



Centers for Medicare & Medicaid Services
CMS Target Life Cycle (TLC)

BUDIGA: An Inventory Management and Point of Sales System for MSME

System Design Document

Version 1.0

5/24/2023

Table of Contents

1. Introduction	1
1.1 Purpose of the SDD	2
2. General Overview and Design Guidelines/Approach	3
2.1 General Overview	3
2.2 Assumptions/Constraints/Risks	3
2.2.1 Assumptions	3
2.2.2 Constraints	3
2.2.3 Risks	5
2.3 Alignment with Federal Enterprise Architecture	6
3. Design Considerations	8
3.1 Goals and Guidelines	8
3.2 Development Methods & Contingencies	8
3.3 Architectural Strategies	9
3.4 Performance Engineering	11
4. System Architecture and Architecture Design	13
4.1 Logical View	14
4.2 Hardware Architecture	14
4.2.1 Security Hardware Architecture	15
4.2.2 Performance Hardware Architecture	15
4.3 Software Architecture	15
4.3.1 Security Software Architecture	16
4.3.2 Performance Software Architecture	16
4.4 Information Architecture	16
4.4.1 Records Management	16
4.5 Internal Communications Architecture	19
4.6 Security Architecture	19
4.7 Performance	20
4.8 System Architecture Diagram	20
5. System Design	21
5.1 Business Requirements	21
5.2 Database Design	22
5.2.1 Data Objects and Resultant Data Structures	22
5.2.2 File and Database Structures	22
5.3 Data Conversion	23
5.4 User Machine-Readable Interface	23
5.4.1 Inputs	23

5.4.2 Outputs	24
5.5 User Interface Design	24
5.5.1 Section 508 Compliance	24
6. Operational Scenarios	25
7. Detailed Design	26
7.1 Hardware Detailed Design	26
7.2 Software Detailed Design	26
7.3 Security Detailed Design	27
7.4 Performance Detailed Design	28
7.5 Internal Communications Detailed Design	28
8. System Integrity Controls	29
9. External Interfaces	30
9.1 Interface Architecture	30
9.2 Interface Detailed Design	30

List of Figures

Figure 1 - High Level Logical View of a Web Application	14
Figure 2 - Presentation, Application, and Data Layers	15
Figure 3 - MVVM Design Pattern	16
Figure 4 - Budiga Application Security Design	21
Figure 5 - Budiga Performance Diagram	22
Figure 6 - System Architecture Diagram	23
Figure 7 - Budiga Entity Relationship Diagram	25
Figure 8 - Budiga Physical Data Model	28
Figure 9 - Admin Modules	30
Figure 10 - Employee Modules	31
Figure 11 - Add Invoice View	32
Figure 12 - Add Item View	33
Figure 13 - Receipt View	34
Figure 14 - Transaction History View	35
Figure 15 - Login Screen	37
Figure 16 - Inventory Screen	37
Figure 17 - Inventory History Screen	38
Figure 18 - Inventory Add and Edit Modals	38
Figure 19 - Invoice Screen	39
Figure 20 - Invoice Add Item Screen	39
Figure 21 - Filled Invoice Table	40
Figure 22 - Invoice Checkout Screen	40
Figure 23 - Sales Screen	41
Figure 24 - Employee Monitor Screen	41
Figure 25 - Employee Add and Edit Modals	42
Figure 26 - Budiga Hardware Detailed Design Diagram	43
Figure 27 - Budiga Security Detailed Design Diagram	66
Figure 28 - Budiga Performance Diagram	66

List of Tables

Table 1 - Hardware Specifications	5
Table 2 - Software Development Tools	17
Table 3 - Business Requirements	24
Table 4 - Data Objects	25
Table 5 - Sales Module	42
Table 6: Inventory Module	45
Table 7: Employee Module	50
Table 8: Invoice Module	56
Table 9: Scanner Module	62
Table 10: System Design Specifications	68
Table 11: Record of Changes	72
Table 12: Table of Acronyms	73
Table 13: Glossary	74
Table 14: Referenced Documents	75
Table 15: Approvals	76

1. Introduction

Throughout the decades, the Ministry of Micro, Small and Medium Enterprises (MSME) has been the dominant business trade in the Philippines. In a 2020 statistic by the Philippine Statistics Authority (PSA), a record of 952,969 (99.51%) of business establishments are MSMEs and 4,651 (0.49%) are large enterprises. In the same statistic, it is also stated that most of these MSMEs are under Wholesale and Retail Trade and Repair of Motor Vehicles and Motorcycles totalling to around 46.74% of the total.

For Wholesale and Retail Trade, there will come a time when managing the stocks and items manually will be very difficult. As the business grows in size, inventory management will be more prone to human error. Error in monitoring the inventory could lead to loss of profit. This also makes it harder to track security issues such as stocks being stolen by employees, especially if an employee is being hired to monitor the stocks.

Another important process in MSME business is the processing of transactions and sales. This is usually done through the use of pen, paper, and calculator. However, this is also vulnerable to human error. Additionally, receipts usually come in limited copies (two in most cases), which can easily be lost.

It is for these reasons that make it necessary to rely on digital softwares to computerize the process of managing inventory. Digitizing business data and inventory makes it easier to store receipts and other important papers. It also mitigates human-related errors since it minimizes the process needed for human interaction such as calculating for inventory and sales.

The proposed software that will mitigate these issues is an Inventory Management System. According to Nicole Pontius (2022), an inventory management system is the "combination of technology (hardware and software) and processes and procedures that oversee the monitoring and maintenance of stocked products, whether those products are company assets, raw materials and supplies, or finished products ready to be sent to vendors or end consumers". Additionally, the specific software that can aid in invoice processing is a Point of Sales (POS) software which can be defined as "the hardware and software that lets you checkout customers, take payment, and make sales." (Sebastian Rankin, 2021).

One MSME in specific that is currently facing issues and is in need of digitizing their inventory management is AJK3 Variety Store. They are located in Ponce Capitol, Cebu City. The client is the store owner Julio Melchor Rosales. A feature that the client mentions that wasn't mentioned before is to add a Barcode scanner for ease-of-use in terms of processing receipts. With that said, the proposed solution to meet their needs would be to create an Inventory Management and Point of Sales System software with an added Barcode Scanner API.

1.1 Purpose of the SDD

The SDD documents and tracks the necessary information required to effectively define architecture and system design in order to give the development team guidance on the architecture of the system to be developed. Design documents are incrementally and iteratively produced during the system development life cycle, based on the particular circumstances of the information technology (IT) project and the system development methodology used for developing the system. Its intended audience is the project manager, project team, and development team. Some portions of this document, such as the user interface (UI), may be shared with the client/user, and other stakeholders whose input/approval into the UI is needed.

With this, the purpose of this Software System Document is to outline the objectives, functionality, and scope of the Inventory Management System (IMS) designed specifically for Micro, Small, and Medium Enterprises (MSMEs). This document aims to provide a comprehensive understanding of the system, its features, and its significance in facilitating efficient inventory management processes through the integration of QR code scanning technology.

2. General Overview and Design Guidelines/Approach

This section describes the principles and strategies to be used as guidelines when designing and implementing the system.

2.1 General Overview

BUDIGA is an Inventory Management and Point of Sales System software with an added Barcode Scanner API that monitors and maintains the materials and products, allows for payment and creation of sales, and processing of receipts.

BUDIGA utilizes an open-source web server package, Apache's XAMPP project. The goal of XAMPP is to build an easy to install distribution for developers to make it convenient for developers. It is a platform that furnishes a suitable environment to test and verify the working of projects based on Apache, Perl, MySQL database, and PHP through the system of the host itself.

BUDIGA's development language is C#. It is a general-purpose, multi-paradigm programming language. It will be integrated with AForge.NET, a software library that extends C#'s capabilities. This framework is used for image processing.

BUDIGA utilizes MySQL Database. MySQL is an open-source relational database management system as a web database for applications. MySQL contains advanced features, management tools and technical support to achieve the highest levels of scalability, security, reliability, and uptime. Additional features on top of the web server functionality, like strong data protection from SSH and SSL, and comprehensive support for every application development need, have helped MySQL be the reliable package of the world's largest such as Facebook and Twitter. Currently MySQL is the most popular open-source web server on the Internet.

Barcode reading and decoding is done through ZXing ("zebra crossing"). ZXing is an open-source, multi-format 1D/2D barcode image processing library implemented in Java, with ports to other languages. This allows a device with imaging hardware to scan barcodes or 2D barcodes and retrieve the data encoded.

Github is used as the software development platform. Github provides version control and source code management. Github is the largest host of source code in the world.

In summary, the existing system design includes the following sub-systems:

- BUDIGA Application
- MySQL Database
- Apache Web Server
- ZXing Barcode Reader
- Github Version and Source Code Control

2.2 Assumptions/Constraints/Risks

2.2.1 Assumptions

This section describes the assumptions or dependencies regarding the system, software and its usage which will be expounded below.

Hardware and Software Dependencies

The system assumes the availability of compatible hardware components, such as barcode scanners, printers, and computers, capable of interfacing with the software. The system depends on the proper functioning and compatibility of these hardware components to ensure accurate scanning and data capture.

The system should run adequately on a 500GB storage drive (taking into consideration local data storage space) with a minimum of 4GB memory. A 720p camera for barcode scanning is enough and any screen resolution of 1280 x 720 and further is recommended.

Operating System Compatibility

The system assumes compatibility with specific operating systems, such as Windows. The system depends on the targeted operating systems to provide the necessary resources, APIs, and drivers for seamless integration and optimal performance.

Barcode Standards and Compatibility

The system assumes the use of widely accepted barcode standards, such as EAN (European Article Number), and their compatibility with the software. The system depends on barcode standards being correctly adhered to and scanned accurately by the barcode scanner hardware to ensure reliable data capture and inventory tracking.

End-User Proficiency

The system assumes that end-users have a basic understanding of barcode scanning technology and its proper usage. The system relies on end-users' ability to operate the barcode scanner correctly, scan barcodes accurately, and follow any necessary procedures for effective inventory management.

2.2.2 Constraints

Information found in this section pertains to the global limitations or constraints that have significant impacts on the design of the system's hardware and software which will be expounded into subsections below.

Hardware Environment

The system was developed on a laptop sporting the hardware specifications stated below:

Table 1: Hardware Specifications

Part	Model	Specification
Processor	Intel i5	9th Generation
Memory	N/A	16 Gigabyte 3200 Megahertz
Storage	Hard Drive Solid State	1 Terabyte 128 Gigabyte
Graphics	NVIDIA 1650super	4 Gigabyte
Camera	Built-in	720p

Certain hardware or software environments may impose limitations on system design. For example, if the target environment has limited processing power or memory capacity, the system design needs to be optimized to ensure efficient resource utilization and avoid performance bottlenecks.

Software Environment

The system was developed on Microsoft Windows. Since the system is built using the Windows environment, it refers to developing the software using Microsoft Windows as the operating system and leveraging the associated tools and frameworks provided by Microsoft. Here are some key points regarding the software environment when built using Windows which led to the development of the system itself:

1. Windows-Specific Development: Developing the system in a Windows environment means utilizing the Windows operating system as the primary platform for development, testing, and deployment. This environment provides a familiar and widely used ecosystem for software development.
2. Windows Development Tools: The Windows environment offers a range of development tools and integrated development environments (IDEs) provided by Microsoft. These tools, such as Visual Studio, offer a comprehensive set of features for coding, debugging, and building software applications for Windows.

3. Windows API/Frameworks: Windows provides its own set of application programming interfaces (APIs) and frameworks that developers can leverage when building software. These APIs enable developers to access various Windows-specific functionalities and services, such as user interface controls, file system operations, networking capabilities, and more. The system made use of Zxing, a .NET framework package, for barcode reading and support. The package is made compatible with .NET framework 4.0 which supports Windows 7 and later versions.
4. Compatibility with Windows-Based Systems: By building the system using the Windows environment, there is a high level of compatibility with Windows-based systems. The software can be optimized to run efficiently on Windows operating systems, ensuring smooth execution and seamless integration with other Windows applications.

Network Communications

The nature of network communications can impose constraints on system design. Bandwidth limitations, network latency, and intermittent connectivity may require the system to employ techniques such as data compression, caching, or offline data synchronization. The choice of communication protocols and data transfer mechanisms can significantly impact system performance and reliability.

A stable internet connection is necessary to use the application and its features such as accessing inventory and reading barcodes as it requires constant communication with the backend and database.

Verification & Validation Requirements

Verification and validation requirements impose constraints on system design to ensure its reliability, correctness, and adherence to specified requirements. The design must incorporate provisions for testing and quality assurance activities, including testability of software modules, test data generation, and compatibility with testing frameworks. During each sprint of the system development, the developers requested several meetings with the project adviser. As a result, the system's documentation has been rigorously analyzed in order to identify and correct any inaccuracies or oversights.

User Acceptance Testing is a kind of testing wherein the end user or customer assesses and approves the software system before deploying it in a production environment. This was done after each sprint or iteration of the product to determine its validity in the eyes of our clients. Every feature provided by the system was subjected to unit testing to ensure that the component's usability and functionality met the defined specifications.

2.2.3 Risks

This section describes the risks associated with the system design and proposed mitigation strategies that accompany it.

Data Accuracy

There is a risk of incorrect or inaccurate scanning and data capture from barcodes, leading to incorrect inventory records and potential operational disruptions. To address such, an implementation of barcode validation mechanisms will be done to ensure scanned data accuracy. Perform regular data reconciliation and verification processes. Provide user training on proper barcode scanning techniques to minimize errors.

System Integration

Possible challenges in integrating the Inventory Management System with existing software systems, such as accounting or POS systems, resulting in data inconsistencies and process inefficiencies. To tackle this, the team will conduct thorough compatibility and integration testing prior to deployment. Ensure the system's architecture supports seamless data exchange through standardized APIs or data formats. Collaborate closely with relevant vendors or the team to address integration challenges.

Security and Data Privacy

Possible unauthorized access to sensitive inventory data, barcode spoofing or tampering, or data breaches compromising confidential information. To address such an issue, the system will implement appropriate access controls, user authentication, and authorization mechanisms. Encrypt sensitive data during transmission and storage. Regularly update software components to address security vulnerabilities. Conduct security audits and penetration testing to identify and address potential vulnerabilities.

User Adoption and Training

End-users facing challenges in adopting the new system, lack of proper training, or resistance to change, resulting in low user acceptance and underutilization of barcode scanning functionality. One way to address such is to develop comprehensive user training materials and conduct training sessions to familiarize users with the system's features and benefits. Provide ongoing support and address user concerns promptly. Seek user feedback and incorporate usability improvements based on user experiences.

2.3 Alignment with Federal Enterprise Architecture

The system design document aligns with the Federal Enterprise Architecture framework by addressing the following aspects:

Business Architecture Alignment:

The document recognizes the Ministry of Micro, Small and Medium Enterprises (MSME) as a dominant business trade in the Philippines, acknowledging its significance and impact.

It highlights the need for efficient inventory management and point of sales processes, which aligns with the business goals and objectives of MSMEs in the Wholesale and Retail Trade sector.

Technology Architecture Alignment:

The document emphasizes the utilization of digital software to automate and computerized inventory management and sales processes, reducing human errors and streamlining operations.

It proposes the development of an Inventory Management and Point of Sales System software, which aligns with the adoption of technology solutions to enhance business operations.

Security Architecture Alignment:

The document touches upon security concerns, such as employee theft and the need to track and secure stocks.

The proposed software solution can incorporate security measures to mitigate risks, aligning with the security architecture principles of the Federal Enterprise Architecture.

Performance Architecture Alignment:

The document acknowledges the potential for human-related errors in manual inventory management and sales processes, which can impact performance and profitability.

By proposing a digital software solution, the document aligns with the aim to improve performance, accuracy, and efficiency in managing inventory and processing sales transactions.

3. Design Considerations

This section describes the issues which need to be addressed or resolved before attempting to devise a complete design solution.

3.1 Goals and Guidelines

The design of the Inventory Management and Point of Sales System for MSME is driven by a set of goals, guidelines, principles, and priorities that aim to create an efficient, user-friendly, and seamlessly integrated solution.

Prioritizing user-friendliness across the system's design is one overarching objective. The focus is on developing a user-friendly interface that is intuitive and requires little training for MSME employees and owners. The system intends to increase overall efficiency in managing inventory and carrying out sales transactions by putting a strong emphasis on user-friendliness.

Speed and efficiency must be taken into account during the design process. The system is built to work as efficiently as possible, with a focus on reducing response times and processing delays. In order to increase productivity and customer happiness, this goal makes sure that inventory management and sales operations can be carried out easily and promptly.

Integration capability is another significant priority. The system is built to smoothly interact with additional useful business tools, such as e-commerce platforms, customer relationship management (CRM) systems, or accounting software. The effective flow of data is made possible by this integration capabilities, which also reduces the need for manual data entry and offers a comprehensive picture of corporate processes.

To guarantee consistency and maintainability of the product, coding rules and guidelines are adhered to. These rules cover topics including naming conventions, code organization, required levels of documentation, and adherence to object-oriented and modular design concepts. These recommendations make the system's code more legible, understandable, and error-free, allowing future maintenance and improvements.

While the design goals and principles concentrate on the overall system behavior and user experience, specific design strategies and tactics have an impact on the specifics of the system's implementation and interface. For instance, the implementation of certain capabilities or the design of the user interface may be influenced by the selection of particular goods or technologies. These choices are based on elements like compatibility, dependability, and scalability, ensuring that the system satisfies the unique requirements of MSMEs while taking cost-effectiveness and future expansion into account.

3.2 Development Methods & Contingencies

The object-oriented design method was chosen due to its ability to represent real-world entities and their interactions effectively. It allows for modular, reusable components, encapsulation, inheritance, and polymorphism. The design leverages the principles of object-oriented programming (OOP) to create a flexible and maintainable system.

The design process incorporates common practices and principles of OOP. These include designing classes and objects to represent entities such as products, orders, customers, and transactions. Encapsulation ensures data privacy and exposes only necessary interfaces, while inheritance promotes code reuse and hierarchy of objects.

The Unified Modeling Language (UML) can be used to create diagrams representing the system's structure, behavior, and relationships between components. UML class diagrams, sequence diagrams, and state diagrams can help visualize the design and aid communication between stakeholders and developers.

Other design methods or approaches, such as structured design or prototyping, may have been considered but not adopted. Structured design, while effective in modularization and structured analysis, may lack the flexibility and reusability offered by an object-oriented approach. Prototyping, although useful for quick validation and feedback, might not have been chosen due to time or budget constraints.

Contingencies that may arise during the design process include:

1. Lack of Interface Agreements

If the system needs to interface with external agencies or systems, but there are no established interface agreements or specifications, it can introduce uncertainties. In such cases, a flexible design approach can be adopted, allowing for future integration based on eventual interface agreements. Workarounds may involve designing loosely coupled components and using integration patterns like adapters or facades to encapsulate external system interactions.

2. Unstable Architectures

If there are architectural instabilities or technology uncertainties, it is important to identify them early. Contingency plans can be developed, such as identifying alternative technologies or architectural patterns that can be substituted if the initial choices prove unstable or unsuitable. Conducting architectural analysis and prototyping can help validate the chosen technologies and address any architectural concerns.

3.3 Architectural Strategies

Layered Architecture

The system employs a layered architecture, dividing the functionality into distinct layers such as presentation, business logic, and data access. This decision promotes separation of concerns, modularization, and maintainability. The presentation layer handles the user interface, the business logic layer contains the application's logic and rules, and the data access layer manages interactions with the database. This layered approach ensures flexibility, ease of testing, and future enhancements.

Model-View-Controller (MVC) Pattern

The system incorporates the MVC pattern to separate the concerns of data representation (model), user interface (view), and user interactions (controller). This design decision allows for independent development, testing, and modification of each component. The model represents the data and business rules, the view handles the user interface rendering, and the controller manages user input and orchestrates interactions between the model and view. By employing the MVC pattern, the system achieves modularity, extensibility, and code reusability.

Database Design

The system employs a relational database management system (RDBMS) for data storage. The design decision involves creating appropriate tables, relationships, and indexes to efficiently store and retrieve data. The goal is to ensure data integrity, optimize query performance, and provide scalability. Normalization techniques are applied to minimize redundancy and maintain data consistency. The design prioritizes a well-structured database schema to support the system's data management requirements.

Scalability and Performance

To address scalability and performance requirements, the system utilizes caching mechanisms, indexing strategies, and optimization techniques. Caching frequently accessed data reduces the load on the database and improves response times. Indexing is employed to speed up data retrieval operations. Additionally, the system employs optimization techniques such as query optimization and database partitioning to enhance performance and accommodate growing data volumes. The trade-off here is the added complexity and maintenance overhead of implementing these strategies, but the benefits in terms of system performance outweigh the drawbacks.

Reuse of Existing Software Components

Instead of building all parts/features of the system from scratch, the decision was made to leverage existing software components or libraries that are widely used and have proven reliability. While CMS may have specific guidelines for component reuse, the chosen components were thoroughly evaluated for compliance with security and interoperability requirements. This approach significantly reduced development time and effort while ensuring the system benefits from mature and well-tested solutions.

Future Extensibility

The system's architecture was designed with future extensibility in mind. While CMS may have recommended a specific architectural style or pattern, the chosen approach prioritizes modularity and loose coupling to facilitate future enhancements. The decision to adopt a modular and layered architecture, such as Model-View-Controller (MVC), allows for independent development and maintenance of system components, enabling seamless integration of new features and technologies as MSME needs evolve.

Error Detection and Recovery

The design decision for error detection and recovery mechanisms may deviate from CMS guidelines based on specific project requirements or technical considerations. For example, instead of relying solely on CMS-prescribed error reporting mechanisms, the decision was made to implement custom logging and exception handling strategies tailored to the system's unique needs. This approach allows for more granular error monitoring, diagnostic capabilities, and enhanced troubleshooting.

Concurrency and Synchronization

The design decision for concurrency and synchronization mechanisms may deviate from CMS guidelines based on the specific needs of the system. CMS recommendations may include specific locking mechanisms or transaction isolation levels. However, if alternative concurrency control techniques, such as optimistic locking or multi-version concurrency control, provide better performance or scalability in the given context, they may be chosen instead.

Communication Mechanisms

Deviations from CMS guidelines regarding communication mechanisms may occur if specific project requirements or technical constraints warrant it. While CMS may prescribe standard protocols, message formats, or security measures, the chosen communication mechanisms may differ based on the integration needs of the system. The decision to deviate would be justified by factors such as compatibility with existing infrastructure, support for real-time data exchange, or specific security requirements.

3.4 Performance Engineering

The system was designed with a focus on minimizing response time to provide a seamless user experience. This involved various strategies and optimizations. Firstly, database queries were optimized by ensuring proper indexing, query tuning, and efficient data retrieval techniques. This reduced the time taken to fetch and manipulate data from the database. Additionally, the system employed efficient algorithms and data structures to perform operations quickly and effectively. Complex tasks were broken down into smaller, optimized steps, minimizing processing time. Caching mechanisms were also implemented to store frequently accessed data, reducing the need for repeated queries and improving response time. By incorporating these optimizations, the system achieved low response times, allowing users to interact with the system swiftly and perform tasks without delays.

The system's design prioritized scalability to accommodate the growth of micro, small, and medium enterprises (MSMEs) and handle increasing data volumes. The chosen architecture allowed for horizontal scaling, where additional servers or resources could be added to distribute the workload and manage the increasing user base. The system utilized distributed systems or cloud-based infrastructure to facilitate this scalability, ensuring that the system could handle a larger number of concurrent users and higher transaction volumes without sacrificing performance. By designing the system with scalability in mind, it can effectively meet the evolving needs of MSMEs and accommodate their future growth.

High throughput was a significant consideration in the system's design. To achieve this, the system employed techniques such as asynchronous processing, parallelization, and load balancing. Asynchronous processing allowed tasks to be executed concurrently, ensuring efficient utilization of system resources and maximizing throughput. Parallelization techniques were used to break down tasks into smaller units that could be processed simultaneously, further enhancing throughput. Load balancing mechanisms distribute the workload evenly across multiple resources, preventing bottlenecks and enabling efficient processing of a large number of transactions. By optimizing throughput, the system could handle a high volume of concurrent requests, providing efficient and responsive performance.

Efficient resource utilization was a key aspect of the system's design. The system optimized CPU, memory, and network bandwidth usage to ensure optimal performance. This involved adopting coding practices that minimize computational overhead, optimizing data structures to reduce memory usage, and minimizing unnecessary data transfers over the

network. By efficiently utilizing system resources, the system could deliver high performance while effectively managing infrastructure costs.

The design of the system prioritized availability and reliability to ensure uninterrupted operation. Redundancy and failover mechanisms were implemented to mitigate the impact of hardware or software failures. This involved replicating critical components and establishing backup systems to ensure continuous service availability. The system also incorporated robust error handling and recovery strategies to handle exceptional situations and maintain data integrity. By prioritizing availability and reliability, the system aimed to minimize downtime and provide a reliable platform for businesses to manage their inventory and sales operations.

The system's design included provisions for testing and quality assurance activities, aligning with the Verification and Validation Requirements. The design aimed to ensure the testability of software modules by employing modular and loosely coupled architectures. This allowed for easier isolation and testing of individual components, enabling efficient and comprehensive testing processes. The system was designed to be compatible with testing frameworks and allowed for the generation of test data to validate its functionality and performance against the defined requirements. By incorporating testability into the design, the system aimed to deliver a high-quality and well-tested software solution.

The system's design considered network efficiency to optimize data transfers and reduce network latency. Data compression techniques were employed to minimize the size of data transmitted over the network, reducing bandwidth requirements and improving overall network performance. Efficient data serialization formats were chosen to optimize data exchange between system components. The system also aimed to minimize unnecessary network round-trips by utilizing techniques such as batch processing or caching. By focusing on network efficiency, the system aimed to deliver a faster and more responsive user experience, particularly in network-dependent operations.

Throughout the design process, trade-offs and optimizations were made to balance the Performance Requirements with other design considerations, such as maintainability, scalability, and security. Different design alternatives were evaluated, and the chosen approaches were based on their ability to meet the Performance Requirements while maintaining the overall system integrity and usability.

4. System Architecture and Architecture Design

This section outlines the system and hardware architecture design of the system.

4.1 Logical View

The Logical view offers a higher-level representation of the system, focusing on the problem domain. In the realm of web applications, the architecture can be divided into three tiers: Presentation Logic, Business Logic, and Database tier. The Presentation Logic tier encompasses the components responsible for generating the User Interface. These components interact exclusively with the Business Logic tier, which holds the necessary knowledge to manipulate the data components residing in the Database tier. The Database tier comprises the components responsible for persistently storing application data, such as customer addresses or product inventories. This three-tiered architecture effectively separates concerns and is particularly beneficial for large-scale web applications.

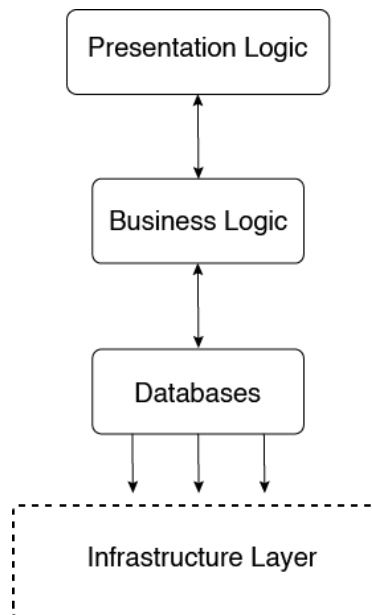


Figure 1: High Level Logical View of a Web Application

4.2 Hardware Architecture

A distributed processing system is chosen for the system's development due to its advantages in fault tolerance, data accessibility, flexibility, and energy efficiency. It enables organizations to create robust and scalable systems capable of handling large workloads, processing data quickly, adapting to changing demands, and delivering reliable and accessible services to users. In contrast, a centralized processing system presents a single point of failure that can impact the entire system. It also has limitations in scalability and higher network dependency.

The figure illustrates the hardware components found in the presentation, application, and data server layers. The process begins with end users connecting to the web server, provided they have access through the firewall and proxy wall. This connection is associated with the presentation layer. The API layer and the application itself reside in the business layer, handling the system's logic. The database service serves as the data layer, which interacts with the business logic.

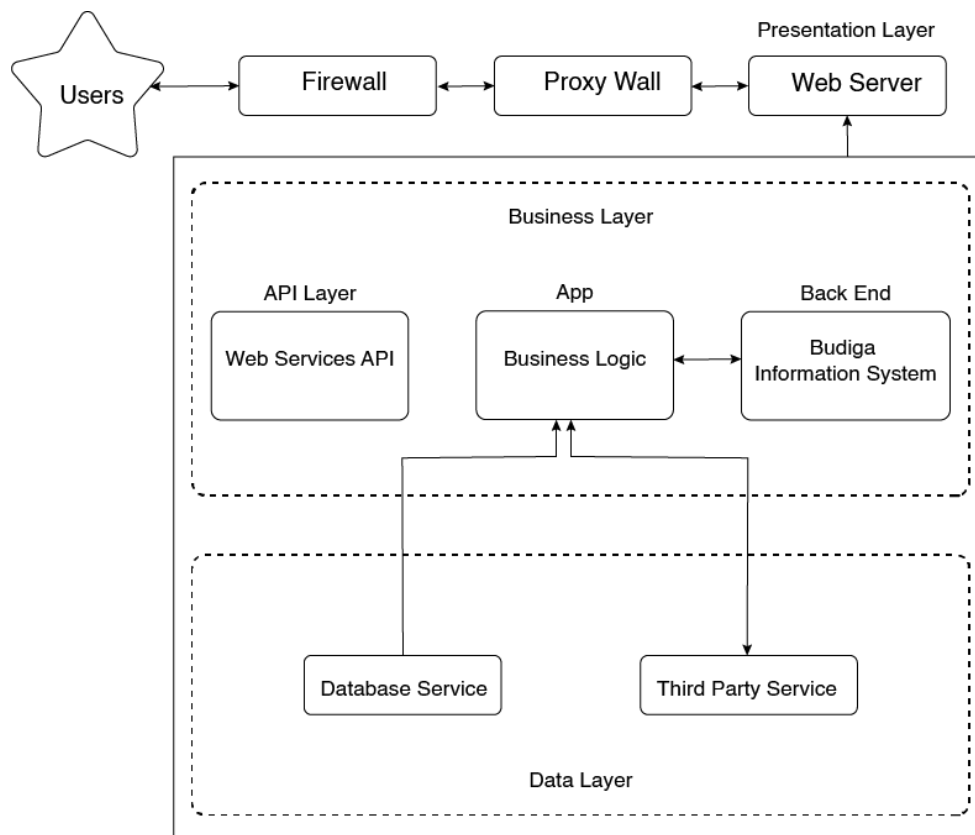


Figure 2: Presentation, Application, and Data Layers

4.2.1 Security Hardware Architecture

Not Applicable

4.2.2 Performance Hardware Architecture

Not Applicable

4.3 Software Architecture

The system adopts the MVVM (Model-View-ViewModel) design pattern which is a widely used architectural pattern for building software applications. It is a software architectural design pattern that separates the user interface (View) from the underlying data (Model) by introducing a mediator called ViewModel. The Model represents the application's data and business logic. It encapsulates the data structures, database interactions, and validation rules. The Model is responsible for managing data persistence and providing an interface for accessing and manipulating the data. Another component is the View which represents the presentation layer of the application, responsible for displaying the user interface and visual elements. It renders the data from the Model and presents it to the users. It may generate HTML, XML, JSON, or other output formats for rendering the user interface. The ViewModel which acts as an intermediary between the View and the Model. It provides data and behavior that the View needs to display and interact with the data. The ViewModel exposes properties and commands that the View can bind to, allowing it to retrieve and update data from the Model. It also handles user interactions and communicates with the Model to perform data operations.

This pattern offers a structured and organized approach to software development, providing benefits such as separation of concerns, code reusability, testability, scalability, parallel development, enhanced user experience, and adaptability.

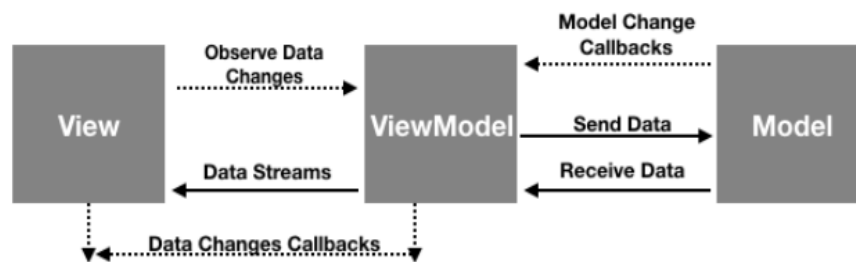


Figure 3: MVVM Design Pattern

In the MVVM architecture, applications can interact with other applications through the use of Application Programming Interface (APIs). It allows other applications to access their functionality or retrieve data. The RESTful API (Representational State Transfer) is an architectural style for designing networked applications that communicate over the web. It is an approach for building web services that adhere to specific principles and constraints. RESTful APIs offer benefits such as simplicity, scalability, performance, platform independence, wide adoption, and integration capabilities. The table below displays the imperative tools that will be utilized for software development.

Table 2: Software Development Tools

Name	Version	Developer	Purpose
Visual Studio Community 2022	17.1.2	Microsoft	Integrated development environment (IDE) to develop windows applications.
C#	7.3	Microsoft	General-purpose, multi-paradigm programming language to create the application.
.NET Framework	4.8	Microsoft	Proprietary software framework to build and run windows application.
ZXing	3.4.1	srowen	Open-source barcode reading and decoding library.
AForge.NET	2.2.5	Andrew Kirillov	C# framework for image processing.
MySQL	8.0	Oracle Corporation	Open-source relational database management system as a web database for the application
XAMPP	7.4.28	Apache	Open-source cross-platform to serve as a web server solution.

4.3.1 Security Software Architecture

User Authentication. Users provide their credentials, typically a username and password, to the system to initiate the authentication process. These credentials are unique to each user and serve as a means to verify their identity.

Password Verification. When a user submits their credentials, the system retrieves the corresponding user information from the database based on the provided username. The stored password is then compared with the submitted password. To validate the password, use a strong hashing algorithm, such as bcrypt which is a popular adaptive hashing algorithm designed for password hashing. It incorporates a work factor that can be increased over time to make it computationally expensive for attackers to crack passwords through brute-force or dictionary attacks., to compare the stored hash with the hash of the submitted password. If the hashes match, the password is considered valid.

Encryption Protocol. Ensure that the user credentials are transmitted securely over the network by using encryption protocols such as HTTPS or SSL/TLS. This prevents unauthorized parties from intercepting and obtaining sensitive user information and securing transactions.

4.3.2 Performance Software Architecture

In a distributed system, where data is spread across multiple nodes or servers, the database can become a single point of failure if it is not properly designed or managed. To mitigate the risks associated with a single point of failure in the database, distributed systems employ strategies such as data replication, distributed databases, sharding, and fault-tolerant database architectures. These approaches distribute the data and processing across multiple nodes, ensuring redundancy, improved performance, and fault tolerance. By avoiding a centralized database architecture and implementing robust data management strategies, the single point of failure concern can be mitigated in a distributed system.

4.4 Information Architecture

Product Information:

Description: Details about the products available in the inventory, including product names, descriptions, SKU numbers, selling prices, and quantities.

PII/IIF/PHI: Product information typically does not include PII, IIF, or PHI as it primarily pertains to the products themselves rather than personal or sensitive data.

Employee Information:

Description: Details about employees registered in the system, including names, contact information, employee IDs, and attendance history.

PII/IIF/PHI: Employee information can potentially include PII or IIF since it relates to identifiable individuals. However, it does not usually involve PHI unless the system is specifically designed for managing health-related employee information.

Sales Transaction Data:

Description: Information about individual sales transactions, including product details, quantities sold, and total transaction amounts.

PII/IIF/PHI: Sales transaction data may not typically include PII, IIF, or PHI if customer information, payment methods, and discounts are not captured as part of the sales process.

4.4.1 Records Management

Federal regulations issued by the National Archives and Records Administration (NARA) are outlined in 36 Code of Federal Regulations (CFR) - Subchapter B - Records Management. Business owners must contact the Office of Strategic Operations and Regulatory Affairs (OSORA) to initiate the record management process.

4.4.1.1 Data

Sales Module:

Data: Sales transaction data

Format: Electronic data (generated through the Point of Sale system)

Supplier: Customers or sales associates who interact with the system to initiate sales transactions. They provide data such as product names, quantities, prices, customer information, payment methods, and any applied discounts or taxes.

Inventory Module:

Data: Product information in the inventory

Format: Electronic data (stored in a database or spreadsheet)

Supplier: Admin or authorized personnel who input product details into the system. They supply information such as product names, descriptions, cost prices, selling prices, and quantities on hand.

Employee Monitor Module:

Data: Employee registration and attendance history

Format: Electronic data (stored in a database or spreadsheet)

Supplier: Admin or authorized personnel responsible for managing employee records. They supply data related to employee registration, including employee names, contact details, and any other relevant information. Additionally, attendance data can be manually entered by employees or recorded through an integrated attendance system.

Invoice Module:

Data: Invoice details and past transactions

Format: Electronic data (generated and stored within the system)

Supplier: Admin or authorized personnel who create invoices and add relevant information such as customer details, product names, quantities, prices, discounts, and payment details. Employees with appropriate permissions may also contribute to creating invoices.

Scanner Module:

Data: Scanned and decoded product information from barcodes

Format: Electronic data (captured by the scanner module and integrated into the system)

Supplier: Employees who use the scanner (camera) module to scan product barcodes. The scanned data, such as product identifiers or SKU codes, is automatically supplied to the system for processing.

4.4.1.2 Manual/Electronic Inputs

Once manual or electronic inputs are entered into the master file/database and verified in the Inventory Management System with a barcode scanner, a seamless and efficient process ensures the accurate and reliable management of your inventory. The steps are:

1. Manual/Electronic Input

Whether you manually enter information into the system or use the barcode scanner to capture data, rest assured that the system is designed to handle both methods effortlessly. You have the flexibility to choose the most convenient approach based on your preference and requirements.

2. Validation and Verification

Once the inputs are entered, the system springs into action, promptly validating and verifying the data. Through adequate validation, the system ensures the accuracy and integrity

of the information. This crucial step eliminates any potential errors or discrepancies, guaranteeing the reliability of the inventory records.

3. Database Integration

The verified inputs seamlessly integrate into the master file or database, the beating heart of your Inventory Management System. With each entry, the system meticulously organizes and categorizes the data, creating a comprehensive repository of the inventory information. This centralized database becomes the single source of truth for the inventory records.

4. Real-time Updates

The power of the system lies in its ability to provide real-time updates. As soon as the inputs are successfully integrated into the database, the system instantaneously reflects the changes across all relevant modules and screens. This dynamic synchronization ensures that end-users always have up-to-date and accurate information at your fingertips.

5. Streamlined Operations

The integration of manual/electronic inputs into the master file/database optimizes the inventory management operations. The system's sophisticated algorithms and efficient data structures enable swift and precise search capabilities, ensuring speedy retrieval of inventory information. This streamlines various tasks, such as inventory replenishment, order processing, and stock audits, saving valuable time and resources.

6. Enhanced Accuracy and Efficiency

By leveraging the power of the Inventory Management System, the chances of errors and inaccuracies in your inventory records are minimized. The reliance on manual data entry is greatly reduced, reducing the risk of human errors and associated costs. The barcode scanner, with its swift and accurate scanning capabilities, revolutionizes your inventory management process, enhancing efficiency and productivity.

4.4.1.3 Master Files

In an Inventory Management System with a barcode scanner, the system/database maintains a rich and comprehensive set of data to facilitate effective inventory control and management. Here is a detailed description of the data typically maintained in such a system/database when stored locally:

1. Product Information

The database stores essential details about each product in your inventory, including the product name, description, SKU (Stock Keeping Unit), barcode information, unit of measurement, and any other pertinent product identifiers. This information enables accurate identification and tracking of products throughout the inventory management process.

2. Stock Levels

The system/database keeps track of the quantity of each product available in stock. It records the current stock levels, including the number of units or items on hand and available for sale. This data helps you monitor inventory levels, prevent stockouts, and plan for restocking or replenishment.

3. Purchase Orders and Invoices

The system/database maintains records of purchase orders and invoices associated with the procurement of inventory items. This includes details such as purchase order numbers, dates, quantities, costs, payment terms, and relevant vendor/supplier information. Storing this data allows for accurate tracking of procurement activities, invoice reconciliation, and financial reporting.

4. Sales Data/Transaction History

The system/database stores data related to sales transactions. This includes details of each sale, such as sales order numbers, dates, quantities sold, and prices. Tracking sales data enables accurate revenue reporting, customer relationship management, and sales analysis.

5. System Logs and Audit Trails

To ensure data integrity and security, the system/database may maintain system logs and audit trails. These logs capture information about user activities, such as login/logout timestamps, database modifications, and system events. Monitoring and reviewing these logs assist in troubleshooting, detecting anomalies, and maintaining compliance with data security and audit requirements.

4.5 Internal Communications Architecture

The proposed solution utilizes existing communication protocols and methods. There are no additional components, servers, or applications to communicate with. The proposed solution will simply extend the current capabilities.

4.6 Security Architecture

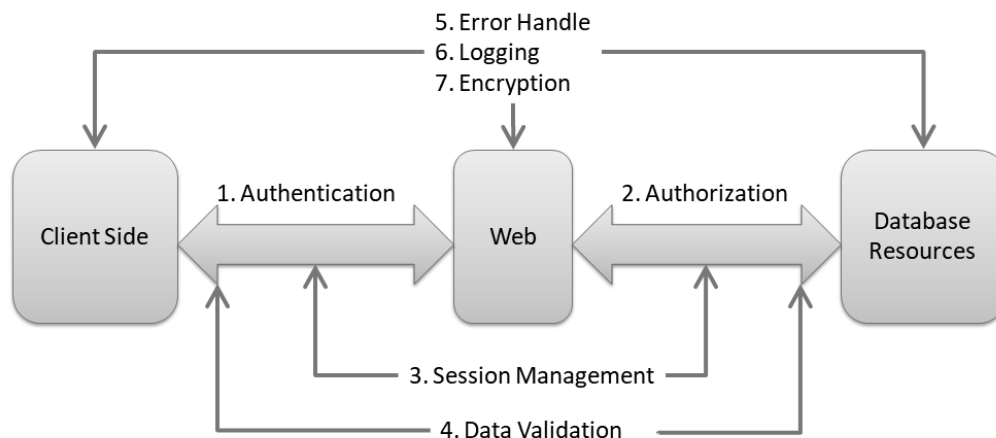


Figure 4. Budiga Application Security Design

4.7 Performance

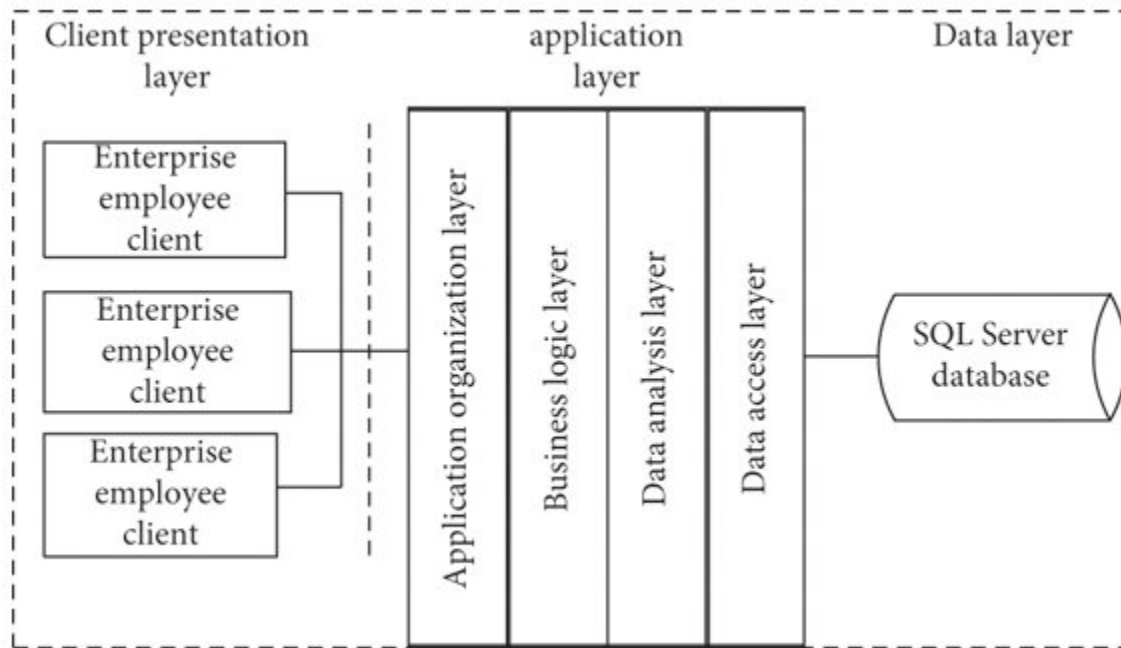


Figure 5. Budiga Performance Diagram

4.8 System Architecture Diagram

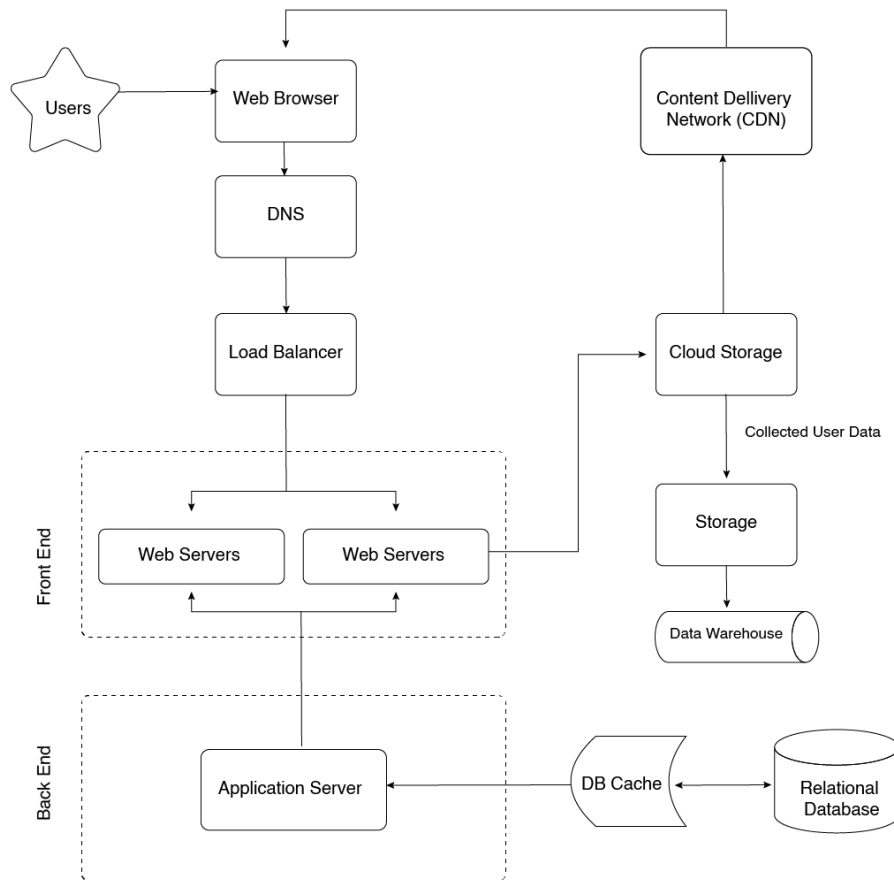


Figure 6: System Architecture Diagram

The interactions between these components involve the user accessing the web application through the web browser. The DNS resolves the domain name to the system's IP address, which is then directed to the load balancer. The load balancer distributes the requests to multiple web servers. The web servers process the requests and may interact with the application server, database cache, relational database, and CDN to retrieve or update data as required. Additionally, the application server may communicate with the database cache and relational database for data processing. The CDN serves static content, and the cloud storage in data warehouses stores and provides access to large volumes of data for analysis and reporting.

5. System Design

5.1 Business Requirements

Table 3: Business Requirements

Inventory Tracking	The system should provide accurate and real-time tracking of inventory levels, enabling businesses to efficiently manage stock and avoid stock outs or overstocking.
Sales Management	The system should support sales-related processes, such as order management, invoicing, and payment processing, to streamline the sales cycle and improve efficiency.
Reporting and Analytics	The system should generate reports and provide analytics capabilities to help businesses gain insights into sales trends, inventory performance, and overall business performance.
Integration with Payment Gateways	The system should integrate with popular payment gateways to enable seamless and secure online transactions
Scalability	The system should be designed to handle increasing data volumes and accommodate the growth of MSMEs without compromising performance or stability.
Security	The system should incorporate robust security measures to protect sensitive customer information, prevent unauthorized access, and comply with relevant data protection regulations.

5.2 Database Design

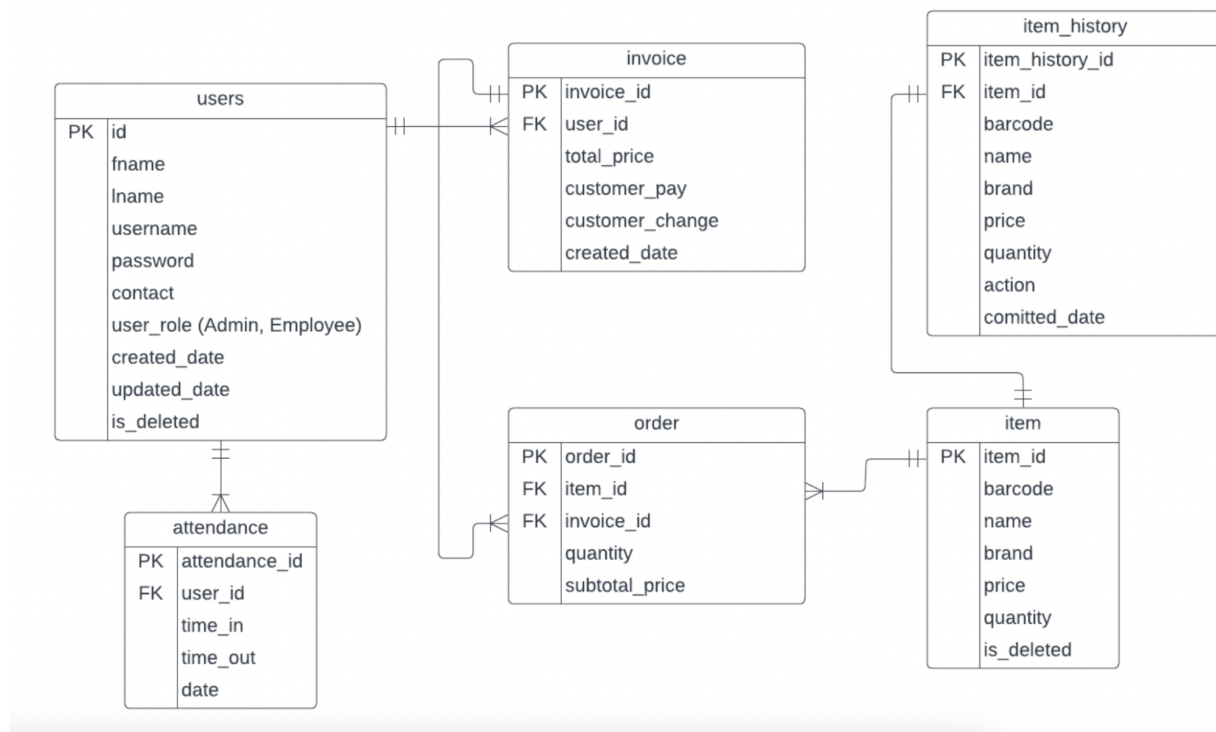


Figure 7. Budiga Entity Relationship Diagram

This is the ERD of Budiga which represents the database structure of the application. The diagram contains 6 entities namely: users, attendance, item, order, invoice, and item_history and shows a graphical representation that depicts relationships between entities.

5.2.1 Data Objects and Resultant Data Structures

These are the following data objects used in the system:

Table 4: Data Objects

Data Objects	Data Structure	Functions
Users	<ul style="list-style-type: none"> id fname lname username password contact user_role created_date updated_date 	<ul style="list-style-type: none"> GetAllEmployee() <ul style="list-style-type: none"> Input: None Output: Employee list AddEmployee() <ul style="list-style-type: none"> Input: None Output: Create new user with employee role EditEmployee() <ul style="list-style-type: none"> Input: user_id Output: Modified user

	<ul style="list-style-type: none"> • is_deleted 	<ul style="list-style-type: none"> • DeleteEmployee() <ul style="list-style-type: none"> ◦ Input: user_id ◦ Output: Delete employee from database • Login() <ul style="list-style-type: none"> ◦ Input: username, password ◦ Output: Create session • Logout() <ul style="list-style-type: none"> ◦ Input: user_id ◦ Output: Destroy session
Attendance	<ul style="list-style-type: none"> • attendance_id • user_id • time_in • time_out • date 	<ul style="list-style-type: none"> • CheckIn() <ul style="list-style-type: none"> ◦ Input: user_id ◦ Output: Create new attendance with empty timeout slot • CheckOut() <ul style="list-style-type: none"> ◦ Input: attendance_id ◦ Output: Updates attendance timeout slot to current time
Item	<ul style="list-style-type: none"> • item_id • barcode • name • brand • quantity • is_deleted 	<ul style="list-style-type: none"> • GetAllItem() <ul style="list-style-type: none"> ◦ Input: None ◦ Output: Item list • AddItem() <ul style="list-style-type: none"> ◦ Input: None ◦ Output: Create new item • EditItem() <ul style="list-style-type: none"> ◦ Input: item_id ◦ Output: Edited item • DeleteItem() <ul style="list-style-type: none"> ◦ Input: item_id ◦ Output Delete item from database • SearchItem() <ul style="list-style-type: none"> ◦ Input: any string ◦ Output: Item list related to the inputted string
Order	<ul style="list-style-type: none"> • order_id • item_id • invoice_id • quantity • subtotal_price 	<ul style="list-style-type: none"> • AddOrder() <ul style="list-style-type: none"> ◦ Input: item_id ◦ Output: Create new order • RemoveOrder() <ul style="list-style-type: none"> ◦ Input: order_id ◦ Output: Remove order from list

		<ul style="list-style-type: none"> • IncreaseQuantity() <ul style="list-style-type: none"> ◦ Input: order_id ◦ Output: Increment quantity value • DecreaseQuantity() <ul style="list-style-type: none"> ◦ Input: order_id ◦ Output: Decrement quantity value • GetOrderFromInvoice() <ul style="list-style-type: none"> ◦ Input: invoice_id ◦ Output: order[]
Invoice	<ul style="list-style-type: none"> • invoice_id • user_id • total_price • customer_pay • customer_change • created_date 	<ul style="list-style-type: none"> • Checkout() <ul style="list-style-type: none"> ◦ Input: order_id[] ◦ Output: Create new invoice • GetReceipt() <ul style="list-style-type: none"> ◦ Input: invoice_id ◦ Output: receipt display, order[] • CalculatePayment() <ul style="list-style-type: none"> ◦ Input: order[] -> subtotal_price ◦ Output: total_payment
Item History	<ul style="list-style-type: none"> • item_history_id • item_id • barcode • name • brand • price • quantity • action • comitted_date 	<ul style="list-style-type: none"> • AddItemHistory() <ul style="list-style-type: none"> ◦ Input: item_id, action (add, delete, edit) ◦ Output: Create a new item history • UndoItemChanges() <ul style="list-style-type: none"> ◦ Input: item_history_id ◦ Output: Undo the changes made from the item and revert back to its previous change

5.2.2 File and Database Structures

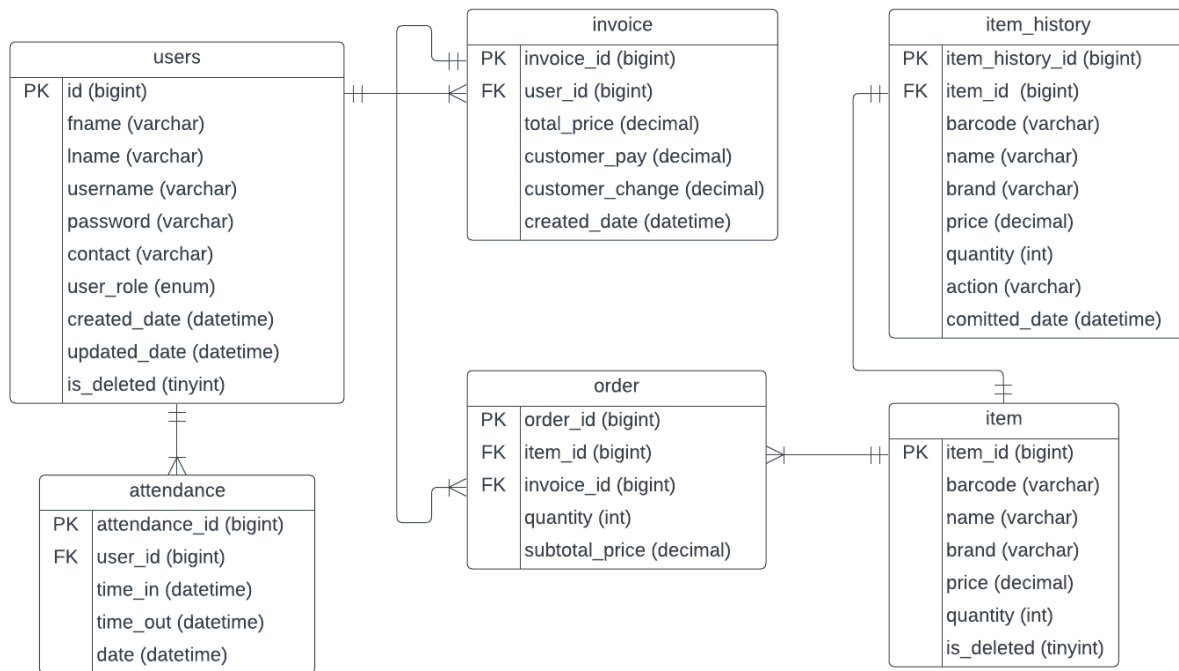


Figure 8. Budiga Physical Data Model

5.2.2.1 Database Management System Files

Not Applicable

5.3 Data Conversion

Not Applicable

5.4 User Machine-Readable Interface

The system consists of two distinct user classes: employee and admin. These user classes have different roles and levels of access within the system. The employee user class is responsible for carrying out tasks related to the Invoice and Inventory modules. They are authorized to create, update, and delete invoices for customer orders, as well as manage item stocks in the inventory. However, employees do not have the ability to register themselves in the application. Instead, the admin user class, which possesses exclusive access to the Employee List module, is responsible for registering and removing employee accounts from the database. This ensures that employee accounts are created and managed by authorized personnel.

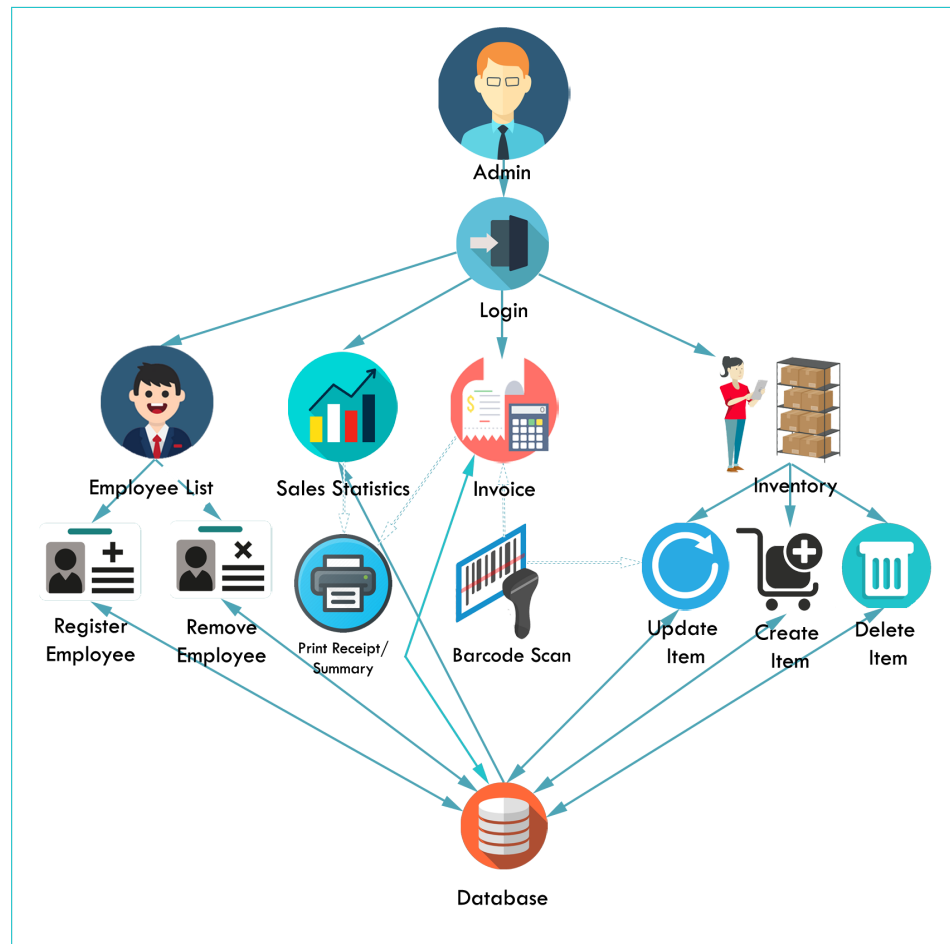


Figure 9. Admin Modules

The admin user class has access to all four modules in the system: Employee List, Sales Statistics, Invoice, and Inventory. Employee List allows the admin to register and remove an employee user from the database. Sales Statistics module allows the admin to view the statistics of the sales based on various factors such as time, date, item, etc. He/she can also choose to print the summary statistic if needed. Invoice module simply allows the admin to make an invoice/receipt for a customer's order. Doing this action will affect the inventory database. The print functionality is also available for this module. Another function that can be used in this module is the Barcode Scan function. The user will be able to select the item to be added in the invoice using a Barcode scanner via the device camera. Lastly, we have the Inventory module. This is where the admin can update, create, delete and monitor item stocks in the inventory.

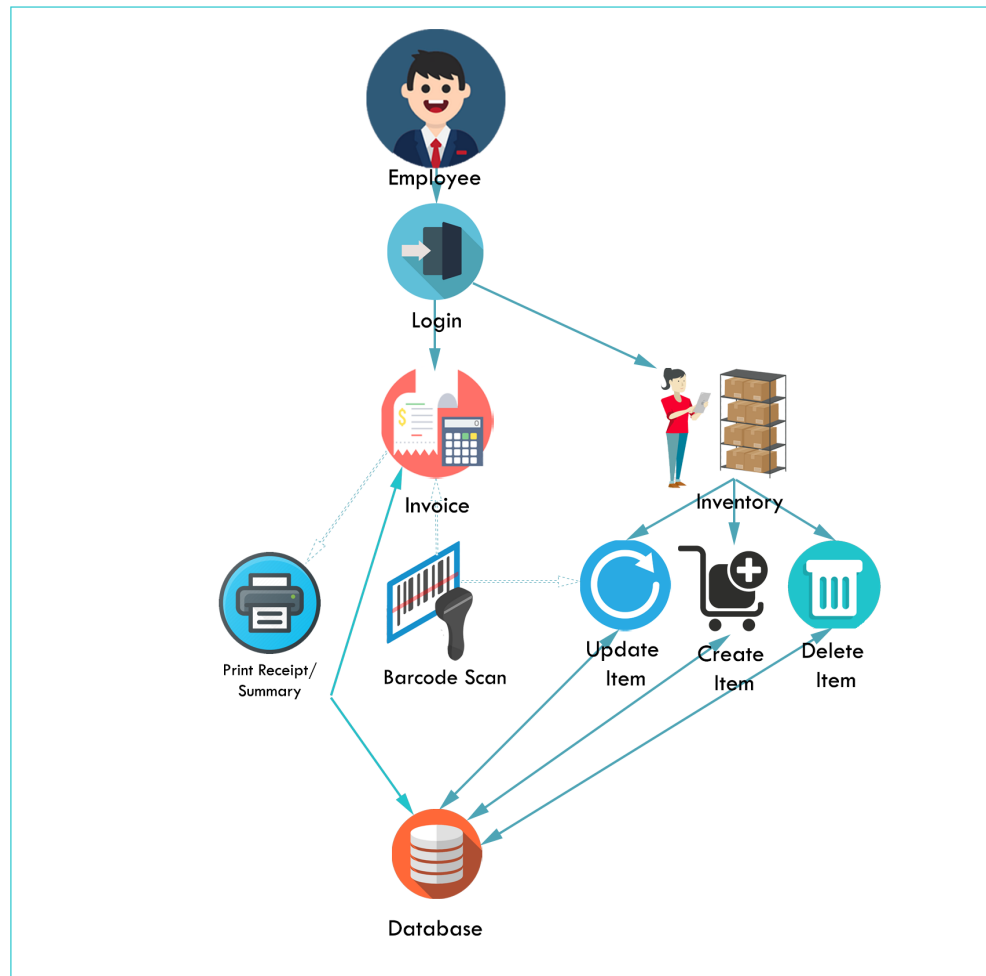


Figure 10. Employee Modules

The Employee only has access to the Invoice and Inventory Module. The same functionalities are present for both modules. What sets the admin and employees apart is that the employee cannot make changes to the employee database and cannot view the sales statistics.

As for the estimated number of users, only one admin user and any number of employees are allowed for each branch or store. This distribution ensures that there is a designated admin responsible for the overall management and administration of the system, while multiple employees carry out day-to-day operational tasks.

5.4.1 Inputs

The screenshot displays the 'Add Invoice View' in the Budiga system. The interface includes a sidebar with 'Inventory' and 'Invoice' tabs, and a top bar with the 'budiga' logo and 'Employee 1' profile. The main area features a table with the following data:

#	Product Information	Quantity	Price	Subtotal
1	Coca Cola 1.5 Liters	1	PHP 69.50	PHP 69.50
2	Colgate Charcoal Toothpaste	2	PHP 100.00	PHP 200.00
3	Durex Ultra Thin Condom	1	PHP 115.00	PHP 115.00

Below the table is an 'Add Items' button with a plus icon. At the bottom, a summary bar shows 'Total Quantity : 3' and 'Total 304.50'. Navigation buttons include 'Completed Invoices', 'Cancel Order', and 'Checkout'. A footer note states '© 2022, Made with ❤ by 3JC Development Team'.

Figure 11. Add Invoice View

In the Add Invoice view, the employee interacts with the system to finalize customer transactions, manage the list of purchased products, cancel transactions if needed, and utilize the barcode scanning feature.

The GUI consist of the following components:

1. **Transaction List:** This section displays the list of products that have been added to the current transaction. It provides a visual representation of the products, including details such as product name, quantity, price, and total amount.
2. **Checkout Button:** This button allows the users to signal the completion of the transaction. Upon clicking this button, the system will process the transaction and generate an invoice or receipt for the customer.
3. **Cancel Order Button:** This button enables the users to cancel the current transaction if the customer decides not to proceed with the purchase or any other unforeseen circumstances arise. Clicking this button will clear the transaction list and reset the view.
4. **Add Item/s Button (+):** This button changes the view of software to the Add Item View. This allows users to add new items to the transaction list.
5. **Remove Item Button (x):** Each item in the transaction list may have a corresponding remove (x) button. This allows the employee to selectively remove specific items from the current transaction, in case of errors or changes in customer preferences.
6. **Transaction History Button:** This button changes the view of the software to the Transaction History View. This section allows the user to view all transactions made using the system.

The data elements associated with the GUI component includes:

- Invoice Data Table: total price
- Order Data Table: subtotal price, quantity
- Item Data Table: name, brand, price

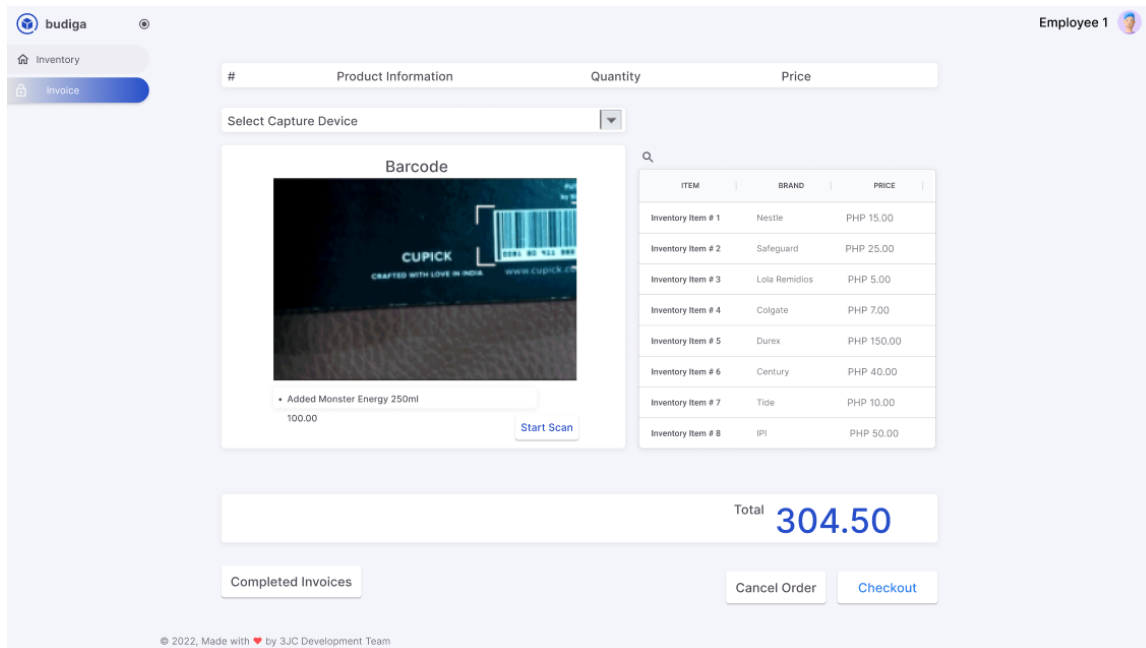


Figure 12. Add Item View

In the Add Item view, employees utilize various input media to add items to the current transaction, including barcode scanning for product identification and a search field to filter and select specific items. These input media enable employees to efficiently manage and update the transaction list.

The GUI consist of the following components:

1. **Start Scan Button:** This button allows the user to utilize the barcode scanner implemented in the system. The barcode scanner serves as the primary input medium for product identification. Employees can use a barcode scanner integrated with the device's camera to scan the barcode on a product. The scanned barcode information is then processed by the system, which retrieves the necessary details about the product, such as name, price, and other relevant information. This information is added to the current transaction list.
2. **Item List:** This section shows the list of available items or products. Each item entry includes details such as the product name, price, and any other relevant information.
3. **Search Field:** The search field within the Add Item view allows employees to filter the list of available items or products. By entering specific keywords or product names into the search field, employees can quickly locate and select the desired item to add to the transaction. This functionality enhances the usability and efficiency of the system, especially when dealing with a large inventory of items.
4. **Checkout Button:** This button allows the users to signal the completion of the transaction. Upon clicking this button, the system will process the transaction and generate an invoice or receipt for the customer.
5. **Cancel Order Button:** This button enables the users to cancel the current transaction if the customer decides not to proceed with the purchase or any other unforeseen circumstances arise. Clicking this button will clear the transaction list and reset the view.

6. Transaction History Button: This button changes the view of the software to the Transaction History View. This section allows the user to view all transactions made using the system.

The data elements associated with the GUI component includes:

- Invoice Data Table: total price
- Item Data Table: name, brand, price, barcode

5.4.2 Outputs

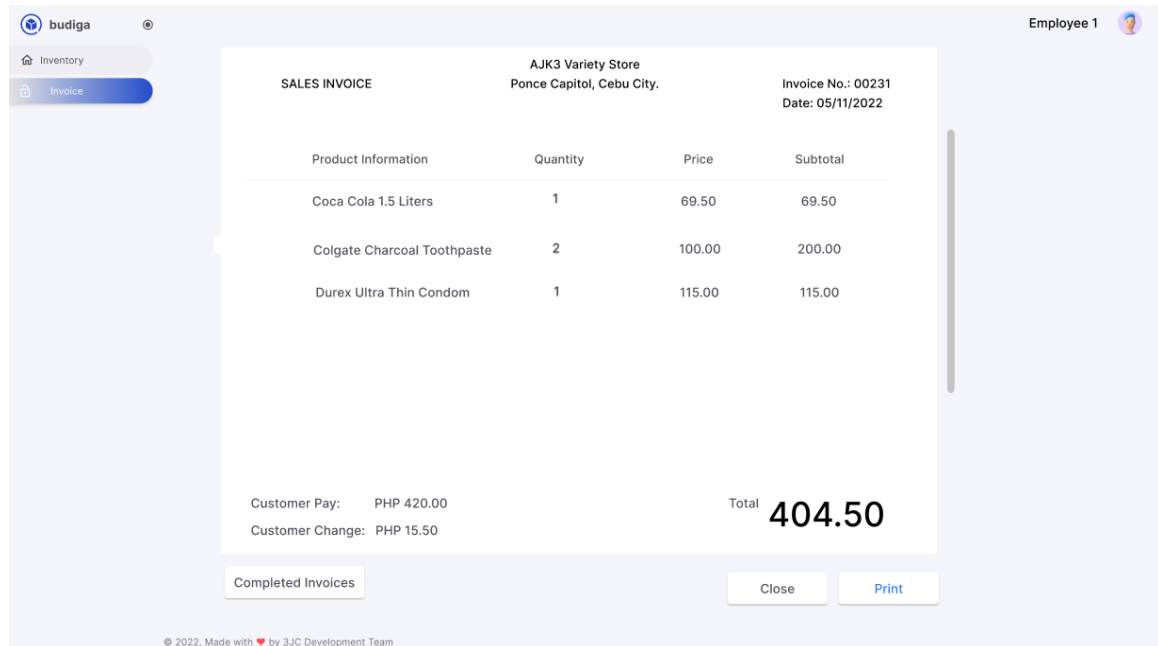


Figure 13. Receipt View

In the Receipt View, users have the capability to view and print completed invoices. This view provides a convenient way to access and generate physical copies of invoices for record-keeping or customer interactions.

The GUI consist of the following components:

1. Invoice Display: This section displays the completed invoices, allowing users to view the details of each invoice. This includes information such as customer name, transaction date, items purchased, quantities, prices, and the total amount. The invoice display provides a clear and organized representation of the completed transactions.
2. Print Button: This button enables users to generate a physical copy of the invoices. By selecting the print option, the system generates a printable version of the invoice, which can be sent to a printer connected to the device or saved as a digital file for later use. This functionality facilitates the creation of tangible receipts that can be provided to customers or maintained for administrative purposes.
3. Close Button: This button changes the view of the software back to the Add Invoice View.
4. Transaction History Button: This button changes the view of the software to the Transaction History View. This section allows the user to view all transactions made using the system.

The data elements associated with the GUI component includes:

- Invoice Data Table: total price, customer pay, customer change, created date, invoice id
- Order Data Table: subtotal price, quantity
- Item Data Table: name, brand, price

INVOICE NO.	DATE	ITEMS	BRAND	PRICE	QUANTITY	SUBTOTAL	TOTAL
#4910	25/04/2022	Coca-Cola 1.5 Liters 3	Coca-cola	PHP 30.00	6	PHP 180.00	PHP 1554.00
		Colgate Charcoal Toothpaste 2	Colgate	PHP 224.00	6	PHP 1344.00	
		Mentos Roll Candy 29g 5	Mentos	PHP 5.00	6	PHP 30.00	
#4911	25/04/2022	Colgate Charcoal Toothpaste 2	Colgate	PHP 248.00	6	PHP 1494.00	PHP 1524.00
		Mentos Roll Candy 29g 5	Mentos	PHP 5.00	6	PHP 30.00	
#4912	25/04/2022	Colgate Charcoal Toothpaste 6	Colgate	PHP 224.00	6	PHP 1344.00	PHP 1524.00
		Coca-Cola 1.5 Liters 2	Coca-Cola	PHP 30.00	6	PHP 180.00	

Figure 14. Transaction History View

In the Transaction History View, users have the ability to access and review a list of previously completed invoices. This view provides a comprehensive overview of past transactions and allows users to view individual invoices as they were originally printed. Additionally, users can choose to reprint specific invoices by selecting the corresponding rows in the list.

The GUI consist of the following components:

1. Invoice List: This section presents a list of previously completed invoices. This list contains relevant information about each invoice, such as customer name, transaction date, invoice number, and total amount. The list provides an organized view of past transactions, allowing users to quickly locate specific invoices based on their attributes.
2. Reprint Option: Users have the option to reprint invoices directly from the Transaction History View. By clicking on the respective rows that contain the desired invoice, users can initiate the reprinting process. This allows users to generate a fresh copy of the invoice, which can be printed or saved digitally as needed.
3. Checkout Button: This button allows the users to signal the completion of the transaction. Upon clicking this button, the system will process the transaction and generate an invoice or receipt for the customer.
4. Cancel Order Button: This button enables the users to cancel the current transaction if the customer decides not to proceed with the purchase or any other unforeseen circumstances arise. Clicking this button will clear the transaction list and reset the view.
5. Transaction History Button: This button changes the view of the software back to the Add Invoice View.

The data elements associated with the GUI component includes:

- Invoice Data Table: total price, created date, invoice id
- Order Data Table: subtotal price, quantity
- Item Data Table: name, brand, price, barcode

5.5 User Interface Design

The user interface design was created using Figma, an online collaboration platform for interface design. The design can be accessed through this link: <https://bit.ly/FigmaBudiga>

5.5.1 Section 508 Compliance

Budiga does not include any accessibility considerations as this system was developed with a well-established understanding of the client and the users. The users do not have disabilities that require special features in the system. However, the user interface was created with a color palette with contrasting colors so that all elements are completely visible. Furthermore, the interface has an intuitive design so that users have a seamless experience in using Budiga.

6. Operational Scenarios



Figure 15. Login Screen

Upon opening the software, the user is welcomed with the login screen as shown in Figure 14. Here, the user is required to input their respective usernames and passwords. If the account credentials refer to an admin account, they will be redirected to the inventory of the admin. Otherwise, they will be referred to the inventory screen for employees.

BARCODE	PRODUCT	BRAND	PRICE ↑	QUANTITY
59421908025	Colgate Fresh 150mL	Colgate	150.00	5
59423908095	Sprite Mismo	Sprite	12.00	10
594213208085	Coca-Cola 1.5 Liters	Coca Cola	50.00	5
543419238055	Mentos Roll 29g	Mentos	99.00	3
59411228625	Milo 500g	Nestle	150.00	5
59421908015	Century Tuna 50 g	Century	40.00	20
59423908045	555 Sardines Can 20 g	555	30.00	35
52325935635	Delimondo Corned Beef	Delimondo	150.00	5

© 2022, Made with ❤️ by 3JC Development Team

Figure 16. Inventory Screen

After the log in screen is the inventory interface for both admins and employees. The page includes a table of all the available items sold in AJK3 Variety Store. The table includes the item barcode, product name, brand, price, and stocked quantity.

BARCODE	PRODUCT	BRAND	PRICE	QUANTITY	ACTIONS	TIMESTAMP
59421908025	Royal Mismo	Royal	12.00	20	Updated	4/23/2022 16:34:23
59454608025	Sprite 1.5 Liter	Sprite	50.00	1	Deleted	4/20/2022 4:20:23
59542249080	Royal Mismo	Royal	12.00	20	Deleted	4/1/2022 8:34:23
5294219085	EQ Diapers	EQ	250.00	1	Created	5/0/2022 6:34:23
5242408025	Century Tuna Can	Century	40.00	2	Created	5/10/2022 8:34:23
55423201025	Delimondo Corned Beef	Delimondo	150.00	3	Updated	5/11/2022 16:4:23
5123908525	C2 Green Tea	C2	50.00	2	Created	5/2/2022 8:34:23
59122925220	Mentos Roll 29g	Mentos	99.00	1	Created	5/23/2022 8:34:23

© 2022, Made with ❤️ by 3JC Development Team

Figure 17. Inventory History Screen

There is also a toggle button to switch between the inventory table and the history. Once the user toggles for the history page, it shows a table that provides an overview of all the changes made within the inventory. The table has columns entitled “Actions” and “Timestamp” to identify the activity enacted and the exact time the effect took place on a particular item.

There are key differences in the inventory according to the user type:

- Admin - The page also includes a search bar to quickly sift through the items and an add button so that the admin may provide additional items into the inventory.
- Employee - There is an absence of the add button and the employee is unable to commit any changes to the available items in the inventory.

INVENTORY ADD

Barcode:

Product:

Brand:

Price:

Qty:

INVENTORY EDIT

Barcode:

Product:

Brand:

Price:

Qty:

Figure 18. Inventory Add and Edit Modals

The admin user type is able to add and edit items in the inventory. When the user clicks the “Add” button at the top of the inventory interface, they are greeted with the “Inventory Add” modal which requires the user multiple inputs about the product. The user will be asked for the following inputs: barcode number, product, brand, price, and quantity. When they are done filling up the form, they can either add the product to the inventory list or cancel the provided information and close the modal.

The admin is also given the authority to update information about the products in the inventory. The user simply has to click the row of the product and a modal will open with inputs filled with information about the respective item.

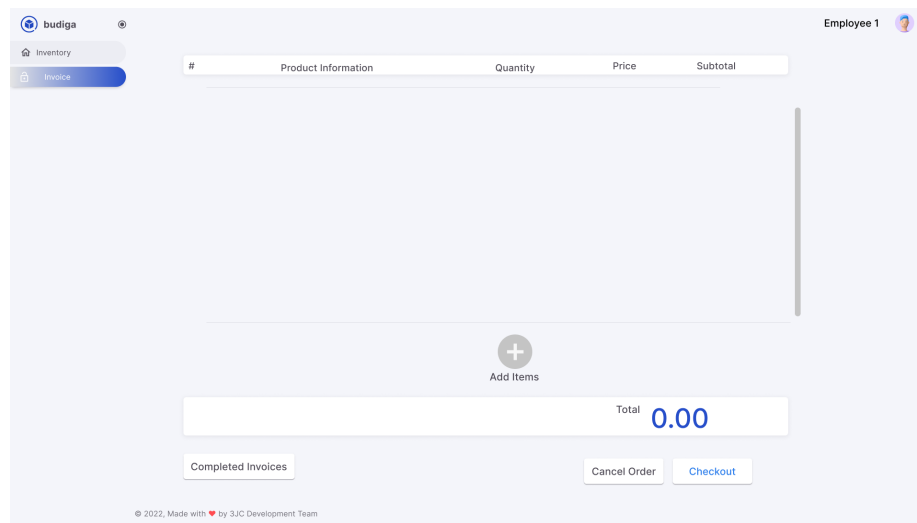


Figure 19. Invoice Screen

The invoice page encapsulates Budiga's point-of-sale portion where transactions are processed. Both employees and admins can access this page and there are no differences for either user type. When there is a transaction, it is placed in a table that includes the item number, product information, quantity, price, and subtotal. Below the table is an "Add Items" button that will lead the user to the Add Item page for items to be added to the transaction. This is followed by the total portion that sums up the amount to be paid by the customer. There are also three buttons below this and they are the following: the "Completed Invoices" button to redirect the user to the history of transactions, the "Cancel Order" button to erase the current data in the transaction, and the "Checkout" button to finalize the transaction.

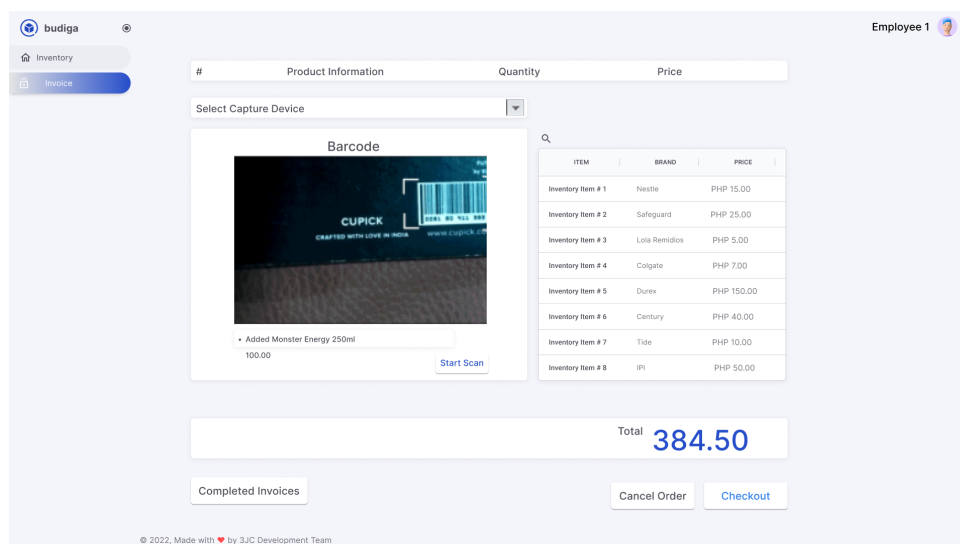


Figure 20. Invoice Add Item Screen

This interface allows the admin or employee to add an item to the transaction. They are given two choices to include an item. First, the user may scan the item's barcode. With the barcode,

the system is able to automatically identify the item name and its price. The user is also given a dropdown input of all the connected capture devices to provide the user with multiple camera options. An alternative way to add an item is by manually choosing the item among the list of available products in the AJK3 Variety Store. This is a table that includes columns of the item name, brand, and price. It also includes a search bar for a quicker process of finding the item.

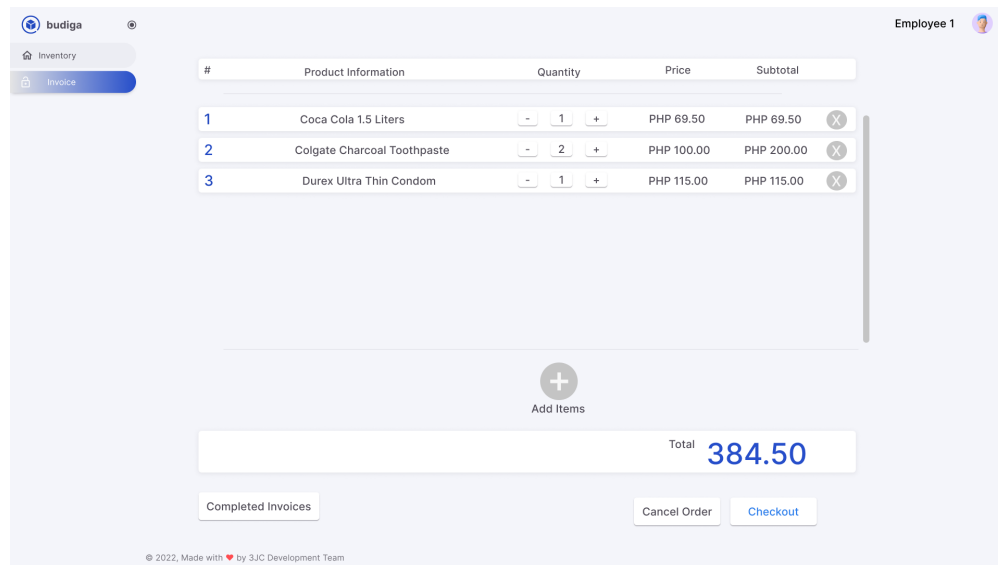


Figure 21. Filled Invoice Table

When the user adds an item, they are brought back to the main page of the invoice page. Here they are listed with all the current items in the transaction. In the respective row of the item, the user is given the ability to decrease or increase the quantity of an item by clicking the minus or plus sign, respectively. Furthermore, they also have the option to completely delete the item from the transaction with the “X” button at the rightmost portion of the item row.

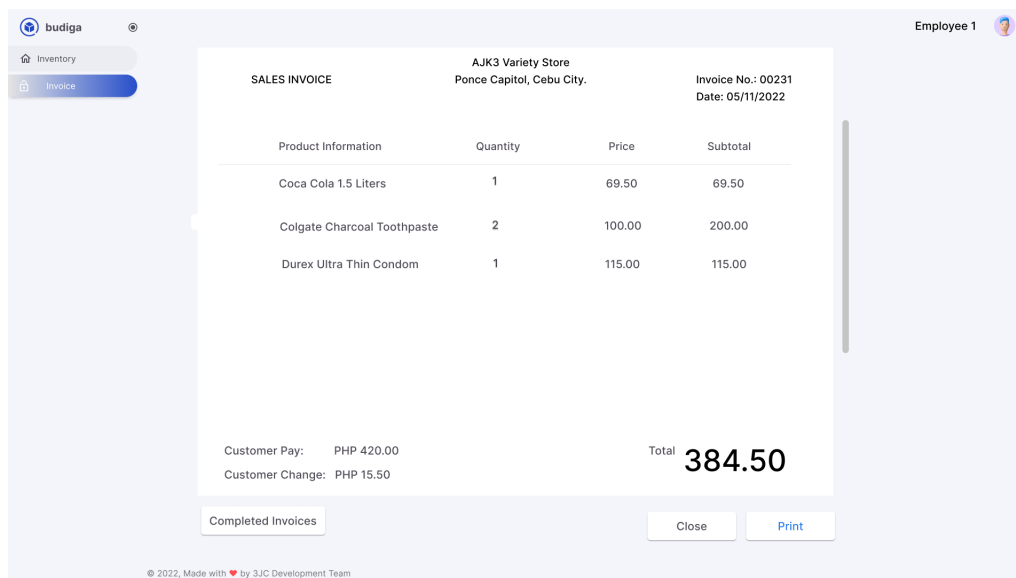


Figure 22. Invoice Checkout Screen

When the user is done inputting all the items in the transaction, they are brought to an interface that summarizes the transaction. This acts as a receipt, so this includes an invoice number,

invoice date, and the option to print the information with the “Print” button. Unlike the main table of the invoice page, this also includes the amount given by the customer page and the appropriate amount to be given back as change.

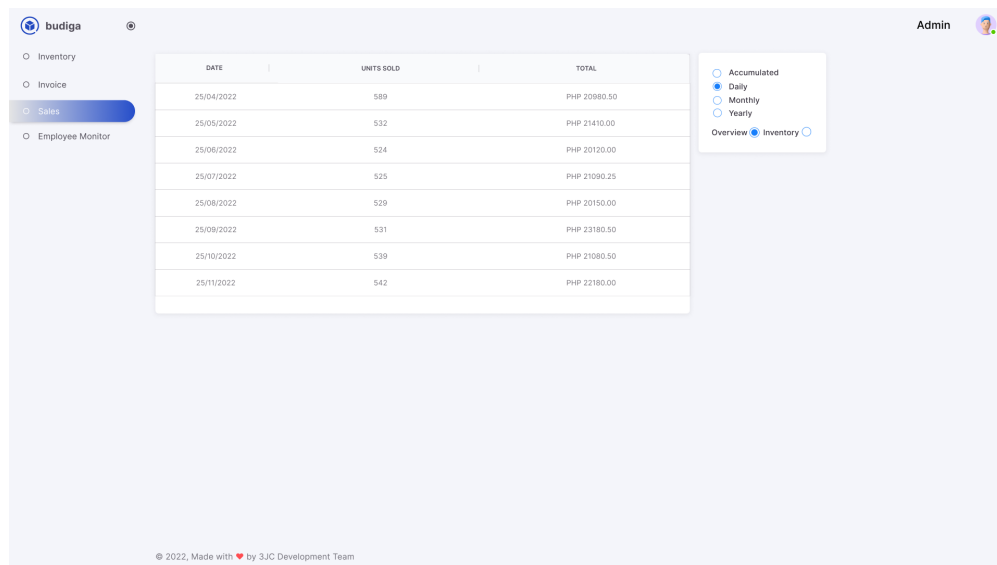


Figure 23. Sales Screen

The sales page is only accessible by the admin user type as this includes an overview of AJK3 Variety Store’s transactions. This includes a table that displays the date, the number of units or items sold, and the total amount of profit for that specific day. The table displays the eight most recent dates when Budiga was used. At the right side of the page, the user is given the option to change the view of the summary. The default is the daily display, but they can also choose the accumulated (or the total of everything), monthly, or yearly.

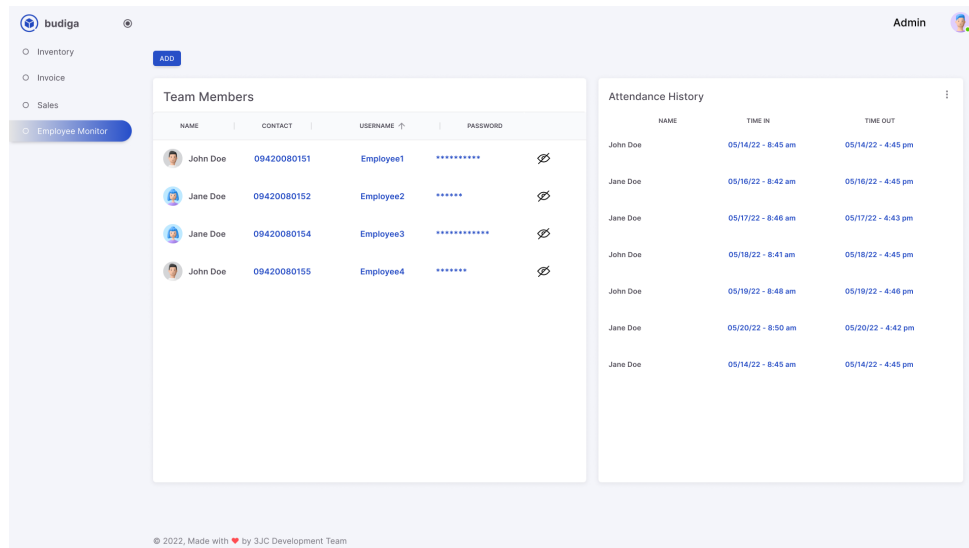


Figure 24. Employee Monitor Screen

The employee monitor interface allows the administrator to have an overview of all the accounts of the people currently employed in the business. The first noticeable feature is the “Add” button at the top. This allows the admin to create accounts for any new, incoming employees. This is

followed by two tables. The first one is the team members. This is a list of all employees and their information such as their names, contact numbers, usernames, and passwords. The right one is an attendance history table where it tracks the working hours of the employees. It includes columns such as the employee's name, the time when they start working, and the time when they get off work.

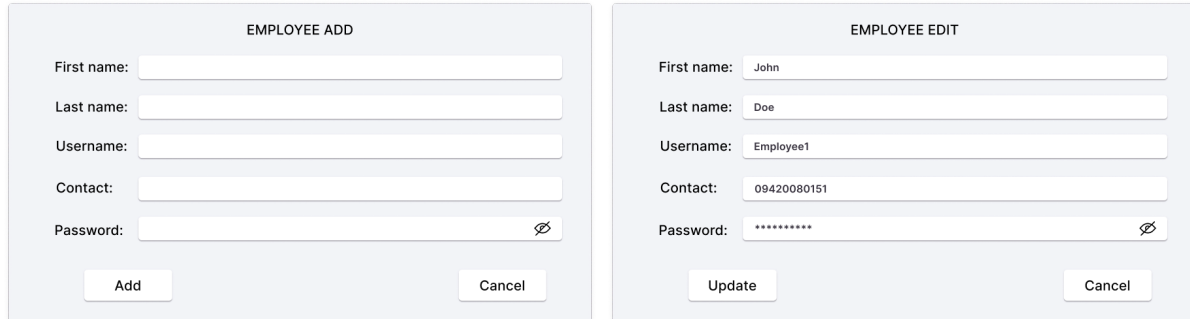


Figure 25. Employee Add and Edit Modals

When the admin clicks the “Add” button from the Employee Monitor page, the “Employee Add” modal will appear and ask for the following information about the employee: their first name, last name, user name, contact number, and password. Once the form is complete, the admin may click the “Add” button so that the new employee may be listed under the team members in the Employee Monitor interface.

To update any information about an employee, the admin only needs to click the row of the appropriate employee under the Team Member page. Once clicked, the input fields from the “Employee Add” modal are automatically filled out with the information of the respective user. Clicking the “Update” button will save and allow the changes to be reflected in the database.

7. Detailed Design

7.1 Hardware Detailed Design

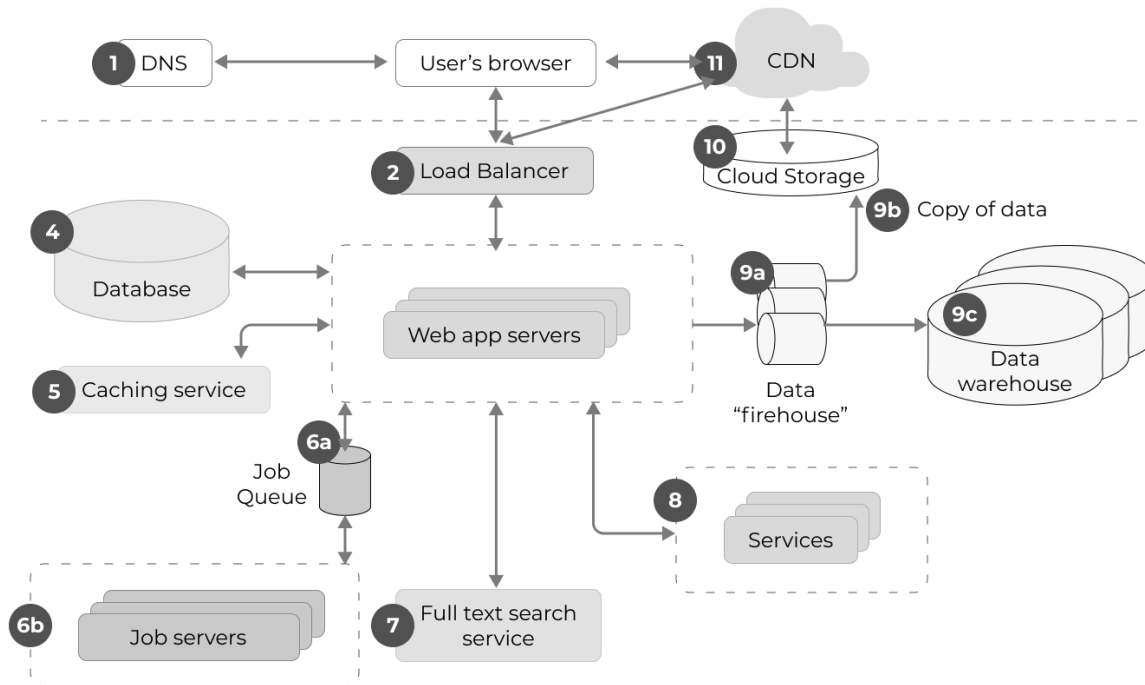


Figure 26. Budiga Hardware Detailed Design Diagram

7.2 Software Detailed Design

Table 5: Sales Module

ID	001
Title	Sales Module
Definition	This module provides a graphical representation of compiled data related to sales. It allows the user to analyze and gain insights into sales trends, overview data, and inventory sales within specified timeframes. The module assists in making informed decisions and monitoring the performance of the sales process.
User/s	Admin only
Data Structure	<pre> public class InventorySalesModel : ObservableObject { private int _id; private int _unitsSold; private float _totalSales; private int _totalTransactions; private string _date; </pre>

	<pre> private ItemModel _item; private ObservableCollection<InventorySalesModel> _inventorySales; public int Id { get { return _id; } set { _id = value; OnPropertyChanged("Id"); } } public int UnitsSold { get { return _unitsSold; } set { _unitsSold = value; OnPropertyChanged("UnitsSold"); } } public float TotalSales { get { return _totalSales; } set { _totalSales = value; OnPropertyChanged("TotalSales"); } } public int TotalTransaction { get { return _totalTransactions; } set { _totalTransactions = value; OnPropertyChanged("TotalTransaction"); } } public string Date { get { return _date; } set { _date = value; OnPropertyChanged("Date"); } } public ItemModel Item { get { return _item; } set { _item = value; OnPropertyChanged("Item"); } } public ObservableCollection<InventorySalesModel> InventorySales { get { return _inventorySales; } set { _inventorySales = value; OnPropertyChanged("InventorySales"); } } } public class OverviewSalesModel : ObservableObject { private int _unitsSold; private float _total; private string _date; private ItemModel _item; private ObservableCollection<OverviewSalesModel> _overviewSales; public int UnitsSold { get { return _unitsSold; } set { _unitsSold = value; OnPropertyChanged("UnitsSold"); } } public float Total { get { return _total; } set { _total = value; OnPropertyChanged("Total"); } } public string Date { get { return _date; } set { _date = value; OnPropertyChanged("Date"); } } public ItemModel Item { get { return _item; } set { _item = value; OnPropertyChanged("Item"); } } public ObservableCollection<OverviewSalesModel> OverviewSales { get { return _overviewSales; } set { _overviewSales = value; OnPropertyChanged("OverviewSales"); } } } </pre>
Main Component	<pre> public class SalesViewModel : ObservableObject { private SalesRepository salesRepository; // Overview Sales Model private OverviewSalesModel _overviewSales; public OverviewSalesModel OverviewSales { get; set; } // Inventory Sales Model private InventorySalesModel _sales; public InventorySalesModel Sales { get; set; } public RelayCommand GetAllCommand { get; set; } public RelayCommand OverviewViewCommand { get; set; } public RelayCommand InventoryViewCommand { get; set; } } </pre>

	<pre> public SalesOverviewViewModel SalesOverviewVM { get; set; } public SalesInventoryViewModel SalesInventoryVM { get; set; } private object _currentView; public object CurrentView { get { return _currentView; } set { _currentView = value; OnPropertyChanged(); } } private static SalesViewModel salesVM; public static SalesViewModel GetInstance { get { if (salesVM == null) { salesVM = new SalesViewModel(); } return salesVM; } } public SalesViewModel() { salesRepository = new SalesRepository(); // Overview Sales Instantiate _overviewSales = new OverviewSalesModel(); OverviewSales = new OverviewSalesModel(); // Inventory Sales Instantiate _sales = new InventorySalesModel(); Sales = new InventorySalesModel(); try { // Create Sales Overview and Inventory View Models SalesOverviewVM = new SalesOverviewViewModel(); SalesInventoryVM = new SalesInventoryViewModel(); // Set Current View to Sales Overview View Model by default CurrentView = SalesOverviewVM; // Command to switch to Sales Overview View OverviewViewCommand = new RelayCommand(o => { CurrentView = SalesOverviewVM; }); // Command to switch to Sales Inventory View </pre>
--	--

	<pre> InventoryViewCommand = new RelayCommand(o => { CurrentView = SalesInventoryVM; }); } catch (Exception ex) { Debug.WriteLine(ex.StackTrace); } // Command to retrieve all sales data GetAllCommand = new RelayCommand(param => GetAll((string)param)); GetAll("Accumulated"); } private void GetAll(string date) { // Retrieve total sales and total transactions Sales.TotalSales = salesRepository.GetTotalSales(date); Sales.TotalTransaction = salesRepository.GetTotalTransactions(date); // Retrieve inventory sales _sales.InventorySales = salesRepository.GetAllSales(date); Sales.InventorySales = _sales.InventorySales; // Retrieve overview sales _overviewSales.OverviewSales = salesRepository.GetAllOverviewSales(date); OverviewSales.OverviewSales = _overviewSales.OverviewSales; } } </pre>
Subcomponents	N/A

Table 6: Inventory Module Software Service

ID	002
Title	Inventory Module
Definition	This module allows users to create, read, update, and delete products in the inventory, providing comprehensive control and management over the inventory data. These capabilities empower them to maintain an accurate and up-to-date inventory system, facilitating efficient product management and ensuring the availability of accurate information for decision-making purposes.
User/s	Both Admin and Employee
Data Structure	<pre> public class ItemModel : ObservableObject { private int _id; </pre>

	<pre> private string _barcode; private string _name; private string _brand; private float _price; private int _quantity; private bool _isDeleted; private ObservableCollection<ItemModel> _itemRecords; public int Id { get { return _id; } set { _id = value; OnPropertyChanged("Id"); } } public string Barcode { get { return _barcode; } set { _barcode = value; OnPropertyChanged("Barcode"); } } public string Name { get { return _name; } set { _name = value; OnPropertyChanged("Name"); } } public string Brand { get { return _brand; } set { _brand = value; OnPropertyChanged("Brand"); } } public float Price { get { return _price; } set { _price = value; OnPropertyChanged("Price"); } } public int Quantity { get { return _quantity; } set { _quantity = value; OnPropertyChanged("Quantity"); } } public bool IsDeleted { get { return _isDeleted; } set { _isDeleted = value; OnPropertyChanged("IsDeleted"); } } public ObservableCollection<ItemModel> ItemRecords { get { return _itemRecords; } set { _itemRecords = value; OnPropertyChanged("ItemRecords"); } } private void ItemRecords_CollectionChanged(object sender, NotifyCollectionChangedEventArgs e) { OnPropertyChanged("ItemRecords"); } } </pre>
Main Component	<pre> public class InventoryViewModel : ObservableObject { private ItemRepository itemRepository; private ItemModel _item; public ItemModel Item { get; set; } public RelayCommand AddItemCommand { get; set; } public RelayCommand EditItemCommand { get; set; } public RelayCommand SearchItemCommand { get; set; } public RelayCommand ItemHistoryCommand { get; set; } public InventoryViewModel() { itemRepository = new ItemRepository(); _item = new ItemModel(); Item = new ItemModel(); AddItemCommand = new RelayCommand(param => AddItem()); EditItemCommand = new RelayCommand(param => EditItem((ItemModel)param)); SearchItemCommand = new RelayCommand(param => SearchItem((string)param)); ItemHistoryCommand = new RelayCommand(param => ItemHistory()); GetAll(); } public void GetAll() { </pre>

	<pre> _item.ItemRecords = itemRepository.GetAllItems(); Item.ItemRecords = _item.ItemRecords; } private void SearchItem(string searchText = "") { Item.ItemRecords = new ObservableCollection<ItemModel>(_item.ItemRecords.Where(i => i.Name.ToLower().Contains(searchText.ToLower()) i.Brand.ToLower().Contains(searchText.ToLower()) i.Barcode.ToLower().Contains(searchText.ToLower())) .ToList()); } private void AddItem() { InventoryAddView inventoryAddView = new InventoryAddView(this); inventoryAddView.ShowDialog(); } private void EditItem(ItemModel item) { InventoryEditView inventoryEditView = new InventoryEditView(this, item); inventoryEditView.ShowDialog(); } private void ItemHistory() { InventoryHistoryView inventoryHistoryView = new InventoryHistoryView(this); inventoryHistoryView.ShowDialog(); } } </pre>
Subcomponents	<pre> public partial class InventoryAddView : Window { private InventoryViewModel _vm; public InventoryAddView(InventoryViewModel vm) { InitializeComponent(); _vm = vm; } private void NumberValidationTextBox(object sender, TextCompositionEventArgs e) { Regex regex = new Regex("[^0-9]+"); e.Handled = regex.IsMatch(e.Text); } private void AddBtn_Click(object sender, RoutedEventArgs e) { if (string.IsNullOrEmpty(productTextBox.Text) string.IsNullOrEmpty(brandTextBox.Text) string.IsNullOrEmpty(priceTextBox.Text) string.IsNullOrEmpty(qtyTextBox.Text)) { </pre>

	<pre> MessageBox.Show("Fill all empty fields!", "Error", MessageBoxButton.OK, MessageBoxImage.Error); } else { ItemModel item = new ItemModel() { Name = productTextBox.Text, Barcode = (string.IsNullOrEmpty(barcodeTextBox.Text)) ? "N/A" : barcodeTextBox.Text, Brand = brandTextBox.Text, Price = float.Parse(priceTextBox.Text), Quantity = int.Parse(qtyTextBox.Text) }; ItemRepository itemRepository = new ItemRepository(); if (itemRepository.AddItem(item)) { ItemHistoryRepository itemHistoryRepository = new ItemHistoryRepository(); itemHistoryRepository.AddItemHistory(item, "ADDED"); _vm.GetAll(); this.Close(); } } private void CancelBtn_Click(object sender, RoutedEventArgs e) { this.Close(); } } public partial class InventoryEditView : Window { private InventoryViewModel _vm; private ItemModel _item; public InventoryEditView(InventoryViewModel vm, ItemModel item) { InitializeComponent(); _vm = vm; _item = item; productTextBox.Text = item.Name; barcodeTextBox.Text = (item.Barcode != null) ? item.Barcode.ToString() : "N/A"; brandTextBox.Text = item.Brand; priceTextBox.Text = item.Price.ToString(); qtyTextBlock.Text = item.Quantity.ToString(); } private void NumberValidationTextBox(object sender, TextCompositionEventArgs e) { Regex regex = new Regex("[^0-9]+"); e.Handled = regex.IsMatch(e.Text); } } </pre>
--	---

	<pre> private void PlusBtn_Click(object sender, RoutedEventArgs e) { qtyTextBlock.Text = (int.Parse(qtyTextBlock.Text)+int.Parse(qtyTextBox.Text)).ToString(); } private void UpdateBtn_Click(object sender, RoutedEventArgs e) { if (string.IsNullOrEmpty(productTextBox.Text) string.IsNullOrEmpty(brandTextBox.Text) string.IsNullOrEmpty(priceTextBox.Text) string.IsNullOrEmpty(qtyTextBox.Text)) { MessageBox.Show("Fill all empty fields!", "Error", MessageBoxButton.OK, MessageBoxImage.Error); } else { ItemModel item = new ItemModel() { Id = _item.Id, Barcode = (string.IsNullOrEmpty(barcodeTextBox.Text)) ? "N/A" : barcodeTextBox.Text, Name = productTextBox.Text, Brand = brandTextBox.Text, Price = float.Parse(priceTextBox.Text), Quantity = int.Parse(qtyTextBlock.Text) }; IRepository itemRepository = new ItemRepository(); if (itemRepository.UpdateItem(item)) { ItemHistoryRepository itemHistoryRepository = new ItemHistoryRepository(); itemHistoryRepository.AddItemHistory(_item, "UPDATED"); _vm.GetAll(); this.Close(); } } private void DeleteBtn_Click(object sender, RoutedEventArgs e) { if (MessageBox.Show("Continue Action?", "Warning", MessageBoxButton.YesNo, MessageBoxImage.Warning) == MessageBoxResult.Yes) { IRepository itemRepository = new ItemRepository(); if (itemRepository.DeleteItem(_item.Id)) { ItemHistoryRepository itemHistoryRepository = new ItemHistoryRepository(); itemHistoryRepository.AddItemHistory(_item, "DELETED"); _vm.GetAll(); } } } </pre>
--	---

	<pre> this.Close(); } } private void CancelBtn_Click(object sender, RoutedEventArgs e) { this.Close(); } } </pre>
--	--

Table 7: Employee Module Software Service

ID	003
Title	Employee Monitor Module
Definition	This module allows users to create, read, update, and delete employees in the system. This is specifically for managing employees, which includes registering new employees and viewing their attendance history
User/s	Admin only
Data Structure	<pre> public class UserModel : ObservableObject { private int _id; private string _fname; private string _lname; private string _username; private string _password; private string _contact; private string _userRole; private DateTime _createdDate; private DateTime? _updatedDate; private ObservableCollection<UserModel> _userRecords; public int Id { get { return _id; } set { _id = value; OnPropertyChanged("Id"); } } public string FName { get { return _fname; } set { _fname = value; OnPropertyChanged("FName"); } } public string LName { get { return _lname; } set { _lname = value; OnPropertyChanged("LName"); } } public string Username { get { return _username; } set { _username = value; OnPropertyChanged("Username"); } } public string Password { get { return _password; } set { _password = value; OnPropertyChanged("Password"); } } public string Contact { get { return _contact; } set { _contact = value; OnPropertyChanged("Contact"); } } public string UserRole { get { return _userRole; } set {_userRole = value; OnPropertyChanged("UserRole"); } } public DateTime CreatedDate { get { return _createdDate; } set { _createdDate = value; OnPropertyChanged("CreatedDate"); } } public DateTime? UpdatedDate { get { return _updatedDate; } set { _updatedDate = value; </pre>

	<pre> OnPropertyChanged("UpdateDate"); } } public ObservableCollection<UserModel> UserRecords { get { return _userRecords; } set { _userRecords = value; OnPropertyChanged("UserRecords"); } } private void UserRecords_CollectionChanged(object sender, NotifyCollectionChangedEventArgs e) { OnPropertyChanged("UserRecords"); } } </pre>
Main Component	<pre> public class EmployeeMainViewModel : ObservableObject { public RelayCommand InventoryViewCommand { get; set; } public RelayCommand InvoiceViewCommand { get; set; } public InventoryViewModel InventoryVM { get; set; } public InvoiceViewModel InvoiceVM { get; set; } private object _currentView; public object CurrentView { get { return _currentView; } set { _currentView = value; OnPropertyChanged(); } } public EmployeeMainViewModel() { try { InventoryVM = new InventoryViewModel(); InvoiceVM = new InvoiceViewModel(); CurrentView = InventoryVM; InventoryViewCommand = new RelayCommand(o => { CurrentView = InventoryVM; }); InvoiceViewCommand = new RelayCommand(o => { CurrentView = InvoiceVM; }); } catch (Exception ex) { Debug.WriteLine(ex.StackTrace); } } } </pre>
Subcomponents	<pre> public partial class EmployeeAddView : Window { EmployeeViewModel _vm; public EmployeeAddView(EmployeeViewModel vm) </pre>

	<pre> { InitializeComponent(); _vm = vm; } private void NumberValidationTextBox(object sender, TextCompositionEventArgs e) { Regex regex = new Regex("[^0-9]+"); e.Handled = regex.IsMatch(e.Text); } private void AddBtn_Click(object sender, RoutedEventArgs e) { if (string.IsNullOrEmpty(fNameTextBox.Text) string.IsNullOrEmpty(lNameTextBox.Text) string.IsNullOrEmpty(usernameTextBox.Text) string.IsNullOrEmpty(contactTextBox.Text) string.IsNullOrEmpty(passwordTextBox.Text)) { MessageBox.Show("Fill all empty fields!", "Error", MessageBoxButton.OK, MessageBoxImage.Error); } else { UserModel user = new UserModel() { FName = fNameTextBox.Text, LName = lNameTextBox.Text, Username = usernameTextBox.Text, Password = passwordTextBox.Text, Contact = contactTextBox.Text, UserRole = "Employee", }; UserRepository userRepository = new UserRepository(); if (userRepository.AddEmployeeUser(user)) { _vm.GetAllEmployee(); this.Close(); } } } private void CancelBtn_Click(object sender, RoutedEventArgs e) { this.Close(); } private void CancelBtn_Click(object sender, RoutedEventArgs e) { this.Close(); } } public partial class InventoryEditView : Window </pre>
--	---

```

{
    private InventoryViewModel _vm;
    private ItemModel _item;

    public InventoryEditView(InventoryViewModel vm,
ItemModel item)
    {
        InitializeComponent();
        _vm = vm;
        _item = item;
        productTextBox.Text = item.Name;
        barcodeTextBox.Text = (item.Barcode != null) ?
item.Barcode.ToString() : "N/A";
        brandTextBox.Text = item.Brand;
        priceTextBox.Text = item.Price.ToString();
        qtyTextBlock.Text = item.Quantity.ToString();
    }

    private void NumberValidationTextBox(object sender,
TextCompositionEventArgs e)
    {
        Regex regex = new Regex("[^0-9]+");
        e.Handled = regex.IsMatch(e.Text);
    }

    private void PlusBtn_Click(object sender,
RoutedEventArgs e)
    {
        qtyTextBlock.Text =
(int.Parse(qtyTextBlock.Text)+int.Parse(qtyTextBox.Text)).ToStri
ng();
    }
    private void UpdateBtn_Click(object sender,
RoutedEventArgs e)
    {
        if (string.IsNullOrEmpty(productTextBox.Text) ||
string.IsNullOrEmpty(brandTextBox.Text) ||
string.IsNullOrEmpty(priceTextBox.Text) ||
string.IsNullOrEmpty(qtyTextBox.Text))
        {
            MessageBox.Show("Fill all empty fields!",
"Error", MessageBoxButton.OK, MessageBoxImage.Error);
        }
        else
        {
            ItemModel item = new ItemModel()
            {
                Id = _item.Id,
                Barcode =
(string.IsNullOrEmpty(barcodeTextBox.Text)) ? "N/A" :
barcodeTextBox.Text,
                Name = productTextBox.Text,
                Brand = brandTextBox.Text,
                Price = float.Parse(priceTextBox.Text),
                Quantity = int.Parse(qtyTextBlock.Text)
            };
            ItemRepository itemRepository = new
ItemRepository();
            if (itemRepository.UpdateItem(item))
            {
                ItemHistoryRepository itemHistoryRepository
= new ItemHistoryRepository();

```

	<pre> itemHistoryRepository.AddItemHistory(_item, "UPDATED"); _vm.GetAll(); this.Close(); } } private void DeleteBtn_Click(object sender, RoutedEventArgs e) { if (MessageBox.Show("Continue Action?", "Warning", MessageBoxButton.YesNo, MessageBoxImage.Warning) == MessageBoxResult.Yes) { ItemRepository itemRepository = new ItemRepository(); if (itemRepository.DeleteItem(_item.Id)) { ItemHistoryRepository itemHistoryRepository = new ItemHistoryRepository(); itemHistoryRepository.AddItemHistory(_item, "DELETED"); _vm.GetAll(); this.Close(); } } private void CancelBtn_Click(object sender, RoutedEventArgs e) { this.Close(); } } public partial class EmployeeEditView : Window { private EmployeeViewModel _vm; private int userId; public EmployeeEditView(UserModel user, EmployeeViewModel vm) { InitializeComponent(); _vm = vm; userId = user.Id; fNameTextBox.Text = user.FName; lNameTextBox.Text = user.LName; usernameTextBox.Text = user.Username; passwordTextBox.Text = user.Password; contactTextBox.Text = user.Contact; } private void NumberValidationTextBox(object sender, TextCompositionEventArgs e) { Regex regex = new Regex("[^0-9]+"); e.Handled = regex.IsMatch(e.Text); } } </pre>
--	---

```

        private void UpdateBtn_Click(object sender,
RoutedEventArgs e)
        {
            if (string.IsNullOrEmpty(fNameTextBox.Text) ||
string.IsNullOrEmpty(lNameTextBox.Text) ||
string.IsNullOrEmpty(usernameTextBox.Text) ||
string.IsNullOrEmpty(contactTextBox.Text) ||
string.IsNullOrEmpty(passwordTextBox.Text))
            {
                MessageBox.Show("Fill all empty fields!",
"Error", MessageBoxButton.OK, MessageBoxImage.Error);
            }
            else
            {
                UserModel user = new UserModel()
                {
                    Id = userId,
                    FName = fNameTextBox.Text,
                    LName = lNameTextBox.Text,
                    Username = usernameTextBox.Text,
                    Password = passwordTextBox.Text,
                    Contact = contactTextBox.Text,
                };
                UserRepository userRepository = new
UserRepository();
                if (userRepository.UpdateUser(user))
                {
                    _vm.GetAllEmployee();
                    this.Close();
                }
            }

            private void CancelBtn_Click(object sender,
RoutedEventArgs e)
            {
                this.Close();
            }
        }

public class InventoryHistoryViewModel
{
    private ItemHistoryRepository itemHistoryRepository;
    public ItemHistoryModel ItemHistory { get; set; }
    public RelayCommand UndoActionCommand { get; set; }

    public InventoryHistoryViewModel()
    {
        itemHistoryRepository = new ItemHistoryRepository();
        ItemHistory = new ItemHistoryModel();
        UndoActionCommand = new RelayCommand(param =>
UndoAction((ItemHistoryModel) param));
        GetAll();
    }

    public void GetAll()
    {
        ItemHistory.ItemHistoryRecords =
itemHistoryRepository.GetAllItemHistory();
    }
}

```

	<pre> private void UndoAction(ItemHistoryModel item) { itemHistoryRepository.UndoAction(item); } } </pre>
--	---

Table 8: Invoice Module Software Service

ID	004
Title	Invoice Module
Definition	This module allows users the authority to generate invoices for transactions and access a comprehensive history of past transactions. Users can create invoices that accurately document customer purchases and financial transactions.
User/s	Both Admin and Employee
Data Structure	<pre> public class InvoiceModel : ObservableObject { private int _id; private int _userId; private float _totalPrice; private float _customerPay; private float _customerChange; private DateTime _createdDate; private UserModel _user; private ObservableCollection<InvoiceModel> _invoiceRecords; private ObservableCollection<OrderModel> _invoiceOrderRecords; public int Id { get { return _id; } set { _id = value; OnPropertyChanged("Id"); } } public int UserId { get { return _userId; } set { _userId = value; OnPropertyChanged("UserId"); } } public float TotalPrice { get { return _totalPrice; } set { _totalPrice = value; OnPropertyChanged("TotalPrice"); } } public float CustomerPay { get { return _customerPay; } set { _customerPay = value; OnPropertyChanged("CustomerPay"); } } public float CustomerChange { get { return _customerChange; } set { _customerChange = value; OnPropertyChanged("CustomerChange"); } } public DateTime CreatedDate { get { return _createdDate; } set { _createdDate = value; OnPropertyChanged("CreatedDate"); } } public UserModel User { get { return _user; } set { _user = value; OnPropertyChanged("User"); } } public ObservableCollection<InvoiceModel> InvoiceRecords { get { return _invoiceRecords; } set { _invoiceRecords = value; OnPropertyChanged("InvoiceRecords"); } } public ObservableCollection<OrderModel> InvoiceOrderRecords { get { return _invoiceOrderRecords; } set { _invoiceOrderRecords = value; OnPropertyChanged("InvoiceOrderRecords"); } } private void InvoiceRecords_CollectionChanged(object </pre>

	<pre> sender, NotifyCollectionChangedEventArgs e) { OnPropertyChanged("InvoiceRecords"); } private void InvoiceOrderRecords_CollectionChanged(object sender, NotifyCollectionChangedEventArgs e) { OnPropertyChanged("InvoiceOrderRecords"); } } </pre>
Main Component	<pre> public class InvoiceViewModel { public RelayCommand AddQuantityCommand { get; set; } public RelayCommand ReduceQuantityCommand { get; set; } public RelayCommand RemoveItemCommand { get; set; } public RelayCommand AddItemCommand { get; set; } public RelayCommand TransactionHistoryCommand { get; set; } public RelayCommand CancelOrderCommand { get; set; } public RelayCommand CheckoutCommand { get; set; } public InvoiceModel Invoice { get; set; } public InvoiceViewModel() { Invoice = new InvoiceModel(); Invoice.InvoiceOrderRecords = new ObservableCollection<OrderModel>(); AddQuantityCommand = new RelayCommand(param => AddQuantity((int)param)); ReduceQuantityCommand = new RelayCommand(param => ReduceQuantity((int)param)); RemoveItemCommand = new RelayCommand(param => RemoveItem((int)param)); AddItemCommand = new RelayCommand(param => AddItem()); TransactionHistoryCommand = new RelayCommand(param => TransactionHistory()); CancelOrderCommand = new RelayCommand(param => CancelOrder()); CheckoutCommand = new RelayCommand(param => Checkout()); } private void AddQuantity(int id) { OrderModel order = Invoice.InvoiceOrderRecords.Where(i => i.ItemId == id).FirstOrDefault(); if (order.Quantity < order.Item.Quantity) { order.Quantity += 1; CalculateSubtotal(order); CalculateTotal(); } } private void ReduceQuantity(int id) { OrderModel order = Invoice.InvoiceOrderRecords.Where(i => i.ItemId == id).FirstOrDefault(); </pre>

	<pre> if (order.Quantity > 1) { order.Quantity -= 1; CalculateSubtotal(order); CalculateTotal(); } } private void RemoveItem(int id) { OrderModel order = Invoice.InvoiceOrderRecords.Where(i => i.ItemId == id).FirstOrDefault(); Invoice.InvoiceOrderRecords.Remove(order); CalculateTotal(); } private void AddItem() { InvoiceAddView invoiceAddView = new InvoiceAddView(this); invoiceAddView.ShowDialog(); } private void TransactionHistory() { InvoiceHistoryView invoiceHistoryView = new InvoiceHistoryView(); invoiceHistoryView.ShowDialog(); } public void CancelOrder() { Invoice.InvoiceOrderRecords.Clear(); CalculateTotal(); } private void Checkout() { if (Invoice.InvoiceOrderRecords.Count > 0) { CalculateTotal(); InvoicePayView invoicePayView = new InvoicePayView(this, Invoice); invoicePayView.ShowDialog(); } else { MessageBox.Show("Invoice list is empty!", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error); } } public void GetItem(ItemModel item) { OrderModel order = Invoice.InvoiceOrderRecords.Where(i => i.ItemId == item.Id).FirstOrDefault(); InvoiceAddQuantityView invoiceAddQuantityView = new InvoiceAddQuantityView(); if (invoiceAddQuantityView.ShowDialog() == true) { </pre>
--	--

	<pre> if (order == null && invoiceAddQuantityView.Quantity <= item.Quantity) { Invoice.InvoiceOrderRecords.Add(new OrderModel() { ItemId = item.Id, Quantity = invoiceAddQuantityView.Quantity, SubtotalPrice = item.Price * invoiceAddQuantityView.Quantity, Item = item, }); CalculateTotal(); MessageBox.Show("Successfully added item to invoice", "Success", MessageBoxButton.OK, MessageBoxImage.Information); } else if (order != null && order.Quantity + invoiceAddQuantityView.Quantity <= item.Quantity) { order.Quantity += invoiceAddQuantityView.Quantity; CalculateSubtotal(order); CalculateTotal(); MessageBox.Show("Successfully updated item quantity to invoice", "Success", MessageBoxButton.OK, MessageBoxImage.Information); } else { MessageBox.Show("Quantity exceeds actual product quantity", "Error", MessageBoxButton.OK, MessageBoxImage.Error); } } public void GetItemByBarcode(string barcode) { ItemModel item = new ItemModel(); ItemRepository itemRepository = new ItemRepository(); OrderModel order = Invoice.InvoiceOrderRecords.Where(i => i.ItemId == item.Id).FirstOrDefault(); item = itemRepository.GetItemByBarcode(barcode); if (item.Id != -1) { InvoiceAddQuantityView invoiceAddQuantityView = new InvoiceAddQuantityView(); if (invoiceAddQuantityView.ShowDialog() == true) { if (order == null && invoiceAddQuantityView.Quantity <= item.Quantity) { Invoice.InvoiceOrderRecords.Add(new OrderModel() { ItemId = item.Id, Quantity = </pre>
--	--

	<pre> invoiceAddQuantityView.Quantity, SubtotalPrice = item.Price * invoiceAddQuantityView.Quantity, Item = item, }); CalculateTotal(); MessageBox.Show("Successfully added item to invoice", "Success", MessageBoxButton.OK, MessageBoxImage.Information); } else if(order != null && order.Quantity + invoiceAddQuantityView.Quantity <= item.Quantity) { order.Quantity += invoiceAddQuantityView.Quantity; CalculateSubtotal(order); CalculateTotal(); MessageBox.Show("Successfully updated item quantity to invoice", "Success", MessageBoxButton.OK, MessageBoxImage.Information); } else { MessageBox.Show("Quantity exceeds actual product quantity", "Error", MessageBoxButton.OK, MessageBoxImage.Error); } } } else { MessageBox.Show("Item does not exist in inventory", "Error", MessageBoxButton.OK, MessageBoxImage.Error); } } private void CalculateSubtotal(OrderModel order) { order.SubtotalPrice = order.Quantity * order.Item.Price; } private void CalculateTotal() { Invoice.TotalPrice = 0; foreach(OrderModel order in Invoice.InvoiceOrderRecords) { Invoice.TotalPrice += order.SubtotalPrice; } } } </pre>
Subcomponents	<pre> public class InvoiceAddViewModel { private IRepository itemRepository; private ItemModel _item; public ItemModel Item { get; set; } public RelayCommand SearchItemCommand { get; set; } } </pre>

	<pre> public InvoiceAddViewModel() { itemRepository = new ItemRepository(); _item = new ItemModel(); Item = new ItemModel(); SearchItemCommand = new RelayCommand(param => SearchItem((string)param)); GetAll(); } public void GetAll() { _item.ItemRecords = itemRepository.GetAllItems(); Item.ItemRecords = _item.ItemRecords; } private void SearchItem(string searchText = "") { Item.ItemRecords = new ObservableCollection<ItemModel>(_item.ItemRecords.Where(i => i.Name.ToLower().Contains(searchText.ToLower()) i.Brand.ToLower().Contains(searchText.ToLower()) i.Barcode.ToLower().Contains(searchText.ToLower())) .ToList()); } } public class InvoiceHistoryViewModel { private InvoiceModel _invoice; private InvoiceRepository invoiceRepository; public InvoiceModel Invoice { get; set; } public RelayCommand GetReceiptCommand { get; set; } public RelayCommand SearchItemCommand { get; set; } public InvoiceHistoryViewModel() { _invoice = new InvoiceModel(); Invoice = new InvoiceModel(); invoiceRepository = new InvoiceRepository(); GetReceiptCommand = new RelayCommand(param => GetReceipt((InvoiceModel)param)); SearchItemCommand = new RelayCommand(param => SearchItem((string)param)); GetAll(); } private void GetAll() { _invoice.InvoiceRecords = invoiceRepository.GetAllInvoice(); Invoice.InvoiceRecords = _invoice.InvoiceRecords; } private void GetReceipt(InvoiceModel invoice) { InvoiceReceiptView invoiceReceiptView = new InvoiceReceiptView(invoice); invoiceReceiptView.ShowDialog(); } } </pre>
--	---

	<pre> } private void SearchItem(string searchText = "") { Invoice.InvoiceRecords = new ObservableCollection<InvoiceModel>(_invoice.InvoiceRecords.Where(i => i.InvoiceOrderRecords.Where(o => o.InvoiceId.ToString().Equals(searchText.ToLower()) o.Item.Name.ToLower().Contains(searchText.ToLower()) o.Item.Brand.ToLower().Contains(searchText.ToLower()))).Any() == true).ToList()); } } </pre>
--	---

Table 9: Scanner Module Software Service

ID	005
Title	Scanner Module
Definition	This module empowers the user with the capability to utilize a barcode scanner to effortlessly scan and decode product barcodes. This functionality facilitates the seamless addition of products to the invoice, as the scanner quickly captures the necessary product information and decodes it.
User/s	Both Admin and Employee
Data Structure	N/A
Main Component	<pre> public partial class InvoiceAddView : Window { private InvoiceViewModel _invoiceVM; public InvoiceAddView(InvoiceViewModel invoiceVM) { InitializeComponent(); _invoiceVM = invoiceVM; } private void InventoryTable_DoubleClick(object sender, MouseButtonEventArgs e) { if (inventoryTable.SelectedItem == null) return; ItemModel selectedItem = inventoryTable.SelectedItem as ItemModel; _invoiceVM.GetItem(selectedItem); this.Close(); } FilterInfoCollection filterInfoCollection; VideoCaptureDevice videoCaptureDevice; private void ScannerControl_Loaded(object sender, RoutedEventArgs e) </pre>

	<pre> { filterInfoCollection = new FilterInfoCollection(FilterCategory.VideoInputDevice); foreach (FilterInfo device in filterInfoCollection) { deviceCmbBox.Items.Add(device.Name); } deviceCmbBox.SelectedIndex = 0; Window window = Window.GetWindow(this); window.Closing += window_Closing; } private void window_Closing(object sender, global::System.ComponentModel.CancelEventArgs e) { if (videoCaptureDevice != null) { if (videoCaptureDevice.IsRunning) { videoCaptureDevice.SignalToStop(); videoCaptureDevice.NewFrame -= new NewFrameEventHandler(VideoCaptureDevice_NewFrame); videoCaptureDevice = null; } } } private void scanBtn_Checked(object sender, RoutedEventArgs e) { scanBtn.Content = "Stop Scanning"; videoCaptureDevice = new VideoCaptureDevice(filterInfoCollection[deviceCmbBox.SelectedIndex].MonikerString); videoCaptureDevice.NewFrame += new NewFrameEventHandler(VideoCaptureDevice_NewFrame); videoCaptureDevice.Start(); } private void scanBtn_Unchecked(object sender, RoutedEventArgs e) { if (videoCaptureDevice != null) { if (videoCaptureDevice.IsRunning) { scanBtn.Content = "Start Scanning"; videoCaptureDevice.SignalToStop(); videoCaptureDevice.NewFrame -= new NewFrameEventHandler(VideoCaptureDevice_NewFrame); videoCaptureDevice = null; } } } [DllImport("gdi32.dll", EntryPoint = "DeleteObject")] [return: MarshalAs(UnmanagedType.Bool)] public static extern bool DeleteObject([In] IntPtr hObject); public ImageSource ImageSourceFromBitmap(Bitmap bmp) </pre>
--	---

```

        {
            var handle = bmp.GetHbitmap();
            try
            {
                return
                Imaging.CreateBitmapSourceFromHBitmap(handle, IntPtr.Zero,
                Int32Rect.Empty, BitmapSizeOptions.FromEmptyOptions());
            }
            finally { DeleteObject(handle); }
        }

        private void VideoCaptureDevice_NewFrame(object sender,
        NewFrameEventArgs eventArgs)
        {
            Bitmap img = (Bitmap)eventArgs.Frame.Clone();
            BarcodeReader reader = new BarcodeReader(null, null,
            ls => new GlobalHistogramBinarizer(ls))
            {
                Options = new DecodingOptions
                {
                    TryHarder = true,
                    //PureBarcode = true,
                    PossibleFormats = new List<BarcodeFormat>
                    {
                        BarcodeFormat.EAN_13
                        //BarcodeFormat.CODE_128
                        //BarcodeFormat.EAN_8,
                        //BarcodeFormat.CODE_39,
                        //BarcodeFormat.UPC_A
                    }
                }
            };
            var result = reader.Decode(img);
            this.Dispatcher.Invoke(new Action(() =>
            {
                if (result != null)
                {
                    outputBlock.Text = result.Text;
                    _invoiceVM.GetItemByBarcode(result.Text);
                    if (videoCaptureDevice != null)
                    {
                        if (videoCaptureDevice.IsRunning)
                        {
                            scanBtn.Content = "Start Scanning";
                            videoCaptureDevice.SignalToStop();
                            videoCaptureDevice.NewFrame -= new
                            NewFrameEventHandler(VideoCaptureDevice_NewFrame);
                            videoCaptureDevice = null;
                        }
                    }
                    this.Close();
                }
                else
                {
                    outputBlock.Text = "No item scanned!";
                }
                scannerCamera.Source =
                ImageSourceFromBitmap(img);
            }));
        }
    }

```

Subcomponents	N/A
----------------------	-----

7.3 Security Detailed Design

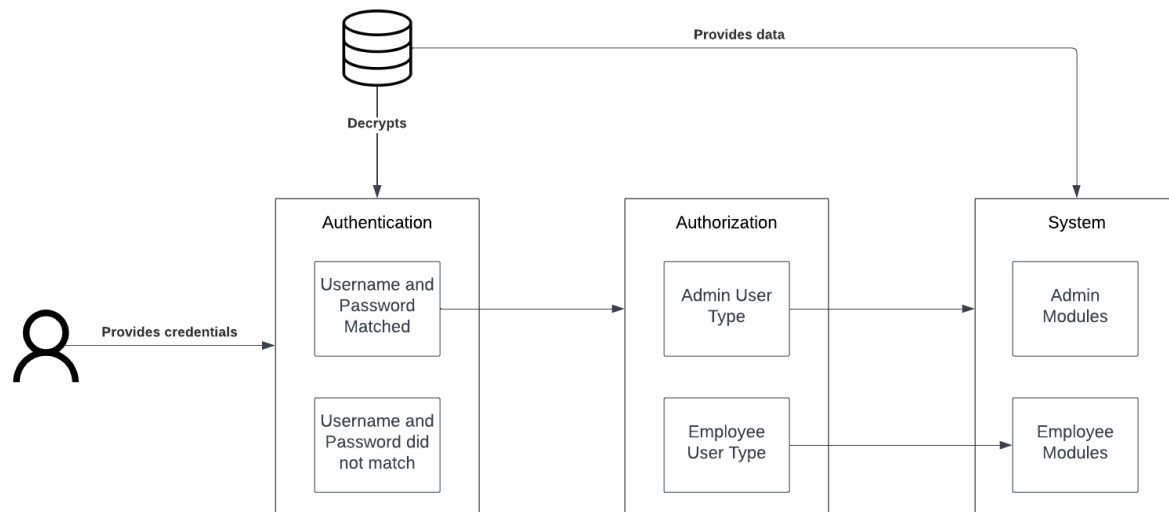


Figure 27. Budiga Security Detailed Design Diagram

7.4 Performance Detailed Design

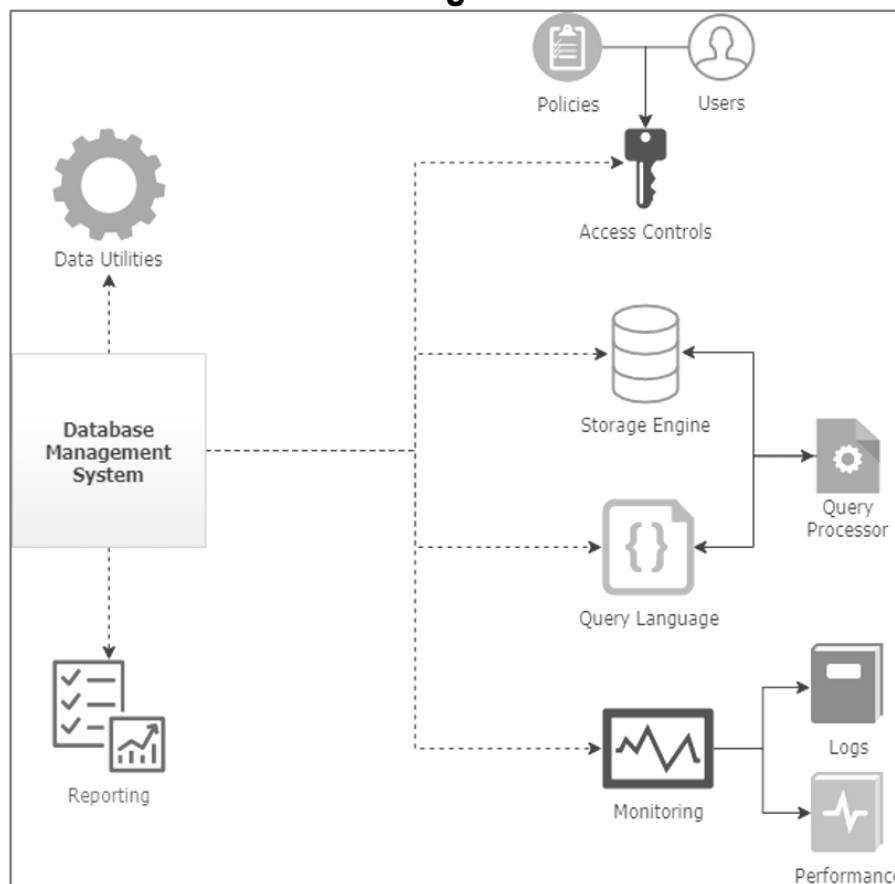


Figure 28. Budiga Performance Diagram

- Storage engine - core component of the DBMS that interacts with the file system at an OS level to store data
- Query language - required for interacting with a database, from creating databases to simply inserting or retrieving data
- Query processor - intermediary between the user queries and the database
- Optimization engine - allows the DBMS to provide insights into the performance of the database in terms of optimizing the database itself and queries
- Metadata catalog - centralized catalog of all the objects within the database
- Log manager - keeps all the logs of the DBMS
- Reporting and monitoring tools - enable users to generate reports while monitoring tools enable monitoring the databases for resource consumption, and user activity
- Data utilities - in addition to all the above, most DBMS software comes with additional inbuilt utilities to provide functionality

7.5 Internal Communications Detailed Design

The proposed solution utilizes existing communication protocols and methods. There are no additional components, servers, or applications to communicate with. The proposed solution will simply extend the current capabilities.

8. System Integrity Controls

Table 10: System Design Specifications

Design Specifications	
Internal Security	<ul style="list-style-type: none"> • Implement role-based access control (RBAC) to restrict access to critical data items based on user roles and privileges. • Use strong authentication mechanisms, such as username and password, to ensure authorized access. • Implement encryption techniques to protect sensitive data in transit and at rest. • Implement access control lists (ACLs) to define specific access permissions for different user types. • Regularly review and update access privileges to align with user requirements and job roles.
Audit Procedures	<ul style="list-style-type: none"> • Establish comprehensive audit procedures to track and record system activities, including operational and management reports. • Define retention periods for audit logs and ensure compliance with relevant regulatory requirements. • Regularly review and analyze audit logs to identify any suspicious activities or anomalies. • Implement backup and recovery mechanisms to ensure the availability and integrity of audit logs.
Application Audit Trails	<ul style="list-style-type: none"> • Implement an application-level audit trail to log retrieval access to designated critical data. • Include relevant information such as user identification, network terminal identification, date, time, and the accessed or changed data. • Ensure that the audit trail captures all relevant events and activities related to critical data access and changes. • Implement mechanisms to protect the integrity and confidentiality of the audit trail data.
Standard Tables for Data Validation	<ul style="list-style-type: none"> • Define and maintain standard tables for validating data fields within the system. • Establish data validation rules and ensure that data entered into the system complies with these rules. • Regularly update and maintain the standard tables to reflect any changes in data validation requirements. • Implement validation checks at various stages of data entry and processing to ensure data integrity and consistency.
Verification Processes for Critical Data Modifications	<ul style="list-style-type: none"> • Implement robust verification processes for additions, deletions, or updates of critical data.

	<ul style="list-style-type: none">• Define appropriate authorization levels and approval workflows for data modifications.• Implement validation checks and confirmation mechanisms to ensure the accuracy and integrity of critical data modifications.• Maintain a log of all critical data modifications, including user identification, date, time, and nature of the modification.
User Identification and Audit Information	<ul style="list-style-type: none">• Implement user identification mechanisms to uniquely identify users accessing the system.• Capture and log user identification, network terminal identification, date, time, and the specific data accessed or changed during each user session.• Ensure that the audit information is securely stored and accessible only to authorized personnel.• Implement mechanisms to easily retrieve and analyze audit information based on user identification, network terminal identification, date, time, and data accessed or changed

9. External Interfaces

9.1 Interface Architecture

The current and proposed solutions utilize a services oriented architecture. The proposed system will utilize the existing interface architecture by implementing a REST framework.

Representational State Transfer (REST)

This is a style or framework for designing integrated applications or services over HTTP. The proposed solution will implement a true RESTful API for interapplication integration and regional coordination. This achieves the following results from an interface architecture perspective:

- Uniform interface
- Client-server
- Stateless
- Cacheable
- Layered system
- Code on demand

Data Exchange

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for developers to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

9.2 Interface Detailed Design

All third party applications and integrations will utilize a RESTful API that will be designed in the application development phase. The API will be open and published. This means that any developer or third party, if they provide the proper security credentials, can access application, database, and published functions. REST Model Clients make standard HTTP requests over an SSL channel and should always validate the certificate of the endpoint with which the client is communicating.

Response Codes

API will respond with standard HTTP response codes appropriate to the result of the request. While the exact meaning of the code varies depending on the request, the general rules are:

200.A response code of 200 means the request was successful and details about the response can be found in the body of the response.

201.The requested POST operation was successful and an object was created in the system.

202.The requested operation has been accepted and the body contains information about an asynchronous job you can query to check on the progress of the request.

204.The requested operation was successful and there is no response body.

- 307.** Please repeat the request using the provided URI. Subsequent requests can use the old URI.
- 400.** Your request was improperly formatted. You should verify that your request conforms to this specification and re-issue the request in a properly formatted manner.
- 404.** The requested resource does not exist.
- 409.** An operational error occurred. The most common reason is an error with the cloud provider itself, but it can also result from any number of cloud state issues.
- 418.** A request was made to create a resource, but the resource was not created and no job was returned.
- 500.** API failed to process the request because of an error inside the system.
- 501.** You requested an action against a resource in a cloud that does not support that action.
- 503.** API undergoing maintenance or is otherwise temporarily unavailable for API queries.

Response Entities

All GET methods respond with the JSON or XML of the resource(s) being requested. HEAD methods have no response entity. POST methods may respond with a 201 CREATED or 202 ACCEPTED response code depending on whether the creation is completed immediately or is an asynchronous operation. If the resource was created immediately, API should provide a JSON or XML entity that includes the new resource's unique ID. If the creation operation takes time, however, the response body will include a Job resource that can be tracked to completion.

PUT and DELETE methods generally respond with 204 NO CONTENT unless the operation is a long lived operation. In those scenarios, the PUT will respond with a 202 ACCEPTED response code and include a Job resource in the response entity.

Appendix A: Record of Changes

By implementing these control and tracking measures, the development and distribution of the Software Development Document in an Inventory Management and Point of Sales System for MSME is effectively managed. This ensures that the document remains accurate, up-to-date, and accessible to the relevant stakeholders throughout the system development life cycle. Provided below is the record of changes within the documentation of the system.

Table 11: Record of Changes

Version Number	Date	Author/Owner	Description of Change
1.0	05 / 24 / 2023	Budiga Corp	Initial Document Created

Appendix B: Acronyms

Table 12: Table of Acronyms

Acronym	Literal Translation
<i>ERD</i>	<i>Entity Relationship Diagram</i>
<i>MSME</i>	<i>Micro, Small and Medium Enterprises</i>
<i>POS</i>	<i>Point of Sales</i>
<i>QR</i>	<i>Quick Response</i>
<i>SKU</i>	<i>Stock Keeping Unit</i>
<i>API</i>	<i>Application Programming Interface</i>
<i>SQL</i>	<i>Structured Query Language</i>
<i>HTTP</i>	<i>Hypertext Transfer Protocol</i>
<i>URL</i>	<i>Uniform Resource Locator</i>
<i>DBMS</i>	<i>Database Management System</i>

Appendix C: Glossary

Table 13: Glossary

Term	Acronym	Definition
<i>Inventory Management</i>	<i>N/A</i>	<i>The process of overseeing and controlling a company's inventory, including acquisition, storage, tracking, and distribution of goods.</i>
<i>Micro, Small, and Medium Enterprises</i>	<i>MSME</i>	<i>Small-scale businesses with limited investment and a smaller workforce.</i>
<i>Point of Sales System</i>	<i>POS</i>	<i>Computerized system used in retail and other businesses for processing sales transactions, including hardware (e.g., cash registers, barcode scanners) and software for recording sales and managing inventory.</i>
<i>Stock Keeping Unit</i>	<i>SKU</i>	<i>A number (usually eight alphanumeric digits) that retailers assign to products to keep track of stock levels internally</i>
<i>Quick Response codes</i>	<i>QR Code</i>	<i>A type of barcode that can be read easily by a digital device and which stores information as a series of pixels in a square-shaped grid.</i>
<i>Invoice</i>	<i>N/A</i>	<i>A document given to the buyer by the seller to collect payment</i>
<i>Application Programming Interface</i>	<i>API</i>	<i>A software intermediary that allows two applications to talk to each other</i>
<i>Structured Query Language</i>	<i>SQL</i>	<i>A programming language used for managing data in relational databases or stream processing in relational data stream management systems.</i>
<i>Hypertext Transfer Protocol</i>	<i>HTTP</i>	<i>The set of rules for transferring files -- such as text, images, sound, video and other multimedia files -- over the web</i>
<i>Uniform Resource Locator</i>	<i>URL</i>	<i>A reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it</i>
<i>Database Management System</i>	<i>DBMS</i>	<i>A software package designed to store, retrieve, query and manage data</i>

:

:-

:

Appendix D: Referenced Documents

The System Development Document (SDD) for the Inventory Management System with a barcode scanner draws upon and references other relevant documents to ensure a comprehensive and cohesive approach to system development. Of particular significance is the utilization of "Budiga: An Inventory Management and Point of Sales System for MSME" as a foundational resource. This document establishes contextual background and relevant technical documentation regarding the system developed. The relationship between the SDD and these documents is integral to ensuring a comprehensive, cohesive, and well-informed approach to system development.




Table 14: Referenced Documents

Document Name	Document Location and/or URL	Issuance Date
<i>Budiga: An Inventory Management and Point of Sales System for MSME</i>	https://bit.ly/Budiga	05/28/2022
<i>Figma Budiga</i>	https://bit.ly/FigmaBudiga	05/30/2022
<i>Budiga Application</i>	https://github.com/therealmai/budiga_app	06/02/2022

Appendix E: Approvals

The undersigned acknowledge that they have reviewed the SDD and agree with the information presented within this document. Changes to this SDD will be coordinated with, and approved by, the undersigned, or their designated representatives.

Table 15: Approvals

Document Approved By	Date Approved
 Name: Rosales, Julyo Melchor, Business Owner - ajk3 Variety Store	Date 05/25/23
 Name: Rosales, Jade Andrie, Stakeholder - ajk3 Variety Store / Budiga Corp.	Date 05/25/23
 Name: Leano, Jomar, Project Manager - Budiga Corp.	Date 05/25/23

Appendix F: Additional Appendices

Not applicable