PaymentServiceTest Documentation

Overview

This document details the unit tests implemented for the PaymentService class in our e-commerce application backend. The PaymentService manages payment processing for orders, including validating bank information, processing payments, and retrieving payment records.

Test Class Structure

Dependencies

The test class uses:

- **JUnit 5**: For test execution and assertions
- **Mockito**: To mock dependencies and simulate interactions
- **Spring Boot Test**: For integration with the Spring testing framework

Mocked Components

- **OrderRepository**: Database access for order operations
- **PaymentRepository**: Database access for payment operations
- **UserRepository**: Database access for user operations
- **ProductRepository**: Database access for product operations
- **CartRepository**: Database access for cart operations
- **InvoiceService**: Service for generating and emailing invoices

Test Setup

Before each test, the following setup is performed:

1. Initialize mocks using MockitoAnnotations
2. Generate random UUIDs for userId and orderId
3. Create test objects:
    - A test user with email address
    - A test order in "Processing" status with paid status set to false
4. Configure the invoiceService mock to do nothing when emailPdfInvoice is called

Test Cases

1. testGetBankInformation_ValidInfo

Purpose: Verify that bank card information is properly validated.

Test Scenario:

- **Arrange**:
    - Prepare test card information (card number, expiry date, CVV)
    - Configure userRepository mock to return the test user
- **Act**:
    - Call paymentService.getBankInformation with userId and card details
- **Assert**:
    - Verify the response message is "Bank information is valid."
    - Verify the response has 200 status code

Business Logic Verified:

- The system can validate bank card information
- Users with valid card information receive appropriate success messages
- Appropriate HTTP status codes are returned

2. testProcessPayment_Successful

Purpose: Verify that payments can be successfully processed for orders.

Test Scenario:

- **Arrange**:
    - Prepare test card information
    - Configure orderRepository mock to return the test order
    - Configure userRepository mock to return the test user
    - Configure paymentRepository save method to return a new payment
    - Configure orderRepository save method to return the updated order
- **Act**:
    - Call paymentService.processPayment with userId, orderId, and card details
- **Assert**:
    - Verify the response message is "Payment processed & invoice emailed!"
    - Verify the response has 200 status code
    - Verify the order's paid flag is set to true
    - Verify the order's status remains "Processing"
    - Verify the order has a paymentId assigned
    - Verify the invoiceService.emailPdfInvoice method was called once with the correct orderId
    - Verify the paymentRepository.save method was called once
    - Verify the orderRepository.save method was called once with the updated order

Business Logic Verified:

- Payments can be processed for orders
- Order status is updated correctly after payment
- Payment records are created and linked to orders

- Invoice emails are sent upon successful payment
- Changes are persisted to the database

3. testGetPaymentByOrderId_ReturnsPayment

Purpose: Verify that payment information can be retrieved for a specific order.

Test Scenario:

- **Arrange**:
    - Create an expected payment object linked to the test orderId
    - Configure paymentRepository mock to return this payment when queried
- **Act**:
    - Call paymentService.getPaymentByOrderId with the orderId
- **Assert**:
    - Verify the result is present (not empty)
    - Verify the returned payment matches the expected payment
    - Verify the paymentRepository.findByOrderId method was called once with the correct orderId

Business Logic Verified:

- The system can retrieve payment information for specific orders
- The correct payment record is returned for an order
- Query operations use the correct parameters

Mocking Strategy

The tests use a consistent mocking strategy to isolate the PaymentService from its dependencies:

1. **Mock Responses**: Return prepared test objects when repository methods are called

2. **Behavior Verification**: Verify that the service calls repository and service methods as expected
3. **State Verification**: Check that objects are modified correctly before being saved
4. **Mock Behavior**: Configure void methods (like emailPdfInvoice) to do nothing when called

Test Coverage

These tests cover the core functionality of the PaymentService:

- Validating bank card information
- Processing payments for orders
- Retrieving payment records by order ID
- Integration with invoice generation and delivery

Conclusion

The PaymentServiceTest thoroughly verifies the core functionality of the payment processing system. The tests ensure that payments can be validated, processed, and retrieved, with appropriate updates to order status and invoice delivery.

These tests help maintain the integrity of the payment system as the application evolves, ensuring secure and reliable payment processing for customers. The payment system integration with order management and invoice delivery is also verified, ensuring a smooth end-to-end payment experience.