CSC 431

StockWise

<Group 5>

System Architecture Specification (SAS)

Eltonia Leonard
Everett Xu
Prateek Gupta

Version History

Version	Date	Author(s)	Change Comments
1.0	4/10/25	Eltonia	Added 1.1 (System Overview) and began creating the system diagram
1.1	4/11/25	Eltonia	Added 1.2 (System Diagram)
1.2	4/12/25	Eltonia	Added 1.3 (Actor Identification)
1.3	4/20/25	Everett	Added 1.4.1
1.4	4/21/25	Everett	Added 1.4.2 and 1.4.3
1.5	4/22/25	Everett	Added UML diagram for 1.4.1 and 1.4.2
1.6	4/22/25	Prateek	Added 2 and 3
1.7	4/22/25	Eltonia	Reformatted Document
1.8	4/22/25	Eltonia	Made slight changes to 2 and 3

Table of Contents

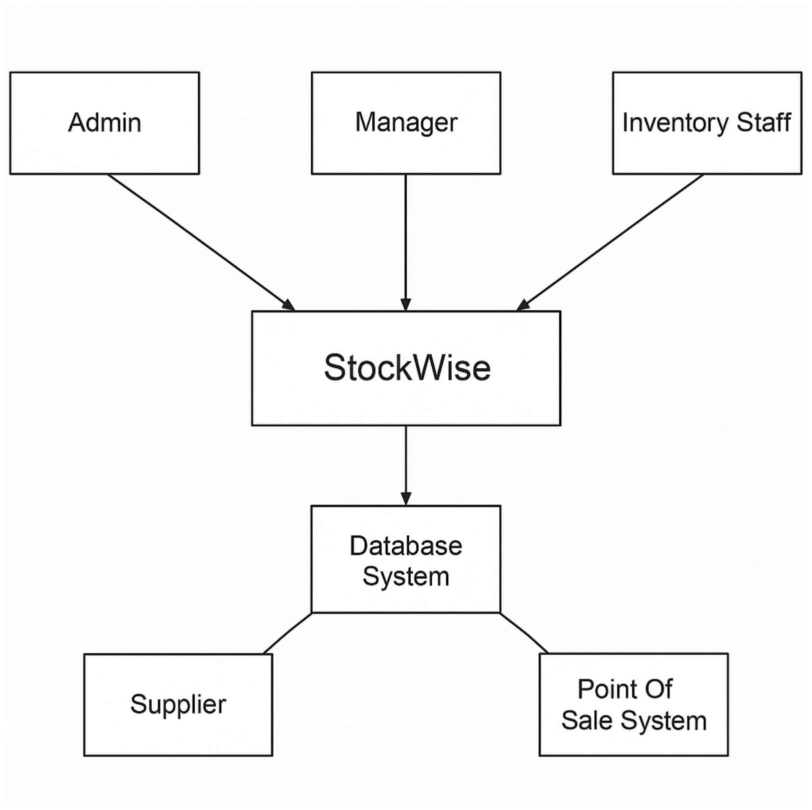
1. System Analysis

1.1 System Overview

StockWise is an advanced inventory management system designed to address the specific needs of small businesses. It offers a complete way to track stock levels by automating key tasks such as real-time inventory updates, sending low-stock alerts, and smoothly handling automatic order placements. With StockWise, business owners can access a comprehensive dashboard that offers clear insights into sales trends, stock changes, and supplier performance, enabling them to make smart, informed, data driven decisions.

The system will be built using a layered architectural approach. This modular design divides the application into distinct segments for user interaction, business logic, and data management. Such an approach supports scalability and ease of maintenance, ensuring that each component functions efficiently while maintaining overall system security.

1.2 System Diagram



1.3 Actor Identification

The StockWise system interacts with a range of actors, both human and external, to perform its core inventory management functions. These actors represent users or systems that either initiate processes, receive outputs, or exchange data with the system. Identifying these actors is essential for understanding the flow of operations, defining system boundaries, and establishing clear roles and responsibilities within the software environment. The following list outlines all key actors that connects with the system directly or indirectly:

Main Actors:

1. Admin - The Admin is responsible for managing and setting up the system. This actor manages user accounts, assigns appropriate roles and permissions, and ensures that system settings align with organizational needs. The Admin holds the highest level of access within StockWise, with full privileges across all functional modules.

2. Manager - The Manager is responsible for monitoring inventory performance and ensuring that stock levels are maintained appropriately. This actor reviews inventory reports, checks trends in sales and demand, and approves restock requests when needed. The Manager plays a key role in maintaining accurate inventory and supporting decision-making based on system insights.

3. Inventory Staff - Inventory Staff are responsible for handling day to day inventory operations. These users interact with the system to process transactions such as sales, returns, and restocks. They ensure that stock updates are entered correctly, and that inventory data remains accurate in real time.

External Actors:

4. Supplier - The Supplier receives automatic reorder requests when an item falls below its reorder threshold. It may also send order confirmations and updates, depending on the level of system integration. This actor supports the restocking process to keep inventory levels stable.

5. Point Of Sale System - The POS System is an external system that communicates directly with StockWise. It sends transaction data such as sales and returns, which trigger automatic inventory updates. This actor enables seamless real-time integration between front-end sales and back-end inventory tracking.

Other Actors:

6. Customer - Customers do not interact with the system directly, but they affect it through their purchases. Sales and returns initiated at the POS level by customers result in inventory changes and may trigger alerts or reorder actions.

7. Database System - The Database System stores all system data including inventory records, user profiles, transaction history, and reports. It ensures data consistency, enables secure access, and supports offline synchronization when connectivity is restored.

1.4 Design Rationale

1.4.1 Architectural Style

The fundamental architectural style selected for StockWise is Layered Architecture.

Rationale: This style is the most suitable for StockWise due to the system's inherent need for separation of concerns. Functionality naturally divides into four distinct logical groups: presentation (handling user interaction across web, desktop, and mobile), application logic (coordinating tasks like generating reports or processing orders), domain logic (implementing core business rules related to inventory, products, suppliers), and infrastructure (dealing with data persistence, network communication, and other technical details).

Benefits:

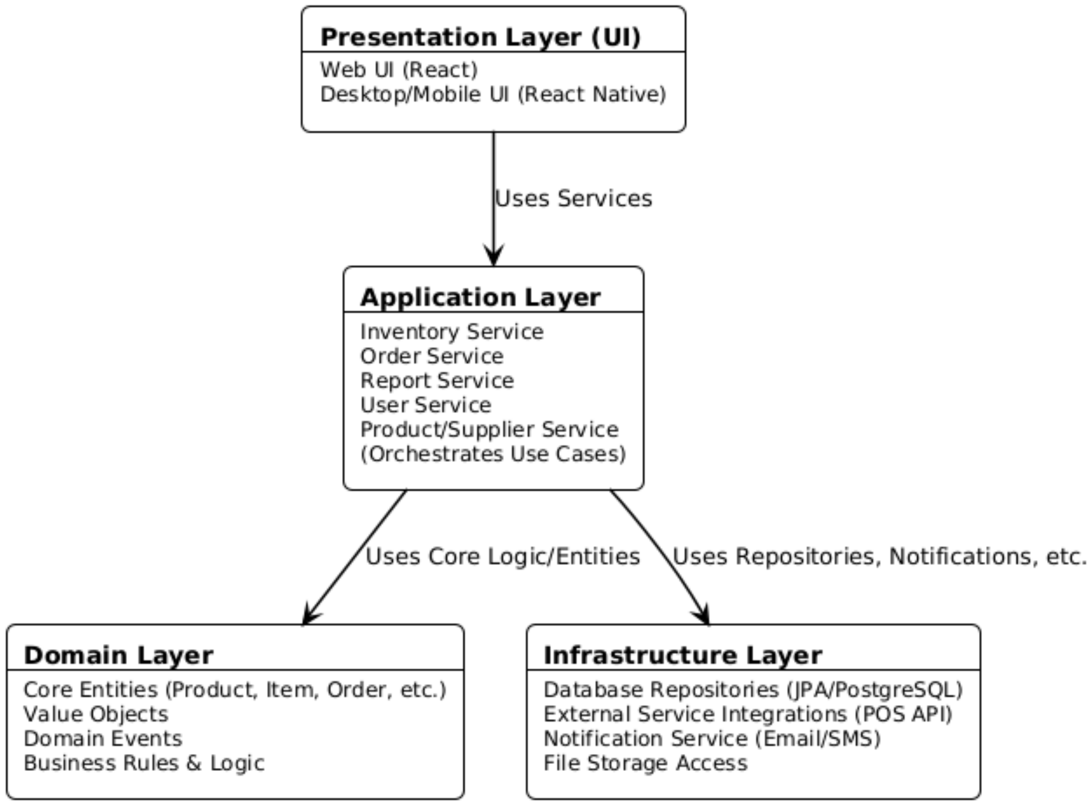
Maintainability: Layers isolate responsibilities. Changes to the user interface in the Presentation Layer, database technology in the Infrastructure Layer, or specific business rules in the Domain Layer can be made with reduced impact on other layers, simplifying updates and bug fixes.

Modularity & Reusability: Each layer encapsulates a specific type of functionality. The core Domain and Application layers can be reused across different Presentation layers (web, desktop, mobile), maximizing code reuse and ensuring consistent business logic application.
Testability: Layers can be tested independently. The Domain and Application layers can be unit-tested without requiring the UI or database, and data access can be mocked for testing business logic.

Technology Independence: The Business and Domain logic remain independent of specific UI frameworks or database technologies, providing flexibility for future technology evolution.
Structured Development: It provides a clear structure for organizing code and development effort, allowing teams to potentially specialize in different layers.

The Layered Architecture provides a clean, robust structure that directly supports the development of a complex, multi-platform system like StockWise, promoting long-term maintainability and scalability.

Diagram of Architectural Style

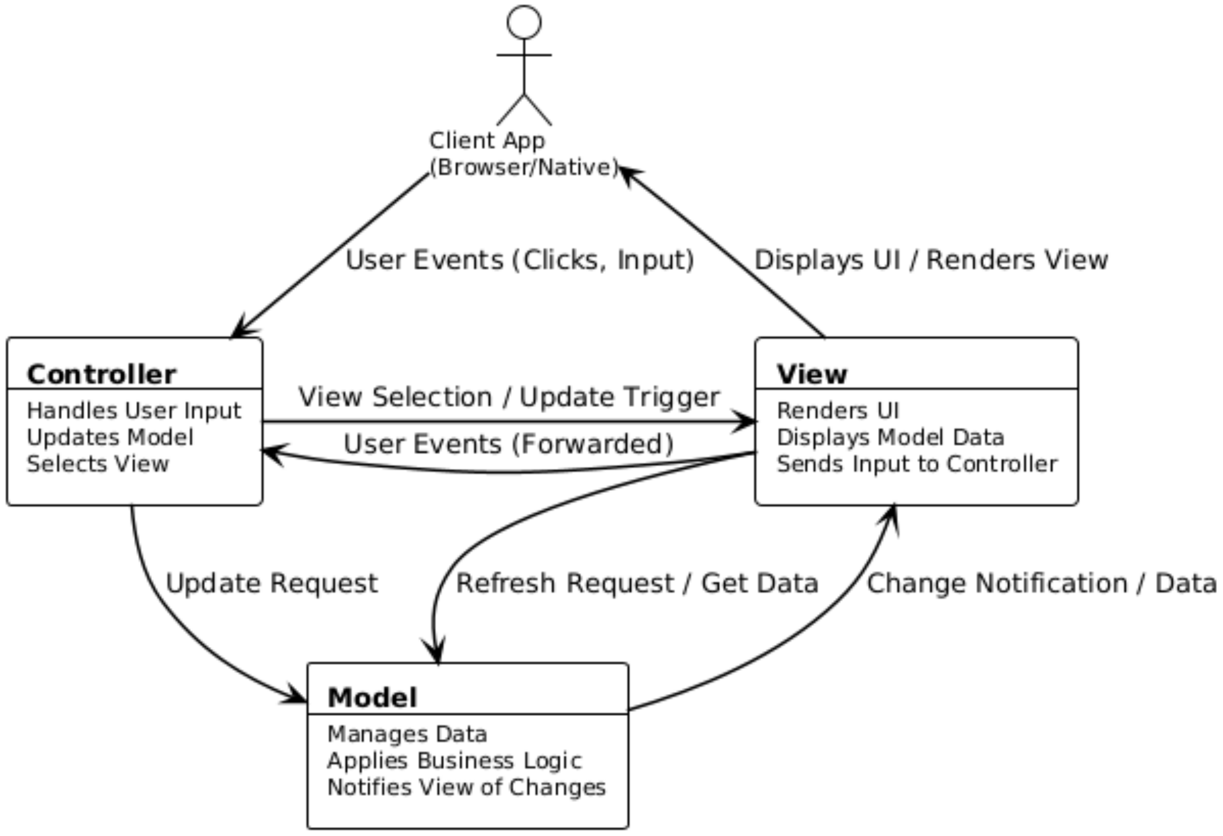


1.4.2 Design Pattern(s)

Within the Layered Architecture, specific design patterns are employed to solve recurring problems and structure components effectively:

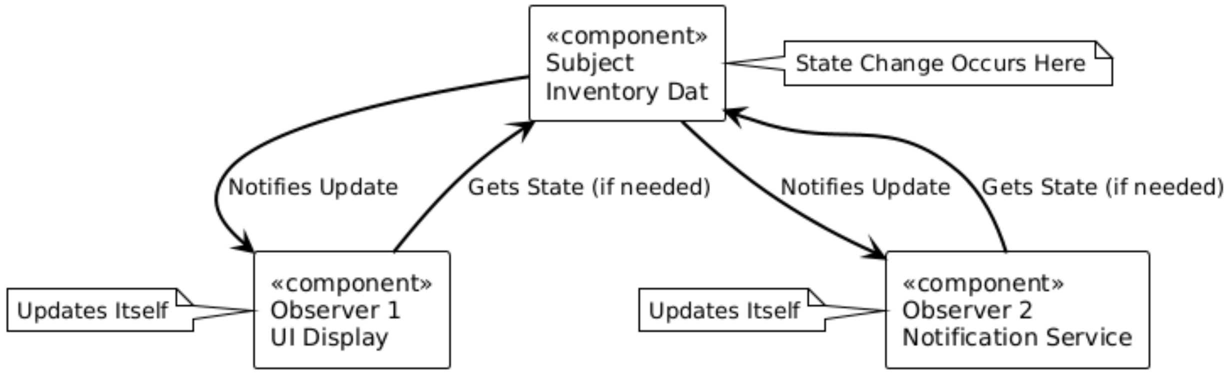
Model-View-Controller (MVC):

Rationale: Applied within the Presentation Layer for the web, desktop, and mobile interfaces. MVC is essential for structuring the UI code. It separates the concerns of data handling and presentation logic (Model) from the UI rendering (View) and user input processing (Controller). This separation is critical for managing the complexity of building and maintaining distinct interfaces for different platforms while ensuring a consistent user experience. It facilitates parallel development of UI and backend logic and makes the UI layer more adaptable to changes, such as implementing role-based UI variations.



Observer Pattern:

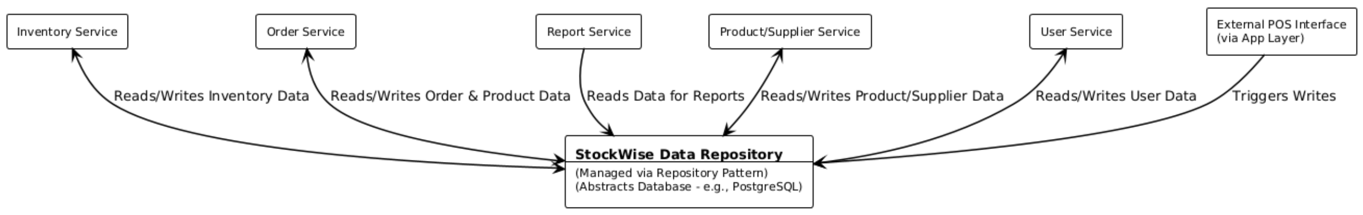
Rationale: This pattern directly addresses the need for live inventory updates and low-stock notifications. The inventory data or specific stock level monitors act as the 'Subject'. Various UI components across different user sessions, dashboard elements, and the notification service act as 'Observers'. When an inventory transaction occurs (sale, return, restock), the Subject notifies all registered Observers, which then update themselves (e.g., refresh a displayed stock count, trigger an alert). This achieves real-time updates efficiently and decouples the notification/display logic from the core inventory update mechanism within the Application/Domain layers.



Repository Pattern:

Rationale: Implemented within the Infrastructure Layer to manage data persistence. StockWise relies heavily on storing and retrieving data for products, inventory, suppliers, users, etc. The Repository pattern provides a clean abstraction over the data storage mechanism(s). It decouples the Application and Domain layers from the specifics of the database implementation (e.g., specific SQL syntax or NoSQL API calls), enhancing flexibility and maintainability. It centralizes data access logic, promotes consistency, simplifies the implementation of data queries (especially for reports), and significantly improves the testability of the Application and Domain layers by allowing data access to be easily mocked.

These patterns complement the Layered Architecture by providing proven solutions for structuring UI interactions, handling state propagation, and managing data access within their respective layers.


Inventory Service
Order Service
Report Service
Product/Supplier Service
User Service
External POS Interface (via App Layer)
Reads/Writes Inventory Data
Reads/Writes Order & Product Data
Reads Data for Reports
Reads/Writes Product/Supplier Data
Reads/Writes User Data
Triggers Writes
StockWise Data Repository
(Managed via Repository Pattern)
(Abstracts Database - e.g., PostgreSQL)

1.4.3 Framework

Specific, widely adopted frameworks are selected to implement the chosen architecture and patterns efficiently:
Backend Framework: Java with Spring Boot
Rationale: Spring Boot provides an excellent foundation for building the backend layers (Application, Domain, Infrastructure). Its strong support for dependency injection facilitates the creation of loosely coupled components across layers. Spring Data JPA simplifies the implementation of the Repository pattern, significantly reducing boilerplate data access code. Spring Security provides robust mechanisms for handling authentication and authorization based on user roles. Furthermore, Spring Boot's ease of configuration, embedded server options, and extensive tooling support streamlined development and deployment, including readiness for cloud environments. Java's performance and ecosystem are well-suited for a reliable inventory management system.

Frontend Framework (Web): React
Rationale: React's component-based model is ideal for building the modular user interfaces required for the web Presentation Layer. It encourages reusability and makes managing complex UIs more tractable. Its virtual DOM contributes to a responsive user experience. When combined with state management libraries (like Redux or Zustand), it effectively implements the Model and View aspects of the MVC pattern within the browser.

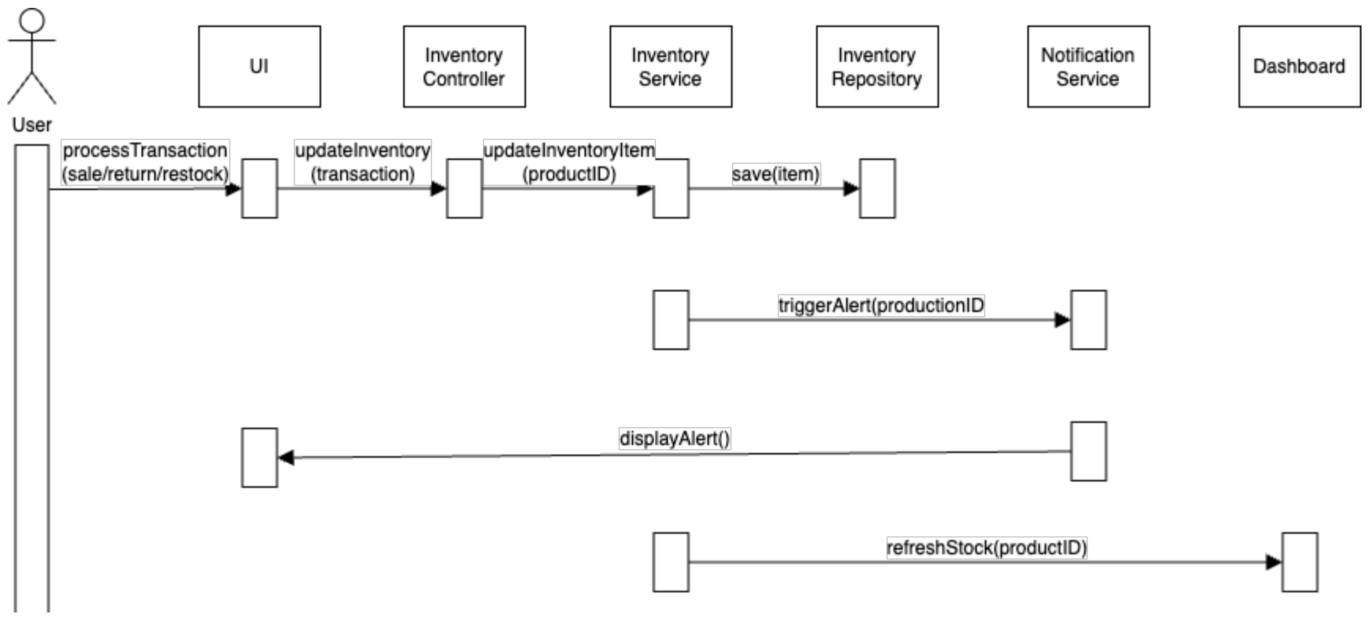
Frontend Framework (Desktop/Mobile): React Native
Rationale: React Native is chosen to deliver the desktop and mobile applications efficiently. It allows for significant code reuse from the React web application, drastically reducing the development time and effort needed to support all required platforms. This directly addresses the universal platform access requirement cost-effectively. React Native enables access to native device features (like the camera for barcode scanning) and allows for the creation of native-like user experiences across iOS, Android, Windows, and macOS from a predominantly single codebase, aligning well with project constraints.

Database & Data Access Framework: PostgreSQL with Spring Data JPA
Rationale: PostgreSQL is selected as the relational database system due to its robustness, open-source nature, extensive feature set, and strong support for transactional integrity, which is crucial for inventory data. To interact with the database, Spring Data JPA (Java Persistence API) will be used. It integrates seamlessly with Spring Boot and significantly simplifies the implementation of the Repository pattern by reducing boilerplate data access code and providing a consistent programming model for data manipulation. This combination ensures reliable data storage and efficient data access within the Infrastructure Layer.

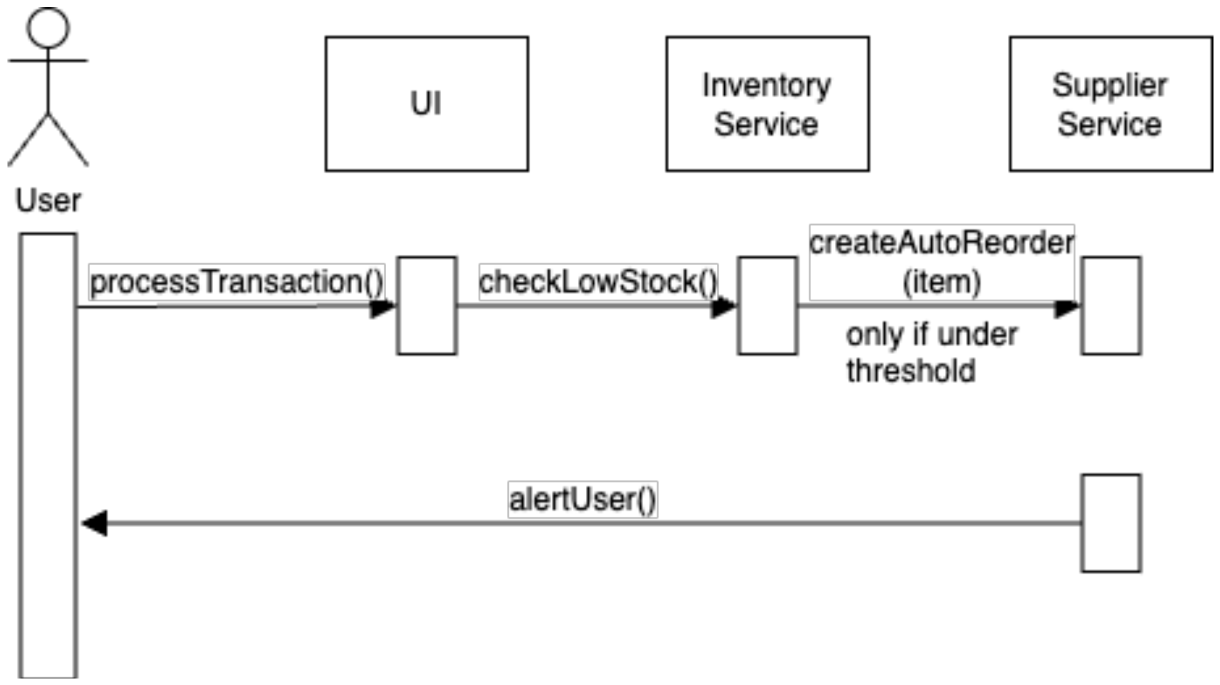
This framework stack provides a modern, productive, and cohesive environment for realizing the StockWise system based on the defined Layered Architecture and associated design patterns.

2. Functional Design

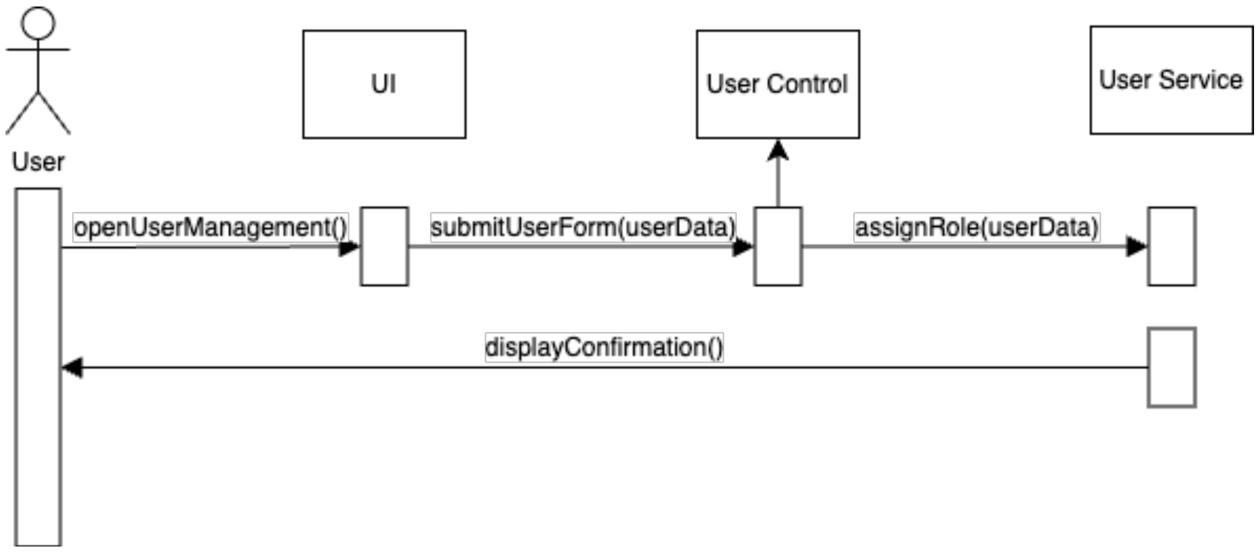
2.1 Live Inventory Updates and Monitoring



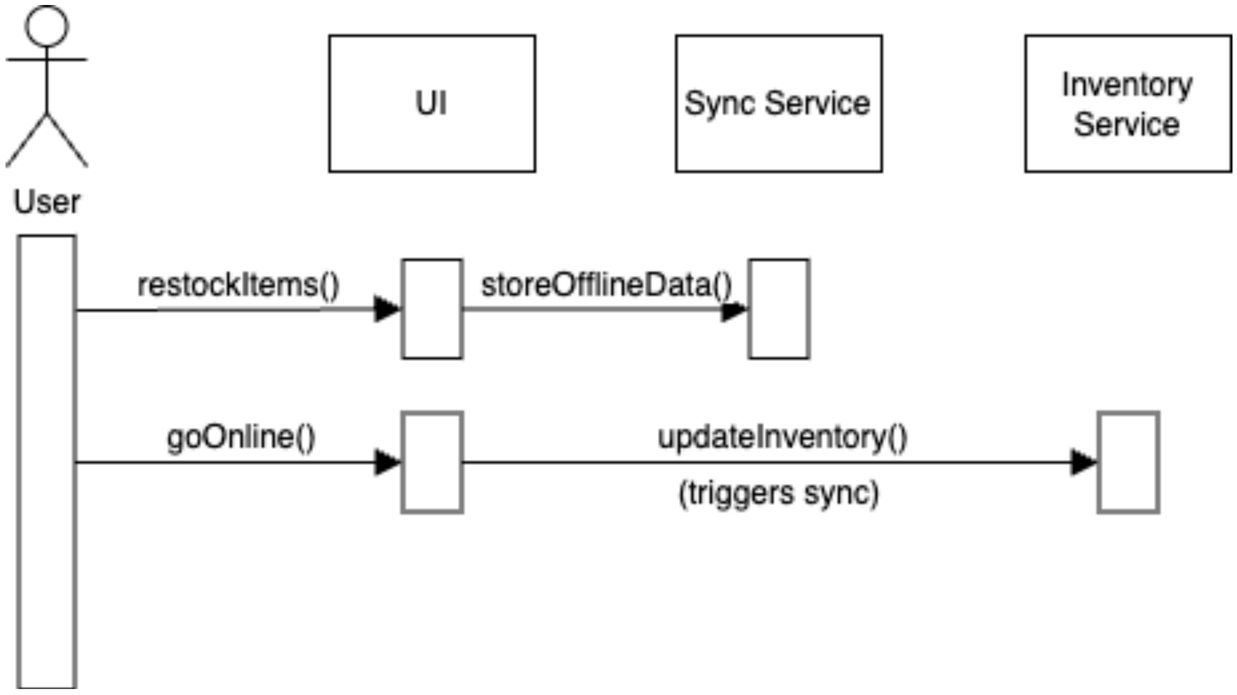
2.2 Low Stock Signal and Automatic Order Placement



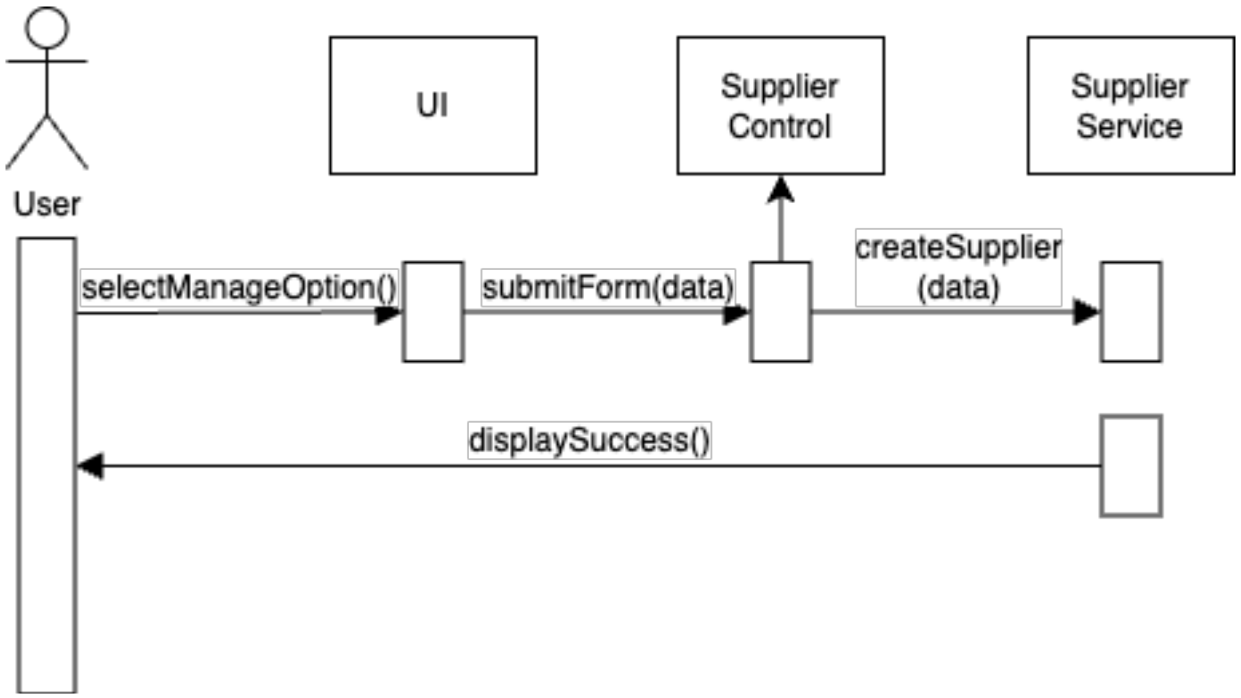
2.3 Account Access and Permission Management



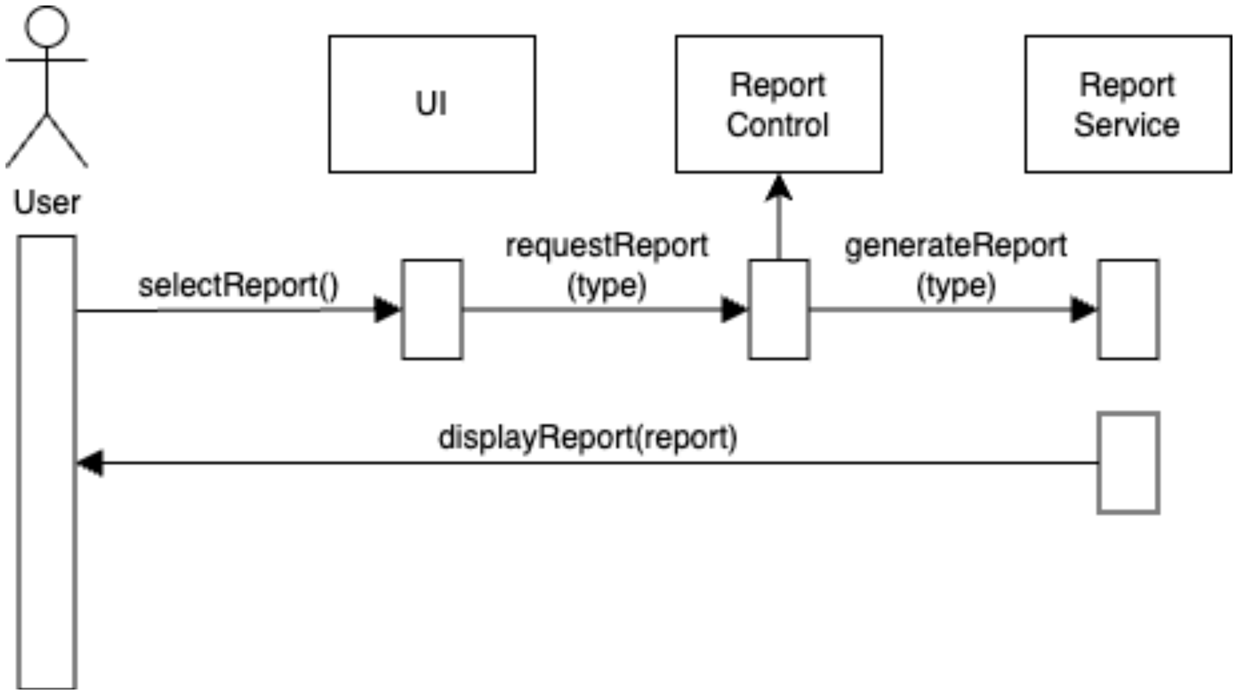
2.4 Offline Access and Data Synchronization



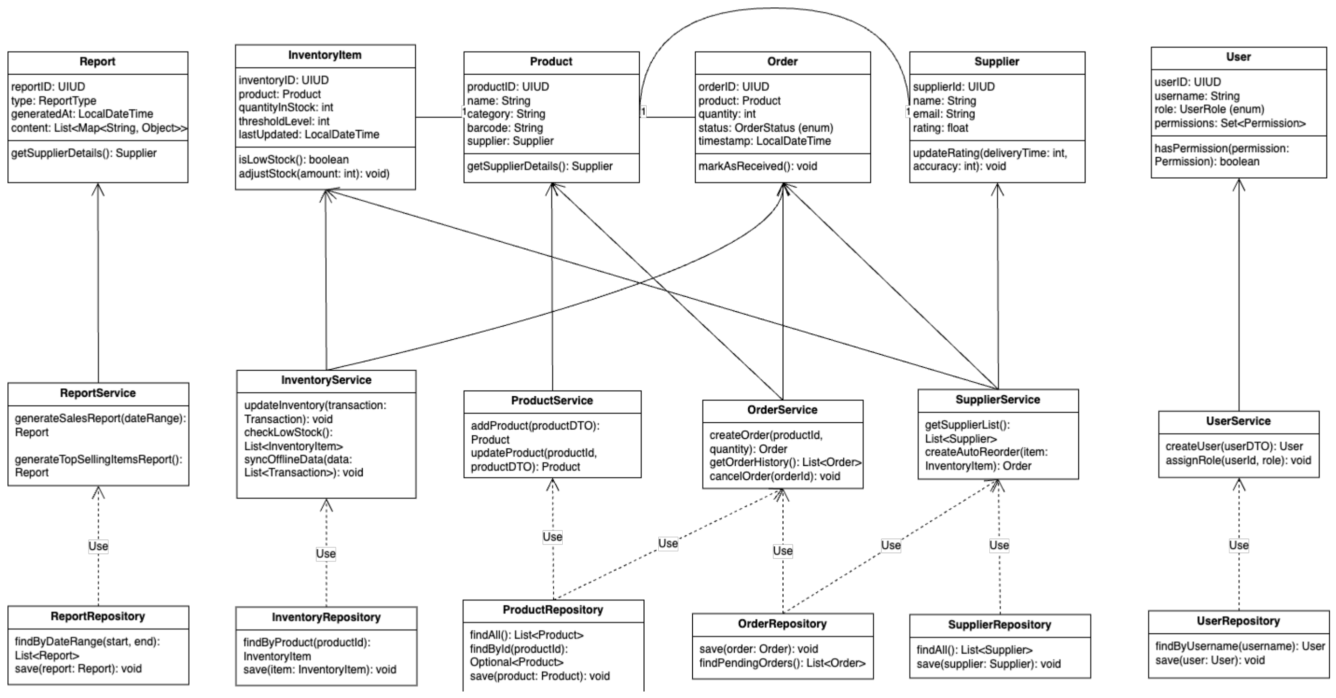
2.5 Product and Supplier Management



2.6 Statistical Insights and Reports



3. Structural Design


Report
reportID: UIUD
type: ReportType
generatedAt: LocalDateTime
content: List<Map<String, Object>>
getSupplierDetails(): Supplier
InventoryItem
inventoryID: UIUD
product: Product
quantityInStock: int
thresholdLevel: int
lastUpdated: LocalDateTime
isLowStock(): boolean
adjustStock(amount: int): void)
1
Product
productID: UIUD
name: String
category: String
barcode: String
supplier: Supplier
getSupplierDetails(): Supplier
1
Order
orderID: UIUD
product: Product
quantity: int
status: OrderStatus (enum)
timestamp: LocalDateTime
markAsReceived(): void
1
Supplier
supplierId: UIUD
name: String
email: String
rating: float
updateRating(deliveryTime: int, accuracy: int): void
User
userID: UIUD
username: String
role: UserRole (enum)
permissions: Set<Permission>
hasPermission(permission: Permission): boolean
ReportService
generateSalesReport(dateRange): Report
generateTopSellingItemsReport(): Report
InventoryService
updateInventory(transaction: Transaction): void
checkLowStock(): List<InventoryItem>
syncOfflineData(data: List<Transaction>): void
ProductService
addProduct(productDTO): Product
updateProduct(productId, productDTO): Product
OrderService
createOrder(productId, quantity): Order
getOrderHistory(): List<Order>
cancelOrder(orderId): void
SupplierService
getSupplierList(): List<Supplier>
createAutoReorder(item: InventoryItem): Order
UserService
createUser(userDTO): User
assignRole(userId, role): void
Use
Use
Use
Use
Use
Use
Use
Use
ReportRepository
findByDateRange(start, end): List<Report>
save(report: Report): void
InventoryRepository
findByProduct(productId): InventoryItem
save(item: InventoryItem): void
ProductRepository
findAll(): List<Product>
findById(productId): Optional<Product>
save(product: Product): void
OrderRepository
save(order: Order): void
findPendingOrders(): List<Order>
SupplierRepository
findAll(): List<Supplier>
save(supplier: Supplier): void
UserRepository
findByUsername(username): User
save(user: User): void

This diagram shows the main parts of the StockWise inventory system. It includes key classes like Product, Order, InventoryItem, Supplier, and User, along with their related services and repositories.

- Services (for example: InventoryService, OrderService) handle system actions like updating stock or placing orders.
- Repositories (for example: ProductRepository) save and load data.
- Each main class has attributes and methods that reflect what it does in the system.

The setup follows a layered structure, making it easier to manage and expand.