LiraFlow:
Where Every Item Finds Its Place

A Senior Project Submitted in Partial Fulfillment of the
Requirements for the Bachelor's Degree
in
Computer Science
Department of Computer Science

Faculty of Natural and Applied Sciences
Notre Dame University – Louaize
Zouk Mosbeh, Lebanon

Anthony Nasry Massaad

Anthony Lahoud

May 2025

Dedication

We dedicate this project to Dean Roger Nakad and Dr. Hoda Maalouf for introducing us to the opportunity to work on a real-world application. Your support was the first step in making this project a reality.

Acknowledgements

We would like to extend our deepest gratitude to the Faculty of Natural and Applied Sciences for their continued support throughout our journey.

A special thank you to Dean Roger Nakad and Dr. Hoda Maalouf for introducing us to the opportunity to develop *LiraFlow* in collaboration with a real-world company.

We are also sincerely grateful to Dr. Nazir Hawi for his unwavering support, guidance, and valuable insights at every stage. Your mentorship has played a crucial role in shaping this application and our learning experience.

Finally, we acknowledge Universal Services, the company that collaborated with us on this project for their trust and cooperation.

Abstract

This project presents the design and implementation of a cross-platform Warehouse Management System (WMS) developed to streamline inventory tracking, item management, and warehouse operations for Universal Services Inc., a fiber optics contracting company. The system is built using Flutter for the frontend, ensuring responsive interfaces for mobile, desktop, and web platforms. The backend is developed in Laravel PHP, with PostgreSQL databases hosted on Neon.tech, structured for authentication and business logic separation. Key features include user authentication via Laravel Sanctum, role-based access control (RBAC), inventory tracking across multiple warehouses, and item-level transaction recording for receiving, releasing, and adjusting. Advanced functionalities, such as AI-powered analytics and predictive insights for inventory control, are also integrated to enhance managerial decision-making. The system ensures high data security with hashed passwords, encrypted sensitive data, and detailed activity logs. Upon adding new items or warehouses, inventory entries are automatically initialized to maintain consistency. The WMS is scalable, maintainable, and built with modular components adhering to SOLID principles. This project contributes a modern, efficient, and secure solution to warehouse logistics challenges, especially for companies transitioning from manual or semi-digital inventory methods. Future expansions may include integration with supplier systems, barcode/RFID scanning, and IoT for real-time stock monitoring.

Keywords: Warehouse Management System, Laravel, Flutter, PostgreSQL, Inventory Tracking, Cross-platform, RBAC, AI Analytics, Universal Services Inc.

Table of Contents

List of Tables

List of Figures

List of Code Listings

Chapter 1: Business Vision, Glossary, and Risks

1.1 Introduction

In today's fast-paced world of logistics and supply chain operations, efficiency isn't just a bonus, it's a necessity. Whether it's a local distributor or a global manufacturer, managing inventory effectively can make or break a business. Yet, many warehouses still rely on outdated systems, manual tracking, and error-prone processes that slow operations and inflate costs.

To address this challenge, we developed and deployed a modern, cross-platform Warehouse Management System (WMS) tailored for our workplace. The system is not a prototype, it is live, actively used by warehouse staff and managers to handle essential operations such as receiving, releasing, adjusting, and tracking inventory in real time.

This project is more than an academic exercise; it's a solution born from real business needs. It brings digital transformation to an environment that needed it, simplifying daily workflows, minimizing human error, and improving overall efficiency.

In this chapter, we present the business vision that guided our development, define essential technical and warehouse-related terms in a glossary, and outline the risks and challenges encountered during the project lifecycle. Our goal is to showcase how a well-designed WMS can serve as a reliable, user-friendly, and impactful tool in the world of warehouse operations.

1.2 Definitions, Acronyms, and Abbreviations

1.2.1 RBAC

Role Base Access Control: A security model that restricts system access based on users' roles and responsibilities.

1.2.2 WMS

Warehouse Management System

1.2.3 User Permissions

Specific actions or features a user can access within the system, often controlled by roles.

1.2.4 Cross-platform

Software that works across multiple operating systems (e.g., Windows, Android, iOS).

1.2.5 Adjust In / Adjust Out

Manual corrections to increase or decrease inventory levels due to errors, damage, or internal movement.

1.3 Positioning

1.3.1 Business Opportunity

Warehouse operations are at the heart of logistics, and in an increasingly digital world, businesses cannot afford delays, inaccuracies, or inefficiencies. Many companies still use manual methods such as spreadsheets or even paper-based systems to manage inventory, which often leads to errors, stockouts, and poor visibility.

With the rise of digital transformation and the growing demand for accurate, real-time inventory tracking, the opportunity to modernize warehouse operations is greater than ever. Our WMS project is designed to bridge that gap, offering a cross-platform solution that helps warehouses improve efficiency, accuracy, and decision-making, while minimizing manual overhead.

1.3.2 Problem Statement

Traditional warehouse management often involves a mix of manual record-keeping and outdated software. This leads to miscommunication, inventory inaccuracies, and time wasted on routine tasks. Warehouse staff may spend hours locating items, adjusting stock levels manually, or reconciling errors due to a lack of real-time visibility.

Our project tackles this challenge by offering a streamlined, user-friendly WMS that digitizes and automates warehouse processes. The goal is to reduce human error, improve operational transparency, and empower warehouse teams to manage inventory efficiently from any device.

Table 1.1: Problem Statement

The problem of	Manual inventory tracking
affects	warehouse staff and managers
the impact of which is	Increased errors, inefficiency, and time wasted on basic and routine operations
a successful solution would be	Automate inventory operations and offer real-time updates

1.3.3 Product Position Statement

Our product is tailored to warehouse managers and field workers who need an efficient way to manage inventory and reduce manual errors. Our WMS is a cross-platform solution that provides real-time tracking, streamlined workflows, improved accuracy, and RBAC. Unlike generic systems or outdated manual methods, our WMS is user-friendly and specifically designed to meet the operational needs of warehouses in the logistics and construction industries.

Table 1.2: Product Positioning

For	Our work, as well as Warehouse managers, supervisors, and field workers at logistics and construction companies
Who	Need a reliable and efficient way to manage inventory, track items, and reduce manual errors
LiraFlow	is a Warehouse Management System
That	Enables real-time inventory tracking, smooth operations, and simplified warehouse workflows across all platforms
Unlike	Paper-based logs or systems that are overly complex
Our product	Offers a customized user-friendly tool with cross-platform compatibility.

1.4 Senior Study Objective

The primary objective of this senior project is to develop and implement a comprehensive WMS tailored specifically to meet the operational requirements of Universal Services. This system aims to significantly enhance inventory management accuracy, improve overall operational efficiency, and streamline warehouse procedures previously managed through manual methods such as Excel spreadsheets and paper-based tracking.

Ultimately, the project seeks to transition the company's warehouse operations from manual, error-prone methods to a fully digitalized, efficient, and secure management system.

1.5 Approach

To achieve the outlined objectives, the following structured approach was adopted throughout our senior project, following an incremental development model. This model allowed the system to be built, tested, and deployed in stages, with each release introducing new features or improvements based on user feedback.

1- Initial Analysis and Requirements Gathering
    a. Conducted interviews and weekly meetings with stakeholders to clarify operational needs, processes, pain points, and requirements.
    b. Analyzed existing inventory management methods, including Excel sheets and manual documentation, to identify gaps and weaknesses.
    c. Automated several existing Excel-based processes in VBA, which helped us understand current workflows and highlight limitations. This step guided the feature set for the custom WMS.
2- System Design
    a. Defined the system architecture and identified core modules.
    b. Designed relational database schemas for inventory tracking, transactions, user management, and role-based permissions.
    c. Developed UI/UX wireframes with a focus on intuitive design and accessibility for non-technical users.
3- Incremental Implementation
    a. Built the system in incremental phases, beginning with essential features such as releasing and adjustments.
    b. Frontend was developed using Flutter for cross-platform compatibility (mobile, web, desktop).
    c. Backend services were built using Laravel PHP with PostgreSQL databases for structured and secure data management.
    d. Each release included new functionality or corrections based on stakeholder testing and feedback.
4- Testing and Validation
    a. Performed thorough feature and unit tests.
    b. Deployed a beta release of the WMS to a limited group of users (couple of managers), to catch bugs and collect feedback before the final release.
    c. Adjustments and fixes were incorporated into subsequent iterations, improving system quality over time.

5- Deployment and Training
   a. Deployed the system incrementally in a live environment, starting with small-scale testing and expanding usage with each release.
   b. Provided training sessions and user guides for warehouse workers and managers to support adoption.

6- Evaluation and Future Recommendations
   a. Collected feedback continuously after each release to refine usability and performance.
   b. Final evaluations guided the roadmap for future development, including advanced analytics, supplier integration, and additional automation features.

1.6 Risks

1.6.1 Incorrect Inventory Data

- **Risk Magnitude or Ranking**

  High importance.

- **Description**

  Discrepancies between physical and recorded inventory due to bugs, data entry errors, or sync issues.

- **Impacts**

  Inaccurate stock levels can lead to unnecessary reordering of items that are already available and withholding the release of items due to the false assumption of insufficient stock. This results in increased costs, delays in order fulfillment, and reduced operational efficiency.

- **Indicators**

  Frequent manual inventory adjustments and user reports of incorrect item counts.

- **Mitigation Strategy:**
    - Implement validation checks for inventory updates.
    - Conduct thorough testing of inventory-related features before deployment.
- **Contingency Plan:**
    - Allow rollback or correction of erroneous entries.
    - Alert warehouse staff to verify physical stock and make adjustments as needed.
    - Document the issue for root cause analysis and apply necessary fixes or patches.

1.6.2 System Downtime

- **Risk Magnitude or Ranking**

  High importance.
- **Description**

  Unexpected system outages or crashes that disrupt WMS operations.
- **Impacts**

  Delayed order processing, inability to track inventory, reduced productivity, financial loss
- **Indicators**

  System logs showing errors or overloads, increased downtime reports, response delays in the system, and repeated server crashes.
- **Mitigation Strategy**
    - Use cloud-based solutions with high availability.
    - Optimize code and database queries to reduce performance issues.

    - Regular system maintenance.
- **Contingency Plan**
    - Notify warehouse managers and temporarily switch to manual operations.
    - Restore services from the most recent backup.

1.6.3 Unauthorized System or Feature Access

- **Risk Magnitude or Ranking**

  High importance.
- **Description**

  Unauthorized access to the system by outside users, or unauthorized access to admin features by application users.
- **Impacts**

  Breach of sensitive user or system information, unauthorized changes, and potential for malicious actions.
- **Indicators**

  Unusual activity logs, role mismatches, unauthorized access attempts to restricted endpoints.
- **Mitigation Strategy**
  - Enforce RBAC.
  - Use Laravel Sanctum for secure authentication.
  - Log all user actions for audit purposes.
- **Contingency Plan**
    - Immediately revoke compromised accounts.

  - Revert unauthorized changes from backups.
  - Notify affected users and conduct a security review.

1.6.4 Frontend Responsiveness Issues

- **Risk Magnitude or Ranking**

  Medium importance.

- **Description**

  Layout or display inconsistencies across different screen sizes due to incomplete or incorrect responsive design implementation.

- **Impacts**

  Poor user experience, difficulty navigating the system on certain devices, and reduced productivity for warehouse staff using mobile or tablet interfaces.

- **Indicators**

  User complaints about broken layouts, overlapping UI elements, unreadable text, or misaligned buttons when using the system on non-desktop devices.

- **Mitigation Strategy**
  - Conduct cross-device testing before deploying updates.
  - Structure the frontend using Flutter’s responsive layout system.
  - Separate feature screens based on screen size (mobile, tablet, desktop) to ensure optimized user experience across all devices.
- **Contingency Plan**
  - Roll back recent frontend changes if critical UI issues are detected.
  - Provide a temporary alternative while resolving the issue.

Chapter 2: Background

2.1 User Summary and Competition

2.1.1 User Summary

Table 2.1: User Summary

Name	**Description**	**Responsibilities**
Warehouse Worker	Personnel directly interacting with physical inventory operations.	• Receives and releases items • Conducts cycle counts • Reports discrepancies
Warehouse Manager	Oversees warehouse operations, ensuring accuracy and efficiency.	• Manages warehouse inventory • Monitors warehouse staff activity • Generates operational reports • Handles employees and warehouse management
Project Manager	Coordinates materials and resources for specific projects.	• Reviews inventory for projects • Requests material releases
IT Administrator	Maintains and supports the technical infrastructure of the WMS.	• Manages user permissions and security • Monitors system performance • Performs database backups and recovery operations • Troubleshoots technical issues

2.1.2 Alternatives and Competition

Initially, the stakeholders considered several options for their warehouse management needs:

- Commercial Off-The-Shelf (COTS) Solutions:
    - Strengths:
        - Quick implementation.
        - Established reliability and ongoing vendor support.
    - Weaknesses:
        - Limited flexibility and customization.
        - Features not fully aligned with specific operational processes of the business.
- Manual Tracking Methods (Excel/Paper-Based Systems):
    - Strengths:
        - Minimal upfront cost and easy initial setup.
        - Simple to operate in smaller-scale environments.
    - Weaknesses:
        - Inefficient and error-prone as volume and complexity increase.
        - Difficulties in data analysis, scalability, and real-time inventory tracking.
        - Increased risk of data loss, inconsistency, and poor data integrity.

Given the shortcomings of existing market solutions and manual methods, stakeholders concluded that developing a customized WMS is the optimal approach to effectively meet their specific operational requirements, scale with business growth, and improve overall efficiency.

2.2 Ethics

Several ethical considerations arise in the development of this WMS. These are listed in order of priority, beginning with preventing data loss, followed by transparency, and finally security:

- **Software Reliability and Correctness:** Ethical responsibility mandates the software must consistently perform accurately and reliably to avoid inventory losses, financial inaccuracies, or operational disruptions. Extensive testing, validation, and strict adherence to software engineering best practices, including SOLID principles, are employed to ensure reliability.
- **Transparency:** Maintaining transparent logs and clear audit trails for inventory adjustments, user actions, and transactions ensures accountability and reduces the potential for fraud or misuse.
- **Data Privacy and Security:** Sensitive data like user credentials, inventory levels, and financial transactions require robust protection. The WMS ensures privacy and security through password hashing, data encryption, and role-based access controls.

2.3 Impact on Society

The WMS significantly impacts individuals, organizations, and society both locally and globally:

- **Local Impact:**
  - Enhances operational efficiency at Universal Services Inc., reducing errors and improving resource allocation.
  - Improves job satisfaction for warehouse staff by simplifying tasks and reducing tedious manual labor.
- **Organizational Impact:**
  - Enables better inventory control and resource planning, resulting in cost savings and higher profitability.

  - Facilitates informed decision-making through real-time analytics and predictive insights, driving organizational growth.

- **Global Impact:**
  - Contributes to sustainability by optimizing warehouse operations, reducing waste, and improving resource management.
  - Offers a scalable model that can be adapted to other industries globally, enhancing overall logistics and supply chain efficiency.

2.4 Other Requirements

2.4.1 Applicable Standards

The development of the WMS adhered to common industry and platform standards to ensure compatibility, security, and performance across all devices. These include:

- **Platform Compliance**: Cross-platform compatibility was ensured by developing the frontend in Flutter, supporting Windows, macOS, Android, and iOS environments.
- **Hosting and Deployment Standards**: The backend is hosted on Heroku, which follows standard cloud deployment practices and provides HTTPS (SSL) for secure communication.
- **Database Management**: The PostgreSQL database is hosted on Neon.tech, which supports ACID compliance for data consistency and reliability.
- **Security Best Practices**: While not governed by formal legal standards, the system incorporates best practices for data security, including:
  - Password hashing and salting
  - Text fields sanitization
  - Role-based access control

2.4.2 System Requirements

The following system requirements support the application:

- **Operating Systems:** Windows, iOS, Android
- **Database:** PostgreSQL
- **Backend:** Laravel PHP
- **Frontend:** Flutter

2.4.3 Performance Requirements

Key performance requirements include:

- Support for concurrent access by at least 5 simultaneous users.
- Response times for transactions under normal load conditions below 2 seconds.
- Database backup and recovery everyday.
- High reliability.

2.4.4 Environmental Requirements

The WMS environmental requirements:

- Capable of handling usage in warehouse environments.
- Robust error handling and recovery processes to maintain business continuity.
- Regular scheduled maintenance without disrupting core business activities

.

Chapter 3: Requirements Elicitation

3.1 Adopted Technique(s)

The adopted techniques for requirements elicitation included interviews with stakeholders, direct observations of current inventory management practices, and analysis of existing spreadsheets currently in use. Additionally, prior to the WMS design and implementation, efforts were made to automate existing processes within the Excel spreadsheets currently used, enabling an in-depth analysis of weaknesses and areas requiring improvement.

3.2 Questions

1. What are the exact processes involved in receiving, storing, and releasing inventory items?
2. What are the critical weaknesses encountered with your existing tracking system?
3. What frequency of inventory updates and cycle counts is currently maintained?
4. Could you list the most critical inventory-related reports you would need to produce?
5. What type of analytics or forecasting capabilities would significantly enhance your decision-making?
6. How many distinct warehouse locations are currently managed, and what are the plans for future expansion?
7. Can you outline your expectations regarding system availability?
8. What are the user roles in the system, what permissions should they have, and how is the role hierarchy defined?
9. What devices or platforms will the system need to support?
10. What is the anticipated volume of transactions the system should support daily or monthly?

11. Are there unique items or categories of inventory that require special handling or tracking rules?
12. What budgetary constraints or financial considerations should be taken into account for development, hosting and maintenance?
13. How important is mobile or remote access capability for your warehouse operations team?
14. What type of backup, recovery, and disaster recovery planning is desired?
15. Can you describe any existing integration requirements with external systems, like supplier databases?
16. How would you measure or define the success of this new Warehouse Management System?

3.3 Stakeholders' General Statement

Stakeholders expressed the necessity for a tailored WMS capable of accurately and efficiently managing inventory, scaling smoothly with company growth, and ensuring reliable performance. They emphasized the importance of usability, intuitive design, and an easy-to-use system since workers are skeptical of new technologies. Additionally, stakeholders highlighted the critical need for comprehensive real-time reporting and analytics to support effective decision-making and inventory tracking.

Chapter 4: Enumerated Requirements and Increments

4.1 Functional Requirements

Table 4.1: List of Functional Requirements

ID	Priority	Description
REQ1	1	The system shall allow warehouse staff to receive items and record their quantities by warehouse and item.
REQ2	1	The system shall allow users to release items from a specific warehouse and update the inventory accordingly.
REQ3	1	The system shall allow users to make quantity adjustments for items from a specific warehouse and update the inventory accordingly.
ST-4	1	As a manager, I want to track inventory and see the transactions' log.
ST-5	1	As an admin, I want to create and manage users and assign roles to control access to different system features.
ST-6	2	As a manager, I want access to an analytics dashboard with visual summaries of stock movements, item usage, and financial value trends, so I can make data-driven decisions.
REQ-7	1	The system shall, when creating a new warehouse, automatically add all items to inventory.
REQ-8	1	The system shall, when creating a new item, automatically add one entry for each warehouse in the inventory.
REQ-9	1	The system shall allow authorized managers to create, update, and soft delete warehouse records.
REQ-10	1	The system shall allow authorized managers to create, update, and soft delete item records.
REQ-11	1	The system shall allow authorized managers to create, update, and soft delete employee records.
REQ-12	1	The system shall allow authorized managers to create, update, and soft delete Crew records.

REQ-13	3	The system shall allow users to filter and search data across all major modules using relevant fields to enable quick access to information.
ST-14	3	As a Manager, I want to download transaction reports in Excel format for auditing and offline analysis.
REQ-15	1	The system shall prevent releasing or adjusting quantities that exceed available stock and shall show a warning.
REQ-16	2	The system shall allow managers to import employee data via an Excel file.
REQ-17	4	Add offline mode with local caching and auto-sync when connection is restored.
REQ-18	4	Add more features exports and imports to excel/pdf.

4.2 Non-Functional Requirements

Table 4.2: List of Non-Functional Requirements

ID	Priority	Description
REQ-19	2	The system shall respond to user actions (e.g., search, filter, submit) within 2 seconds under normal network conditions.
REQ-20	2	The system shall support responsive layouts for mobile, tablet, and desktop using Flutter’s adaptive UI components.
REQ-21	1	The system shall be usable by non-technical users, featuring a clean UI, consistent design, and field validations to prevent errors.
REQ-22	1	The system shall validate all user inputs for forms and API requests, enforcing required fields, correct data types, and business rules before saving data.
REQ-23	2	The system shall return consistent JSON responses for all API endpoints, including standardized success and error formats, status codes, and message structures.
REQ-24	2	The system shall restrict access to sensitive operations based on role-based access control (RBAC).
REQ-25	2	The system shall log all critical operations (e.g., inventory changes,

		imports, user creation) for auditability.
REQ-26	1	The system shall back up all critical data (users, inventory, transactions) at least once per day automatically.
REQ-27	4	Add Admin Dashboard with user activity logs for auditing and security tracking.
REQ-28	4	Add facial recognition or biometric login for warehouse staff.

4.4 Increments

To ensure a smooth and structured development process, the functional requirements were divided into multiple release backlogs and sprint cycles. The first release includes all core functionalities required for the system's initial deployment. The second backlog focuses on improvements and extended capabilities based on user needs. The final backlog outlines features planned for future implementation.

4.4.1 Release Backlogs

Table 4.3.1: Core Features Release Backlog

Requirement IDs					
REQ-1	REQ-2	REQ-3	ST-4	ST-5	REQ-7
REQ-8	REQ-9	REQ-10	REQ-11	REQ-12	REQ-15
REQ-21	REQ-22	REQ-26			

These core requirements allow users to manage key features of the application, including items, warehouses, employees, and crews. They also enable inventory operations such as item receiving, releasing, and quantity adjustments. Additionally, users can view transaction logs for traceability, while the system provides a user-friendly interface designed for non-technical staff. Plus, to ensure data safety, the system performs automatic daily database backups.

Table 4.4.2: Improvement Features Release Backlog

Requirement IDs					
ST-6	REQ-13	ST-14	REQ-16	REQ-19	REQ-20
REQ-23	REQ-24	REQ-25			

Table 4.3.3: Future Release Backlog

Requirement IDs					
REQ-17	REQ-18	REQ-27	REQ-28		

4.4.2 Sprints

Table 4.5: Core Features Release Backlog Sprints

Requirement IDs					
Sprint 1		Sprint 2		Sprint 3	
REQ-7	REQ-8	REQ1	REQ-2	REQ-21	REQ-22
REQ-9	REQ-10	REQ-3	ST-4	REQ-26	
REQ-11	REQ-12	ST-5	REQ-15		

- Sprint 1: Focused on the foundational setup of the system, including the management of items, warehouses, employees, and crews. This sprint established the core data structures and CRUD operations required for system functionality.
- Sprint 2: Implemented core inventory operations such as releasing items, adjusting quantities, and tracking these transactions in the system log.
- Sprint 3: Focused on system usability and reliability, including a user-friendly interface for non-technical users, field and API-level validation, and automatic daily database backups to ensure data integrity.

Chapter 5: Functional Requirements Specification

5.1 Use Case Diagram

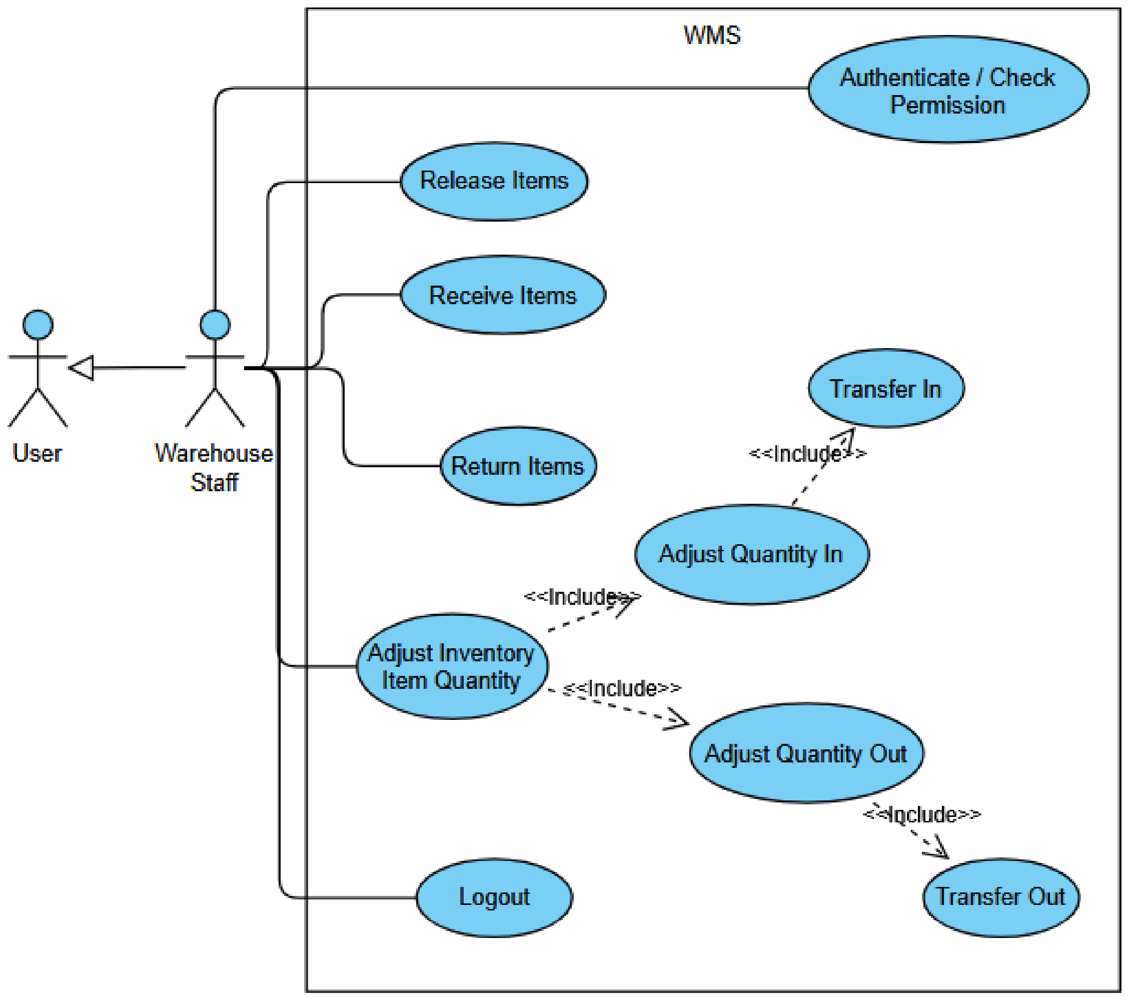


Figure 5.1: WMS, Warehouse Staff Operations Use Case Diagram

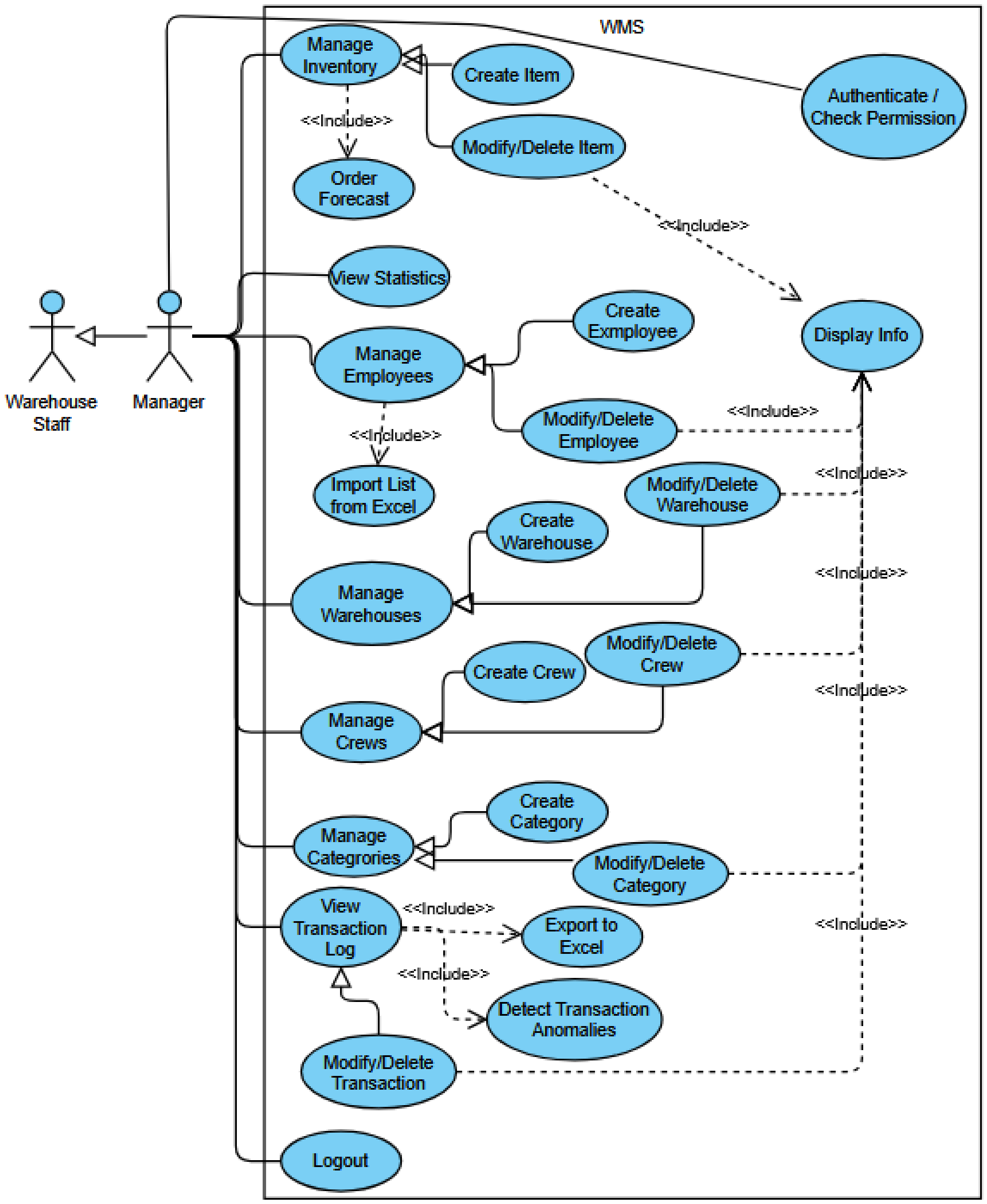


Figure 5.2: WMS, Warehouse Manager Operations Use Case Diagram

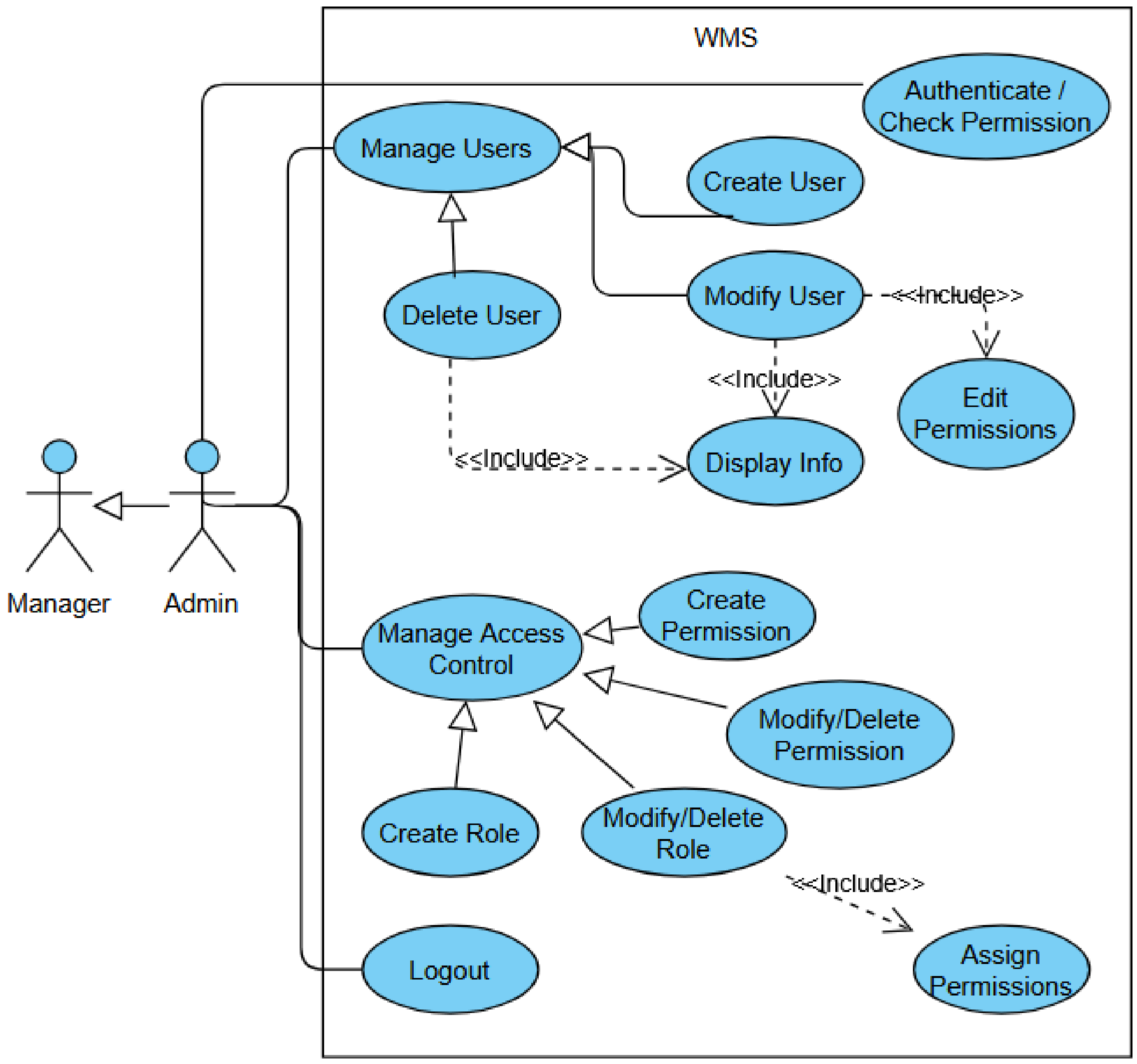


Figure 5.3: WMS, Admin Operations Use Case Diagram

5.2 Use Case Specification for Warehouse Staff Operations

5.2.1 Brief Description

The Warehouse Staff operations use case involves the warehouse staff performing simple daily inventory management tasks, such as releasing items, receiving new items, handling returns, and adjusting inventory quantities. The operations include authentication checks, transferring items between warehouse locations, and logging out of the system after completing tasks.

5.2.2 Actors

- Warehouse Staff

5.2.3 Basic Flow of Events

The use case begins when the warehouse staff logs into the WMS.

1- The system authenticates the warehouse staff credentials and checks permissions.

2- Once authenticated, the warehouse staff selects one of the following actions:

a. Release

Warehouse staff selects the release details, including foreman, crew, supervisor, the source warehouse, and the employee receiving the items. The system displays available items from the selected warehouse. The staff chooses the items to release, proceeds to the next screen to confirm their selection, and specifies the quantities for each item. A second confirmation prevents accidental releases. The system updates inventory accordingly. After the release, the staff is redirected to the initial release screen, retaining the previously selected crew, foreman, and warehouse, but clearing the employee selection to streamline the release process for another employee.

b. Adjustments

Warehouse staff selects a warehouse, and the system retrieves items available in that warehouse. The staff selects an item to adjust and chooses the adjustment type (quantity in or out). They then specify a reason from the available options: received, returned, transferred in, adjusted, or transferred out. After entering the quantity, the staff confirms the adjustment, and the system updates the inventory accordingly. The staff is redirected back to the initial warehouse selection screen.

3- After completing tasks, warehouse staff selects the logout option.

4- The system logs out the warehouse staff and returns to the login screen.

5.2.4 Alternative Flows of Events

< Authentication Failure >

If authentication fails, the system notifies the warehouse staff, requesting them to retry.

< Quantity Exceeds Available Inventory >

If the warehouse staff attempts to release or adjust out more than the available quantity, the system blocks the operation and prompts the staff to retry with correct quantities.

< Restricted Permissions Visibility >

If a warehouse staff user lacks permissions for release or adjustment operations, these options will not be visible in the sidebar, preventing unauthorized access attempts.

5.2.5 Special Requirements

< Performance Requirement >

The system must respond within 2 seconds for all inventory-update operations.

< Usability Requirement >

The user interface must be intuitive and require as few clicks as possible, to address staff skepticism toward new technology.

< Audit Logging Requirements >

Every transaction (release or adjustment) must be recorded with full details, including date/time, affected items, quantities, and user identity, to support tracking and audit.

5.2.6 Preconditions

< Authentication Precondition >

Warehouse staff must be authenticated with valid credentials.

< Permission Precondition >

The user's account must include the appropriate permissions for release and adjustment operations.

< Database Availability Precondition >

The inventory database must be online and accessible.

5.2.7 Postconditions

< Inventory Updated Postcondition >/time, item information, quantities, user identity)

Inventory levels accurately reflect all transactions performed.

< Audit Log Postcondition >

Each transaction is recorded in the audit trail with full details (date, quantity, etc.)

5.2.8 Extension Points

<Validate User Permissions >

After authentication, before presenting action options.

< Retrieve Warehouse Inventory >

After warehouse selection, before displaying item list.

< Verify Quantity Availability >

After the user enters quantities, before the confirmation dialog.

5.3 Use Case Specification for Manager Operations

5.3.1 Brief Description

Managers oversee inventory, manage employees, crews, warehouses, and item categories. They also view statistics, handle transactions, export data, and detect transaction anomalies.

5.3.2 Actors

- Warehouse Manager

5.3.3 Basic Flow of Events

The use case begins when the warehouse manager logs into the WMS.

1- The system authenticates the warehouse staff credentials and checks permissions.

2- After authentication, the warehouse manager selects one of the following actions:

a. Home Dashboard

View today's insights, such as key inventory metrics and recent activity summaries.

b. Inventory

View current inventory status, create new items, update existing item details, or delete items from inventory. Additionally, the manager can access an AI forecasting tool that predicts the required items and quantities to order for a specified future period.

c. Release

Same as the warehouse staff flow; refer to section 5.3.3.

d. Adjustments

Same as the warehouse staff flow; refer to section 5.3.3.

e. Transactions

View the transaction log and identify anomalies flagged by the AI system.

f. Reports

Access analytics and reports on warehouse performance and inventory data.

g. Settings

i. Manage Warehouses

- Create Category
- Modify/Delete Category

ii. Manage Employees

- Import Employee List from Excel: Manager uploads an Excel file with employee details, which updates employee records in the system.
- Create Employee.
- Modify/Delete Employee.

iii. Manage Crews

- Create Crew.
- Modify/Delete Crew.

iv. Manage Categories

- Create Category
- Modify/Delete Category

3- After completing tasks, the warehouse manager selects the logout option.

4- The system logs out the warehouse manager and returns to the login screen.

5.3.4 Alternative Flows of Events

< Authentication Failure >

If authentication fails, the system notifies the warehouse staff, requesting them to retry.

< Insufficient Permissions >

If the manager lacks permission for certain tasks, those tasks do not appear in the system menu.

< Employee Excel Import Error >

If the Excel file upload fails, the system informs the manager to correct the file and retry.

< Forecasting Error >

If the forecasting calculation fails, the system prompts the manager to retry or contact support.

< Transaction Log Export Error >

If exporting transaction logs fails, the system notifies the manager to retry or use another method.

5.3.5 Special Requirements

< Performance Requirement >

The system must respond within 2 seconds for all operations.

< Usability Requirement >

The system interface should be simple, intuitive, and require minimal clicks for navigation.

< Decision Support Requirements >

System should highlight critical issues (e.g., low stock, unusual transactions) without requiring much manual search.

5.3.6 Preconditions

< Authentication Required >

Warehouse manager must be authenticated with valid credentials.

< Permission Precondition >

The user's account must include the appropriate permissions.

< System Integrity >

Database, AI tools, and services must be running and available.

5.3.7 Postconditions

< Data Updated >

Any changes to items, users, crews, categories, employees or warehouses must be saved and immediately visible to all authorized users.

5.3.8 Extension Points

No Extension Points

5.4 Use Case Specification for Admin Operations

5.4.1 Brief Description

The admin manages users and access control by creating, updating, or removing user accounts, roles, and permissions. The admin ensures that only authorized users can access specific features.

5.4.2 Actors

- Admin

5.4.3 Basic Flow of Events

The use case begins when the admin logs into the WMS.

5- The system authenticates the admin credentials and verifies access rights.

6- After login, the admin selects one of the following actions:

a. Home Dashboard

Same as the warehouse manager’s flow; refer to section 5.4.3.

b. Inventory

Same as the warehouse manager’s flow; refer to section 5.4.3.

c. Release

Same as the warehouse staff flow; refer to section 5.3.3.

d. Adjustments

Same as the warehouse staff flow; refer to section 5.3.3.

e. Transactions

Same as the warehouse manager’s flow; refer to section 5.4.3.

f. Reports

Same as the warehouse manager’s flow; refer to section 5.4.3.

g. Settings

Same as the warehouse manager's flow; refer to section 5.4.3. The admin has the below additional features:

i. Manage Users:

- Create/Modify/Delete User
- Display Info
- Edit Permissions: Admin may adjust a specific user’s permissions

ii. Manage Access Control

- Create/Modify/Delete roles and assign permissions to it.
- Create/Modify/Delete permissions.

7- After completing tasks, the admin selects logs out.

8- The system logs out the warehouse manager and returns to the login screen.

5.4.4 Alternative Flows of Events

< Authentication Failure >

If authentication fails, the system notifies the admin, requesting them to retry.

5.4.5 Special Requirements

< Access Control Requirement >

Only users with admin privileges can access user and role management features.

< Audit Requirement >

All admin changes to users, roles, and permissions must be logged with timestamps and admin identity.

5.4.6 Preconditions

< Authentication Required >

Warehouse manager must be authenticated with valid credentials.

< System Integrity >

System must be online and responsive.

5.4.7 Postconditions

< Data Updated >

All user, role, and permission changes are saved in the system.

5.4.8 Extension Points

No Extension Points.

Chapter 6: Domain Analysis

6.1 Domain Model Class Diagram

This diagram illustrates how the Warehouse Management System handles transactions, inventory management, and user roles. It shows how users perform operations based on their assigned roles and permissions, such as releasing or adjusting items. For traceability, each transaction is linked to a specific item, warehouse, and, optionally, to an employee, a foreman, and a crew. The diagram also reflects supporting features like employee record imports and transaction exports, capturing both operational flow and administrative functions. Analytics-related methods are included to highlight how the system tracks and summarizes activity for reporting and decision-making.

➢ **User, Role, and Permissions:**

- User is the class for system users.
- Roles can be changed dynamically, and new roles can be created dynamically too. The main roles that are currently in the system are:
  - **Admin:** Has access to everything, including user and Access Control management.
  - **Manager:** Has access to everything, except user and Access Control management.
  - **Worker (Warehouse Staff):** Has access to the release screen only.
- Permissions define what users can and cannot do. They are assigned to roles, and users who have a role automatically get all the associated permissions. On top of that, a user can have custom/extra permissions, for example, a user who has a Worked role can get access to the adjustments screen by having the custom permissions of the adjustments.

- **InventoryItem, Category and Warehouse:**

- Warehouse stores the warehouse name and location where items are stored.
- “InventoryItem” represents the item and the inventory of that item (quantity), which is present in a specific warehouse.
- Items exist in categories, each group of items shares a similar category (ex: Gloves, Helmets)

- **Transactions:**

- Logs of past transactions made by users.
- It has three types: Released, Adjusted in, and Adjusted out.
- Stores the date, quantity, total Price at the time the transaction was made, who the transaction was sent to (in case of a released transaction), etc.

- **Crew and Employee:**

- Employee represents the person to whom an item can be sent.
- Crew represents where that employee was working at the time the items were sent out to him/her.

- **SearchFilter:**

- A dynamic search that searches across multiple columns in a database table and across multiple columns from related tables.

- **Metrics, Alerts, and Charts:**

- Alerts are used to notify Admins and managers about low stock levels when items go below their threshold.
- Metrics are short insights about today's system usage (Number of Received items, number of releases, etc.)
- Charts are custom graphs that show System activities over a specified date range.

- **DemandForecast:**

- Analyzes historical inventory and transaction data, then forecasts future demand for each warehouse, and recommends optimal restocking quantities based on those trends.

- **AnomalyDetector:**

- Utilizes an Isolation Forest algorithm to scan historical transaction and inventory data, automatically flagging outliers, such as unusually large releases, duplicate entries, or unexpected stock adjustments, for review and corrective action.

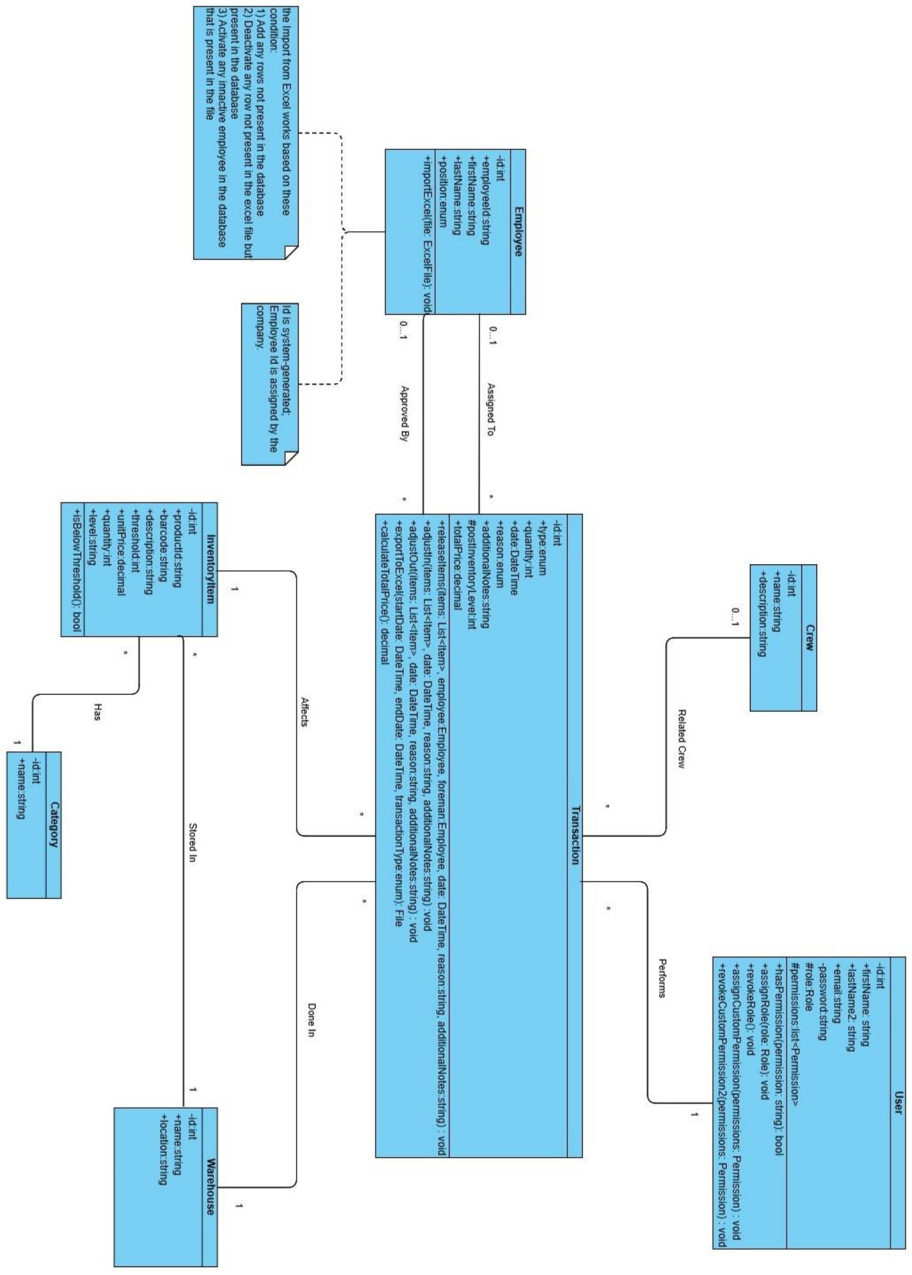

User
-id:int
+firstName: string
+lastName2: string
+email:string
-password:string
#role:Role
#permissions:list<Permission>
+hasPermission(permission: string): bool
+assignRole(role: Role): void
+revokeRole(): void
+assignCustomPermission(permissions: Permission) : void
+revokeCustomPermission2(permissions: Permission) : void
Crew
-id:int
+name:string
+description:string
0...1
Related Crew
*
Performs
1
*
Transaction
-id:int
+type:enum
+quantity:int
+date:DateTime
+reason:enum
+additionalNotes:string
#postInventoryLevel:int
+totalPrice:decimal
+releaseItems(items: List<Item>, employee:Employee, foreman:Employee, date: DateTime, reason:string, additionalNotes:string) : void
+adjustIn(items: List<Item>, date: DateTime, reason:string, additionalNotes:string) :void
+adjustOut(items: List<Item>, date: DateTime, reason:string, additionalNotes:string) : void
+exportToExcel(startDate: DateTime, endDate: DateTime, transactionType:enum): File
+calculateTotalPrice(): decimal
Employee
-id:int
+employeeId:string
+firstName:string
+lastName:string
+position:enum
+importExcel(file: ExcelFile): void
0...1
Assigned To
*
0...1
Approved By
*
the Import from Excel works based on these condition:
1) Add any rows not present in the database
2) Deactivate any row not present in the excel file but present in the database
3) Activate any innactive employee in the database that is present in the file
Id is system-generated;
Employee Id is assigned by the company.
*
Affects
1
*
Done In
1
InventoryItem
-id:int
+productId:string
+barcode:string
+description:string
+threshold:int
+unitPrice:decimal
+quantity:int
+level:string
+isBelowThreshold(): bool
*
Stored In
1
Warehouse
-id:int
+name:string
+location:string
*
Has
1
Category
-id:int
+name:string

Figure 6.1: Domain Model Representing Transaction Flow in the WMS

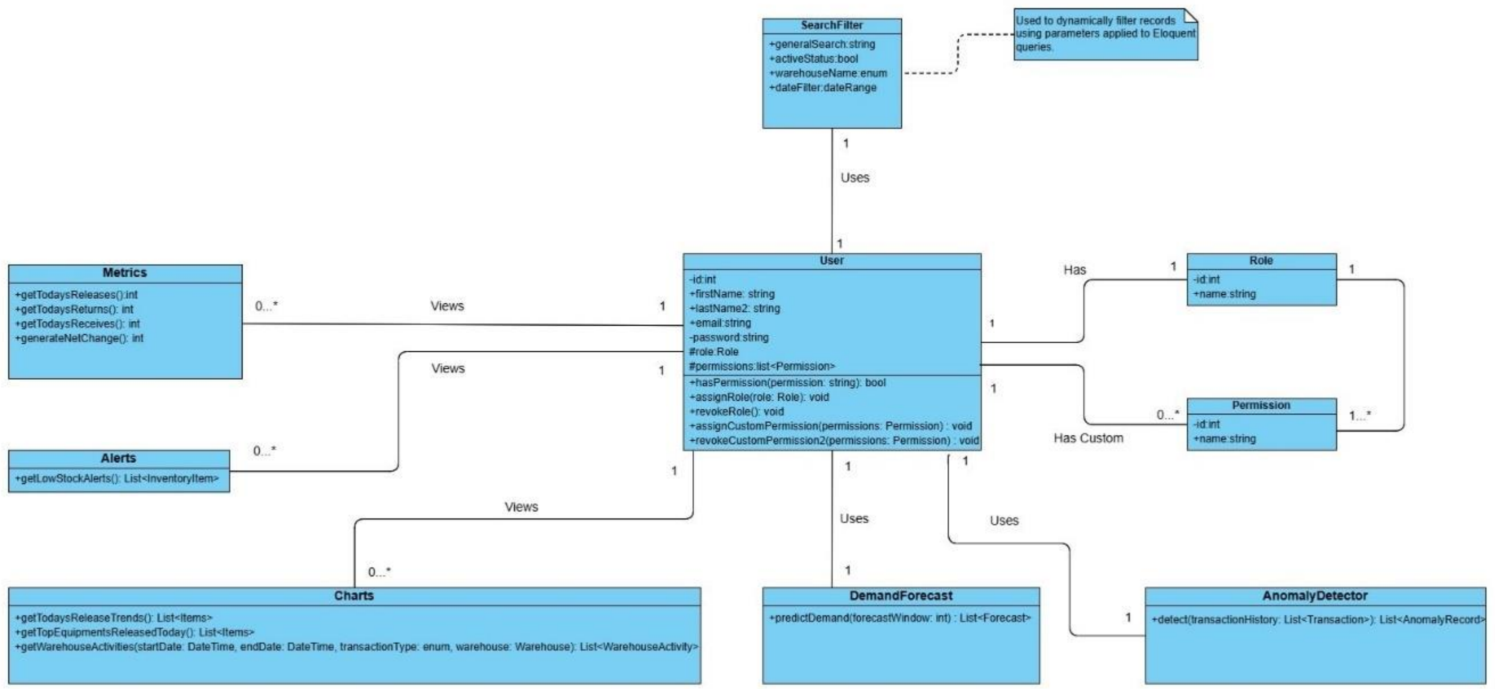

SearchFilter
+generalSearch:string
+activeStatus:bool
+warehouseName:enum
+dateFilter:dateRange
Used to dynamically filter records using parameters applied to Eloquent queries.
Uses
User
-id:int
+firstName: string
+lastName2: string
+email:string
-password:string
#role:Role
#permissions:list<Permission>
+hasPermission(permission: string): bool
+assignRole(role: Role): void
+revokeRole(): void
+assignCustomPermission(permissions: Permission) : void
+revokeCustomPermission2(permissions: Permission) : void
Metrics
+getTodaysReleases():int
+getTodaysReturns(): int
+getTodaysReceives(): int
+generateNetChange(): int
Views
Alerts
+getLowStockAlerts(): List<InventoryItem>
Views
Charts
+getTodaysReleaseTrends(): List<Items>
+getTopEquipmentsReleasedToday(): List<Items>
+getWarehouseActivities(startDate: DateTime, endDate: DateTime, transactionType: enum, warehouse: Warehouse): List<WarehouseActivity>
Views
DemandForecast
+predictDemand(forecastWindow: int) : List<Forecast>
Uses
AnomalyDetector
+detect(transactionHistory: List<Transaction>): List<AnomalyRecord>
Uses
Role
-id:int
+name:string
Has
Permission
-id:int
+name:string
Has Custom
0...*
1...*
1

Figure 6.2: Domain Model Representing RBAC, AI, and Analytics in the WMS

6.2 Traceability Matrix

UC1: Login

UC2: Receive Items

UC3: Release Items

UC4: Return Items

UC5: Adjust Quantity

UC6: Transfer Items

UC7: Manage Items

UC8: Manage Warehouses

UC9: Manage Employees

UC10: Manage Crews

UC11: View Transactions

UC12: Export Reports

UC13: Forecast Demand

UC14: Detect Anomalies

UC15: Manage Users

UC16: Manage Roles & Permissions

Table 6.1: Domain Concept / Use Case Traceability Matrix

Domain Concept / Use Case	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
Admin	✓	✓	✓	✓	✓	✓	✓	✓
Manager	✓	✓	✓	✓	✓	✓	✓	✓
Warehouse Staff	✓		✓					

Domain Concept / Use Case	UC9	UC10	UC11	UC12	UC13	UC14	UC15	UC16
Admin	✓	✓	✓	✓	✓	✓	✓	✓
Manager	✓	✓	✓	✓	✓	✓		
Warehouse Staff								

Chapter 7: Interaction and Data Flow Diagrams

7.1 Sequence Diagrams

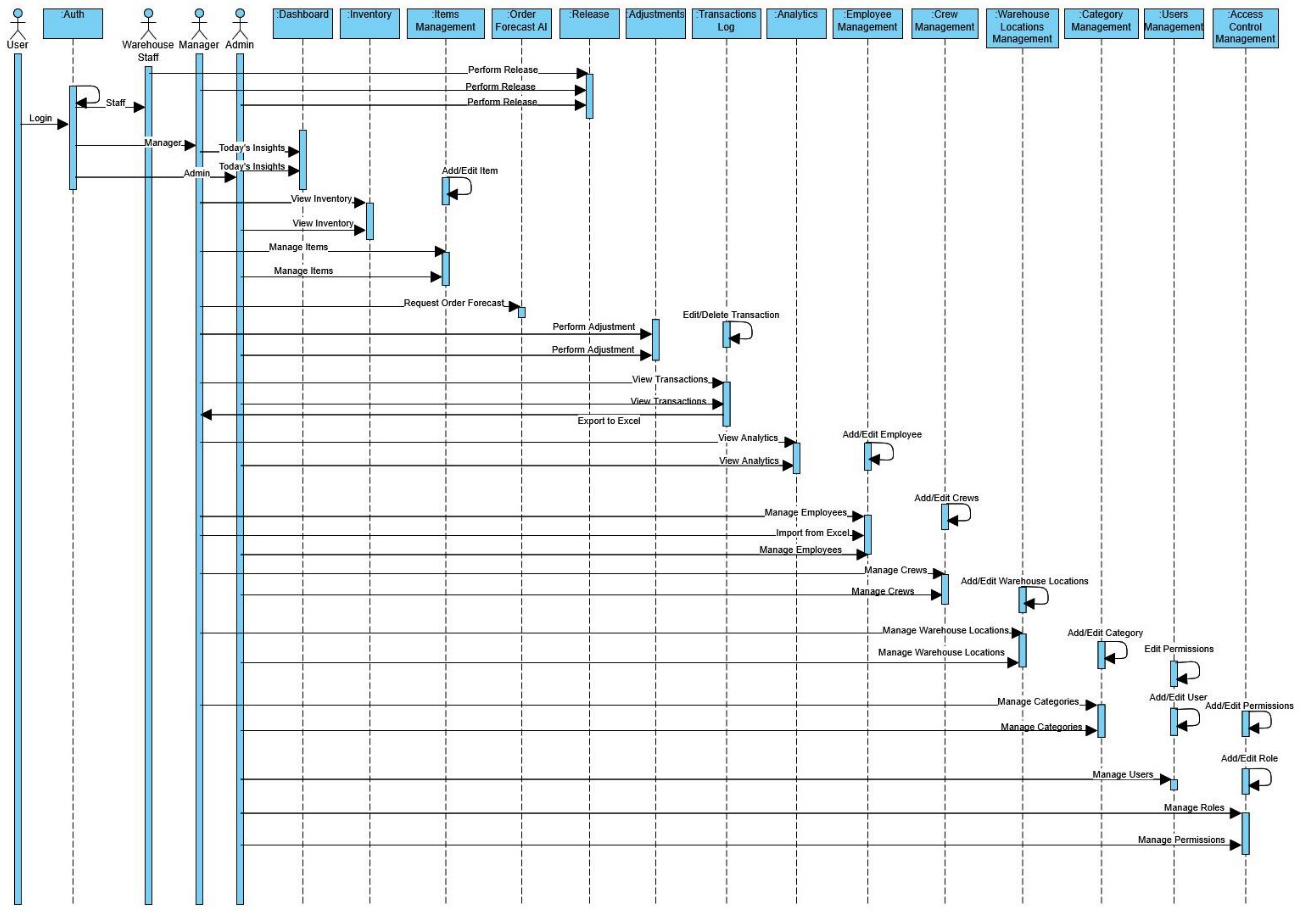


Figure 7.1: LiraFlow Sequential Diagram

The sequential diagram above captures the flow of actions such as login, inventory management, item release and adjustment, transaction tracking, analytics viewing, and administrative tasks like user and permission management. Each request follows a clear sequence from the user to the relevant module and back with a response.

7.2 Data Flow Diagram

7.2.1 Context Diagram (Level 0)

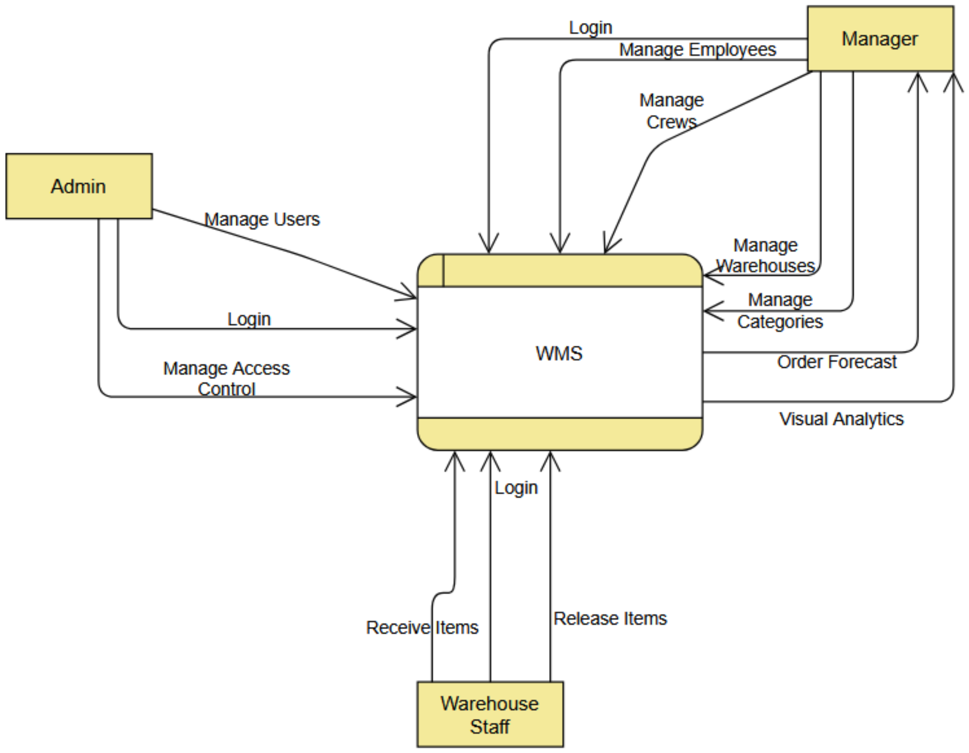


Figure 7.2: LiraFlow DFD Level 0

This diagram shows the main actors interacting with the WMS. It includes Admin, Manager, and Warehouse Staff. Each actor performs key tasks such as logging in, managing data, or handling items. The system is treated as one central process.

7.2.2 Level 1

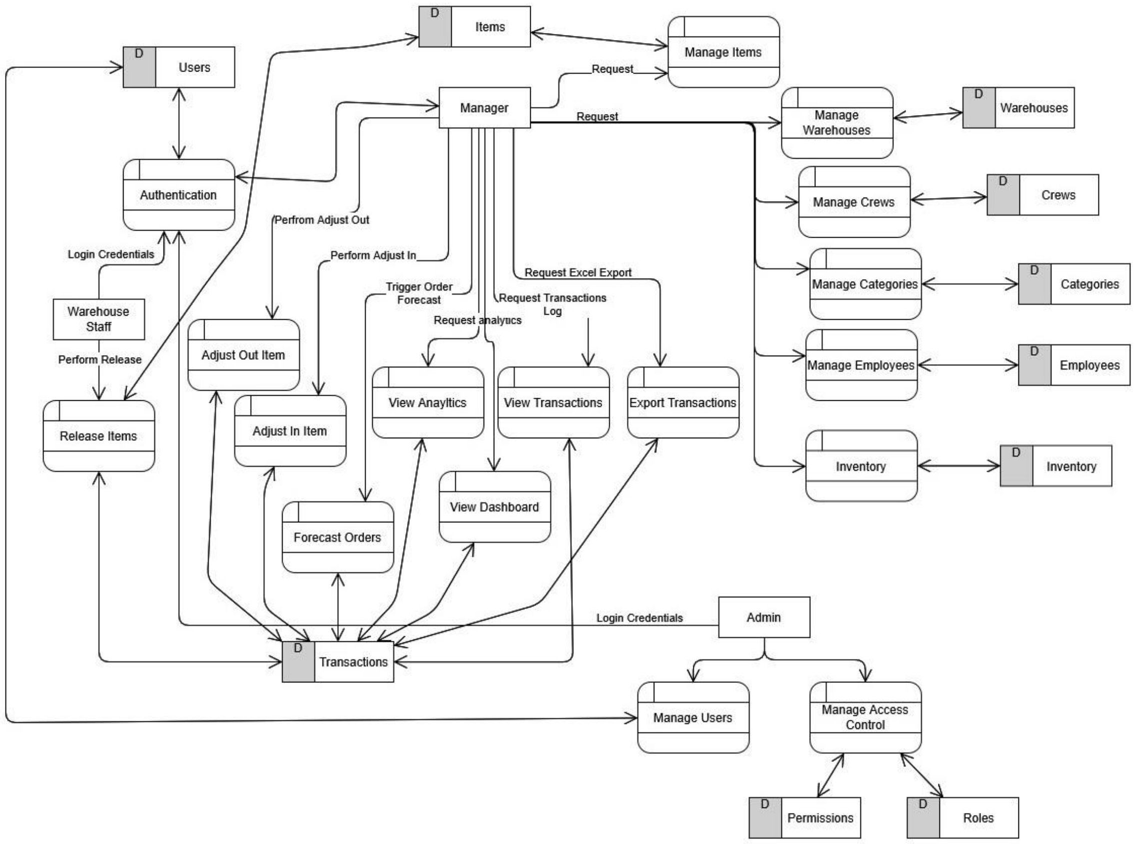


Figure 7.3: LiraFlow DFD Level 1

The Level 1 DFD expands the central WMS process into detailed subprocesses such as authentication, user and access control, inventory handling, and analytics. It introduces internal data stores like Users, Inventory, Transactions, and shows how data flows through specific modules handled by different system roles.

Chapter 8: Software Architecture and Design

8.1 Physical Tiers

The system is organized into a multi-tier architecture that separates client applications, backend logic, and data services. The client tier consists of Flutter-based mobile and desktop applications used by warehouse personnel and managers. These clients interact with the backend tier, which is built using the Laravel PHP framework and hosted on Heroku. The backend exposes a RESTful API for operations such as user authentication, inventory transactions, and permission management.

Laravel also functions as the orchestrator of the system, coordinating requests between the frontend and two specialized microservices hosted as independent FastAPI applications: one for demand forecasting and another for anomaly detection. These services are accessed internally via API calls, enabling the system to offer AI-powered analytics without burdening the main backend.

The data tier is managed by a PostgreSQL database hosted on Neon Tech, which stores structured data related to users, transactions, inventory, permissions, and analytics.

Communication between all components occurs over secure HTTPS connections, following a modular design that supports scalability, separation of concerns, and maintainability.

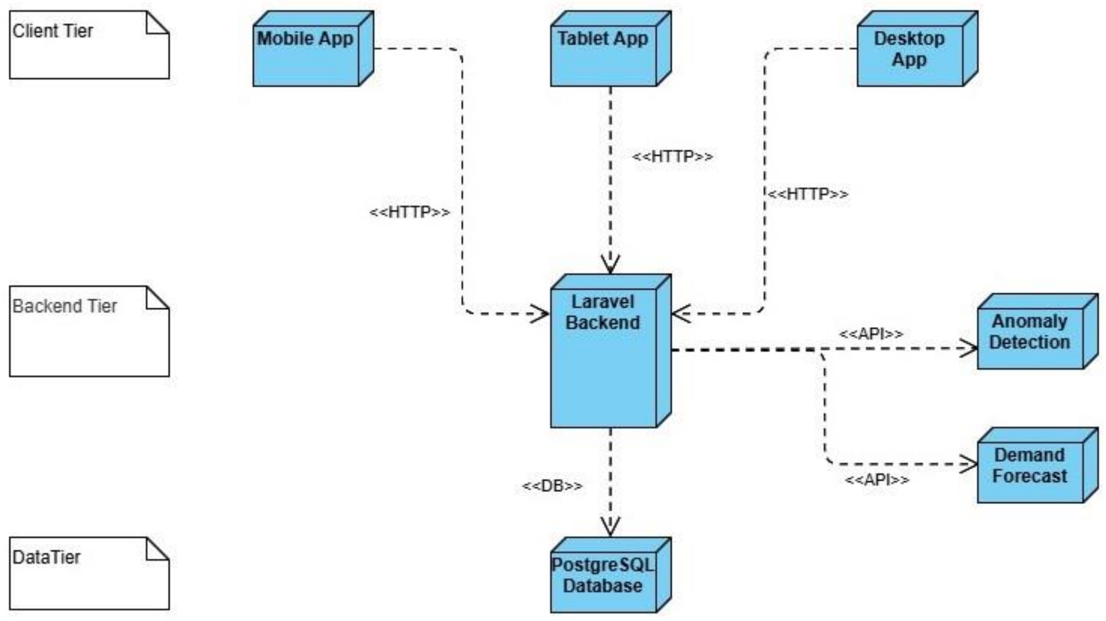


Figure 8.1: Physical Tier Diagram

8.2 Logical Layers

The system follows a layered architectural model that separates concerns across four primary logical layers. This approach enhances maintainability, scalability, and clarity throughout the codebase.

8.2.1 Presentation Layer (Client Interface)

This layer is implemented in Flutter for mobile, tablet, and desktop platforms. It is responsible for:

- Rendering user interfaces
- Handling user input and navigation
- Making secure HTTP API calls to the backend
- Managing state and routing screens

The Flutter frontend follows a feature-based folder structure, allowing for modular development where each feature (e.g., inventory, release, etc.) encapsulates its own views and controllers. This structure supports clean separation between UI and logic, improves readability, and scales well as features grow.

8.2.2 Application Layer (Request Coordination)

This layer exists primarily in Laravel, handling:

- Routing requests to the appropriate controllers
- API authentication and authorization using Laravel Sanctum and Spatie Permissions
- Excel export/import operations using Laravel Maatwebsite Excel, which are structured into dedicated service and Excel handler classes
- Data validation using Laravel FormRequest classes
- Response transformation using Laravel Http\Resources to standardize API responses across modules
- Permission checks, which enforce system integrity

To ensure modularity and flexibility, the service layer uses Dependency Injection, a key Object-Oriented Design (OOD) principle. This approach promotes loose coupling between controllers and services, facilitates unit testing, and aligns with the Dependency Inversion Principle of the SOLID design methodology. Reusable logic across controllers and services is abstracted into Traits, promoting DRY principles.

8.2.3 Domain Layer (Business Logic)

This layer contains the core logic of the system. It includes:

- Business rules like transaction validation, role-specific access restrictions, and inventory consistency
- Core services for features such as releasing items, adjusting stock, and generating reports
- Model relationships

- Custom exception handling for enforcing strict business rules

This logic is centralized in dedicated service classes within Laravel, ensuring clean separation from controller logic and reusability across endpoints. These service classes are structured with Object-Oriented Design principles in mind, promoting encapsulation, single responsibility, and ease of reuse across multiple endpoints.

8.2.4 Data Access Layer

The data access layer is responsible for:

- Communicating with the PostgreSQL database using Eloquent ORM
- Executing queries for entities such as users, transactions, inventory, and warehouses
- Handling connections to AI microservices (Forecasting and Anomaly Detection) using HTTP clients within Laravel service classes

The database is hosted on Neon Tech.

This structured layer model supports maintainable code, easier debugging, and a scalable foundation for future enhancements such as extended reporting or additional microservices.

8.3 Package Diagram

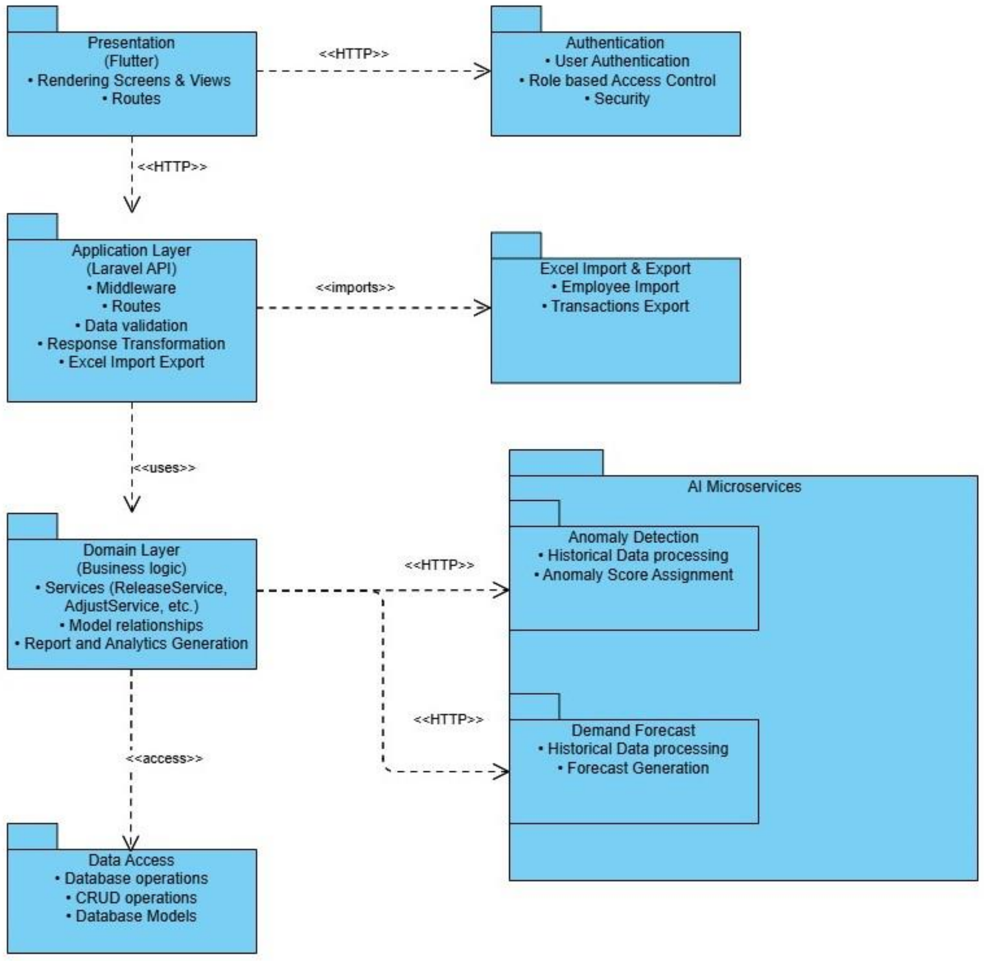


Figure 8.2: Package Diagram of the WMS System

8.4 Architectural Patterns

In our project we applied two complementary patterns: MVC on the backend (Laravel) and an MVVM-inspired, feature-based structure on the frontend (Flutter)—to keep concerns cleanly separated and maximize testability and maintainability.

8.4.1 Laravel Backend: MVC + Service Layer

- Models
  - Eloquent classes (Warehouse, Transaction, Item, etc.) map directly to database tables and encapsulate relationships, scopes, and accessors.
- Views / Resources
  - JSON Resources transform Eloquent models into API-friendly payloads, enforcing a clear boundary between raw data and client contracts.
- Controllers
  - Route-bound controllers (e.g. ReleaseController, AdjustmentController) handle HTTP request flow, delegate validation to FormRequest classes, and call into the service layer for business logic.
- FormRequest Classes
  - Centralize validation rules and authorization logic, keeping controllers slim and focused on orchestration.
- Service Layer & Traits
  - All core application rules live in dedicated service classes (e.g. WarehouseActivityService, ReleasesByHourService), ensuring the Single Responsibility Principle.
  - Traits factor out cross-cutting concerns (like Search Filer, Finding a model by id, etc.) so they can be reused without inheritance-based coupling.

This combination preserves Laravel’s native MVC flavor while elevating complex logic into reusable code.

8.4.2 Flutter Frontend: Feature-Based MVC

- Feature Folders
  - We organized the app by feature (Inventory, Warehouses, Releases,

Analytics), with each feature folder containing its own:

    - Model: Plain Dart data classes and parsing logic.
    - View (Screen): Widget trees that render state, compose reusable UI components, and respond to user input.
    - Controller: GetX controllers that expose observable properties, handle all business logic, and invoke API calls.

- MVC-Based Flow
    - Renders UI and forwards user interactions (taps, form submissions) to its Controller.
    - Controller fetches data from the Laravel API, processes it, and updates observables.
    - Model classes represent JSON payloads and drive UI rendering.

By coupling each screen’s UI with its own controller and model in a single feature module, we achieve Modularity, Clarity, and Reusability.

8.5 Design Class Diagram

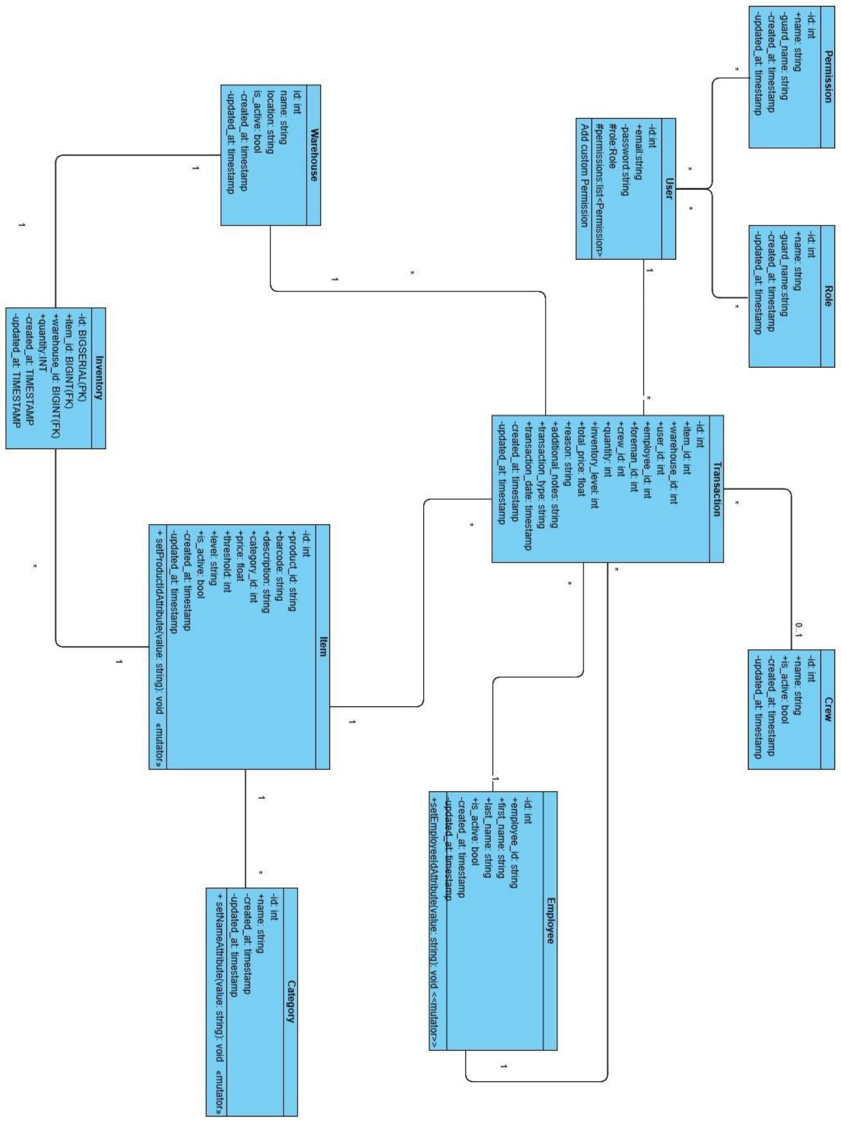


Figure 8.3 Design Class Diagram for Models in the System

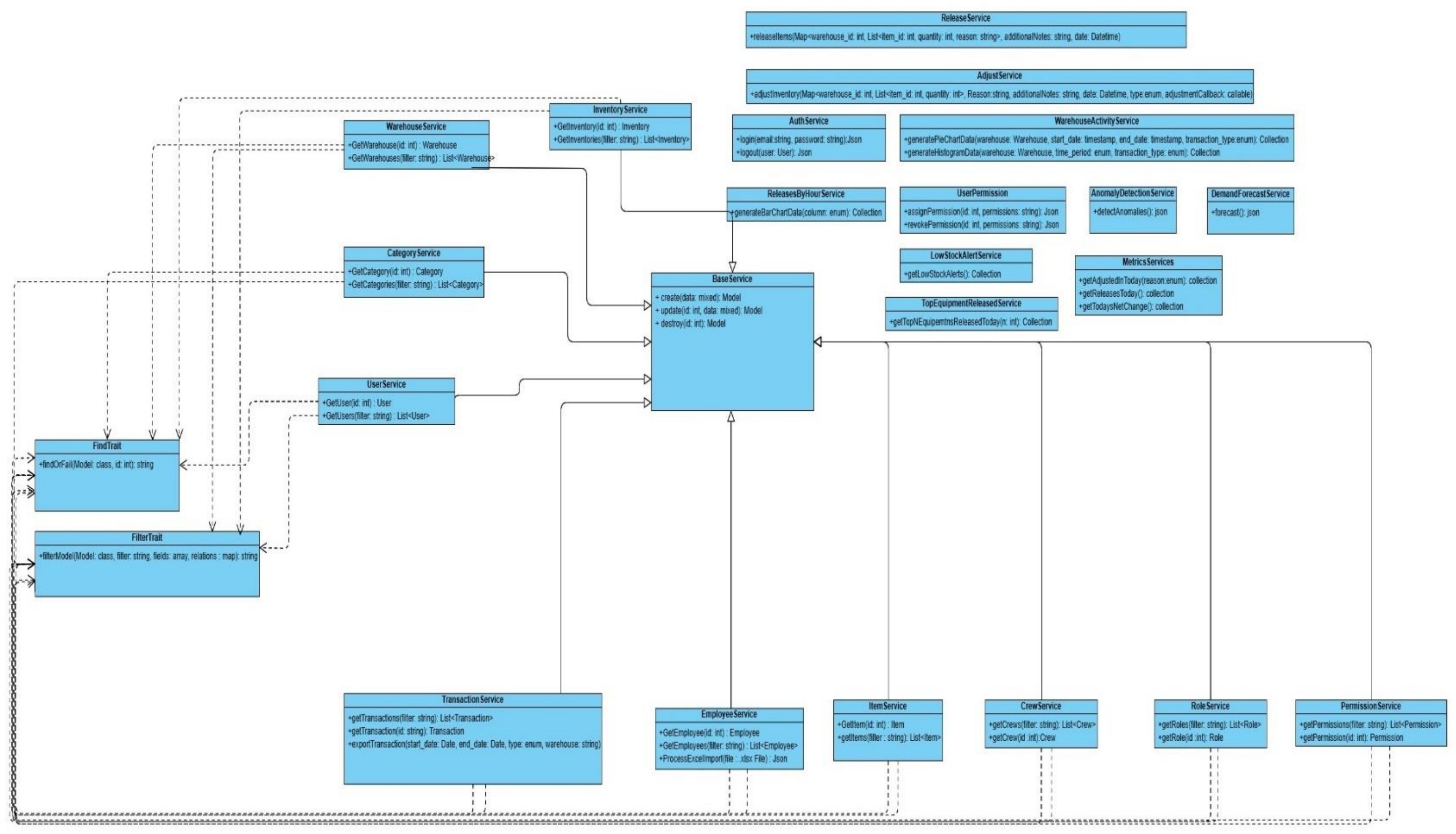

ReleaseService
+releaseItems(Map<warehouse_id: int, List<item_id: int, quantity: int, reason: string>, additionalNotes: string, date: Datetime)
AdjustService
+adjustInventory(Map<warehouse_id: int, List<item_id: int, quantity: int>, Reason:string, additionalNotes: string, date: Datetime, type:enum, adjustmentCallback: callable)
InventoryService
+GetInventory(id: int) : Inventory
+GetInventories(filter: string) : List<Inventory>
WarehouseService
+GetWarehouse(id: int) : Warehouse
+GetWarehouses(filter: string) : List<Warehouse>
AuthService
+login(email:string, password: string):Json
+logout(user: User): Json
WarehouseActivityService
+generatePieChartData(warehouse: Warehouse, start_date: timestamp, end_date: timestamp, transaction_type:enum): Collection
+generateHistogramData(warehouse: Warehouse, time_period: enum, transaction_type: enum): Collection
ReleasesByHourService
+generateBarChartData(column: enum): Collection
UserPermission
+assignPermission(id: int, permissions: string): Json
+revokePermission(id: int, permissions: string): Json
AnomalyDetectionService
+detectAnomalies(): json
DemandForecastService
+forecast(): json
CategoryService
+GetCategory(id: int) : Category
+GetCategories(filter: string) : List<Category>
LowStockAlertService
+getLowStockAlerts(): Collection
MetricsServices
+getAdjustedInToday(reason:enum): collection
+getReleasesToday(): collection
+getTodaysNetChange(): collection
BaseService
+ create(data: mixed): Model
+ update(id: int, data: mixed): Model
+ destroy(id: int): Model
TopEquipmentReleasedService
+getTopNEquipemtnsReleasedToday(n: int): Collection
UserService
+GetUser(id: int) : User
+GetUsers(filter: string) : List<User>
FindTrait
+findOrFail(Model: class, id: int): string
FilterTrait
+filterModel(Model: class, filter: string, fields: array, relations : map): string
TransactionService
+getTransactions(filter: string): List<Transaction>
+getTransaction(id: string): Transaction
+exportTransaction(start_date: Date, end_date: Date, type: enum, warehouse: string)
EmployeeService
+GetEmployee(id: int) : Employee
+GetEmployees(filter: string) : List<Employee>
+ProcessExcelImport(file : .xlsx File) : Json
ItemService
+GetItem(id: int) : Item
+getItems(filter : string): List<Item>
CrewService
+getCrews(filter: string): List<Crew>
+getCrew(id :int):Crew
RoleService
+getRoles(filter: string): List<Role>
+getRole(id :int): Role
PermissionService
+getPermissions(filter: string): List<Permission>
+getPermission(id: int): Permission

Figure 8.4: Design Class Diagram for Services in the System

These class diagrams show how the system is structured using Object-Oriented Design principles. The first diagram (Figure 8.1) represents the main model classes in the system, including User, Item, Inventory, Transaction, Warehouse, Category, Crew, and Employee. Each class contains relevant attributes and methods needed to manage the warehouse operations. Relationships between classes are clearly defined to reflect how records are linked in the database.

The second diagram (Figure 8.2) shows the service layer of the system. Each feature (e.g., items, users, warehouses, transactions) has its own dedicated service class. These services contain the business logic for their respective modules and are injected into the controller classes using Laravel's dependency injection system. This allows controllers to remain focused on handling HTTP requests while delegating logic to services. Each controller also uses Laravel Form Requests to validate incoming data and Laravel Resources to format the outgoing JSON responses. This structure keeps the code clean, reusable, and easy to test. It follows key Object-Oriented Design principles like the Single Responsibility Principle and Dependency Inversion Principle.

8.6 Database Design Diagram

The database design diagram shows the relational structure of the WMS. It includes key tables such as users, roles, permissions, items, inventory, transactions, warehouses, employees, crews, and categories. Relationships are defined using foreign keys to maintain data consistency, for example, linking transactions to items, warehouses, crews, and employees. The employee table is referenced twice in the transactions table: once as the foreman and once as the regular employee. This structure supports all core operations like receiving, releasing, adjusting, and tracking items, while ensuring reliable and consistent data across the system.

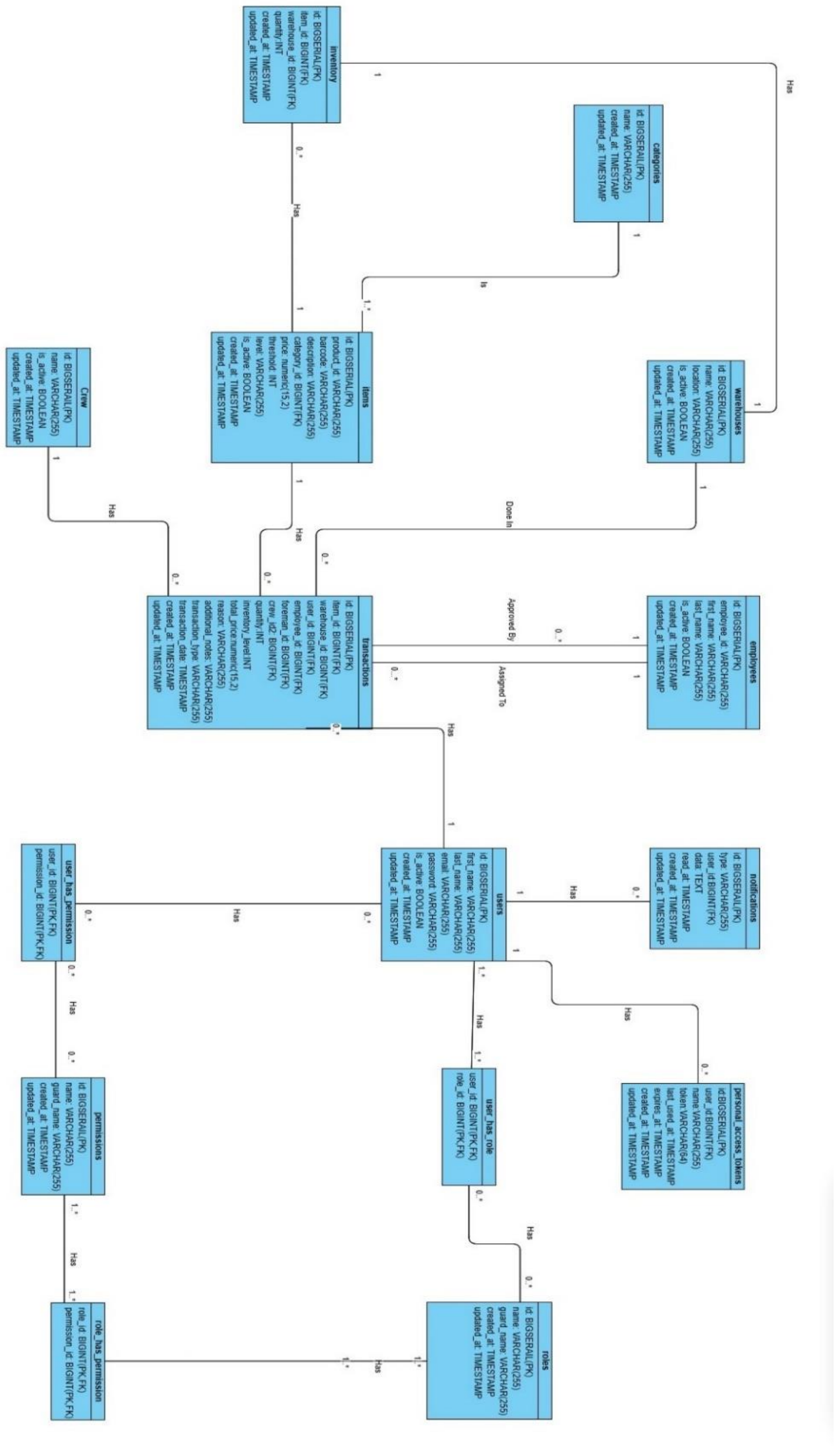


Figure 8.5: Database Design Diagram

Chapter 9: User Interface Design

9.1 Preliminary UI Sketches

User interface and user experience were essential parts of our app development, particularly because many of the staff members initially had doubts about using new technology. Understanding these concerns, we aimed to create an app that feels simple and easy to use, minimizing the steps needed to complete daily tasks.

Early on, we used wireframes and sketches, like the ones shown here, to map out the user journey clearly. These visuals helped us simplify interactions, ensuring the app felt intuitive from the login screen onward.

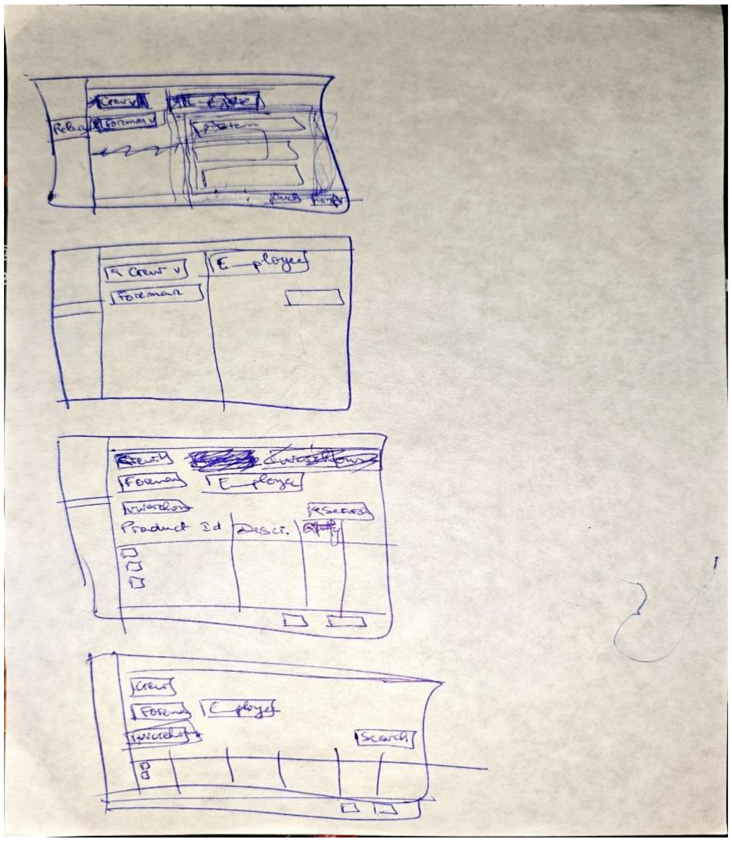


Figure 9.1: Initial Sketches for Releasing Feature

One important suggestion from our supervisor was adding radio buttons to the login screen. This allowed users to quickly select their email domain instead of typing it, making login faster and easier. Small changes like this have made a big difference in user comfort and efficiency.

We also considered psychology in our design choices, especially with colors. We used blue in the interface to create a sense of comfort and trust, making the experience more welcoming and reassuring for users who might feel anxious about using technology.

These initial sketches and thoughtful design considerations helped align our development closely with user preferences, boosting user confidence and improving overall satisfaction with the app.

9.2 Final UI Design

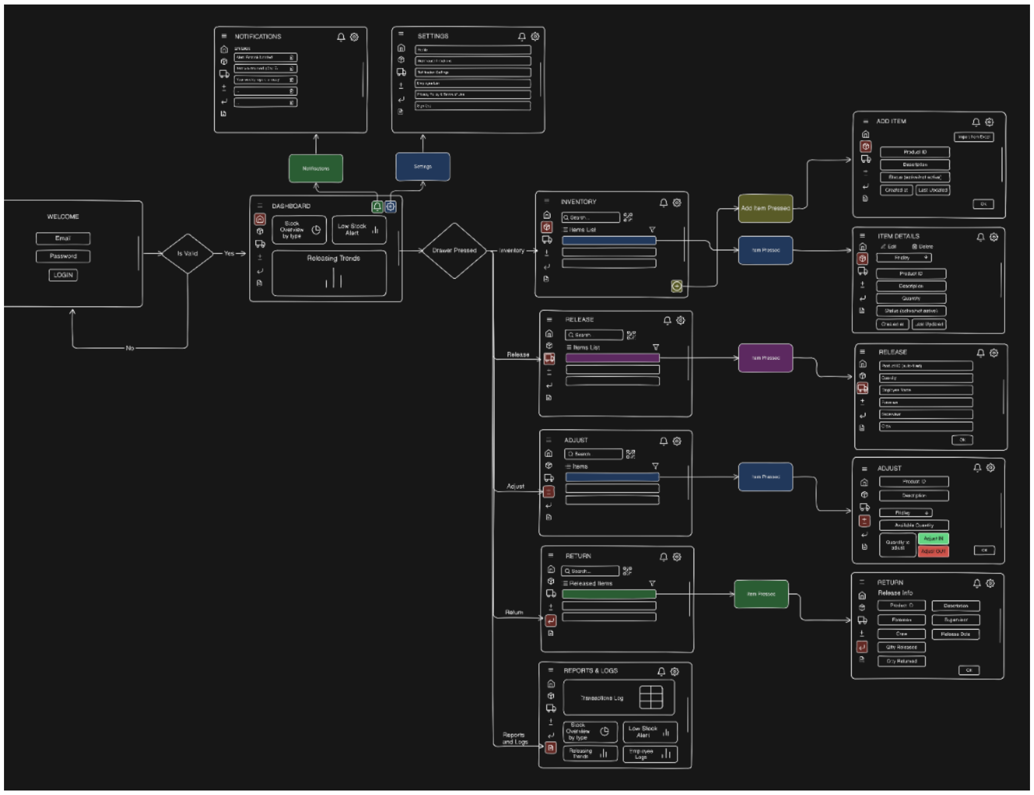


Figure 9.2: High Level UI Wireframe

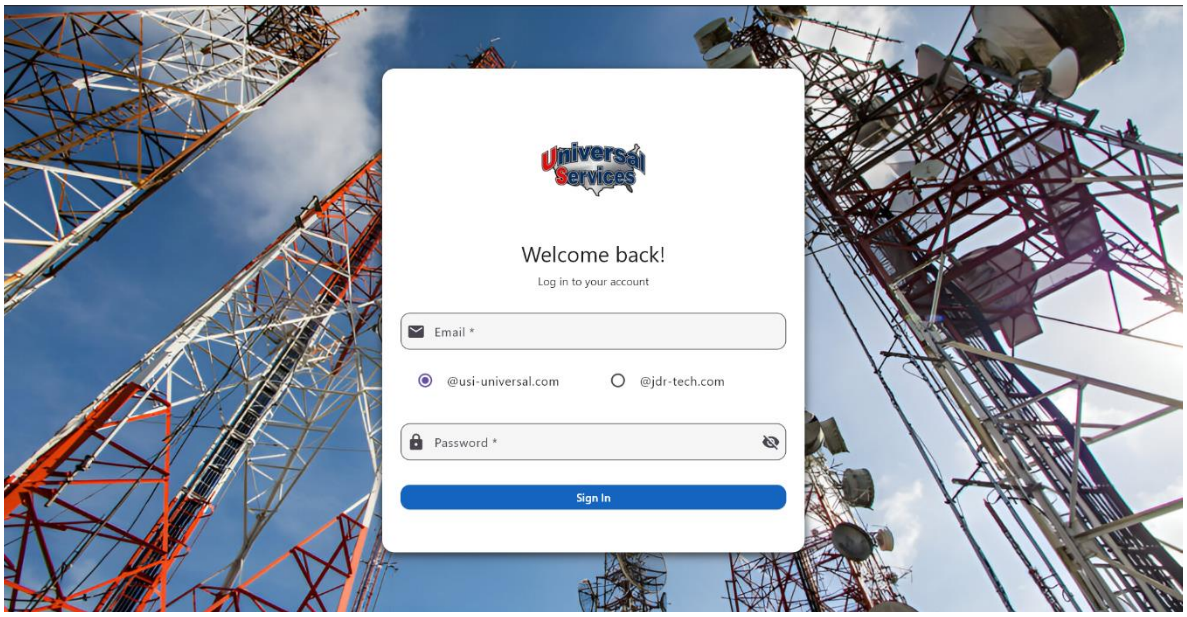


Figure 9.3: Login Screen

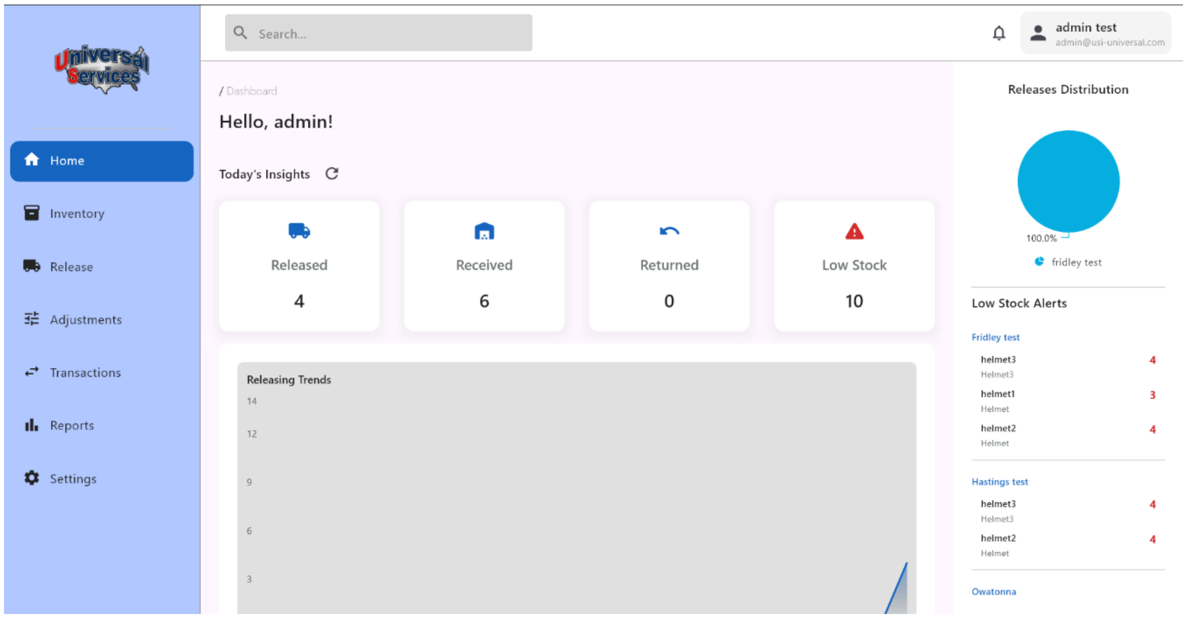
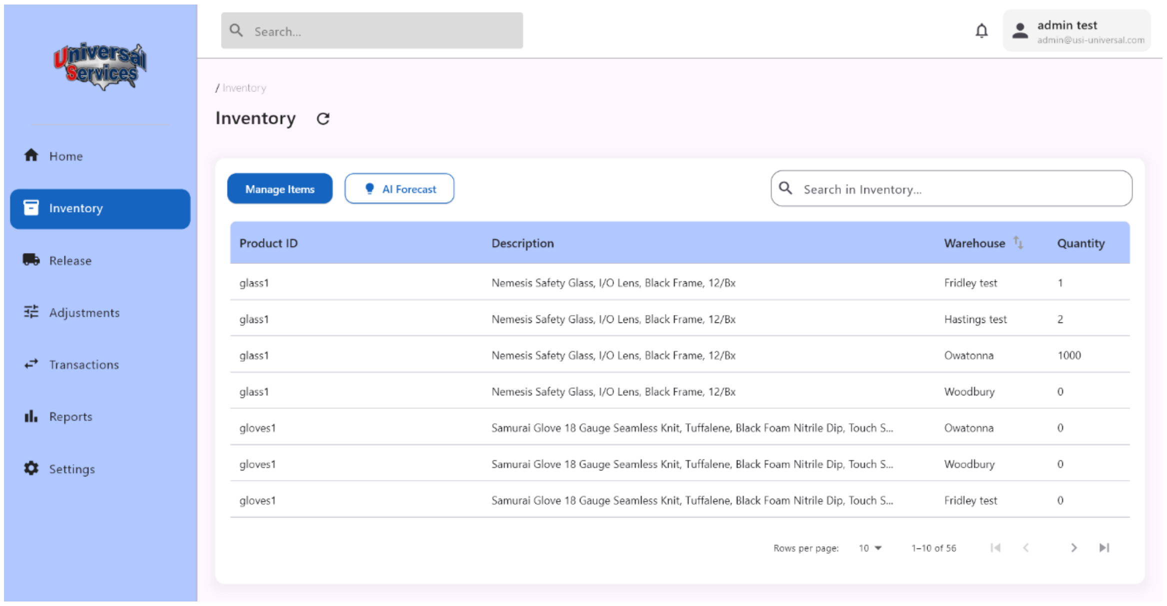
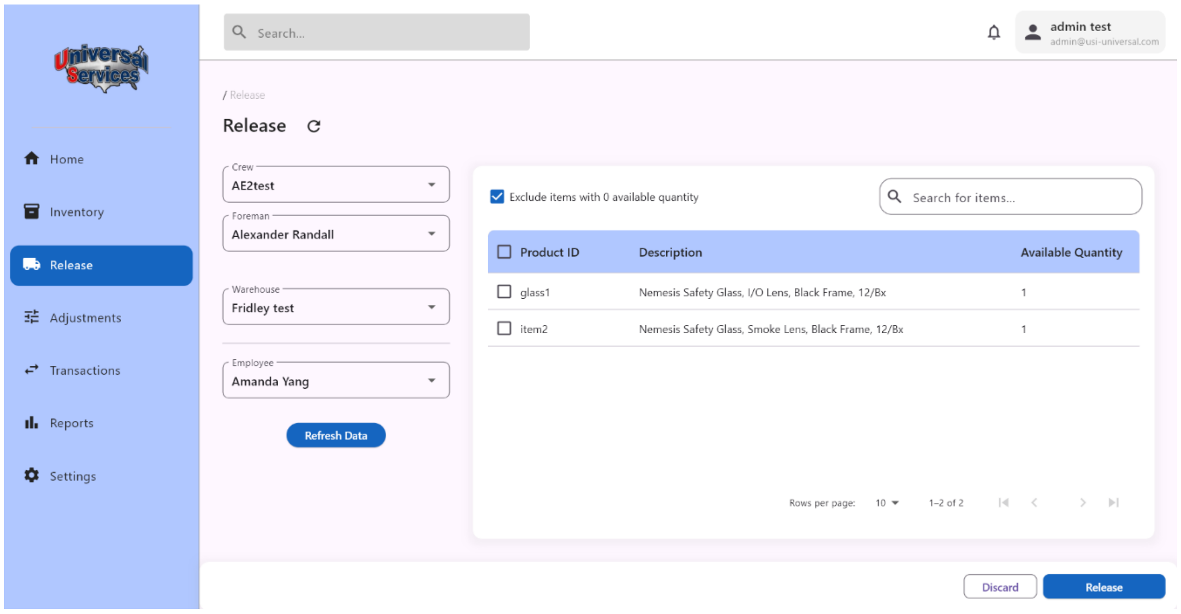


Figure 9.4: Dashboard



Product ID	Description	Warehouse	Quantity
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	Fridley test	1
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	Hastings test	2
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	Owatonna	1000
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	Woodbury	0
gloves1	Samurai Glove 18 Gauge Seamless Knit, Tuffalene, Black Foam Nitrile Dip, Touch S...	Owatonna	0
gloves1	Samurai Glove 18 Gauge Seamless Knit, Tuffalene, Black Foam Nitrile Dip, Touch S...	Woodbury	0
gloves1	Samurai Glove 18 Gauge Seamless Knit, Tuffalene, Black Foam Nitrile Dip, Touch S...	Fridley test	0

Figure 9.5: Inventory

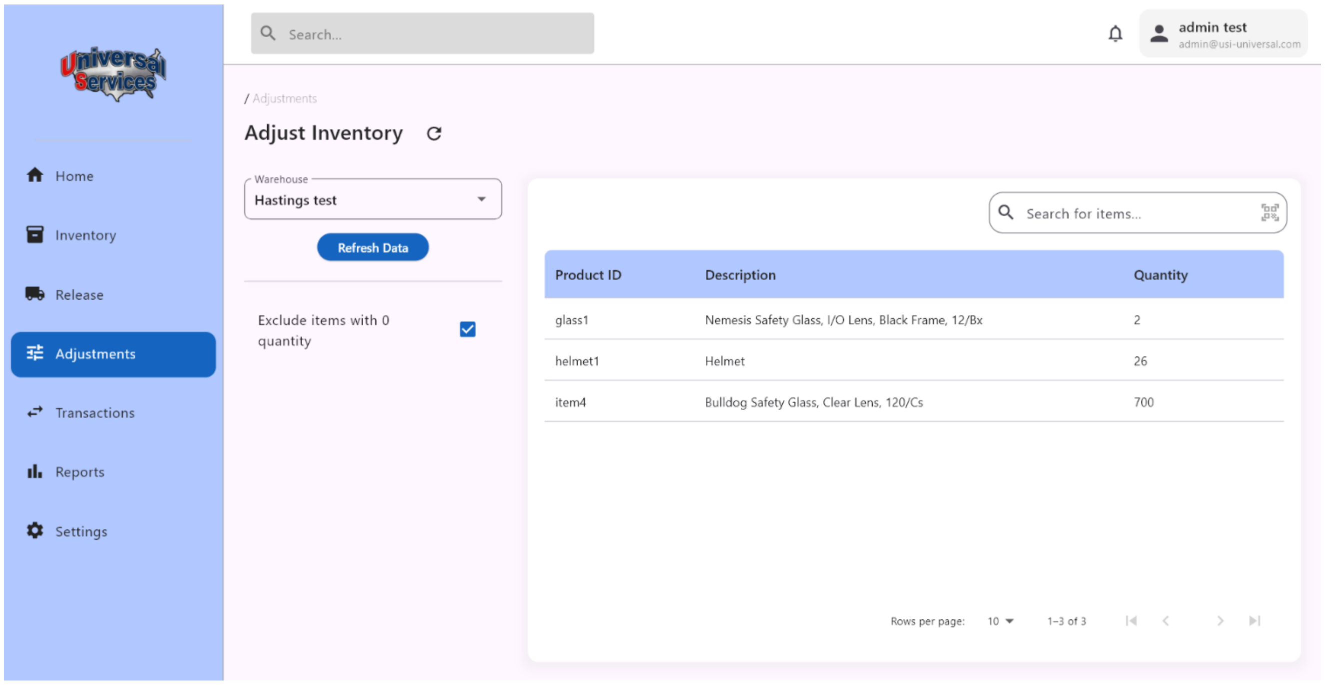


Crew: AE2test
Foreman: Alexander Randall
Warehouse: Fridley test
Employee: Amanda Yang

☑ Exclude items with 0 available quantity

Product ID	Description	Available Quantity
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	1
item2	Nemesis Safety Glass, Smoke Lens, Black Frame, 12/Bx	1

Figure 9.6: Release



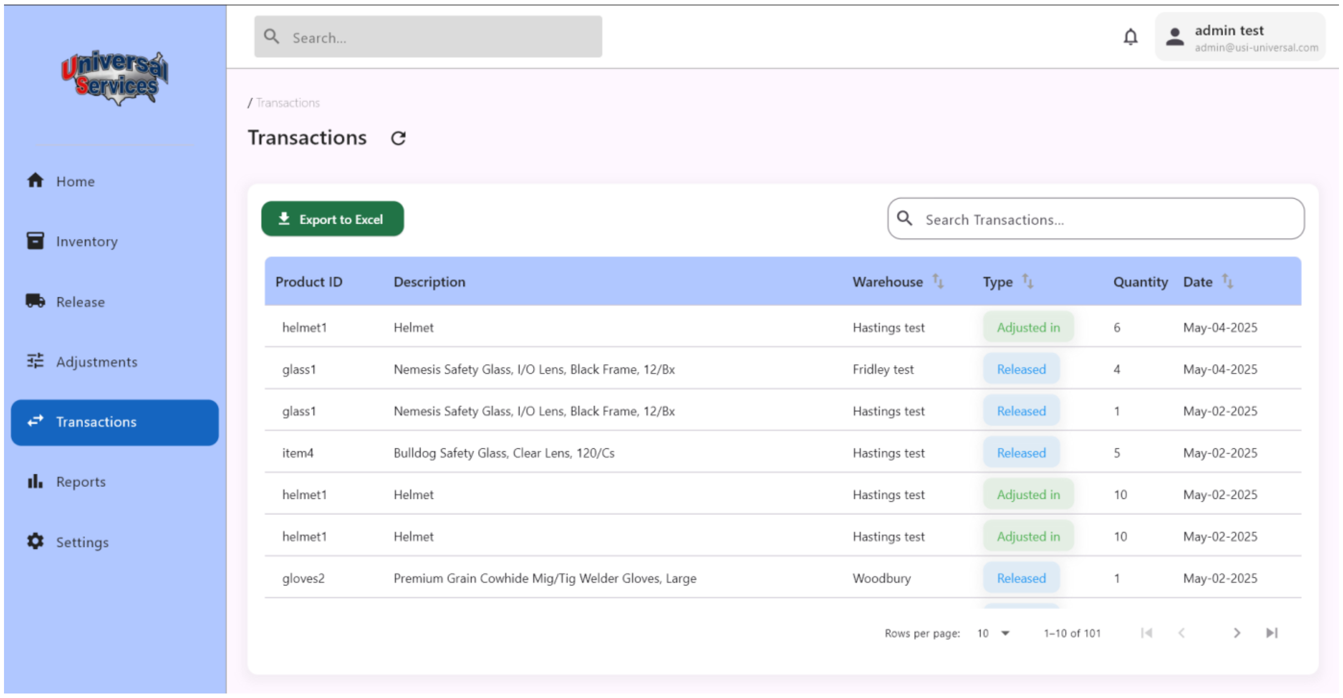
Adjust Inventory

Warehouse: Hastings test

Exclude items with 0 quantity ✓

Product ID	Description	Quantity
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	2
helmet1	Helmet	26
item4	Bulldog Safety Glass, Clear Lens, 120/Cs	700

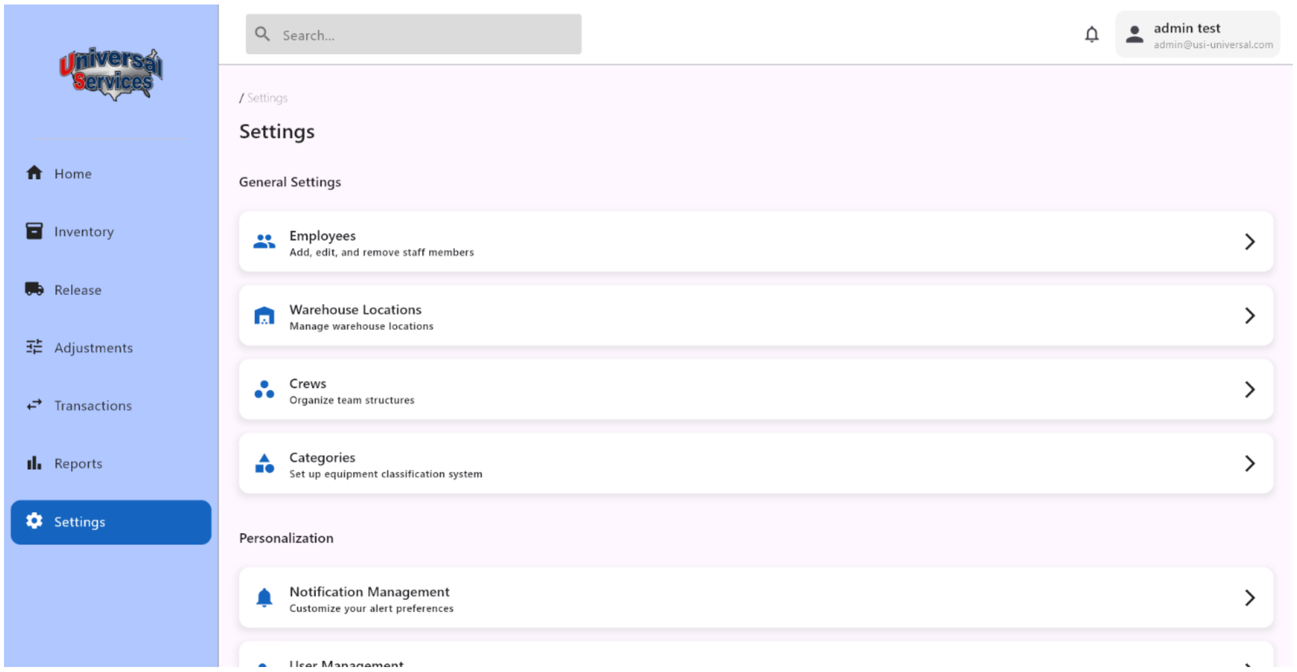
Figure 9.7: Adjustments



Transactions

Product ID	Description	Warehouse	Type	Quantity	Date
helmet1	Helmet	Hastings test	Adjusted in	6	May-04-2025
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	Fridley test	Released	4	May-04-2025
glass1	Nemesis Safety Glass, I/O Lens, Black Frame, 12/Bx	Hastings test	Released	1	May-02-2025
item4	Bulldog Safety Glass, Clear Lens, 120/Cs	Hastings test	Released	5	May-02-2025
helmet1	Helmet	Hastings test	Adjusted in	10	May-02-2025
helmet1	Helmet	Hastings test	Adjusted in	10	May-02-2025
gloves2	Premium Grain Cowhide Mig/Tig Welder Gloves, Large	Woodbury	Released	1	May-02-2025

Figure 9.8: Transactions Log



Settings

General Settings

Employees
Add, edit, and remove staff members

Warehouse Locations
Manage warehouse locations

Crews
Organize team structures

Categories
Set up equipment classification system

Personalization

Notification Management
Customize your alert preferences

User Management

Figure 9.9: Settings

Chapter 10: Coding and Testing

10.1 Algorithms

10.1.1 Histogram Data Aggregation Algorithm

Listing 10.1: Laravel service method to generate histogram data with date bucketing and transaction aggregation

```
public function generateHistogramData(array $data)
  {
      // Extract input filters or use default values
      $warehouseName = $data['warehouse'] ?? null;
      $transactionType = $data['transaction_type'] ?? null;
      $time = $data['time'] ?? 'last_7_days';

      // Determine the start date based on the selected time interval
      $startDate = $this->getStartDate($time);

      // Determine the appropriate date format for grouping (daily, weekly, or monthly)
      $dateFormat = $this->getDateFormat($time);

      // Pre-fill all expected date intervals with 0 to ensure consistent histogram bars
      $allDates = $this->initializeDateBuckets($time, $startDate);

      // Build the query and apply filters: warehouse name, transaction type, and date range
      $query = $this->applyFilters(
          Transaction::query()->with('warehouse'),
          $warehouseName,
          $transactionType,
          $startDate
      );


// Retrieve transactions, group by formatted date, and sum the quantities per group
      $actualData = $query->get()
          ->groupBy(fn ($item) => $item->transaction_date-> format($dateFormat))
          ->map(fn ($group) => $group->sum('quantity'));
```

```
        // Populate the initialized $allDates array with actual data where available
        foreach ($actualData as $date => $totalQuantity) {
            $allDates[$date] = $totalQuantity;
        }

        // Convert the final array to a collection and format for chart consumption (label +
quantity)
        return collect($allDates)->map(fn ($quantity, $date) => [
            'barDate' => $date,
            'barQuantity' => $quantity,
        ])->values();
    }
```

10.1.2 Hourly Bar Chart Aggregation Algorithm

Listing 10.2: Laravel method to compute hourly totals of released transactions over the past 24 hours

```
 public function generateBarChartData($column = 'quantity')
   {
      // Define the time range: from 24 hours ago up to the current hour (inclusive)
      $end  = now()->addHour()->startOfHour();         // Round up to the next hour
      $start = $end->copy()->subDay()->startOfHour();   // Subtract 24 hours from the end
time

      // Fetch all 'released' transactions within the defined time range
      $transactions = Transaction::where('transactions_type', 'released')
          ->whereBetween('transaction_date', [$start, $end])
          ->get();

      // Group transactions by the number of hours since the start (i.e., hour buckets 0 to 23)
      $transactionsGroupedByHour = $transactions->groupBy(function ($transaction) use
($start) {
          return (int) $start->diffInHours($transaction->transaction_date);
      });

      // Initialize a result collection with 24 hourly points (0 to 23)
      $result = collect(range(0, 23))->map(function ($hour) use
($transactionsGroupedByHour, $start, $column) {
          // Get all transactions for the current hour bucket
          $transactionOfCurrentHour = $transactionsGroupedByHour->get($hour, collect());

          return [
              // Format the current hour as a timestamp label
              'hour'  => $start->copy()->addHours($hour)->toDateTimeString(),

              // Sum the specified column (e.g., 'quantity') for this hour
```

```
            'total' => $transactionOfCurrentHour->sum($column),
        ];
    });

    // Return the complete 24-hour bar chart data as a collection
    return $result;
}
```

10.1.3 Demand Forecasting Algorithm

Listing 10.3: Python function to forecast demand using released transaction quantities with current inventory and generate restock suggestions using Prophet

```
def generate_forecast(df: pd.DataFrame, forecast_window: int = 7) -> Dict[str, Any]:

    df['transaction_date'] = pd.to_datetime(df['transaction_date'])

    df_release = df[df['transaction_type'] == 'released']
    if df_release.empty:
        return {"error": "No release transactions available for forecast"}

    # Sum quantities by day
    ts = df_release.groupby('transaction_date')['quantity'].sum().reset_index()
    ts.columns = ['ds', 'y']

    # Check if there are enough data points
    if len(ts) < 10:
        return {"error": "Not enough data to forecast (minimum 10 data points required)"}

    # Initialize and fit the Prophet model with daily seasonality
    model = Prophet(daily_seasonality=True)
    try:
        model.fit(ts)
    except Exception as e:
        logger.error("Error fitting Prophet model: %s", e)
        return {"error": "Model fitting failed: " + str(e)}

    # Calculate tomorrow's date
    tomorrow = datetime.now().date() + timedelta(days=1)
    # Create a range of dates starting from tomorrow with the desired forecast window but
only generate weekdays (Mon–Fri) B as a freq means bussiness days
    future_dates = pd.date_range(start=tomorrow, periods=forecast_window, freq='B')

    # Build the future DataFrame for Prophet
    future = pd.DataFrame({"ds": future_dates})
    forecast = model.predict(future)[['ds', 'yhat']]
    forecast['yhat'] = forecast['yhat'].clip(lower=0)

    # Sum the forecasted values and convert to a plain Python float
    total_forecast = float(forecast['yhat'].sum())

    # Retrieve the latest inventory level and convert to float
    current_inventory = float(df.sort_values('transaction_date').iloc[-1]['inventory_level'])
    restock_suggestion = max(0, total_forecast - current_inventory)

    # Build the response payload, ensuring all numeric values are converted.
    response_payload = {
        "item_id": None,
        "warehouse_id": None,
        "forecast": [
            {
                "date": row['ds'].strftime('%Y-%m-%d'),
                "predicted_quantity": float(round(row['yhat'], 2))
            }
            for _, row in forecast.iterrows()
        ],
        "inventory_level": float(round(current_inventory, 2)),
        "total_forecast_next_days": float(round(total_forecast, 2)),
        "restock_suggestion": float(round(restock_suggestion, 2))
    }
    return response_payload;
```

10.2 Unit Tests

Table 10.1: Summary of Unit Tests

Description	Input	Expected Output
Test 1: 7-Day Released Quantity Histogram Calculation	1 “released” transaction with a quantity of 100 items made 3 days ago	
	Method call: time=last_7_days, warehouse=Main, transaction type=released	7-element histogram array with bar quantity sum equal to at least100
Test 2: 7-Day Histogram with Unknown Warehouse Returns Zeros	Method call with a non-existent warehouse name	7-element histogram array with bar quantity sum equal to 0
Test 3: 24-Hour Released Quantity Bar Chart Calculation	Method Call with 1 “released” transaction with a quantity of 75 items made 2 hours ago.	24 hourly buckets: • Each entry has hour & total • One bucket’s total is 75, while all others should be 0
Test 4: 24-Hour Chart with No Transactions Returns All Zeros	Method Call with no transactions at all	24 hourly buckets, each with a total of 0
Test 5: API Endpoints & User Acceptance	Full WMS Postman collection run against the testing server	• All endpoints return correct HTTP status codes and JSON schemas • Errors handled properly
	Two weeks of use by warehouse staff	UI/UX and bug feedback

10.2.1 Test 1: 7-Day Released Quantity Histogram Calculation

```
public function test_histogram_7_days_returns_correct_totals()
   {
      // Create a single “released” transaction with 100 quantity and a date of three days ago
      Transaction::factory()->create([
         'transactions_type' => 'released',
         'transaction_date' => now()->subDays(3),
         'quantity' => 100,
         'warehouse_id' => 1,
         'item_id' => 1,
         'employee_id' => 1,
         'foreman_id' => 1,
         'crew_id' => 1,
         'user_id' => 1,
      ]);

      // Generate data for a 7-day histogram on the “Main” warehouse
      $service = new WarehouseActivityService();
      $data    = $service->generateHistogramData([
         'time' => 'last_7_days',
         'warehouse'=> 'Main',
         'transaction_type'=> 'released',
      ]);

      // Assert that we still get seven days, and that the total released is at least 100
      $this->assertCount(7, $data);
      $this->assertGreaterThanOrEqual(100, collect($data)->sum('barQuantity'));
   }
```

10.2.2 Test 2: 7Day Histogram with Unknown Warehouse Returns Zeros

```
 public function test_histogram_with_invalid_warehouse_returns_zero_totals()
   {
      // Generate data for a 7-day histogram for a warehouse name that doesn’t exist
      $service = new WarehouseActivityService();
      $data    = $service->generateHistogramData([
         'time' => 'last_7_days',
         'warehouse' => 'FakeMain',
         'transaction_type' => 'released',
      ]);

      // Assert that We still get seven days, but no releases recorded (all zero)
```

```
        $this->assertCount(7, $data);
        $this->assertEquals(0, collect($data)->sum('barQuantity'));
    }
```

10.2.3 Test 3: 24-Hour Released Quantity Bar Chart Calculation

```
public function test_hourly_chart_returns_24_entries()
    {
        // Create a single “released” transaction two hours ago
        Transaction::factory()->create([
            'transactions_type'=> 'released',
            'transaction_date'=> now()->subHours(2),
            'quantity'=> 75,
            'warehouse_id' => 1,
            'item_id' => 1,
            'employee_id' => 1,
            'foreman_id' => 1,
            'crew_id' => 1,
            'user_id' => 1,
        ]);

        // Generate the 24-hour releases chart data
        $service = new ReleasesByHourService();
        $data    = $service->generateBarChartData();

        // Assert that we get exactly 24 hourly buckets, and each bucket entry has an “hour”
and a “total”
        $this->assertCount(24, $data);
        $this->assertArrayHasKey('hour', $data[0]);
        $this->assertArrayHasKey('total', $data[0]);
    }
```

10.2.4 Test 4: 24-Hour Chart with No Transactions Returns All Zeros

```
public function test_hourly_chart_with_no_data_returns_all_zeroes()
    {
        // Generate the 24-hour releases chart with no transactions at all
        $service = new ReleasesByHourService();
        $data = $service->generateBarChartData();

        // Assert that every “total” value in the 24 buckets is zero
        $totals = collect($data)->pluck('total');
        $this->assertTrue($totals->every(fn ($qty) => $qty === 0));    }
```

10.2.5 Test 5: API Endpoints & User Acceptance

Description:

1. Postman Smoke Testing:
   - Imported our full WMS collection into Postman (see screenshot below).
   - Ran each CRUD request (Inventory, Warehouses, Items, Employees, Transactions, Analytics) against our testing server.
   - Verified that each endpoint returned the correct HTTP status (200/201/404/etc) and JSON schema against many scenarios.

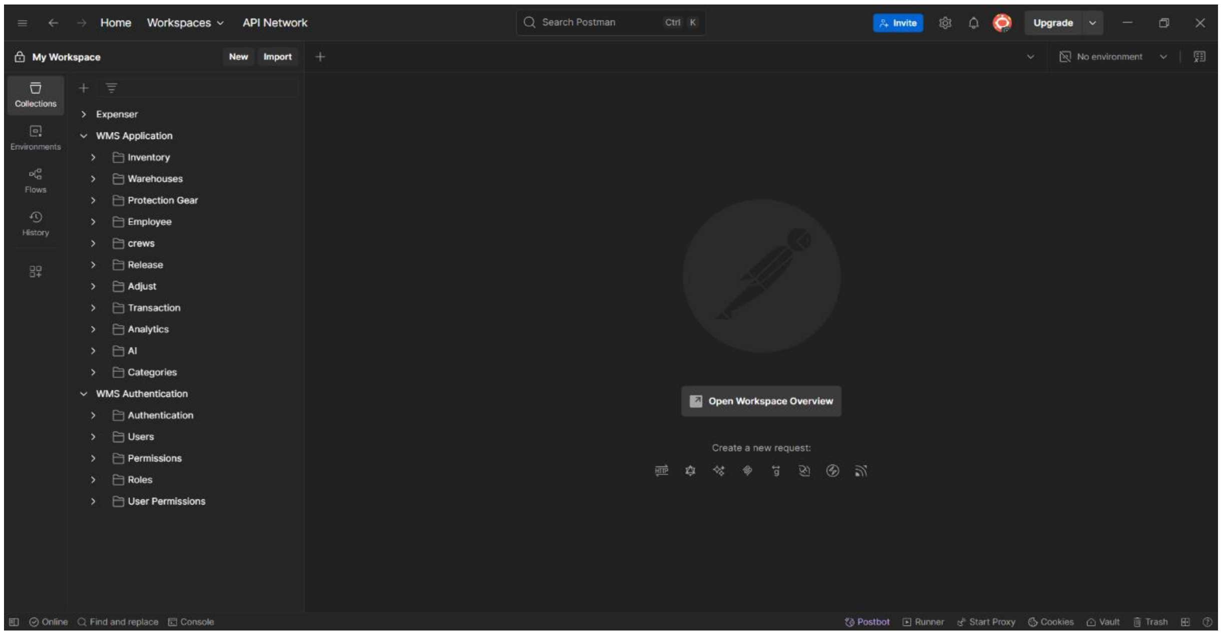


Figure 10.1: Postman API Collection for WMS Application

2. Production Roll-Out & Feedback:
   - Deployed the app to our partner company’s environment.
   - Had real warehouse staff and managers use the system for two weeks.
   - Collected bug reports, UX suggestions, and performance notes via a shared feedback channel.

Observations:

- All endpoints were iteratively refined until they consistently passed smoke tests with the expected codes and payloads.
- Validation failures, “Not Found” errors, and other edge cases are handled gracefully, returning the proper HTTP status and error message.
- Users found the UI intuitive and requested enhancements such as faster bulk uploads via the Excel import and expanded analytics dashboards.
- No critical production issues were ever found, only minor bugs and UI tweaks, which were quickly patched.

Chapter 11: Collaboration and Individual Contribution

11.1 Collaboration Approach

There were two fixed remote weekly meetings with the stakeholders (the company we worked for), during which we received continuous feedback, clarified new requirements, and confirmed upcoming deadlines. In addition, we held daily check-in meetings, either remotely or face-to-face, each lasting a minimum of 20 minutes.

To support effective teamwork and coordination, we relied on several tools throughout the project:

- **Instant Communication:** We used WhatsApp and Microsoft Teams to exchange updates, coordinate tasks, and stay aligned on daily progress.
- **Virtual Meetings:** Stakeholder meetings were held via Microsoft Teams to ensure ongoing engagement and quick resolution of any issues.
- **In-Person Work Sessions:** We regularly gathered at the university library to collaborate on major milestones, including system integration, testing, and final refinements.
- **Source Code Management:** GitHub served as our main platform for managing code versions, collaborating on features, and tracking development history.

We followed an incremental development approach, which suited our case due to frequent changes in requirements. The first release included the minimum viable set of features, after which we continued improving and extending the system. A new version was deployed every two to three weeks, based on stakeholder feedback and testing outcomes.

11.2 Problems that Occurred

Several challenges were encountered during the project and had to be addressed directly:

- **Backend Technology Change:** We originally began the project using Dart for backend logic. However, we faced limitations in terms of structure, scalability, and API management. As a result, we decided to switch to Laravel, which better supported backend separation and secure API handling. This shift required reworking the backend code, but it improved the system in the long run.
- **Scheduling Conflicts:** Because our client was based in the United States and we were in Lebanon, there was a significant time zone difference. Combined with university classes, this made it difficult to find common meeting times. To solve this, we set fixed weekly meetings and used messaging apps and shared documents to coordinate outside those sessions.
- **Miscommunication of Requirements:** In some cases, the requirements we received were unclear or incomplete. This led us to work on features that later had to be redone. We handled this by asking for more specific feedback in every meeting and confirming details before starting new work.
- **Frequent Requirement Changes:** The client often changed their mind or refined their expectations after we had already completed certain features. To manage this, we designed the system to be flexible and modular from the start, allowing us to make changes with minimal rework.

Each of these problems required fast decisions and team coordination. We adapted quickly and improved our processes to keep the project on track.

11.3 Individual Contribution Breakdown

Table 11.1: Individual Contribution

Task	**Anthony Lahoud**	**Anthony Nasry Massaad**
UI Design and Frontend Development	5%	95%
Backend and API Development	95%	5%
Database Design and Implementation	60%	40%
Demand Forecast	95%	5%
Anomaly Detection	5%	95%
Report Writing and Documentation	50%	50%

Chapter 12: Conclusions and Recommendations

12.1 Summary of the Main Results

This senior project aimed to design and develop a Warehouse Management System WMS for tracking and managing inventory across multiple warehouses. The system includes mobile, tablet, and desktop applications, built using Flutter for the frontend and Laravel for the backend, with PostgreSQL used for database management. The project was completed over a period of approximately eight months.

During the first two months, we worked with the company using Excel VBA tools to understand their existing workflow and started collecting initial requirements. In the third month, we continued gathering more detailed requirements while testing their current systems and tools. From the fourth to sixth month, we began the actual development process, focusing on core features such as item receiving, releasing, inventory tracking, user and warehouse management, and transaction logging. In the final two months, we worked on advanced functionalities, including role-based access control, reporting features, system analytics, and AI integration for forecasting and anomaly detection.

12.2 Main Contributions

This project contributed in several important ways:

- **Academic Contribution:** It covered multiple fields within computer science, including Software Engineering, App Development, Database Design, RESTful APIs, AI integration, and Data Sciences.
- **Technical Skills:** We gained hands-on experience with Laravel, Flutter, PostgreSQL, clouds, and deployment methods. We also implemented a secure backend structure with authentication and permissions middleware.
- **Real-World Impact:** The system was built from scratch, covering all phases of the software development lifecycle, from analysis and design to testing and

deployment. The project required continuous collaboration with stakeholders and adapting to real business needs.

12.3 Possible Extensions and Future Work

While the core system is fully functional, there are several improvements and extensions that could be made in the future, such as:

- Allowing offline operation and background sync for unstable warehouse internet conditions
- Advanced Analytics and Dashboards: Add more customizable interactive graphs, trends, and filters to help managers make data-driven decisions.
- Expanded AI Features: Introduce new AI components such as computer vision for automatic object detection during receiving or releasing, and pattern recognition for item usage trends.

This system is currently in active use by a company based in the United States called Universal Service, and is deployed across multiple warehouses in different states. The project has transitioned from a student academic project to a live, working enterprise application.

List of References

Laravel Documentation: https://laravel.com/docs

PHP Documentation: https://www.php.net/docs.php

Flutter Documentation: https://docs.flutter.dev

PostgreSQL Documentation: https://www.postgresql.org/docs

Laracasts (Laravel Learning Platform): https://laracasts.com

Martin, R. C. (2008). Clean code: A handbook of agile software craftsmanship.

Appendix A: Anomaly Detection

```
def detect_anomalies(data: List[Transaction]):
    try:
        df = pd.DataFrame([d.dict() for d in data])

        if df.empty:
            raise ValueError("Input data is empty.")

        # Convert date to datetime and add time-based features
        df["date"] = pd.to_datetime(df["date"])
        df["hour"] = df["date"].dt.hour
        df["day_of_week"] = df["date"].dt.dayofweek
        df["is_weekend"] = (df["day_of_week"] >= 5).astype(int)

        results = []
        grouped = df.groupby(['item_id', 'warehouse_id'])

        for (item_id, warehouse_id), group in grouped:
            if len(group) < 5:
                for _, row in group.iterrows():
                    results.append(AnomalyResult(
                        transaction_id=row["transaction_id"],
                        anomaly_score=0.0,
                        importance="Low"
                    ))
                continue

            # Prepare features
            features = ["net_quantity", "inventory_level", "hour", "day_of_week",
"is_weekend"]
            scaler = StandardScaler()
            X_scaled = scaler.fit_transform(group[features])

            # Train model
            model = IsolationForest(contamination='auto', random_state=42)
            model.fit(X_scaled)

            group["anomaly_score"] = model.decision_function(X_scaled)
            group["importance"] = group["anomaly_score"].apply(categorize_importance)

            for _, row in group.iterrows():
                results.append(AnomalyResult(
                    transaction_id=row["transaction_id"],
                    anomaly_score=row["anomaly_score"],
                    importance=row["importance"]
                ))
        return results
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
```