders_user ON orders(user_id);

-- Indexes for status-based queries
CREATE INDEX idx_orders_status_date ON orders(order_status, order_date);

-- Full-text search indexes
CREATE FULLTEXT INDEX idx_products_search ON products(name, description);
```

2. **Query Optimization**: Repository interfaces implement optimized queries using JPA and native SQL where appropriate:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    @Query("SELECT DISTINCT p FROM Product p " +
           "LEFT JOIN FETCH p.category " +
           "LEFT JOIN FETCH p.images " +
           "WHERE p.category.id = :categoryId")
    List<Product> findByCategoryWithImages(@Param("categoryId") Long categoryId);

    @Query(value = "SELECT p.* FROM products p " +
                   "WHERE MATCH(name, description) AGAINST(?1 IN BOOLEAN MODE)",
           nativeQuery = true)
    List<Product> fullTextSearch(String searchTerm);
}
```

3. **Connection Pool Configuration**: The application uses HikariCP for connection pooling with optimized settings:

```
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
spring.datasource.hikari.idle-timeout=300000
spring.datasource.hikari.connection-timeout=20000
spring.datasource.hikari.max-lifetime=1200000
```

Caching Implementation

The caching strategy is implemented at multiple levels to optimize performance:

```
@Service
public class ProductService {
```

```
    @Cacheable(value = "products", unless = "#result == null")
    public List<ProductDto> getRecommendedProducts(Long userId) {
        return recommendationService.getRecommendations(userId)
            .stream()
            .map(this::convertToDto)
            .collect(Collectors.toList());
    }
}
```

Asynchronous Processing

The application uses asynchronous processing for non-critical operations to improve response times:

```
@Service
public class OrderService {
    @Async
    public CompletableFuture<Void> processOrderAsync(Order order) {
        return CompletableFuture.runAsync(() -> {
            // Process order asynchronously
            notificationService.sendOrderConfirmation(order);
            inventoryService.updateStock(order);
        });
    }
}
```

9. Deployment Architecture

Environment Configuration

The application uses Spring profiles to manage different environment configurations:

```
spring:
  profiles:
    active: ${SPRING_PROFILES_ACTIVE:dev}
  datasource:
    url: jdbc:mysql://${DB_HOST:localhost}:${DB_PORT:3306}/${DB_NAME}
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
```

Database Migration

Flyway manages database schema evolution with versioned migrations:

```
-- V1__schema.sql
CREATE TABLE products (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    -- other columns
);
```

```
-- V3__indexes.sql
CREATE INDEX idx_products_name ON products(name);
```

Monitoring and Logging

The application implements comprehensive logging and monitoring:

```
@Aspect
@Component
public class PerformanceMonitoringAspect {
    private static final Logger logger =
        LoggerFactory.getLogger(PerformanceMonitoringAspect.class);

    @Around("@annotation(Monitored)")
    public Object monitorPerformance(ProceedingJoinPoint joinPoint)
            throws Throwable {
        long start = System.currentTimeMillis();
        Object result = joinPoint.proceed();
        long executionTime = System.currentTimeMillis() - start;

        logger.info("Method {} executed in {} ms",
            joinPoint.getSignature(), executionTime);
        return result;
    }
}
```

10. Conclusions and Future Improvements

Current Achievements

- Successfully implemented a scalable e-commerce platform
- Robust security implementation with dual authentication
- Efficient recommendation system with multiple strategies
- (Comprehensive testing coverage)
- Optimized performance through caching and indexing

Future Improvements

Improvements include:

1. **Enhanced Caching**:
    - Implement Redis for distributed caching
    - Add cache warming strategies
    - Implement cache versioning
2. **Security Enhancements**:
    - Add OAuth2 support
    - Implement rate limiting
    - Add IP-based blocking
3. **Performance Optimizations**:

    - Implement query result caching
    - Add database connection pooling metrics
    - Optimize image loading and caching
4. **Frontend Improvements**:
    - Implement lazy loading for images
    - Add client-side caching
    - Enhance mobile responsiveness
5. **Monitoring and Logging**:
    - Add comprehensive application metrics
    - Implement centralized logging
    - Add performance monitoring