|  |
| --- |
| IMS |
| Technical design doc |
| Version: 1.0.0 |

|  |
| --- |
| Author Me  10-15-2024 |

Contents

[**1** **Introduction** 3](#_Toc179896574)

[**1.1** **Purpose** 3](#_Toc179896575)

[1.2 Scope 3](#_Toc179896576)

[1.3 Definitions and Acronyms 4](#_Toc179896577)

[2 System Overview 5](#_Toc179896578)

[2.1 System Architecture 5](#_Toc179896579)

[2.2 Key components 6](#_Toc179896580)

[2.2.1 Frontend (User Interface) 6](#_Toc179896581)

[2.2.2 Backend (Business Logic Layer) / Logic Tier 6](#_Toc179896582)

[2.2.3 Database (DBMS) 7](#_Toc179896583)

[2.2.4 Security and Authentication 7](#_Toc179896584)

[2.2.5 Notifications and Alerts 8](#_Toc179896585)

[2.2.6 Reporting and Analytics 8](#_Toc179896586)

[3 Architecture Design 9](#_Toc179896587)

[3.1 System Architecture Diagram 9](#_Toc179896588)

[3.2 Technical Architecture 9](#_Toc179896589)

[3.2.1 Frontend (Presentation Layer) 9](#_Toc179896590)

[3.2.2 Backend (Business Logic Layer) 10](#_Toc179896591)

[3.2.3 Database Layer (Storage) 11](#_Toc179896592)

[3.2.4 Security 12](#_Toc179896593)

[3.2.5 Performance Optimization 12](#_Toc179896594)

[4 Component Design 13](#_Toc179896595)

[4.1 User Interface 13](#_Toc179896596)

[4.1.1 Framework: 13](#_Toc179896597)

[4.1.2 UI Components: 13](#_Toc179896598)

[4.2 Backend Services 14](#_Toc179896599)

[4.2.1 Architecture 14](#_Toc179896600)

[4.2.2 API Design 14](#_Toc179896601)

[4.3 Business Logic 14](#_Toc179896602)

[4.3.1 Algorithms: 14](#_Toc179896603)

[4.3.2 Validation Rules: 14](#_Toc179896604)

[5 Data Design 15](#_Toc179896605)

[5.1 Database Design 15](#_Toc179896606)

[5.1.1 Database Type: 15](#_Toc179896607)

[5.1.2 Entity-Relationship Diagram (ERD): 15](#_Toc179896608)

[5.1.3 Data Flow Diagrams (DFD): 15](#_Toc179896609)

[6 Security Design 16](#_Toc179896610)

[6.1 Authentication and Authorization: 16](#_Toc179896611)

[6.2 Data Protection: 16](#_Toc179896612)

[7 Integrations points 16](#_Toc179896613)

[7.1 External Services: 16](#_Toc179896614)

[7.2 Internal Services: 17](#_Toc179896615)

[8 Performance considerations 17](#_Toc179896616)

[8.1 Scalability: 17](#_Toc179896617)

[8.2 Response Time: 17](#_Toc179896618)

[9 Deployment Strategy 17](#_Toc179896619)

[9.1 Environment Setup: 17](#_Toc179896620)

[9.2 Deployment Process: 18](#_Toc179896621)

[10 Testing Strategy 19](#_Toc179896622)

[10.1 Test Types: 19](#_Toc179896623)

[10.2 Test Environment: 20](#_Toc179896624)

[11 Conclusion 20](#_Toc179896625)

[12 Appendices 20](#_Toc179896626)

# **Introduction**

## **Purpose**

The purpose of this **Technical Design Document** is to provide a comprehensive description of the architecture, components, and technical specifications of the **Inventory Point of Sale (POS) System**. It outlines the system design, including both high-level and detailed descriptions of the system's components, how they interact, and the technical solutions chosen to meet business requirements.

This document is intended for developers, system architects, project managers, and technical stakeholders involved in the development, maintenance, and deployment of the Inventory POS system. It serves as a blueprint for implementing the system and ensures alignment between development and system requirements.

## Scope

The scope of this document covers the technical aspects of the **Inventory POS System**, including:

* **System architecture**: Describes the overall structure of the system and its components.
* **Data flow**: Outlines how data moves between different components of the system.
* **Database design**: Details the structure of the database, including tables, relationships, and constraints.
* **APIs and integrations**: Describe external services and internal APIs that the system interacts with.
* **User interfaces**: Provides an overview of the user-facing components of the system (e.g., cashier interface, inventory management UI).
* **Security**: Outlines the security measures to protect system data and ensure secure transactions.
* **Performance considerations**: Addresses performance optimizations and system scalability.
* **Error handling**: Describes mechanisms for dealing with system errors and exceptions.

## Definitions and Acronyms

* **POS**: Point of Sale – A system used in retail environments for managing sales transactions and inventory.
* **SKU**: Stock Keeping Unit – A unique identifier for each distinct product or item in inventory.
* **API**: Application Programming Interface – A set of tools and protocols that allow different software components to communicate with each other.
* **CRUD**: Create, Read, Update, delete – Refers to the basic operations on data in a database.
* **UoM**: Unit of Measure – The standard unit used to quantify items in inventory (e.g., kilograms, liters).
* **DBMS**: Database Management System – Software that provides an interface for interacting with databases, such as MySQL or PostgreSQL.
* **UI**: User Interface – The visual elements through which users interact with the system.
* **TDD**: Test-Driven Development – A software development process where tests are written before the code is implemented.

# System Overview

## System Architecture

The **Inventory POS System** is a distributed application designed to manage sales, inventory, and reporting in a retail environment. The system follows a **three-tier architecture** consisting of a **frontend (user interface) / presentation Tier**, a **backend (server logic) / logic Tier**, and a **database / Data Tier** for persistent storage. This architecture ensures scalability, separation of concerns, and maintainability. The communication between the frontend and backend is handled via RESTful APIs, and the backend communicates with the database through a Database Management System (DBMS).

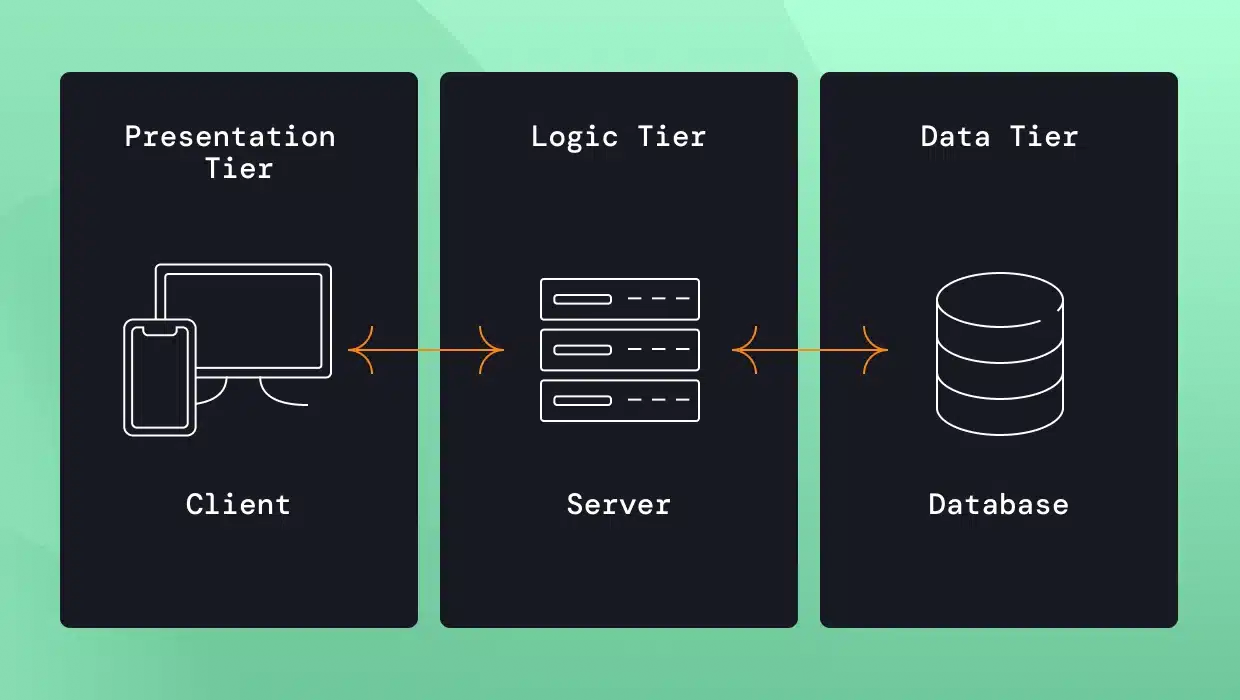
[](https://vfunction.com/blog/3-tier-application/)

Figure 2‑1

* **Frontend (User Interface)**: The frontend is the layer that interacts with the user, allowing for tasks such as sales transactions, inventory tracking, and report generation.
* **Backend (Business Logic Layer)**: The backend handles the business rules, data processing, and communication with the database.
* **Database**: The database stores information about products, inventory levels, sales transactions, pricing, and other relevant data.

## Key components

### **Frontend (User Interface)**

* **Description**: The frontend is the interface through which users interact with the POS Inventory system. This includes a web-based or mobile UI, designed for ease of use in a retail setting.
* **Components**:
  + **Sales Interface**: Used by sales personnel to enter sales.
  + **Inventory Management UI and Reporting Dashboard**: Enables admins to:
    - track stock levels, update inventory, and manage product categories
    - View insights into sales performance, inventory status, and more.
* **Technology Stack**:
  + Web: React.js or NextJS
  + Mobile: React JS or NextJS PWA
  + Communication with backend: RESTful APIs (using JSON)

### Backend (Business Logic Layer) / Logic Tier

* **Description**: The backend is responsible for processing requests from the frontend, applying business logic, interacting with the database, and ensuring system security.
* **Components**:
  + **Sales Management**: Manages the creation, update, and retrieval of sales transactions.
  + **Inventory Management**: Manages stock levels, product information, and pricing.
  + **Authentication and Authorization**: Handles user authentication, roles, and permissions.
  + **Notification System**: Sends alerts to the admin when stock levels fall below the reorder threshold.
* **Technology Stack**:
  + Server: Node.js (Express) / SailsJS
  + Communication with database: ORM (Prisma for Node.js) / MYSQL2

### **Database (DBMS)**

* **Description**: The database stores persistent information related to products, sales transactions, users, stock levels, and more. It ensures data consistency, integrity, and efficient access to data.
* **Components**:
  + **Product Table**: Contains details of each product such as name, SKU, price, and category.
  + **Sales Table**: Logs each transaction with a reference to products sold, quantities, and total sales amounts.
  + **Inventory Table**: Tracks current stock levels for each product and monitors when items reach reorder levels.
  + **User Table**: Stores user information, roles, and permissions.
* **Technology Stack**:
  + Database Management System (DBMS): (depending on ORM requirements)
  + Backup and Restore: Automated backup solutions to ensure data durability.

### **Security and Authentication**

* **Description**: This component ensures that all sensitive operations (e.g., inventory updates, financial transactions) are secure and accessible only by authorized users. Authentication is implemented via or JWT (JSON Web Tokens).
* **Components**:
  + **Role-Based Access Control (RBAC)**: Limits access based on user roles (admin, sales).
  + **Encryption**: All sensitive data (e.g., passwords, credit card information) is encrypted both at rest and in transit.

### **Notifications and Alerts**

* **Description**: The system sends notifications when important events occur, such as low stock levels, successful sales, or potential security breaches.
* **Technology Stack**:
  + Email: SMTP server integration (SendGrid)
  + Push Notifications: N/A
  + In-App Notifications: Real-time updates using WebSockets

### **Reporting and Analytics**

* **Description**: This component generates reports on inventory levels, sales performance, and other metrics, providing actionable insights to management.
* **Components**:
  + **Sales Reports**: Daily, weekly, monthly reports on total sales, popular products, and cashier performance.
  + **Inventory Reports**: Detailed reports on stock levels, reorder levels, and trends in stock depletion.
  + **Financial Reports**: Reports on revenue, profit margins, and overall financial health.
* **Technology Stack**:
  + Data visualization tools: D3.js, Chart.js
  + Report generation: JasperReports or similar tools

# Architecture Design

## System Architecture Diagram

The **System Architecture Diagram** for the **Inventory POS System** represents the high-level view of how various components interact with each other, ensuring seamless communication between the **frontend**, **backend**, and **database** layers. This architecture supports separation of concerns, scalability, and security. Below is a textual representation of the architecture, which can be converted into a graphical diagram:

## Technical Architecture

The technical architecture describes the **technology stack**, **frameworks**, and **deployment environments** used in building, running, and maintaining the **Inventory POS System**.

### **Frontend (Presentation Layer)**

* **Technology Stack**:
  + **Web Application**: Built using **React.js**, or **NEXTJS** for responsive user interfaces.
  + **Mobile Application**: Developed using **Progressive Web App technology**.
  + **Styling**: CSS Frameworks : **Tailwind CSS** , Material UI , raw CSS for styling.
  + **API Communication**: Frontend communicates with backend using **Axios** or the **Fetch API** or Redux for sending and receiving JSON-formatted data via REST APIs.
  + **State Management**: **Redux** (for React)) to manage the application’s state.
* **Deployment Environment**:
  + **Web Application**: Deployed on **Netlify** / **Vercel / Render**.
  + **Mobile Application**: PWA Packaged for iOS and Android platforms

### **Backend (Business Logic Layer)**

* **Technology Stack**:
  + **API Framework**:
    - **Node.js** with **Express.js** / SailsJS (for JavaScript applications).
    - **Typescript for typechecking**
  + **Security**:
    - **JWT** for securing API access and managing user sessions
  + Testing
    - JEST for unit testing
    - Cypress for endtoend
  + **Authentication and Authorization**:
    - **Role-Based Access Control (RBAC)** to restrict access to certain parts of the system based on user roles (e.g., admin, cashier, manager).
    - **JWT** for securing API access and managing user sessions
  + **Notification System**:
    - **Email notifications** (**SMTP servers** like SendGrid).
    - **Real-time alerts** (via **WebSockets** ).
  + **Error Handling and Validation**:
    - Global error handling middleware
    - input validation using libraries like **Joi** or **ExpressValidator.js**.
* **Deployment Environment**:
  + **Server Deployment**:
    - **Render** for hosting the backend server.
  + **Containerization**:
    - **Docker** for containerizing the backend service to ensure consistency across environments.
  + **Version Control**:
    - Hosted on **GitHub** with CI/CD pipelines for continuous integration and automated deployment.

### **Database Layer (Storage)**

* **Technology Stack**:
  + **Database Management System (DBMS)**:
    - **MySQL** or **PostgreSQL** for relational data storage
  + **Object-Relational Mapping (ORM)**:
    - **Prisma** (for Node.js)
  + **Backup and Recovery**:
    - N/A
  + **Data Structure**:
    - **Tables**: Product, Sales, Inventory, Users, Categories.
    - **Relationships**: Properly normalized database with foreign key relationships between tables (e.g., product, sales, and inventory tables).
    - **Triggers for logging**
* **Deployment Environment**:
  + **Database Hosting**:
    - **Render container, Vercel**
  + **Backup Strategies**:
    - Automated backup systems (Daily, weekly backups) with replication in case of failure.

### **Security**

* **Encryption**:
  + **TLS/SSL** encryption for all data transmitted between the frontend and backend.
  + **Encryption at rest** for sensitive data stored in the database (e.g., user passwords using bcrypt).
* **API Security**:
  + **JWT**-based authentication for securing APIs.
  + Role-based access controls to ensure only authorized users can perform certain actions.
* **Firewall and Intrusion Detection**:
  + Use of **Cloudflare** to prevent unauthorized access and DDoS attacks.

### **Performance Optimization**

* **Caching**:
  + Use of **Redis** or **Memcached** for caching frequently accessed data to reduce database load.
* **Load Balancing**:
  + **Nginx** for distributing incoming requests across multiple backend servers to ensure high availability.
* **Scalability**:
  + **N/A**

# Component Design

## User Interface

### Framework:

* **React.js** is chosen as the UI framework for the **Inventory POS System**. React enables the creation of reusable components, provides efficient updates through its virtual DOM, and integrates seamlessly with modern front-end development tools.

### UI Components:

* **Navigation Bar**: Provides navigation options for different modules like Orders, Sales, Inventory, and Reports.
* **Product List**: Displays the available products with details such as name, category, price, stock level, and reorder status.
* **Sales Form**: Allows users to input product details and units sold, fetches prices from the ProductPriceTable, and calculates total cost dynamically.
* **Order Management**: Enables users to view, fulfill, update, or delete orders and products. Handles batch and individual product updates.
* **Notification Center**: Displays real-time notifications about reorder levels, successful transactions, or errors (e.g., out-of-stock alerts).

## Backend Services

### Architecture

* The backend follows a **RESTful service architecture** built with **Node.js** and **Express.js**. It exposes API endpoints for creating, retrieving, updating, and deleting orders, products, and stock levels. The service handles business logic, validation, and communication with the database.

### API Design

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Endpoint | Method | Description | Request Format | Response Format |
|  |  |  |  |  |
|  |  |  |  |  |

## Business Logic

### Algorithms:

* **Order Processing Algorithm**: When an order is submitted, it checks if the requested quantity is available in stock. If valid, it reduces the stock by the order quantity, and if the stock reaches the reorder level, it triggers a notification.
* **Inventory Update Algorithm**: When a new shipment arrives or sales are recorded, the system updates the stock table, recalculating quantities and alerting the admin if stock levels fall below predefined thresholds.

### Validation Rules:

* **Product Price Validation**: Ensure that the price entered matches the unit price for the given product.
* **Stock Validation**: Ensure the requested sale quantity does not exceed the current stock.
* **Form Validation**: Ensure all required fields (e.g., product name, quantity, unit of measure) are filled before form submission.

# Data Design

## Database Design

### Database Type:

* The database used for this system is **MySQL**. It supports relational data storage, strong consistency, and complex queries.

### Entity-Relationship Diagram (ERD):

The ERD defines the relationships between entities like **Orders**, **Products**, **Sales**, **Stock**, and **Categories**:

* **Order (1-to-Many with Products)**: Each order can contain multiple products.
* **Product (1-to-1 with Stock)**: Each product has a corresponding stock entry.
* **Category (1-to-Many with Products)**: Each category can have multiple products, with nested subcategories.

### Data Flow Diagrams (DFD):

The **Data Flow Diagram (DFD)** illustrates how data moves through the system:

* **User Inputs**: Product details, sales data.
* **API Layer**: Receives requests, interacts with the database, and returns results.
* **Backend**: Handles validation, processes data, updates records.
* **Database**: Stores all relevant data, including product prices, sales, and stock levels.

# Security Design

## Authentication and Authorization:

* The system uses **JWT (JSON Web Token)** for stateless authentication. Users log in with credentials, and a token is generated for subsequent API requests.
* **Role-based access control (RBAC)** restricts certain operations to authorized users, such as only admins being able to update prices or manage stock levels.

## Data Protection:

* **Encryption**: All sensitive data, including user passwords, is encrypted using **bcrypt**.
* **TLS/SSL**: Ensures that all data transferred between client and server is encrypted.
* **Access Control**: Only authorized personnel can access sensitive data like sales records, stock levels, and product pricing.

# Integrations points

## External Services:

* **Payment Gateway Integration**: If required, the system can integrate with external payment services (e.g., **Stripe**, **PayPal**) to handle transactions.
* **Email Notifications**: Integrated with **SendGrid** or similar services for email-based notifications to alert users and admins about stock levels or important system events.

## Internal Services:

* **Notification Service**: Internal service for handling system notifications and alerts based on stock levels, order processing, and sales updates.
* **Logging Service**: Records events like stock updates, sales transactions, and user activity.

# Performance considerations

## Scalability:

* The system is designed to scale horizontally with the addition of more instances of the backend service and database replicas.
* Use of **Redis** for caching frequently accessed data to reduce database load.

## Response Time:

* Expected response times are under **200ms** for critical operations like order submissions, stock updates, and fetching product prices.
* Use of **lazy loading** for large datasets to reduce initial page load time.

# Deployment Strategy

## **Environment Setup:**

* **Development Environment:**
  + Purpose: Used by developers for building and testing features.
  + Tools: Describe the tech stack (e.g., local servers, databases, IDEs, Docker, VMs).
  + Configuration: Include settings like environment variables, API keys, etc.
* **Staging Environment:**
  + Purpose: Mirrors the production environment for final testing.
  + Tools: Include tools similar to production (e.g., containerization, cloud setup).
  + Testing Focus: Integration, performance, and user acceptance testing (UAT).
* **Production Environment:**
  + Purpose: Live system for end-users.
  + Tools: Specify cloud services or servers (e.g., AWS, Azure), monitoring tools, and security.
  + Configuration: Final configuration settings and hardening steps (e.g., database replication, caching).

## **Deployment Process:**

* **Version Control and CI/CD:**
  + Process: Describe how the code is pushed, reviewed, and merged using version control (e.g., Git, Bitbucket).
  + CI/CD Pipeline: Define the Continuous Integration and Continuous Deployment pipeline (e.g., Jenkins, GitHub Actions).
  + Automation: What is automated (build, test, deploy) and what requires manual intervention.
* **Rollout Strategy:**
  + **Blue-Green Deployment:** Two environments (one live, one for staging) for seamless switching.
  + **Canary Release:** Incremental release to a subset of users before full deployment.
  + **Rollback Plan:** Steps to revert to a stable version in case of failure.
* **Monitoring & Logging:**
  + Define tools for monitoring (e.g., Prometheus, Datadog) and logging (e.g., ELK stack) during and after deployment.

# **Testing Strategy**

## **Test Types:**

* **Unit Testing:**
  + Scope: Testing individual components or units of the system.
  + Tools: Describe frameworks used (e.g., Jest, Mocha, JUnit).
* **Integration Testing:**
  + Scope: Testing how different modules or services work together.
  + Tools: Mention tools and services (e.g., Postman, Selenium, SOAP UI).
* **End-to-End Testing:**
  + Scope: Simulating real user scenarios from start to finish.
  + Tools: Cypress, Selenium, or other e2e testing frameworks.
* **Performance Testing:**
  + Scope: Measuring system performance under different loads.
  + Tools: JMeter, Locust.
* **Security Testing:**
  + Scope: Identifying security vulnerabilities.
  + Tools: OWASP ZAP, Burp Suite.
* **User Acceptance Testing (UAT):**
  + Scope: Verifying that the system meets user requirements and business goals.
  + Participants: Usually performed by actual users or stakeholders.

## **Test Environment:**

* **Setup:**
  + Replicate production environment as closely as possible for testing (hardware, databases, network, etc.).
  + Include mock services for testing external dependencies (e.g., APIs, third-party services).
* **Tools:**
  + Virtualization or containerization tools (e.g., Docker, Kubernetes) for isolated test environments.
  + Automated testing tools and frameworks as mentioned above.

# **Conclusion**

* **Summary:** Recap the main aspects of the system, including key design decisions, technologies used, and potential risks.
* **Next Steps:**
  + Outline any further work or recommendations, such as additional features to be developed, testing phases, or deployment schedules.
  + Include timelines, if applicable.

# **Appendices**

* **Diagrams:**
  + Include architecture diagrams, data flow diagrams, or deployment workflows.
* **References:**
  + List any external resources or documentation referenced.
* **Additional Information:**
  + Any data definitions, API contracts, or detailed code snippets that support the main content.