# ABC’s Inventory Management

## Architecture Notebook

## Purpose

The purpose of this document is to describe the architecture philosophy, decisions, constraints, justifications, significant elements, and other overarching aspects of ABC’s Inventory Management System that shape the design and implementation of the system. The main purpose of this document is to propose an architecture for the system to be built which will help fulfil the requirements. This document will firstly describe the goals and philosophy of the architecture. This will be followed by any assumptions and dependencies effecting the architectural decisions. The document will then describe the architecturally significant requirements, decisions, constraints and justifications made. Different architectural mechanisms driving the design and implementation will also be described in this document. Finally, the document will present the key abstractions, architectural framework and architectural views of the system.

## Architectural goals and philosophy

The architecture to build the system will be based on a multi-tiered client server architecture. The main architectural goal of ABC’s Inventory Management System is to provide a minimal graphic inventory management application which satisfies the functional and non-functional requirements for the system. The system needs to be built from scratch since the business is new and has never used a digitized online inventory management system. Moreover, the system should be able to handle at least 1000 requests per hour. The system must also be strong and avoid crashes as much as possible as this might affect the daily business in store and the warehouse. The wait time for users for any action should not be more than 5 seconds. Scanners and printers are essential part of the business hence the system, the system should be able to access these devices to scan barcodes, print documents and so on. Multi-tiered architecture will help achieve all these requirements as it is flexible. The justification for all architectural decisions are described in detail below.

## Assumptions and Dependencies

The following are the assumptions and dependencies that drive the architectural decisions.

* All product and current inventory data will be available during the implementation of the system.
* Store and Warehouse employees have basic knowledge about computers and windows operating system.
* Project team members are familiar with the Unified Process.
* Project team members are familiar with core Java, Java EE, XML, JavaScript, MySQL and Restful API.
* Computers with enough capabilities are available to deploy the system.
* Physical devices like scanner and printers are available.

## Architecturally significant requirements

The following are the architecturally significant requirements of the system:

**Performance**

The system must have a reasonable response time. Lagging of the program should be avoided. Queries should be processed in a reasonable time (less than 3 seconds). Processing time must be less than 3 seconds. During high traffic, the processing time should be 5 seconds at most.

**Security**

The system must have login requirements for the users. The system should only accept new passwords that are at least 8 characters in length and that have digits or special characters in them. Similarly, the store staff must not have admin functions (high level functionalities like adding new user, delete stock, add stock and so on). The warehouse staff should have more functionalities like manage user and manage stock. The system should logout automatically after 5 minutes of inactivity.

**Availability**

The system should be available 24/7. Store and warehouse staffs will need to use the system to check stock level, stock availability and status at any time, the system should be always available to deliver. The server and database must be deployed to the cloud so that it can be accessed from any location at any time. If maintenance needs to be made, a notice must be sent to all staff and should be done outside business hours so that the business is not highly affected during system maintenance.

**Reliability**

The system must be able to generate real time data. The system must make changes to the database accurately and give out appropriate messages when something goes wrong. The system can have at most 100 hours of downtime per year. After breakdowns, the system must be able to restart and continue normal functions.

**Usability**

The system should have a very basic and consistent UI. The system must have a white or gray background while buttons and links should have black font color so that these elements are very clear to see. The system must be available in English. The system must be displayed full screen when launching the application by default. However, the system should allow users to change the size. The system must display appropriate messages. The system should follow Neilson’s 10 usability principles to improve usability.

**Capacity**

The system must be able to handle 100s of requests per hour. The business still being of small scale, the system will not be swamped with requests from different stores. The system should be able to handle the order, delivery and request handling. However, in case of a business growth in the next 3 years, the system should be able to handle at least 1000 requests per hour. Similarly, the system should be able to store data and information of all products, stores and the warehouse.

## Decisions, constraints and justifications

The following are the decisions, constraints, and justification regarding the architectural approaches:

|  |  |
| --- | --- |
| Decisions or Constraints | Justifications |
| Use of minimum UI design and decorations following Nielsen’s 10 usability heuristics | * Users will find it easy to navigate through the application improving the **usability** of the application. * Less time can be spent on beautifying the user interface and more time can be spent on solving the business issues the application is designed to solve. * Minimal display will make improve the application’s **performance**. |
| Use of Apache Tomcat server | * The Apache Tomcat server will be used on the business logic layer making it free from the data layer or the user interface layer which improves the **reliability, capacity and availability** of the application. This server will run the server-side application which then will be deployed to the AWS cloud. |
| Use of multitier architecture | * Three tier application architecture boosts the application’s flexibility since each tier can be managed and maintained independently. * It also improves the application’s availability since each component is separated from each other. The server and the database will be deployed to the AWS cloud which allows the system to be available even during some maintenance or if a particular component of the client application fails. * Using the layered architecture and following proven security protocol improves the security of the application to the highest of standards. Each layer will be following best practices for **security.** Moreover, the separation of components protects each component even if one gets compromised. * Multitier architecture might affect the **performance** of the system. However, this performance drawback will not be drastic as the system is still small scale which does not need to process massive data. |
| Using MySQL relational database in the Data Layer | * The use of MySQL will allow enough space for the system to store data about the product, stores and the warehouse which increases the capacity of the system. Moreover, the data stored will be independent from other layers therefore even if the interface or business layer is modified, the data will not be affected. |

## Architectural Mechanism

The following are the architectural mechanisms of the system with a brief description. The mechanisms have been ordered according to their importance:

|  |  |  |
| --- | --- | --- |
| Mechanism | State | Description |
| Performance Mechanism | Implementation | The decision to use the layered architecture (3-layered architecture) will make the system more flexible and maintainable. However, this approach affects the performance of the system. The benefits provided by this architecture exceeds the drawbacks. Layered architecture helps fulfill other non-functional requirements directly or indirectly which have been described below. Moreover, the performance issues that arises from the architecture can be compensated using other methods. For instance, proven best programming practices and structure can be used to improve overall performance of the system. An MVC pattern can be used to build client-side application which structures the code and ultimately helps in the overall performance of the system. Similarly, deploying the server in a trusted third-party cloud like AWS with proven capabilities and minimal graphics in the user interface layer can be done to improve the performance of the system. |
| Availability Mechanism | Implementation | The system must be available for staff to check stock content in each location, send stock to different location and accept stock from another location. Therefore, availability is a crucial requirement for the system. A multi-layered approach helps to fulfil this as multi layered architecture helps to prevent a single point of failure in a system. The server-side application and the database will also be deployed to the AWS cloud making both the server and the database online 24/7 therefore allowing any client application access it at anytime from anywhere unless a specific IP address is blocked. Similarly, since classes for each layer are separated from each other, diagnosing the point of failure will be easier decreasing the effects of a problem. This will help decrease the downtime of the system when a problem occurs therefore increasing the availability. Replication of core modules can also be deployed in different servers to increase availability. However, the business being small there are cost constraints which prevents the use of this tactic for increasing availability. |
| Reliability Mechanism | Implementation | Reliability is a major requirement for any system. To fulfil this requirement, redundancies can be introduced in core modules. To be more specific, in core modules which are absolutely essential for core system to work, redundancies (creation of parallel paths) can be developed. This however effects the performance of the system. Using layered architecture along with deliberately introducing parallel paths could significantly affect the performance of the system and increases cost because of greater programming time. Therefore, in this case using a reliable application server for processing data and requests for the business in the business logic layer can be used to improve reliability. Similarly, various unit testing and integration testing will be conducted for each functionality of the system to check the system’s reliability. |
| Capacity Mechanism | Implementation | The use of multi layered architecture allows data access and data to be stored in a separate layer. MySQL RDBMS must be used in the data access layer to store data. The use of a reliable application server in the business logic layer and using programming best practices like closing a connection to database once connection is not required allows the system to handle required number of requests for the business in the next 3 years therefore fulfilling the capacity requirements of the system. |
| Security Mechanism | Design | A multi-layered approach allows the system to be more secure as mentioned above in the Decision, constraints and justification section. Different layer must interact with each other in multi-layer architecture for the application to run smoothly. Authentication must be required for data to be communicated between two tiers. This can be achieved using various mechanisms like IP validation, SSL and so on. However, since the system in this case does not require maximum security and applying it will raise the cost because of the increase in programming time, this can be skipped. A simple login form including username and password to access functionalities should fulfil the security requirements for the system. Similarly, during product deployment in the cloud, access can be granted to certain IP addresses belonging to the company using Amazon AWS tools. Similarly, user interface must be limited to the UI layer of the architecture. The back end of the login form must be developed in the business logic layer using appropriate APIs like REST. Likewise, all users must not be allowed to make crucial changes to data and the meta data. Therefore, functionalities for each staff type (warehouse and store staff) must be separated in the business logic layer. |
| Usability Mechanism | Design | The UI of the system must be limited to the user interface layer of the architecture. The programming practice for this layer must support Nielsen’s usability principles. Minimal graphic must be used in the UI layer to improve the performance of the system and save time which decreases cost. |

All in all, the layered architectural approach allows the system to fulfil the system requirements in one way or other with some compromises. This approach also makes the system more scalable and flexible. Therefore, if the system needs to be upgraded in the future because of rapid business growth even though it is not a requirement now, it can be done so with much ease. Therefore, a multi layered architecture more specifically the 3-layered architecture is the best architectural approach to fulfil the requirements for this system.

## Key Abstractions

The following are the key abstractions of the system:

* Warehouse – This represents the warehouse that distributes the stock to stores.
* Store – This represent the stores where product is sold.
* Product – Items that are sold, requested to be transferred from warehouse and sent through deliveries.
* User – Warehouse and store staff that will be using the system.

## Layers or architectural framework

The following diagram represents the layers of the system:

User Interface Layer

Business Logic Layer

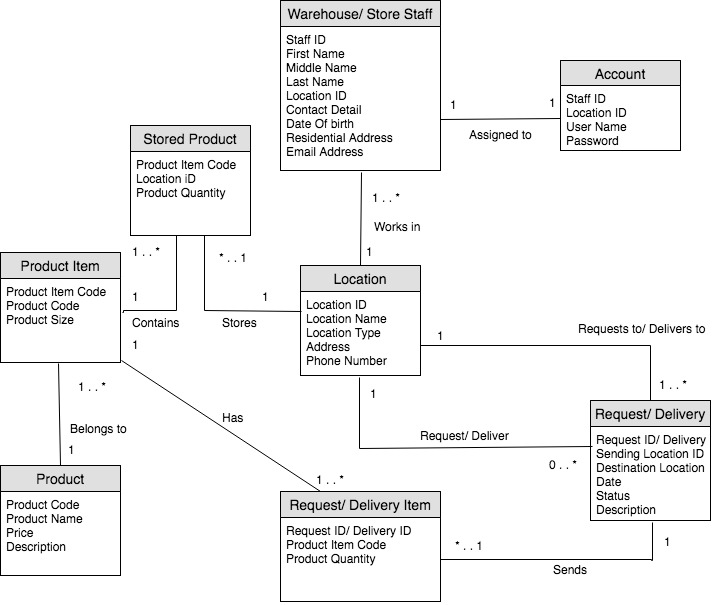
Data Layer

## Architectural Views

The following are the architectural views that will describe the software architecture:

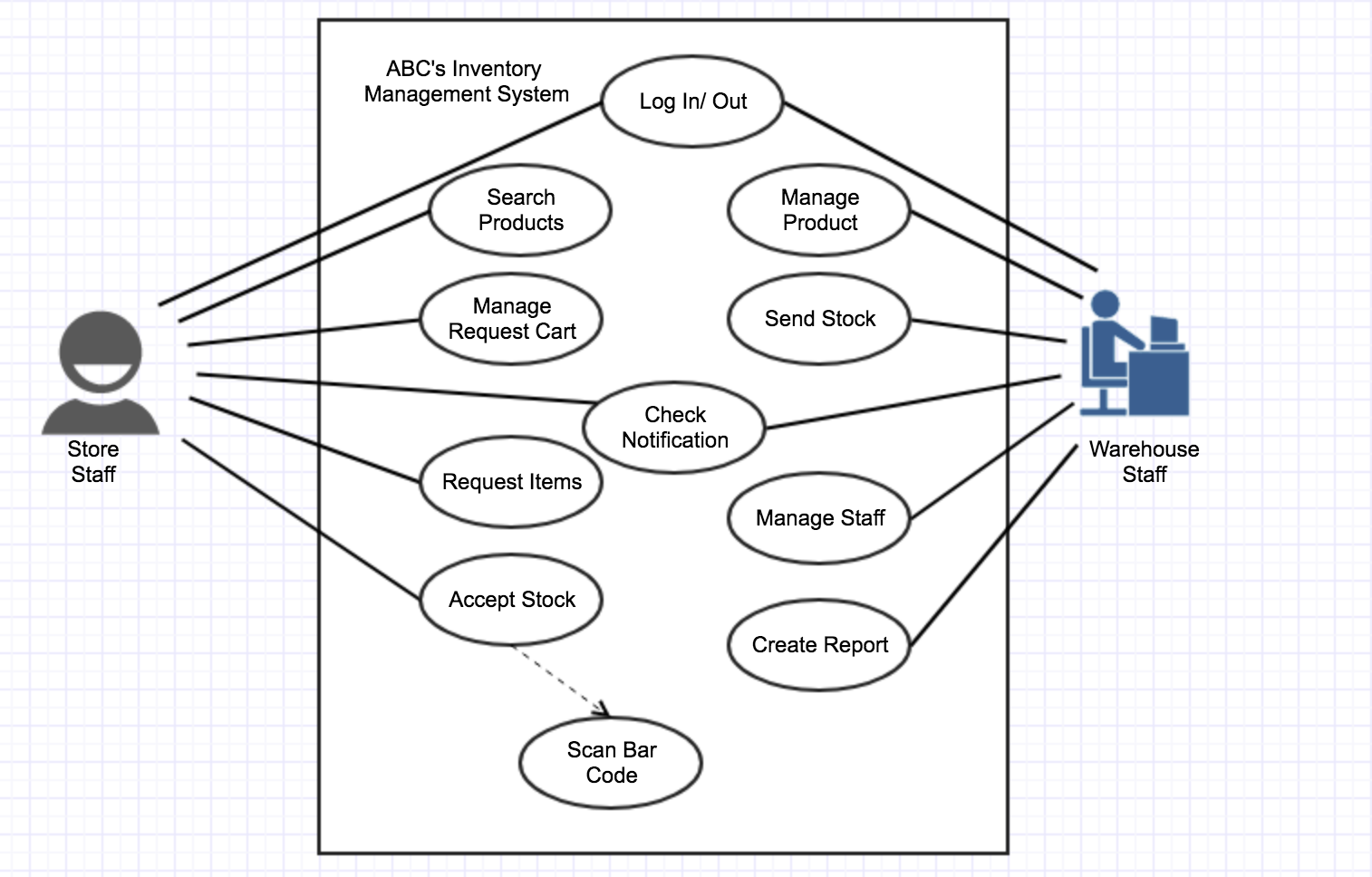
**Logical**

The following is the logical view that describes the software architecture:



**Use Case**

The following is the use case view of the software architecture:



**Physical**

The following is the physical view that describes the architecture:

