Software Reengineering Project

Reengineering Legacy POS System to Modern Web Application

Muhammad Sarmad Hassan (22i-8792)
Mirza Mukarram Haseeb (22i-2488)
Muhammad Awaimer Zaeem (20i-2616)

December 7, 2025

Contents

Executive Summary

This document presents a comprehensive report on the software reengineering project of the SG Technologies Point-of-Sale (POS) system. The project involved transforming a legacy desktop-based Java application into a modern, web-based system following the Software Reengineering Process Model. The reengineered system utilizes Spring Boot for the backend, React with TypeScript for the frontend, and PostgreSQL for data storage.

1.1 Project Objectives

- Analyze and document the legacy system architecture
- Restructure code to improve maintainability
- Migrate from file-based storage to relational database
- Implement modern web-based architecture
- Ensure feature parity with legacy system
- Improve security, scalability, and user experience

1.2 Key Achievements

- Successfully completed all 6 phases of reengineering process
- Eliminated 130+ lines of duplicate code
- Designed normalized database schema with 11 tables
- Implemented complete REST API with 15+ endpoints
- Built modern React frontend with Material-UI
- Achieved 100% feature parity with legacy system
- Created comprehensive test suite with 30+ backend tests and frontend tests
- Configured H2 in-memory database for isolated testing
- Achieved approximately 60% overall test coverage

Introduction

2.1 Project Background

The SG Technologies POS system was originally developed as a desktop application using Java Swing for the user interface and plain text files for data storage. While functional, the system suffered from several limitations:

- **Single-user limitation**: Desktop application restricted to one user at a time

- **File-based storage**: Data stored in text files without proper relationships
- **Security concerns**: Plain text password storage
- **Maintainability issues**: Code duplication, magic numbers, tight coupling
- **No scalability**: Cannot handle concurrent transactions efficiently

2.2 Reengineering Goals

The reengineering project aimed to address these limitations by:

1. **Modernizing architecture**: Transition to web-based, multi-user system
2. **Improving data management**: Migrate to normalized relational database
3. **Enhancing security**: Implement JWT authentication and password hashing
4. **Refactoring code**: Eliminate duplication and improve maintainability
5. **Improving UX**: Modern web interface with better usability

Work Division

Member	**Responsibility**
Muhammad Sarmad Hassan	Frontend Implementation
Mirza Mukarram Haseeb	Backend Implementation
Awaimer Zaeem	Documentation and Static Analysis

Table 1: Work division among group members

Reengineering Process Model

The project followed the Software Reengineering Process Model, which consists of six distinct phases:

4.1 Phase 1: Inventory Analysis

4.1.1 Objectives

The inventory analysis phase involved cataloging all assets of the legacy system, including source code files, data files, documentation, and dependencies.

4.1.2 Findings

- **Source Code**: 20 Java classes totaling approximately 3,000 lines of code
- **Data Files**: 9 text-based database files storing different entities
- **Test Coverage**: Less than 5% test coverage
- **Dependencies**: Minimal external dependencies, mostly Java standard library

4.1.3 Asset Classification

Assets were classified into three categories:

- **Keep**: Core business logic, domain models
- **Refactor**: Classes with code smells, duplicate code
- **Replace**: File I/O operations, Swing UI components

4.2 Phase 2: Document Restructuring

4.2.1 Objectives

Reorganize and enhance documentation to support the reengineering effort.

4.2.2 Deliverables

- Data dictionary for all file formats
- Operational scenarios and use cases
- Architecture diagrams
- API documentation structure

4.3 Phase 3: Reverse Engineering

4.3.1 Architecture Analysis

The legacy system followed a layered architecture with the following components:

1. **Presentation Layer**: Swing-based GUI components
2. **Business Logic Layer**: Core POS operations (Sales, Rentals, Returns)
3. **Data Access Layer**: File I/O operations for data persistence

4.3.2 Code Smells Identified

1. **God Classes**: POSSystem, PointOfSale, Management classes with excessive responsibilities
2. **Long Methods**: Methods exceeding 80 lines of code
3. **Code Duplication**: deleteTempItem method duplicated in 3 classes
4. **Magic Numbers**: Hardcoded values throughout the codebase
5. **Feature Envy**: Classes accessing data from other classes excessively
6. **Data Clumps**: Related data not grouped into objects
7. **Primitive Obsession**: Using primitives instead of value objects

8. **Inappropriate Intimacy**: Classes knowing too much about each other
9. **Lazy Class**: Classes with minimal functionality
10. **Speculative Generality**: Over-abstracted code

4.3.3 Data Smells Identified

- **Denormalized Data**: Customer and rental data mixed in single file
- **No Referential Integrity**: No foreign key relationships
- **Inconsistent Formats**: Mix of space and comma delimiters
- **No Data Validation**: No constraints on data values

4.4 Phase 4: Code Restructuring

4.4.1 Refactoring Activities

Ten major refactorings were performed:

1. **Extract Constants Class**: Centralized 20+ magic numbers and strings
2. **Extract SystemUtils Class**: Consolidated OS detection and line separator logic
3. **Apply Constants to POSSystem**: Replaced hardcoded file paths
4. **Apply Constants to PointOfSale**: Replaced magic values
5. **Apply Constants to POS**: Updated sale-related constants
6. **Apply Constants to POR**: Updated rental-related constants
7. **Apply Constants to POH**: Updated return-related constants
8. **Apply Constants to Management**: Updated user database constants
9. **Apply Constants to EmployeeManagement**: Updated employee constants
10. **Extract Duplicate Method**: Consolidated deleteTempItem into base class

4.4.2 Results

- Eliminated approximately 130 lines of duplicate code
- Centralized all magic values in Constants class
- Improved code maintainability and readability
- Zero behavior changes (safe refactorings)
- All tests passing after refactoring

4.5 Phase 5: Data Restructuring

4.5.1 Database Design

A normalized PostgreSQL schema was designed with 11 tables:

1. **employees**: Employee accounts and authentication
2. **items**: Inventory items with pricing
3. **customers**: Customer information
4. **coupons**: Discount codes
5. **sales**: Sales transactions
6. **sale_items**: Items in each sale
7. **rentals**: Rental transactions
8. **rental_items**: Items in each rental
9. **returns**: Return transactions
10. **return_items**: Items in each return
11. **audit_logs**: System audit trail

4.5.2 Normalization

The schema follows Third Normal Form (3NF):

- No repeating groups
- No partial dependencies
- No transitive dependencies
- Proper foreign key relationships

4.5.3 Data Migration Strategy

A migration utility was developed to:

1. Parse legacy text files
2. Transform data to normalized structure
3. Load data into PostgreSQL
4. Validate data integrity

4.6 Phase 6: Forward Engineering

4.6.1 Technology Stack Selection

Backend:

- Spring Boot 3.2.0
- Spring Data JPA
- Spring Security with JWT
- PostgreSQL
- Maven

Frontend:

- React 18.2
- TypeScript 5.3
- Material-UI 5.14
- Vite 5.0
- Axios 1.6

4.6.2 Architecture Design

The reengineered system follows a layered architecture:

1. **Presentation Layer**: React components with Material-UI
2. **API Layer**: Spring Boot REST controllers
3. **Business Logic Layer**: Spring services
4. **Data Access Layer**: Spring Data JPA repositories
5. **Database Layer**: PostgreSQL

4.6.3 Key Features Implemented

- JWT-based authentication
- Role-based access control (Admin/Cashier)
- Sales processing with tax and coupon support
- Rental management with customer lookup
- Return processing with overdue tracking
- Inventory management
- Employee management (Admin only)

Implementation Details

5.1 Backend Implementation

5.1.1 Entity Design

JPA entities were created for all database tables with proper relationships:

Listing 1: Employee Entity Example

```java
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String position; // Admin or Cashier

    @Column(name = "password_hash", nullable = false)
    private String passwordHash;

    // Getters and setters
}
```

5.1.2 Service Layer

Business logic was implemented in service classes:

Listing 2: Sale Service Example

```java
@Service
public class SaleService {
    @Transactional
    public Sale processSale(UUID employeeId, SaleRequest
        request) {
        // Calculate totals
        // Update inventory
        // Apply tax and discounts
        // Save transaction
    }
}
```

5.1.3 API Endpoints

RESTful endpoints were created for all operations:

- `POST /api/auth/login` - Authentication

- `POST /api/sales` - Process sale
- `POST /api/rentals` - Process rental
- `POST /api/returns` - Process return
- `GET /api/items` - Get inventory
- `GET /api/employees` - Employee management

5.2 Frontend Implementation

5.2.1 Component Structure

React components were organized into:

- **Pages**: Login, CashierDashboard, AdminDashboard
- **Components**: SalesPage, RentalsPage, ReturnsPage, etc.
- **Services**: API service layer
- **Context**: Authentication context

5.2.2 Authentication Flow

1. User enters credentials on login page
2. Frontend sends POST request to /api/auth/login
3. Backend validates credentials and returns JWT token
4. Token stored in localStorage
5. Token included in all subsequent API requests
6. Protected routes check authentication status

Testing

6.1 Testing Strategy

A comprehensive testing strategy was implemented covering unit tests, integration tests, and end-to-end API tests. The test suite ensures reliability and maintainability of the reengineered system.

6.2 Backend Testing

JUnit 5 tests were created with comprehensive coverage:

- **Controller Tests**: Authentication controller with MockMvc
- **Service Tests**: Sale, Rental, Return, Employee, and Inventory services
- **Repository Tests**: Item and Customer repository queries
- **Integration Tests**: Complete sale workflow with coupon application

6.2.1 Test Configuration

Tests are configured to use H2 in-memory database for isolation:

- H2 database dependency added for test scope
- Test profile configuration (`application-test.properties`)
- `@ActiveProfiles("test")` annotation on all test classes
- JWT configuration for test environment
- Transaction rollback after each test

6.2.2 Test Coverage

The backend test suite includes:

- 30+ test methods covering all critical paths
- Service layer: 100% coverage of business logic
- Repository layer: All query methods tested
- Controller layer: Authentication and error handling
- Integration tests: End-to-end workflows

6.3 Frontend Testing

React Testing Library with Vitest tests were created for:

- **Component Tests**: Login component with form validation
- **Context Tests**: Authentication context and state management
- **Service Tests**: API service layer mocking
- **Integration Tests**: Component interaction and user flows

6.3.1 Test Configuration

Frontend tests use:

- Vitest as the test runner (replacing Jest)
- React Testing Library for component testing
- jsdom environment for DOM simulation
- Mock service layer for API calls

6.4 API Testing

Comprehensive API testing scripts were created:

- **PowerShell Scripts**: Windows-compatible integration tests
- **Bash Scripts**: Linux/Mac end-to-end test suites
- **Integration Tests**: Complete workflows (login, sales, rentals, returns)
- **Manual Testing**: Postman collection for API exploration

6.4.1 Test Scenarios

API tests cover:

- Authentication and authorization flows
- Sales processing with multiple items and coupons
- Rental creation and customer management
- Return processing with inventory restoration
- Inventory management operations
- Employee management (Admin only)

Results and Improvements

7.1 Code Quality Improvements

- **Code Duplication**: Reduced by 130+ lines
- **Maintainability**: Improved through centralized constants
- **Test Coverage**: Increased from ¡5% to approximately 60% overall coverage
  - Backend: 30+ test methods covering services, controllers, and repositories
  - Frontend: Component and context tests with Vitest
  - Integration: End-to-end API test scripts
- **Documentation**: 10+ comprehensive markdown documents
- **Test Infrastructure**: H2 in-memory database for isolated testing

7.2 Architecture Improvements

- **Scalability**: Multi-user web application vs. single-user desktop
- **Maintainability**: Layered architecture with clear separation
- **Testability**: Dependency injection enables unit testing
- **Extensibility**: Easy to add new features

7.3 Data Management Improvements

- **Normalization**: Proper relational structure
- **Integrity**: Foreign key constraints
- **Concurrency**: Database transactions
- **Performance**: Indexed queries

7.4 Security Improvements

- **Authentication**: JWT tokens vs. plain text
- **Password Storage**: BCrypt hashing
- **Authorization**: Role-based access control
- **API Security**: Protected endpoints

Challenges and Solutions

8.1 Challenge 1: Data Migration

Problem: Migrating data from text files to normalized database.
Solution: Created DataMigrationUtil class to parse files and transform data.

8.2 Challenge 2: Legacy Code Understanding

Problem: Understanding legacy code structure and business logic.
Solution: Comprehensive reverse engineering analysis and documentation.

8.3 Challenge 3: Feature Parity

Problem: Ensuring all legacy features work in new system.
Solution: Detailed feature mapping and comprehensive testing.

8.4 Challenge 4: Test Configuration

Problem: Initial test failures due to ApplicationContext loading issues. Tests were attempting to connect to PostgreSQL database which may not be available during test execution.
Solution:

- Configured H2 in-memory database for test environment
- Created test profile with `application-test.properties`
- Added `@ActiveProfiles("test")` to all test classes
- Configured JWT properties for test environment
- Tests now run independently without external database dependency

8.5 Challenge 5: Spring Data JPA Query Method Naming

Problem: Repository method `findByQuantityLessThanOrEqual` caused ApplicationContext loading failure with error: "No property 'equal' found for type 'Item'".

Solution:

- Identified that Spring Data JPA does not support `LessThanOrEqual` keyword
- Changed method name to `findByQuantityLessThanEqual` (removed "Or")
- Updated service layer to use corrected method name
- All tests now pass successfully

Conclusion

The software reengineering project successfully transformed the legacy POS system into a modern, web-based application. All six phases of the reengineering process were completed, resulting in:

- Improved code quality and maintainability
- Modern web-based architecture
- Secure authentication and authorization
- Normalized database structure
- Enhanced user experience
- Comprehensive documentation

The reengineered system maintains feature parity with the legacy system while providing significant improvements in scalability, security, and maintainability.

Future Work

Potential enhancements for future iterations:

- Increase test coverage to 80%+ with additional edge case tests
- Implement reporting and analytics features
- Add mobile application support (React Native)
- Implement real-time inventory updates with WebSockets
- Add barcode scanning support
- Implement loyalty program features
- Add automated CI/CD pipeline with test execution
- Implement performance testing and optimization
- Add comprehensive API documentation with Swagger/OpenAPI

Appendix A: Database Schema

A.1 Schema Overview

The complete database schema DDL is available in `Database/schema.sql`. This file contains the Data Definition Language (DDL) script for the PostgreSQL database schema, including all 11 tables, their relationships, constraints, and indexes. The schema follows Third Normal Form (3NF) normalization principles and includes proper foreign key relationships for data integrity.

A.2 Table Structure

The database schema consists of the following tables:

1. **employees**: Stores employee authentication and profile information
   - Primary Key: `id` (UUID)
   - Unique Constraints: `username`
   - Fields: username, first_name, last_name, position, password_hash, created_at, updated_at
   - Relationships: Referenced by sales, rentals, and returns tables
2. **items**: Inventory items with pricing and quantity information
   - Primary Key: `id` (UUID)
   - Unique Constraints: `item_id` (business identifier)
   - Fields: item_id, name, price, quantity, created_at, updated_at
   - Relationships: Referenced by sale_items, rental_items
3. **customers**: Customer information for rentals and returns
   - Primary Key: `id` (UUID)
   - Unique Constraints: `phone`
   - Fields: first_name, last_name, phone, email, created_at, updated_at
   - Relationships: Referenced by rentals table
4. **coupons**: Discount codes for sales transactions
   - Primary Key: `id` (UUID)
   - Unique Constraints: `code`
   - Fields: code, discount_percent, active, valid_from, valid_to, created_at
5. **sales**: Sales transaction records
   - Primary Key: `id` (UUID)
   - Foreign Keys: employee_id → employees

- Fields: transaction_date, total_amount, tax_amount, discount_amount, final_total, coupon_code, created_at
- Relationships: One-to-many with sale_items

6. **sale_items**: Individual items in sales transactions
   - Primary Key: `id` (UUID)
   - Foreign Keys: sale_id → sales, item_id → items
   - Fields: quantity, unit_price, subtotal, created_at
7. **rentals**: Rental transaction records
   - Primary Key: `id` (UUID)
   - Foreign Keys: employee_id → employees, customer_id → customers
   - Fields: rental_date, due_date, total_amount, tax_amount, created_at
   - Relationships: One-to-many with rental_items, one-to-many with returns
8. **rental_items**: Individual items in rental transactions
   - Primary Key: `id` (UUID)
   - Foreign Keys: rental_id → rentals, item_id → items
   - Fields: quantity, unit_price, returned, return_date, days_overdue, created_at
9. **returns**: Return transaction records
   - Primary Key: `id` (UUID)
   - Foreign Keys: rental_id → rentals, employee_id → employees
   - Fields: return_date, total_refund, created_at
   - Relationships: One-to-many with return_items
10. **return_items**: Individual items in return transactions
    - Primary Key: `id` (UUID)
    - Foreign Keys: return_id → returns, rental_item_id → rental_items
    - Fields: quantity, refund_amount, created_at

A.3 Normalization Details

The schema achieves Third Normal Form (3NF) through:

- **Elimination of Repeating Groups**: Transaction items stored in separate tables (sale_items, rental_items, return_items)
- **Removal of Partial Dependencies**: All non-key attributes fully depend on primary keys
- **Removal of Transitive Dependencies**: No indirect dependencies between attributes

- **Referential Integrity**: Foreign key constraints ensure data consistency
- **Unique Constraints**: Prevent duplicate business identifiers (usernames, item IDs, phone numbers)

A.4 Indexes and Performance

The schema includes indexes on:

- Foreign key columns for faster joins
- Unique constraint columns (username, item_id, phone, code)
- Frequently queried columns (rental_date, due_date, transaction_date)

A.5 Data Types

- **UUID**: Used for all primary keys to ensure global uniqueness
- **NUMERIC(10,2)**: Monetary values (prices, totals, taxes)
- **NUMERIC(5,2)**: Percentage values (discount percentages)
- **VARCHAR**: Variable-length strings with appropriate size limits
- **BOOLEAN**: Flags for active status, returned status
- **TIMESTAMP**: Date and time fields with timezone support
- **DATE**: Date-only fields (due dates, return dates)

A.6 Migration from Legacy System

The migration process transforms legacy text file data into the normalized schema:

- Employee data from `employeeDatabase.txt` → `employees` table
- Item data from `itemDatabase.txt` → `items` table
- Coupon codes from `couponNumber.txt` → `coupons` table
- Transaction data parsed and split into parent and child tables
- Passwords hashed using BCrypt before storage
- UUIDs generated for all new records

Appendix B: API Documentation

B.1 API Overview

The reengineered POS system provides a comprehensive RESTful API built with Spring Boot. All endpoints are prefixed with `/api` and follow REST conventions. The API uses JWT (JSON Web Tokens) for authentication and returns JSON responses.

B.2 Base URL

- Development: `http://localhost:8081/api`
- Production: `https://api.sgtech-pos.com/api`

B.3 Authentication

All endpoints except authentication and migration require a valid JWT token in the Authorization header:

Listing 3: Authentication Header Format

```
Authorization: Bearer <jwt_token>
X-Employee-Id: <employee_uuid>
```

B.4 API Endpoints

B.4.1 Authentication Endpoints

- **POST /api/auth/login**
  - Description: Authenticate user and receive JWT token
  - Request Body: {`username: string, password: string`}
  - Response: {`token: string, employeeId: string, username: string, fullName: string, position: string`}
  - Status Codes: 200 (success), 400 (invalid credentials)
  - Example:

```
POST /api/auth/login
{
  "username": "110001",
  "password": "1"
}
```

- **GET /api/auth/health**
  - Description: Health check endpoint
  - Response: `"Auth service is running"`
  - Authentication: Not required

B.4.2 Sales Endpoints

- **POST /api/sales**
  - Description: Process a new sale transaction
  - Request Body: {`items: Array<`{`itemId: number, quantity: number`}`>, couponCode?: string`}
  - Response: Complete sale object with calculated totals

  - Authentication: Required (JWT token)
  - Business Logic: Calculates subtotal, applies tax (8.25%), applies coupon discount if valid, updates inventory quantities

- **GET /api/sales**
  - Description: Retrieve all sales transactions
  - Response: Array of sale objects
  - Authentication: Required (Admin only)

- **GET /api/sales/{id}**
  - Description: Retrieve specific sale by ID
  - Response: Sale object with sale_items
  - Authentication: Required

B.4.3 Rental Endpoints

- **POST /api/rentals**
  - Description: Create a new rental transaction
  - Request Body: {`customerPhone: string, items: Array<{itemId: number, quantity: number}>, dueDate: string`}
  - Response: Complete rental object
  - Authentication: Required
  - Business Logic: Creates or finds customer, calculates rental total, sets due date, updates inventory

- **GET /api/rentals**
  - Description: Retrieve all rental transactions
  - Response: Array of rental objects
  - Authentication: Required

- **GET /api/rentals/{id}**
  - Description: Retrieve specific rental by ID
  - Response: Rental object with rental_items
  - Authentication: Required

B.4.4 Return Endpoints

- **POST /api/returns**
  - Description: Process a return transaction
  - Request Body: {`rentalId: UUID, items: Array<{rentalItemId: UUID, quantity: number}>`}

  - Response: Complete return object with refund amounts
  - Authentication: Required
  - Business Logic: Calculates refunds, handles overdue items, restores inventory quantities

- **GET /api/returns**
  - Description: Retrieve all return transactions
  - Response: Array of return objects
  - Authentication: Required

B.4.5 Inventory Endpoints

- **GET /api/items**
  - Description: Retrieve all inventory items
  - Response: Array of item objects
  - Authentication: Required
  - Query Parameters: `?search=<name>` (search by name), `?lowStock=<threshold>` (low stock items)
- **GET /api/items/{id}**
  - Description: Retrieve specific item by UUID
  - Response: Item object
  - Authentication: Required
- **GET /api/items/item-id/{itemId}**
  - Description: Retrieve item by business ID (integer)
  - Response: Item object
  - Authentication: Required
- **PUT /api/items/{id}/quantity**
  - Description: Update item quantity
  - Request Body: {`quantity: number`}
  - Response: Updated item object
  - Authentication: Required (Admin only)

B.4.6 Employee Management Endpoints

- **GET /api/employees**
  - Description: Retrieve all employees
  - Response: Array of employee objects (without password hashes)
  - Authentication: Required (Admin only)

- **POST /api/employees**
  - Description: Create new employee
  - Request Body: {`username: string, firstName: string, lastName: string, position: string, password: string`}
  - Response: Created employee object
  - Authentication: Required (Admin only)
- **PUT /api/employees/{id}**
  - Description: Update employee information
  - Request Body: Employee fields to update
  - Response: Updated employee object
  - Authentication: Required (Admin only)
- **DELETE /api/employees/{id}**
  - Description: Delete employee account
  - Response: Success message
  - Authentication: Required (Admin only)

B.4.7 Data Migration Endpoints

- **POST /api/migration/migrate**
  - Description: Migrate data from legacy text files to database
  - Query Parameters: `?databasePath=<path>`
  - Response: Success message with migration statistics
  - Authentication: Not required (public endpoint for initial setup)
  - Migrates: Employees, Items, Coupons from legacy files

B.5 Error Handling

All endpoints return consistent error responses:

Listing 4: Error Response Format

```
{
  "error": "Error message description",
  "status": "400" or "500"
}
```

Common HTTP status codes:

- **200 OK**: Successful request
- **400 Bad Request**: Invalid request data or business logic error
- **401 Unauthorized**: Missing or invalid JWT token

- **403 Forbidden**: Insufficient permissions (e.g., Cashier accessing Admin endpoint)
- **404 Not Found**: Resource not found
- **500 Internal Server Error**: Server-side error

B.6 Request/Response Examples

Complete API documentation with detailed examples is available in the project repository. The API follows OpenAPI/Swagger conventions and can be explored using tools like Postman or Swagger UI.

Appendix C: Test Results

C.1 Test Suite Overview

The reengineered POS system includes a comprehensive test suite covering unit tests, integration tests, and end-to-end API tests. All tests are configured to run independently using H2 in-memory database, ensuring fast execution and isolation from external dependencies.

C.2 Backend Test Suite

C.2.1 Test Structure

Backend tests are located in `pos-backend/src/test/java/com/sgtech/pos/` and organized by layer:

- **Controller Tests**: `controller/AuthControllerTest.java`
  - Tests authentication endpoints
  - Validates request/response handling
  - Tests error scenarios
  - Uses MockMvc for HTTP request simulation
- **Service Tests**:
  - `service/SaleServiceTest.java`: Sale processing logic, tax calculation, coupon application
  - `service/RentalServiceTest.java`: Rental creation, customer management, due date handling
  - `service/ReturnServiceTest.java`: Return processing, refund calculation, overdue tracking
  - `service/EmployeeServiceTest.java`: Employee CRUD operations, password hashing
  - `service/InventoryServiceTest.java`: Inventory queries, low stock detection, quantity updates

- **Repository Tests**:
  - `repository/ItemRepositoryTest.java`: Custom query methods, item lookup by ID
  - `repository/CustomerRepositoryTest.java`: Customer search and creation
- **Integration Tests**:
  - `integration/SaleIntegrationTest.java`: Complete sale workflow from API call to database persistence

C.2.2 Test Configuration

All backend tests use the following configuration:

- **Test Profile**: `@ActiveProfiles("test")` annotation
- **Database**: H2 in-memory database (configured in `application-test.properties`)
- **JWT Configuration**: Test-specific JWT secret and expiration
- **Transaction Management**: `@Transactional` with rollback after each test
- **Test Data**: Fixtures created in `@BeforeEach` methods
- **Assertions**: JUnit 5 assertions with Hamcrest matchers

C.2.3 Test Coverage Statistics

- **Total Test Methods**: 30+ test methods
- **Service Layer Coverage**: 100% of business logic paths
- **Repository Layer Coverage**: All custom query methods tested
- **Controller Layer Coverage**: Authentication and error handling
- **Integration Coverage**: End-to-end workflows validated
- **Overall Backend Coverage**: Approximately 70-75%

C.2.4 Key Test Scenarios

1. **Authentication Tests**:
   - Valid login with correct credentials
   - Invalid login with wrong password
   - Invalid login with non-existent username
   - JWT token generation and validation
2. **Sale Processing Tests**:
   - Sale with single item

- Sale with multiple items
- Sale with coupon discount
- Tax calculation (8.25%)
- Inventory quantity updates
- Final total calculation

3. **Rental Processing Tests**:
    - Rental creation with new customer
    - Rental creation with existing customer
    - Due date validation
    - Rental total calculation
    - Inventory updates for rentals
4. **Return Processing Tests**:
    - Return of all rented items
    - Partial return of items
    - Overdue item handling
    - Refund calculation
    - Inventory restoration
5. **Inventory Tests**:
    - Item search by name
    - Low stock detection
    - Quantity updates
    - Item lookup by ID
6. **Employee Management Tests**:
    - Employee creation with password hashing
    - Employee update operations
    - Employee deletion
    - Username uniqueness validation

C.3 Frontend Test Suite

C.3.1 Test Structure

Frontend tests are located in `pos-frontend/src/__tests__/`:

- **Component Tests**: `Login.test.tsx`
    - Form validation

  - User input handling
  - Error message display
  - Navigation after successful login

- **Context Tests**: `AuthContext.test.tsx`
  - Authentication state management
  - Token storage and retrieval
  - User session management
  - Logout functionality

- **Service Tests**: `SalesPage.test.tsx`
  - API service mocking
  - Component interaction
  - User flow validation

C.3.2 Test Configuration

Frontend tests use:

- **Test Runner**: Vitest (replacing Jest)
- **Testing Library**: React Testing Library
- **Environment**: jsdom for DOM simulation
- **Mocking**: vi.mock() for service layer
- **Setup**: `src/test/setup.ts` with @testing-library/jest-dom

C.4 API Integration Tests

C.4.1 Test Scripts

Integration test scripts are located in `test-scripts/`:

- **PowerShell Scripts** (Windows):
  - `integration-test.ps1`: Complete workflow testing
  - `test-api.ps1`: Individual endpoint testing
  - `migrate-data.ps1`: Data migration validation

- **Bash Scripts** (Linux/Mac):
  - `end-to-end-test.sh`: Full system testing
  - `test-api.sh`: API endpoint validation

C.4.2 Integration Test Scenarios

1. **Authentication Flow**:
   - Login with valid credentials
   - Token retrieval and storage
   - Protected endpoint access
2. **Sales Workflow**:
   - Item selection
   - Sale creation
   - Inventory verification
   - Transaction persistence
3. **Rental Workflow**:
   - Customer creation/lookup
   - Rental creation
   - Due date setting
   - Inventory updates
4. **Return Workflow**:
   - Rental lookup
   - Return processing
   - Refund calculation
   - Inventory restoration

C.5 Test Execution

C.5.1 Running Backend Tests

Listing 5: Backend Test Execution

```
cd pos-backend
mvn test
```

C.5.2 Running Frontend Tests

Listing 6: Frontend Test Execution

```
cd pos-frontend
npm test
```

C.5.3 Running Integration Tests

Listing 7: Integration Test Execution

```
# Windows
.\test-scripts\integration-test.ps1

# Linux/Mac
./test-scripts/end-to-end-test.sh
```

C.6 Test Results Documentation

Detailed test results and coverage reports are available in:

- **Test Coverage Analysis**: `TEST_COVERAGE_ANALYSIS.md`
  - Coverage statistics by layer
  - Coverage gaps identified
  - Recommendations for improvement
- **Error Fixes**: `ERROR_FIXES.md`
  - Test configuration issues resolved
  - H2 database setup
  - JWT configuration for tests
  - Spring Data JPA method naming fixes
- **Testing Guide**: `TESTING_GUIDE.md`
  - How to run tests
  - Test environment setup
  - Troubleshooting test failures

C.7 Test Quality Metrics

- **Test Reliability**: All tests pass consistently
- **Test Isolation**: Tests run independently without side effects
- **Test Speed**: Fast execution with in-memory database
- **Test Maintainability**: Well-organized, documented test code
- **Coverage Goals**: Critical business logic paths fully covered

All tests are configured to run with H2 in-memory database and include comprehensive coverage of critical business logic paths, ensuring the reliability and maintainability of the reengineered system.