1. Problem Statement & Objectives

Problem Statement

Many businesses, especially small to medium-sized enterprises, struggle with inefficient inventory tracking, leading to issues such as stockouts, overstocking, inaccurate records, and delayed decision-making. Manual inventory management processes or outdated systems often result in human errors, poor visibility of stock levels, and an inability to scale operations effectively. The lack of a centralized, user-friendly, and automated system hinders operational efficiency and profitability.

Objectives

- Develop an Inventory Management System to automate inventory tracking and management processes.
- Provide real-time visibility into stock levels, product details, and transaction history.
- Enable efficient stock updates, order processing, and reporting for better decision-making.
- Design a scalable and maintainable system using the MVC architecture to separate concerns and improve development flexibility.
- Ensure the system is user-friendly, secure, and capable of handling multiple users with different roles (e.g., admin, warehouse staff).

3. Functional & Non-Functional Requirements

Functional Requirements

- **User Authentication**: Support role-based access (Admin vs. Staff) with secure login.
- **Inventory Management**: Enable CRUD operations (Create, Read, Update, Delete) for inventory items.
- **Order Processing**: Record stock inflows (e.g., supplier deliveries) and outflows (e.g., customer orders).
- **Search & Filter**: Allow users to search products by name, category, or ID and filter by stock levels.
- **Real-Time Updates**: Reflect inventory changes instantly across all user views.
- **Reporting**: Generate reports on stock status, transactions, or historical data.

Non-Functional Requirements

- **Performance**: Handle up to 1,000 concurrent users with response times under 2 seconds.
- **Scalability**: Support growth in inventory size (e.g., 10,000+ products) and user base.
- **Reliability**: Ensure 99.9% uptime with robust error handling.
- **Usability**: Provide an intuitive interface requiring minimal training.
- **Security**: Encrypt sensitive data (e.g., user credentials) and implement session timeouts.
- **Maintainability**: Use MVC to ensure code is modular and easy to update.

4. Software Architecture

High-Level Design

The Inventory Management System follows the MVC (Model-View-Controller) architecture, which separates concerns into three interconnected components:

Model:

- Represents the data and business logic.
- Components: Database (e.g., SQL-based like MySQL or PostgreSQL), Inventory Entities (e.g., Product, Order, User), and Data Access Layer.
- Responsibilities: Manages inventory data, enforces business rules (e.g., prevent negative stock), and handles persistence.

View:

- The user interface layer.
- Components: Web pages (e.g., HTML/CSS/JavaScript), dashboards, forms, and reports.
- Responsibilities: Displays inventory data, accepts user inputs, and visualizes reports.

Controller:

- Mediates between Model and View.
- Components: Application logic, request handlers, and API endpoints.
- Responsibilities: Processes user requests (e.g., add product, generate report), updates the Model, and refreshes the View.

Interactions

- User → Controller: A user interacts with the View (e.g., clicks "Add Product"), triggering a request to the Controller.
- Controller → Model: The Controller validates the request and updates the Model (e.g., inserts a new product into the database).
- Model → Controller: The Model confirms the update or returns data (e.g., updated stock list).
- Controller → View: The Controller sends the updated data to the View for display.

Architecture Style

MVC: Chosen for its separation of concerns, making the system modular, testable, and maintainable.

Layered Approach: Includes Presentation (View), Business Logic (Controller), and Data (Model) layers.

Technology Stack Example:

- **Frontend**: HTML, CSS, JavaScript (e.g., React or Angular for dynamic views).
- **Backend**: C# (ASP.NET MVC), Java (Spring MVC), or Python (Django).
- **Database**: Relational (e.g., MySQL, PostgreSQL) for structured inventory data.

Scalability Considerations

- Deploy on a cloud platform (e.g., AWS, Azure) with load balancing for high traffic.
- Use caching (e.g., Redis) for frequently accessed data like stock levels.