BEACONHOUSE NATIONAL UNIVERSITY

PRJ-F23/333

WASAIL

EXTERNAL SUPERVISOR

HAMZA ZAFAR

INTERNAL SUPERVISOR

HUDA SARFRAZ

GROUP MEMBERS

FATIMA ALI TIRMIZI	**F2020-718**
FIZZA ADEEL	**F2020-336**
IRTAZA AHMED KHAN	**F2020-153**
MALAIKA SULTAN	**F2020-661**

SCHOOL OF COMPUTER & INFORMATION TECHNOLOGY

Project Title	Wasail	
Project ID	PRJ-F23/333	
Project Supervisors	Hamza Zafar (External)	
	Huda Sarfraz (Internal)	
Group Members	Fatima Ali Tirmizi	F2020-718
	Fizza Adeel	F2020-336
	Irtaza Ahmed Khan	F2020-153
	Malaika Sultan	F2020-661
Academic Session	2023-24 (Sept 2023 to June 2024)	
Credit Hours	6	

PROJECT APPROVAL

This Project is approved in partial fulfilment of the requirements of BSc (Hons.) in Computer Science degree conducted by the School of Computer and IT, Beaconhouse National University, Lahore.

_____

Hamza Zafar
External Supervisor

_____

(Name of Dean)
Dean School of Computer and IT

_____

Huda Sarfraz
Internal Supervisor

Date: _____

Acknowledgement

We would like to express our deepest gratitude to all those who have supported and guided us throughout the duration of our final year project. First and foremost, we are immensely grateful to our project supervisor, Ms. Huda Sarfraz, for her invaluable advice, continuous encouragement, and insightful feedback, which were crucial in shaping the direction and success of this project.

We would also like to express our profound appreciation towards our external supervisor, Mr. Hamza Zafar, for his invaluable encouragement and guidance which enabled us to reach our goal.

Additionally, We extend our heartfelt thanks to our department faculty members for their teaching and guidance, which has been instrumental in providing us with the foundational knowledge necessary for this project.

Table of Contents

Introduction to the Project.....12
Existing Systems.....14
Literature review.....22
Requirement Analysis.....25
Requirement Gathering and Fact Finding.....27
System Environment.....30
Machine Learning Requirements.....30
Data Collection.....30
Local Vendor Dataset.....30
Local Pharmacy Dataset.....32
Corporación Favorita Grocery Sales Forecasting (Kaggle).....33
Instacart Market Basket Analysis (Kaggle).....34
Other Datasets for Further Exploration.....35
Feature Engineering.....35
Model Shortlisting.....37
Approach Considerations.....37
Ensemble Approach.....37
Non-Ensemble Approach.....37
Type of ML Models.....37
Autoregressive Models.....37
Exponential Smoothing Models.....37
Machine Learning Regression Models.....38
Ensemble Models.....38
Deep Learning Models.....38
Competitive Analysis Models.....38
Model Training and Testing.....38
Integration with User Interface.....40
Recommendation Updates.....40
Feedback and Learning.....40
Performance and Scalability.....41
Testing and Validation.....41
User Roles.....41
User Stories.....41
User Story: Placing the Order.....41
User Story: Searching for a Product.....42
Functional Requirements.....43

Grocery Store and Vendor (FR1)..... 43
FR1.1: Language Selection..... 43
FR1.2: Phone Registration..... 44
FR1.3: Phone Number Confirmation..... 46
FR1.4: Phone Number Exists..... 47
FR1.5: OTP Code Generation and Delivery..... 47
FR1.6: Login..... 49
FR1.7: Logout..... 50
FR1.8: Reset Password..... 50
FR1.9: View Profile..... 50
Grocery Store (FR2)..... 50
FR2.9: Account Details..... 50
FR2.10: Search Product..... 52
FR2.11: Search Vendor..... 53
FR2.12: Browse Category..... 54
FR2.13: View Vendor Profile..... 55
FR2.14: Add Vendor to Vendor List..... 56
FR2.15: Contact Vendor..... 57
FR2.16: View Products on the Vendor's Profile..... 58
FR2.17: View Searched Product..... 59
FR2.18: Select Products..... 60
FR2.19: Order Recommendation..... 61
FR2.20: Quantity Selection..... 62
FR2.21: Remove Product..... 63
FR2.22: Order Placement..... 64
FR2.23: View Order..... 65
FR2.24: Order Tracking..... 66
FR2.25: View Vendor List..... 67
FR2.26: View Order History..... 68
FR2.27: Edit Profile..... 68
FR2.28: Add to Cart..... 68
FR2.29: View Cart..... 68
FR2.30: Update Product Quantity in Cart..... 69
FR2.30: Clear Cart..... 69
FR2.31: Search Category..... 69
Vendor (FR3)..... 70
FR3.9: Vendor Registration..... 70
FR3.10: Valid Password..... 70
FR3.11: Username Exists..... 70

FR3.12: Search Product in Inventory..... 71
FR3.13: Add Product to Inventory..... 72
FR3.14: All Products Search..... 73
FR3.15: Remove Product..... 74
FR3.16: Edit Details of Product..... 75
FR3.17: View Inventory..... 76
FR3.18: View Current Orders..... 77
FR3.19: View Grocery Stores List..... 78
FR3.20: View Grocery Store Profile..... 79
FR3.21: View Grocery Store Current Order..... 80
FR3.22: View Orders History..... 81
FR3.23: Order Dispatch Tracking..... 82
FR3.24: Edit Profile..... 83
FR3.25: Restrict Product Duplication..... 83
Admin Portal (FR4)..... 83
FR4.1: Add New User..... 83
FR4.2: Add New User Details..... 85
FR4.3: User Login..... 86
FR4.4: Edit User Profile..... 87
FR4.5: Delete User Profile..... 88
FR4.6: View Grocery Store's Profile..... 89
FR4.7: Disable Grocery Store's Profile..... 90
FR4.8: View Vendor's Profile..... 91
FR4.9: Disable Vendor's Profile..... 92
FR4.10: Select Grocery Store's ML Model..... 93
FR4.11: Display RMSLE value..... 94
FR4.12: Display Analytics..... 95
FR4.13: Add Category..... 96
FR4.14: Update Category..... 97
FR4.15: Delete Category..... 98
FR4.16: Add Product..... 99
FR4.17: Update Product..... 100
FR4.18: Delete Product..... 101
FR4.19: View Grocery Store Count..... 101
FR4.20: View Vendor Count..... 102
FR4.21: View Product Count..... 102
FR4.22: View Category Count..... 102
FR4.23: Search Admin..... 102
FR4.24: Search Grocery Store..... 103

FR4.25: Search Vendor..... 103
FR4.26: Search Product..... 103
FR4.27: Search Category.....103
Non-Functional Requirements..... 104
Future Improvements.....104
Design.....106
Development Tools.....108
Programming Languages.....108
Machine Learning..... 108
Cloud Services.....108
Web Development..... 108
Data Storage.....108
Object Relational Mapper.....108
Mobile App Development.....109
Version Control..... 109
Project Management..... 109
Collaboration.....109
System Architecture..... 109
System Context Diagram.....109
Container Diagram..... 110
Component Diagram.....111
Class Diagram.....111
Data Design..... 112
Sequence Diagrams..... 113
FR 2.19 Order Recommendation.....113
FR 2.24 Order Tracking.....114
FR 3.14 All Products Search.....115
FR 1.4 OTP Code Generation and Delivery.....116
FR 2.14 Add Vendor to Vendor List..... 117
API Design.....118
Machine Learning Design..... 119
Introduction..... 119
Feature Engineering.....120
Local Pharmacy Dataset..... 120
Corporación Favorita Grocery Sales Forecasting..... 122
Model Shortlisting.....126
Project Part I.....126
Random Forest.....126
XGBoost.....126

Prophet (Facebook)....127
RNN (LSTM/GRU)....127
Project Part II....127
LightGBM....127
N-BEATS....127
DeepAR (Amazon)....127
Neural Network Architecture....127
Deployment Design....128
Machine Learning....128
Conclusion....129
Implementation....130
App Development....132
Front End....132
Vendor App....132
Phase 1: Initial Front-End Design....132
Phase 2: App Development Progress....139
Phase 3: Front-End and Back-End Integration....143
Phase 4: Final Front-End Design....145
Grocery Store App....152
Phase 1: Initial Front-End Design....152
Phase 2: App Development Progress....157
Phase 3: Front-End and Back-End Integration....160
Phase 4: Final Front-End Design....163
Admin Portal....172
Phase 1: Initial Front-End Design....172
Phase 2: App Development Progress....176
Phase 3: Front-End and Back-End Integration....179
Phase 4: Final Front-End Design....182
Back End....193
Database Design....193
Spring Boot....195
Installation of Sequelize....196
User Model Implementation....198
Implementation of ENV....199
Vendor Model Implementation....201
Implementation of Controller....202
Implementation of Relations....205
Implementation of Routes....206
Implementation of Routes in App....207

Postman Testing....207
Conclusion....210
Test Cases....211
Test Case 1: Language Selection....211
Test Case 2: Phone Registration....211
Test Case 3: Phone Number Confirmation....211
Test Case 4: Phone Number Exists....212
Test Case 5: OTP Code Generation and Delivery....212
Test Case 6: Login....213
Test Case 7: Logout....214
Test Case 8: Reset Password....214
Test Case 9: View Profile....215
Test Case 10: Vendor Registration....215
Test Case 11: Valid Password....216
Test Case 12: Username Exists....216
Test Case 13: Search Product in Inventory....217
Test Case 14: Add Product to Inventory....217
Test Case 15: All Product Search....218
Test Case 16: Remove Product....218
Test Case 17: Edit Details of Product....219
Test Case 18: View Inventory....220
Test Case 19: View Current Orders....220
Test Case 20: View Grocery Store List....221
Test Case 21: View Grocery Store Profile....221
Test Case 22: View Grocery Store Current Order....221
Test Case 23: View Order History....222
Test Case 24: Edit Profile....222
Test Case 25: Restrict Product Duplication....223
Machine Learning....225
Introduction....225
Training and Testing....225
Local Pharmacy Dataset....225
Random Forest....225
XGBoost....230
Prophet....231
Recurrent Neural Networks....235
Summary....237
Corporación Favorita Grocery Sales Forecasting....238
Random Forest....242

XGBoost....246
Prophet.... 248
Recurrent Neural Networks.... 249
Summary....253
Deployment.... 253
Conclusion.... 262
References.... 263
Appendices.... 265
Appendix A: Interview Transcripts....265
Z Mart.... 265
Express Store....266

Introduction to the Project

Inefficiencies, wastage, and imprecise demand forecasting due to traditional inventory management processes have been an emerging challenge in the landscape of the food industry, specifically the supply chain system. There is a dire need for digitisation and automation within the industry, as evident by businesses' inability to streamline processes, predict demand accurately, and optimise resource usage. This deficiency hinders the possibility of an agile and responsive supply chain imperative to today's data-driven decision-making.

To tackle the aforementioned issue, our proposed solution employs advanced data analytics, specifically Machine Learning (ML) and Deep Learning (DL), to enhance the supply chain in grocery and vendor retail and address current supply chain vulnerabilities. This will be done through an accurately predicted customer demand, owing to a demand forecasting system, having utilised sales, holiday, location, and fuel prices.

The system will be integrated into a mobile-based platform connecting grocery stores with vendors to enable an efficient and profit-maximising operationality. An intuitive platform design such as ours would assist sales and purchase of goods by reducing physical visits made by vendors and enhancing operational efficiency through a tracking feature for grocery stores. Novice grocery stores would be assisted in searching for specific products from reliable registered vendors.

This platform stands out from Tajir, Dastgyr, Jugnu, Bazaar, and Retailo with the aid of a ML based demand forecasting system. This move toward modernisation and automation sets it apart in pursuing the same goal. As efficiency and accuracy arise in the inventory management, the precision by ML will allow stores to order optimal quantities, mitigating risks of overstocking and understocking.

This approach, inspired by the insights presented in 'Advances in Supply Chain Management' [1], aims to redefine forecasting, curb waste, and optimise operational efficiency for grocery stores and vendors. Ultimately, the solution would strive to augment overall profitability in the food industry and create a smarter supply chain system.

Existing Systems

In the current supply chain scenario in Pakistan, the movement of products relies heavily on outdated and paperwork-intensive processes. This traditional approach poses significant challenges, including vulnerability to disruptions, limited visibility due to infrastructure constraints, and communication hurdles. Retailers face stockouts, unpredictable supply arrivals, and a slow replenishment process, leading to operational inefficiencies and working capital restrictions. The conventional procurement structure demands substantial time investment, with retailers spending an average of 25 hours per week navigating wholesale markets [2].

A notable player addressing these challenges is Tajir, shown in Figure 1.1, it is a platform revolutionising inventory management for mom-and-pop stores in Pakistan. Tajir acts as a vendor, offering a seamless solution for purchasing inventory. It enables retailers to order at their convenience, receive on-demand deliveries, access transparent pricing, and choose from an extensive product selection. Inspired by successes like Tajir, our proposed software solution aims to complement and enhance the existing landscape by providing unique features tailored to the specific needs of small and medium retailers in Pakistan.

Figure 1.1 Tajir

Adding to this is Jugnu, a Business to Business (B2B) e-commerce platform founded in 2020. Jugnu, seen in Figure 1.2 strives for social and economic empowerment by connecting large suppliers to small and medium enterprises, driving growth in local economies. The platform operates across major cities in Pakistan, offering a user-friendly platform for ordering and receiving deliveries within a 24-hour window.

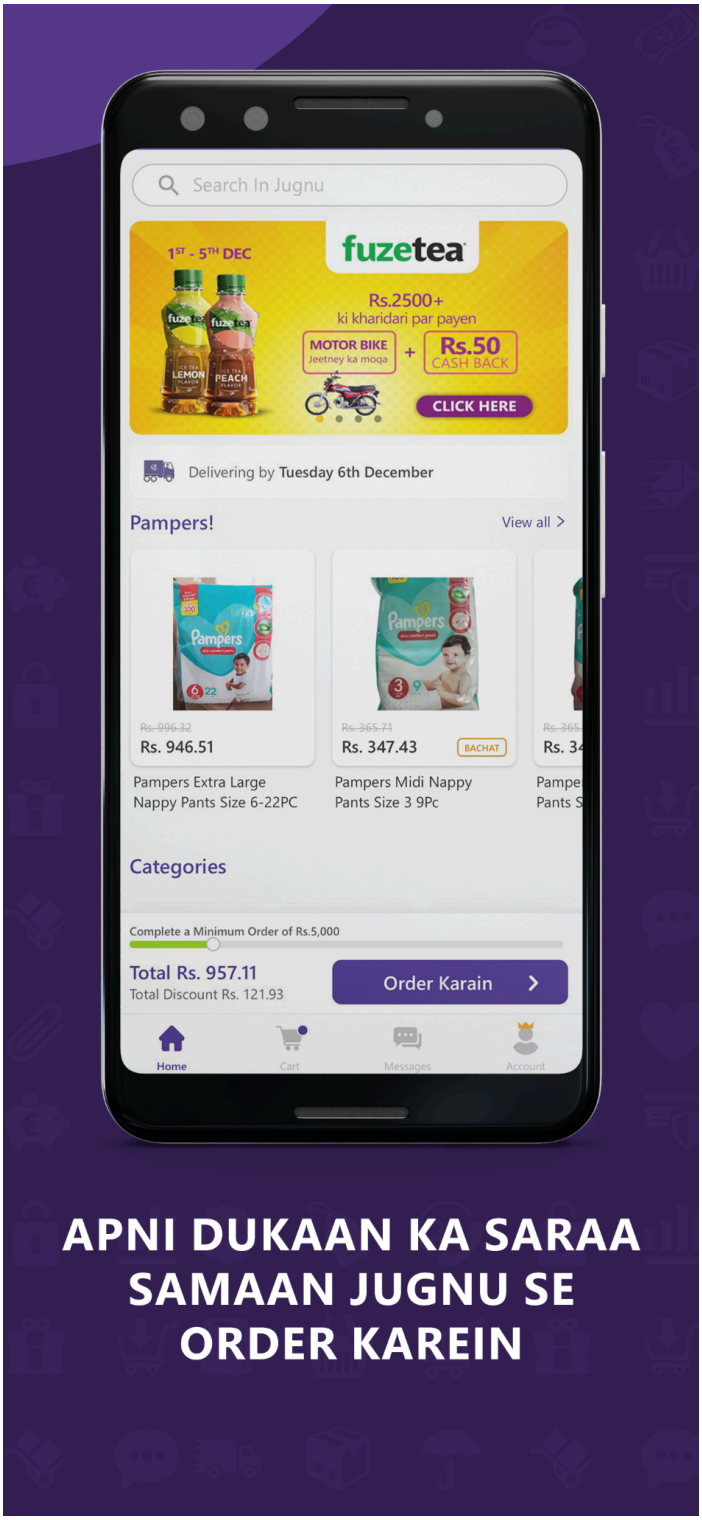


Figure 1.2 Jugnu

Below in Figure 1.3 is Retailo, another B2B marketplace, allows retailers to restock their shops conveniently with features like instant price comparisons and next-day delivery, eliminating the need for multiple distributors and weekly restocking hassles.

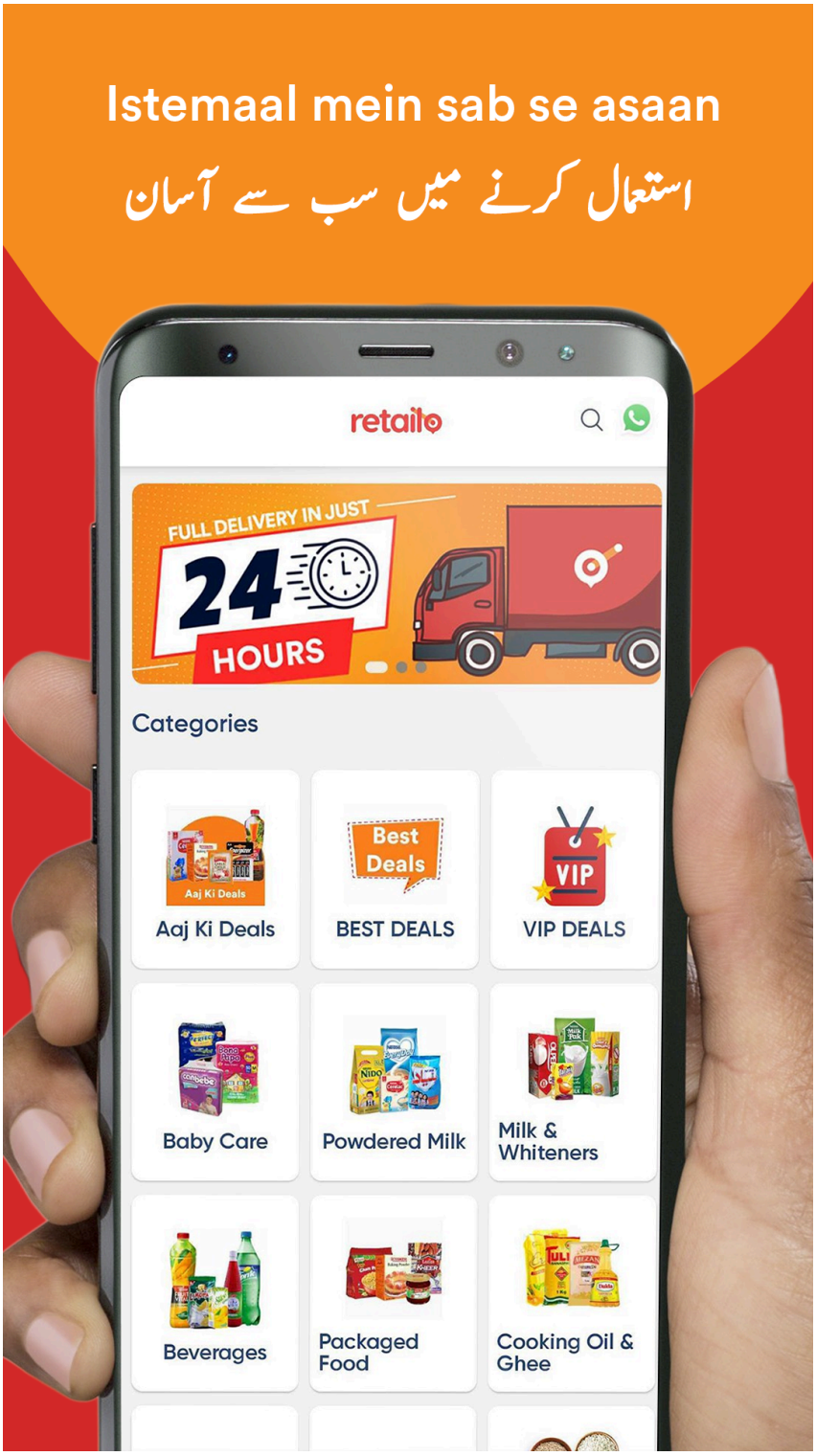


Figure 1.3 Retailo

Dastgyr, another B2B marketplace, addresses the pain points of small retailers by connecting them with manufacturers and suppliers. It offers a one-stop solution for inventory needs, allowing retailers to place

orders in seconds, see Figure 1.4, and receive on-demand delivery. Additionally, retailers working with Dastgyr for three months become eligible for credit purchases, addressing the challenge of limited access to external capital for growth.

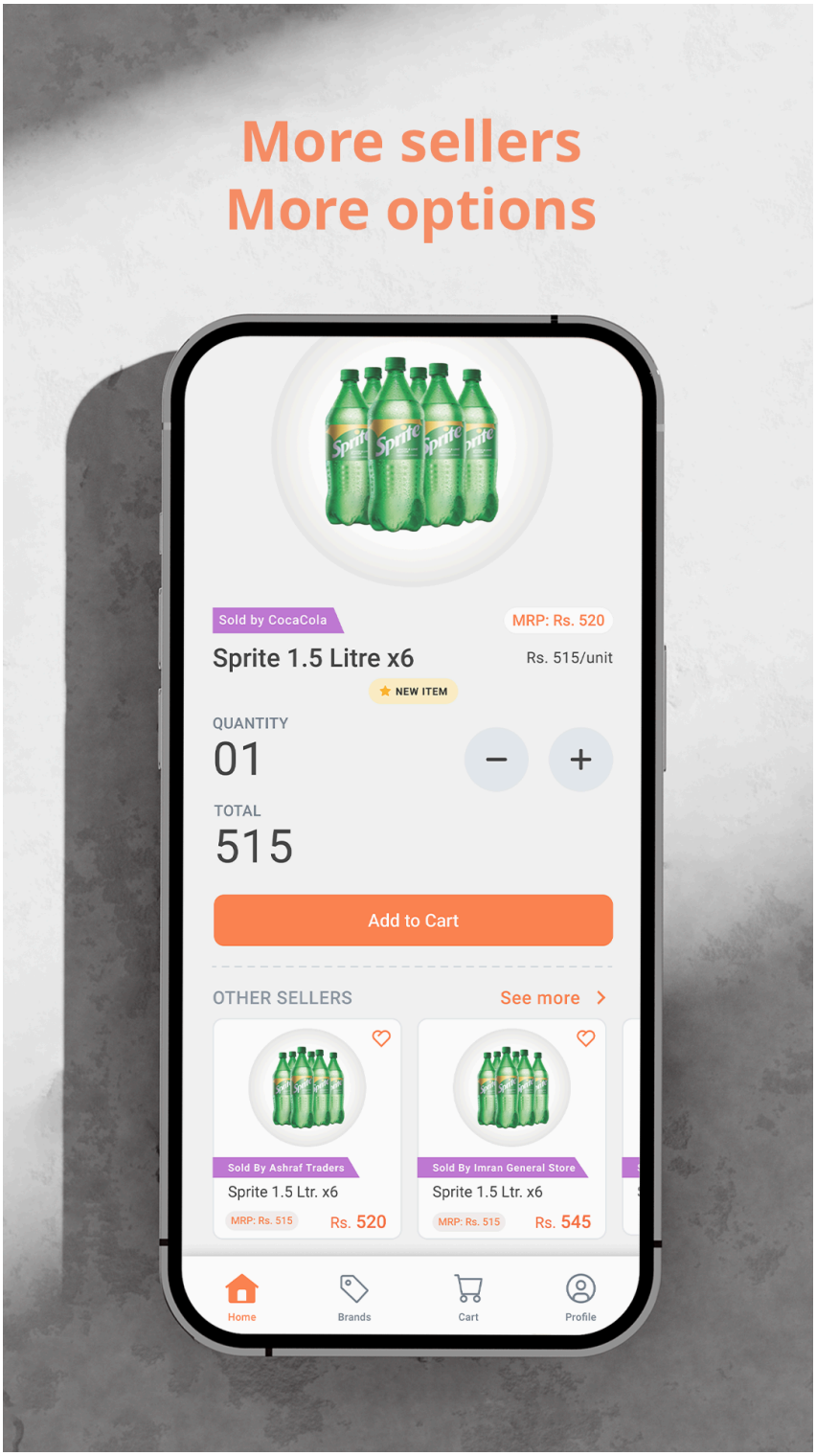


Figure 1.4 Dastgyr

Next is Bazaar in Figure 1.5 which contributes to digitising and growing businesses in Pakistan through its mobile app, providing small business owners access to a wide assortment of goods with free next-day delivery.

Figure 1.5 Bazaar

Additionally, we also explored Candela RMS, an enterprise retail software solution focusing on inventory management and POS for all kinds of retail. Candela RMS offers features like 'Manage Inventory Shrinkage' which is essentially a reorder level feature that sends an alert when an item in the inventory reaches a lower limit and 'Edibles Expiry Management' another feature that allows the entry of expiration dates during stock entry and subsequently printing the expiry date along with the barcode. While this feature aids in managing the expiration of products, it doesn't directly address the nuanced challenge of demand forecasting.

As we undertake this software solution project, we draw insights from the successes of these platforms, aiming to contribute to the modernisation and resilience of the supply chain ecosystem in Pakistan. Our market survey and research findings indicate a gap in the current solutions, with a need for a system that not only helps manage inventory but also accurately predicts demand and facilitates smooth communication between grocery stores and vendors. By incorporating elements from these innovative solutions, we seek to offer a comprehensive and effective software solution tailored to the specific challenges faced by small and medium grocery store retailers in the country.

Literature review

In addressing the multifaceted challenges of the food industry, recent studies have illuminated innovative solutions, offering insights that span crucial domains. One study explored the effectiveness of high-tech inventory management within supermarkets, attributing a significant 56.7% of their performance to automation [3]. Another research framework proposed the strategic implementation of AI and robotics to combat food loss during the pandemic, emphasising sensory enhancement and collaborative automation [4].

Recent advancements in deep learning, as highlighted in various studies, signal a transformative shift in the landscape of demand forecasting. Techniques such as multi-modal sales forecasting networks and the application of LSTM demonstrate superior accuracy and effectiveness [5, 6]. The debate between traditional and machine learning forecasting emerges, with a study showcasing the promise of a support vector machine in handling multiple demand series [7].

A groundbreaking study utilised low-cost sensors and machine learning to achieve a remarkable 92.65% accuracy in predicting for preventing food wastage, addressing a critical concern in the industry [8]. Global food supply chains are explored in another study, emphasising the importance of efficiency and behaviour change, particularly in affluent economies, to combat waste [9]. Strategies for proactive food waste reduction in the grocery sector are elucidated in a study that carefully balances customer satisfaction and inventory management [10].

The evolving role of e-grocery as an alternative to traditional retailing is highlighted, emphasising in-stock availability in customer decisions [11]. Generalised Additive Models for Location, Scale, and Shape (GAMLSS) are recommended in another exploration, specifically focusing on-demand distribution tails in e-grocery [12]. A simulation model dissects the benefits and drawbacks of Vendor-Managed Inventory (VMI) in the grocery supply chain, revealing that manufacturers reap more significant benefits from VMI adoption [13].

The chocolate industry is subject to a study employing machine learning for refined predictions based on regular and promotional sales data [14]. Retail firms, including Walmart, Costco, and Kroger, are analysed as economic indicators through statistical regression and machine learning, exposing operational inefficiencies [15]. The study on 'Corporacion Favorita,' a major grocery chain in Ecuador, offers insights into optimising predictions and mitigating stock-out and over-stocking issues [16].

In addition, abroad there are platforms like Shelf Engine shown in Figure 2.1 and Guac shown in Figure 2.2, both leverage machine learning for demand forecasting and order optimisation in the grocery retail sector. The major difference lies in their approach; Guac provides recommendations, leaving the final order decision to stores, while Shelf Engine actively decides orders and assists in placing them with vendors. Drawing inspiration from global industry trends, our proposed system aligns with the approach of Guac, aiming to leverage technology, embrace automation, and address current vulnerabilities in our supply chain. Despite notable progress in these studies and platforms, challenges such as training set

size, overfitting, and model complexity persist, necessitating further exploration for the development of a responsive and sustainable food system.

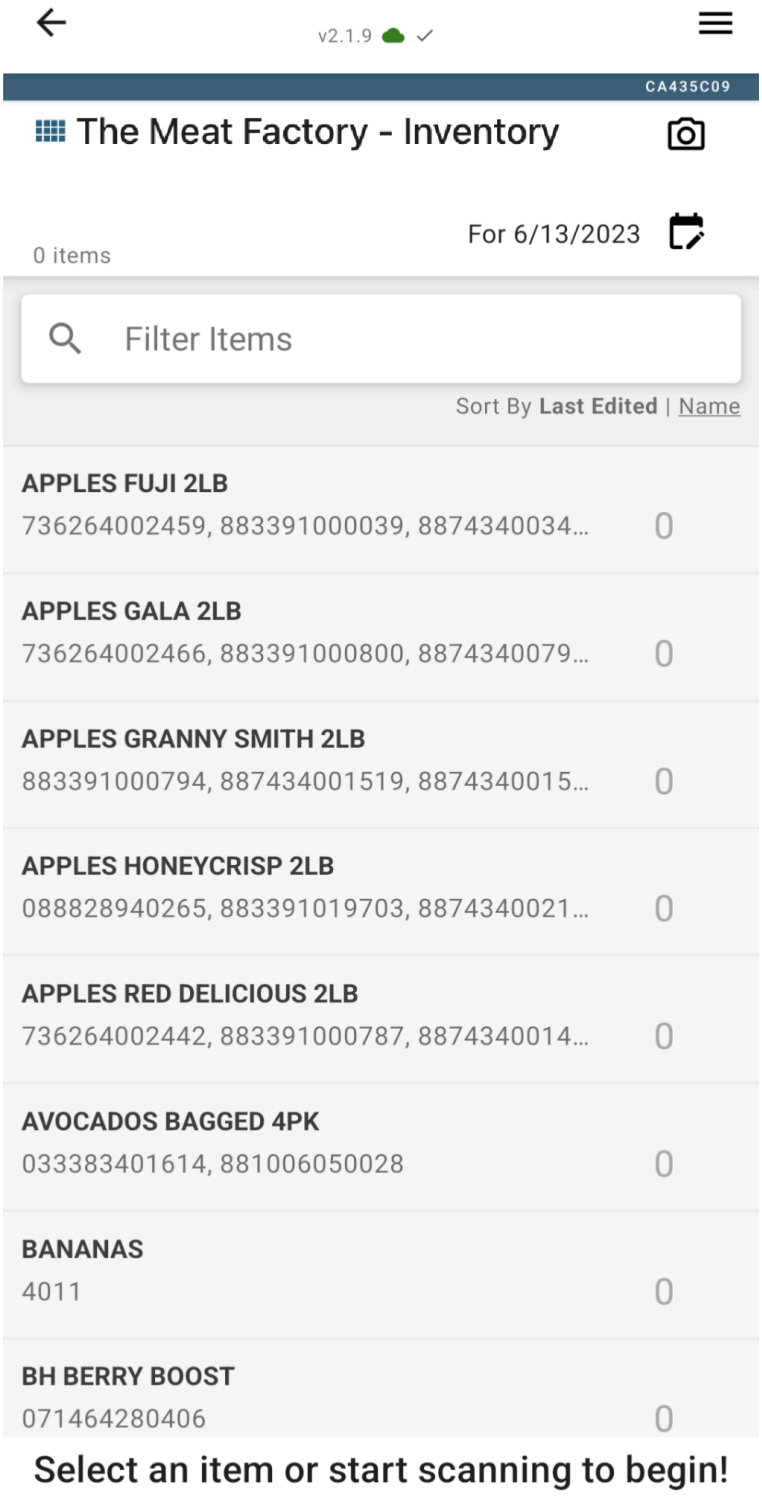


Figure 2.1 Shelf Engine

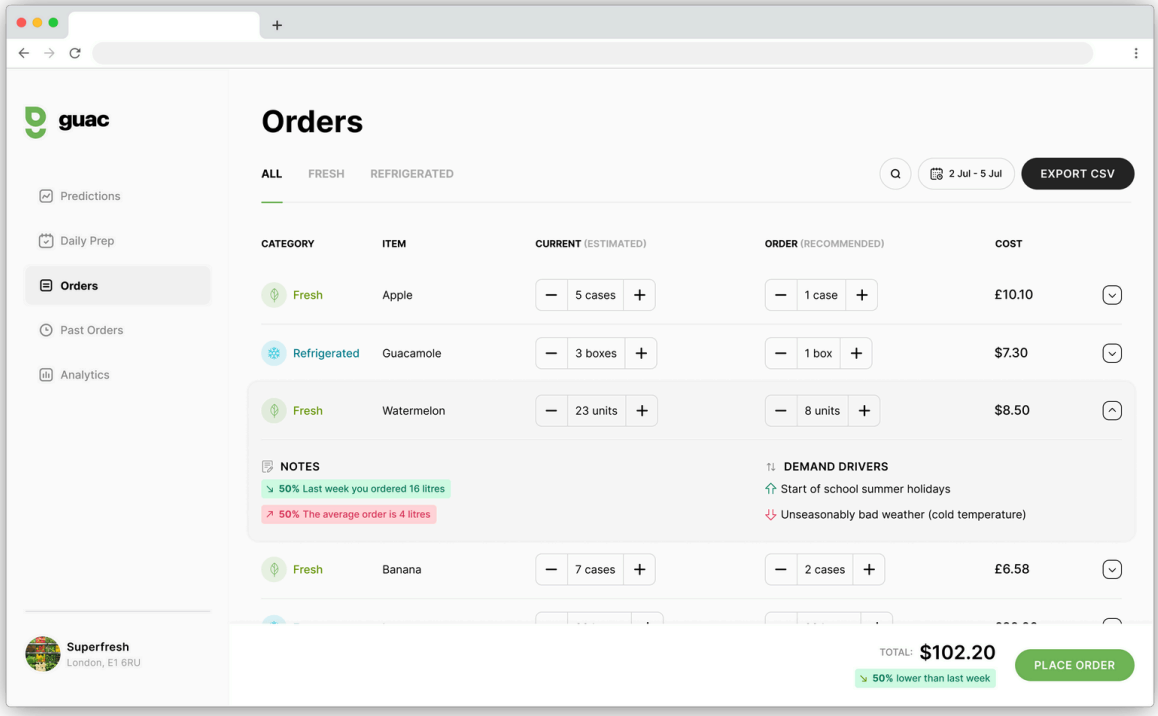


Figure 2.2 Guac

Requirement Analysis

The requirement analysis document presented here outlines a comprehensive approach taken into writing down the important requirements and gathering the necessary data for the development of a demand forecasting system for grocery stores, the mobile applications for both the vendor and the grocery store, and the admin portal. The document first begins with the requirement gathering and fact finding section which will lay the foundation for understanding the needs and expectations of users and stakeholders. Then for the machine learning model, we will outline the process for data collection, followed by feature engineering then shortlisting potential models, training and testing and then the integration with user interface. The requirements for the two mobile applications (Vendor and Grocery Store Application) will be elaborated next, focusing on user interface design, functionality, and platform-specific considerations. Furthermore, the admin portal requirements will be specified, highlighting the necessary features for user management, and product management. Finally, we will address the non-functional requirements applicable to the machine learning model, mobile applications, and admin portal. These will include performance, scalability, security, and usability considerations to ensure that the system delivers a seamless and secure user experience.

Requirement Gathering and Fact Finding

We engaged grocery stores through unstructured interviews (transcripts in Appendix) and navigating the stores to discuss concepts like strategic shelf placement for marketing and promotion.

Data Collection Technique: Interviews and Observation

> **Reasoning:** Interviews provide a strategic and adaptable approach, capturing detailed, context-specific information essential for understanding the challenges faced by grocery store owners in the supply chain.

Issues Identified

Amongst the challenges found, an outdated ordering system is the biggest one. This manifests in discounted products nearing expiration made available at stores as seen in Figure 3.1, which raises concerns over the quality and safe use, and monopolisation of certain products where grocery stores do not allow specific products to be sold by vendors to any other grocery store than them e.g. a grocery store in a housing society tells the vendor that they will only purchase a specific product if the vendor does not sell it to any other store in that housing society. In-person interviews highlighted the impact of these issues, emphasising that existing tools (inventory management systems and point of sale systems) are rendered ineffective due to their complexity. Even grocery store workers who are tech-savvy try to avoid inventory-related tasks, such as adding received inventory. They only use it for selling as it makes the process of selling faster and more convenient for them. Notably, the grocery store workforce is not technologically inept; rather, the current inventory systems are user-unfriendly. There is also an issue of ‘mobilers’, a community term for individuals that are fake/pretend vendors selling either fake or expired products.

Figure 3.1 Jalal Son's

Summarised

An outdated ordering system that has consequences such as:

- Existing tools are too difficult to use for grocery stores
- 'Mobilers' selling fake or expired products
- Limited vendor options for new products discoverability
- Product Monopolisation

Interview Insights

Insights into the ordering process included manual observation of daily sales i.e. qualitative analysis by the grocery store owner to decide the quantity for the next order, with budget constraints such as inflation leading to reduced stock purchases. Some grocery stores do not have access to specific products by a shared vendor on demand of another grocery store.

On the digital frontier, software challenges include a dependency on electricity, a lack of backup during power outages, and manual data entry during disruptions. The workforce's adeptness with technology clashes with the current software's complexity, leading to underutilization.

Summarised

1. Ordering Process:
    - Qualitative analysis to determine order quantities
    - Budget constraints impact stock purchasing decisions
    - Limited access to specific products due to shared vendor restrictions (grocery stores restricting vendors to only sell a specific product to them)

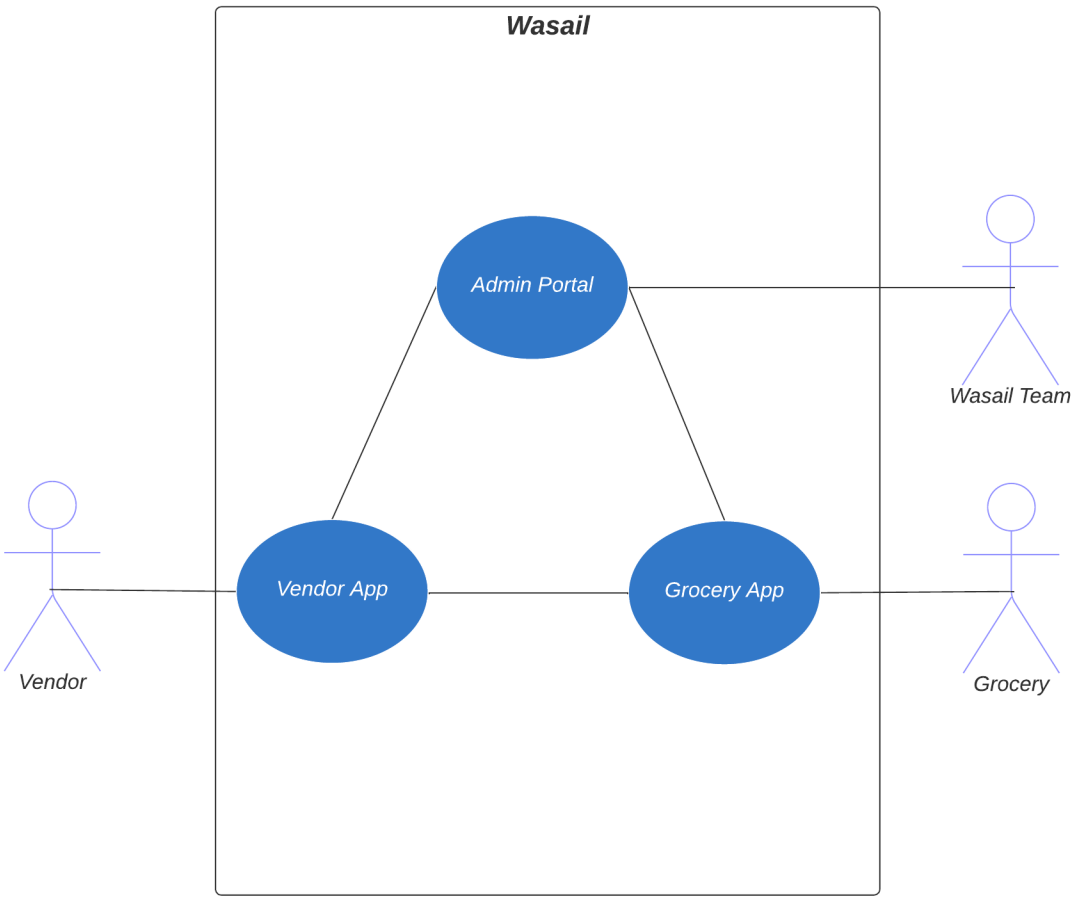
2. Software Issues:
    - Dependency on electricity for software usage
    - Lack of backup in case of power outages
    - Manual data entry during power disruptions

Conclusion: The insights gathered through interviews with grocery stores highlight critical issues in the current supply chain, forming a crucial foundation for the development of our software solution. The experiences shared by Z Mart, Express Store, and My Store underscore the necessity for a digital solution to tackle challenges related to order management, inventory control, and system reliability. Furthermore, the additional research reveals significant gaps in the existing mechanisms, including the absence of a formal communication channel with vendors, authentication issues, and a reliance on your vendors as the sole source for new products.

Potential Improvements & Proposed Solution

- **Demand Forecasting and Automation:** Develop a sophisticated system for demand-forecasting, integrated in a platform managing orders and distribution, supported by machine learning and use variables like sales trends, weather, location, and holidays, optimising the inventory management.
- **Waste Reduction and Profitability Augmentation:** Redefine forecasting to curb food wastage whilst optimising operational efficiency for both grocery stores and vendors. As sales increase so does profitability.
- **Streamlining Processes:** Introduce an easier-to-use system for a seamless user experience.
- **Increased Visibility:** Enhance vendor lookup and search functionalities.
- **Collaborative Platform:** Verify the legitimacy of vendors for a secure collaborative environment and reduce physical visits to grocery stores by vendors through the platform.

System Environment



Machine Learning Requirements

Objective

Provide a recommended amount of product to order for the grocery stores.

Use Case

“As a grocery store, I want to receive a recommended amount of product to order, based on my previous sales, that would ensure maximum profit.”

Data Collection

Local Vendor Dataset

Attributes:

1. Item ID
2. Item Description
3. Date

4. Qty Received
5. Item Cost
6. Actual Cost
7. Assembly Qty
8. Assembly ($)
9. Adjust Qty
10. Adjust ($)
11. Quantity Sold
12. Cost of Sales
13. Remaining Qty
14. Remain Value

Instances: 16,000

Format: CSV

Time Frame: October 2019 to December 2021

Description: The data, seen in Figure 4.1, needs to be cleaned. Assembly Qty, Assembly ($), Adjust Qty, and Adjust ($) need to be removed as they are mostly empty. Instances where a product has been purchased (received), the attributes related to sales are empty and vice versa. Hence, the data needs to be separated into two different datasets, one for purchases and one for sales, to get rid of all the empty cells. Analysis of this dataset aids to better understanding of how vendors sell and grocery store purchase products. It can also be used in the future to offer demand forecasting for vendors as well. This dataset is uploaded on our GitHub repository.

Item ID	Item Description	Date	Qty Received	Item Cost	Actual Cost	Assembly Qty	Assembly ($)	Adjust Qty	Adjust ($)	Quantity Sold	Cost of Sales	Remaining Qty	Remain Value
AAP001	AGAR AGAR POWDER 25G												
AAP001	AGAR AGAR POWDER 25G	1/27/20	100.00	297.44	29,744.00							100.00	29,744.00
AAP001	AGAR AGAR POWDER 25G	1/29/20								100.00	29,744.00		
ABC01	ABC SWEET SOY SAUCE 500ML												
ABC01	ABC SWEET SOY SAUCE 500ML	7/24/20	96.00	721.18	69,233.28							96.00	69,233.28
ABC01	ABC SWEET SOY SAUCE 500ML	7/25/20								24.00	17,308.32	72.00	51,924.96
ABC01	ABC SWEET SOY SAUCE 500ML	7/25/20								72.00	51,924.96		
ABC01	ABC SWEET SOY SAUCE 500ML	8/7/20	120.00	710.25	85,230.00							120.00	85,230.00
ABC01	ABC SWEET SOY SAUCE 500ML	8/8/20								36.00	25,569.00	84.00	59,661.00
ABC01	ABC SWEET SOY SAUCE 500ML	8/11/20								84.00	59,661.00		
ABC01	ABC SWEET SOY SAUCE 500ML	8/23/20	72.00	672.60	48,427.20							72.00	48,427.20
ABC01	ABC SWEET SOY SAUCE 500ML	8/25/20								72.00	48,427.20		
ABC01	ABC SWEET SOY SAUCE 500ML	9/7/20	72.00	672.60	48,427.20							72.00	48,427.20
ABC01	ABC SWEET SOY SAUCE 500ML	9/12/20								48.00	32,284.80	24.00	16,142.40
ABC01	ABC SWEET SOY SAUCE 500ML	9/27/20	288.00	714.30	205,718.40							312.00	221,860.80
ABC01	ABC SWEET SOY SAUCE 500ML	9/30/20								60.00	41,857.20	252.00	180,003.60
ABC01	ABC SWEET SOY SAUCE 500ML	11/3/20								60.00	42,858.00	192.00	137,145.60
ABC01	ABC SWEET SOY SAUCE 500ML	11/21/20								120.00	85,716.00	72.00	51,429.60
ABC01	ABC SWEET SOY SAUCE 500ML	11/24/20								68.00	48,572.40	4.00	2,857.20
ABC01	ABC SWEET SOY SAUCE 500ML	1/31/21								4.00	2,857.20		
AC01	AMERI COLOR												
ACC01	ANCHOR CHEDDAR KG												
ACTC01	ANTICA CANTINA SALTED 200G												
ACTC01	ANTICA CANTINA SALTED 200G	8/29/20								29.00	18,880.74	-29.00	-18,880.74
ACTC01	ANTICA CANTINA SALTED 200G	8/29/20	30.00	651.06	19,531.80							1.00	651.06
ACTC02	ANTICA CANTINA CHILLI 200G												
ACTC02	ANTICA CANTINA CHILLI 200G	8/29/20								30.00	19,531.80	-30.00	-19,531.80
ACTC02	ANTICA CANTINA CHILLI 200G	8/29/20	30.00	651.06	19,531.80								
ACTC03	ANTICA CANTINA N.CHEESE 200G												
ACTC03	ANTICA CANTINA N.CHEESE 200G	8/29/20								30.00	19,531.80	-30.00	-19,531.80
ACTC03	ANTICA CANTINA N.CHEESE 200G	8/29/20	30.00	651.06	19,531.80								
ACTC04	ANTICA CANTINA BBQ 200G												
ACTC04	ANTICA CANTINA BBQ 200G	8/29/20								30.00	19,531.80	-30.00	-19,531.80
ACTC04	ANTICA CANTINA BBQ 200G	8/29/20	30.00	651.06	19,531.80								
ACTD01	ANTICA CANTINA MILD SALSA 300G												
ACTD01	ANTICA CANTINA MILD SALSA 300G	8/29/20								12.00	10,982.28	-12.00	-10,982.28
ACTD01	ANTICA CANTINA MILD SALSA 300G	8/29/20	12.00	915.19	10,982.28								
ACTD02	ANTICA CANTINA MED SALSA 300G												
ACTD02	ANTICA CANTINA MED SALSA 300G	8/29/20								12.00	10,982.28	-12.00	-10,982.28
ACTD02	ANTICA CANTINA MED SALSA 300G	8/29/20	12.00	915.19	10,982.28								

Figure 4.1 Local Vendor Dataset

Local Pharmacy Dataset

Attributes:

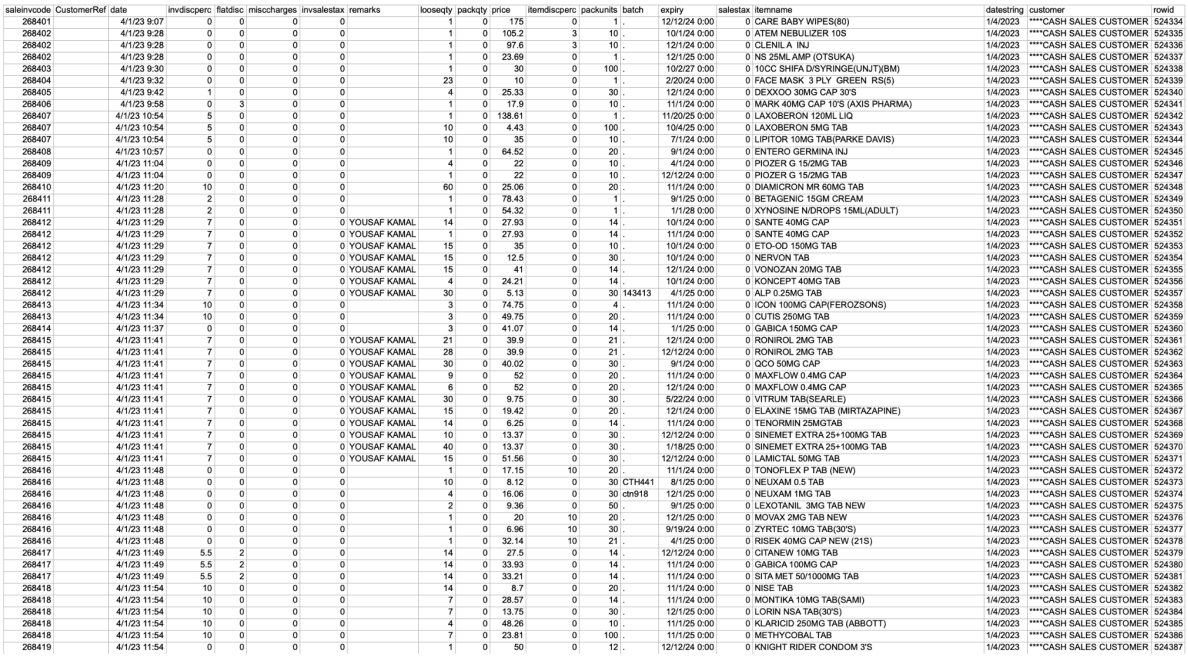
1. saleinvcode
2. customerref
3. date
4. invdiscperc
5. flatdisc
6. misccharges
7. invsalestax
8. remarks
9. looseqty
10. packqty
11. price
12. itemdiscperc
13. packunits
14. batch
15. expiry
16. salestax
17. itemname
18. datestring
19. customer
20. rowid

Instances: 300,000

Format: XLS

Time Frame: July 2022 to June 2023

Description: The data, seen in Figure 4.2, needs to be cleaned. saleinvcode, customerref, invdiscperc, flatdisc, misccharges, invsalestax, packqty, itemdiscperc, batch, salestax, datestring customer, and rowid need to be removed as they are either empty, zero, same for every entry, or irrelevant to demand forecasting. The remarks attribute is used to enter the names of customers, therefore, it should be removed to ensure privacy. The date attribute contains the date and the time, it should be split into two parts, date and time. The sales are only recorded in terms of looseqty. The dataset also contains expiry dates which will enable us to analyse the perishable aspect of products. This dataset is uploaded on our GitHub repository.



saleinvcode	CustomerRef	date	invdiscperc	flatdisc	misccharges	invsalestax	remarks	looseqty	packqty	price	itemdiscperc	packunits	batch	expiry	salestax	itemname	datestring	customer	rowid
268401		4/1/23 9:07	0	0	0	0		1	0	175	0	1	.	12/12/24 0:00	0	CARE BABY WIPES(80)	1/4/2023	****CASH SALES CUSTOMER	524334
268402		4/1/23 9:28	0	0	0	0		1	0	105.2	3	10	.	10/1/24 0:00	0	ATEM NEBULIZER 10S	1/4/2023	****CASH SALES CUSTOMER	524335
268402		4/1/23 9:28	0	0	0	0		1	0	97.6	3	10	.	12/1/24 0:00	0	CLENIL A INJ	1/4/2023	****CASH SALES CUSTOMER	524336
268402		4/1/23 9:28	0	0	0	0		1	0	23.69	0	1	.	12/1/25 0:00	0	NS 25ML AMP (OTSUKA)	1/4/2023	****CASH SALES CUSTOMER	524337
268403		4/1/23 9:30	0	0	0	0		1	0	30	0	100	.	10/2/27 0:00	0	10CC SHIFA D/SYRINGE(UNJT)(BM)	1/4/2023	****CASH SALES CUSTOMER	524338
268404		4/1/23 9:32	0	0	0	0		23	0	10	0	1	.	2/20/24 0:00	0	FACE MASK 3 PLY GREEN RS(5)	1/4/2023	****CASH SALES CUSTOMER	524339
268405		4/1/23 9:42	1	0	0	0		4	0	25.33	0	30	.	12/1/24 0:00	0	DEXXOO 30MG CAP 30'S	1/4/2023	****CASH SALES CUSTOMER	524340
268406		4/1/23 9:58	0	3	0	0		1	0	17.9	0	10	.	11/1/24 0:00	0	MARK 40MG CAP 10'S (AXIS PHARMA)	1/4/2023	****CASH SALES CUSTOMER	524341
268407		4/1/23 10:54	5	0	0	0		1	0	138.61	0	1	.	11/20/25 0:00	0	LAXOBERON 120ML LIQ	1/4/2023	****CASH SALES CUSTOMER	524342
268407		4/1/23 10:54	5	0	0	0		10	0	4.43	0	100	.	10/4/25 0:00	0	LAXOBERON 5MG TAB	1/4/2023	****CASH SALES CUSTOMER	524343
268407		4/1/23 10:54	5	0	0	0		10	0	35	0	10	.	7/1/24 0:00	0	LIPITOR 10MG TAB(PARKE DAVIS)	1/4/2023	****CASH SALES CUSTOMER	524344
268408		4/1/23 10:57	0	0	0	0		1	0	64.52	0	20	.	9/1/24 0:00	0	ENTERO GERMINA INJ	1/4/2023	****CASH SALES CUSTOMER	524345
268409		4/1/23 11:04	0	0	0	0		4	0	22	0	10	.	4/1/24 0:00	0	PIOZER G 15/2MG TAB	1/4/2023	****CASH SALES CUSTOMER	524346
268409		4/1/23 11:04	0	0	0	0		1	0	22	0	10	.	12/12/24 0:00	0	PIOZER G 15/2MG TAB	1/4/2023	****CASH SALES CUSTOMER	524347
268410		4/1/23 11:20	10	0	0	0		60	0	25.06	0	20	.	11/1/24 0:00	0	DIAMICRON MR 60MG TAB	1/4/2023	****CASH SALES CUSTOMER	524348
268411		4/1/23 11:28	2	0	0	0		1	0	78.43	0	1	.	9/1/25 0:00	0	BETAGENIC 15GM CREAM	1/4/2023	****CASH SALES CUSTOMER	524349
268411		4/1/23 11:28	2	0	0	0		1	0	54.32	0	1	.	1/1/28 0:00	0	XYNOSINE N/DROPS 15ML(ADULT)	1/4/2023	****CASH SALES CUSTOMER	524350
268412		4/1/23 11:29	7	0	0	0	YOUSAF KAMAL	14	0	27.93	0	14	.	10/1/24 0:00	0	SANTE 40MG CAP	1/4/2023	****CASH SALES CUSTOMER	524351
268412		4/1/23 11:29	7	0	0	0	YOUSAF KAMAL	1	0	27.93	0	14	.	11/1/24 0:00	0	SANTE 40MG CAP	1/4/2023	****CASH SALES CUSTOMER	524352
268412		4/1/23 11:29	7	0	0	0	YOUSAF KAMAL	15	0	35	0	10	.	10/1/24 0:00	0	ETO-OD 150MG TAB	1/4/2023	****CASH SALES CUSTOMER	524353
268412		4/1/23 11:29	7	0	0	0	YOUSAF KAMAL	15	0	12.5	0	30	.	10/1/24 0:00	0	NERVON TAB	1/4/2023	****CASH SALES CUSTOMER	524354
268412		4/1/23 11:29	7	0	0	0	YOUSAF KAMAL	15	0	41	0	14	.	12/1/24 0:00	0	VONOZAN 20MG TAB	1/4/2023	****CASH SALES CUSTOMER	524355
268412		4/1/23 11:29	7	0	0	0	YOUSAF KAMAL	4	0	24.21	0	14	.	10/1/24 0:00	0	KONCEPT 40MG TAB	1/4/2023	****CASH SALES CUSTOMER	524356
268412		4/1/23 11:29	7	0	0	0	YOUSAF KAMAL	30	0	5.13	0	30	143413	4/1/25 0:00	0	ALP 0.25MG TAB	1/4/2023	****CASH SALES CUSTOMER	524357
268413		4/1/23 11:34	10	0	0	0		3	0	74.75	0	4	.	11/1/24 0:00	0	ICON 100MG CAP(FEROZSONS)	1/4/2023	****CASH SALES CUSTOMER	524358
268413		4/1/23 11:34	10	0	0	0		3	0	49.75	0	20	.	11/1/24 0:00	0	CUTIS 250MG TAB	1/4/2023	****CASH SALES CUSTOMER	524359
268414		4/1/23 11:37	0	0	0	0		3	0	41.07	0	14	.	1/1/25 0:00	0	GABICA 150MG CAP	1/4/2023	****CASH SALES CUSTOMER	524360
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	21	0	39.9	0	21	.	12/1/24 0:00	0	RONIROL 2MG TAB	1/4/2023	****CASH SALES CUSTOMER	524361
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	28	0	39.9	0	21	.	12/12/24 0:00	0	RONIROL 2MG TAB	1/4/2023	****CASH SALES CUSTOMER	524362
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	30	0	40.02	0	30	.	9/1/24 0:00	0	QCO 50MG CAP	1/4/2023	****CASH SALES CUSTOMER	524363
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	9	0	52	0	20	.	11/1/24 0:00	0	MAXFLOW 0.4MG CAP	1/4/2023	****CASH SALES CUSTOMER	524364
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	6	0	52	0	20	.	12/1/24 0:00	0	MAXFLOW 0.4MG CAP	1/4/2023	****CASH SALES CUSTOMER	524365
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	30	0	9.75	0	30	.	5/22/24 0:00	0	VITRUM TAB(SEARLE)	1/4/2023	****CASH SALES CUSTOMER	524366
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	15	0	19.42	0	20	.	12/1/24 0:00	0	ELAXINE 15MG TAB (MIRTAZAPINE)	1/4/2023	****CASH SALES CUSTOMER	524367
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	14	0	6.25	0	14	.	11/1/24 0:00	0	TENORMIN 25MGTAB	1/4/2023	****CASH SALES CUSTOMER	524368
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	10	0	13.37	0	30	.	12/12/24 0:00	0	SINEMET EXTRA 25+100MG TAB	1/4/2023	****CASH SALES CUSTOMER	524369
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	40	0	13.37	0	30	.	1/18/25 0:00	0	SINEMET EXTRA 25+100MG TAB	1/4/2023	****CASH SALES CUSTOMER	524370
268415		4/1/23 11:41	7	0	0	0	YOUSAF KAMAL	15	0	51.56	0	30	.	12/12/24 0:00	0	LAMICTAL 50MG TAB	1/4/2023	****CASH SALES CUSTOMER	524371
268416		4/1/23 11:48	0	0	0	0		1	0	17.15	10	20	.	11/1/24 0:00	0	TONOFLEX P TAB (NEW)	1/4/2023	****CASH SALES CUSTOMER	524372
268416		4/1/23 11:48	0	0	0	0		10	0	8.12	0	30	CTH441	8/1/25 0:00	0	NEUXAM 0.5 TAB	1/4/2023	****CASH SALES CUSTOMER	524373
268416		4/1/23 11:48	0	0	0	0		4	0	16.06	0	30	ctn918	12/1/25 0:00	0	NEUXAM 1MG TAB	1/4/2023	****CASH SALES CUSTOMER	524374
268416		4/1/23 11:48	0	0	0	0		2	0	9.36	0	50	.	9/1/25 0:00	0	LEXOTANIL 3MG TAB NEW	1/4/2023	****CASH SALES CUSTOMER	524375
268416		4/1/23 11:48	0	0	0	0		1	0	20	10	20	.	12/1/25 0:00	0	MOVAX 2MG TAB NEW	1/4/2023	****CASH SALES CUSTOMER	524376
268416		4/1/23 11:48	0	0	0	0		1	0	6.96	10	30	.	9/19/24 0:00	0	ZYRTEC 10MG TAB(30'S)	1/4/2023	****CASH SALES CUSTOMER	524377
268416		4/1/23 11:48	0	0	0	0		1	0	32.14	10	21	.	4/1/25 0:00	0	RISEK 40MG CAP NEW (21S)	1/4/2023	****CASH SALES CUSTOMER	524378
268417		4/1/23 11:49	5.5	2	0	0		14	0	27.5	0	14	.	12/12/24 0:00	0	CITANEW 10MG TAB	1/4/2023	****CASH SALES CUSTOMER	524379
268417		4/1/23 11:49	5.5	2	0	0		14	0	33.93	0	14	.	11/1/24 0:00	0	GABICA 100MG CAP	1/4/2023	****CASH SALES CUSTOMER	524380
268417		4/1/23 11:49	5.5	2	0	0		14	0	33.21	0	14	.	11/1/24 0:00	0	SITA MET 50/1000MG TAB	1/4/2023	****CASH SALES CUSTOMER	524381
268418		4/1/23 11:54	10	0	0	0		14	0	8.7	0	20	.	11/1/24 0:00	0	NISE TAB	1/4/2023	****CASH SALES CUSTOMER	524382
268418		4/1/23 11:54	10	0	0	0		7	0	28.57	0	14	.	11/1/24 0:00	0	MONTIKA 10MG TAB(SAMI)	1/4/2023	****CASH SALES CUSTOMER	524383
268418		4/1/23 11:54	10	0	0	0		7	0	13.75	0	30	.	12/1/25 0:00	0	LORIN NSA TAB(30'S)	1/4/2023	****CASH SALES CUSTOMER	524384
268418		4/1/23 11:54	10	0	0	0		4	0	48.26	0	10	.	11/1/25 0:00	0	KLARICID 250MG TAB (ABBOTT)	1/4/2023	****CASH SALES CUSTOMER	524385
268418		4/1/23 11:54	10	0	0	0		7	0	23.81	0	100	.	11/1/25 0:00	0	METHYCOBAL TAB	1/4/2023	****CASH SALES CUSTOMER	524386
268419		4/1/23 11:54	0	0	0	0		1	0	50	0	12	.	12/12/24 0:00	0	KNIGHT RIDER CONDOM 3'S	1/4/2023	****CASH SALES CUSTOMER	524387

Figure 4.2 Local Pharmacy Dataset

[Corporación Favorita Grocery Sales Forecasting](#) (Kaggle)

Datasets:

Training Data

1. Date
2. On Promotion
3. Unit Sales
4. Store Number
5. Item Number

Stores

1. City
2. State
3. Type
4. Cluster

Items

1. Class
2. Perishable
3. Family

Transactions

1. Date
2. Transactions

Oil

1. Date
2. Oil Price

Holidays

1. Type
2. Locale

Instances: 126 Million
Size: 4.7 GB (Training Data)
Format: CSV
Time Frame: January 2013 to August 2017

Description: This is a dataset from Corporación Favorita Grocery Sales Forecasting competition hosted on Kaggle 6 years ago. The dataset contains millions of instances, spanning over 5 years, including key variables such as holidays, oil prices, and location. The dataset is considered credible based on the fact it's shared by a large grocery store chain, hosted as a competition on Kaggle with prizes of $30,000 and 1500+ participants, and most importantly is used for demand forecasting in several research papers including [16] and [25].

Current Work: Even though the Corporación Favorita Grocery Sales Forecasting competition ended on Jan 16, 2018, Kaggle created a new competition Store Sales - Time Series Forecasting with the same data which runs indefinitely with a rolling leaderboard (we intend to submit predictions on this data of our final ML model in the competition). This has proven to be a very helpful resource as we have been able to access recent works of people using the latest advancements in neural networks for time series analysis.

Instacart Market Basket Analysis (Kaggle)

Datasets:
Aisles

1. aisle_id (1, 2, 3)
2. aisle (prepared soups salads, specialty cheeses, energy granola bars)

Departments

1. department_id (1, 2, 3)
2. department (frozen, other, bakery)

Prior Product Orders

1. order_id (1, 1, 1)
2. product_id (49302, 11109, 10246)
3. add_to_cart_order (1, 2, 3)
4. reordered (1, 1, 0)

Orders

1. order_id (2539329, 2398795, 473747)
2. user_id (1, 1, 1)
3. eval_set (prior, prior, prior)
4. order_number (1, 2, 3)
5. order_dow (2, 3, 3)

6. order_hour_of_day (08, 07, 12)
7. days_since_prior_order (NA, 15.0, 21.0)

Products

1. product_id (1, 2, 3)
2. product_name (Chocolate Sandwich Cookies, All-Seasons Salt, Robust Golden Unsweetened Oolong Tea)
3. aisle_id (61, 104, 94)
4. department_id (19, 13, 7)

Instances: 3.4 Million
Format: CSV
Description: This is a dataset from the Instacart Market Basket Analysis competition hosted on Kaggle 6 years ago. The dataset contains millions of instances, including key variables such as aisle, department, day of week, and hour of day. The dataset is considered credible based on the fact it's shared by a large grocery delivery company, hosted as a competition on Kaggle with prizes of $25,000 and 2500+ participants.

Other Datasets for Further Exploration

- Grupo Bimbo Inventory Demand
- Store Item Demand Forecasting Challenge

Feature Engineering

Date

The date provides a time series of sales data, enabling the model to identify and capture seasonal patterns, trends, and recurring events that influence demand. For instance, it can recognize increased sales during holidays, weekends, or specific times of the year. For perishable products, knowing the date of sales is crucial for managing inventory effectively, ensuring that products are sold before they reach their expiration dates. [11] explains how the variable date can be further divided into three numeric attributes which are the day of the month, the month, and the year to maintain the important weekly, monthly and yearly seasonal information.

Product Name

The product name allows for the categorization of items into specific product types or categories. This categorization is essential for understanding and forecasting demand patterns within different product groups. Different products may exhibit varying levels of demand volatility. By considering the product name, the forecasting model can account for the unique demand characteristics of each item, whether it's a fast-moving consumer good or a slow-moving, seasonal product.The product name is critical for inventory management. It enables the model to forecast demand for each product individually, allowing businesses to optimise stock levels, reduce overstocking or understocking issues, and minimise the risk of waste. Certain products may experience fluctuations in demand based on seasons or trends. The

product name allows the model to identify and capture these variations, helping in accurate demand forecasting.

Sales

Sales revenue directly reflects the monetary value of products sold. By analysing historical sales revenue data, the forecasting model can gain insights into the overall financial performance of specific products, categories, or the entire business. Sales revenue data provides a comprehensive view of demand trends and patterns over time. Analysing revenue fluctuations allows the model to identify seasonal variations, product life cycles, and other factors influencing demand. Changes in sales revenue may be linked to pricing strategies. The model can analyse how alterations in product prices impact revenue and, consequently, adjust demand forecasts based on pricing dynamics. [16] has used the sales variable as a numeric value which represents the number of units sold.

Holiday

Holidays often lead to an increase in consumer spending, as people purchase more goods for celebrations, and gatherings. Incorporating holiday data into demand forecasting allows the model to anticipate and accommodate this surge in demand. Consumer preferences for certain products often change during holidays. For example, there may be increased demand for specific food items like vermicelli, sugar or milk during the eid holidays. Holiday data helps the model identify and predict shifts in product preferences. Holidays can lead to variations in product demand, and businesses need to adjust their inventory levels accordingly. Knowing the timing and significance of holidays allows for better inventory planning to meet increased demand during these periods. [11] mentions that since holidays affect sales they used it as a binary attribute where '0' indicated that it was an ordinary day and '1' indicated that it was a holiday. However, in [16] they have divided the variable 'holiday' into multiple variables including holiday type, holiday locale, holiday locale name, holiday description, and holiday transferred.

Location

Different locations may exhibit variations in consumer preferences, purchasing power, and demand for specific products. Geographical data allows the model to differentiate between regions and tailor demand forecasts accordingly. Geographical data helps identify areas with higher population density, which may experience different demand patterns than sparsely populated regions. This information is valuable for understanding the potential customer base in each location. Economic conditions can vary by location, affecting consumer spending habits, enabling more accurate predictions of demand based on local economic factors.

Weather

Weather patterns are often closely tied to seasons. Understanding how weather changes with the seasons helps the model predict seasonal variations in demand. For example, demand for cold drinks and juices increases during the summers or demand for dry fruits increases during the winters. Rain, snow, or other forms of precipitation can impact consumer mobility and preferences. For example, heavy rainfall may reduce foot traffic at brick-and-mortar stores. Weather data helps the model account for

these effects. Weather information including maximum and minimum temperature, and relative humidity were also used while preparing their data and selecting the variables for demand forecasting as explained in [11].

Model Shortlisting

Approach Considerations

Ensemble Approach

1. **Diverse Patterns:** When demand patterns vary significantly, an ensemble approach is explored to combine models effectively capturing diverse patterns for robust predictions.
2. **Model Complementarity:** Combining models with complementary strengths enhances the accuracy of the forecasting system.
3. **Increased Robustness:** Ensemble models minimise overfitting risks, ensuring reliable predictions with new data.

Non-Ensemble Approach

1. **Interpretability is Crucial:** For critical interpretability and easy explanation, a non-ensemble approach, such as linear regression, may be considered.
2. **Computational Efficiency:** In cases of limited computational resources or a need for quicker predictions, non-ensemble models offer a more efficient solution.
3. **Homogeneous Patterns:** For relatively homogeneous demand patterns, a single, well-tailored model may suffice without the added complexity of an ensemble.

Type of ML Models

Autoregressive Models

1. **Description:** Capture relationships between observations and lagged data, suitable for time series forecasting.
2. **Example:** Predicting daily sales based on historical sales data.

Exponential Smoothing Models

1. **Models:** SES, Holt's method, Holt-Winters method.
2. **Description:** Address trends and seasonality in time series data through exponential decay of past observations.
3. **Example:** Forecasting monthly revenue, considering regular patterns and long-term trends.

Machine Learning Regression Models

1. **Models:** Linear regression, polynomial regression, decision trees.
2. **Description:** Utilise historical data and relevant features for predicting future demand.
3. **Example:** Predicting weekly sales based on factors like promotions, holidays, and location.

Ensemble Models

1. **Models:** Random Forest Regressor, Gradient Boosting Regressor, XGBoost.
2. **Description:** Combine predictions from multiple models for enhanced accuracy and robustness.
3. **Example:** Integrating Random Forest with Gradient Boosting for a diverse and accurate demand forecast.

Deep Learning Models

1. **Models:** Recurrent Neural Networks (RNNs), LSTM, GRU.
2. **Description:** Capture sequential dependencies for handling complex patterns in time series data.
3. **Example:** Using LSTM to predict daily sales, considering sequential dependencies and temporal nuances.

Competitive Analysis Models

1. **Highlighted Model:** Random Forest Regressor for flexibility, simplicity, and explainability.
2. **Other Models:** Gradient Boosting techniques and LSTM recognized for capturing nuanced demand patterns.

Model Training and Testing

Ensemble Approach

Training Process

- **Data Source:** Historical data from various stores.
- **Implementation:** Collaborative contribution of models (Random Forest, Gradient Boosting, LSTM) for collective sales prediction, leveraging unique strengths of each model.

Validation Process

- **Validation Data Set:** 20% of historical data allocated for comprehensive accuracy assessment.
- **Evaluation Metrics:** RMSLE, RMSE, MAPE, MAE provide insights into ensemble performance.

Iterative Improvement

- **Continuous Learning:** Designed for ongoing learning from new data, enhancing collective model knowledge.
- **Model Exploration:** Exploration of new models and techniques ensures a dynamic system adapting to changing demand patterns.

Non-Ensemble Approach

Training Process

- **Data Source:** Historical data from major stores.
- **Implementation:** Relies on a single, well-tailored model (e.g., Linear Regression) for sales forecasting, chosen based on suitability for specific demand patterns.

Validation Process

- **Validation Data Set:** 20% of historical data reserved for validation purposes.
- **Evaluation Metrics:** Same metrics (RMSLE, RMSE) used to assess the performance of the selected non-ensemble model.

Iterative Improvement

- **Continuous Learning:** Similar to the ensemble approach, the non-ensemble system continuously learns from new data, refining the single model for improved accuracy.

Various models, including Linear Regression and Random Forest Regression, were traditionally used for short-term demand. However, boosting algorithms like Gradient Boosting Regressor [17], Light Gradient Boosting Machine Regressor [18], XGBoost [19], and Cat Boost Regressor [20] outperform traditional methods, especially when dealing with both numerical and categorical features. Additionally, Long-Short Term Memory (LSTM) models [21], including Bidirectional LSTMs, prove effective for sequential data like time series, making them suitable for long-term demand scenarios due to their memory retention capabilities.

Recently, Recurrent Neural Networks (RNNs), Long-Short Term Memory (LSTM), and Bi-directional Long-Short Term Memory (Bi-LSTM) gained prominence for their capacity to model nonlinear functions and capture long-term time-dependent patterns. Hewamalage et al. [22] conducted an empirical study on RNN forecasting models, revealing their ability to directly model seasonality with uniform patterns. However, deseasonalization is necessary for non-uniform patterns. LSTM, a specialised RNN, handles longer input-output connections, as demonstrated by Xu and Wang's [23] sales forecasting based on univariate time series. Bi-LSTM, examined for multivariate time series data, outperformed statistical methods due to the prevalent nonlinear trends in most time series data [24].

Integration with User Interface

The ML model will generate a recommended amount to order for each product as shown in Figure 5.1 while the grocery store is placing the order. The grocery store will display this recommended amount next to the input field for quantity. The grocery store can either order the recommended amount or enter the amount to order manually.

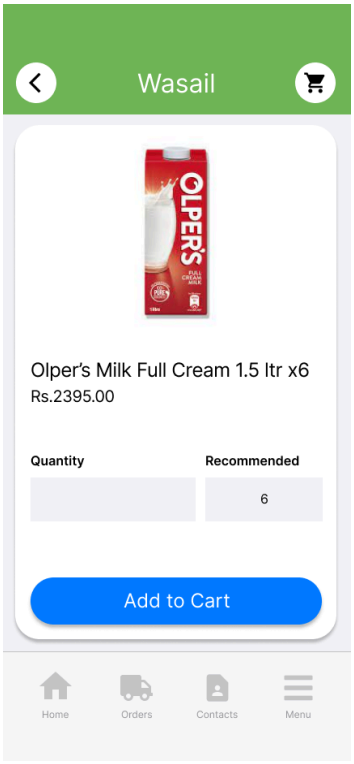


Figure 5.1

Recommendation Updates

To generate recommendations, the ML model will be provided with real-time data - product name, price, date, and store's location from the system and then holiday, exchange rate, and weather through APIs (such as Open Weather API).

Feedback and Learning

The data accumulated through the inventory management module will be used in the next iteration of the training. An increase in data overtime will increase the overall performance of the model as well. However, even though increasing the data is the most influential factor, it is not the only way to improve performance. The LSTM model (1995) has been around for some time, whereas TFT (2019) and N-HiTS (2022) are relatively new models. We shall continue to explore new discoveries in deep learning models, they might be more apt to target our problem.

Performance and Scalability

To ensure the performance and scalability of the ML component, the model will be deployed on Digital Ocean.

Testing and Validation

The ML model will be trained and evaluated on a dataset split into an 80:20 ratio for training and validation, respectively, with cross-validation applied during training.

User Roles

Wasail has the following user roles:

- Grocery Store: A user who is looking for a product for their store.
- Vendor: A user who is selling the products to the grocery store.
- Admin: A user who is part of the developer team.

User Stories

User Story: Placing the Order

As a grocery store, I would want to place an order for a product while receiving a recommended amount to order that would ensure maximum profit (as seen below in Figure 5.2).

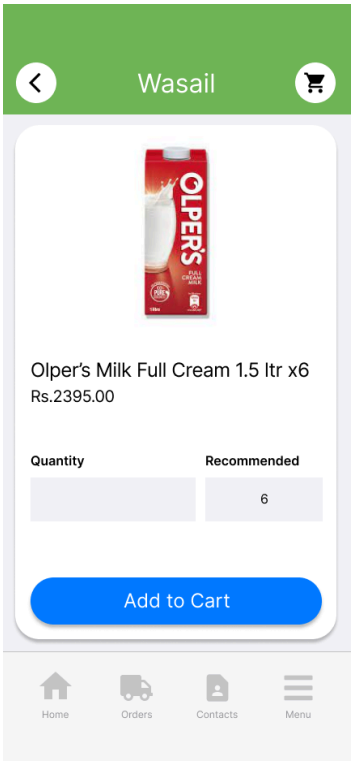


Figure 5.2

Acceptance Criteria:

- Once I have clicked on the vendor's profile, the app should display the vendor's details which should include the contact information, the product listings, and an option to connect *(future improvement: ratings and reviews)*.
- While I'm on the vendor's profile, the app displays the product that I have searched for under the "Searched Product" section, and the other products the vendors sell under the "Popular Products" section.
- I can click on the product I want to order, and the app will take me to another screen where I will be able to select the quantity of the product.
- A recommended amount to order would be displayed.
- I should be able to order the recommended amount or enter the amount to order myself.
- Once I have filled out the aforementioned criteria, I would select the "Add to Cart" button and my order would be added to the cart.
- Then, I can open the cart and place the order.

User Story: Searching for a Product

As a grocery store, I would like the ability to look up products online (as seen below in Figure 5.3 and 5.4).

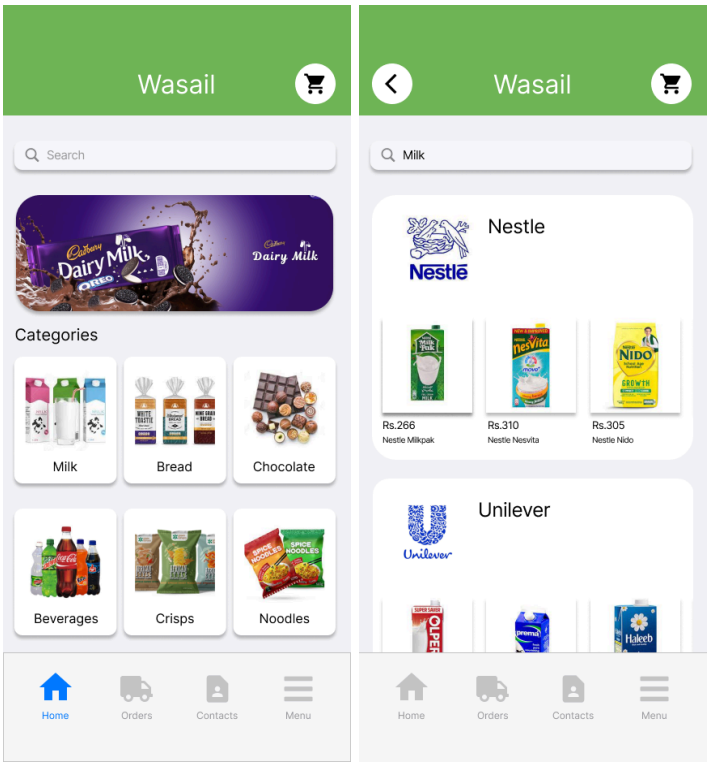


Figure 5.3 & 5.4

Acceptance Criteria:

- When I log into the app, I should see a search bar on the main screen along with popular categories.
- I can search the product by the name of the product (e.g. Milkpak), category of the product (e.g. Milk), and other suggestions as alternatives.
- After entering a search term, the app should present a list of vendors offering the product I need (along with the products), taking into account their ability to deliver to my store's location.
- I can click on the vendor's profile to view more details.
- (Future improvement: Sorting by price low to high, ratings, orders completed, etc)
- (Future improvement: Be able to filter products based on quantity such as a pack, half a pack or individual items, amount of litres of bottles, etc)

Functional Requirements

This section details functional requirements for:

- Grocery Store Mobile App
- Vendor Mobile App
- Admin Web App

Grocery Store and Vendor (FR1)

FR1.1: Language Selection

- **Description**: The system should allow the user to select a language (Fig 5.5).
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user has not specified their preferred language for displaying the text in the app.
- **Postcondition**: The user has specified their preferred language.
- **Details**:
    1. The user is given the option to select a language.
    2. The available options are English, Roman Urdu, and Urdu.
    3. The user's choice for language is saved.

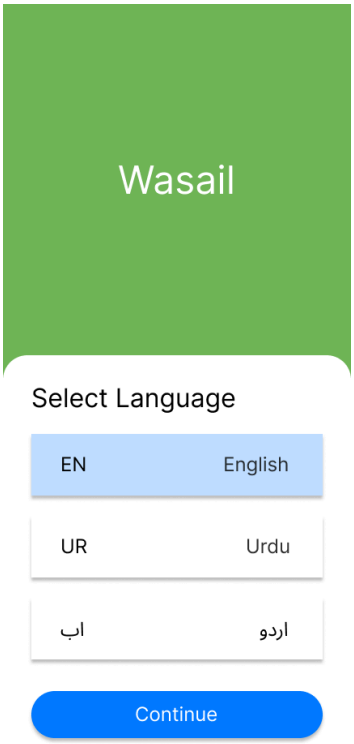


Figure 5.5

FR1.2: Phone Registration

- **Description**: The system should allow users to create an account using their phone number (Figure 5.6).
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user has opened the app.
- **Postcondition**: The user is directed to the phone number confirmation screen.
- **Details**:
    1. User would provide a valid phone number.
    2. The system validates the phone number format ensuring that it has been entered correctly.

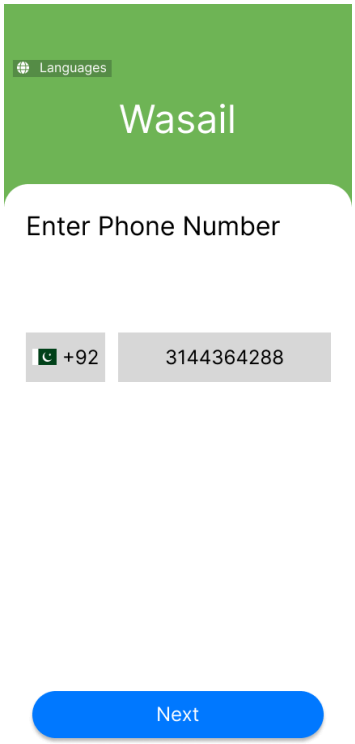


Figure 5.6

FR1.3: Phone Number Confirmation

- **Description**: The system should allow users to confirm if they entered the number correctly (Figure 5.7).
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user has entered their phone number.
- **Postcondition**: The user is directed to the one-time password (OTP) screen.
- **Details**:
    1. The user is asked to confirm if their phone number has been entered correctly.
    2. The system gives the user the option to edit their phone number in case it has not been entered correctly.
    3. Upon confirmation, the system should ask the user to create an account.

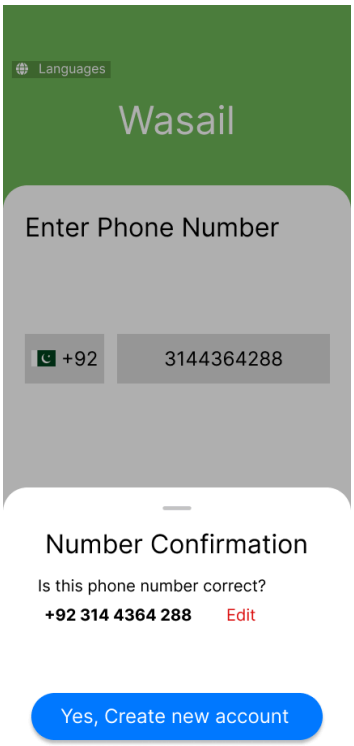


Figure 5.7

FR1.4: Phone Number Exists

- **Description**: The system should check if the phone number exists in the database.
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user has confirmed their phone number.
- **Postcondition**: The user is redirected to either the login or the registration page.
- **Details**:
    1. After phone number confirmation, the system checks if it exists in the database.
    2. If the phone number exists in the database, the user is redirected to the login page.
    3. If the phone number does not exist, that means that it is a new user and the system redirects the user to the registration page.

FR1.5: OTP Code Generation and Delivery

- **Description**: The system should send an OTP code to the user's phone number (Figure 5.8 & 5.9).
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user has confirmed their phone number.
- **Postcondition**: The user has entered the OTP and is directed to the account details page.
- **Details**:
    4. The system generates a 4-digit unique OTP code.
    5. The system sends the OTP code to the user's phone number via SMS.
    6. The OTP code can be resent after 60 seconds and the timer is being shown to the user.
    7. The user is asked to enter the OTP.
    8. The system verifies the validity of the OTP code by comparing it to the generated code sent to the given phone number.
    9. Upon verification, the user shall be directed to the account details page.

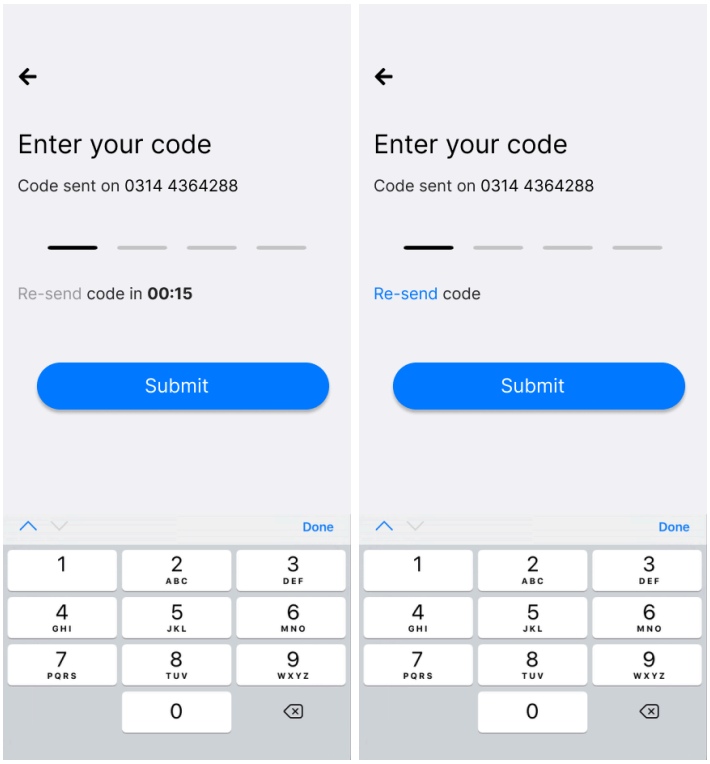

Enter your code
Code sent on 0314 4364288
Re-send code in 00:15
Submit
Done
1
2 ABC
3 DEF
4 GHI
5 JKL
6 MNO
7 PQRS
8 TUV
9 WXYZ
0
Enter your code
Code sent on 0314 4364288
Re-send code
Submit
Done
1
2 ABC
3 DEF
4 GHI
5 JKL
6 MNO
7 PQRS
8 TUV
9 WXYZ
0

Figure 5.8 & 5.9

FR1.6: Login

- **Description**:The system should allow the registered users to log in using their credentials (Figure 5.10 & 5.11).
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user is not logged in.
- **Postcondition**: The user is logged in.
- **Details**:
    1. The user is asked to enter their phone number.
    2. If the phone number is registered already with the system, the user is asked to enter their password.
    3. The system validates the user's credentials.
    4. Upon successful validation, the user is granted access to their profile.

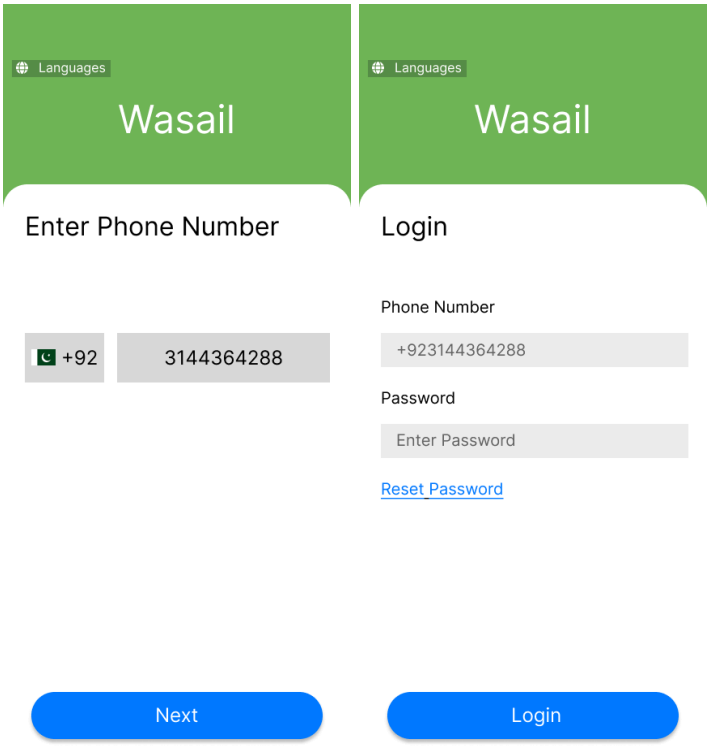


Figure 5.10 & 5.11

FR1.7: Logout

- **Description**: The system should allow the user to logout
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user is logged in.
- **Postcondition**: The user is logged out.
- **Details**:
    1. The system shall give the user the option to log out
    2. The user can log out of the system by using the option

FR1.8: Reset Password

- **Description**: The system should allow the user to reset their password
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user is registered.
- **Postcondition**: The user's password is successfully reset
- **Details**:
    1. The system should give the user the option for resetting their password
    2. In order to reset their password, the system generates a 4 digit unique OTP code.
    3. The system sends the OTP code to the user's phone number via SMS.
    4. Upon verification, the user shall enter their new password.
    5. The user shall re enter their password for confirmation.
    6. After confirmation, the user shall save the password.
    7. The system updates the password in the database.

FR1.9: View Profile

- **Description**: The system should allow the user to view their own profile
- **Actor**: Grocery Store, Vendor
- **Precondition**: The user is on their own profile.
- **Postcondition**: The user is able to view their profile.
- **Details**:
    1. The system displays the user's profile.
    2. The user can view their profile to see their account details.

Grocery Store (FR2)

FR2.9: Account Details

- **Description**: The system should allow the user to enter their account details (Figure 5.12).
- **Actor**: Grocery Store
- **Precondition**: The user has verified their phone number.
- **Postcondition**: The user has registered.
- **Details**:

1. The user enters their account details including their name, store's name, store's address, password, and location.
2. User account information is stored in the database.

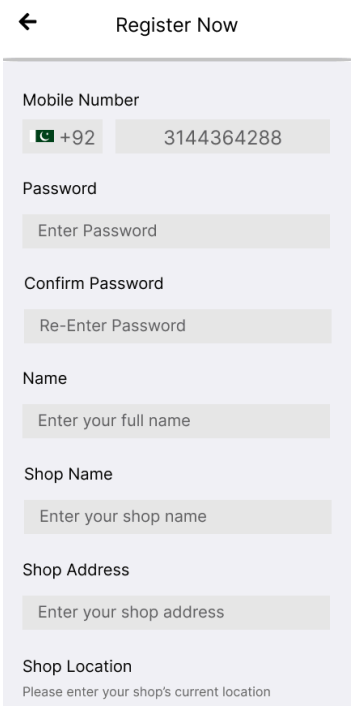


Figure 5.12

FR2.10: Search Product

- **Description**: The system should allow the users to search products based on product name and category (Figure 5.13 & 5.14).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the home page.
- **Postcondition**: The system displays the list of matching products along with the vendors that sell them.
- **Details**:
    1. The user can enter the search criteria such as product name or category.
    2. The system retrieves and displays the list of matching products along with the vendors that sell them.
    3. If no matches are found, the system provides appropriate feedback.

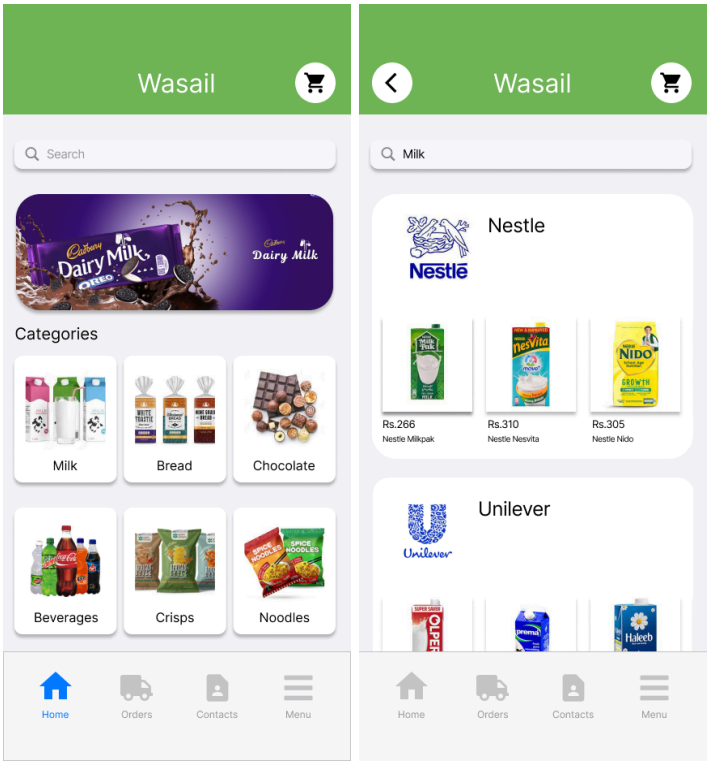


Figure 5.13 & 5.14

FR2.11: Search Vendor

- **Description**: The system should allow the users to search vendors based on their name (Figure 5.15).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the home page.
- **Postcondition**: The system displays a list of vendors that deliver in their area based on the search.
- **Details**:
    1. The user can enter the search criteria such as vendor name.
    2. The system retrieves and displays the vendor along with its popular products.
    3. If no matches are found, the system provides appropriate feedback.

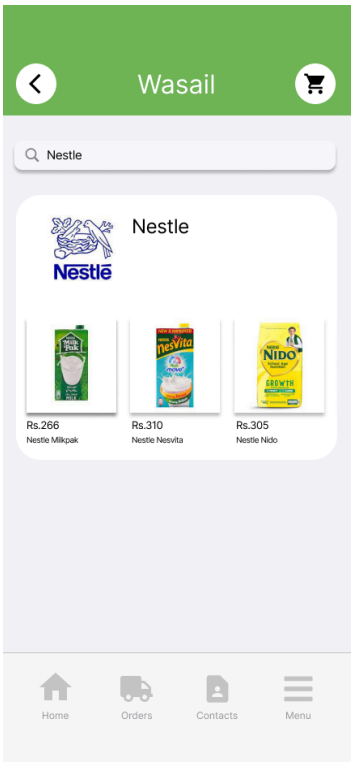


Figure 5.15

FR2.12: Browse Category

- **Description**: The system should allow the user to select a category from the home page directly (Figure 5.16 & 5.17).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the home page.
- **Postcondition**: The system displays a list of products based on category selection.
- **Details**:
    1. The user can select the categories that are being displayed to them.
    2. The system retrieves and displays products that fall under the category along with the vendors who sell them.
    3. If no match is found, the system provides appropriate feedback.

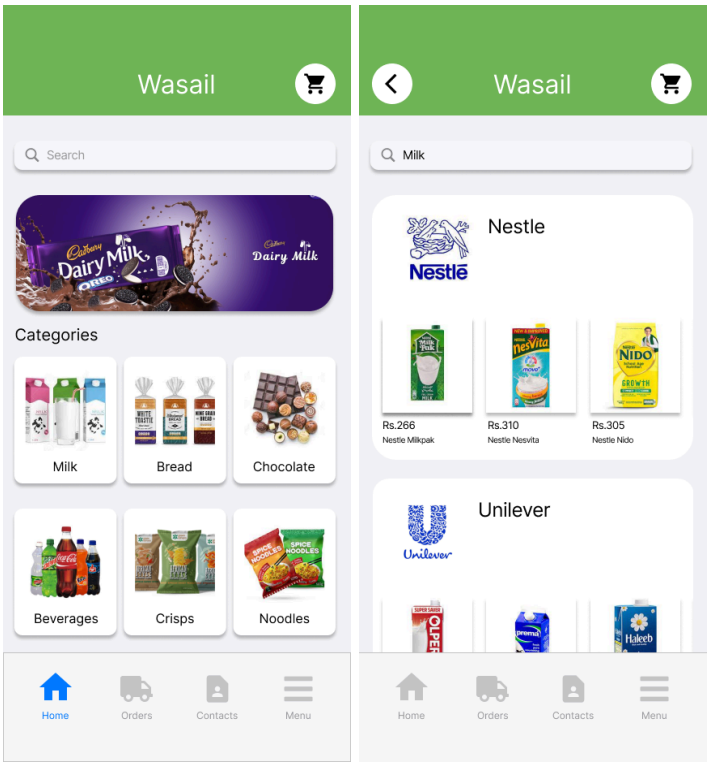


Figure 5.16 & 5.17

FR2.13: View Vendor Profile

- **Description**: The system should allow the user to view the vendor's profile (Figure 5.18).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and has searched the vendor or is on the vendor list page.
- **Postcondition**: The system displays the vendor’s profile.
- **Details**:
    1. The user can select the vendor’s profile in order to view it.
    2. The system retrieves the vendor’s information including their profile picture, name, product listing, and displays it for the user.

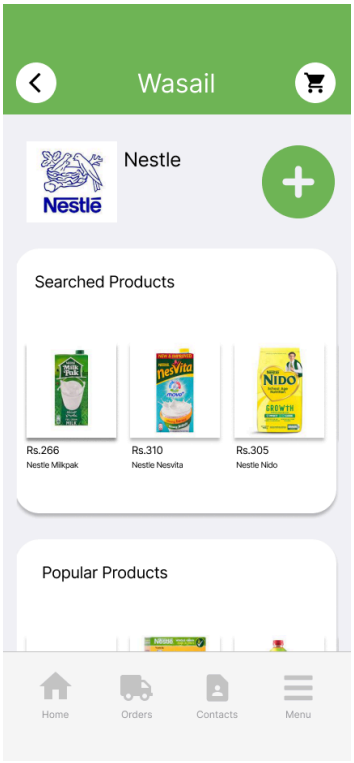


Figure 5.18

FR2.14: Add Vendor to Vendor List

- **Description**: The system should allow the user to add the vendor to their vendor list (Figure 5.19).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the vendor's profile.
- **Postcondition**: The vendor is added to the vendor list.
- **Details**:
    1. The user can select the option to add the vendor.
    2. The vendor is added to the vendors list of the user.

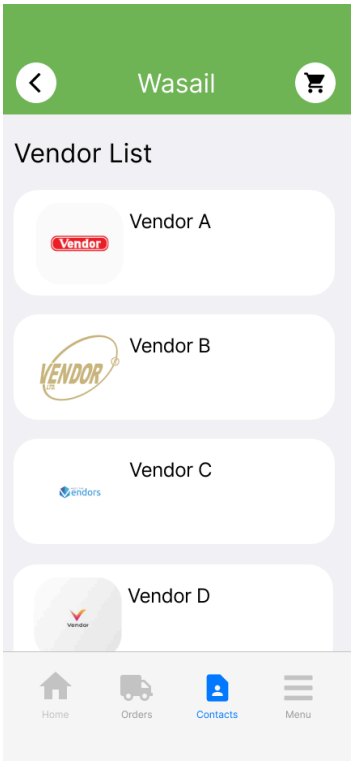


Figure 5.19

FR2.15: Contact Vendor

- **Description**: The system should allow the user to contact the vendor (Figure 5.20).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the vendor’s profile.
- **Postcondition**: The user has contacted the vendor.
- **Details**:
    1. The system displays the vendor's phone number on the profile.
    2. The user can contact the vendor via message or phone call.

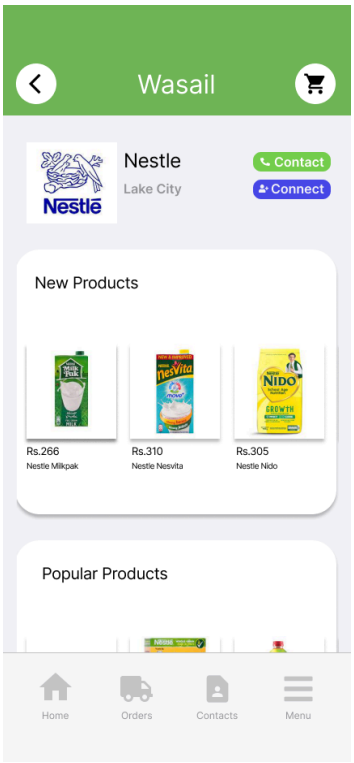


Figure 5.20

FR2.16: View Products on the Vendor’s Profile

- **Description**: The system should allow the user to view all the products that the vendors sell on their profile (Figure 5.21).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the vendor’s profile.
- **Postcondition**: The user has viewed all the products.
- **Details**:
    1. The system retrieves all the products that the vendor sells and displays it to the user.
    2. The user can view the products and scroll through them.

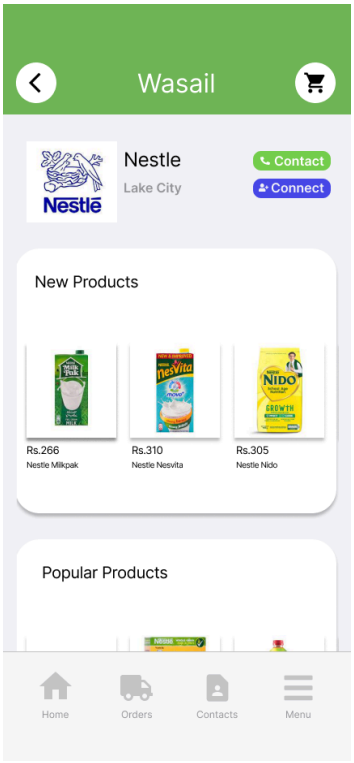


Figure 5.21

FR2.17: View Searched Product

- **Description**: The system should allow the user to view the searched product on the vendor's profile (Figure 5.22).
- **Actor**: Grocery Store
- **Precondition**: The user has searched the product and is on the vendor’s profile.
- **Postcondition**: The user has viewed the searched product on the vendor's profile.
- **Details**:
    1. The system displays the searched product on the vendor’s profile.
    2. The user can view the products along with their prices and scroll through them.

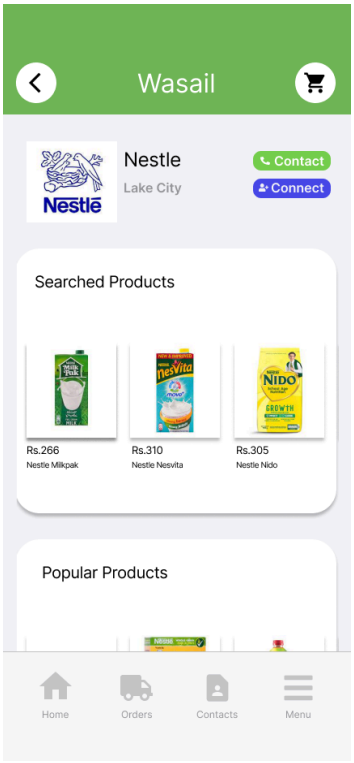


Figure 5.22

FR2.18: Select Products

- **Description**: The system should allow the user to select the product that they want to order (Figure 5.23).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the vendor's profile.
- **Postcondition**: The user is directed to the order placement page.
- **Details**:
    1. The system should display products on the vendor's profile.
    2. The user can select from the product list that is being displayed so they can order it.
    3. The user can proceed to place the order.

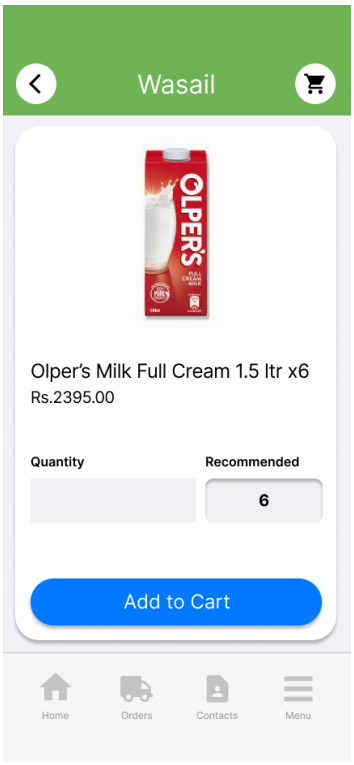


Figure 5.23

FR2.19: Order Recommendation

Description: The system should allow the user to view the recommended amount to order that would ensure maximum profit (Figure 5.24).

- **Actor**: Grocery Store
- **Precondition**: The user is logged in and has selected the product.
- **Postcondition**: The user is given a recommendation for the amount of product to order.
- **Details**:
    1. After product selection, the user is recommended an amount to order.
    2. To generate recommendations, the ML model will be provided with real-time data - product name, price, date, and store's location from the system and then holiday, exchange rate, and weather through APIs (such as Open Weather API).

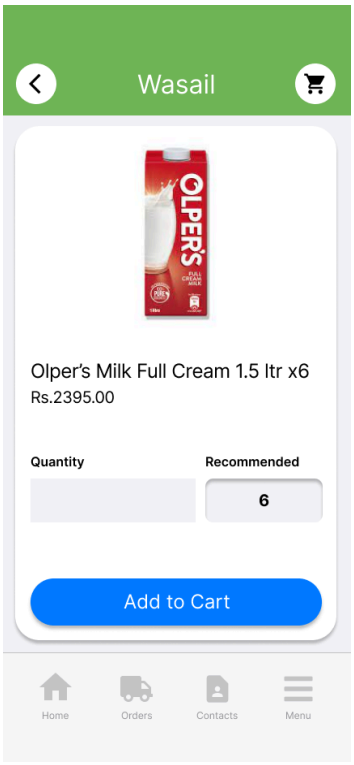


Figure 5.24

FR2.20: Quantity Selection

- **Description**: The system should allow the user to add their own amount to order (Figure 5.25).
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and has selected the product.
- **Postcondition**: The user has entered the quantity of the product.
- **Details**:
    1. The user is given the option to enter the amount of product they want to order.
    2. The user enters the amount of product.
    3. The user adds the product to the cart.

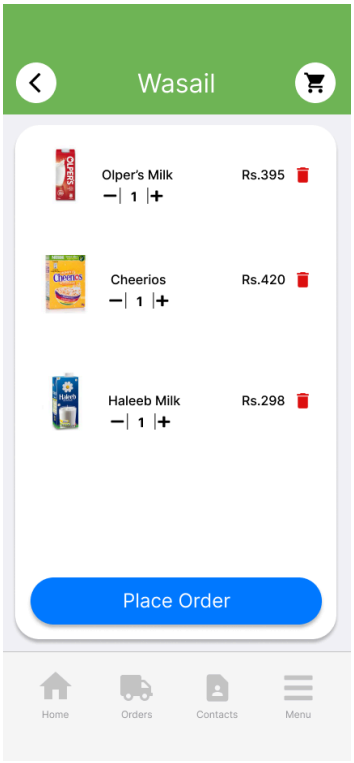


Figure 5.25

FR2.21: Remove Product

- **Description**: The system should allow the user to remove the product (Figure 5.26).
- **Actor**: Grocery Store
- **Precondition**: The product is added to the cart.
- **Postcondition**: The product is removed from the cart.
- **Details**:
    1. The system gives the option to the user to remove the product from the cart.
    2. The user removes the product they do not want to order from the cart.
    3. The system stops displaying that product in their cart.

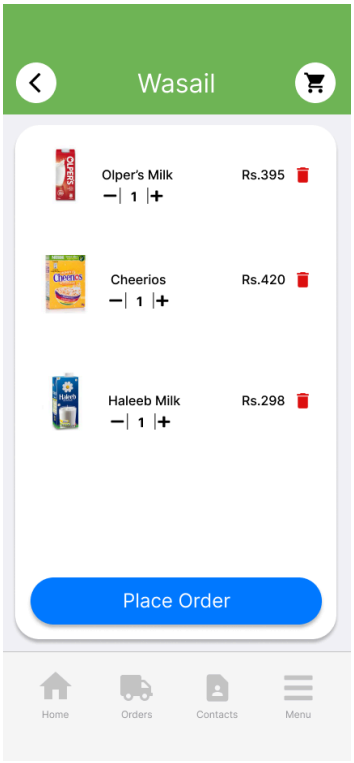


Figure 5.26

FR2.22: Order Placement

- **Description**: The system should allow the user to place the order (Figure 5.27).
- **Actor**: Grocery Store
- **Precondition**: The product is added to the cart.
- **Postcondition**: The order is placed.
- **Details**:
    1. After the user has added the products to their cart, the system gives the option to place the order.
    2. The order is placed to the vendor.
    3. The order is stored in the database.

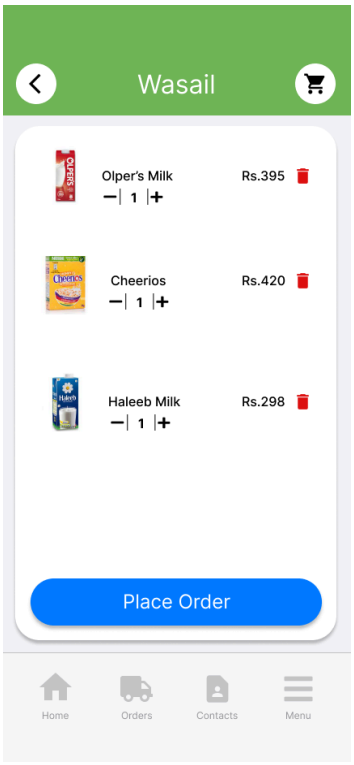


Figure 5.27

FR2.23: View Order

- **Description**: The system should allow the users to view the current orders that they have placed (Figure 5.28).
- **Actor**: Grocery Store
- **Precondition**: The user is on the orders page.
- **Postcondition**: The user has viewed the orders that they have placed.
- **Details**:
    1. The system should display the orders that the user has placed.
    2. The user can view their orders.

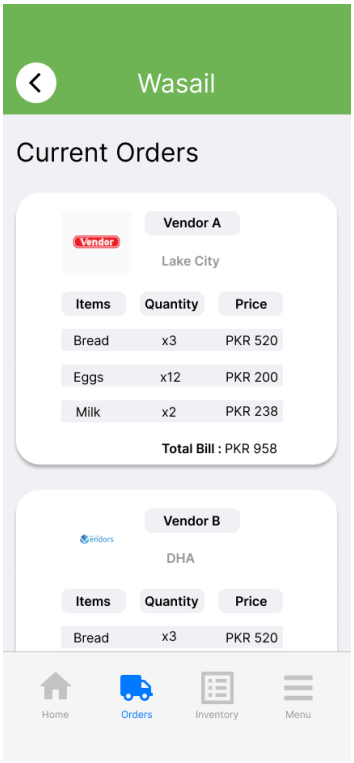


Figure 5.28

FR2.24: Order Tracking

- **Description**: The system should allow the user to track the order (Figure 5.29, 5.30 & 5.31).
- **Actor**: Grocery Store
- **Precondition**: The user is on the orders page.
- **Postcondition**: The user is able to track orders.
- **Details**:
    1. The system displays all the orders that are placed by the user.
    2. The user can select the order which they want to track.
    3. Upon order selection, the user can see which process the order is in. The three options are in process, on its way, and delivered.
    4. The system notifies the user when the order is delivered
    5. The order is marked as delivered in the system

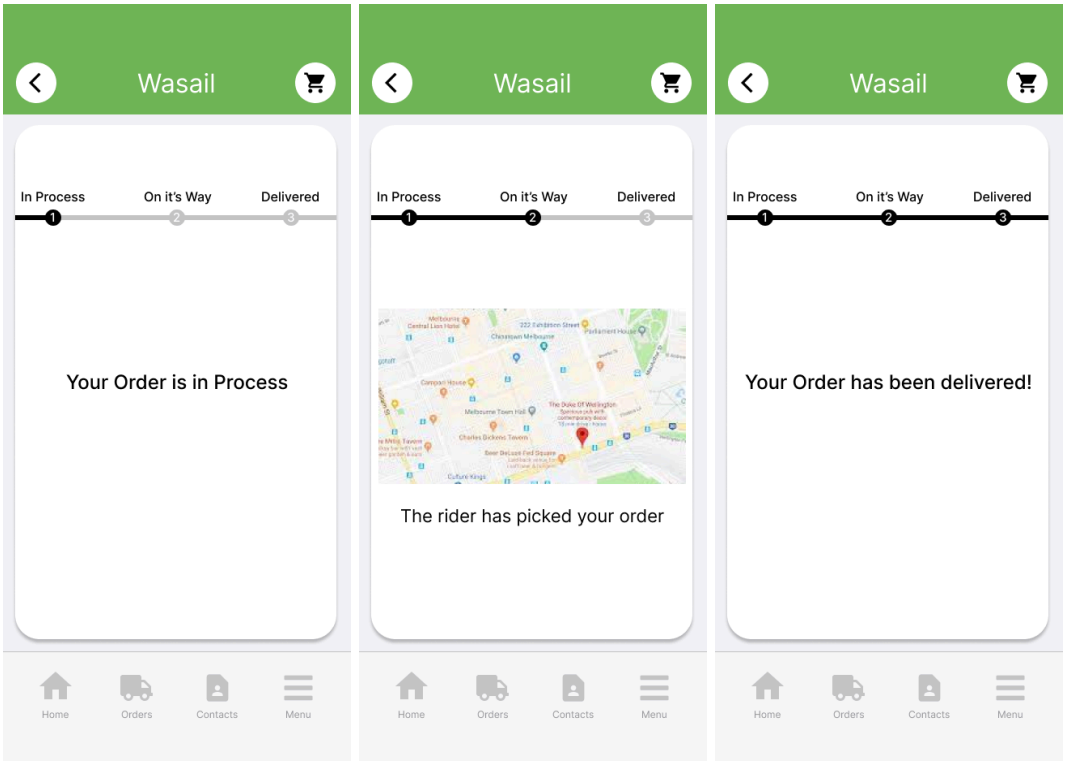


Figure 5.29, 5.30 & 5.31

FR2.25: View Vendor List

- **Description**: The system should allow the user to view the vendors in the vendor list (Figure 5.32).
- **Actor**: Grocery Store
- **Precondition**: The user is on the vendor page.
- **Postcondition**: The user is able to view the vendors that they have added.
- **Details**:
    1. The system displays all the vendors that the user has added.
    2. The user can view the vendors in the vendors list.

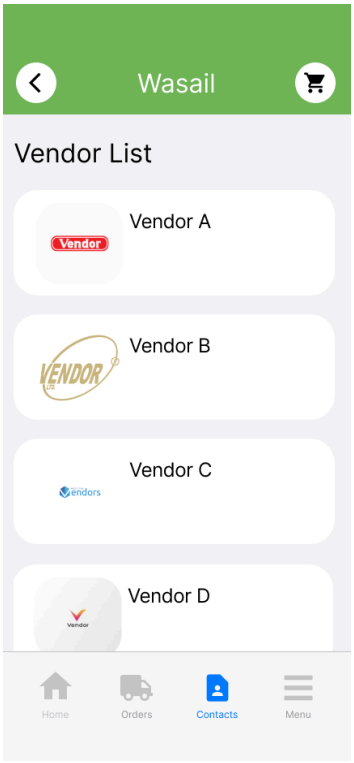


Figure 5.32

FR2.26: View Order History

- **Description**: The system should allow the user to view their previous orders.
- **Actor**: Grocery Store
- **Precondition**: The user is on the orders history page.
- **Postcondition**: The user is able to view their previous orders.
- **Details**:
    1. The system displays all the previous orders the user has placed.
    2. The user can view all their previous orders.

FR2.27: Edit Profile

- **Description**: The system should allow the user to edit their profile details.
- **Actor**: Grocery Store
- **Precondition**: The user is on their own profile.
- **Postcondition**: The user's profile is edited.
- **Details**:
    1. The system displays the profile to the user.
    2. The user can edit the account details that include name, store name, address.
    3. The system updates the information in the database.

FR2.28: Add to Cart

- **Description**: The system should allow the user to add products to their cart in order to place an order.
- **Actor**: Grocery Store
- **Precondition**: The user has selected the quantity of the product they want to order.
- **Postcondition**: The product has been added to cart.
- **Details**:
    1. After the user has selected the quantity of the product they want to order, the system gives the option to add the product to the cart.
    2. The user can select the option and the product gets added to the cart.
    3. The system updates the cart with the selected product and quantity.

FR2.29: View Cart

- **Description**: The system should allow the user to view their cart.
- **Actor**: Grocery Store
- **Precondition**: The user has added a product to the cart.
- **Postcondition**: The user is able to view the cart.
- **Details**:
    1. The system should display the products in the cart.

2. The user can view the cart at any time to review the added products and quantities.

FR2.30: Update Product Quantity in Cart

- **Description**: The system should allow the user to update the quantity of the product added in their cart.
- **Actor**: Grocery Store
- **Precondition**: The user has added a product to the cart.
- **Postcondition**: The user has updated the quantity of the product.
- **Details**:
    1. The system should display the products in the cart.
    2. The system should give the option to increase or decrease the quantity of the product.
    3. The user can update the quantity of the product by either increasing or decreasing it inside the cart.

FR2.30: Clear Cart

- **Description**: The system should allow the user to clear the cart in order to remove all the products from it.
- **Actor**: Grocery Store
- **Precondition**: The user is inside the cart.
- **Postcondition**: The cart is empty.
- **Details**:
    1. The system should display the products in the cart.
    2. The system should give the option to clear the cart.
    3. Once the user selects the option, the system should clear the cart and remove all the products from it.

FR2.31: Search Category

- **Description**: The system should allow the users to search categories based on their name.
- **Actor**: Grocery Store
- **Precondition**: The user is logged in and on the home page.
- **Postcondition**: The system displays the list of matching categories along with products that belong to them.
- **Details**:
    1. The user can enter the search criteria such as category name.
    2. The system retrieves and displays the list of matching categories along with products that belong to them.
    3. If no matches are found, the system provides appropriate feedback.

Vendor (FR3)

FR3.9: Vendor Registration

- **Description**: The system should allow the user to enter their account details
- **Actor**: Vendor
- **Precondition**: The user has verified their phone number.
- **Postcondition**: The user has registered.
- **Details**:
    1. The user enters their account details including their password, full name, username, and delivery areas.
    2. User account information is stored in the database.

FR3.10: Valid Password

- **Description**: The system should allow the user to enter password
- **Actor**: Vendor
- **Precondition**: The user is on the registration page.
- **Postcondition**: The user has entered a valid password.
- **Details**:
    1. Once the user has entered their password, the system checks whether it is valid (against the criteria) or not.
    2. The criteria for checking the password is that the password's length should be a minimum of eight characters, it should have a special character.
    3. If the password meets the criteria, the system indicates this to the user.
    4. If the password does not meet the criteria, the system prompts the user to enter the password according to the criteria.

FR3.11: Username Exists

- **Description**: The system should check if the username exists in the database.
- **Actor**: Vendor
- **Precondition**: The user is on the registration page.
- **Postcondition**: The user has entered a username.
- **Details**:
    1. Once the user has entered their username, the system checks if it exists in the database.
    2. If the username already exists in the database (belongs to another user), the system asks the user to enter a different username.
    3. If the username does not exist, the system allows the user to use that username.

FR3.12: Search Product in Inventory

- **Description**: The system should allow the user to search for a product from their inventory (Figure 5.33 & 5.34).
- **Actor**: Vendor
- **Precondition**: The user is logged in and is on the home page.
- **Postcondition**: The system retrieves the matching product's page.
- **Details**:
    1. The user can enter the search criteria namely product name.
    2. The system retrieves the matching product's page.
    3. If no matches are found, the system provides appropriate feedback.

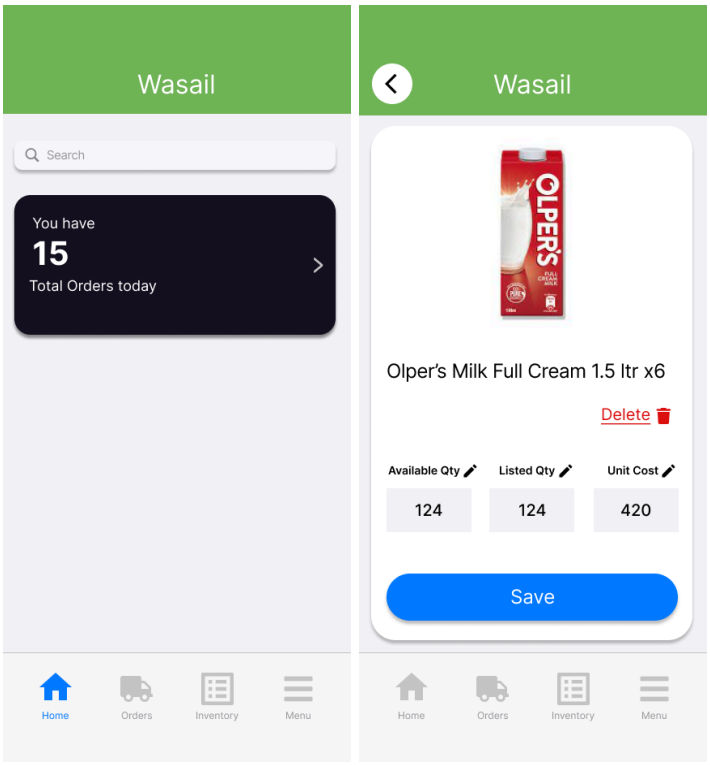


Figure 5.33 & 5.34

FR3.13: Add Product to Inventory

- **Description**: The system should allow the user to add a new product to their inventory (Figure 5.35).
- **Actor**: Vendor
- **Precondition**: The user is logged in and on the inventory page.
- **Postcondition**: The system has directed the user to the all products search page.
- **Detail**
    1. The user wants to add a new product to their inventory.
    2. The system gives the user an option to add a new product.
    3. The user selects the option to add a new product to the inventory.

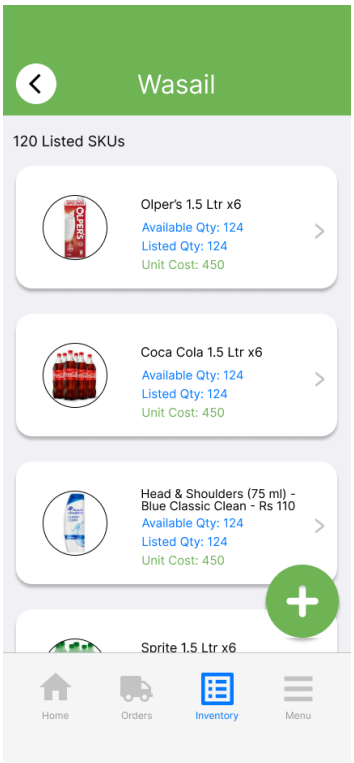


Figure 5.35

FR3.14: All Products Search

- **Description**: The system should allow the user to search all products from the system's product listing (Figure 5.36, 5.37 & 5.38).
- **Actor**: Vendor
- **Precondition**: The user is on the all products search page.
- **Postcondition**: The product is added to their inventory.
- **Details**:
    1. The user searches for an item from among all products.
    2. The system is prompted to generate a list of item suggestions.
    3. The user selects an item from the suggestions.
    4. The item is added to the inventory list.
    5. The user is redirected to update details of the selected item namely its listed quantity, available quantity, and unit cost.

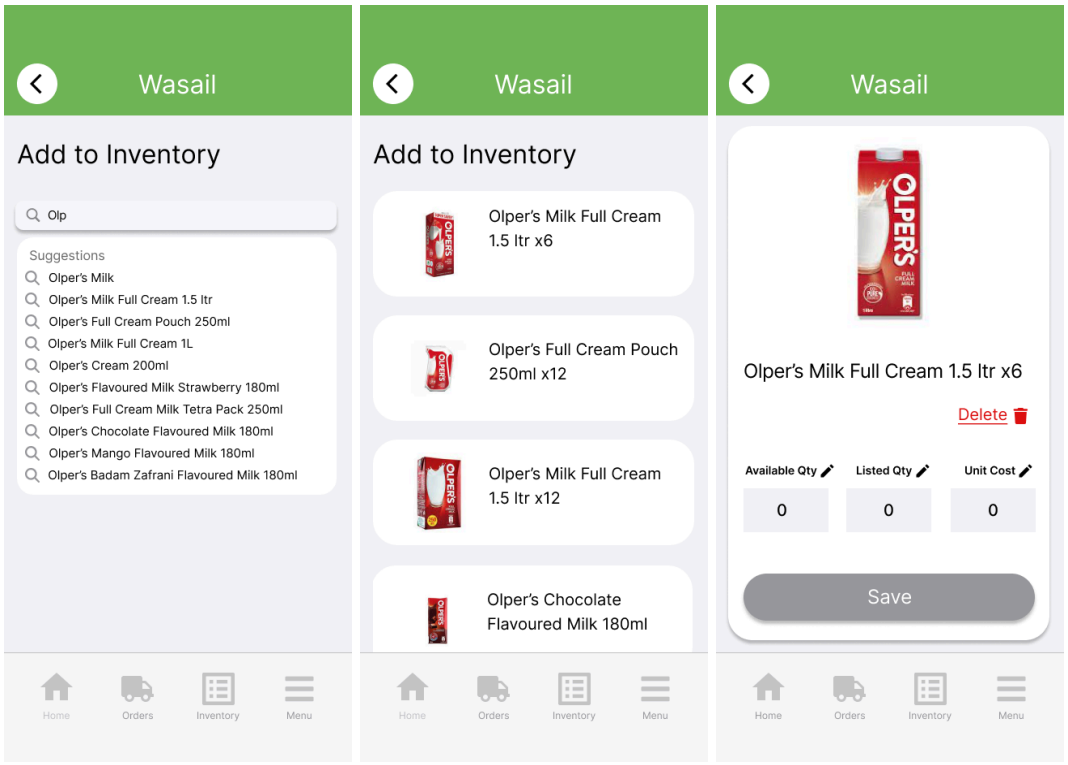


Figure 5.36, 5.37 & 5.38

FR3.15: Remove Product

- **Description**: The system should allow the user to remove a product from the inventory (Figure 5.39).
- **Actor**: Vendor
- **Precondition**: The user is on the product's page.
- **Postcondition**: The product has been removed from their inventory.
- **Details**:
    1. The system gives the option to the user to remove the product from the inventory.
    2. The user removes the product they do not want to keep in the inventory.
    3. The system stops displaying that product in the inventory.

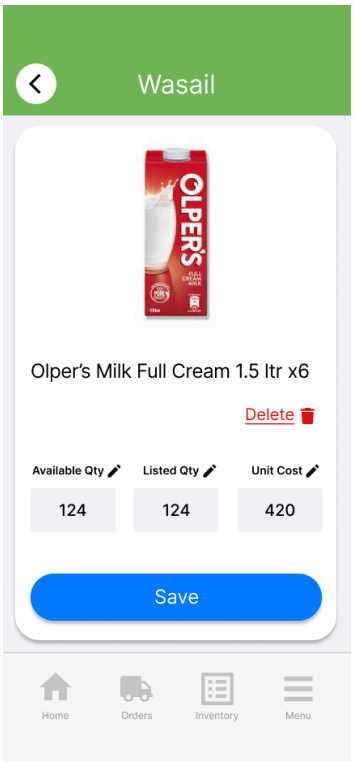


Figure 5.39

FR3.16: Edit Details of Product

- **Description**: The system should allow the user to edit the details of products already in their inventory (Figure 5.40).
- **Actor**: Vendor
- **Precondition**: The user has added the product to their inventory
- **Postcondition**: The details of the product have been edited.
- **Details**:
    1. The system displays the product page for the user.
    2. The system gives the user the option to edit the product details.
    3. The user edits the product details namely its listed quantity, available quantity, and unit cost.
    4. The system will save the edited product details.

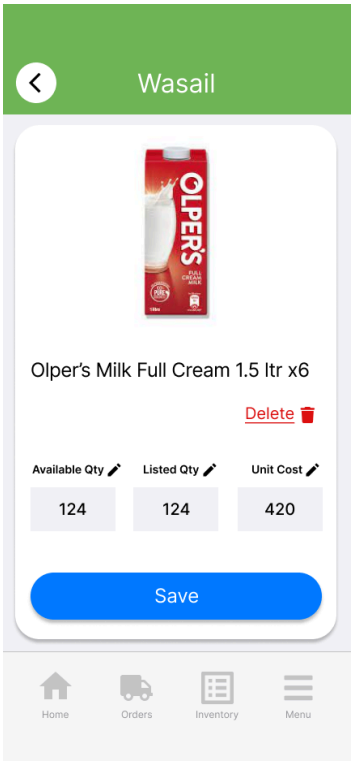


Figure 5.40

FR3.17: View Inventory

- **Description**: The system should allow the user to view their inventory i.e. product listings (Figure 5.41)
- **Actor**: Vendor
- **Precondition**: The user is logged in and on the inventory page.
- **Postcondition**: The user is able to view the inventory.
- **Details**:
    1. The system should display the inventory of the user.
    2. The user can view their inventory.

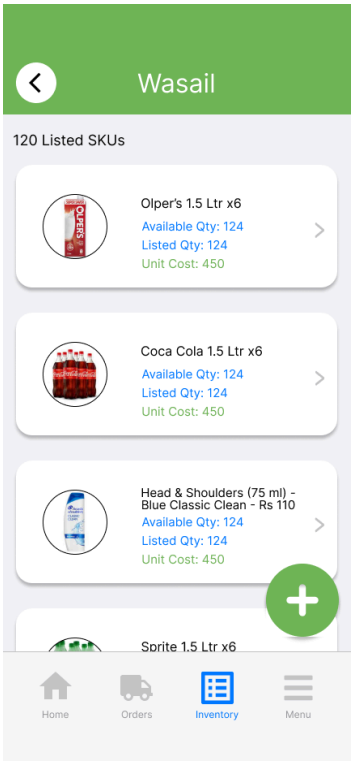


Figure 5.41

FR3.18: View Current Orders

- **Description**: The system should allow the user to view the total number of current orders from all grocery stores (Figure 5.42).
- **Actor**: Vendor
- **Precondition**: The user is logged in and on the orders page.
- **Postcondition**: The user can view all the orders received.
- **Details**:
    1. The system displays all the orders that the user has received and has to deliver.
    2. The user can view all the orders.

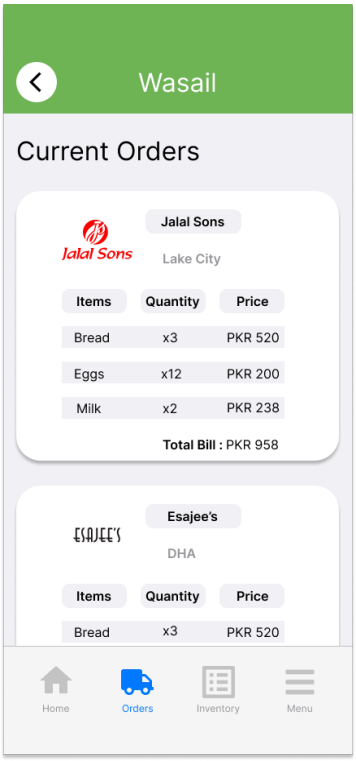


Figure 5.42

FR3.19: View Grocery Stores List

- **Description**: The system should allow the user to view the grocery store list (Figure 5.43).
- **Actor**: Vendor
- **Precondition**: The user is logged in and on the store list page.
- **Postcondition**: The user is able to view the list of grocery stores that placed orders.
- **Details**:
    1. The system displays all the grocery stores that have placed orders.
    2. The user can view the grocery stores in the stores list.

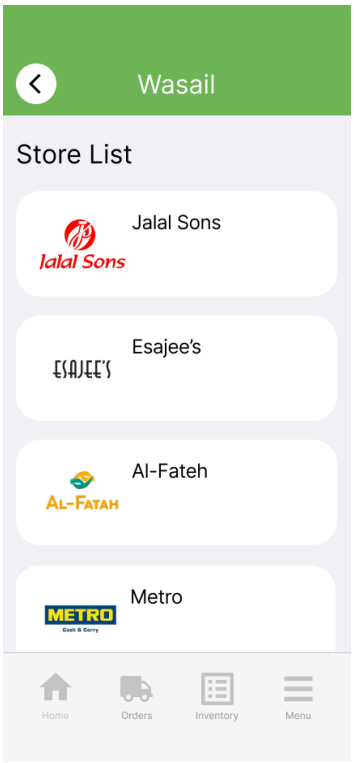


Figure 5.43

FR3.20: View Grocery Store Profile

- **Description**: The system should allow the user to view the grocery store profile (Figure 5.44).
- **Actor**: Vendor
- **Precondition**: The user is on the store list page
- **Postcondition**: The system displays the grocery store's profile.
- **Details**:
    1. The user can select the grocery store's profile to view it.
    2. The system retrieves the grocery store's information including their profile picture, name, and area in which they are, and displays it for the user.

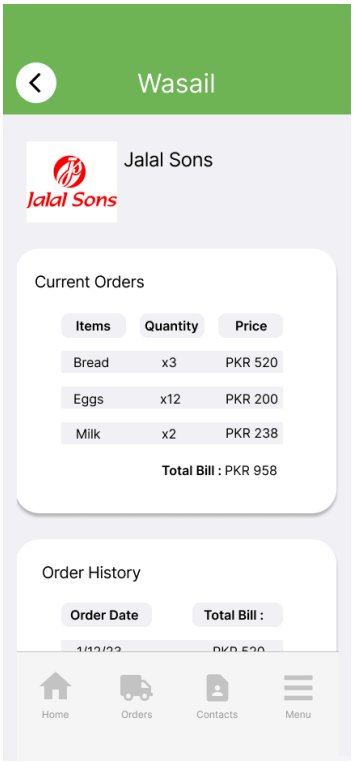


Figure 5.44

FR3.21: View Grocery Store Current Order

- **Description**: The system should allow the user to view the current order placed by the grocery store (profile), (Figure 5.45).
- **Actor**: Vendor
- **Precondition**: The user is on the grocery store's profile page.
- **Postcondition**: The user has viewed the current order that they have received from the grocery store.
- **Details**:
    1. The system should display the current orders that the user has received from the grocery store.
    2. The user can view their current orders received.

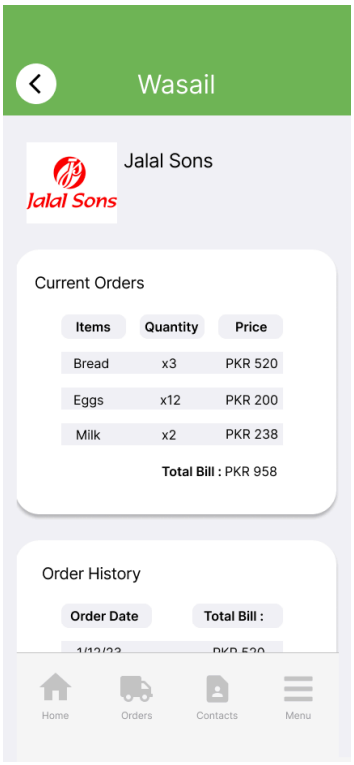


Figure 5.45

FR3.22: View Orders History

- **Description**: The system should allow the user to view the orders they have already completed delivering (Figure 5.46).
- **Actor**: Vendor
- **Precondition**: The user is on the orders history page.
- **Postcondition**: The user is able to view the delivered orders.
- **Details**:
    1. The system displays all the orders the user has delivered.
    2. The user can view all their delivered orders.

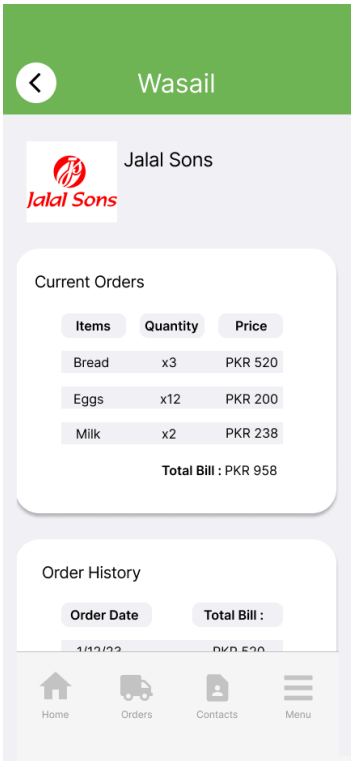


Figure 5.46

FR3.23: Order Dispatch Tracking

- **Description**: The system should allow the user to track the dispatched order (Fig 5.47 & 5.48).
- **Actor**: Vendor
- **Precondition**: The user is on the orders page.
- **Postcondition**: The user is able to track order dispatches.
- **Details**:
    1. The system displays all the orders that are dispatched by the user.
    2. The user can select the order in which they want to track dispatch progress.
    3. Upon order selection, the user can update which process the order is in. The three options are in process, on their way, and delivered.
    4. The user updates the system on the progress of order delivery.
    5. The user successfully completes the delivery and updates the system.
    6. The system notifies of completed delivery.

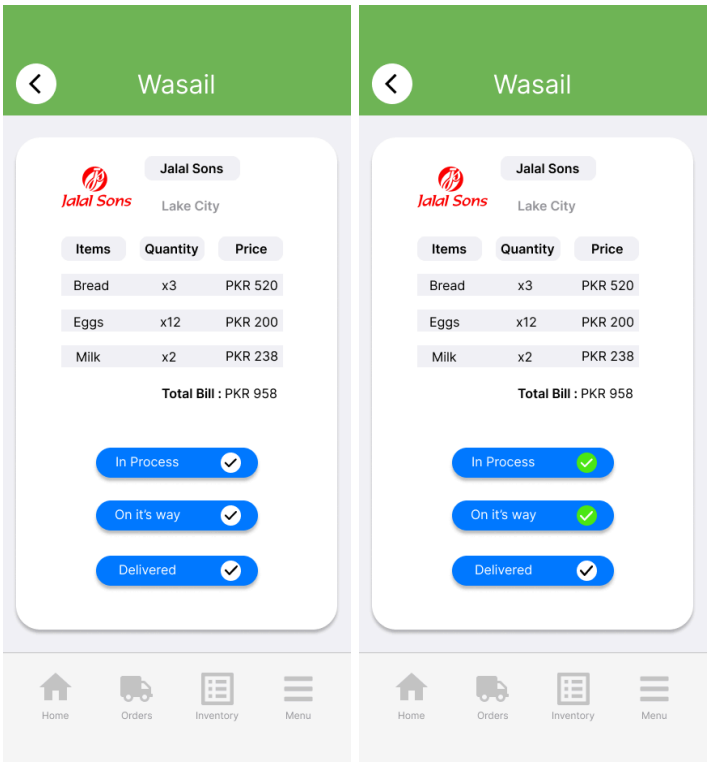


Figure 5.47 & 5.48

FR3.24: Edit Profile

- **Description**: The system should allow the user to edit their profile details
- **Actor**: Vendor
- **Precondition**: The user is on their own profile.
- **Postcondition**: The user's profile is edited.
- **Details**:
    1. The system displays the profile to the user
    2. The user can edit the account details including name, store name, and address.
    3. The system updates the information in the database

FR3.25: Restrict Product Duplication

- **Description**: The system should restrict duplication of products.
- **Actor**: Vendor
- **Precondition**: The user is attempting to add an already existing product to their inventory.
- **Postcondition**: The system prevents the addition of duplicate products and alerts the user.
- **Details**:
    1. The user tries to add a product with the same name or category as an existing product in their inventory.
    2. The system will alert the user that the product already exists in their inventory and instead offer to update the existing product.
    3. The system does not allow duplication of an already existing product within the inventory.

Admin Portal (FR4)

FR4.1: Add New User

- **Description**: The system should allow the existing user to add a new user (Figure 6.1).
- **Actor**: Admin
- **Precondition**: The existing user is logged in.
- **Postcondition**: The existing user is directed to the add details page.
- **Details**:
    1. The system shall display the option to add a new user to the existing user.
    2. After selecting the option to add a new user, the system shall direct the user to the add details page.

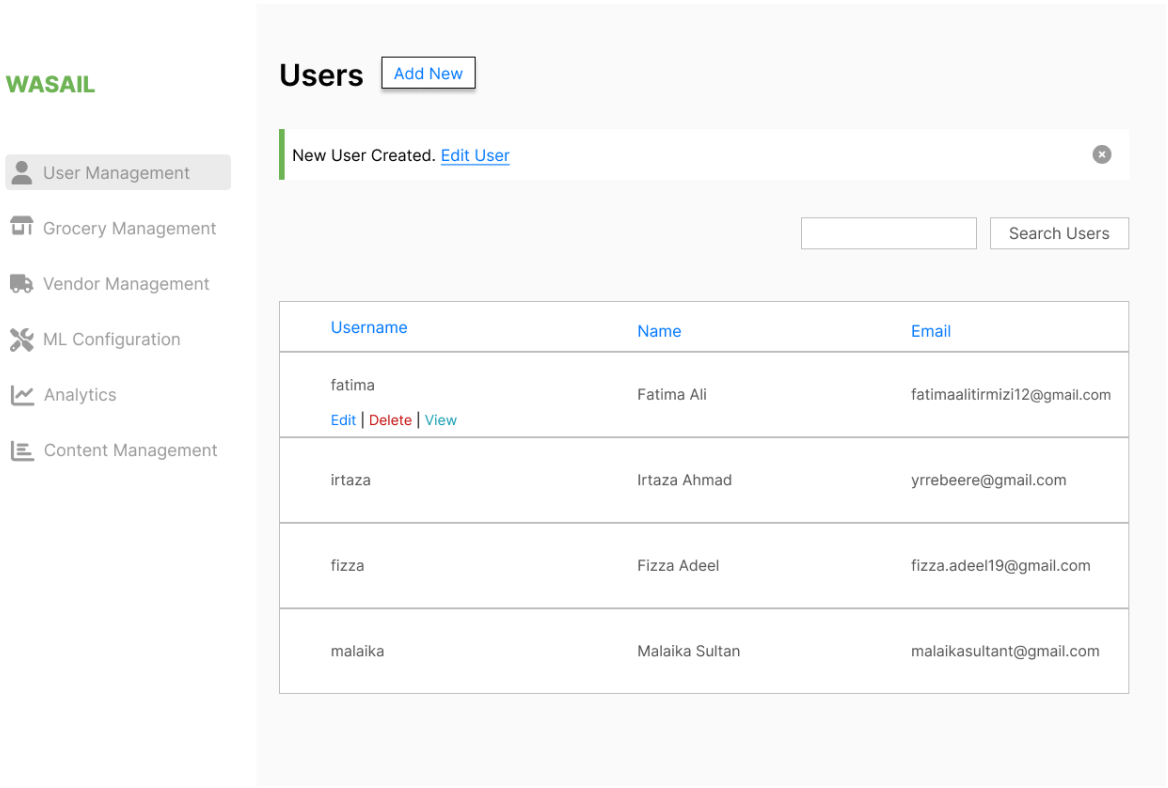

WASAIL
User Management
Grocery Management
Vendor Management
ML Configuration
Analytics
Content Management
Users
Add New
New User Created. Edit User
Search Users
Username
Name
Email
fatima
Edit | Delete | View
Fatima Ali
fatimaalitirmizi12@gmail.com
irtaza
Irtaza Ahmad
yrrebeere@gmail.com
fizza
Fizza Adeel
fizza.adeel19@gmail.com
malaika
Malaika Sultan
malaikasultant@gmail.com

Figure 6.1

FR4.2: Add New User Details

- **Description**: The system should allow the existing user to add details of the new user (Figure 6.2).
- **Actor**: Admin
- **Precondition**: The existing user is on the add details page.
- **Postcondition**: A new user is added to the Admin Portal.
- **Details**:
    1. The existing user adds details of the new user which includes username, email, first name, last name, and password.
    2. The user selects the add new user option after adding the details.
    3. The system shall notify the new user via email.
    4. The system shall add the new user to the database.

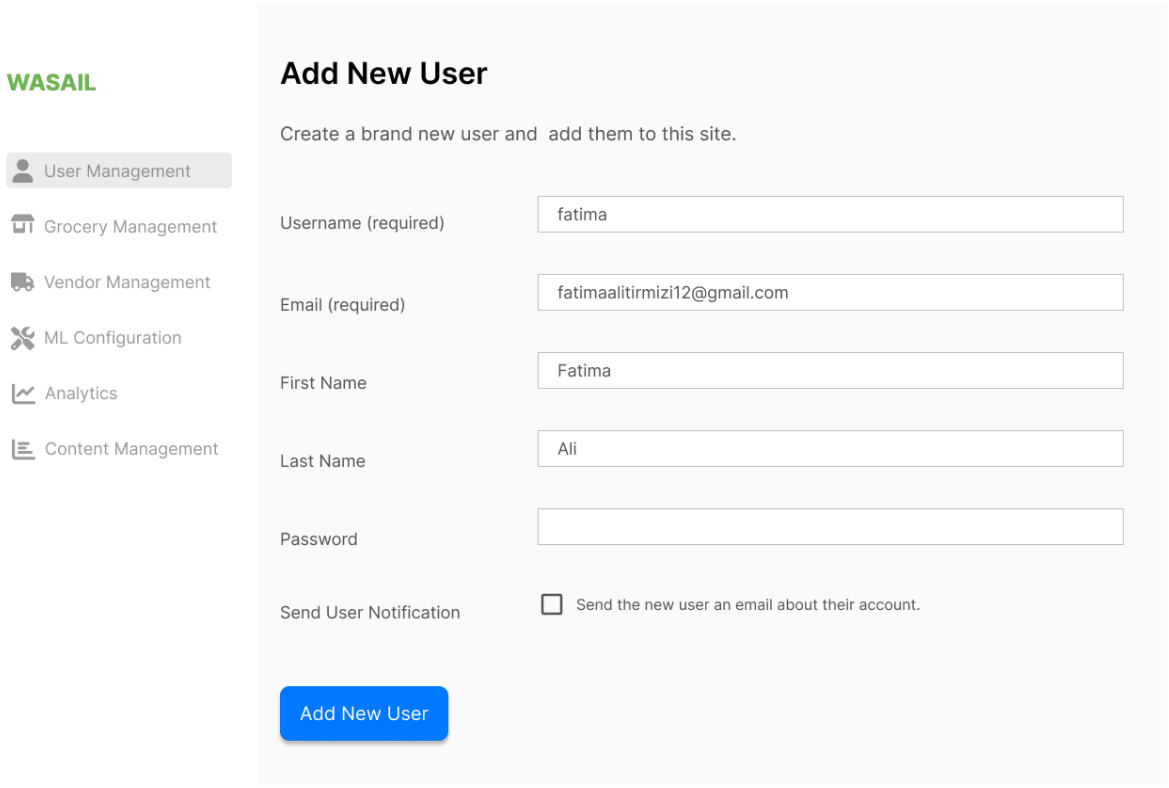


Figure 6.2

FR4.3: User Login

- **Description**: The system should allow the user to login using their credentials (Figure 6.3).
- **Actor**: Admin
- **Precondition**: The user is not logged in.
- **Postcondition**: The user is logged in.
- **Details**:
    1. User provides their valid email and password.
    2. The system validates the user's credentials.
    3. Upon successful validation, the user is granted access.

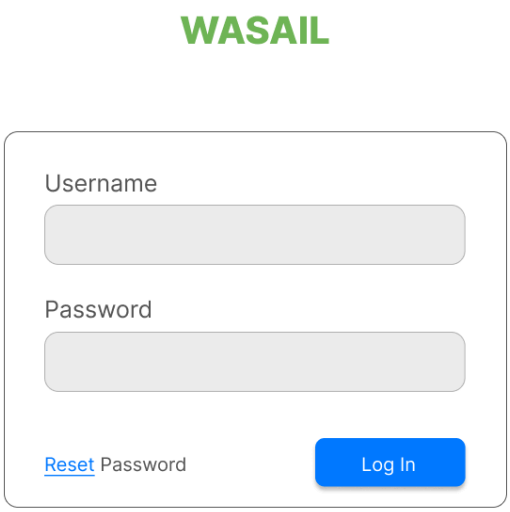


Figure 6.3

FR4.4: Edit User Profile

- **Description**: The system should allow the user to edit their profile (Figure 6.4).
- **Actor**: Admin
- **Precondition**: The user is logged in.
- **Postcondition**: The user's profile has been edited.
- **Details**:
    1. The system shall display the option to edit the profile.
    2. The user can edit their credentials including username, email, first name, last name, and password.
    3. After editing the user shall save the updated version.
    4. The system shall update the database.

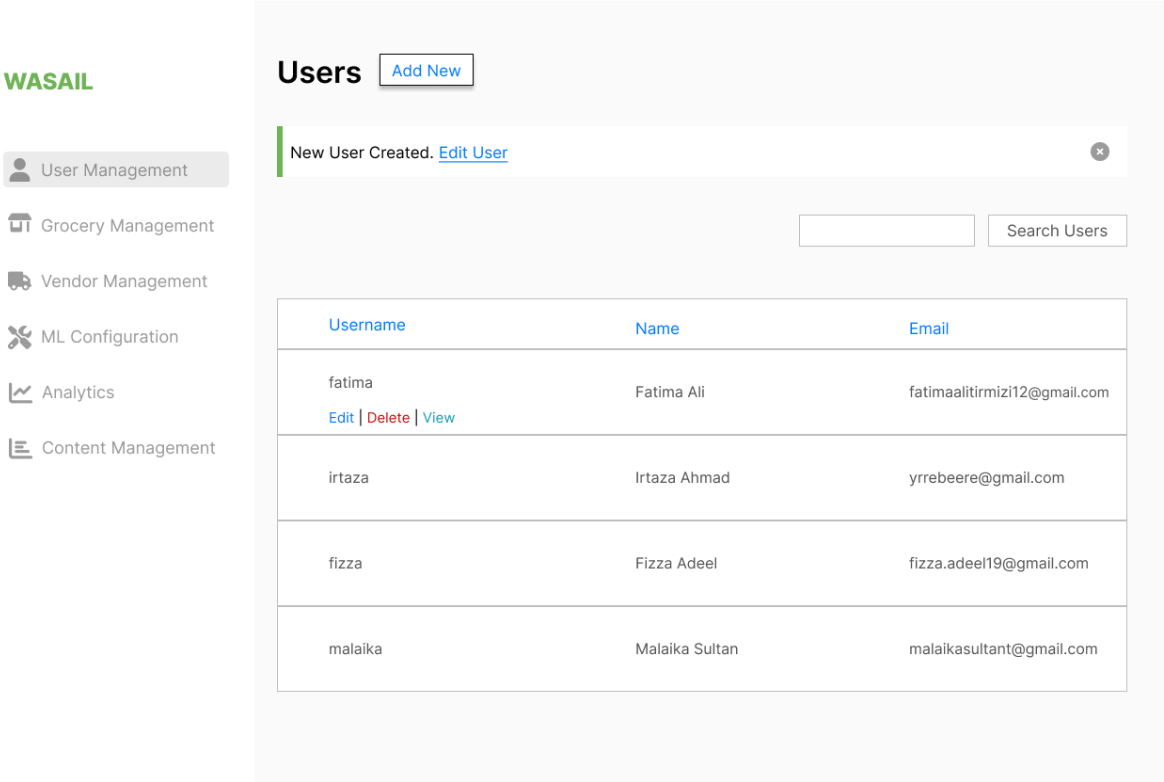


Figure 6.4

FR4.5: Delete User Profile

- **Description**: The system should allow the user to delete their profile (Figure 6.5).
- **Actor**: Admin
- **Precondition**: The user is logged in.
- **Postcondition**: The user's profile has been deleted.
- **Details**:
    1. The system shall display the option to delete the profile.
    2. After successful deletion, the system shall remove it from the database.

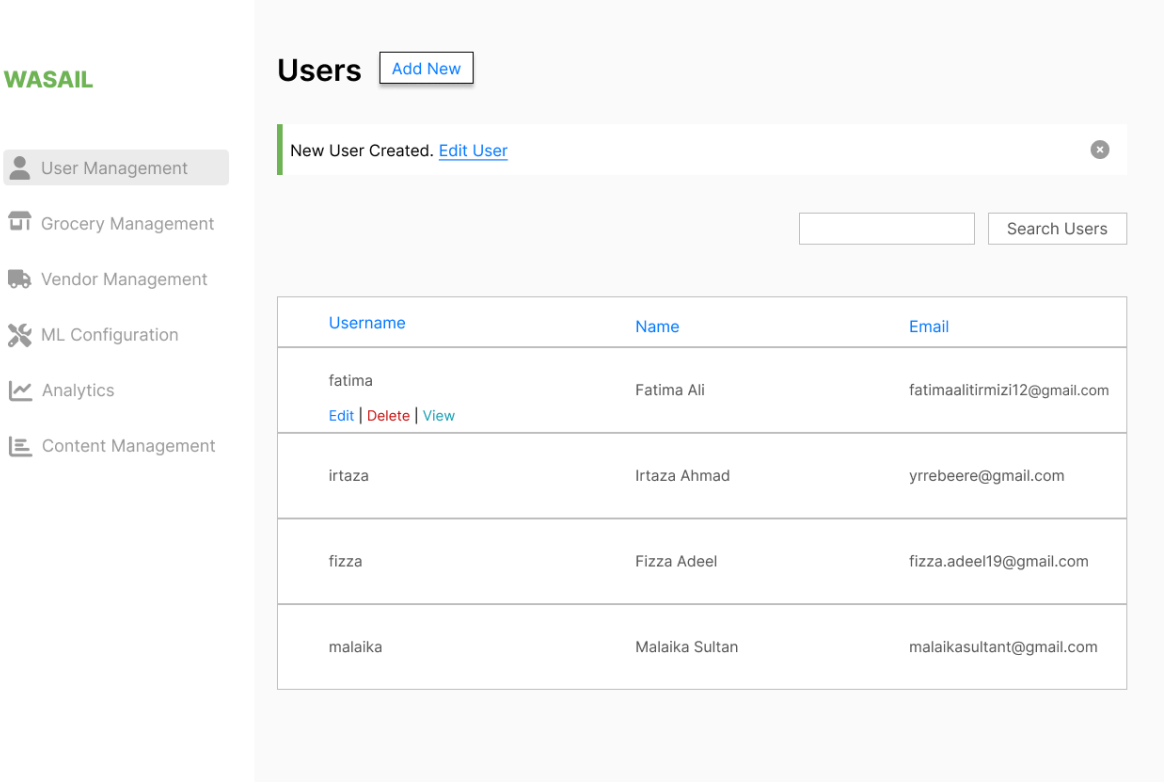


Figure 6.5

FR4.6: View Grocery Store's Profile

- **Description**: The system should allow the user to view the grocery store's profile (Figure 6.6).
- **Actor**: Admin
- **Precondition**: The user is logged in.
- **Postcondition**: The system displays the grocery store's profile.
- **Details**:
    1. The user can select the grocery store's profile to view it.
    2. The system retrieves the grocery store's information including name, shop name, mobile number, address, and location.

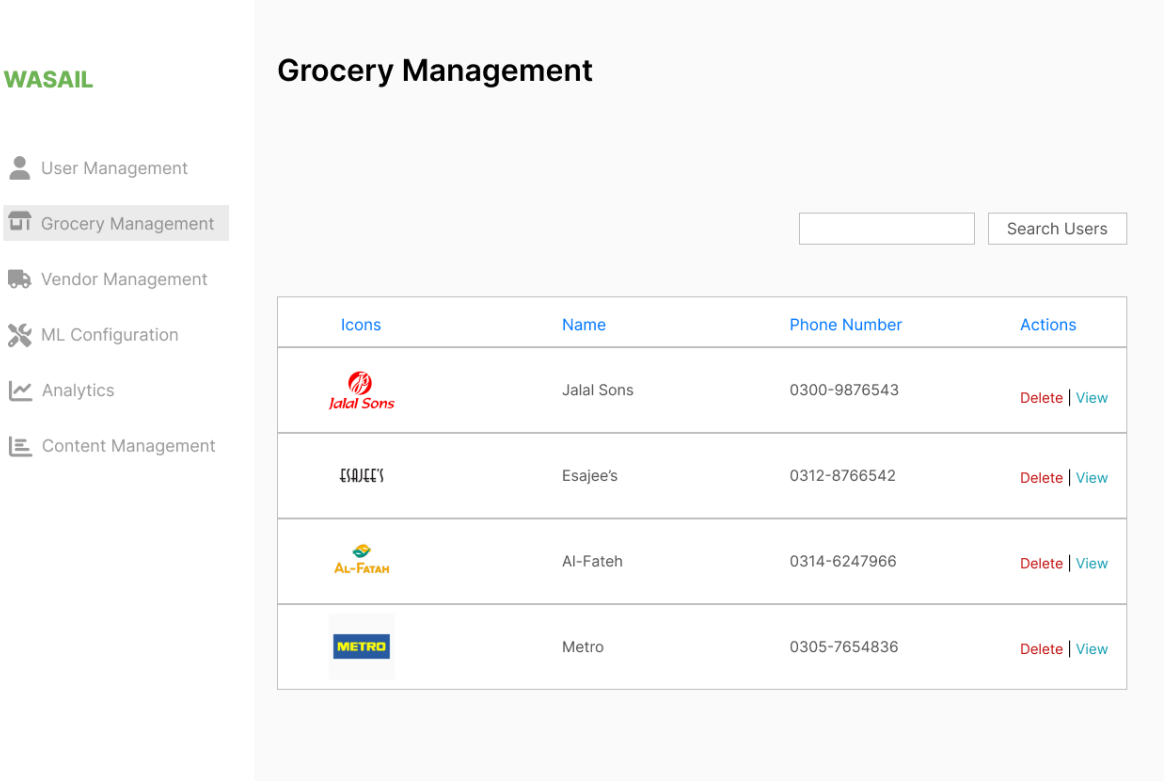


Figure 6.6

FR4.7: Disable Grocery Store's Profile

- **Description**: The system should allow the user to delete the grocery store's profile (Figure 6.7).
- **Actor**: Admin
- **Precondition**: The user is logged in.
- **Postcondition**: The system has disabled the grocery store's profile.
- **Details**:
    1. The user can select the grocery store's profile to disable it.
    2. The system disables the grocery store's profile.

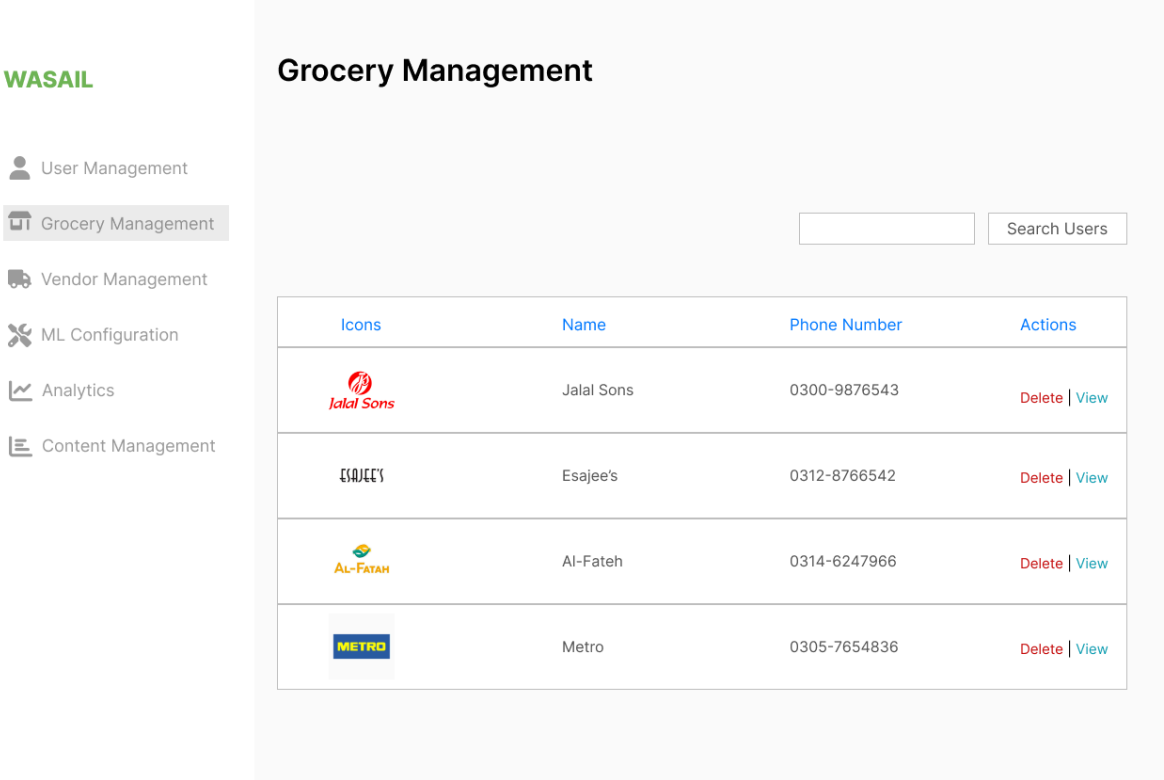


Figure 6.7

FR4.8: View Vendor’s Profile

- **Description**: The system should allow the user to view the vendor's profile (Figure 6.8).
- **Actor**: Admin
- **Precondition**: The user is logged in.
- **Postcondition**: The system displays the vendor’s profile.
- **Details**:
    1. The user can select the vendor’s profile in order to view it.
    2. The system retrieves the vendor’s information including name, mobile number, areas in which they deliver.

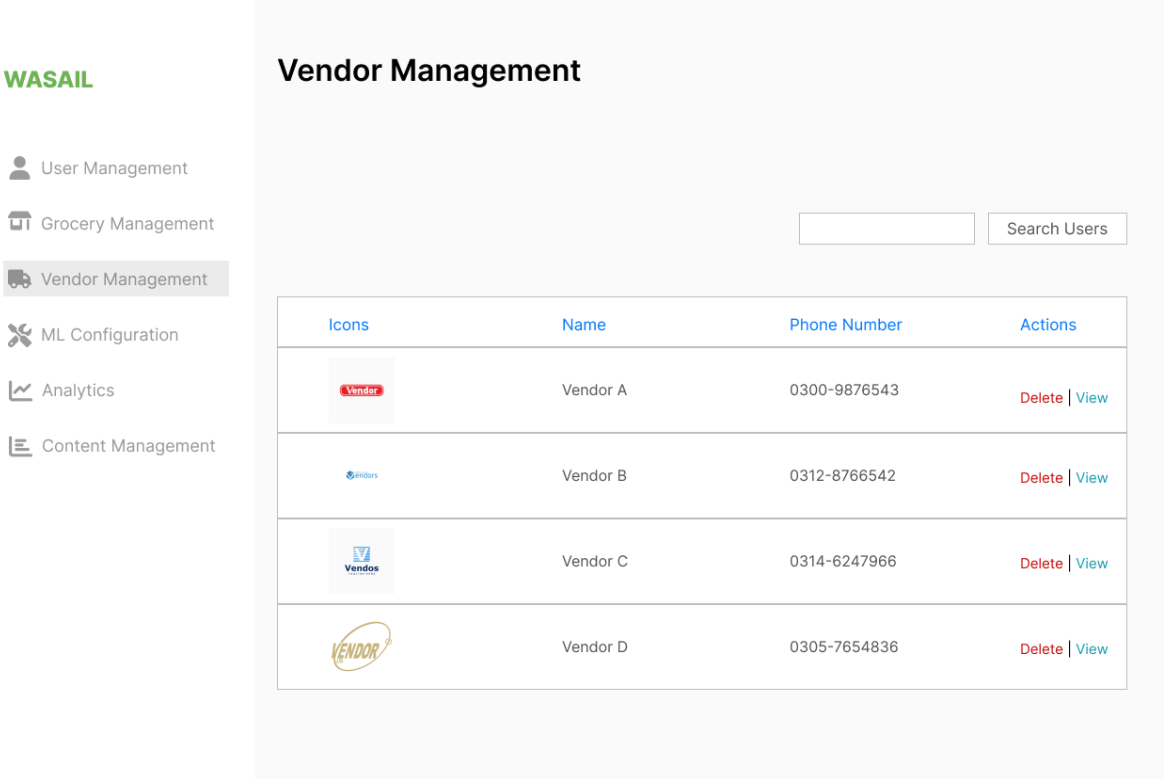


Figure 6.8

FR4.9: Disable Vendor’s Profile

- **Description**: The system should allow the user to delete the vendor's profile (Figure 6.9).
- **Actor**: Admin
- **Precondition**: The user is logged in.
- **Postcondition**: The system has disabled the vendor’s profile.
- **Details**:
    1. The user can select the vendor’s profile to disable it.
    2. The system disables the vendor’s profile.

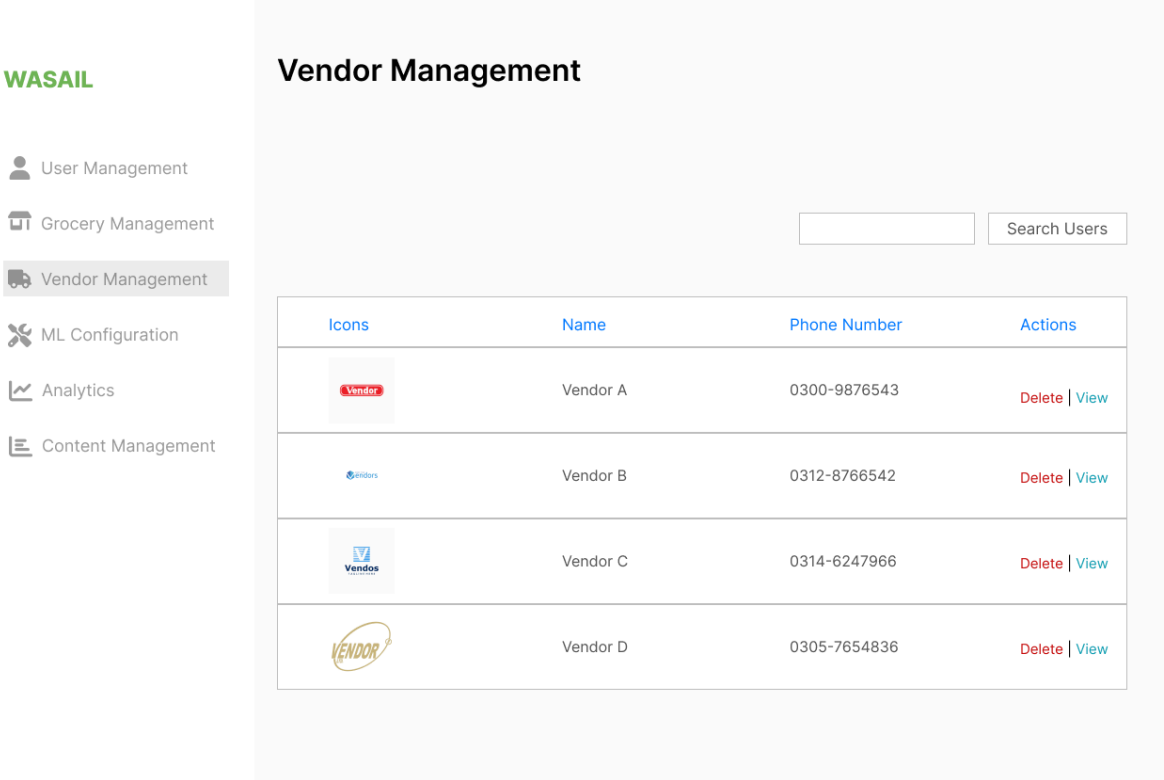


Figure 6.9

FR4.10: Select Grocery Store's ML Model

- **Description**: The system should allow the user to select the ML model for the grocery store (Figure 6.10).
- **Actor**: Admin
- **Precondition**: The user is logged in.
- **Postcondition**: The system has allowed the user to select the ML model for the grocery store.
- **Details**:
    1. The user can select a ML model from a list generated by the system.
    2. The system will save the option selected by the user.

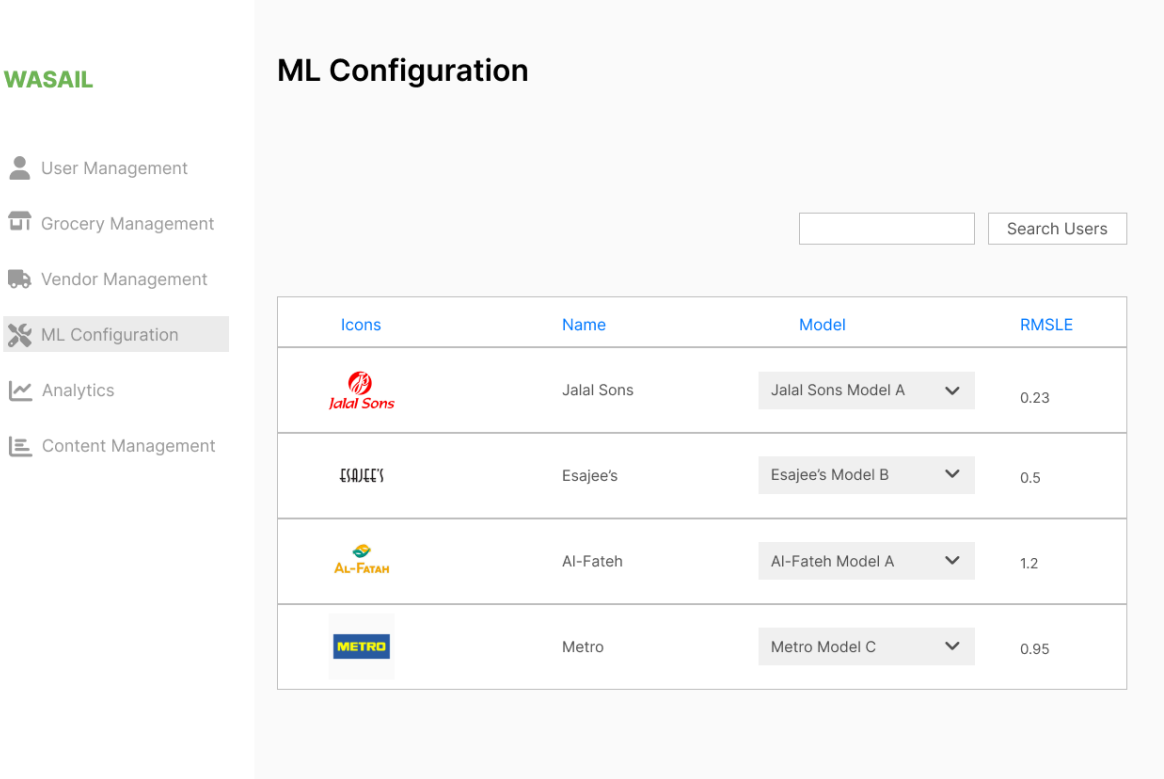


Figure 6.10

FR4.11: Display RMSLE value

- **Description**: The system should display the RMSLE value to the user (Figure 6.11).
- **Actor**: Admin
- **Precondition**: The user has selected a ML model.
- **Postcondition**: The system has displayed the RMSLE value.
- **Details**:
    1. The system calculates the RMSLE value against the selected ML model.
    2. The system displays the RMSLE value for the selected ML model.

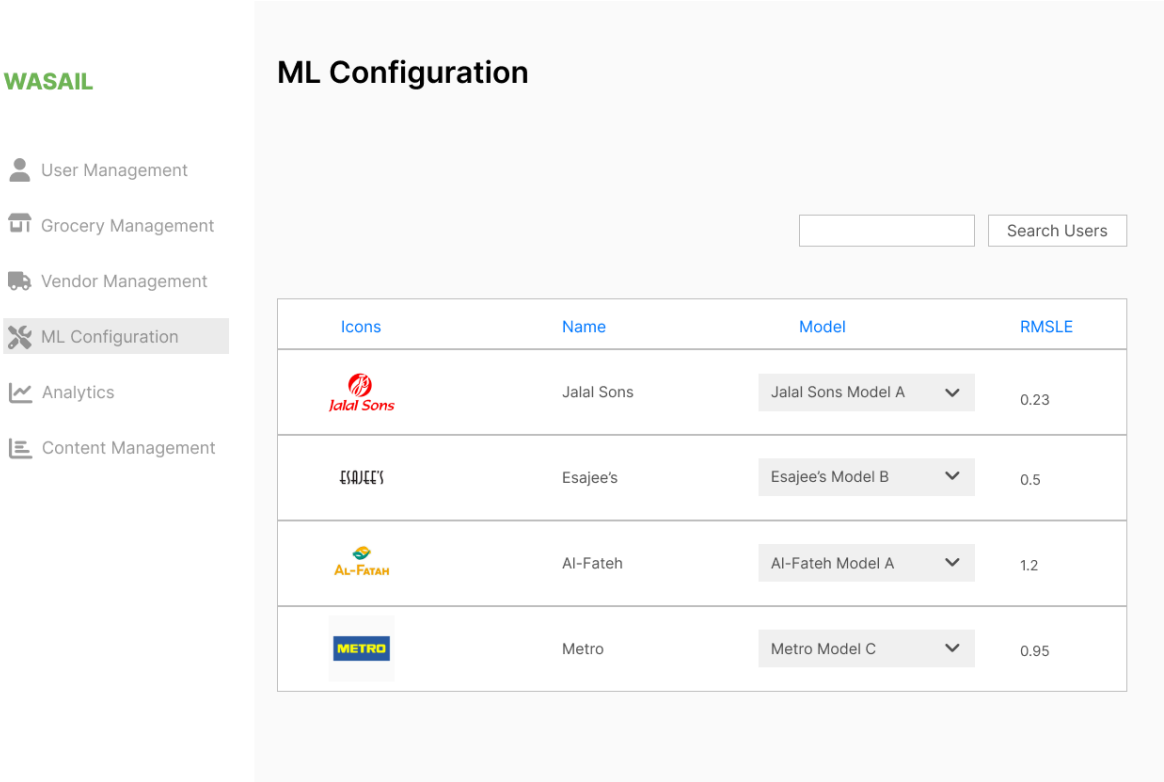


Figure 6.11

FR4.12: Display Analytics

- **Description**: The system should display the analytics to the user (Figure 6.12).
- **Actor**: Admin
- **Precondition**: The user is logged in and on the analytics page.
- **Postcondition**: The system has displayed the analytics to the user.
- **Details**:
    1. The systems shall display the option to view analytics.
    2. Once selected, the system shall display different analytics including the total number of groceries stores that have registered, the total number of vendors that have registered, the total number of SKUs that are present.
    3. The user shall be able to view these analytics.

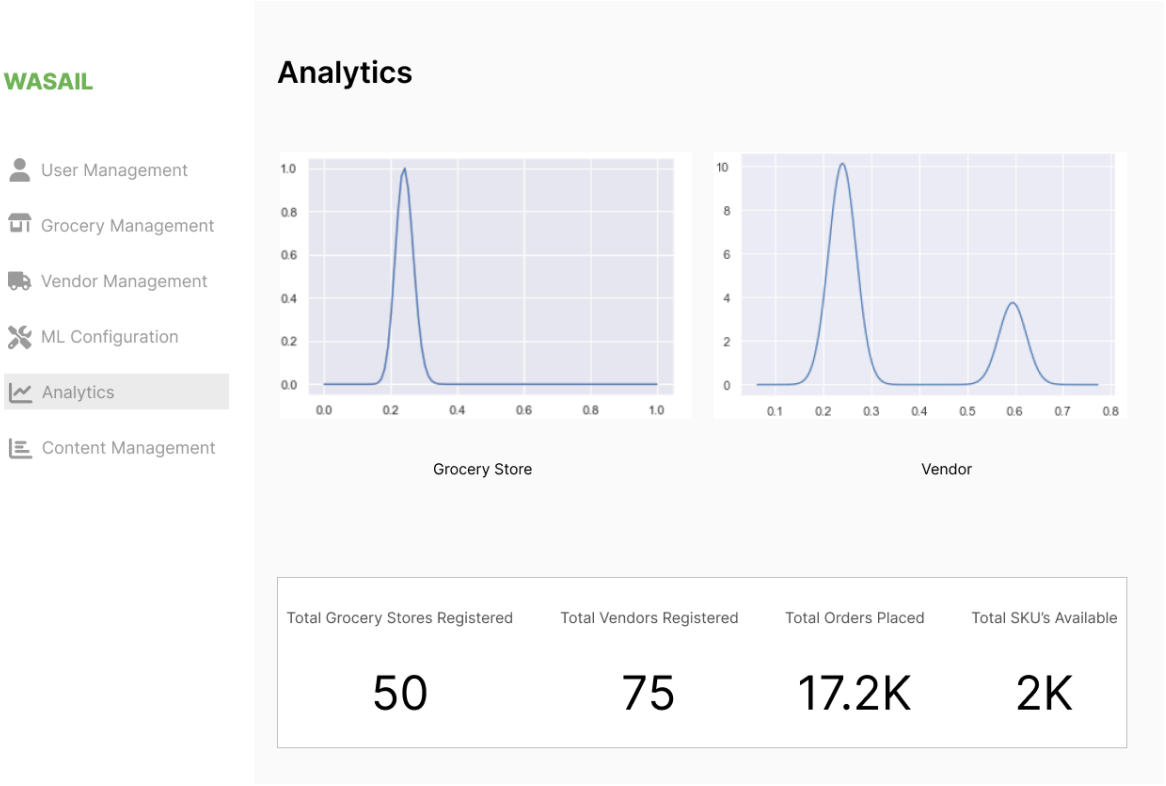


Figure 6.12

FR4.13: Add Category

- **Description**: The system should allow the user to add a category of products (Figure 6.13).
- **Actor**: Admin
- **Precondition**: The user is logged in and on the products listings page.
- **Postcondition**: The user has added a category.
- **Details**:
    1. The system shall display an option to add a category of the products for the database (through which the vendors would be able to add products to their own inventory)
    2. The user can add the category name for example dairy, vegetables, condiments, meat etc.
    3. The system shall save them in the database.

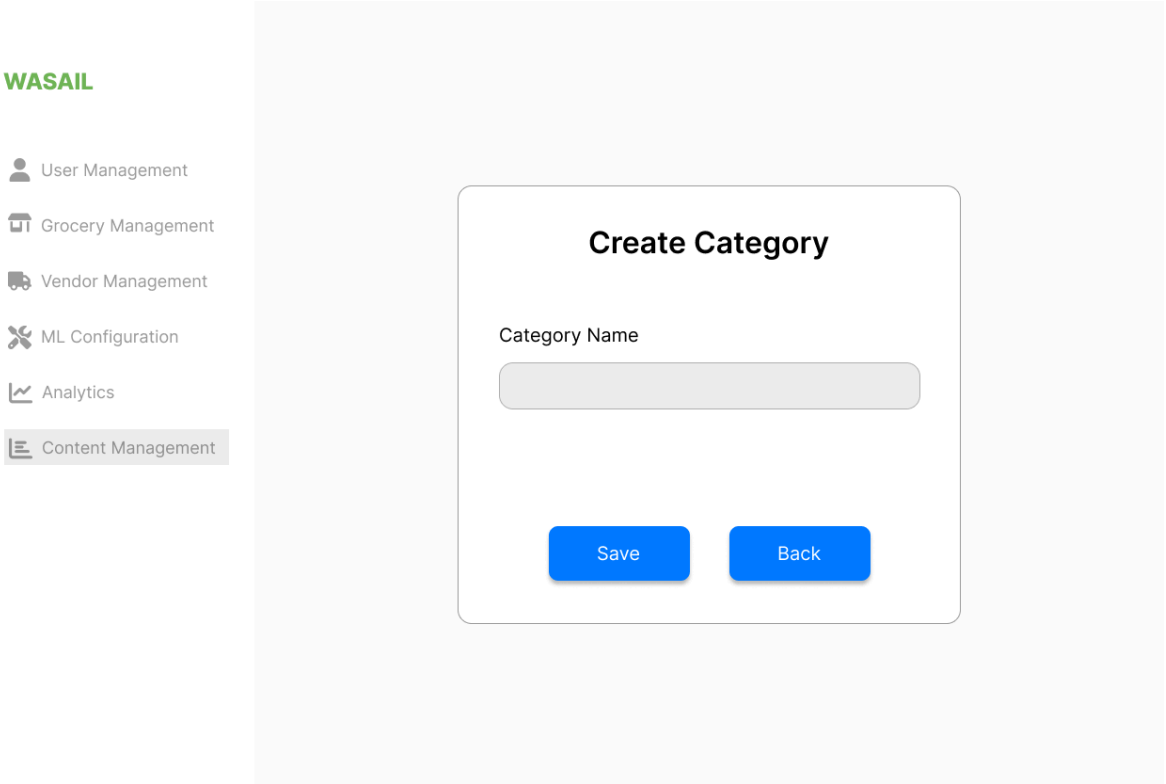


Figure 6.13

FR4.14: Update Category

- **Description**: The system should allow the user to update the category (Figure 6.14).
- **Actor**: Admin
- **Precondition**: The user is logged in and on the products listings page.
- **Postcondition**: The user has updated the category.
- **Details**:
    1. The system shall give the option to update the category name.
    2. The user can select the option and update the category name.
    3. The system shall save the changes in the database.

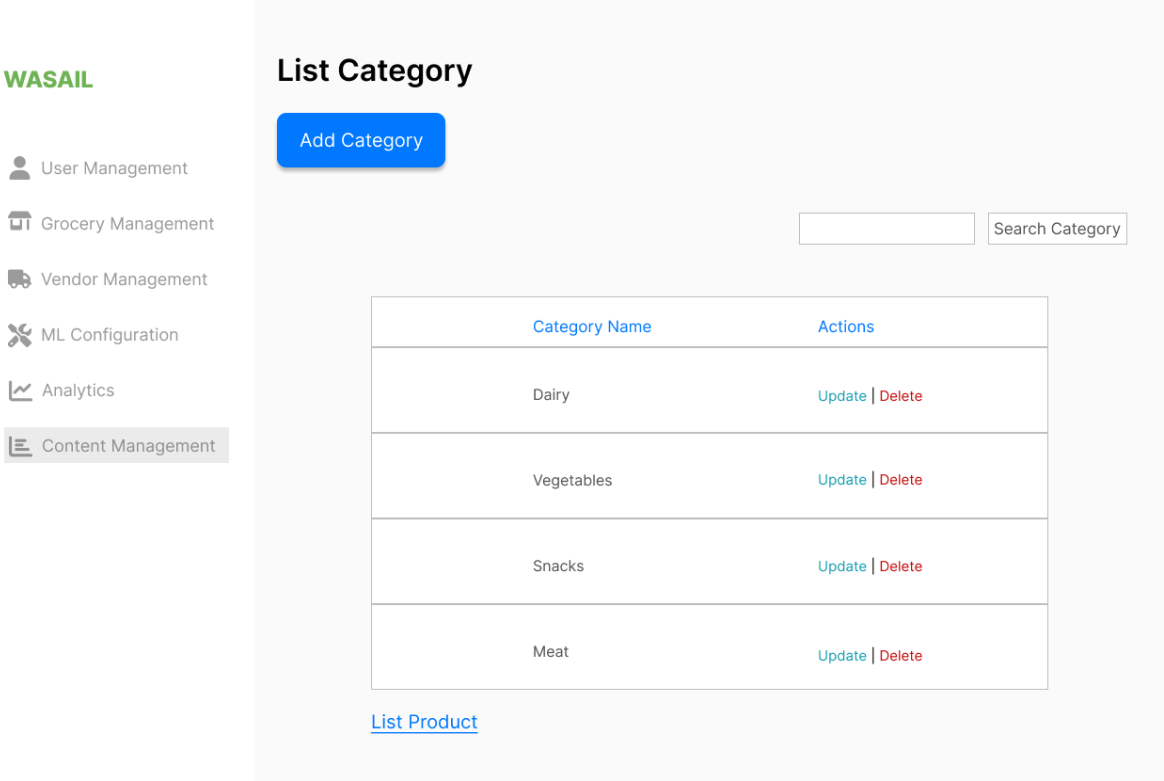


Figure 6.14

FR4.15: Delete Category

- **Description**: The system should allow the user to delete the category (Figure 6.15).
- **Actor**: Admin
- **Precondition**: The user is logged in and on the products listings page.
- **Postcondition**: The user has deleted the category.
- **Details**:
    1. The system shall give the option to delete the category.
    2. Once the user has deleted the category, the system shall remove the category from the database as well.

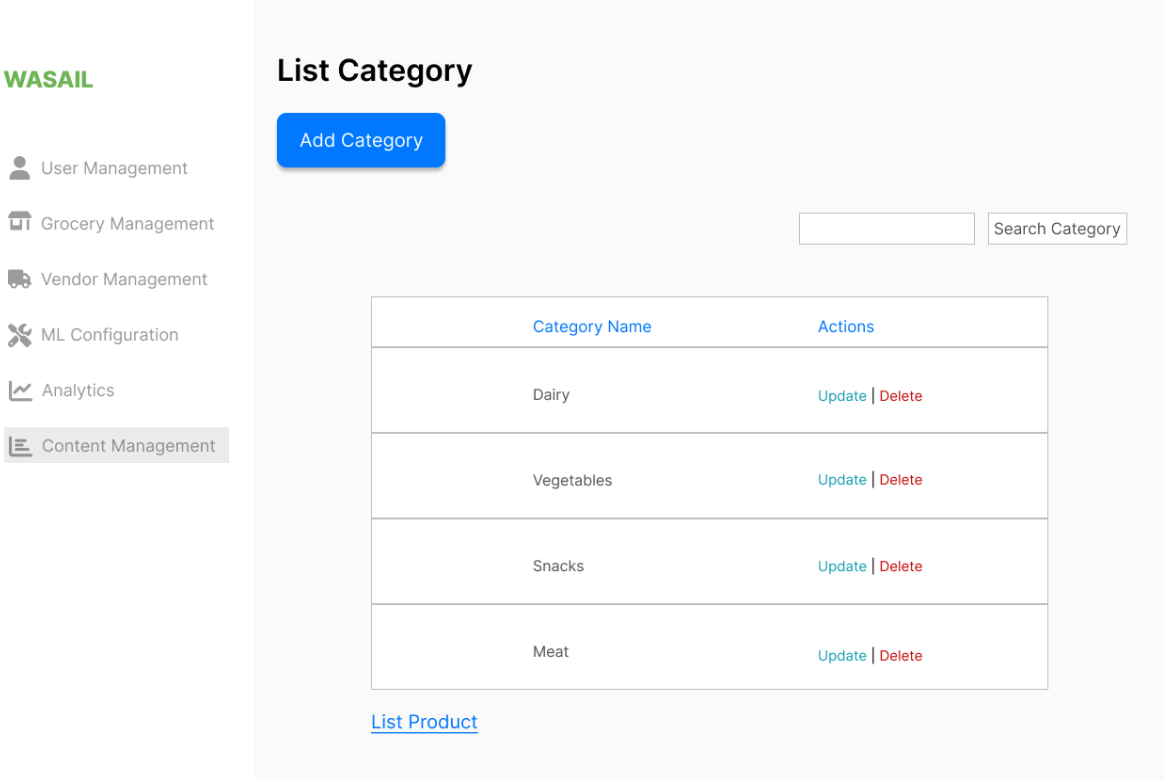


Figure 6.15

FR4.16: Add Product

- **Description:** The system should allow the user to add product details (Figure 6.16).
- **Actor**: Admin
- **Precondition**: The user is logged in and on the products listings page.
- **Postcondition**: The user has added a product.
- **Details**:
    1. The system shall display an option to add a product to the database (through which the vendors would be able to add products to their own inventory)
    2. The user can add the product name, can upload an image, and can select from the category.
    3. The system shall save them in the database.

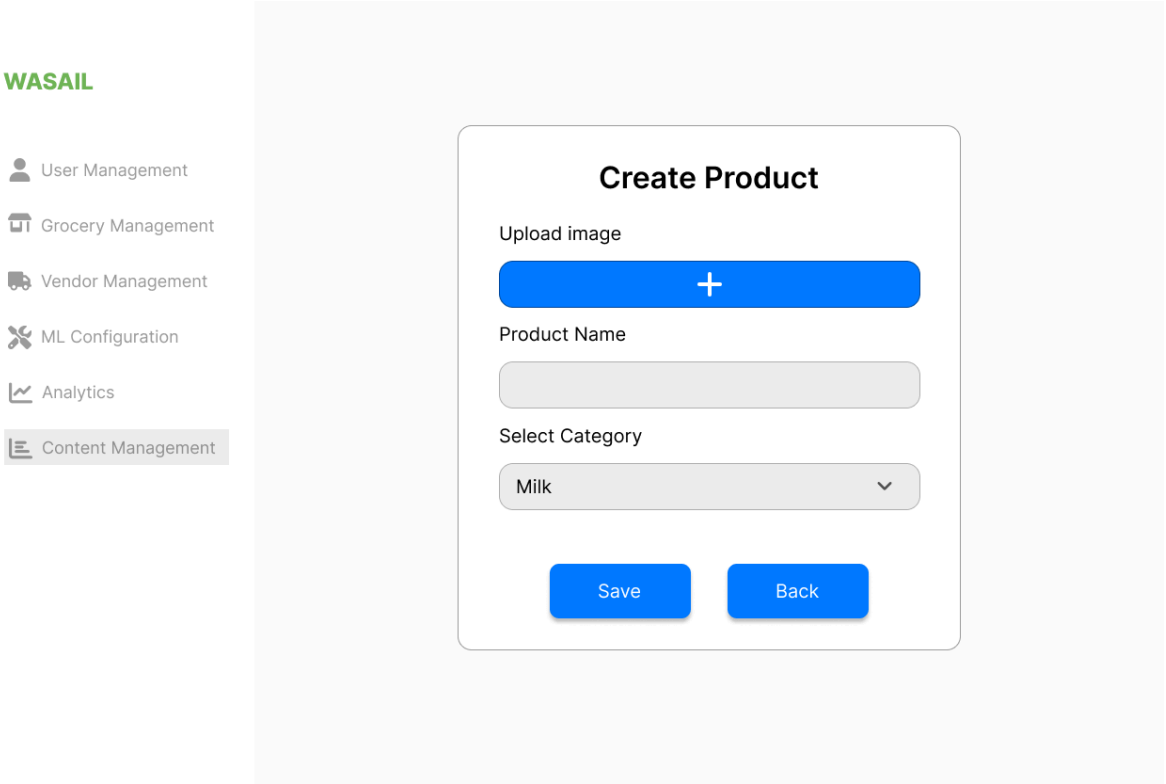


Figure 6.16

FR4.17: Update Product

- **Description**: The system should allow the user to update the product (Figure 6.17).
- **Actor**: Admin
- **Precondition**: The user is logged in and on the products listings page.
- **Postcondition**: The user has updated the product.
- **Details**:
    1. The system shall give the option to update the product name, the picture or the category.
    2. The user can select the option and update the product name, the picture or the category.
    3. The system shall save the changes in the database.

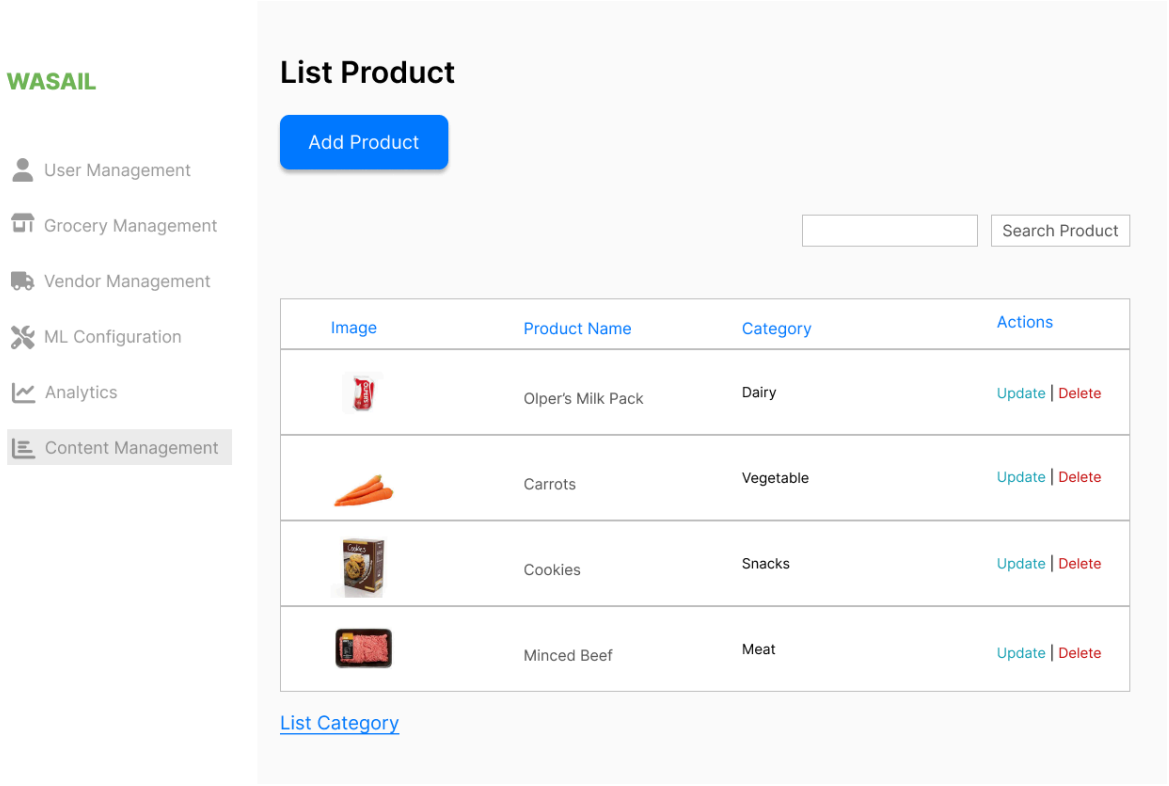


Figure 6.17

FR4.18: Delete Product

- **Description**: The system should allow the user to delete the product (Figure 6.18).
- **Actor**: Admin
- **Precondition**: The user is logged in and on the products listings page.
- **Postcondition**: The user has deleted the product.
- **Details**:
    3. The system shall give the option to delete the product.
    4. Once the user has deleted the product, the system shall remove the product from the database as well.

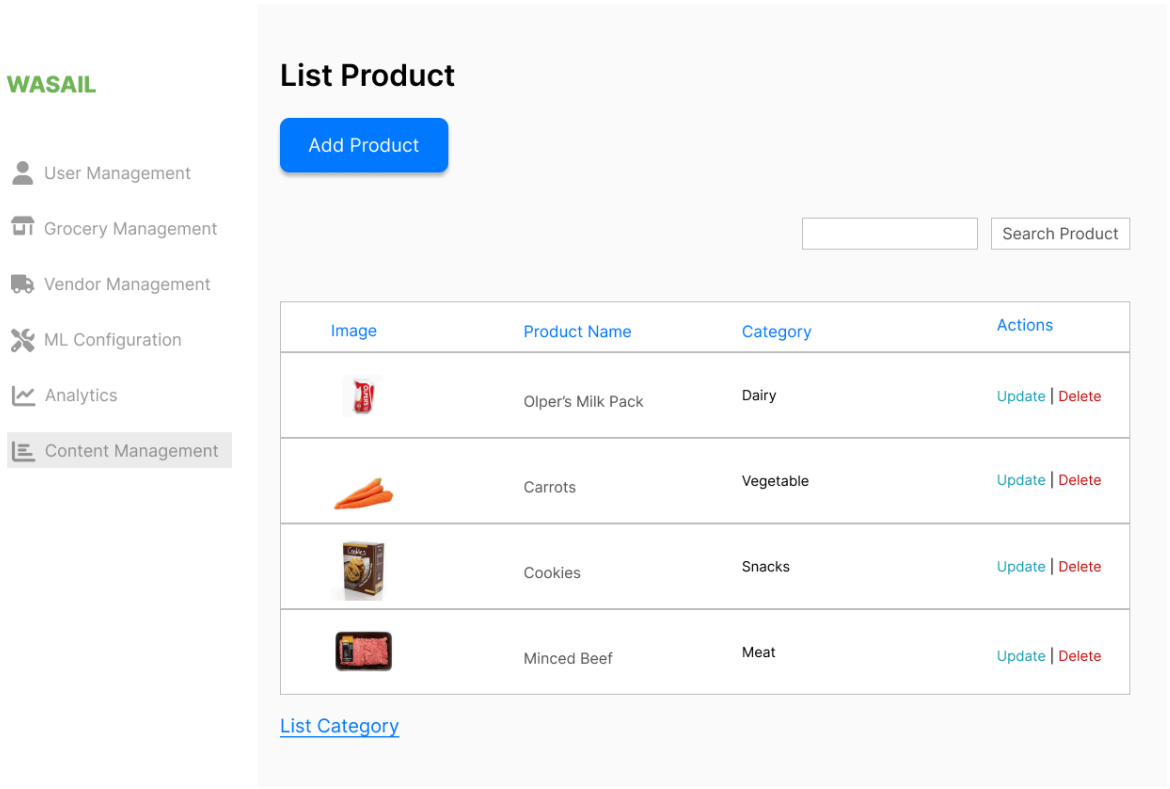


Figure 6.18

FR4.19: View Grocery Store Count

- **Description**: The system should allow the user to view the total number of grocery stores that have been registered.
- **Actor**: Admin
- **Precondition**: The user is logged in and on the analytics page.
- **Postcondition**: The system displays the total grocery store count.
- **Details**:
    1. The system displays the total number of grocery stores that have been registered.
    2. The user can view the total count.

FR4.20: View Vendor Count

- **Description**: The system should allow the user to view the total number of vendors that have been registered.
- **Actor**: Admin
- **Precondition**: The user is logged in and on the analytics page.
- **Postcondition**: The system displays the total vendor count.
- **Details**:
    1. The system displays the total number of vendors that have been registered.
    2. The user can view the total count.

FR4.21: View Product Count

- **Description**: The system should allow the user to view the total number of products in the database.
- **Actor**: Admin
- **Precondition**: The user is logged in and on the analytics page.
- **Postcondition**: The system displays the total product count.
- **Details**:
    1. The system displays the total number of products in the database.
    2. The user can view the total count.

FR4.22: View Category Count

- **Description**: The system should allow the user to view the total number of categories in the database.
- **Actor**: Admin
- **Precondition**: The user is logged in and on the analytics page.
- **Postcondition**: The system displays the total category count.
- **Details**:
    3. The system displays the total number of categories in the database.
    4. The user can view the total count.

FR4.23: Search Admin

- **Description**: The system should allow the user to search the admin on the admin page.
- **Actor**: Admin
- **Precondition**: The user is logged in and is on the admin page.
- **Postcondition**: The system displays the information.
- **Details**:
    1. The user can enter the username of the admin in the search bar.

2. The system retrieves the information about the admin that was searched.
3. The system displays the information.

FR4.24: Search Grocery Store

- **Description**: The system should allow the user to search the grocery store on the grocery store page.
- **Actor**: Admin
- **Precondition**: The user is logged in and is on the grocery store page.
- **Postcondition**: The system retrieves the information.
- **Details**:
    1. The user can enter the name of the grocery store in the search bar.
    2. The system retrieves the information about the grocery store that was searched.
    3. The system displays the information.

FR4.25: Search Vendor

- **Description**: The system should allow the user to search the vendor on the vendor page.
- **Actor**: Admin
- **Precondition**: The user is logged in and is on the vendor page.
- **Postcondition**: The system retrieves the information.
- **Details**:
    1. The user can enter the name of the vendor in the search bar.
    2. The system retrieves the information about the vendor that was searched.
    3. The system displays the information.

FR4.26: Search Product

- **Description**: The system should allow the user to search the product on the product page.
- **Actor**: Admin
- **Precondition**: The user is logged in and is on the product page.
- **Postcondition**: The system retrieves the information.
- **Details**:
    1. The user can enter the name of the product in the search bar.
    2. The system retrieves the information about the product that was searched.
    3. The system displays the information.

FR4.27: Search Category

- **Description**: The system should allow the user to search the category on the category page.
- **Actor**: Admin

- **Precondition**: The user is logged in and is on the category page.
- **Postcondition**: The system retrieves the information.
- **Details**:
    1. The user can enter the name of the category in the search bar.
    2. The system retrieves the information about the category that was searched.
    3. The system displays the information.

Non-Functional Requirements

- **Performance**
    - **Scalability**: The system should be able to handle a growing number of users, both grocery stores and vendors, without a significant decrease in performance.

- **Security**
    - **Access Control**: Role-based access control should be in place to restrict unauthorised access to sensitive functionalities and data.

- **Usability**
    - **Multilingual Support**: The system should support multiple languages (English, Urdu) as per FR1.1 to cater to a diverse user base.
    - **User-Friendly Interface**: The user interface should be intuitive and easy to use, ensuring a seamless experience for both grocery stores and vendors.

- **Data Storage**
    - **Data Capacity**: The system should be capable of handling a large volume of user data, product listings, and order history efficiently.

- **Error Handling**
    - **Error Messages**: Clear and informative error messages should be provided to assist users in troubleshooting issues.

- **Mobile Responsiveness**
    - **Cross-Platform Compatibility**: The system should be accessible and user-friendly on various mobile devices.

Future Improvements

1. Convert “Grocery Store” actors into two actors: “Owner” and “Employee”

2. ‘Roman Urdu’ should be added as an option for language as well.
3. Allow “Vendor” to add multiple people (delivery person) under one account
4. Allow “Site Admins” to have different roles (Admin, Editor, Viewer)
5. To enhance security, the system should send an OTP code to the user's phone number for verification
6. Notifications should be sent to the users to keep them more informed and updated

Design

The design document presented here outlines a comprehensive approach to developing a demand forecasting system for grocery stores and creating an environment to aid the communication between them and the vendors. Leveraging a diverse set of technologies, the system integrates machine learning models with a robust backend infrastructure and a user-friendly front end. The document delves into various components such as development tools, programming languages, data storage, machine learning frameworks, and cloud services. It also provides detailed insights into the system architecture, data design, and the integration of machine learning for demand forecasting. With a focus on feature engineering and model selection, the document showcases the meticulous process involved in preparing datasets, cleaning, and transforming them for accurate predictions.

Development Tools

The following tools were used while developing our project.

Programming Languages

- Python: Python is being used for the machine learning section of the project.
- JavaScript: Javascript is used for the development of backend on NodeJS.
- Dart: Dart is used for developing the mobile application on Flutter.

Machine Learning

- TensorFlow
- PyTorch
- Scikit-Learn
- Prophet (Facebook)
- LightGBM
- N-BEATS
- DeepAR (Amazon)
- Temporal Fusion Transformer (Google)

Cloud Services

- Digital Ocean

Web Development

- Front-End
    - HTML
    - CSS
    - Bootstrap
    - React
- Back-End
    - Node.js
    - Express.js
    - Flask

Data Storage

- MySQL

Object Relational Mapper

- Sequelize

Mobile App Development

- Flutter

Version Control

- GitHub

Project Management

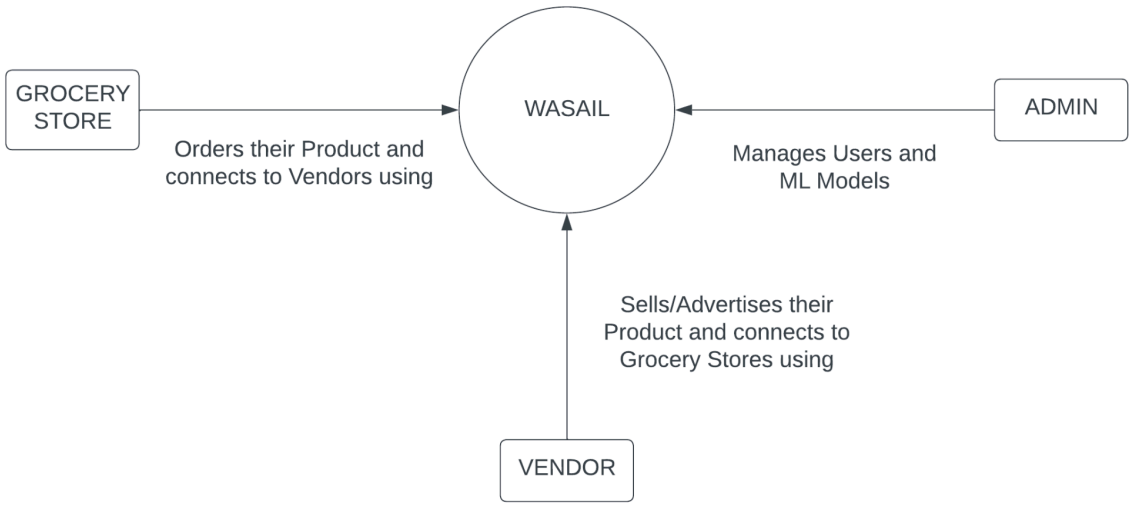
- Jira

Collaboration

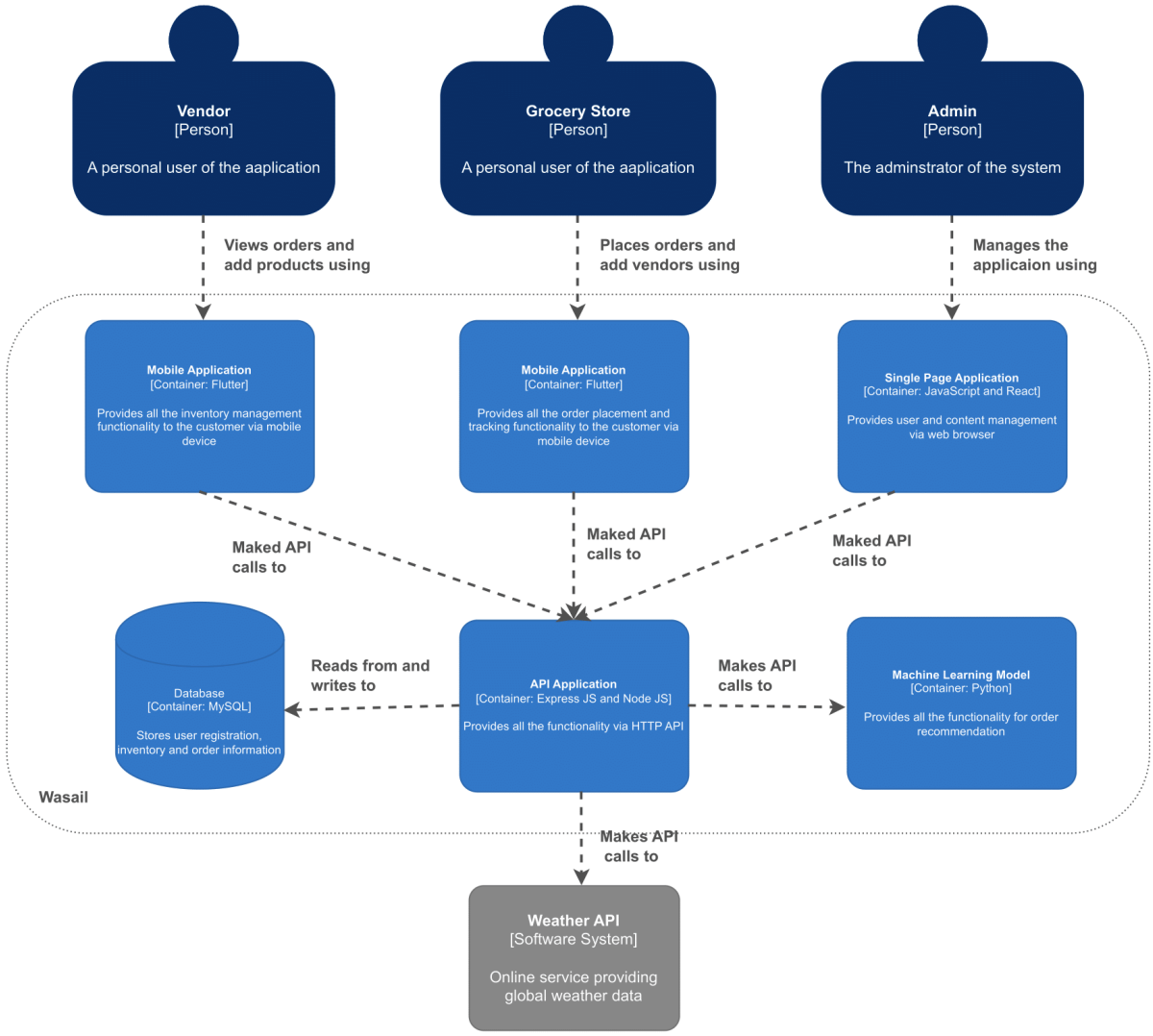
- Discord

System Architecture

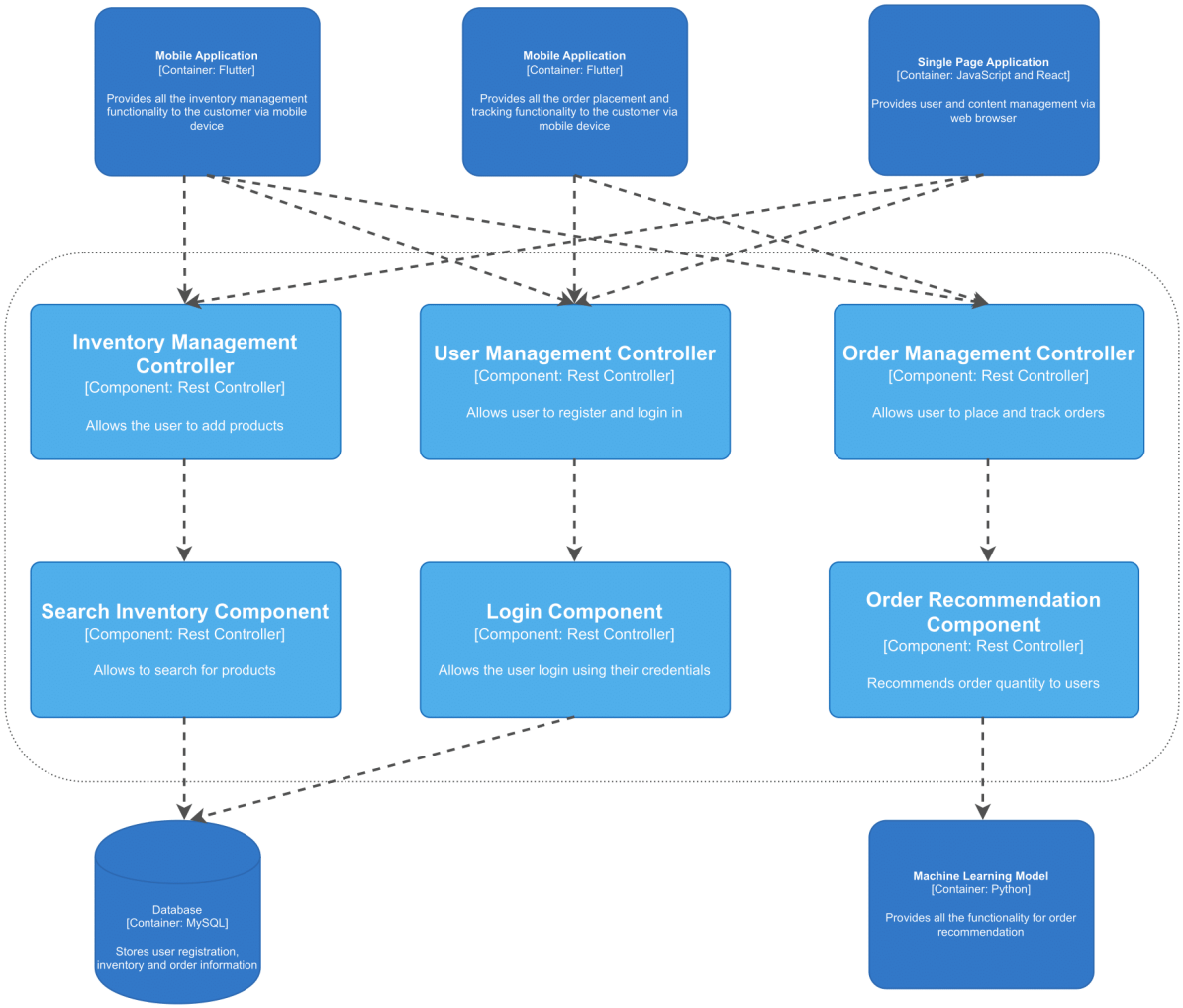
System Context Diagram



Container Diagram

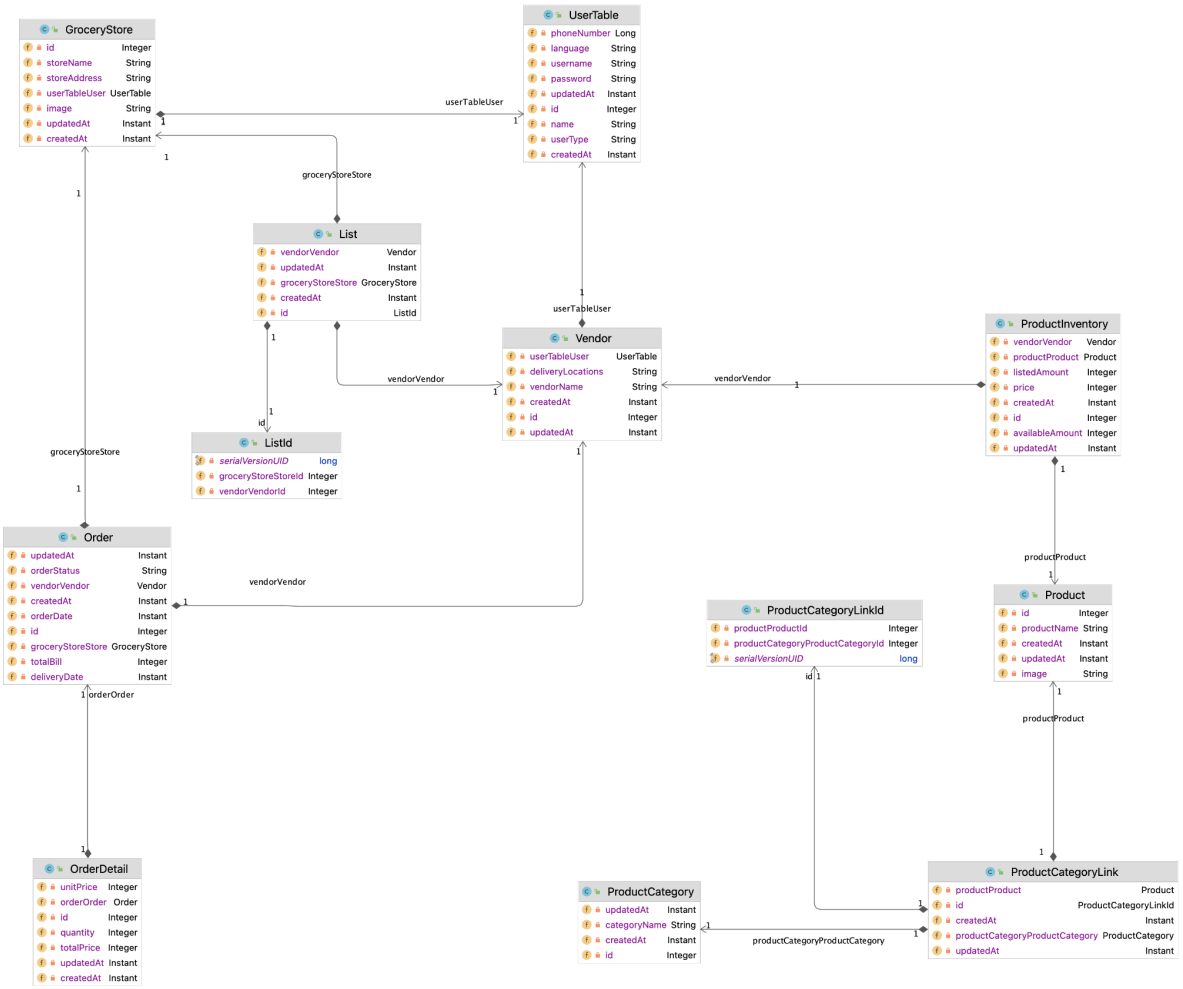


Component Diagram



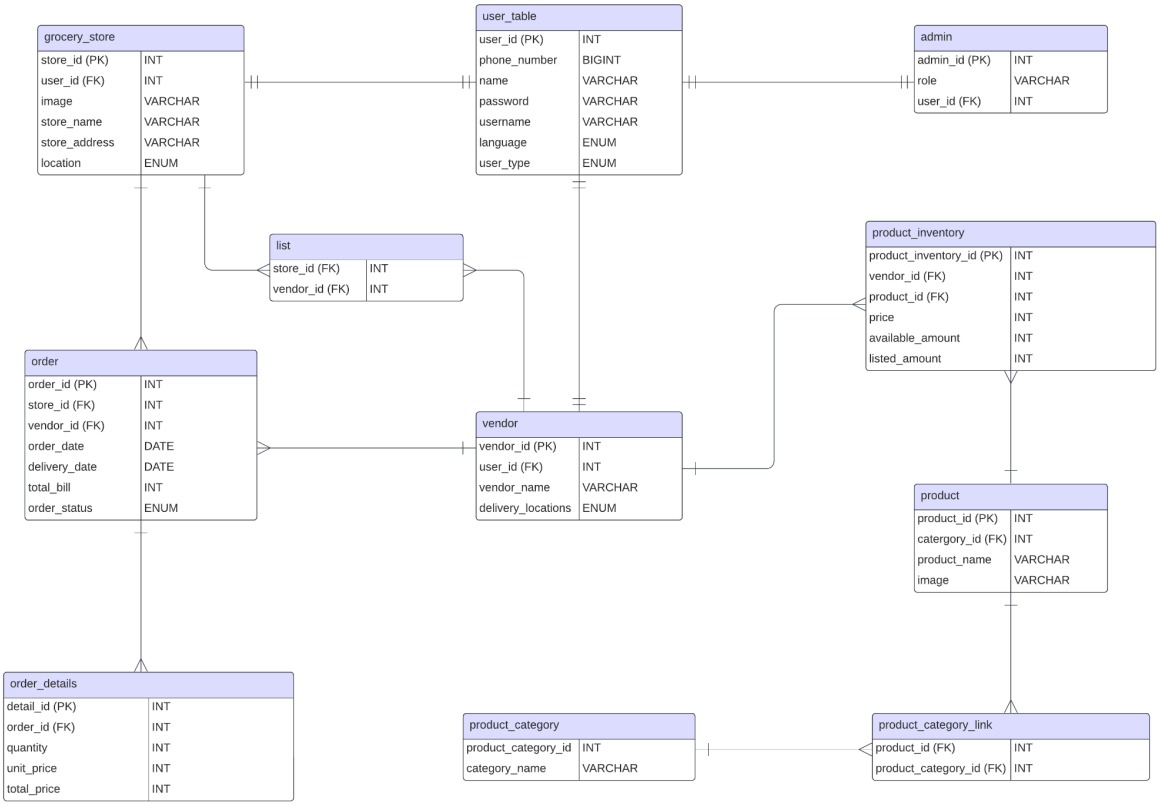
Class Diagram

The class diagram has been automatically generated using JetBrains. It is identical to the database design, mirroring the tables constructed in the database to classes.



Data Design

When designing the database diagram, it was mainly divided into three sections; user management, inventory management, and order management. Under user management, the tables that were created were user table, vendor, grocery store, and admin. All the common attributes including phone number, name, password, user name, language, and user type were in the user table which had a one-to-one relation with the grocery store, vendor, and admin. Since a grocery store and vendor had a many-to-many relationship (one grocery store can have many vendors and vice versa), it required a pivot table (called lists), since a many-to-many relation can not be made in the diagram. Next, under the inventory management section, the tables that were created were product (a table from which the vendor will add products to their own inventory), product inventory (the vendors' own inventory), product category (the table that will contain the categories of the product) and product category link. Similar to the vendor and grocery store table, the product and product category tables also have a many-to-many relation and hence require a pivot table. Lastly, under order management, order and order details were the two tables that were created.

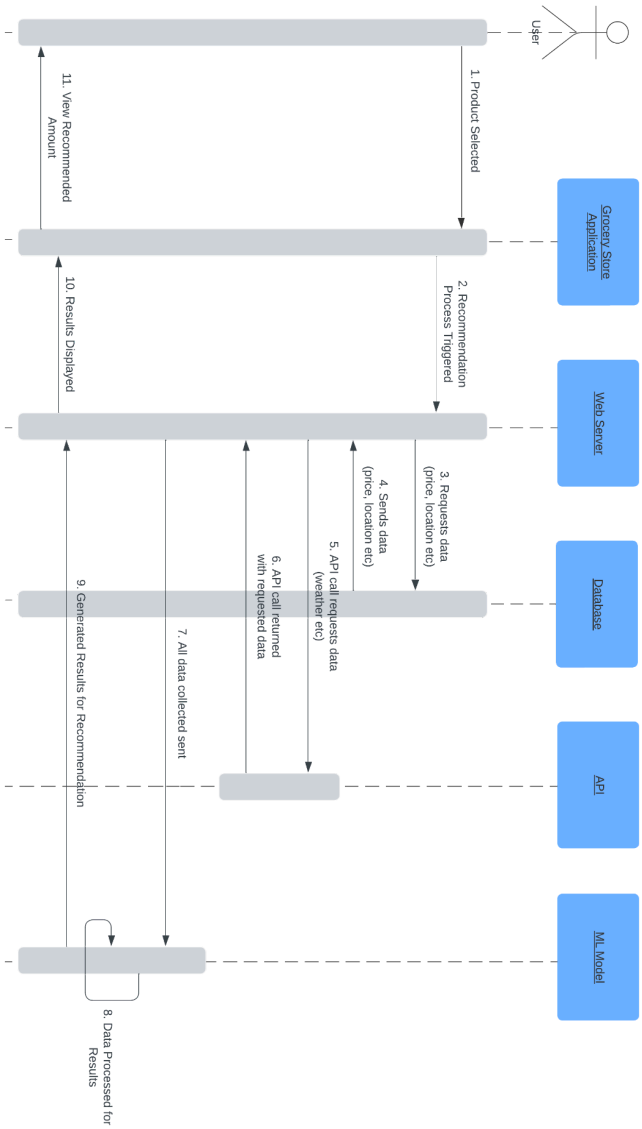


Sequence Diagrams

The functional requirements (FRs) that have been chosen for sequence diagram illustrations highlight the important and useful features of the system, emphasising its key functions. Important user situations including order recommendations, tracking, product search, OTP security, and vendor management are covered by these criteria. The sequence diagrams are designed to show the operations in a logical and uncomplicated manner, giving a fair assessment of the system's capabilities without needless detail. By focusing on useful functionality and system dependability, each FR that was selected makes a significant contribution to the user experience.

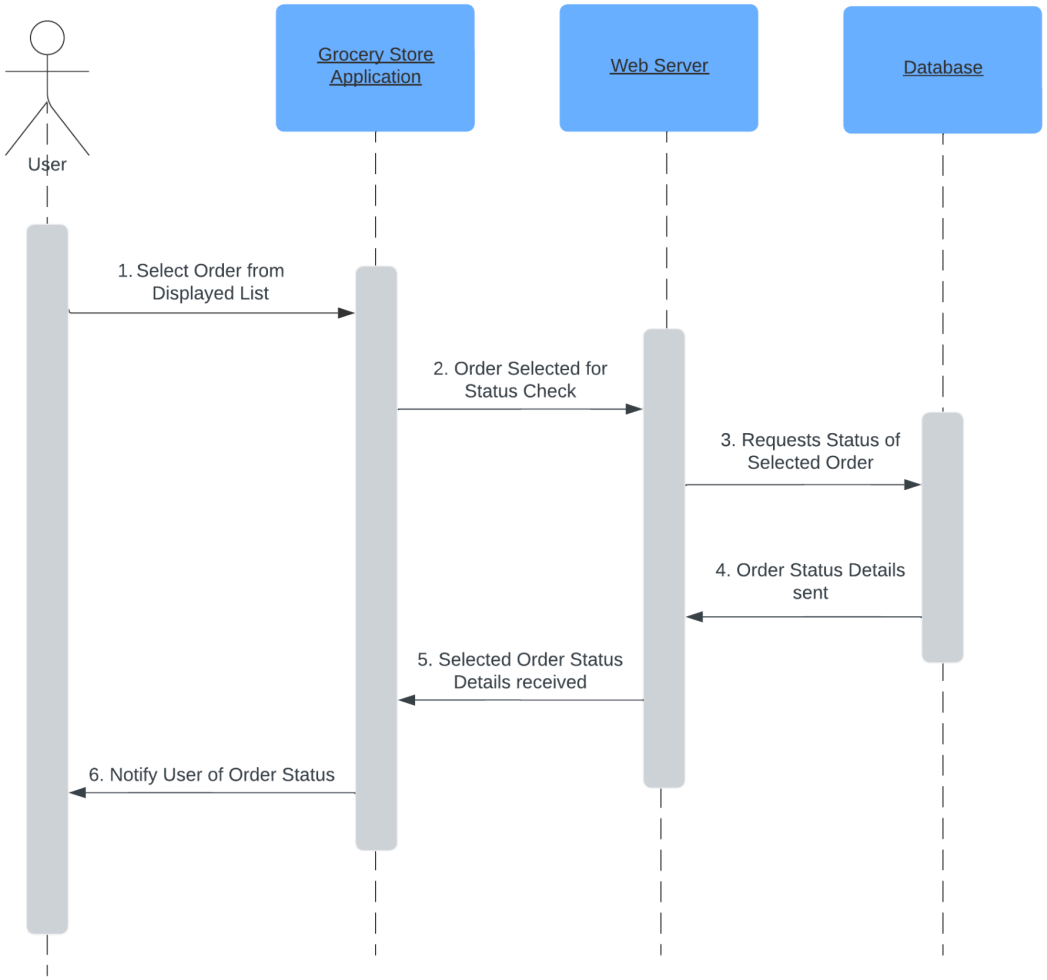
FR 2.19 Order Recommendation

Description: The sequence diagram illustrates the system recommending order amounts to a grocery store based on real-time data and external API inputs.



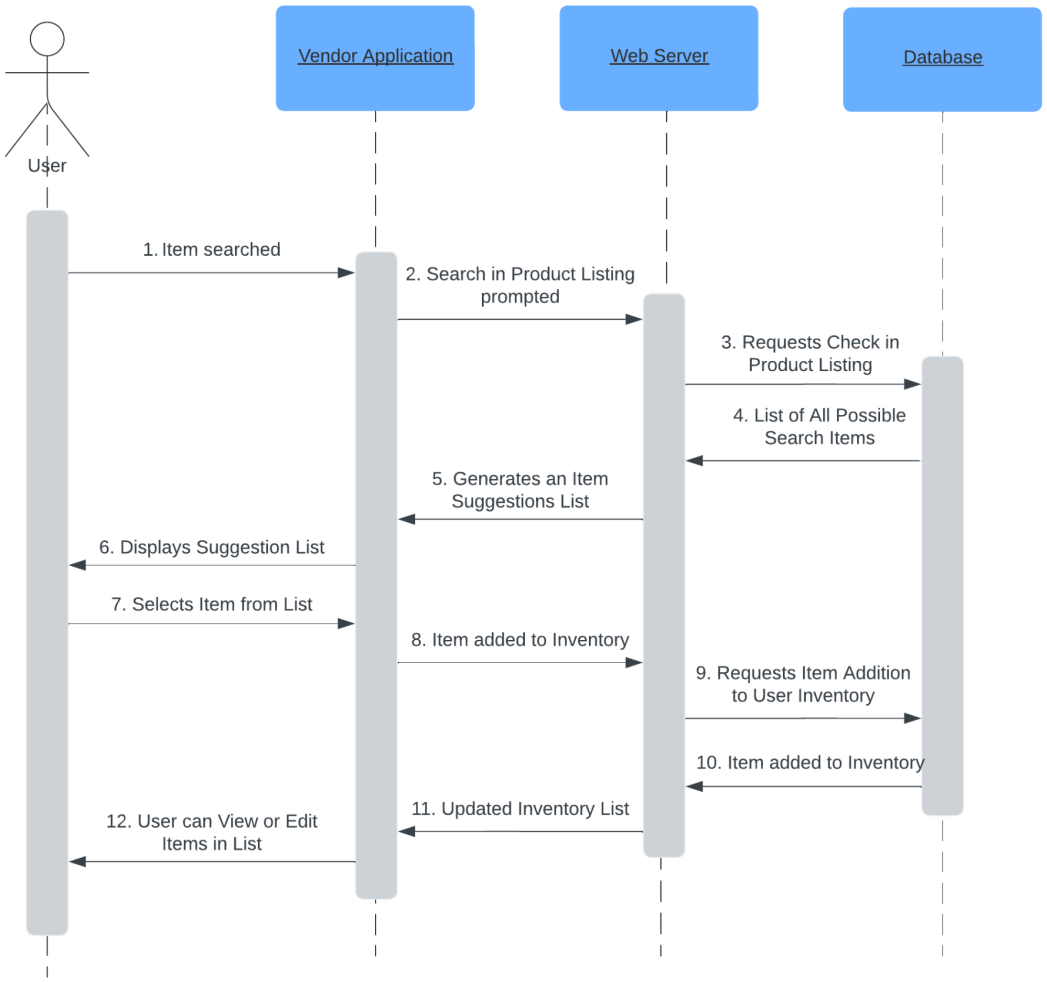
FR 2.24 Order Tracking

Description: The sequence diagram represents the logical flow of actions for a user to track their orders, from selection to delivery notification.



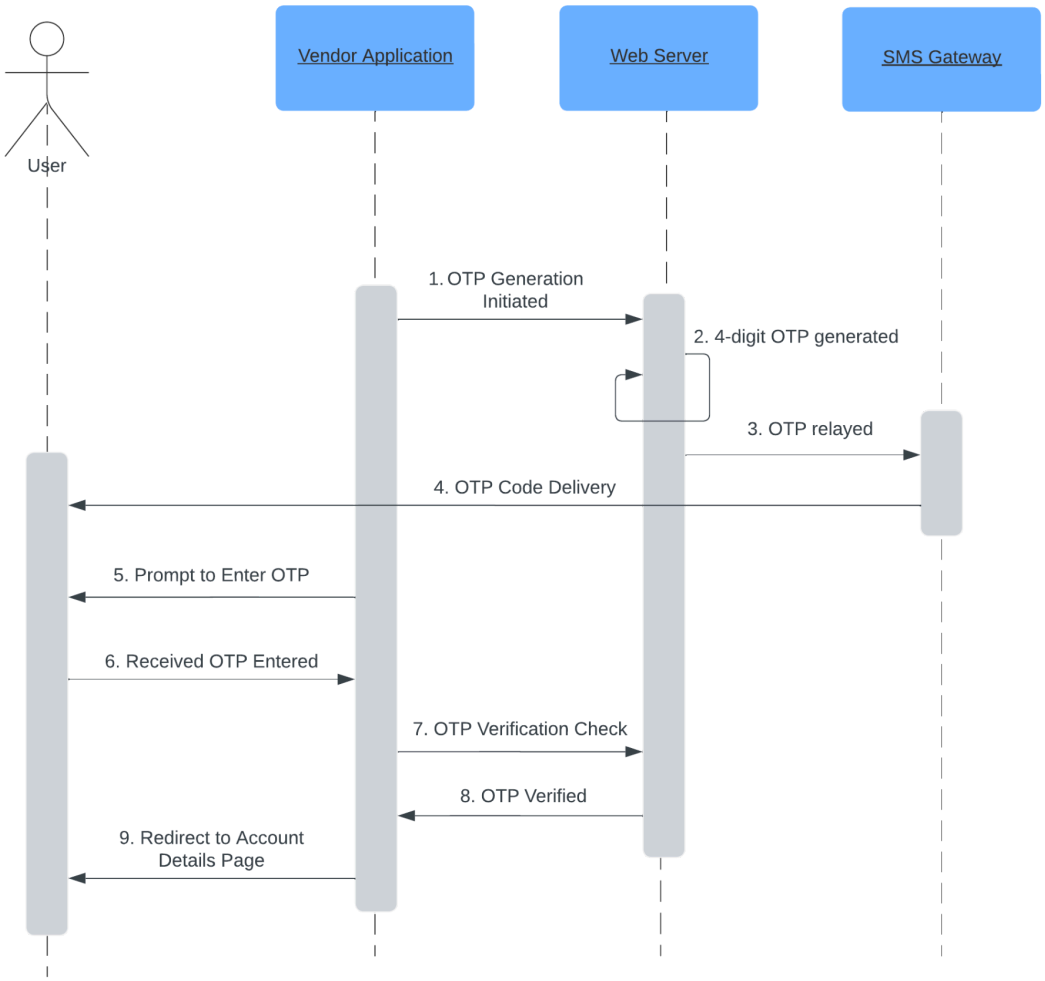
FR 3.14 All Products Search

Description: The sequence diagram depicts the practical steps for a vendor to search, select, and add products to their inventory.



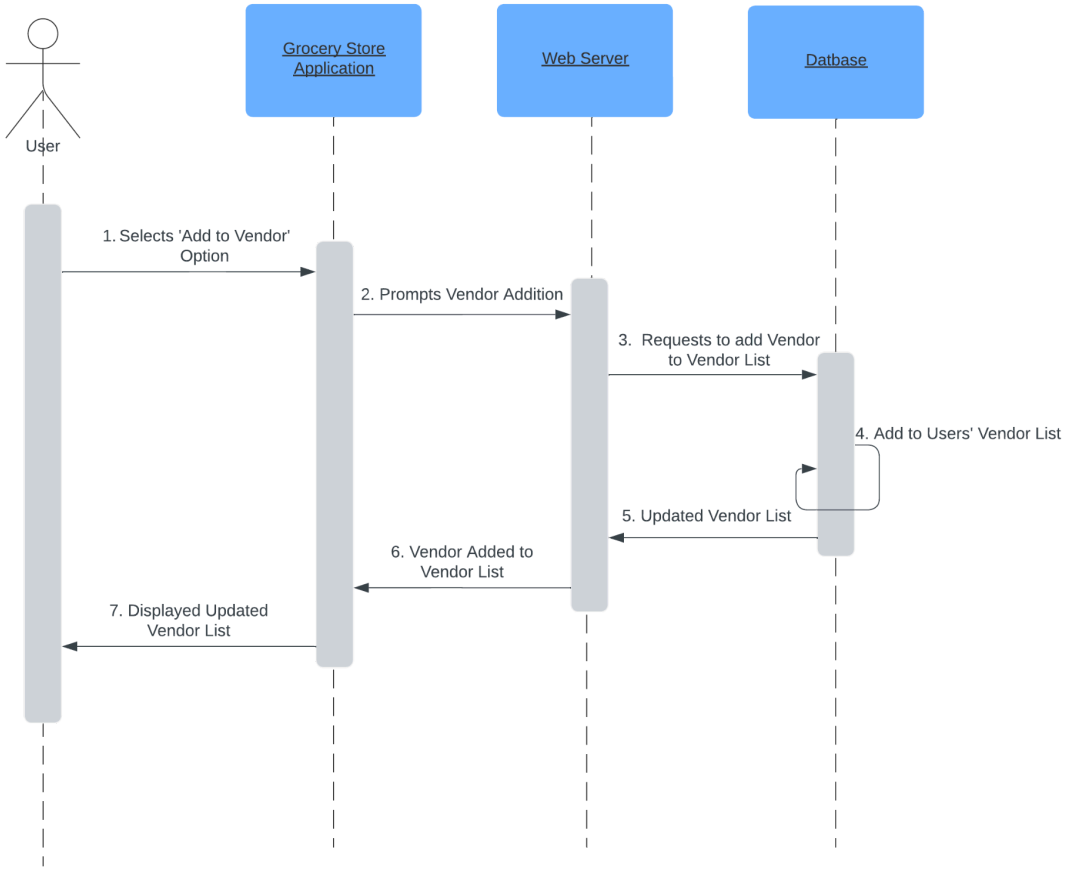
FR 1.4 OTP Code Generation and Delivery

Description: The sequence diagram illustrates the secure process of generating and delivering OTP codes for user verification.



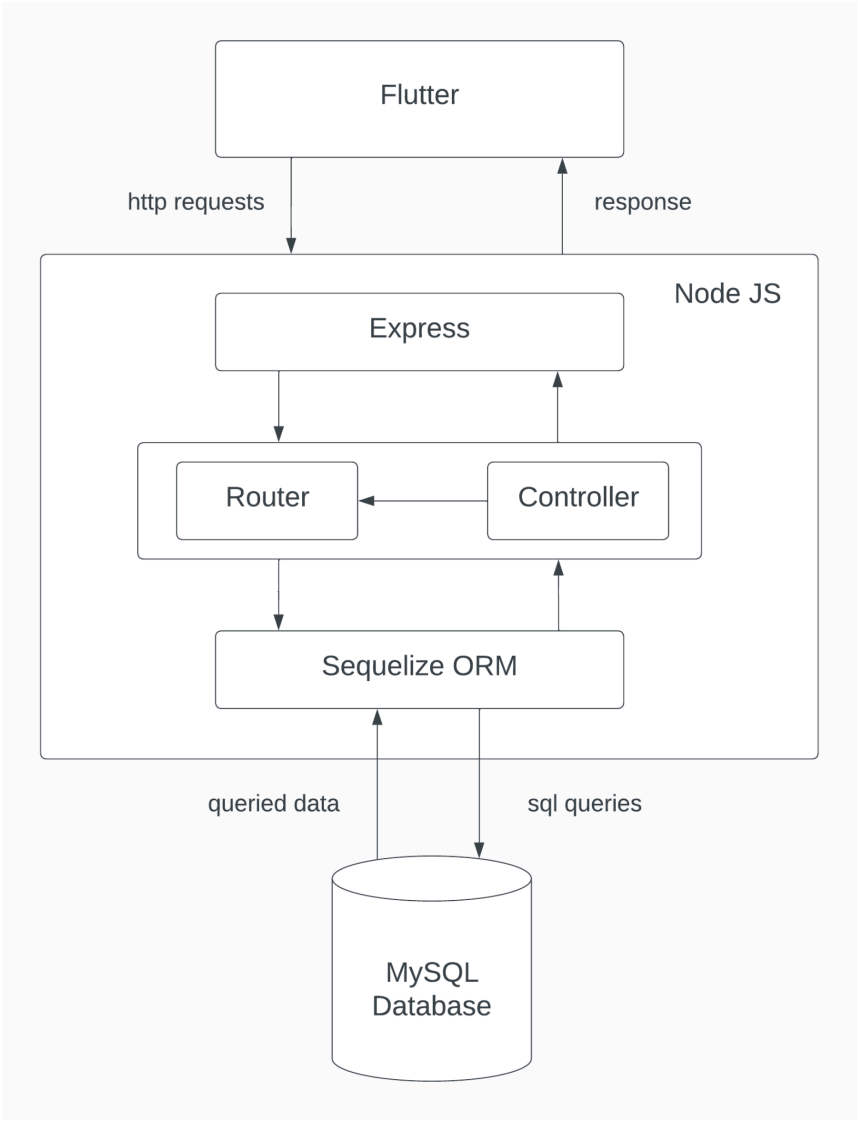
FR 2.14 Add Vendor to Vendor List

Description: The sequence diagram represents the user-friendly process of a grocery store adding a vendor to its list for efficient management.



API Design

The API Design explains the interaction that takes place between the frontend (mobile application), the backend and the database (implemented on MySQL).Our mobile application, which has been made through flutter, interacts with the backend using http requests. Those http requests are sent to specific endpoints that have been defined in the router. One example of the endpoints defined in the router is '/api/grocery_store/allgrocerystores' for fetching all grocery stores.The router then calls the controller which contains all the functionality (including the CRUD functions, search function etc). We are also using Sequelize ORM in Node JS. Sequelize ORM or object relational mapper is used to perform database operations. ORMS basically connects objects in the code with the table in the database. So it translates ORM calls into SQL queries and interacts with the MySQL database. For example, fetching all grocery stores involves using the 'findAll' method to query the 'grocery_stores' table. The database then returns queried data to the backend. Backend then processes the data and sends an appropriate HTTP response



to the mobile ap p.

Machine Learning Design

Introduction

This section covers the feature engineering performed on the Local Pharmacy and Corporación Favorita datasets, the shortlisting of the ML models for FYP I and FYP II, the neural network architecture of our best performing LSTM model, and lastly the process of deployment.

Feature Engineering

Local Pharmacy Dataset

We collected one year's data from the pharmacy. There was a separate csv for each month. So, the first step was to combine the csv files into one dataset (Figure 1).

```
import pandas as pd
import glob
```

```
csv_files_path = '../../Data/Pharmacy/'
```

```
csv_files = sorted(glob.glob(csv_files_path + '*.csv'))
```

```
combined_data = pd.DataFrame()
```

```
for csv_file in csv_files:
    month_data = pd.read_csv(csv_file)
    combined_data = pd.concat([combined_data, month_data], ignore_index=True)
```

Figure 1 Combining monthly csv files into one dataset

Then, the dataset is inspected (Figure 2). It is noticed that the data needs to be cleaned. saleinvcode, customerref, invdiscperc, flatdisc, misccharges, invsalestax, packqty, itemdiscperc, batch, salestax, datestring customer, and rowid need to be removed as they are either empty, zero, same for every entry, or irrelevant to demand forecasting. The remarks attribute is used to enter the names of customers, therefore, it should be removed to ensure privacy (Figure 3).

```
In [3]: df.head()
```

saleinvcode	CustomerRef	date	invdiscperc	flatdisc	misccharges	invsalestax	remarks	looseqty	packqty	price	itemdiscperc	packunits	batch	expiry	salestax	itemname	datestring	customer	rowid
115439	NaN	7/1/22 0:02	0.0	0.0	0	0	NaN	1	0	18.00	0.0	1	.	7/5/24 0:00	0	PEDITRAL (LEMON) ORS SACHET	1/7/2022	****CASH SALES CUSTOMER	226819
115440	NaN	7/1/22 0:03	0.0	3.0	0	0	NaN	3	0	6.31	0.0	20	.	2/8/24 0:00	0	SPIROMIDE 20MG TAB	1/7/2022	****CASH SALES CUSTOMER	226820
115440	NaN	7/1/22 0:03	0.0	3.0	0	0	NaN	2	0	7.83	0.0	30	.	8/1/25 0:00	0	CARLOV 6.25MG TAB(30'S)	1/7/2022	****CASH SALES CUSTOMER	226821
115440	NaN	7/1/22 0:03	0.0	3.0	0	0	NaN	2	0	34.29	0.0	14	.	10/1/24 0:00	0	NEXUM 40MG CAP	1/7/2022	****CASH SALES CUSTOMER	226822
115441	NaN	7/1/22 0:04	0.0	0.0	0	0	NaN	8	0	6.31	0.0	20	.	2/8/24 0:00	0	SPIROMIDE 20MG TAB	1/7/2022	****CASH SALES CUSTOMER	226823

Figure 2 Inspecting the dataset

```
df.head()
```

	date	looseqty	price	packunits	expiry	itemname	datestring
0	7/1/22 0:02	1	18.00	1	7/5/24 0:00	PEDITRAL (LEMON) ORS SACHET	1/7/2022
1	7/1/22 0:03	3	6.31	20	2/8/24 0:00	SPIROMIDE 20MG TAB	1/7/2022
2	7/1/22 0:03	2	7.83	30	8/1/25 0:00	CARLOV 6.25MG TAB(30'S)	1/7/2022
3	7/1/22 0:03	2	34.29	14	10/1/24 0:00	NEXUM 40MG CAP	1/7/2022
4	7/1/22 0:04	8	6.31	20	2/8/24 0:00	SPIROMIDE 20MG TAB	1/7/2022

Figure 3 Dataset after removing the aforementioned columns

The date attribute contains the date and the time, it should be split into two parts, date and time (Figure 4).

```
df[['date', 'time']] = df['date'].str.split(' ', 1, expand=True)
```

```
df.head()
```

	date	time	itemname	packunits	expiry	price	looseqty
0	1/7/2022	0:02	PEDITRAL (LEMON) ORS SACHET	1	7/5/24	18.00	1
1	1/7/2022	0:03	SPIROMIDE 20MG TAB	20	2/8/24	6.31	3
2	1/7/2022	0:03	CARLOV 6.25MG TAB(30'S)	30	8/1/25	7.83	2
3	1/7/2022	0:03	NEXUM 40MG CAP	14	10/1/24	34.29	2
4	1/7/2022	0:04	SPIROMIDE 20MG TAB	20	2/8/24	6.31	8

Figure 4 Date is split in date and time

There were multiple instances of sales for each item for each day. So, daily sales data was aggregated based on date and itemname (Figure 5).

```
agg_functions = {
    'packunits': 'first',
    'expiry': 'first',
    'price': 'first',
    'looseqty': 'sum'
}
```

```
combined_df = df.groupby(['date', 'itemname'], as_index=False).agg(agg_functions)
```

```
combined_df.head()
```

	date	itemname	packunits	expiry	price	looseqty
0	2022-07-01	10CC SHIFA D/SYRINGE(UNJT)(BM)	100	12/12/24	30.00	6
1	2022-07-01	1CC BD SYRINGE	100	12/12/24	30.00	1
2	2022-07-01	3CC SYRINGE INJEKT	100	3/1/24	15.00	3
3	2022-07-01	ACCU CHECK LANCET (CHINA)	200	12/12/24	3.00	50
4	2022-07-01	ACDERMIN GEL	1	5/1/23	278.44	1

Figure 5 Aggregating daily sales data for each item

Lastly, we filtered the data for 'PANADOL TAB' and saved the filtered dataset (Figure 6) .

```
filtered_df = df[df['itemname'] == 'PANADOL TAB']
```

```
filtered_df.head(10)
```

	date	itemname	packunits	expiry	price	looseqty
376	01/07/2022	PANADOL TAB	200	4/25/24	1.70	60
952	02/07/2022	PANADOL TAB	200	4/25/24	1.70	70
1490	03/07/2022	PANADOL TAB	200	4/25/24	1.70	55
2671	05/07/2022	PANADOL TAB	200	4/25/24	1.45	20
4407	08/07/2022	PANADOL TAB	200	4/25/24	1.70	70

Figure 6 Filtering Panadol Tab rows

Corporación Favorita Grocery Sales Forecasting

The initial step involved loading the various datasets required for the analysis including the training set, test set, holiday events (Figure 1.1), oil prices (Figure 1.2), store information (Figure 1.3), and transaction data. After loading the datasets, we examined their content and shape to gain insights into the structure and size of each dataset as part of data exploration.

```
In [3]: holiday_events.head()
```

Out[3]:

	date	type	locale	locale_name	description	transferred
0	2012-03-02	Holiday	Local	Manta	Fundacion de Manta	False
1	2012-04-01	Holiday	Regional	Cotopaxi	Provincializacion de Cotopaxi	False
2	2012-04-12	Holiday	Local	Cuenca	Fundacion de Cuenca	False
3	2012-04-14	Holiday	Local	Libertad	Cantonizacion de Libertad	False
4	2012-04-21	Holiday	Local	Riobamba	Cantonizacion de Riobamba	False

Figure 1.1 Holiday Events

```
In [4]: oil.head()
```

Out[4]:

	date	dcoilwtico
0	2013-01-01	NaN
1	2013-01-02	93.14
2	2013-01-03	92.97
3	2013-01-04	93.12
4	2013-01-07	93.20

Figure 1.2 Oil Prices

```
In [6]: stores.head()
```

Out[6]:

	store_nbr	city	state	type	cluster
0	1	Quito	Pichincha	D	13
1	2	Quito	Pichincha	D	13
2	3	Quito	Pichincha	D	8
3	4	Quito	Pichincha	D	9
4	5	Santo Domingo	Santo Domingo de los Tsachilas	D	4

Figure 1.3 Store Information

Then, the date feature in each dataset was, initially a string, converted to the datetime data type (Figure. 1.4). This step is crucial for time series analysis.

```
In [10]: holiday_events["date"] = pd.to_datetime(holiday_events.date)
         oil["date"] = pd.to_datetime(oil.date)
         test["date"] = pd.to_datetime(test.date)
         train["date"] = pd.to_datetime(train.date)
         transactions["date"] = pd.to_datetime(transactions.date)
```

Figure 1.4 Converting date feature to datetime data type

To understand the sales patterns over time, a visualisation (Figure 1.5) of the time series was plotted.

```
In [11]: def plot_series(time, series, format="-", start=0, end=None):
             fig, ax = plt.subplots(figsize=(14,5))
             plt.plot(time[start:end], series[start:end], format)
             plt.xlabel("Time")
             plt.ylabel("Sales")
             plt.grid(True)
             plt.show()
             plt.close()

In [12]: plot_series(train["date"], train["sales"], format="-", start=0, end=None)
```

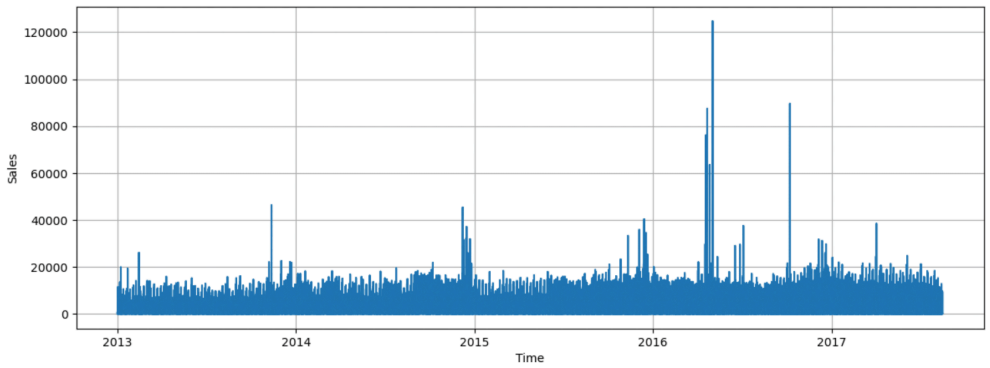


Figure 1.5 Visualising time series data

Next, data merging was performed (Figure 1.6), holiday events, oil prices, and store information were merged with the training and test sets. Transactions data was not utilised because it isn't available for the test set. The merging operations were conducted sequentially: first with holiday events based on the 'date' column, then second with the oil prices using the 'date' column; and finally with store-specific details using the 'store_nbr' column. The final expanded dataset was as seen in Figure 1.7.

```
In [13]: holiday_events = holiday_events.drop_duplicates(subset=['date'], keep='last')

In [14]: train.shape

Out[14]: (3000888, 6)

In [15]: train = pd.merge(train,holiday_events,how="left",on='date', validate="many_to_one")

In [16]: train.shape

Out[16]: (3000888, 11)

In [17]: train = pd.merge(train,oil,how="left",on='date')

In [18]: train.shape

Out[18]: (3000888, 12)

In [19]: train = pd.merge(train,stores,how="left",on="store_nbr",suffixes=("holiday","stores"))

In [20]: train.shape

Out[20]: (3000888, 16)
```

Figure 1.6 Joining holiday_events, oil, and stores

```
In [21]: train.head()
```

Out[21]:

	id	date	store_nbr	family	sales	onpromotion	typeholiday	locale	locale_name	description	transferred	dcoilwtico	city	state	typestores	cluster
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13
1	1	2013-01-01	1	BABY CARE	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13
2	2	2013-01-01	1	BEAUTY	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13
3	3	2013-01-01	1	BEVERAGES	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13
4	4	2013-01-01	1	BOOKS	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13

Figure 1.7 Training with holiday_events, oil, and stores features

Extracting time features like the day of the week, month, and year from the date in both training and test sets was crucial (Figure 1.8). The 'day_of_week', ‘month’, and ‘year’ features, breaking down the week into days, played a key role in capturing and adapting to sales patterns on a weekly, monthly, and yearly basis. This enhanced the model's ability to recognize variations within a week, improving time series forecasting.

```
In [29]: train['day_of_week'] = train['date'].dt.day_of_week
         train['day_of_week'] = train['day_of_week']+1
         train['month'] = train['date'].dt.month
         train['year'] = train['date'].dt.year

In [30]: train.head()
```

Out[30]:

	id	date	store_nbr	family	sales	onpromotion	typeholiday	locale	locale_name	description	transferred	dcoilwtico	city	state	typestores	cluster	day_of_week	month	year
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13	2	1	2013
1	1	2013-01-01	1	BABY CARE	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13	2	1	2013
2	2	2013-01-01	1	BEAUTY	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13	2	1	2013
3	3	2013-01-01	1	BEVERAGES	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13	2	1	2013
4	4	2013-01-01	1	BOOKS	0.0	0	Holiday	National	Ecuador	Primer dia del ano	False	NaN	Quito	Pichincha	D	13	2	1	2013

Figure 1.8 Extracting weekday, month, and year

The 'transferred' column in the holiday events dataset was used to identify and convert transferred holidays to normal days (Figure 1.9). This step ensures consistency in the holiday feature across the dataset.

```
In [37]: test["typeholiday"]=np.where(test["transferred"]==True,'NDay',test["typeholiday"])
         test["typeholiday"]=np.where(test["typeholiday"]=='Work Day','NDay',test["typeholiday"])
         test["typeholiday"]=test["typeholiday"].fillna("NDay")

In [38]: display(test["typeholiday"].value_counts(dropna=False))

typeholiday
NDay       26730
Holiday     1782
Name: count, dtype: int64
```

Figure 1.9 Changing transferred holidays to normal days

Then, zeros in the oil prices column were addressed by interpolating the data. Visualisations before (Figure 1.10) and after interpolation (Figure 1.11) are presented.

```
In [41]: plot_series(train["date"], train["dcoilwtico"], format="-", start=0, end=None)
```

Figure 1.10 Missing values in oil prices

```
In [43]: train["dcoilwtico"]=np.where(train["dcoilwtico"] ==0, np.nan, train["dcoilwtico"])
         train.dcoilwtico.interpolate(limit_direction='both',inplace=True)

In [44]: train.shape

Out[44]: (3000888, 19)

In [45]: plot_series(train["date"], train["dcoilwtico"], format="-", start=0, end=None)
```

Figure 1.11 Fixing missing values in oil prices

Based on info seen (Figure 1.12), 'locale', 'locale_name', 'description', 'transferred' were dropped and new csv files were generated comprising the final test and training dataset that the model was trained on.

```
In [50]: test.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 28512 entries, 0 to 28511
Data columns (total 18 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   id           28512 non-null  int64
 1   date         28512 non-null  datetime64[ns]
 2   store_nbr    28512 non-null  int64
 3   family       28512 non-null  object
 4   onpromotion  28512 non-null  int64
 5   typeholiday  28512 non-null  object
 6   locale       1782 non-null   object
 7   locale_name  1782 non-null   object
 8   description  1782 non-null   object
 9   transferred  1782 non-null   object
 10  dcoilwtico   28512 non-null  float64
 11  city         28512 non-null  object
 12  state        28512 non-null  object
 13  typestores   28512 non-null  object
 14  cluster      28512 non-null  int64
 15  day_of_week  28512 non-null  int32
 16  month        28512 non-null  int32
 17  year         28512 non-null  int32
dtypes: datetime64[ns](1), float64(1), int32(3), int64(4), object(9)
memory usage: 3.6+ MB
```

Figure 1.12 Removing ID and columns with mostly null values

Model Shortlisting

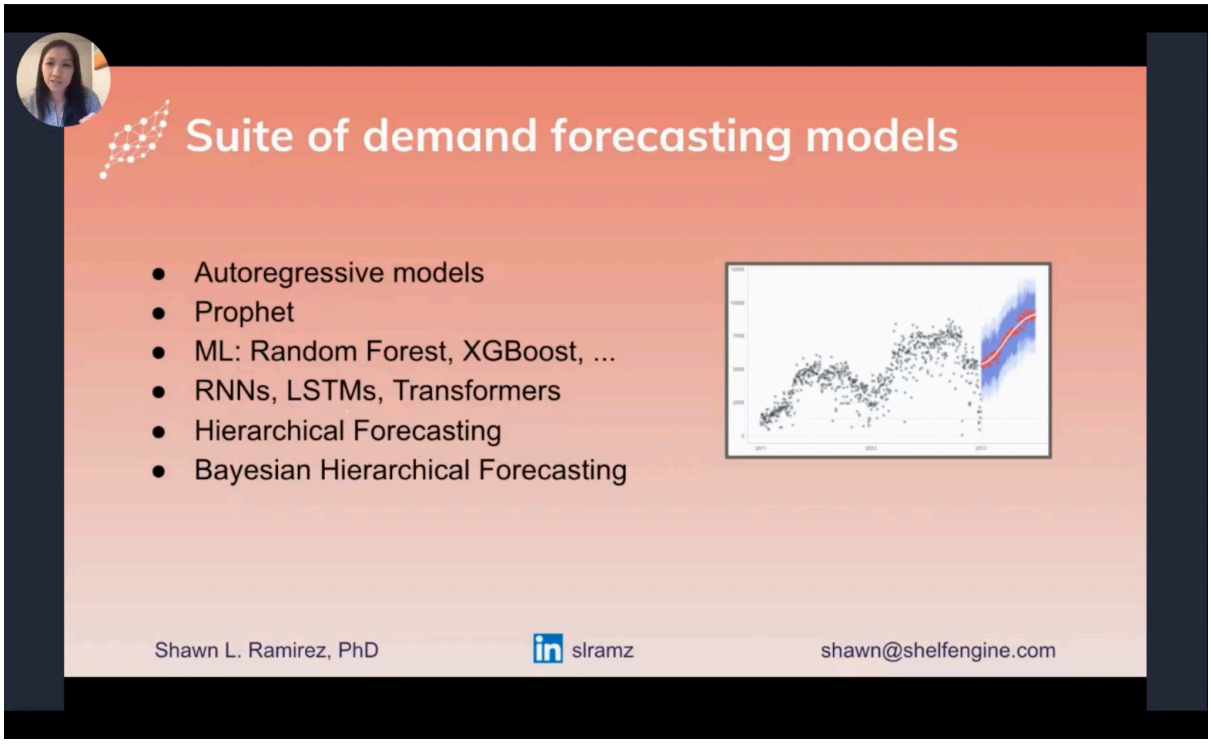


Figure 1: **Shelf Engine: Demand Forecasting Models**

Project Part I

As the inspiration behind our idea of demand forecasting was Shelf Engine, we initially decided to use the same models that they did, since their results using these models were extremely accurate.

Random Forest

We started the exploration with Random Forest because of its simplicity, flexibility, and our past experience with it making it a good choice. What makes Random Forest special, is its ability to capture complex non-linear patterns without extensive tuning.

XGBoost

Since we have structured data, XGBoost was considered a suitable choice. It is known to perform well in time series forecasting problems as evident by its popularity in competitions related to demand forecasting on kaggle. Due to the size of our dataset, XGBoost's speed and efficiency really helped in going over several iterations of training and testing.

Prophet (Facebook)

Prophet is designed specifically for time series forecasting with daily observations that display patterns like seasonality and holidays. As we had historical data for several seasons, Prophet seemed like a solid choice.

RNN (LSTM/GRU)

RNNs are powerful in modelling sequential patterns, making them suitable for time series forecasting. They can capture temporal dependencies in sales data over different time intervals.

Project Part II

The models that were used for FYP I provided a solid foundation, whereas the models that will be implemented in FYP II leverage recent advancements to further enhance prediction accuracy and address specific challenges in the time series domain.

LightGBM

LightGBM has shown excellent performance in time series forecasting, as evidenced by its success in the M5 competition. Its efficiency makes it suitable for handling large grocery store datasets.

N-BEATS

N-BEATS has proven effective in recent competitions, outperforming well-established methods, such as demonstrating superior performance compared to traditional statistical approaches.

DeepAR (Amazon)

DeepAR model is based on a set of RNN models, where each model is trained on a specific subset of the data. Finally, the predictions from all of the models are combined allowing deepAR to handle multiple time series effectively.

Neural Network Architecture

We created a sequential model (Figure 1), a model with a linear stack of layers. The first layer of the model is an InputLayer, where we pass our training sequences of length 7 and 14 features per time step. The second layer is an LSTM layer with 64 neurons. The third layer is a Dense layer with 8 ReLUs. Lastly, the output layer is a dense layer with a single ReLU neuron as it’s a regression task.

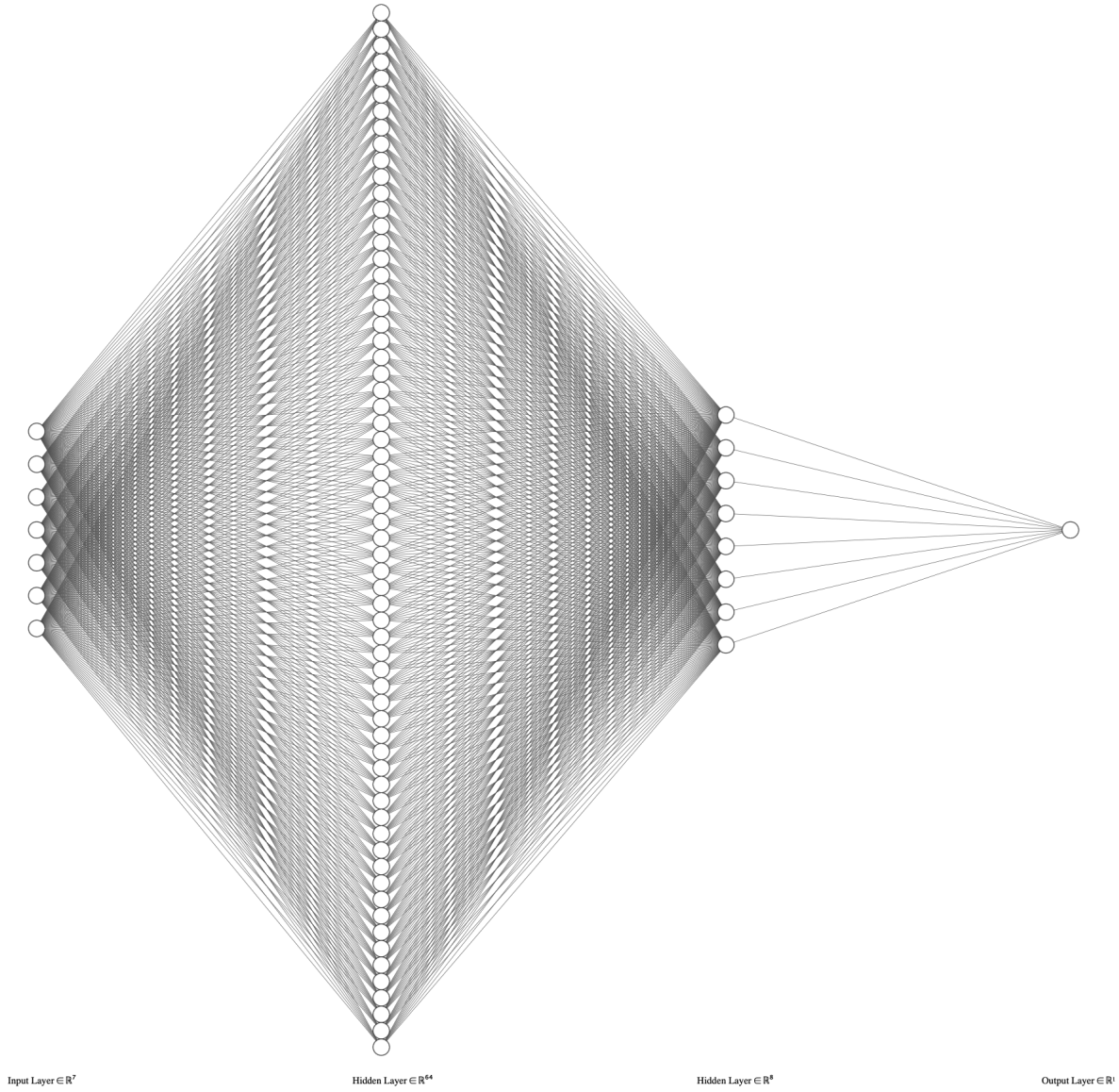


Figure 1 Neural network architecture draw using NN SVG

Deployment Design

Machine Learning

We created a sub directory, ModelDeployment, in our GitHub repository for version control. Installed Anaconda for package management and created a Conda environment (deployment) for the project. Developed a Flask web application in Python. Incorporated a simple form in the html template to take input (family and date) from the user. Added our demand forecasting model, trained on the kaggle dataset, and saved as joblib files. Updated the Flask application to handle model predictions based on the family and date user enters, along with related features (whether it was a holiday on that date or

what was the price of the oil on that date). Modified the html to display the model's prediction and the actual sales from the selected date for the selected family. Uploaded pre-trained models to a 'models' folder in the project directory. Adjusted the app.py file to load models and handle form submissions. Checked the functioning of the model locally by selecting the date and family and obtaining predictions. Prepared for deployment by creating a requirements.txt file listing necessary libraries. Used Digital Ocean to deploy the Flask application, connecting it to the GitHub repository. Resolved errors during deployment and awaited the completion of the deployment process. Checked the live url to access the deployed demand forecasting web application (Figure 1).

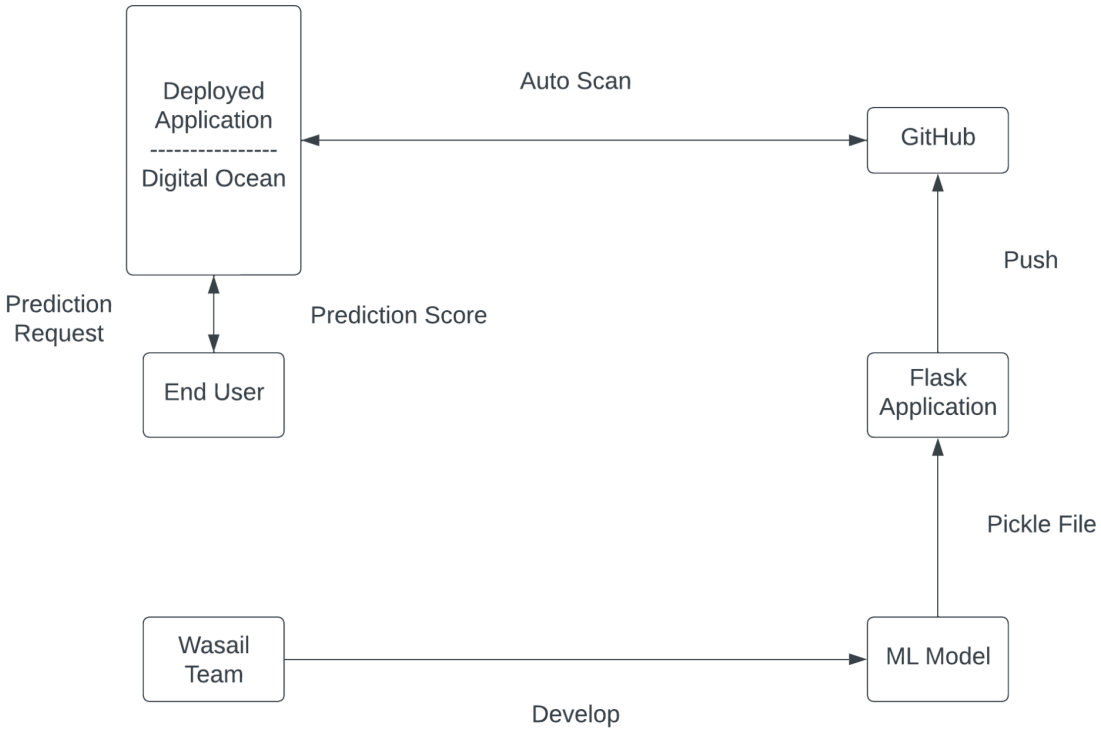


Figure 1 Deployment Diagram (FYP I)

Conclusion

In conclusion, this design document provides a comprehensive blueprint for developing a sophisticated demand forecasting system tailored for the grocery retail industry. The integration of diverse technologies, ranging from backend development tools to machine learning frameworks, highlights the depth and complexity of the solution. The document not only emphasises the importance of data design and feature engineering but also sheds light on the selection and deployment of machine learning models for accurate demand predictions. The deployment on Digital Ocean, underscores the practicality and real-world applicability of the proposed solution. Overall, this document serves as a guide for building a robust and scalable system that can significantly enhance the efficiency of inventory management in the grocery retail sector.

Implementation

The following implementation document provides an outline of all the work completed in the first phase of our final year project. It is divided into three main sections: the vendor app, the grocery store app, and the admin portal, with each section detailing different phases of development. The final phase of each section covers the work done in FYP 2.

In the vendor mobile application section, the document begins by explaining the development of both the front end and back end. The front-end development journey started with designing the application's prototype in Figma and implementing it using Flutter. For the back end, the process involved designing the database, implementing it in MySQL, and developing REST APIs using Node.js. Test cases for all functional requirements are also included in this section.

The grocery store app section follows a similar structure. It details the front-end development process, which also began with prototyping in Figma and was brought to life with Flutter. The back-end development covered database design, implementation in MySQL, and REST API development using Node.js. This section also includes test cases for the functional requirements.

In the admin portal section, the focus is on developing an efficient and user-friendly portal using Node.js and React. This section covers the initial setup, database integration, and the implementation of key features required to manage users, grocery stores, vendors, products, and categories. The document also includes test cases specific to the admin portal’s functionality.

Additionally, the document outlines the machine learning (ML) section, which showcases the implementation of shortlisted ML models for various datasets. It also details the development of the Flask application and its deployment on the cloud, highlighting how these models are integrated into the overall system.

The cumulative work across all sections and phases demonstrates a comprehensive approach to developing a robust and scalable application suite as part of our final year project.

App Development

This section discusses the entire cycle of developing the vendor app, store app, and the admin portal. It has been further divided into sections including the front end, the backend, and the test cases.

Front End

Vendor App

As we started the front-end development process, our main objective was to produce a smooth and intuitive vendor app as per our deliverable for FYP I. The journey is broken down into phases that correspond with the functional requirements (FR) that guide the functionality and design of the app. Flutter, Google's UI toolkit, was used as it allows for a single codebase to run on both iOS and Android platforms, streamlining the development process.

Phase 1: Initial Front-End Design

The development process was smoother due to a Figma prototype already designed before the front-end development. Hence, the starting of this development started with the design or 'foundational' phase, the focus was on creating a user-friendly interface while adhering to the following functional requirements.

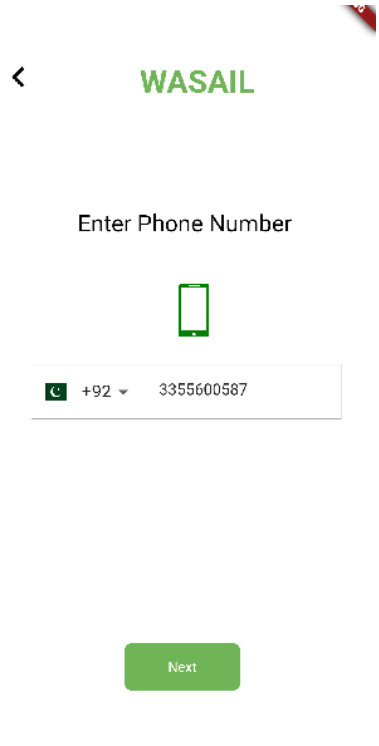


Figure 1.0

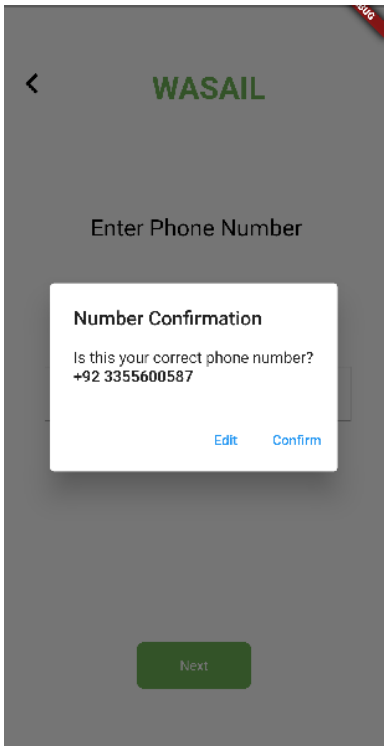


Figure 1.1

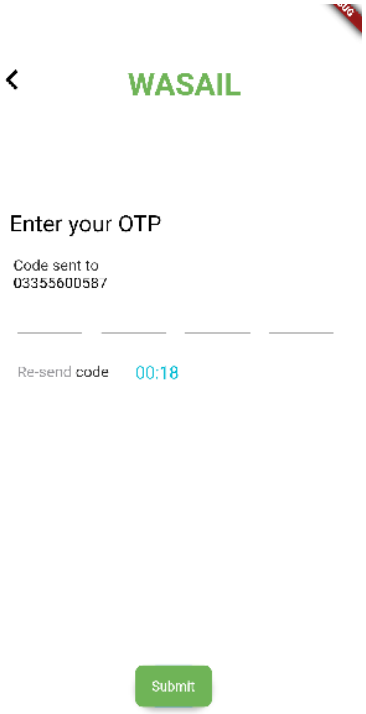


Figure 1.2

As seen in the figures above, up until the OTP process is the phone confirmation and checking process whereby new users are distinguished from existing ones.

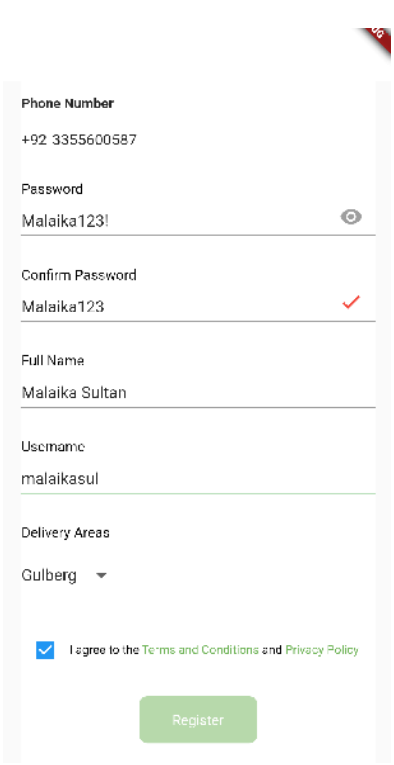


Figure 1.3

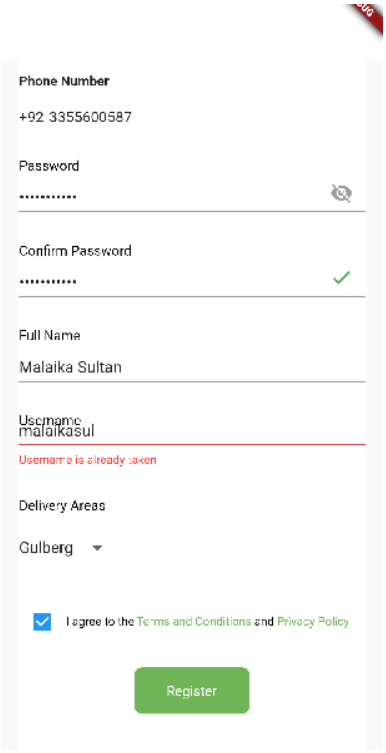


Figure 1.4

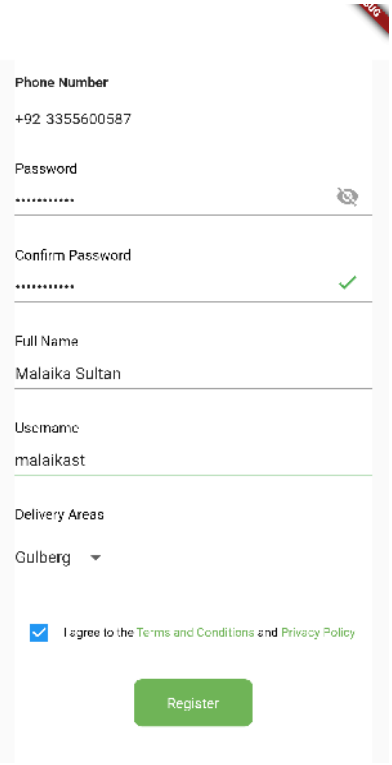


Figure 1.5

User Registration (Figure 1.0 to 1.5): Designed a secure and efficient registration process, ensuring user privacy and information collection. New users' phones are verified and then registered as will be seen in the sequence above. The registration comprises several validation checks including unique username validation, special characters password, password confirmation, and all filled-out fields for information required regarding vendors. Post successful registration are users taken to the login page.

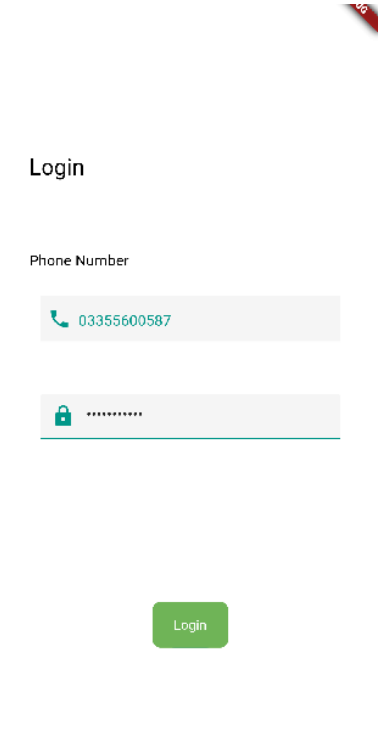


Figure 1.6

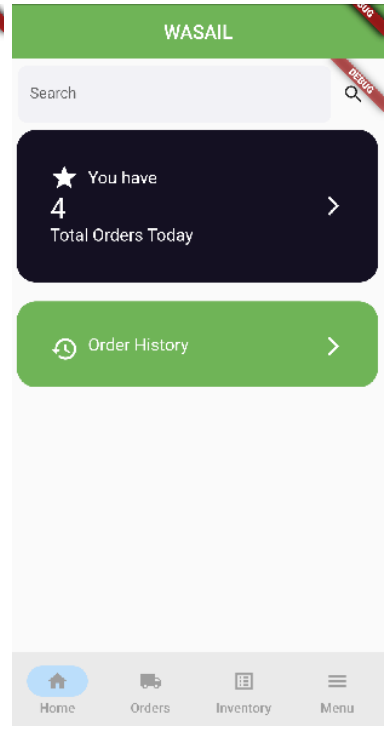


Figure 1.7

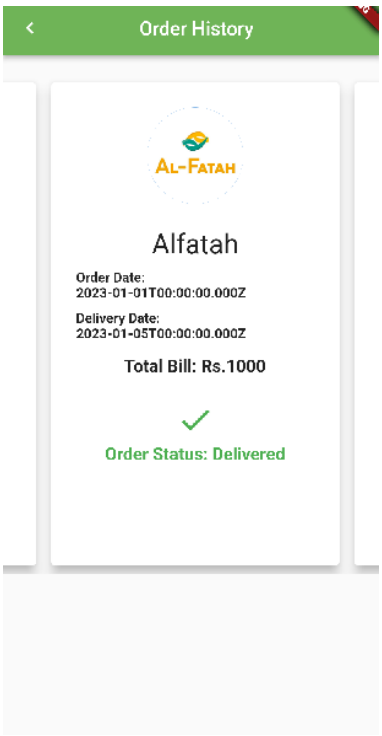


Figure 1.8

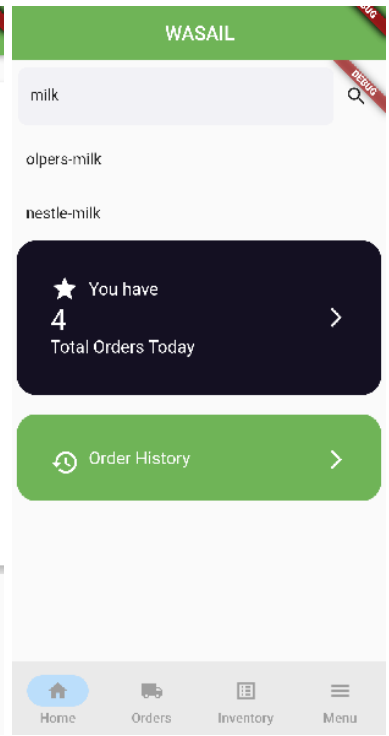


Figure 1.9

Home Page (Figure 1.6 to 1.9): Implemented a Home Page with intuitive navigation elements and key information display for an optimal user experience. As expected after a successful registration, the new

user will be directed towards the login screen, upon which correct credentials will open the app to the Home Page. The app's navigation is done through the navigation bar which facilitates the user's journey throughout the app. As seen in the figures above, the Home Page highlights the current orders placed by grocery stores to the vendors in a count tile which on clicking opens further details (shown later in Orders). The Home Page also stores previous orders marked delivered under Order History which stores all relevant details for each order.

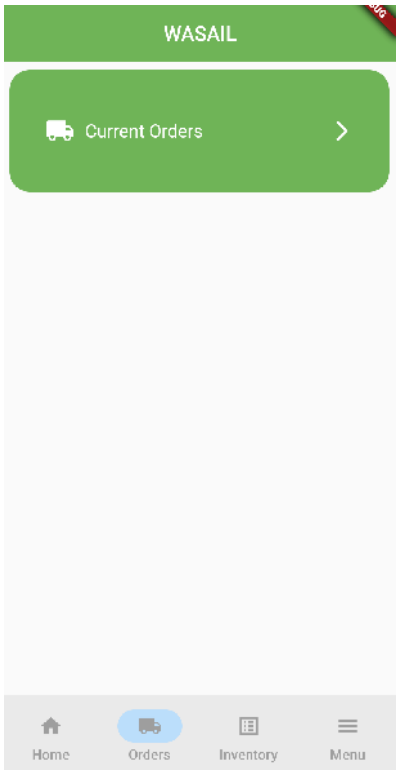


Figure 1.10

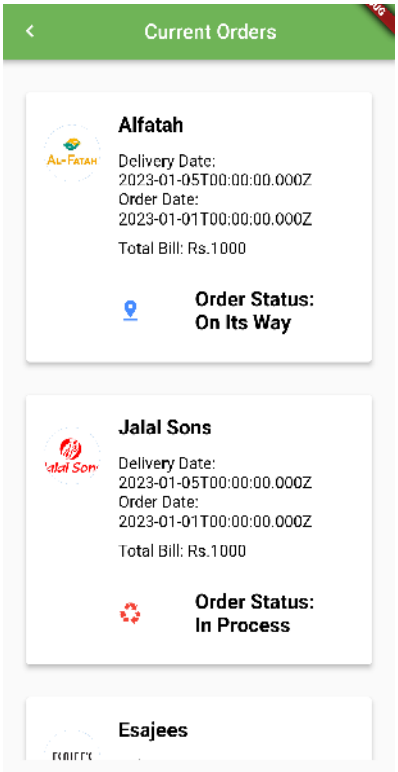


Figure 1.11

Furthermore, the search bar on the Home Page allows for searching in Vendors' own inventory i.e. items already present since the vendor will later have an extensive list of SKUs added, this would in quick search. Orders Page (Figure 1.10 to 1.11): Tracked all current orders, condition 'In Process' or 'On Its Way' for order tracking and management.

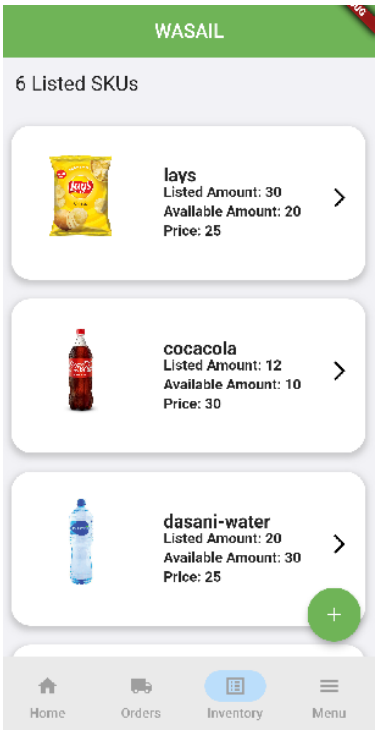


Figure 1.12

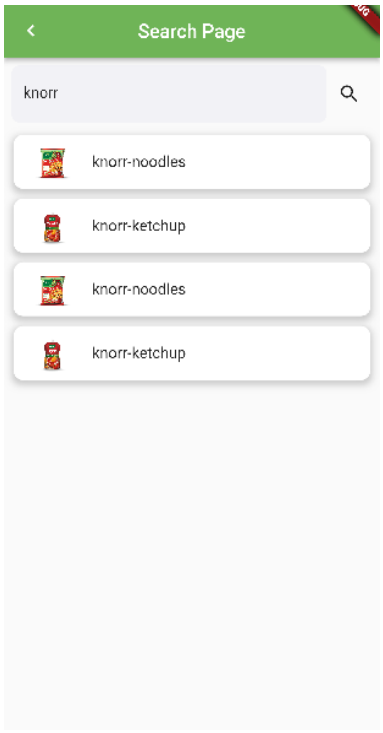


Figure 1.13

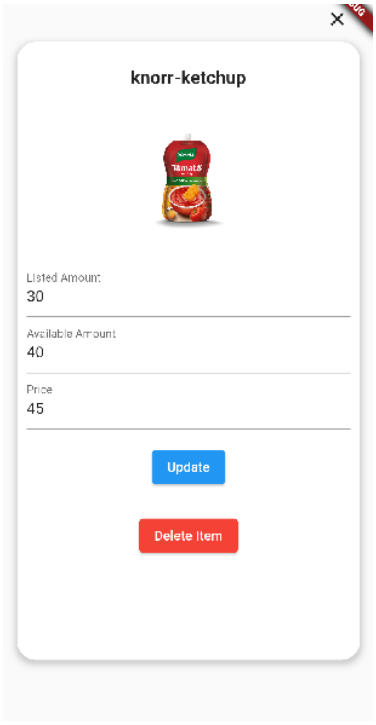


Figure 1.14

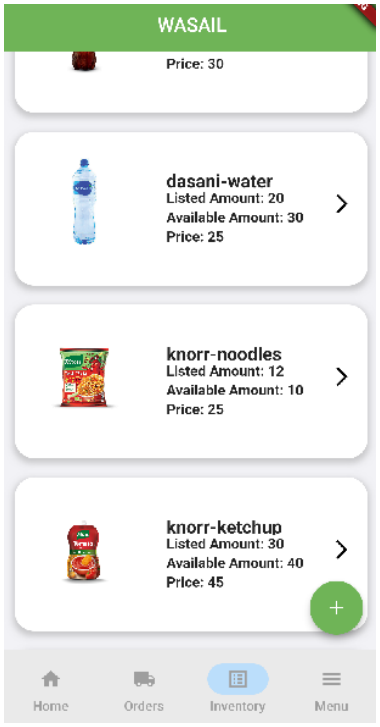


Figure 1.15

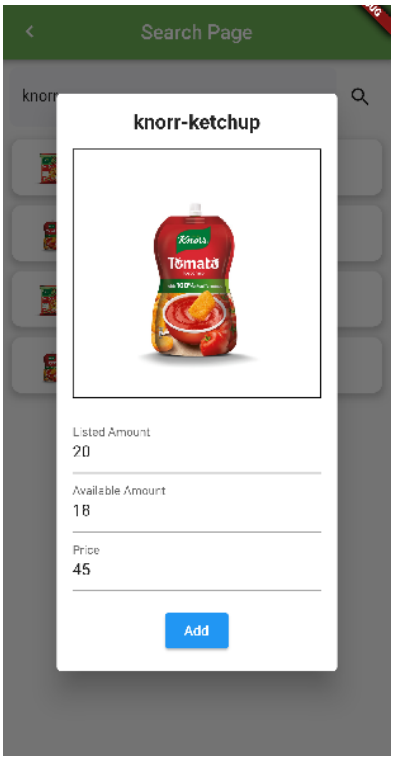


Figure 1.16

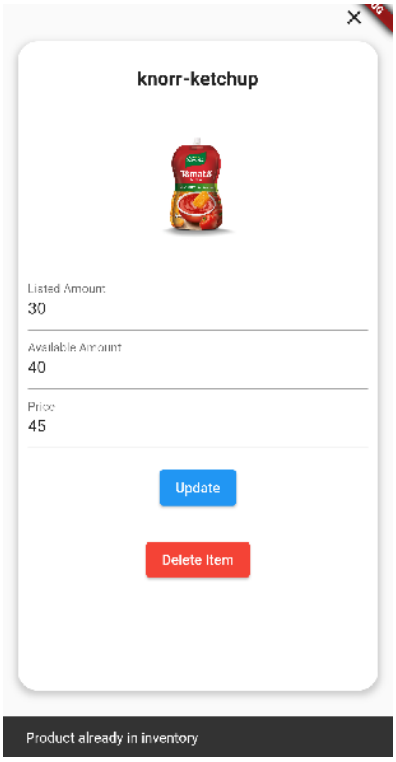
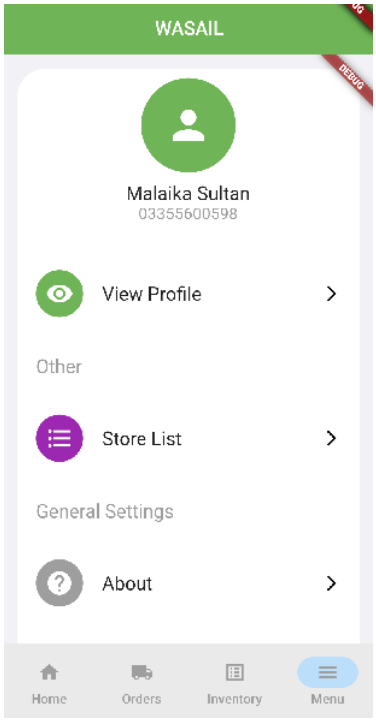
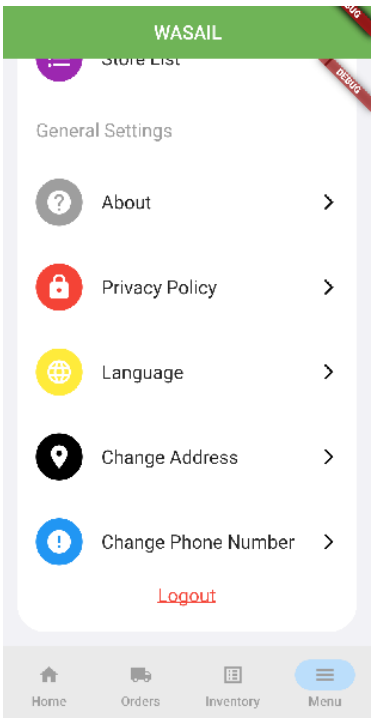


Figure 1.17

In the Orders Page, the user who is a vendor can track their current orders, which were earlier mentioned. An order will comprise the grocery store that placed the order, its order, and tentative delivery dates, as the order is still either 'In Process' or 'On Its Way', and the total bill of that order. Inventory Page (Figure 1.12 to 1.17): Showcased listed SKUs with detailed information and provided convenient options for managing inventory.

The second core part of the app is Inventory Management after Orders Management, in the Inventory Page, again accessed through the navigation bar showcases the complete list of inventory items of the vendor i.e. the user. The SKU count is displayed and an add button to add items from the system inventory to the user's inventory. The add icon is clicked and the Search Page opens. In this example, the item present in the system inventory is searched, displayed, and selected by the user. The item's details; listed amount, available amount, and price are entered and then added to the inventory. The updated SKU list is shown as Figure 1.15. Each SKU in the list can be clicked for item details which will then give the option to update or delete the item. Any time the user tries to add the same item again to their inventory from the system inventory, the system will prevent this duplication, inform the user of the item already present, and instead direct toward the update dialog.





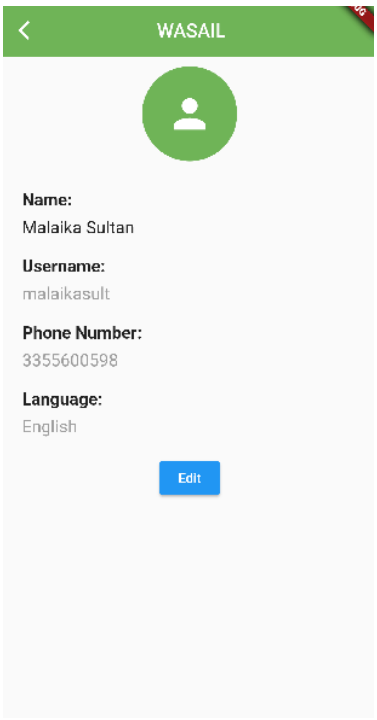
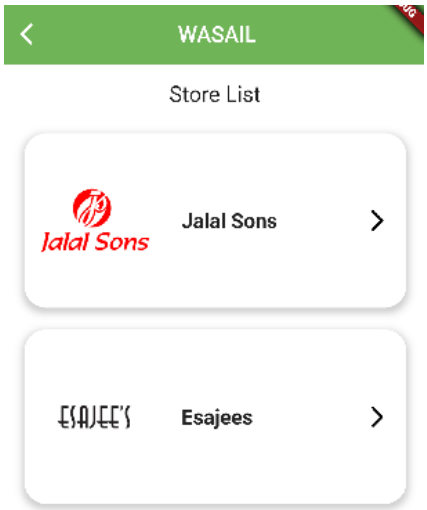


Figure 1.18 **Figure 1.19** **Figure 1.20**



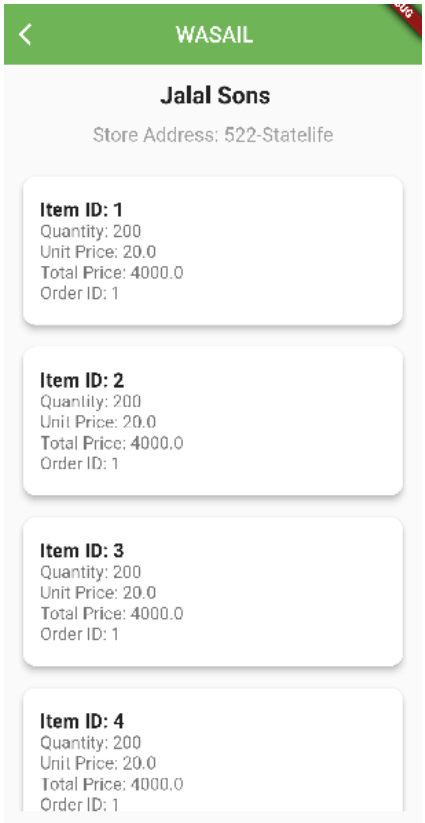


Figure 1.21 **Figure 1.22**

Menu Page (Figure 1.18 to 1.22): Created a structured layout for managing user profiles, organising the store list and app settings. Finally, when the user comes to the Menu section of the navigation, their profile details are visible, and under View Profile even further detailed. The user can edit their name only, as evident, and also, we will see later, switch their languages. Edited details are then updated accordingly. Among the app's general settings, an impertinent feature is the Store List. This holds all information regarding every store that has been dealt with by the vendor, regardless of order status. The store tile, clicked, will show all store details; their location, and also list down item-by-item orders placed by the store. Each item is displayed with its corresponding information; quantity, unit price, and total price as well as the Item ID and the ID of the order the item belongs to.

Phase 2: App Development Progress

Transitioning to mobile app development using Flutter, we implemented key features aligned with functional requirements, as seen in Phase I, the functionalities although not visible in the picture had been implemented in this phase. Although the prototype was the basis for the app, the early stage was the app progressing towards what has been shown above, the final product. The figures attached cannot showcase the navigation and gesture-based interaction that were employed in the app however they have been attempted to be exhibited through the sequence of the attached figures. The following were used whilst making the app functional.

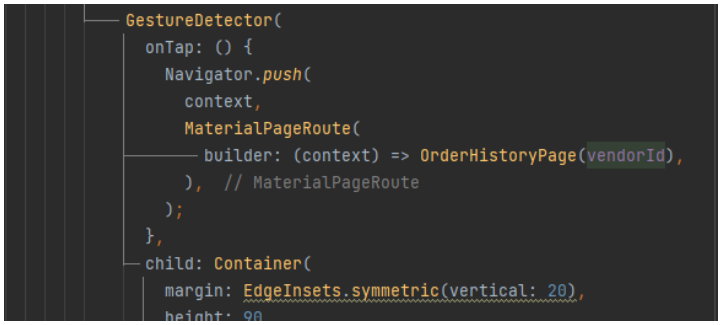


Figure 1.23 showing Gesture Detector Usage for accessing Order History in the Home Page

Gesture-Based Interactions: Enhanced user engagement with gestures for dynamic language selection across all pages using Flutter's GestureDetector as shown in Figure 1.23.
Stateful Widgets: Achieved seamless transitions from language selection to user registration and login processes.

```dart
class NavBar extends StatefulWidget {
  const NavBar({super.key});

  @override
  State<NavBar> createState() => _NavBarState();
}

class _NavBarState extends State<NavBar> {
  int index = 0;

  final screens = [
    Home(),
    Order(),
    Inventory(),
    Menu(),
  ];
```

Figure 1.24 shows the navigation bar using a stateful widget to handle navigation.

Additionally, This phase focused on global accessibility and visual personalization, aligned with functional requirements.

```
"app_name": "WASAIL",
"select_lang": "Select Language",
"continue_button": "Continue",
"enter_number": "  Enter Phone Number",
"welcome_msg" : "Welcome to Wasail",
"next_button" : "Next",
"login" : "Login",
```

```
"app_name": "وسائل ",
"select_lang":"زبان منتخب کریں",
"continue_button": "جاری رہے",
"enter_number": "      فون نمبر درج کریں    ",
"welcome_msg": "وسیل میں خوش آمدید",
"next_button" : "اگلے مرحلے پر",
"login" : "لاگ ان",
```

Figures 1.25 and 1.26 show English and Urdu .arb file snippets used throughout the app

```dart
child: Text(
  AppLocalizations.of(context)!.login,
  style: TextStyle(
    color: Colors.black,
```

Figure 1.27 shows how the app context carried the language selected and used its respective locale

```dart
static const List<LocalizationsDelegate<dynamic>> localizationsDelegates = <LocalizationsDelegate<dynamic>>[
  delegate,
  GlobalMaterialLocalizations.delegate,
  GlobalCupertinoLocalizations.delegate,
  GlobalWidgetsLocalizations.delegate,
];

/// A list of this localizations delegate's supported locales.
static const List<Locale> supportedLocales = <Locale>[
  Locale('en'),
  Locale('ur')
]; // <Locale>[]
```

Figure 1.28 shows declarations of localisation needed to internationalise a page

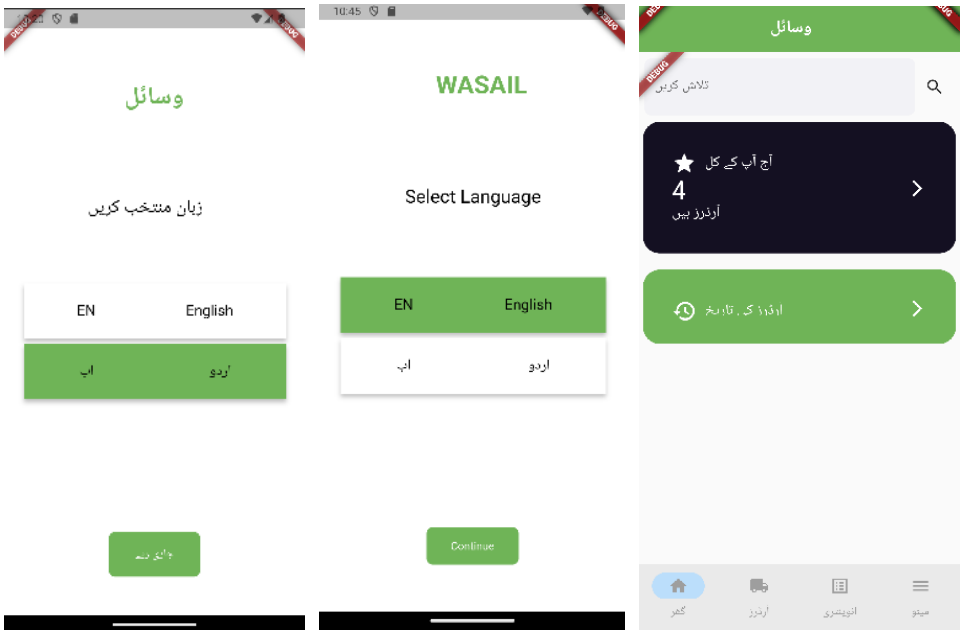


Figure 1.29 shows an overview of the app in Urdu

```
class _LanguageState extends State<Language> {
  String selectedLanguage = 'English';

  @override
  Widget build(BuildContext context) {
    LanguageProvider languageProvider =
        Provider.of<LanguageProvider>(context, listen: false);
    Locale? selectedLocale = languageProvider.selectedLocale;

    double screenWidth = MediaQuery.of(context).size.width;
    double screenHeight = MediaQuery.of(context).size.height;

    return MaterialApp(

      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: AppLocalizations.supportedLocales,
```

Figure 1.30 shows how the language locale was provided by the provider

```
class MyVendorApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MultiProvider(
     providers: [
       ChangeNotifierProvider<LanguageProvider>(
         create: (context) => LanguageProvider(),
       ),  // ChangeNotifierProvider
       ChangeNotifierProvider<PhoneNumberProvider>(
         create: (context) => PhoneNumberProvider(),
       ),  // ChangeNotifierProvider
       ChangeNotifierProvider<LoginChangeNotifier>(
         create: (context) => LoginChangeNotifier(),
       ),  // ChangeNotifierProvider
```

Figure 1.31 shows how we used Provider for various things across the app other than locale

As part of ensuring the locale was selected once and implemented across the app, we used Provider which manages the app-wide state. Screens or pages that have to use the saved state, in our case, the locale grasp the state via ‘Listeners’ which are informed of it by ‘Notifiers’ as shown in Figure 1.31.

Language Selection, Internationalisation, and State Management (Figure 1.25 to 1.31): Enabled users to navigate the app in their preferred language using Flutter's localization features. To implement language selection and internationalisation, we leveraged Flutter's built-in internationalisation (i10n) features. We utilised the Intl package to handle translations and ensure that the app's interface seamlessly adapts to different languages. The attached figure demonstrates the language selection in action, allowing users to effortlessly switch between languages in respect to our app which has Urdu and English. This entailed the use of ARB, an Application Resource Bundle, a JSON-based format used for managing localised resources and translations in Flutter applications. We had alternative translations for each string that the user had to see.

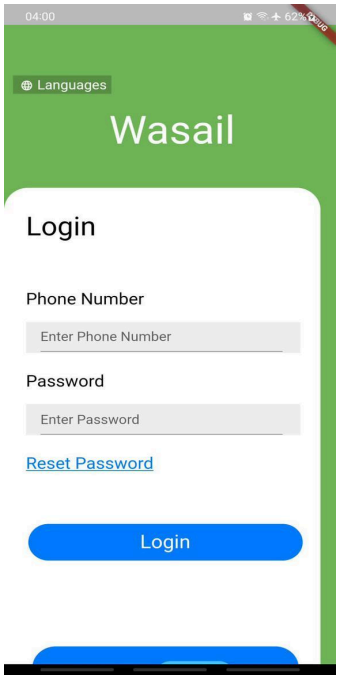


Figure 1.32 shows the Login Page in the early stage being unresponsive on a different device

Since two people were working on app development, the issue of responsiveness immediately became apparent and was resolved early on. The very first page, Login, designed showed a lack of responsiveness on a different device.

```
@override
Widget build(BuildContext context) {
  double screenWidth = MediaQuery.of(context).size.width;
  double screenHeight = MediaQuery.of(context).size.height;
      return MaterialApp(
      ├─ home: Scaffold(
        ├─ appBar: AppBar(
        │   backgroundColor: Color(0xFF6FB457),
        │  ─ title: Padding(
        │     padding: const EdgeInsets.only(left: 145),
        │    ─ child: Text('WASAIL'),
```

Figure 1.33 shows the code of MediaQuery implementation

The responsive design tool in Flutter creates adaptive layouts by dynamically adjusting the user interface to accommodate various device characteristics such as screen sizes, resolutions, and orientations. To do it, the figure above demonstrates. Media Query and Responsiveness (Figure 1.32 to 1.33): Allows the application to handle responsiveness across various devices. Visual Personalization - Dark-Bright Mode: Conducted experiments with dark-bright mode functionality. We briefly experimented with theme variation as shown in one of the weekly submissions by having a dark-light mode toggle in the app, however for FYP I, we decided to leave it out.

Phase 3: Front-End and Back-End Integration

In this phase of mobile development, working towards integrating the front-end and back-end began, incorporating functionalities aligned with functional requirements. The final product shown in the first stage as a visual demo of the app was possible due to REST API Integration and HTTP protocols using JSON encoding/decoding.

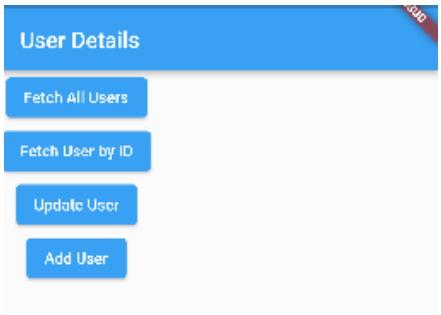


Figure 1.34 shows the code of MediaQuery implementation

To become familiar, we initially implemented a basic User CRUD application in Flutter as shown in Figure 1.34.

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;
```

Figure 1.35 shows which imports were used for integration

Integrating required use of imports, libraries of Flutter, as seen in Figure 1.35 above, which were used in JSON encoding and decoding as well as HTTP protocol functions; GET, UPDATE, etc.

```dart
Future<void> _fetchAndDisplayProducts(String query) async {
  final response = await http.get(
    Uri.parse('http://10.0.2.2:3000/api/product/searchproduct/$query'),
  );

  if (response.statusCode == 200) {
    final List<dynamic> data = jsonDecode(response.body);
    setState(() {
      suggestions = data.map((item) => InventoryItem.fromJson(item)).toList();
    });
  } else {
    print('Failed to load products');
  }
}
```

Figure 1.36 shows a GET operation on the product table for inventory management

The operations looked as above in Figure 1.36 where Uri are the endpoints where information is being handled in the backend/database.

```dart
factory InventoryItem.fromJson(Map<String, dynamic> json) {
  return InventoryItem(
    productId: json['product_id'],
    name: json['product_name'],
    imageUrl: json['image'],
  );
}
}
```

Figure 1.37

```dart
factory productInventory.fromJson(Map<String, dynamic> json) {
  return productInventory(
    productInventoryId: json['product_inventory_id'],
    price: json['price'],
    availableAmount: json['available_amount'],
    listedAmount: json['listed_amount'],
    vendorVendorId: json['vendor_vendor_id'],
    productProductId: json['product_product_id'],
  );
}
}
```

Figure 1.38

The tables involved JSON decoding and encoding which had syntax as shown in the Figures above for two different tables. Figure 1.37 is decoding an item from inventory and Figure 1.38 is encoding a product into the product inventory.

We successfully integrated REST APIs using Flutter, enabling real-time data updates and dynamic content, and implemented HTTP protocols and JSON encoding/decoding for standardised data transmission. However, there was a problem with endpoints, where we learned that the ones for emulators (Flutter's

simulation of a Mobile Phone) and for local hosts vary. We were able to resolve the issue and implement operations for updating, deleting, and managing user and product information as seen earlier.

Phase 4: Final Front-End Design

After experimenting with different designs for the navigation bar and colour schemes, we concluded that the most effective choice for the vendor app was an orange colour theme. This decision was made to ensure clear differentiation between the vendor app and other apps within the platform.

Figure 1.39 English

Figure 1.40 Urdu

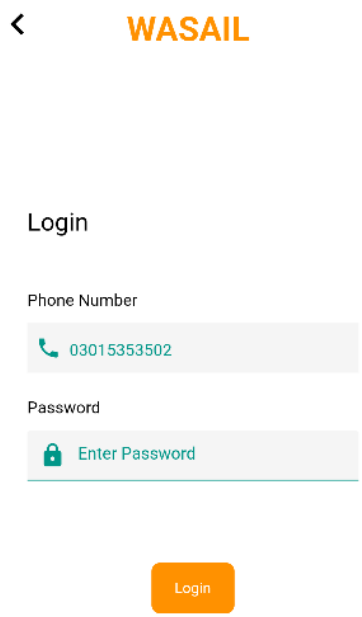


Figure 1.41 Login

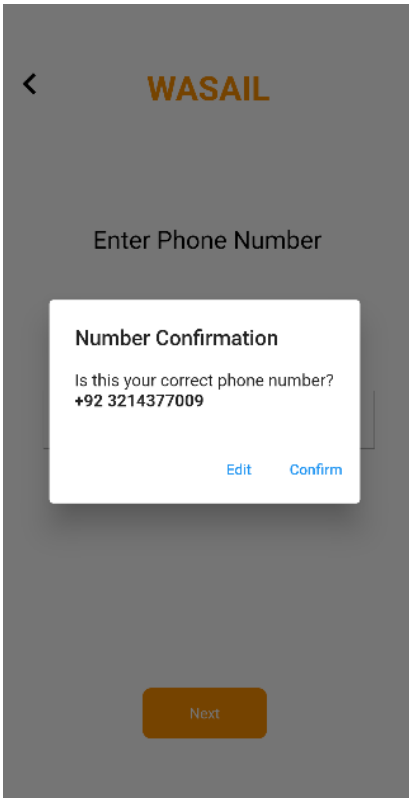


Figure 1.42 Confirmation

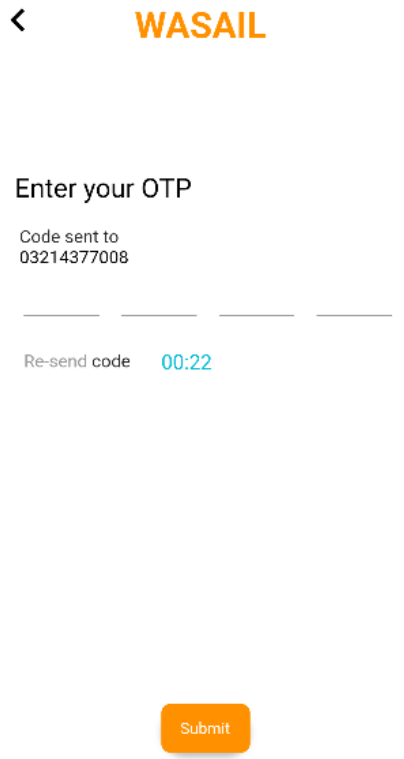


Figure 1.43 OTP

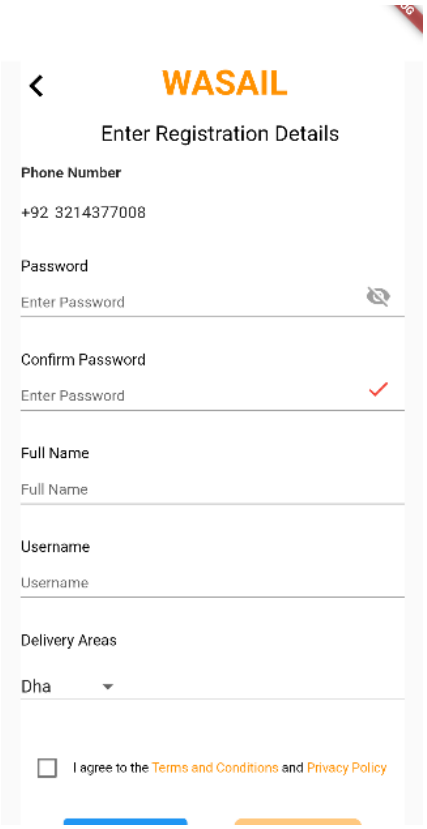


Figure 1.44 Registration

The use of orange as the primary colour not only adds vibrancy and visual appeal but also helps users easily distinguish the vendor app from other applications, enhancing the overall user experience and usability of the platform as shown in Figure 1.39 to Figure 1.44.

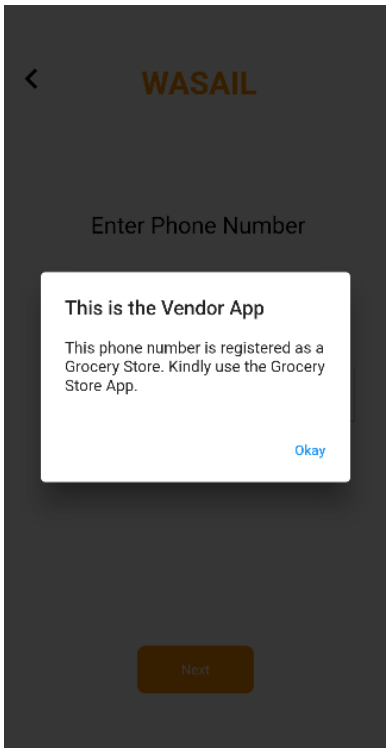


Figure 1.45 Error Message

Vendors are restricted from logging into the grocery store app, and vice versa. This means that if a phone number is already registered with the vendor app, it cannot be used to register for the store app. This separation of user accounts ensures distinct and secure access for vendors and store users, maintaining the integrity of each platform and preventing unauthorized access across different app functionalities as shown in the above Figure 1.45.

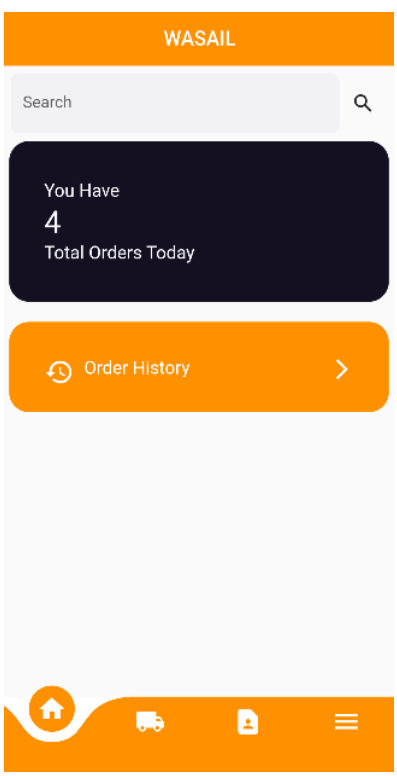


Figure 1.46 Home Page

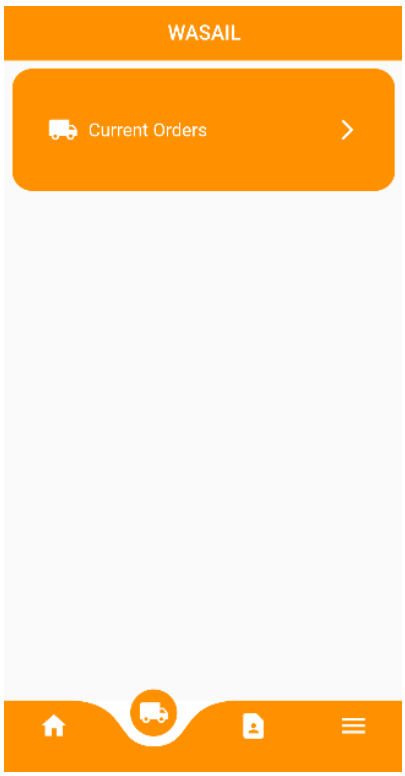


Figure 1.47 Current Orders

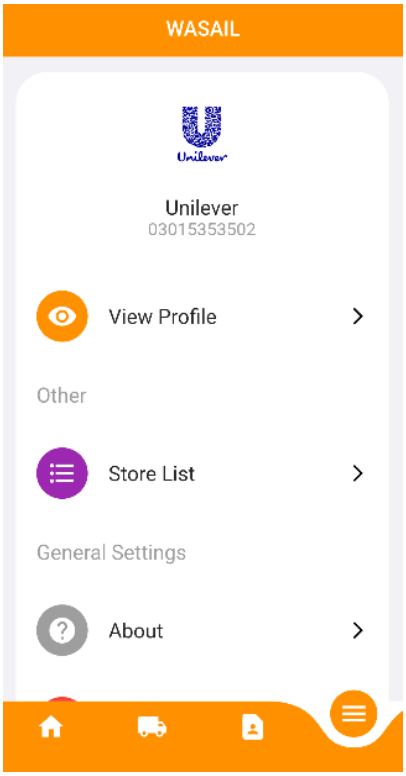
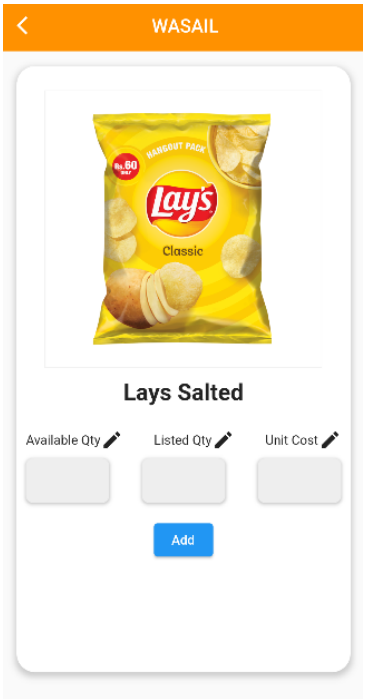
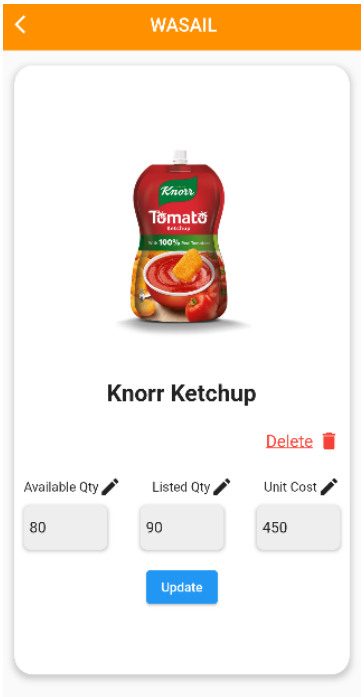


Figure 1.48 Menu

Additionally, we implemented a curved navigation bar design for the vendor app. This curved navigation bar not only adds a modern and sleek look to the app but also enhances user navigation by providing a more intuitive and user-friendly interface. This navigation can be seen in Figure 1.46 to Figure 1.48.





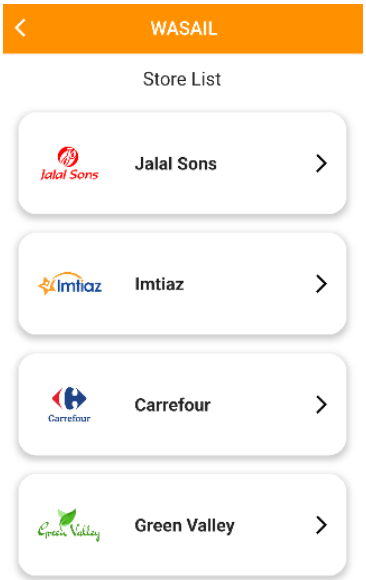


Figure 1.49 Add Product **Figure 1.50** Edit Product **Figure 1.51** Store List

Additionally, changes were made to the pages to align them more closely with the prototype. These adjustments aimed to ensure consistency and accuracy in the app's design, bringing it closer to the envisioned prototype's layout and functionality.These changes can be seen in the above Figure 1.49 to Figure 1.51.

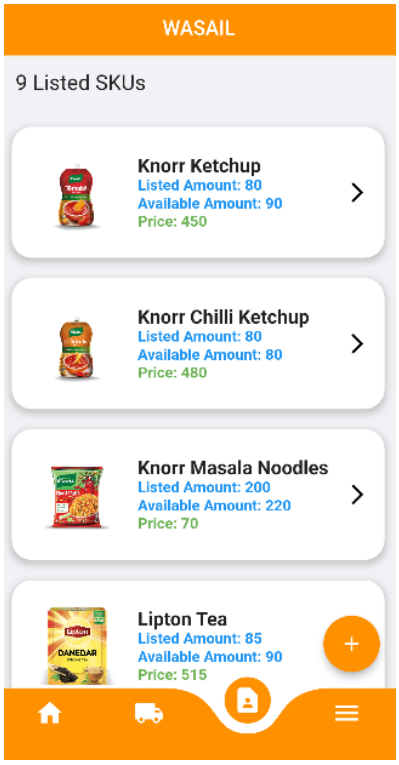


Figure 1.52 Inventory

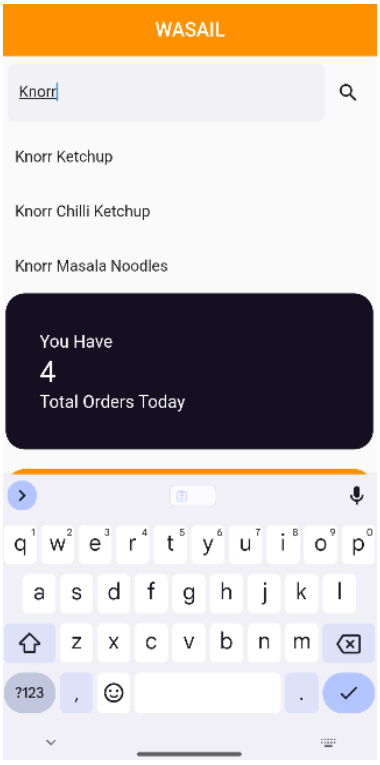


Figure 1.53 Search Inventory

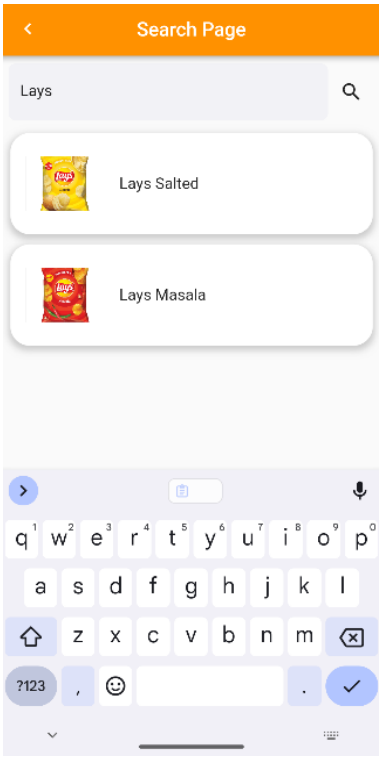


Figure 1.54 Search Product

Visual enhancements were implemented to improve the overall appearance and user experience of the app. These changes focused on refining the design elements to create a more appealing and cohesive visual presentation throughout the app as can be seen in Figure 1.52 to Figure 1.54.

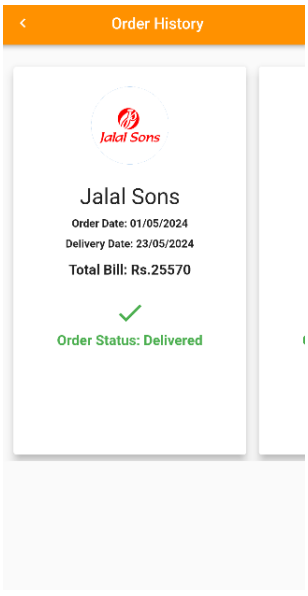


Figure 1.55 order History

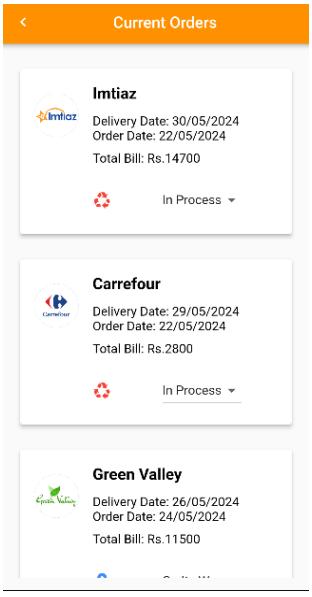


Figure 1.56 Current Orders

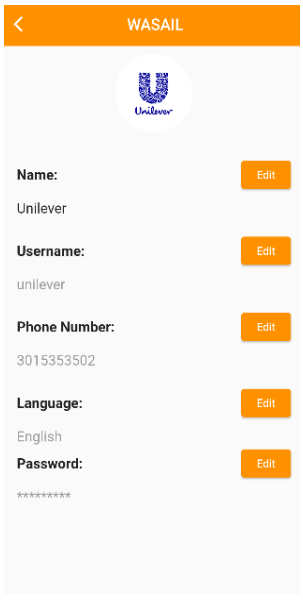


Figure 1.57 Edit profile

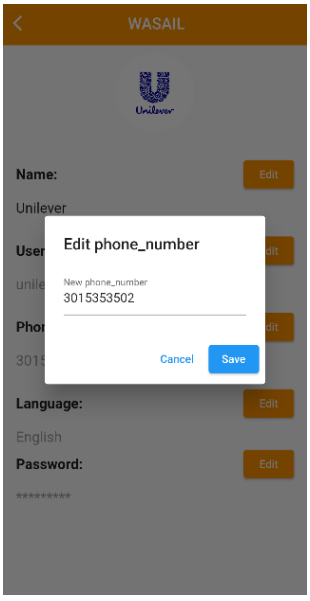


Figure 1.58 Edit Number

Various other changes were also implemented during this period. These modifications encompassed a range of aspects such as functionality enhancements, performance optimizations, and bug fixes, aimed at improving the overall quality and user satisfaction of the app as can seen in above Figure 1.55 to Figure 1.58.

Grocery Store App

In FYP II, our primary focus was on crafting a seamless and user-friendly grocery store app, in line with the objectives outlined in our project deliverables. The development journey was structured into phases that corresponded with the functional requirements (FR), guiding the functionality and design of the app specifically tailored for grocery store operations. To differentiate between the two apps we chose a green colour for our store app.

Similar to our vendor app development, we continued to leverage Flutter, Google's UI toolkit, for building the grocery store app. This technology choice allowed us to maintain a single codebase that could run seamlessly on both iOS and Android platforms. By utilizing Flutter, we aimed to streamline the development process and ensure a consistent user experience across different devices for grocery store owners

Phase 1: Initial Front-End Design

Initially, this phase of the development process involved experimenting with the app to display all the content first. The primary focus was to ensure that all necessary information and features were visible, while other adjustments and refinements were made concurrently. This approach allowed for a clear understanding of the app's layout and functionality before making more detailed changes.

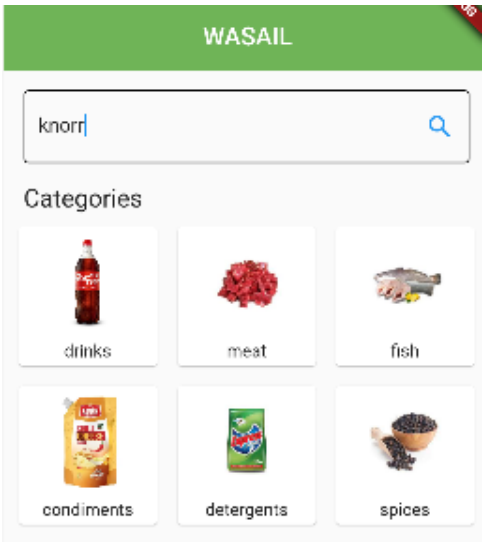


Figure 1.59 Search Product

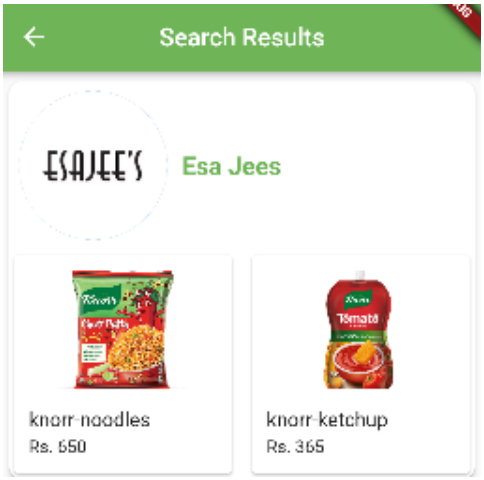


Figure 1.60 Search Results

When designing the home page of the vendor app, the team initially integrated the front end with the back end. During this phase, we focused on implementing functionalities related to categories and a search bar for searching inventory items as shown in above Figure 1.59 and Figure 1.60

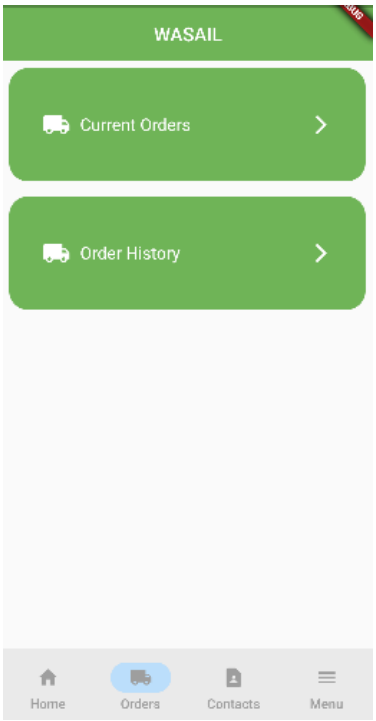


Figure 1.61 Current Orders & Order History Sections

Following the initial implementation of the home page,we proceeded to develop the 'Current Orders' page and 'Order History' page. These pages were made accessible from the Orders Section within the

app's navigation bar, as illustrated in Figure 1.61. We provided vendors with the capability to view their current orders and order history seamlessly. The pages were designed to fetch and display relevant order data specific to the vendor, including details such as delivery dates, order statuses, and total bills.

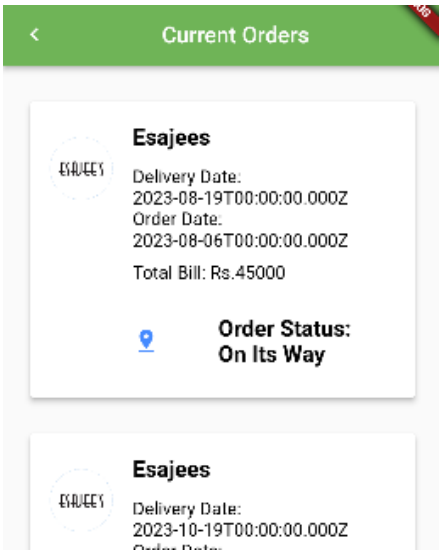


Figure 1.62 Current Orders

The page dynamically generated a user-friendly interface, displaying each order with pertinent information and visual indicators for quick status checks. Overall, it provided vendors with tools to monitor ongoing transactions efficiently, facilitating streamlined operations and customer service as shown in Figure 1.62.

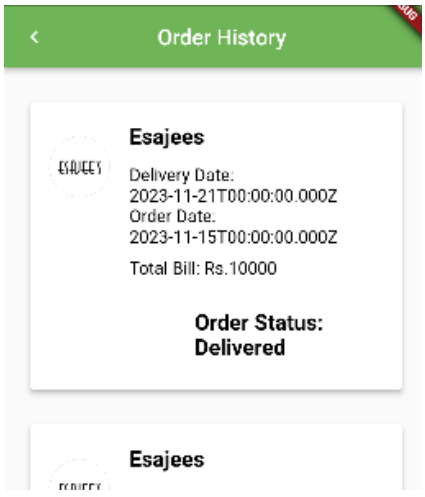


Figure 1.63 Order History

Subsequently, the page dynamically constructed a user-friendly interface, Figure 1.63, presenting each past order as a list item with relevant information. Visual cues, such as icons, are used to offer quick

insights into the status of each transaction. Ultimately, the 'Order History' page is to display vendors with insights into their previous transactions with the grocery store.

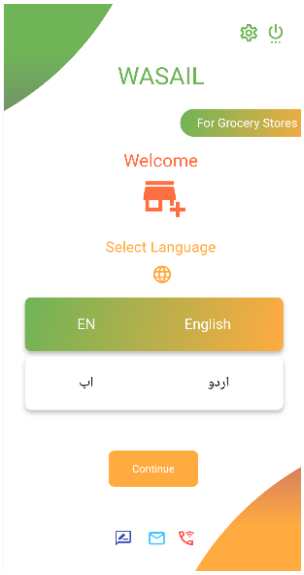


Figure 1.64 Languages

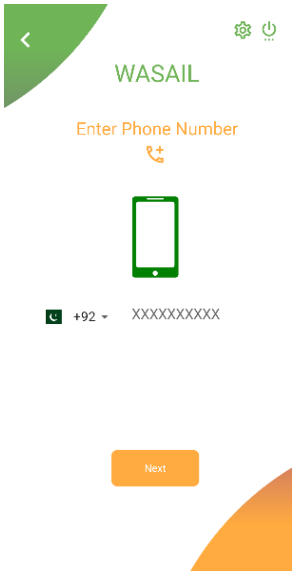


Figure 1.65 Phone Number

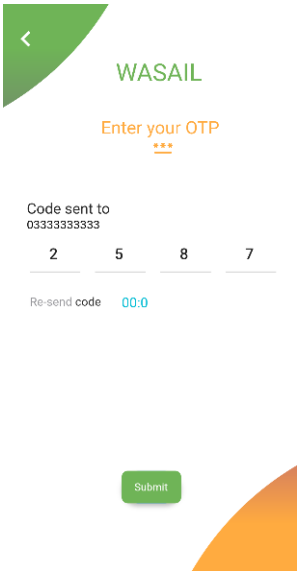
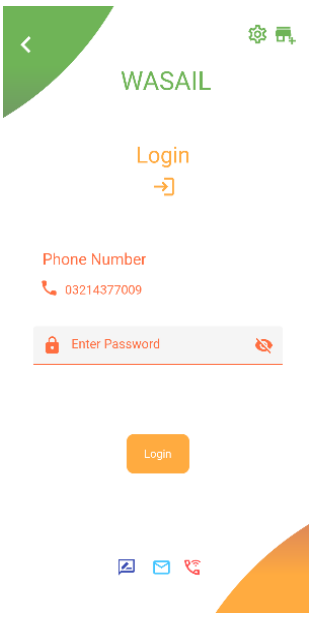
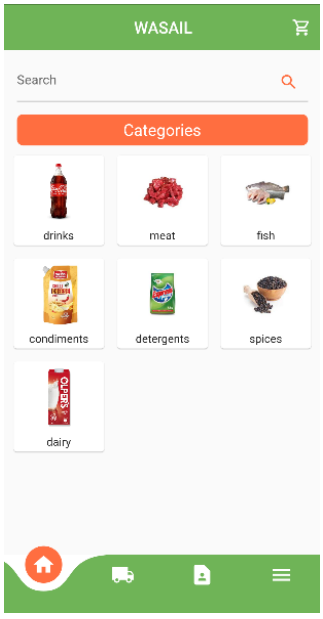


Figure 1.66 OTP





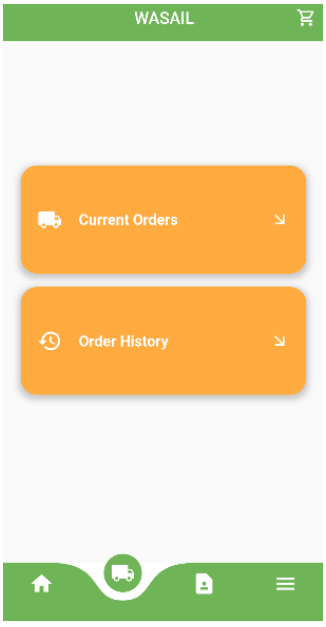


Figure 1.67 Login

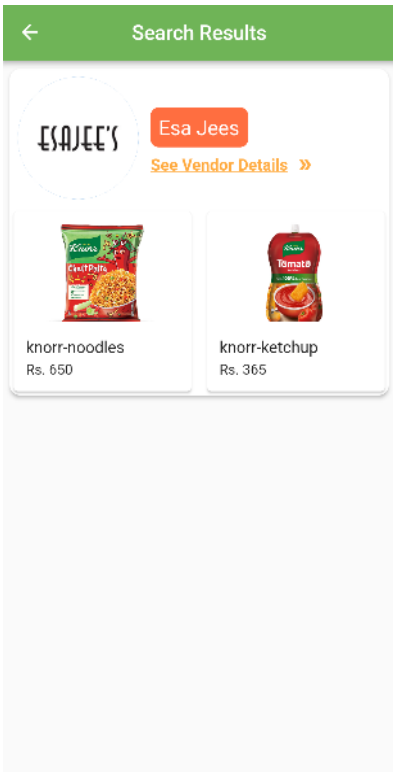


Figure 1.68 Home

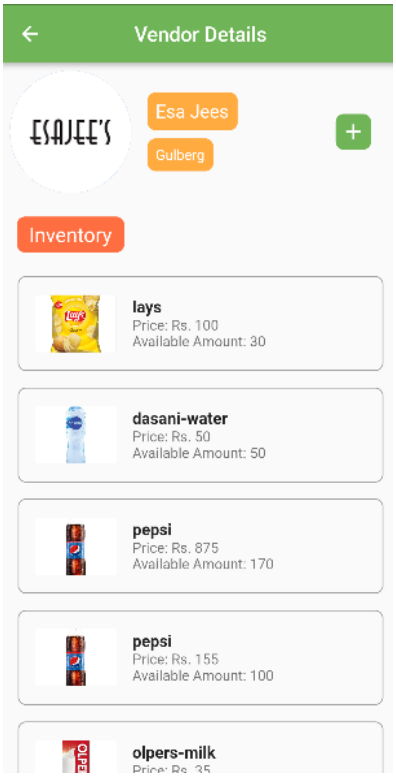


Figure 1.69 Orders

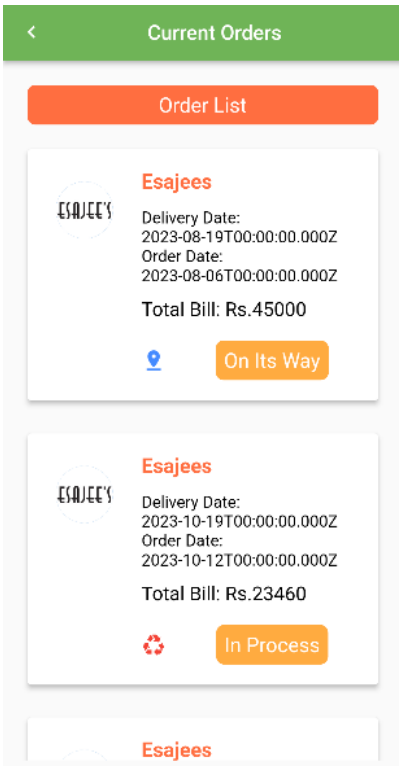


Figure 1.70 Results

Figure 1.71 Vendor Details

Figure 1.72 Current Orders

We experimented with various templates to find a suitable and efficient option for the Flutter app. Despite facing challenges, the focus remained on implementing low-level changes like adjusting colour schemes and templates to inject a more vibrant look into the app. This experimentation aimed to test different design elements and assess their impact on the overall user experience. Recognizing the importance of thorough testing and refinement as shown in above Figures 1.64 to Figure 1.72.

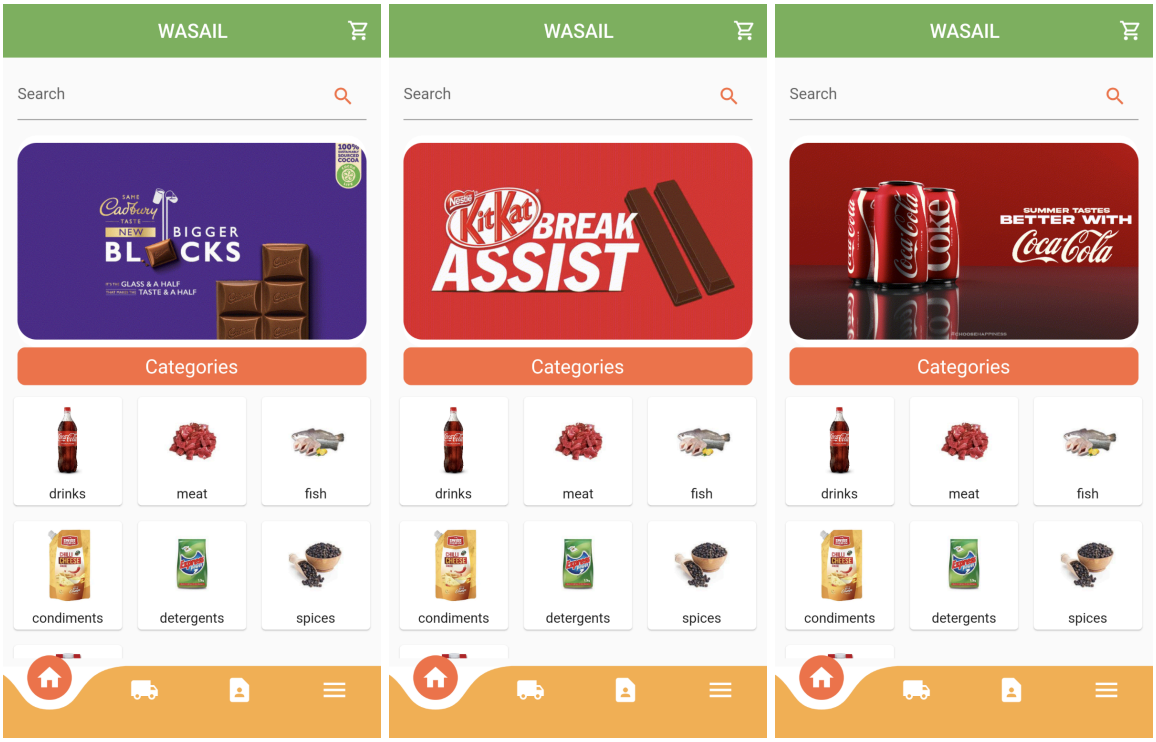


Figure 1.73, **Figure 1.74**, and **Figure 1.75** (Home Page)

Initially, when building the store app, the focus was also on its appearance. The navbar was completely revamped by incorporating the CurvedNavigationBar component, enhancing its visual appeal and providing a more modern and attractive look. This change not only adds aesthetic value but also improves the overall user experience by offering a visually appealing navigation experience. Additionally, various colours were experimented with to further elevate the app's design, ensuring it is both visually striking and user-friendly.At this stage the colour for the both apps were being decided to distinguish between them as shown in Figure 1.73 to 1.75.

Phase 2: App Development Progress

Transitioning to mobile app development using Flutter, key features were implemented to align with the functional requirements specified in Phase I. Although not visible in the provided screenshots, these functionalities were integral to this phase. While the prototype served as a foundational reference, the app gradually evolved into the final product as depicted.

```
Expanded( // Use Expanded to allow the text to take available space
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Text(
        "${productItem.name}",
        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 19),
        maxLines: 2, // Limit to 2 lines
        overflow: TextOverflow.ellipsis, // Handle overflow with ellipsis
      ),  // Text
      Text(
        "${AppLocalizations.of(context)!.listed_amount}: ${item.listedAmount}",
        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 15,color: Colors.blue),
      ),  // Text
      Text(
        "${AppLocalizations.of(context)!.available_amount}: ${item.availableAmount}",
        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 15,color: Colors.blue),
      ),  // Text
      Text(
        "${AppLocalizations.of(context)!.price}: ${item.price}",
        textDirection: TextDirection.ltr,
        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 15,color:Color(0xFF6FB457)),
```

Figure 1.76

Seen above in Figure 1.76 is the Flutter Implementation of the SKU tiles shown above in Inventory List.

itemdetails.dart | inventory.dart

```
}

Widget _buildSmallShadowBoxTextField(TextEditingController controller) {
  return Container(
    width: 90,
    decoration: BoxDecoration(
      color: Colors.grey[200],
      borderRadius: BorderRadius.circular(8.0),
      boxShadow: [
        BoxShadow(
          color: Colors.grey.withOpacity(0.5),
          spreadRadius: 1,
          blurRadius: 2,
          offset: Offset(0, 1),
        ),  // BoxShadow
      ],
    ),  // BoxDecoration
    child: TextFormField(
      controller: controller,
      decoration: InputDecoration(
        border: InputBorder.none,
        contentPadding:
        EdgeInsets.symmetric(horizontal: 6.0, vertical: 8.0),
      ),  // InputDecoration
    ),  // TextFormField
  );  // Container
```

Figure 1.77

Seen above in Figure 1.77 is the Flutter Implementation of the text fields for input shown above in Update Page.

These little changes made a significant impact in this phase of the app development process. They contributed to enhancing the app's functionality, usability, and overall user experience, marking substantial progress in the journey toward creating the final product.

```
ProductDetailsPage({required this.product});

@override
Widget build(BuildContext context) {
  TextEditingController listedAmountController = TextEditingController();
  TextEditingController availableAmountController = TextEditingController();
  TextEditingController priceController = TextEditingController();

  return Scaffold(
    appBar:  AppBar(...),  // AppBar
    body: SingleChildScrollView(
      padding: EdgeInsets.all(16.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Container(
            height: 655, // Increased height of the background card
            width: 370,
            decoration: BoxDecoration(
              color: Colors.white,
              borderRadius: BorderRadius.circular(20),
              boxShadow: [
```

Figure 1.78

Seen above in Figure 1.78 is the Flutter Implementation of the text fields for input shown above in the Add To Inventory Page.

Phase 3: Front-End and Back-End Integration

During this phase of mobile development, efforts were concentrated on seamlessly integrating the front-end and back-end, ensuring that all functionalities met the specified requirements. The initial visual demo of the app was successfully achieved through the implementation of REST APIs and the utilization of HTTP protocols for JSON data exchange. This integration was crucial in bringing the app to life and demonstrating its core features effectively.

```
Future<Map<String, dynamic>> fetchVendorProfile(int vendorId) async {
  final response = await http.get(
      Uri.parse('http://10.0.2.2:3000/api/vendor/vendorprofile/$vendorId'));

  if (response.statusCode == 200) {
    final Map<String, dynamic> responseBody = jsonDecode(response.body);
    return responseBody;
  } else {
    throw Exception('Failed to load vendor profile');
  }
}
```

Figure 1.79 HTTP Function Vendor Profile

The front-end and back-end integration for fetching vendor profiles as seen above was done through HTTP protocol as usual, the code snippet for it is attached above, Figure 1.79.

```
final url =
    'http://10.0.2.2:3000/api/list/addvendorlist/$storeId/$vendorId';

try {
  final response = await http.post(Uri.parse(url));

  if (response.statusCode == 200) {
    setState(() {
      isVendorAdded = true;
    });
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text('Vendor added to your list'),
      ),  // SnackBar
    );
  } else if (response.statusCode == 409) {
    setState(() {
      isVendorAdded = true;
    });
    ScaffoldMessenger.of(context).showSnackBar(
```

Figure 1.80 HTTP Function for Adding Vendor

The HTTP function adding a vendor to 'Vendor List' implemented above can be seen in the attached Figure 1.80 above which uses the logic of adding selected vendor through $vendorId to the *vendor list* of the grocery store signed in/adding, taken from the $storeId implemented on back-end.

```
Future<List<Map<String, dynamic>>> _fetchAndDisplayCombinedData(int vendorId) async {
  try {
    final response = await http.get(
      Uri.parse('http://10.0.2.2:3000/api/order/storecurrentorder/$vendorId'),
    );


    if (response.statusCode == 200) {
      final List<dynamic> data = jsonDecode(response.body);
      List<Map<String, dynamic>> combinedData = [];

      for (final item in data) {
        final orderDate = item['order_date'];
        final deliveryDate = item['delivery_date'];
        final totalBill = item['total_bill'];
        final orderStatus = item['order_status'];
        final groceryStoreId = item['groceryStoreStoreId'];

        if (groceryStoreId != null) {
          print('Fetching grocery store data for ID: $groceryStoreId');

          final groceryStoreResponse = await http.get(
            Uri.parse('http://10.0.2.2:3000/api/grocery_store/$groceryStoreId'),
```

Figure 1.81 HTTP Request for Current Orders

As seen in Figure 1.81, are the current orders for the vendor ‘Esa Jees’ by the grocery stores.

The 'Current Orders' page serves as a dashboard for vendors, fetching and presenting ongoing order details associated with the vendor's ID. It initiates an HTTP request, Figure 1.81, to retrieve current order data, parsing essential information like order dates, delivery dates, total bills, and statuses. Additional requests are made for associated grocery store details.

```
try {
  final response = await http.get(
    Uri.parse('http://10.0.2.2:3000/api/order/storeorderhistory/$vendorId'),
  );


  if (response.statusCode == 200) {
    final List<dynamic> data = jsonDecode(response.body);
    List<Map<String, dynamic>> combinedData = [];

    for (final item in data) {
      final orderDate = item['order_date'];
      final deliveryDate = item['delivery_date'];
      final totalBill = item['total_bill'];
      final orderStatus = item['order_status'];
      final groceryStoreId = item['groceryStoreStoreId'];

      if (groceryStoreId != null) {
        print('Fetching grocery store data for ID: $groceryStoreId');

        final groceryStoreResponse = await http.get(
          Uri.parse('http://10.0.2.2:3000/api/grocery_store/$groceryStoreId'),
```

Figure 1.82 HTTP Request for Order History

The 'Order History' page offers vendors a comprehensive view of their past transactions, aiding in business analysis and decision-making. Upon accessing the page, it sends an HTTP request to the server API endpoint designated for retrieving historical order data linked to the vendor's ID, as seen in Figure 1.82 above. Following a successful response, the page processes the received data, extracting vital details such as order dates, delivery dates, total bills, and order statuses.

Phase 4: Final Front-End Design

In this final phase, meticulous attention was given to ensuring consistency throughout the app. The integration between front-end and back-end was made seamless, functionality was thoroughly verified, and a cohesive theme was maintained across all pages. This effort was crucial to delivering a polished and unified user experience.

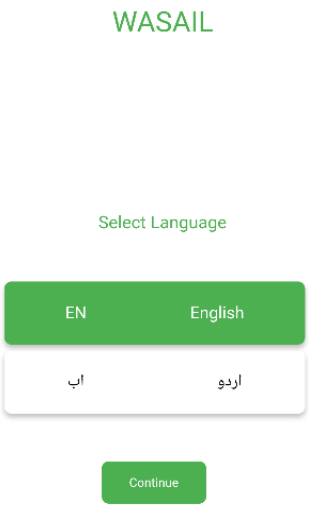


Figure 1.83 Languages

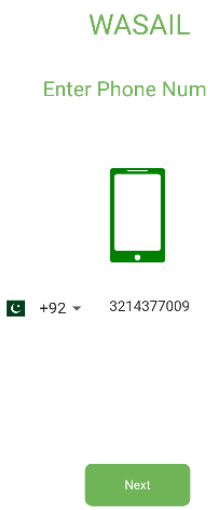


Figure 1.84 Phone Number

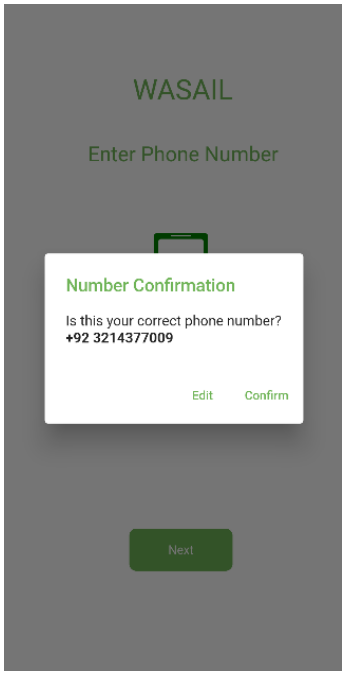


Figure 1.85 Confirmation

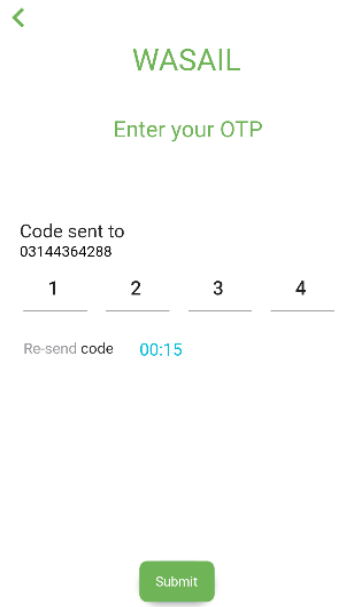


Figure 1.86 OTP

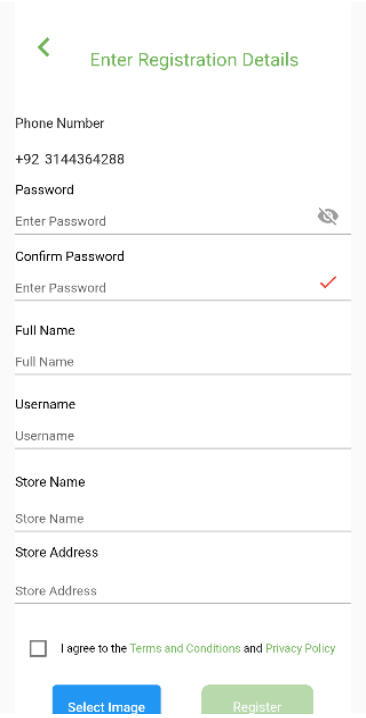


Figure 1.87 Registration

The registration and login process for the store app mirrors the one used in the vendor app. Initially, the OTP process handles phone confirmation and checks to distinguish between new and existing users. New users' phone numbers are verified and subsequently registered, following the sequence outlined below. During registration, several validation checks are performed, including ensuring the username is unique, verifying the password contains special characters, confirming the password, and ensuring all required fields are filled out with information pertaining to vendors. Upon successful registration, users are redirected to the login page. Importantly, vendors cannot log in to the store app and vice versa, ensuring a clear distinction and secure access control between different user roles. This streamlined process ensures both the vendor app and the store app maintain a consistent and secure method for user onboarding and authentication as shown in Figure 1.83 to Figure 1.87.

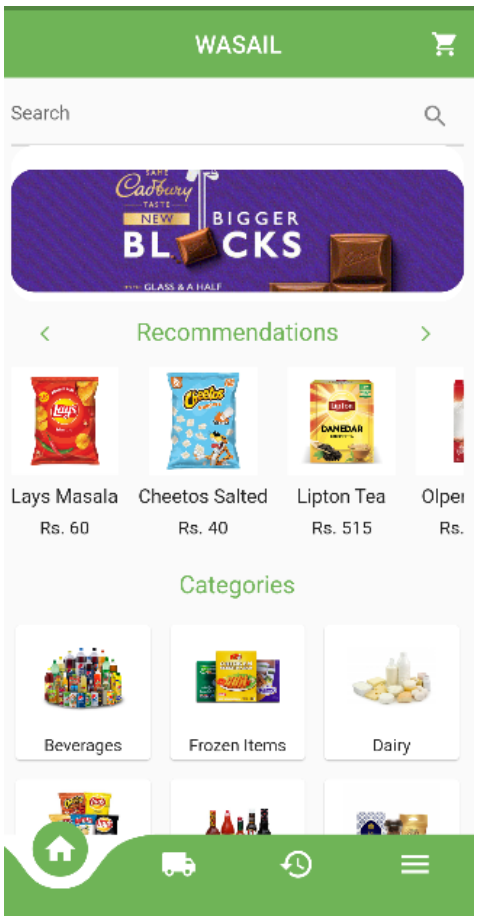


Figure 1.88 Home

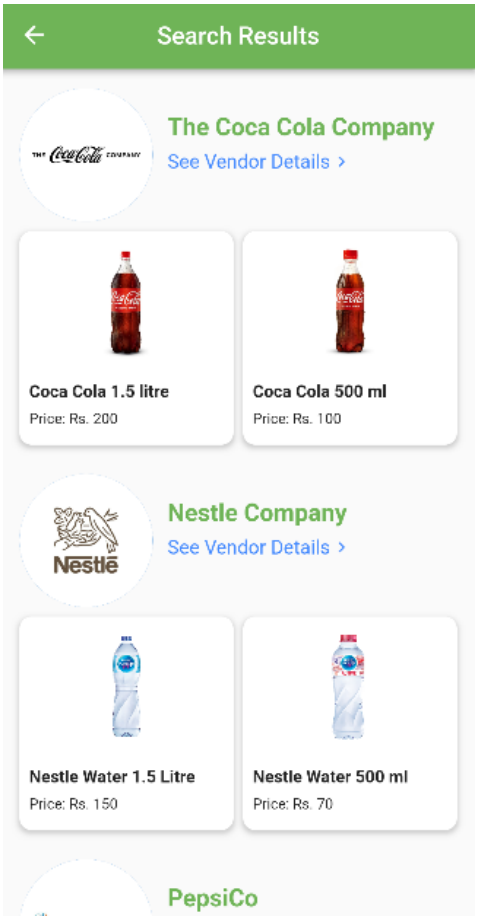


Figure 1.89 Search Results

After successfully registering, new users are directed to the login screen. Entering the correct credentials will then grant access to the Home Page of the app. On the Home Page, there is a search bar that allows users to search for items already in their inventory as shown in Figure 1.88. Clicking on a searched item directs users to the inventory page. Additionally, the Home Page features a scrollable list of recommendations and various categories. Clicking on a category displays different vendors' profiles, with items listed according to each category as shown in Figure 1.89 Navigation within the app is managed through the navigation bar, which seamlessly guides users throughout their journey within the app.

There is also a dedicated section for vendors to advertise their products to the grocery stores as shown in Figure 1.88. This feature allows vendors to showcase their products directly to potential buyers, enhancing visibility and facilitating business growth. Through this section, vendors can create enticing advertisements, provide detailed product information, and reach out to a targeted audience of grocery store owners and managers. This platform fosters collaboration and partnership opportunities between vendors and grocery stores, promoting a thriving ecosystem within the industry.

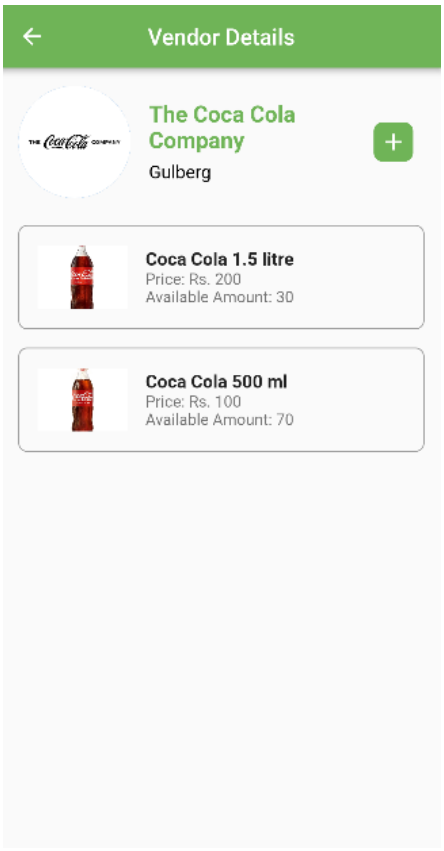


Figure 1.90 Vendor Details

Upon clicking a vendor's profile, users can view the vendor's details, such as the items in their inventory, name, and location. Additionally, users have the option to add the vendor if they are not already in the system. If the vendor is already added, a pop-up notification will appear on the screen indicating that the vendor already exists as shown in the above Figure 1.90.

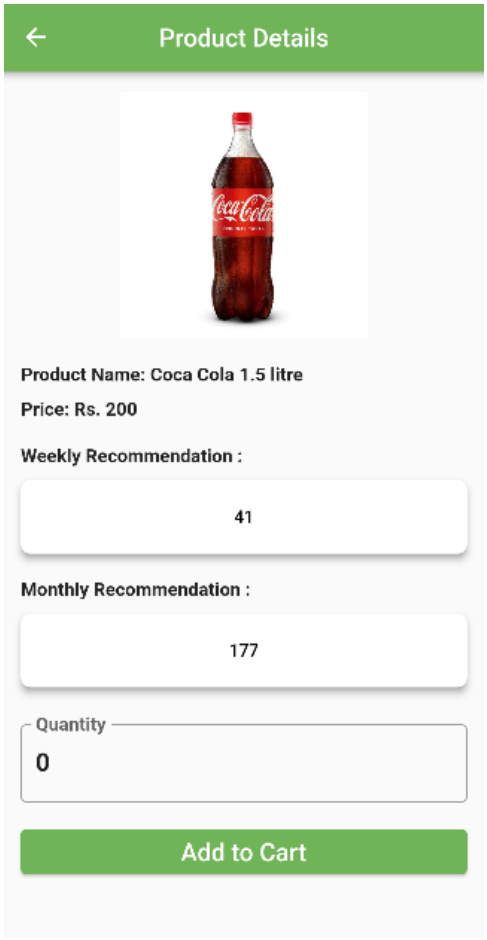


Figure 1.91 Vendor Details

Clicking on the items listed under each vendor opens the product details page, where users can view the product image, name, and price. Additionally, the app provides weekly and monthly recommendations generated by our models. Users have the option to select these recommendations or manually enter the quantity they desire. Upon clicking "add to cart," the item is added to the user's cart, making it convenient to track and manage selected items for purchase as shown in Figure 1.91.

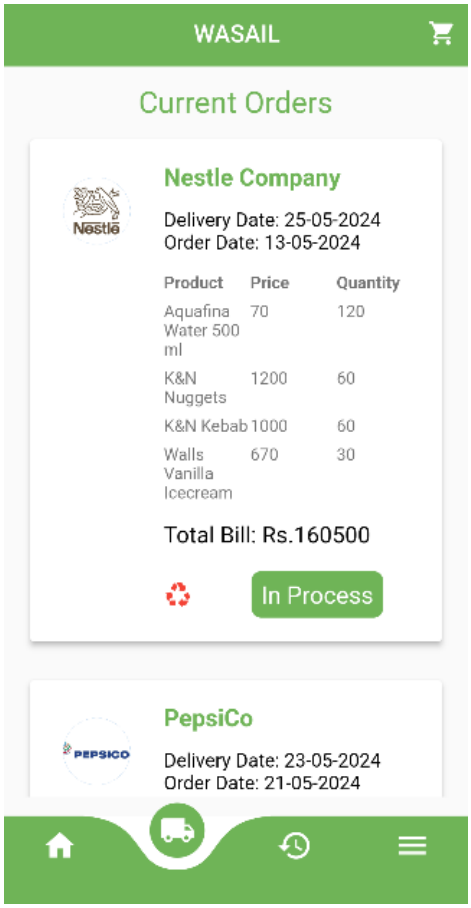


Figure 1.92 Current Orders

In our current orders page, users can view a comprehensive list of their orders, including those that are in process or on their way for delivery. This feature provides users with real-time updates on the status of their orders, ensuring transparency and convenience throughout the purchasing process. The order tracking functionalities are seamlessly integrated and controlled by the vendors through the vendor app. Vendors can update order statuses, such as order processing, shipping, and delivery, directly through their app interface. This collaborative approach between users and vendors enhances communication and efficiency in managing orders, leading to a smoother and more satisfying shopping experience for all parties involved as shown in the above Figure 1.92.

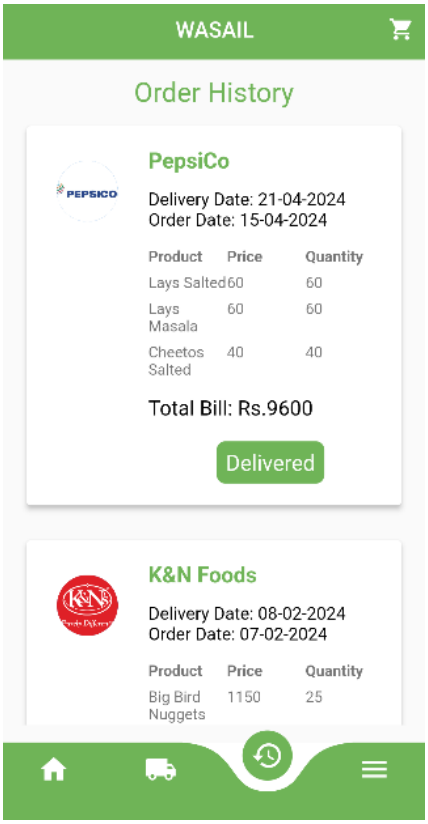


Figure 1.93 Order History

In the order history section, users can access a comprehensive record of their past orders, including both the order date and delivery date. This feature provides a detailed overview of completed orders. Each order entry includes essential information such as the order date, delivery date, items purchased, quantities, prices, and the status of the order at the time of purchase. Users can easily track their purchasing history, review past transactions, and monitor any updates or changes made to previous orders. This functionality enhances user experience by offering clear visibility into the timeline of their purchases and deliveries, aiding in better decision-making and management of account activities as shown in Figure 1.93.

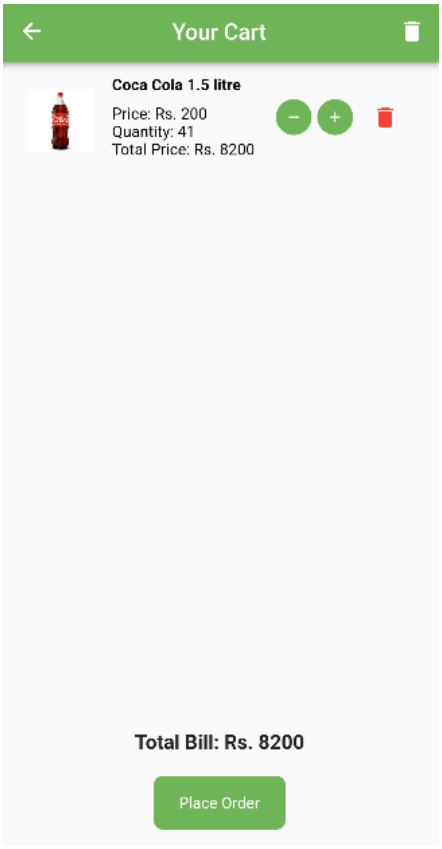


Figure 1.94 Cart

In our cart page, users can view the products they've added and adjust the quantity as needed. They have the option to increase or decrease the quantity of each item before placing the order as shown in Figure 1.94.Once the order is placed, vendors can access and view these orders. This seamless integration allows for efficient order management, ensuring that vendors can fulfil orders promptly and accurately.

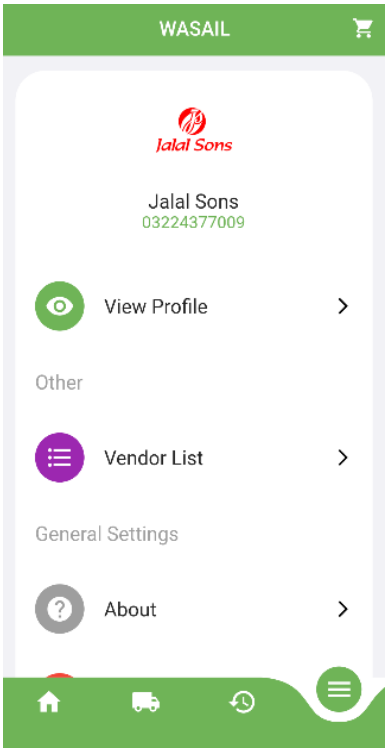


Figure 1.95 Menu

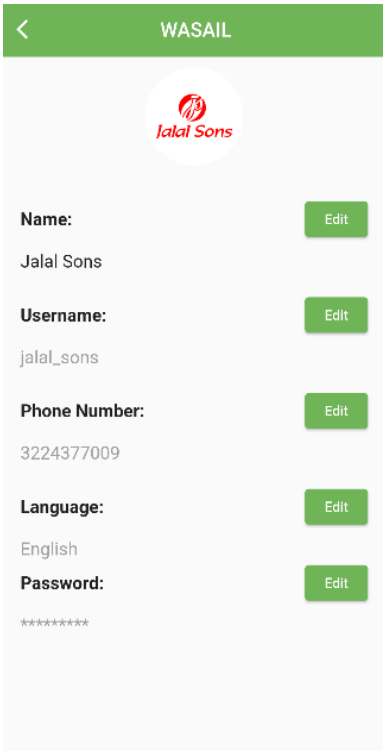


Figure 1.96 Profile

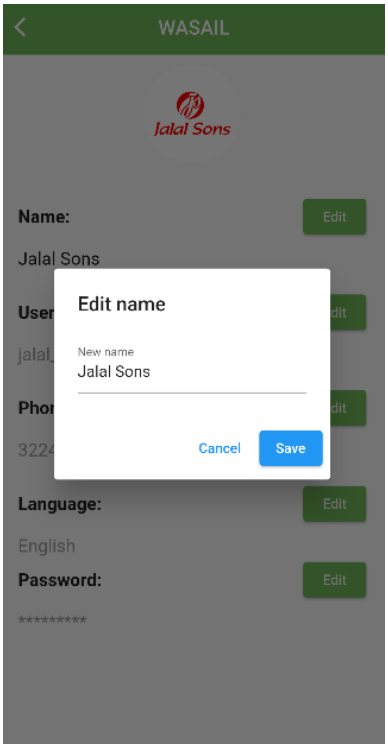


Figure 1.97 Edit

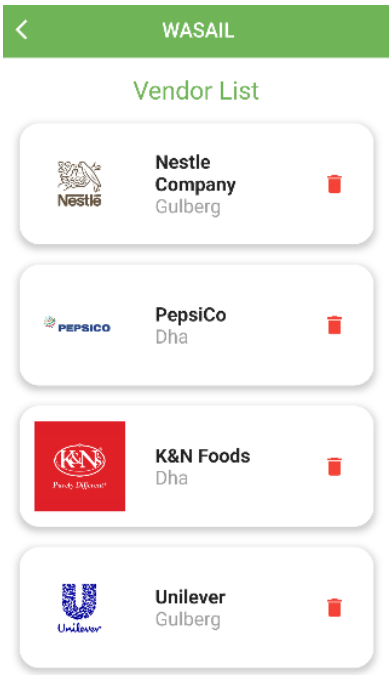


Figure 1.98 Cart

Finally ,On the menupage Figure 1.95, users can access their profile where they have the option to edit store name, username, phone number, password, and language settings as shown in Figure 1.96 and Figure 1.97. Additionally, there is a vendor list feature where users can view all added vendors. By clicking on the list, users can see a comprehensive list of vendors associated with their account and have the ability to delete vendors they no longer wish to keep in their list as shown in Figure 1.98. This functionality provides users with control over their store and account management, ensuring a personalised and efficient experience within the platform.

Admin Portal

Another focus for FYP II was the admin portal that is going to be used by our team. For this project, we leveraged Node.js as the core technology stack for backend development. Node.js offers a robust server-side runtime environment, facilitating efficient handling of administrative tasks and data management. Its event-driven architecture and non-blocking I/O operations ensured optimal performance and scalability, crucial for supporting our team's needs and requirements within the portal. Additionally, Node.js's extensive package ecosystem and libraries enabled seamless integration of various functionalities and services, enhancing the overall usability and functionality of the admin portal.

Phase 1: Initial Front-End Design

Since the admin portal is intended for use by the Wasail team, it serves as a centralized platform for managing various aspects of the business. Through this portal, we have the capability to handle user management, grocery stores, vendors, products, and categories seamlessly. Additionally, the admin portal provides powerful analytics tools that enable us to gather and analyze key data points, facilitating informed decision-making and strategic planning. This comprehensive functionality within the admin portal streamlines operations and enhances efficiency, empowering the Wasail team to effectively manage and optimise all aspects of the business from a single interface.

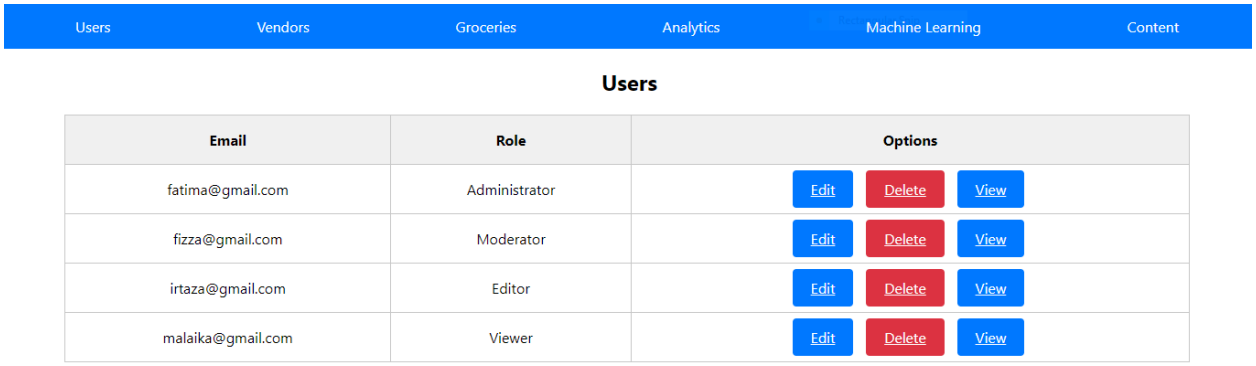


Figure 1.99 User Management

Initially, when designing the admin portal, the groundwork was laid to ensure a solid foundation for future development. One of the key aspects focused on was integrating a sleek and responsive navbar, enhancing the user experience and navigation within the admin section, as shown in Figure 1.99.

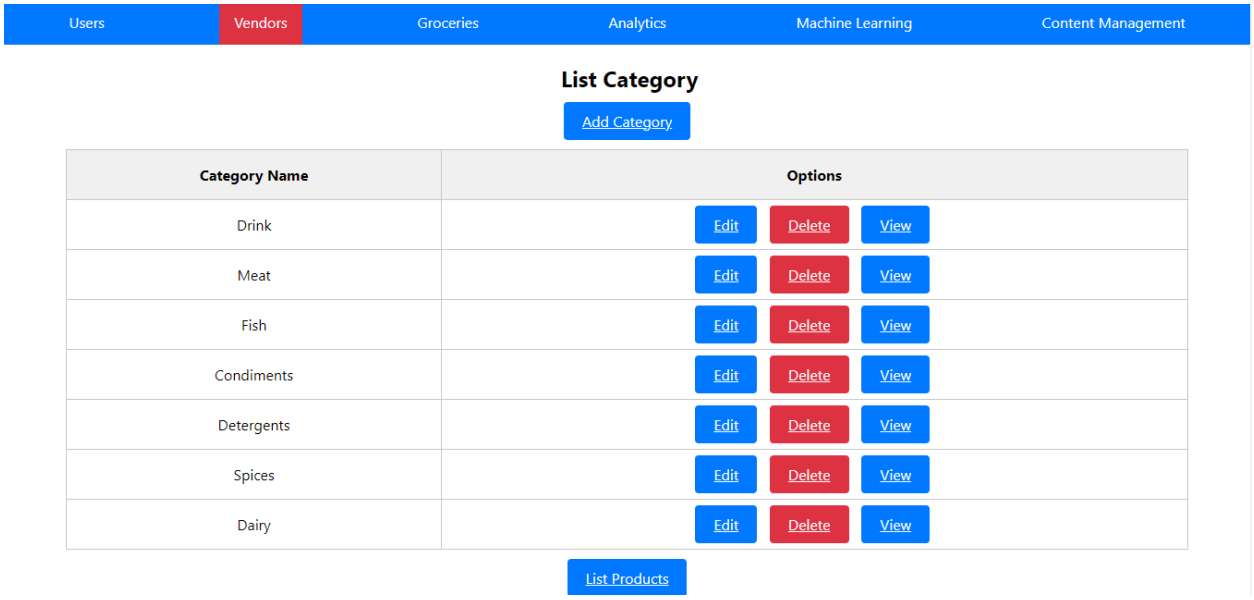


Figure 2.0 Content Management (List Category)

We created a list category page for our admin portal, maintaining consistency in the navbar design across the portal as shown in the above Figure 2.0.

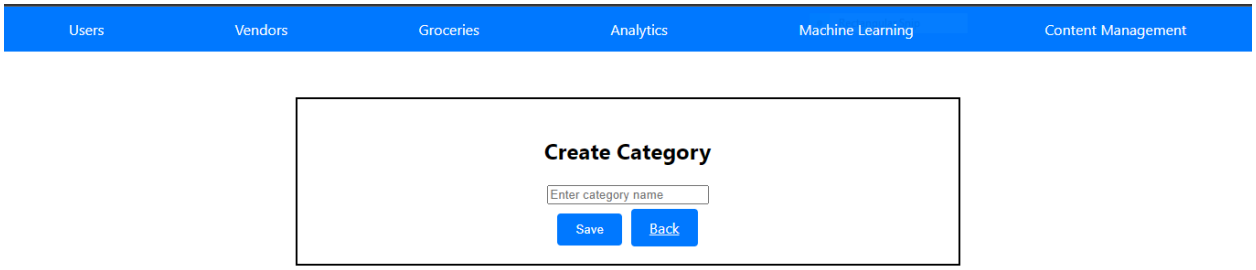


Figure 2.1 Content Management (Add Category)

We added a "Create Category" page to our admin portal. Users can input a category name into the provided text field. When they click the "Save" button, the entered category is saved in the backend database as shown in Figure 2.1.

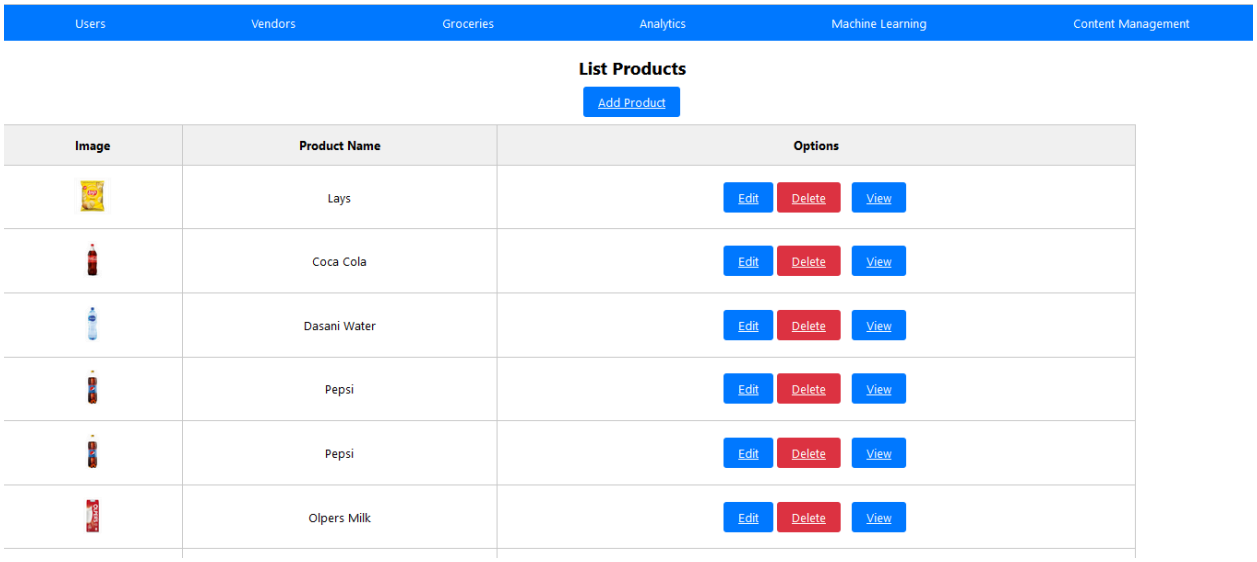


Figure 2.2 Content Management (List Product)

Referring to the product data being fetched from the database, it is done similarly to Category, as seen listed above. This page for our admin portal, also maintains consistency in the navbar design throughout as shown in Figure 2.2.

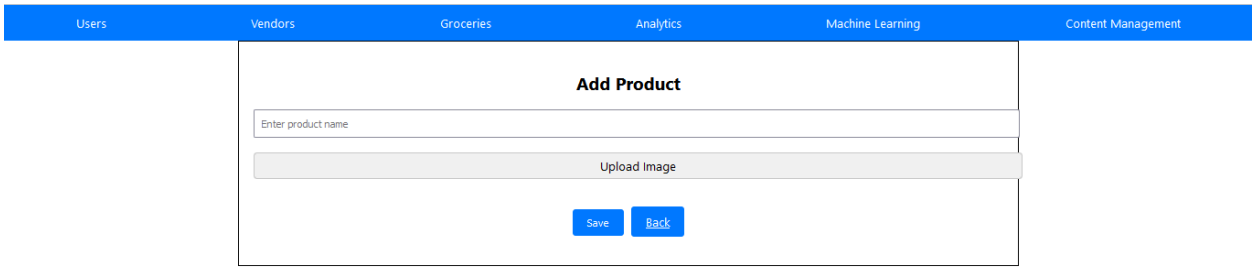


Figure 2.3 Content Management (Add Product)

Furthermore, we introduced a "Create Product" page to our admin portal. Users can input product details into the provided fields and upload images. Upon clicking the "Save" button, the entered product information is saved in the backend database as shown in Figure 2.3.

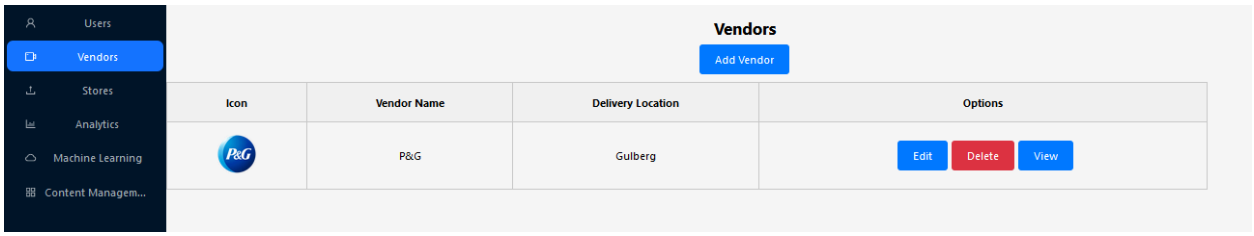


Figure 2.4 Vendor Listing

On the website, the vendor listing was displayed as shown in Figure 2.4 above.

Figure 2.5 Vendor Addition

On the website, the vendor addition was displayed as shown in Figure 2.5 above.

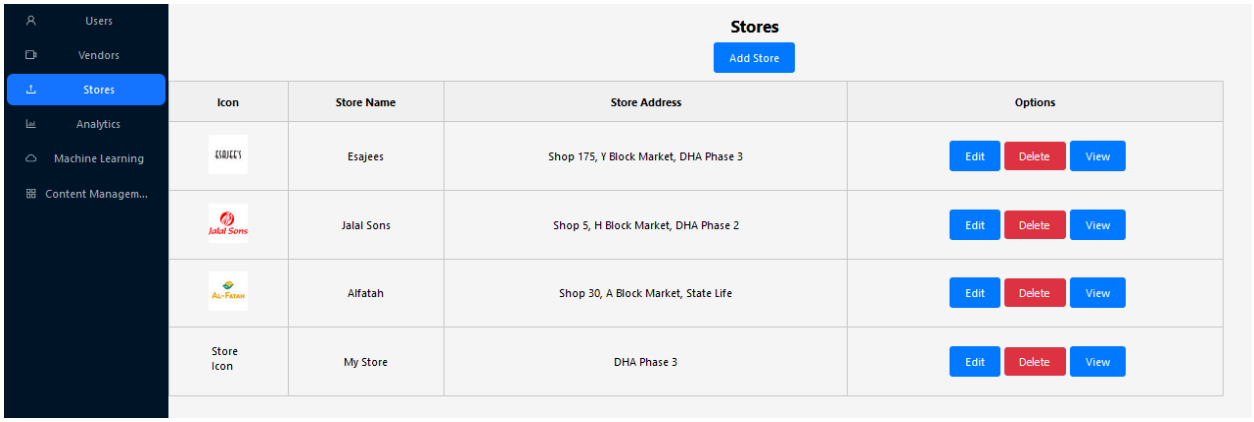


Figure 2.6 Grocery Store Listing

On the website, the grocery store listing was displayed as shown in Figure 2.6 above.

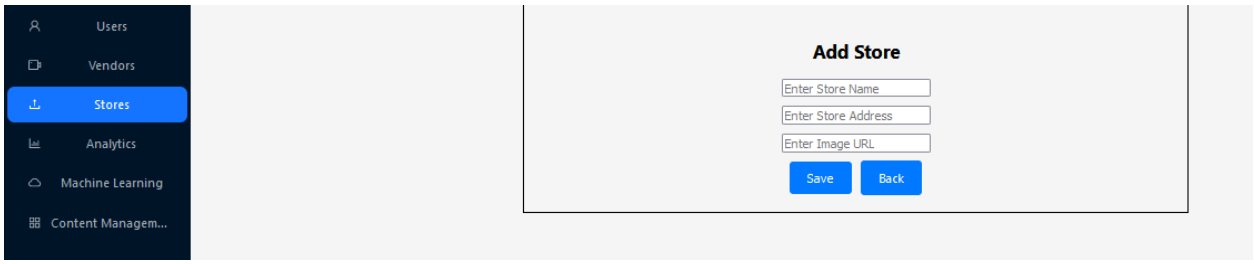


Figure 2.7 Grocery Store Addition

On the website, the grocery store addition was displayed as shown in Figure 2.7 above.

Phase 2: App Development Progress

In this phase, buttons of various sizes were incorporated, tables were created, and extensive experimentation with the overall look of the admin portal was conducted. These efforts aimed to enhance the visual appeal and functionality of the portal, ensuring it met the desired standards.

```
{admins.map(admin => (
    <tr key={admin.admin_id}>
        <td>{admin.email}</td>
        <td>{admin.admin_role}</td>
        <td align="center">
            <Link to={`/`} className="btn btn-primary" style={{ marginLeft: '5px' }}>Edit</Link>  
            <Link to={`/`} className="btn btn-danger" style={{ marginLeft: '5px' }}>Delete</Link>  
            <Link to={`/`} className="btn btn-primary" style={{ marginLeft: '5px' }}>View</Link>  
        </td>
    </tr>
))}
```

Figure 2.8 Add,Delete,View

We implemented functionality for adding, viewing, and deleting data by incorporating corresponding buttons into the admin portal as seen in Figure 2.8.

```
const AddCategory = ({ addCategory }) => {  Show usages   unknown
    const [categoryName : string , setCategoryName] = useState( initialState: '');
    const location : Location  = useLocation();

    // Function to check if the current path is '/contentManagement'
    const isActive = () : boolean  => location.pathname === '/contentManagement';

    const handleSave = () : void  => {  Show usages   unknown
        if (categoryName.trim() !== '') {
            addCategory(categoryName);
            setCategoryName( value: '');
        } else {
            alert('Please enter a category name.');
        }
    };
```

Figure 2.9 Create Category Function

We implemented the functionality that enhances the portal's capability to manage and organise categories effectively, providing a seamless experience for administrators adding new categories to the system as seen in Figure 2.9.

```
3 usages   malaikasultant *
const AddProduct = ({ addProduct }) => {
    const [productName : string , setProductName] = useState( initialState: '');
    const [image, setImage] = useState( initialState: null);
    const location : Location  = useLocation();

    1 usage   malaikasultant
    const isActive = () : boolean  => location.pathname === '/listProducts';

    1 usage   malaikasultant *
    const handleSave = () : void  => {
        if (productName.trim() !== '' && image !== null) {
            console.log("Product Name:", productName);
            console.log("Image File:", image);
            setProductName( value: '');
            setImage( value: null);
        } else {
            alert('Please enter both product name and upload an image.');
        }
    };

    1 usage   malaikasultant
    const handleImageChange = (e) : void  => {
```

Figure 2.10 Create Product Function

This functionality enhances the portal's capability to manage and organise products effectively, providing a seamless experience for administrators adding new products to the system, as illustrated in Figures 4.5 and 2.10.

```
3 usages   malaikasultant
const AddVendorComponent = () => {
    const [vendorName : string , setVendorName] = useState( initialState: '');
    const [deliveryLocation : string , setDeliveryLocation] = useState( initialState: '');
    const [image : string , setImage] = useState( initialState: '');
    const location : Location  = useLocation();

    no usages   malaikasultant
    const isActive = () : boolean  => location.pathname === '/content-management';

    1 usage   malaikasultant
    const handleSave = () : void  => {
        if (vendorName.trim() !== '' && deliveryLocation.trim() !== '' && image.trim() !== '') {
            VendorService.addVendor( info: { name: vendorName, deliveryLocation: deliveryLocation, image: image });
            setVendorName( value: '');
            setDeliveryLocation( value: '');
            setImage( value: null);
        } else {
            alert('Please enter both vendor name, delivery location, and upload an image.');
        }
    };
```

Figure 2.11 Vendor Addition Code Snippet

To facilitate the addition of new vendors to our system, we have implemented a user-friendly interface where administrators can input vendor details such as image URLs, name, and delivery location information. Upon submission, the details are stored in our database for future reference. Above is Figure 2.11 showing the vendor addition form.

```
3 usages   malaikasultant *
const AddStoreComponent = () => {
    const [storeName : string , setStoreName] = useState( initialState: '');
    const [storeAddress : string , setStoreAddress] = useState( initialState: '');
    const [image : string , setImage] = useState( initialState: '');
    const locationPath : string  = useLocation().pathname;
    no usages   malaikasultant
    const isActive = () : boolean  => locationPath === '/content-management';
    1 usage   malaikasultant
    const handleSave = async () : Promise<void>  => {
        if (storeName.trim() !== '' && storeAddress.trim() !== '' && image.trim() !== '') {
            try {
                await StoreService.addStore( info: { store_name: storeName, store_address: storeAddress, image: image
                setStoreName( value: '');
                setStoreAddress( value: '');
                setImage( value: '');
                console.log('Store added successfully');
            } catch (error) {
                console.error('Error adding store:', error);
            }
        } else {
```

Figure 2.12 Grocery Store Addition Code Snippet

Finally, administrators can add new grocery stores to the system using a simple and intuitive form. The form collects details such as store name, address, and an image URL representing the store. Upon submission, the newly added store is stored in our database for future reference. Figure 2.12 is the code snippet for the grocery store addition form.

Phase 3: Front-End and Back-End Integration

During this phase of admin portal development, efforts were focused on seamlessly integrating the front-end and back-end, ensuring all functionalities met the specified requirements. The initial visual demo of the portal was successfully achieved by implementing REST APIs and utilizing HTTP protocols for JSON data exchange. This integration was essential in bringing the portal to life and effectively demonstrating its core features.

```
const ListContentManagementComponent = ({}) => {  Show usages  unknown
    const [categories : any[] , setCategories] = React.useState( initialState: []); // State for content management data
    const location : Location  = useLocation(); // Get current location

    React.useEffect( effect: () : void  => {
        const refreshCategories = async () : Promise<void>  => {  Show usages  unknown
            console.log("Get All Product Categories");
            try {
                const response : AxiosResponse<any>  = await ContentManagementService.getAllProductCategories();
                setCategories(response.data);
            } catch (error) {
                console.error("Error fetching product categories:", error);
            }
        };

        refreshCategories();
```

Figure 2.13 Fetching Categories

We successfully integrated functionality to fetch categories directly from the backend, ensuring that the list category page displays accurate and up-to-date information as shown in Figure 2.13.

```
3 usages  malaikasultant +1
const ListContentManagementComponent = () => {
    const [categories : any[] , setCategories] = useState( initialState: []);
    const [products : any[] , setProducts] = useState( initialState: []);
    const [showProducts : boolean , setShowProducts] = useState( initialState: false);
    const location : Location  = useLocation();

    useEffect( effect: () : void  => {
        1 usage  malaikasultant +1
        const fetchData = async () : Promise<void>  => {
            try {
                const categoriesResponse = await ContentManagementService.getAllProductCategories();
                setCategories(categoriesResponse.data);

                const productsResponse = await ContentManagementService.getAllProducts();
                setProducts(productsResponse.data);
            } catch (error) {
                console.error("Error fetching data:", error);
            }
        };

        fetchData();
    }, deps: []);
```

Figure 2.14 Fetching Products

The successful integration functionality to fetch product data directly from the backend, ensuring that the product list page displays accurate and up-to-date information, is as shown in Figures 2.14.

```
3 usages   malaikasultant *
const ListVendorComponent = () => {
    const [vendors : any[] , setVendors] = useState( initialState: []);
    const location : Location  = useLocation(); // Get current location

    no usages   new *
    const isVendorActive = () : boolean  => location.pathname === '/vendors';

    useEffect( effect: () : void  => {
        1 usage   malaikasultant
        const refreshVendors = async () : Promise<void>  => {
            console.log("Get All Vendors");
            try {
                const response = await VendorService.getAllVendors();
                setVendors(response.data);
            } catch (error) {
                console.error("Error fetching vendors:", error);
            }
        };

        refreshVendors();
    }, deps: []);
```

Figure 2.15 Vendor Listing Code Snippet

To streamline vendor management, we have incorporated a feature to list all vendors along with their essential details. This includes their image, name, and delivery location information. Figure 2.15 provided above illustrates how the vendor list is rendered.

```
3 usages   malaikasultant
const ListStoreComponent = () => {
    const [stores : any[] , setStores] = useState( initialState: []);
    const [showStores : boolean , setShowStores] = useState( initialState: true); // Initially show stores
    const location : Location  = useLocation();

    useEffect( effect: () : void  => {
        1 usage   malaikasultant
        const fetchData = async () : Promise<void>  => {
            try {
                const response = await StoreService.getAllStores();
                setStores(response.data);
            } catch (error) {
                console.error("Error fetching data:", error);
            }
        };

        fetchData();
    }, deps: []);
```

Figure 2.16 Grocery Listing Snippet

Similar to vendors, grocery stores are vital entities in our system. We have provided a feature to list all grocery stores along with pertinent details such as name, addresses, and an image representing the store. This enables administrators to have a comprehensive view of all available grocery stores. Above is Figure 2.16 showcasing how the grocery store list is rendered.

Phase 4: Final Front-End Design

In the final phase of the admin portal, we refined the navbar to match the prototype and ensured that every update, addition, and deletion functioned flawlessly.

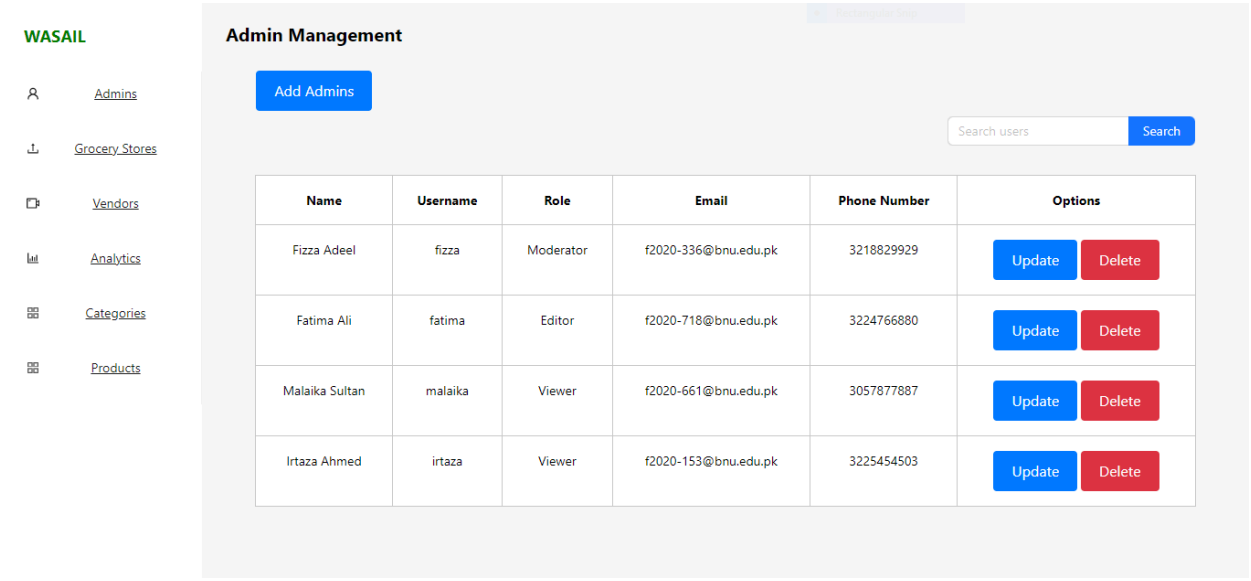


Figure 2.17 Admin Management

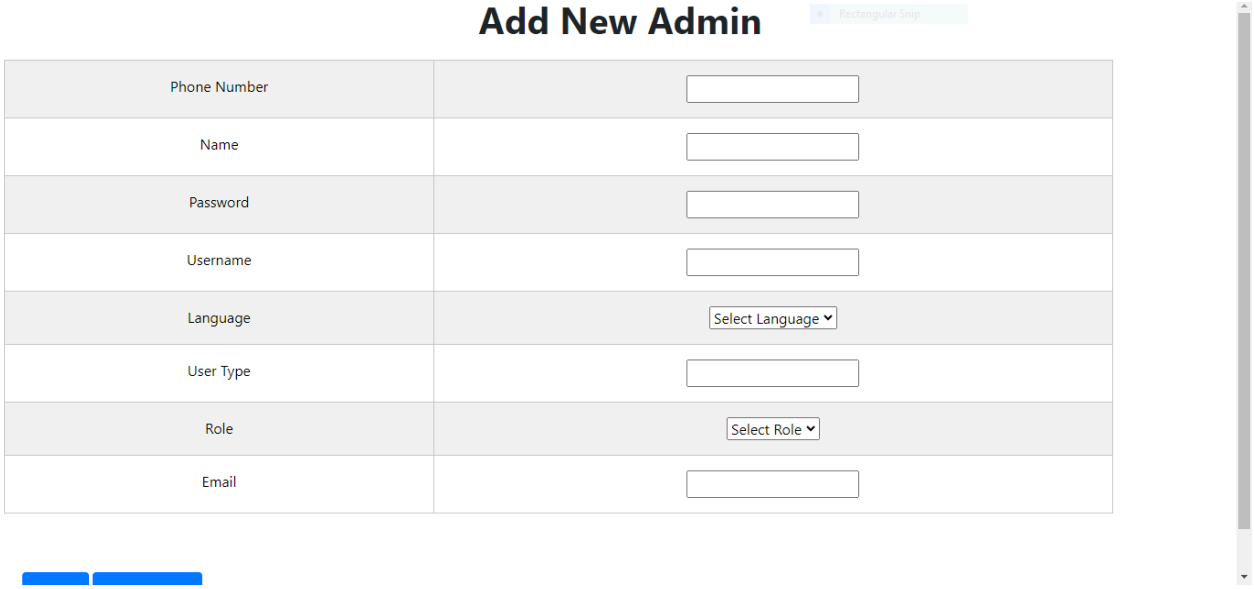


Figure 2.18 Add New Admin

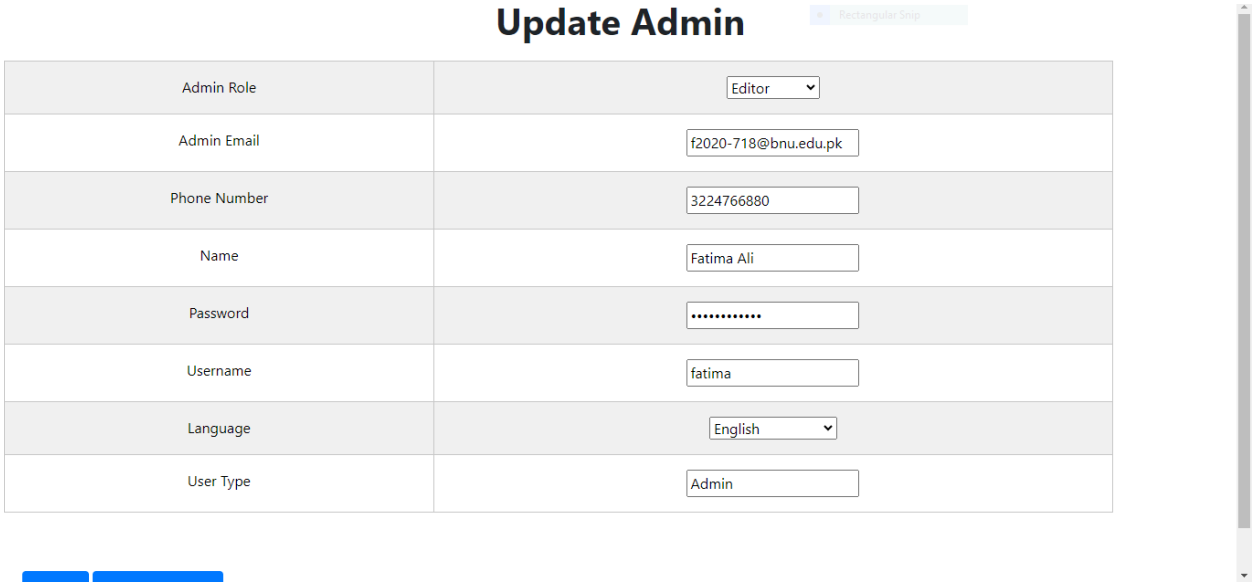


Figure 2.19 Update Admin

In the admin page, Users who are members of the Wasail team, are listed in a tabular form as shown in figure 2.17. This organised layout allows for easy access and management of user information. From this page, we have the capability to add new admins to the system, providing necessary access and permissions as required as shown in Figure 2.18. Similarly, we can also delete users or update their information as needed as shown in Figure 2.19, ensuring that user management tasks can be efficiently handled within the admin portal. This functionality enhances administrative control and facilitates smooth operations within the Wasail team.

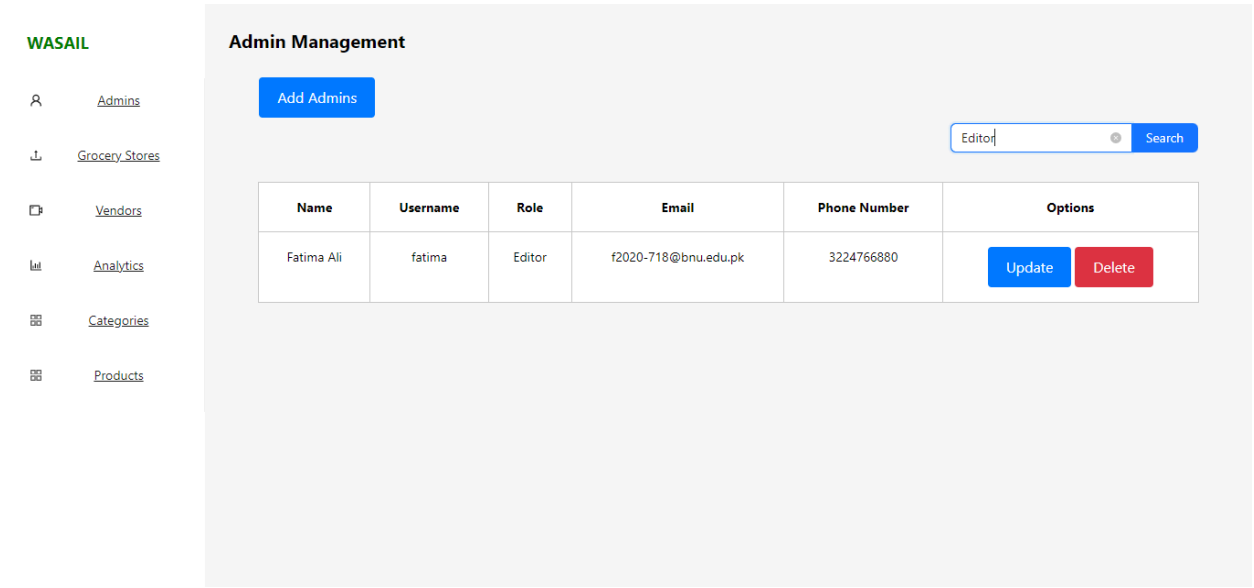


Figure 2.20 Search Admin

We have implemented a search feature specifically designed for admins. This search functionality allows us to enter a role or email and retrieve users matching the search criteria. As a result, we can quickly locate specific admins within the system based on their roles or email addresses as shown in figure 2.20. This capability streamlines user management tasks, making it easier to find and manage admins within the Wasail team.

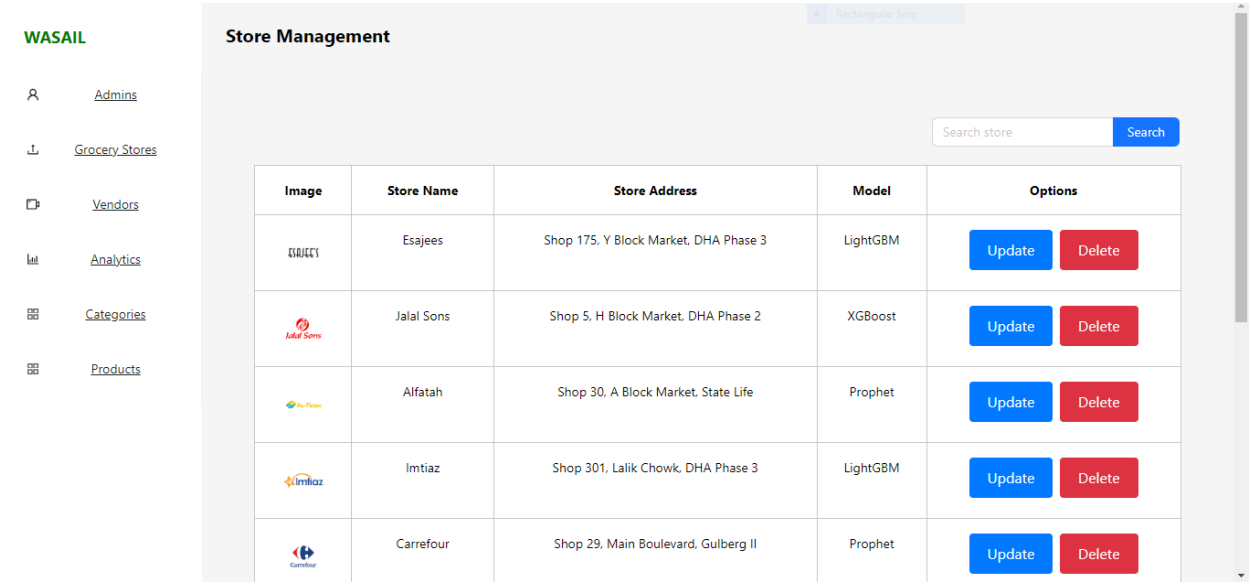


Figure 2.21 Store Management

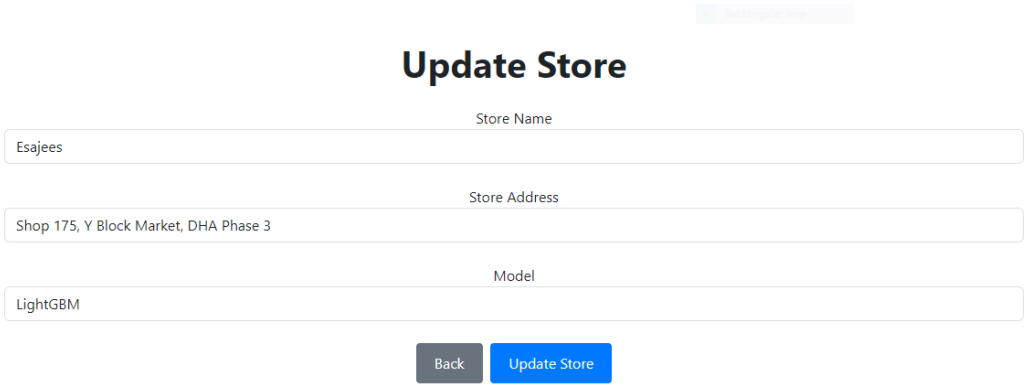


Figure 2.22 Update Store

In store management, all the stores stored in the database are presented in a structured tabular format as shown in Figure 2.21.This layout provides a clear overview of each store's information, such as store name, address, and other relevant details. From this interface, we have the capability to update store information or remove a store from the database, ensuring accurate and up-to-date records as shown in Figure 2.22.

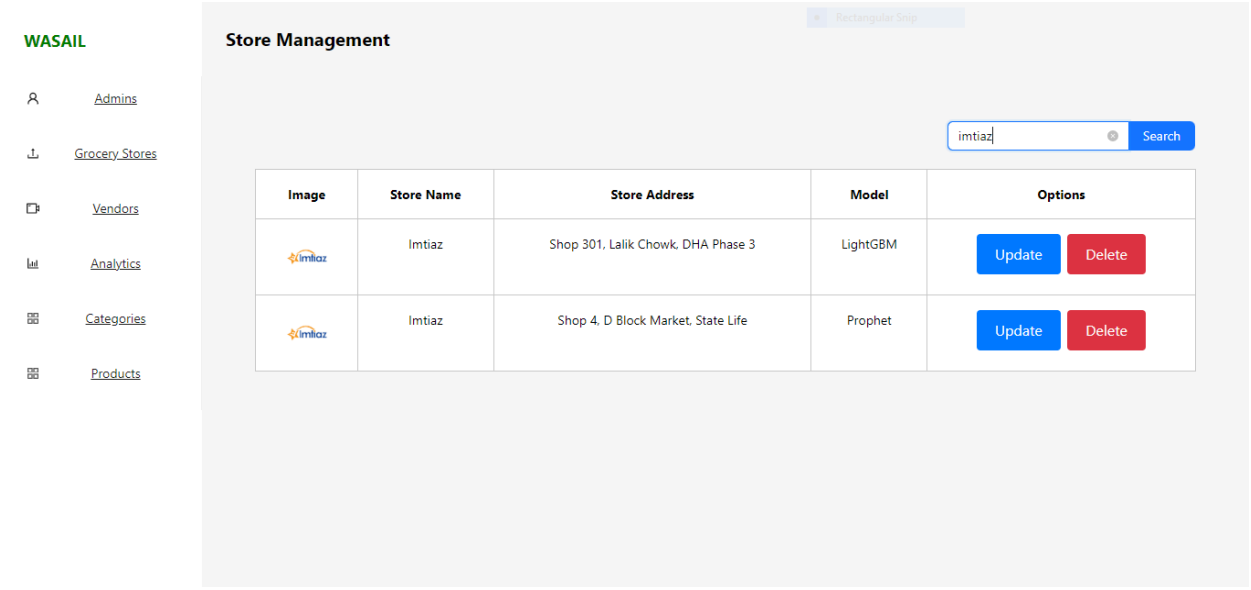


Figure 2.23 Update Store

Additionally, a search functionality has been implemented to facilitate quick access to specific stores. Users can enter a store name or address in the search bar, and the system will return the corresponding row or rows containing the matching information. This search feature enhances efficiency in store management tasks, enabling users to find and handle stores effectively within the system as shown in figure 2.23.

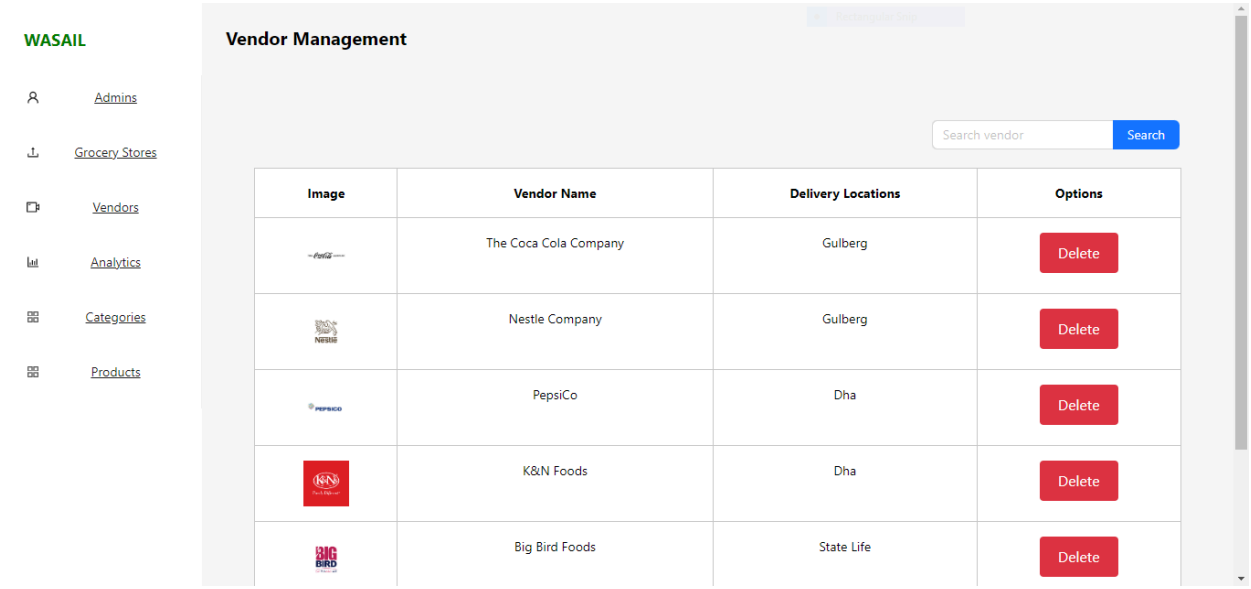


Figure 2.24 Vendor Management

In vendor management, all the vendors stored in the database are displayed in a tabular format for easy reference. Users have the capability to delete vendors from the database directly through this interface as shown in Figure 2.24.

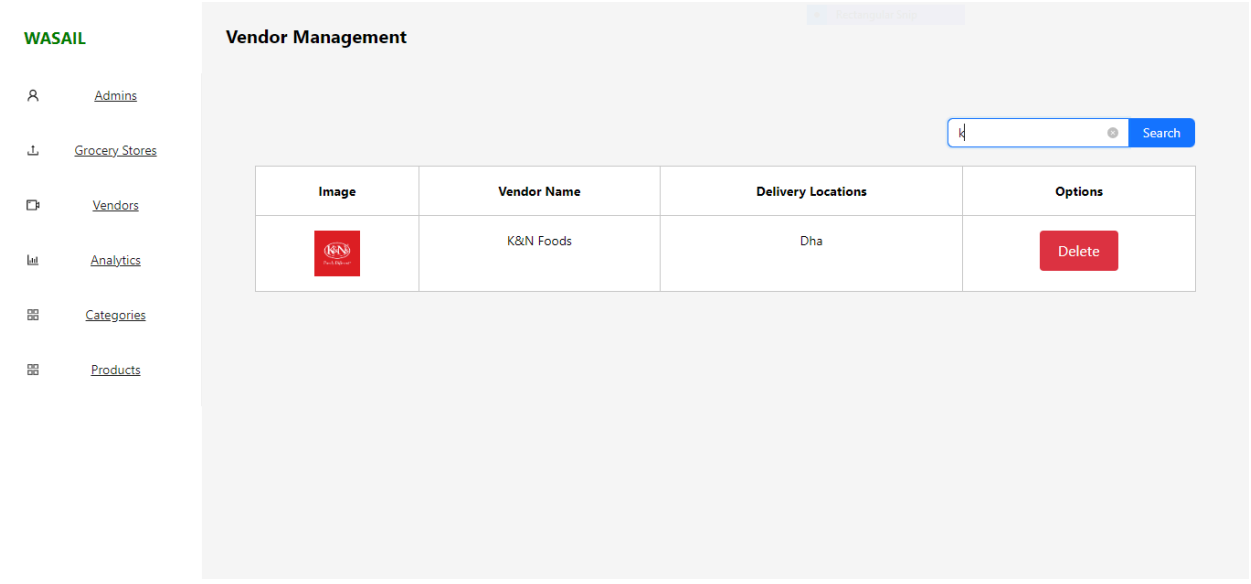


Figure 2.25 Search Vendor

Additionally, a search bar is provided specifically for searching vendors by their name. Users can enter a vendor's name in the search bar, and the system will return the corresponding row or rows containing the matching vendor information. This search functionality simplifies the process of locating specific vendors within the system, facilitating efficient vendor management as shown in Figure 2.25.

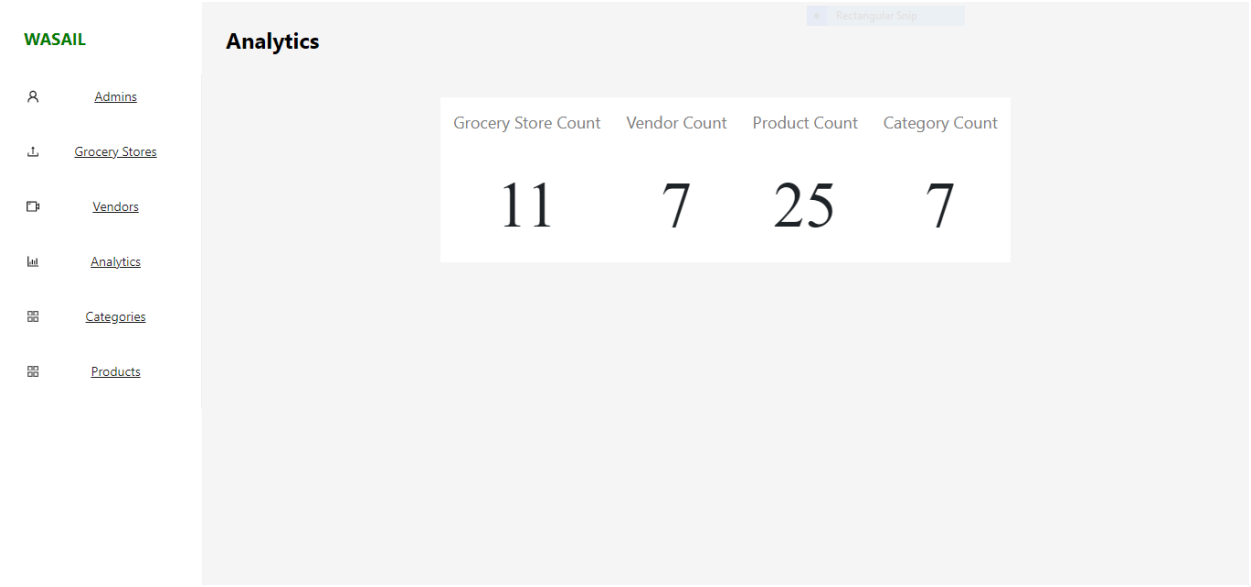


Figure 2.26 Analytics

In Analytics, we provide an overview of key metrics related to our platform. This includes the count of grocery stores, vendors, products, and categories. By displaying these counts, we offer valuable insights into the scale and scope of our platform, enabling stakeholders to assess performance and make data-driven decisions. This analytics feature enhances transparency and facilitates strategic planning, contributing to the overall effectiveness of our platform as shown in the above Figure 2.26.

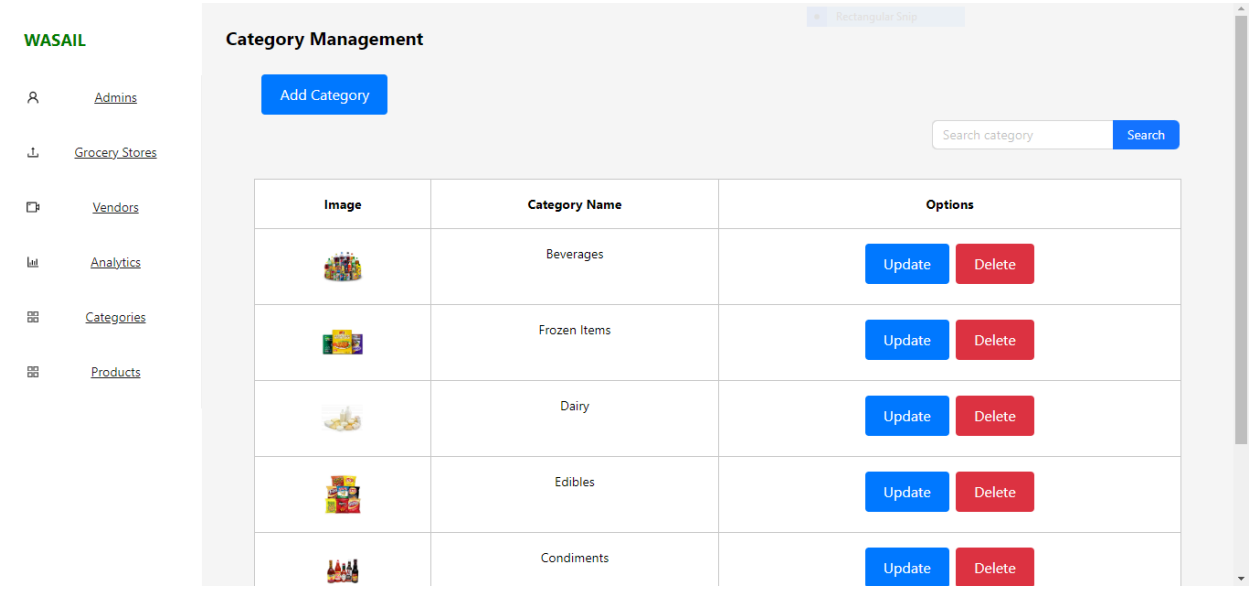


Figure 2.27 Category Management

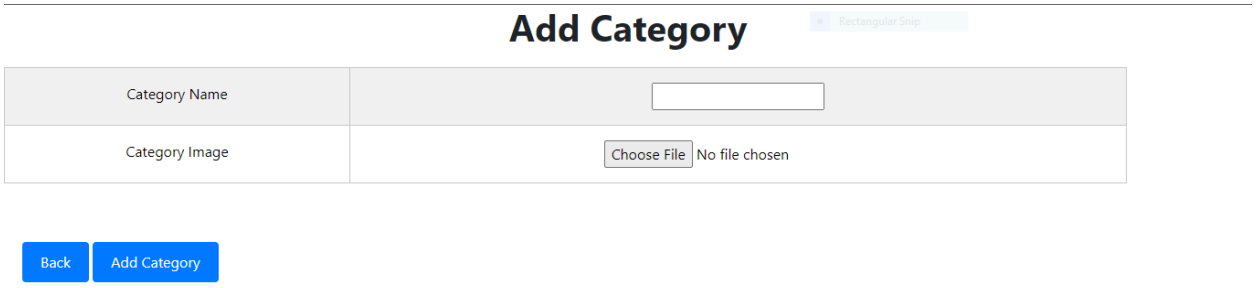


Figure 2.28 Add Category

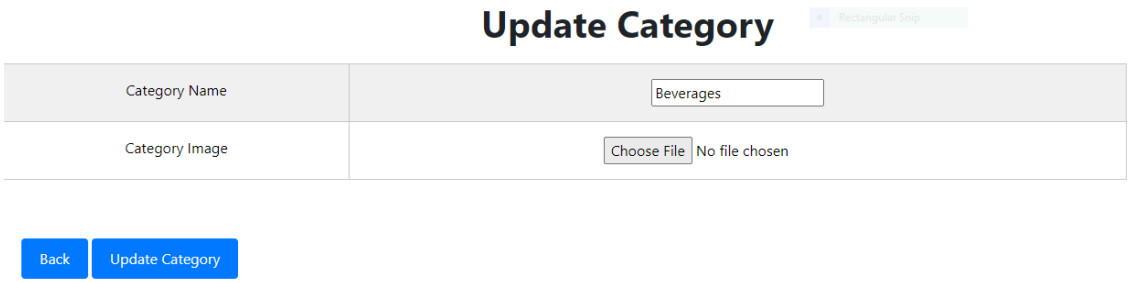


Figure 2.29 Update Category

In category management, we have comprehensive functionality for managing categories efficiently as shown in Figure 2.27. Users can add new categories to the system (Figure 2.28), update existing categories (Figure 2.29) or delete categories as needed.

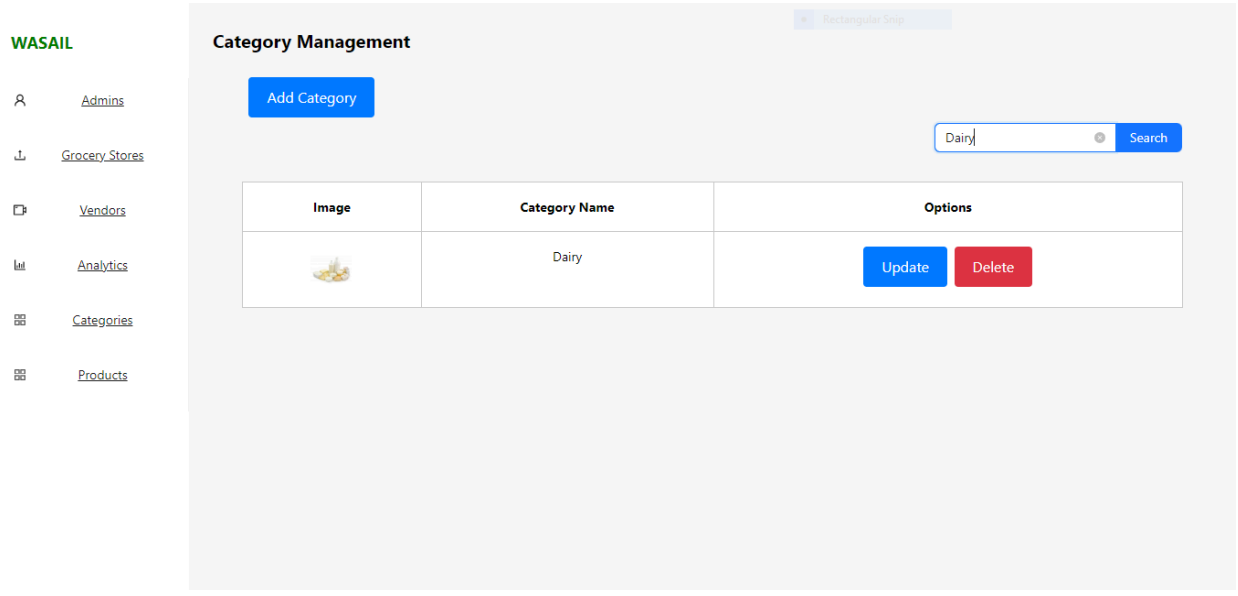


Figure 2.30 Search Category

Additionally, a search feature is implemented specifically for categories. Users can enter a category name in the search bar, and the system will return the corresponding row containing the matching category information. This search functionality streamlines category management tasks, making it easy to find and handle categories within the system as shown in Figure 2.30.

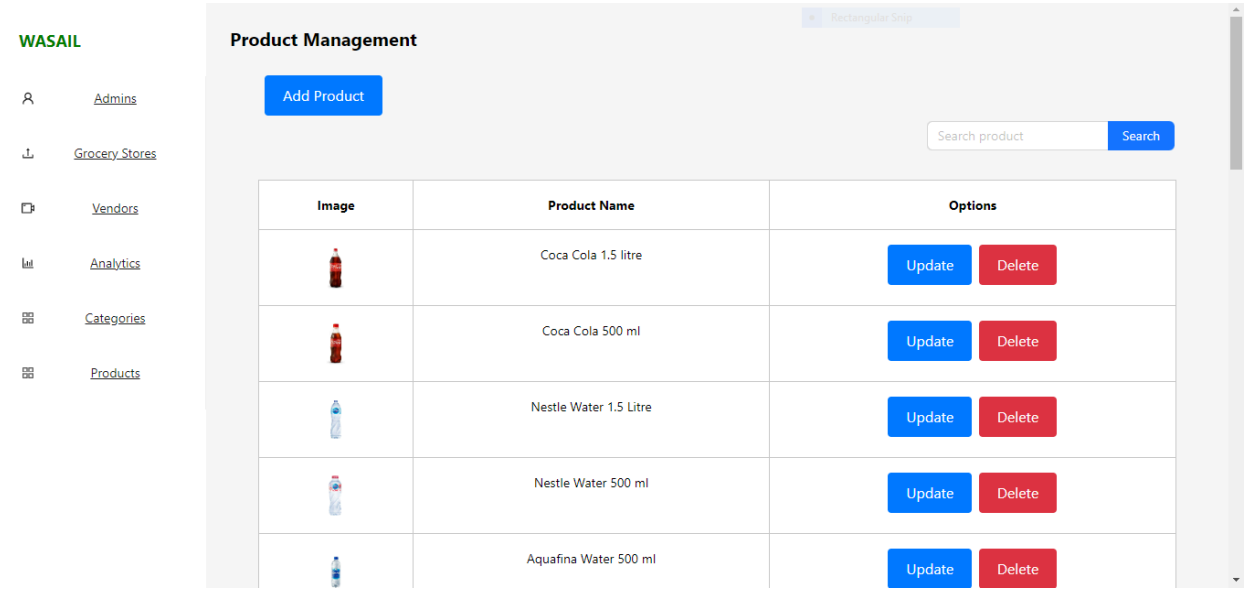


Figure 2.31 Product Management

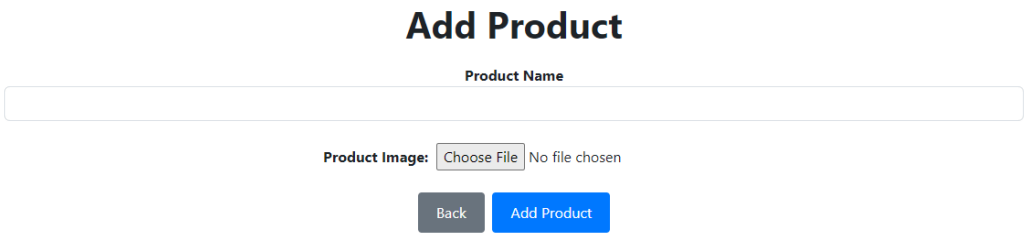


Figure 2.32 Add Product

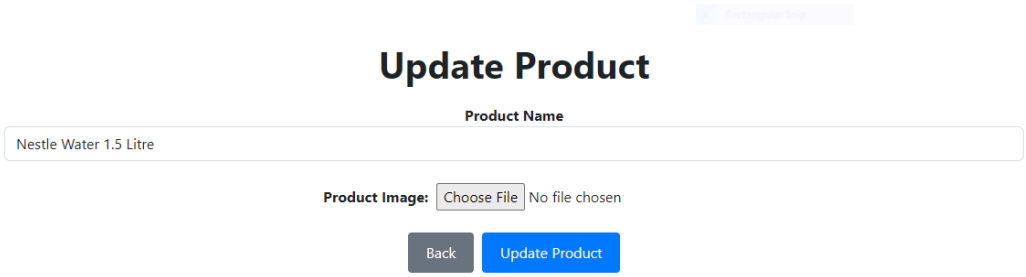


Figure 2.33 Update Product

In product management, users have extensive capabilities for managing products within the system as shown in Figure 2.31. This includes the ability to add new products (Figure 2.32), update existing product information (Figure 2.33), and delete products if necessary.

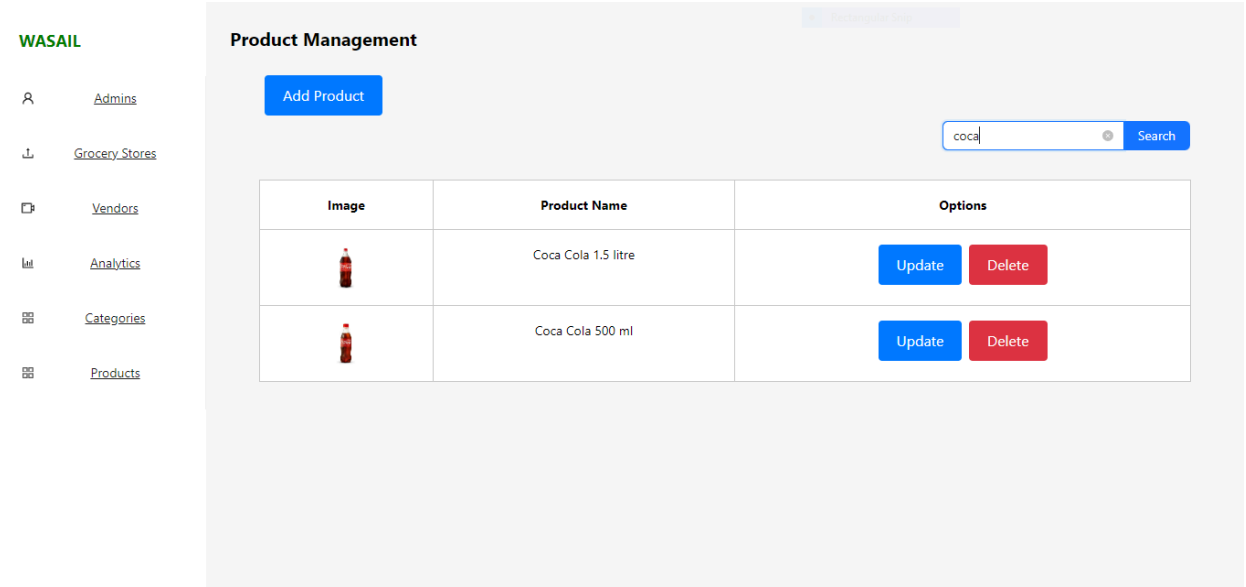


Figure 2.34 Search Product

Similarly, a search feature is integrated into the product management interface. Users can enter a product name or any relevant information in the search bar, and the system will return the

corresponding row or rows containing the matching product information. This search functionality simplifies product management tasks, allowing users to quickly locate and handle products within the system as shown in Figure 2.34.

Back End

In this section all the phases of backend from database design all the way to the completion of the entire backend and its testing on postman has been described.

Database Design

We first started with creating the database model on Lucidchart. Lucidchart was chosen because we had worked on it before to create the database models for previous courses that we have studied and it was also recommended to us by our external advisor. The database model was created in a way that it was divided into three main parts which included User management (user table, grocery store, vendor and admin), Inventory management (product, product inventory, and product category) and lastly Order management (order and order details).There three main types of relations between the tables that were designed. These included One to One (for example between user table and vendor), One to Many (for example between order and order details) and Many to Many (for example between vendor and grocery stores). Since Many to Many relation can not be directly implemented in the database, it is being done through a pivot table so for instance the pivot table between grocery stores and vendors is called lists. The model went through multiple iterations. For example figure 1.0 shows the first iteration of the model which included a separate table for language, location and order status but then we were advised to convert these tables into attributes.

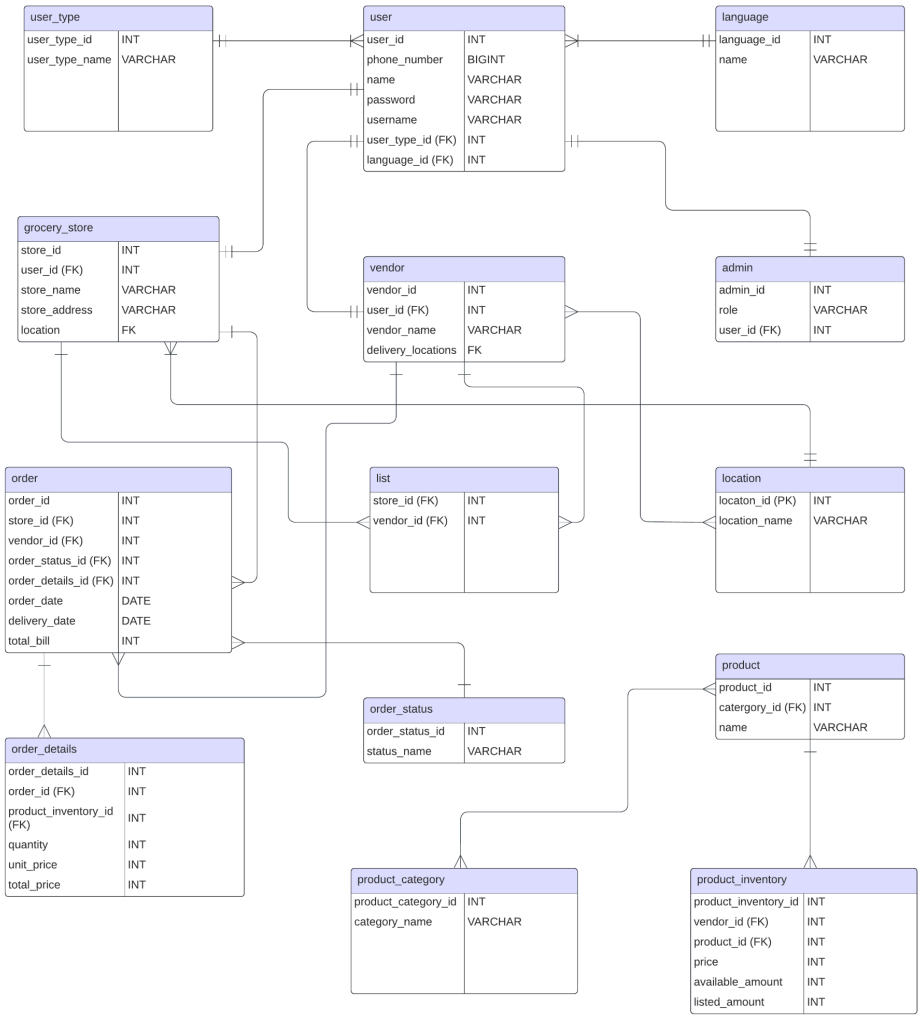


Figure 1.0 First iteration of the database model

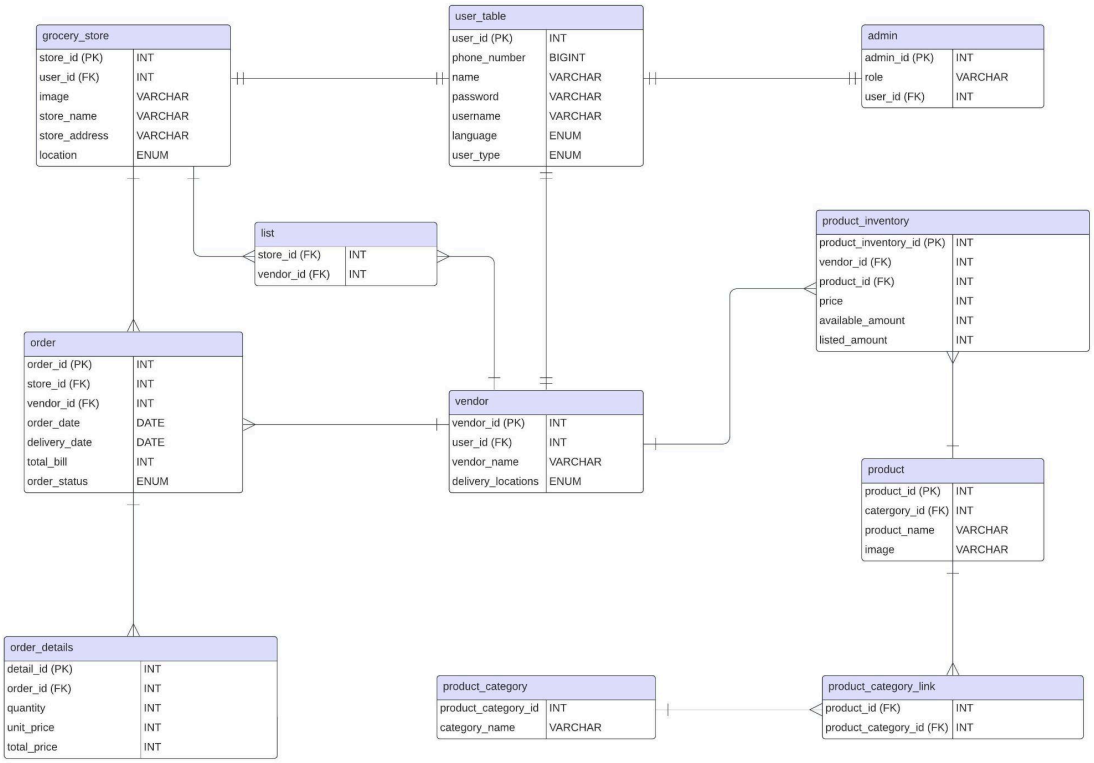


Figure 1.1 Last iteration of database model

Spring Boot

Then since we had mentioned both Spring Boot and Node JS in our initial proposal document, we were advised to first experiment with both and then make our final decision on which framework to use in the end. As we had worked with Spring Boot before in our previous courses, we decided to experiment with it first. All the CRUD operations of a table called Grocery Store were implemented using IntelliJ and its database was made using SQL on XAMPP. A simple front end was also developed so that we could run all the CRUD operations smoothly. Figure 1.2 shows the front end of the implementation with all the CRUD operations working. We have uploaded the Spring Boot project on GitHub.

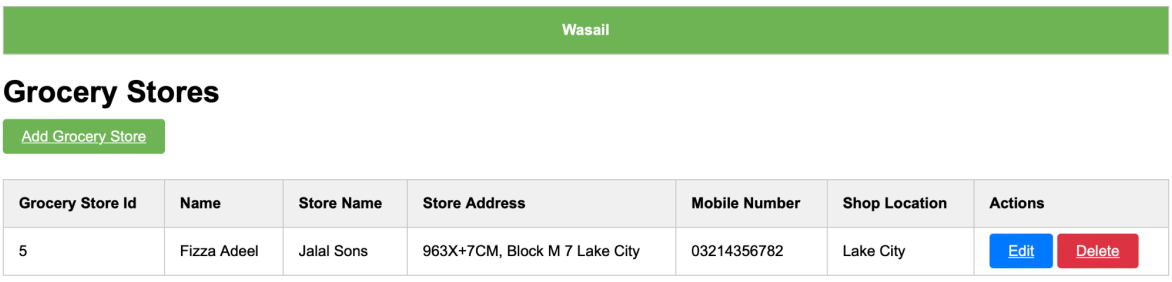


Figure 1.2 Implementation using Spring Boot

Installation of Sequelize

However, we decided to work with Node JS instead. The first step was to download Node JS, Express JS and WebStorm. After downloading them, the following tutorials were followed to set up the project, download different libraries including sequelize, using sequelize and implementing the first table:

- Tutorial Series (Tutorial Series on Youtube)
- Sequelize (Documentation for Sequelize)

We first started with using sequelize in Node JS which is basically an object relational mapper or an ORM which helps with handling databases by representing data as objects. It can be used to create models as well as perform the CRUD operations and create relationships between the models easily. So we first started with installing sequelize. Since sequelize does not have any in-built database drivers so we had to install that separately as well. As we are using MySql for database the driver installed for it was mysql2. Figure 1.3 shows the installation of sequelize. Figure 1.4 shows the installation of MySQL2.

```
PS C:\Users\fizza\Documents\Final Year Project\prj-403\BackEnd\Node.js\Wasail> npm install --save sequelize
```

Figure 1.3 Installation of Sequelize

The aforementioned tutorial installed version 5.2.7 of sequelize but the one that we installed was sequelize version 6.35.2.

```
PS C:\Users\fizza\Documents\Final Year Project\prj-403\BackEnd\Node.js\Wasail> npm install --save mysql2
```

Figure 1.4 Installation of MySQL2

The aforementioned tutorial installed version 1.6.5 of MySQL2 but the one that we installed was MySQL2 version 3.6.5.

```
PS C:\Users\fizza\Documents\Final Year Project\prj-403\BackEnd\Node.js\Wasail> npm install -g sequelize-cli
```

Figure 1.5 Installation of sequelize-cli

Figure 1.5 shows the installation of sequelize cli which is another library needed to make the models work with sequelize.

```
  "dependencies": {
    "body-parser": "^1.20.2",
    "chai": "^4.3.10",
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "dotenv": "^16.3.1",
    "express": "^4.18.2",
    "http-errors": "~1.6.3",
    "morgan": "~1.9.1",
    "mysql": "^2.18.1",
    "mysql2": "^3.6.5",
    "pug": "2.0.0-beta11",
    "sequelize": "^6.35.2",
    "validate.js": "^0.13.1"
  },
  "devDependencies": {
    "nodemon": "^3.0.1",
    "sequelize-cli": "^6.6.2"
  }
}
```

Figure 1.6 Dependencies installed for the project

Figure 1.6 shows all the dependencies that were added in the package.json file in the project.

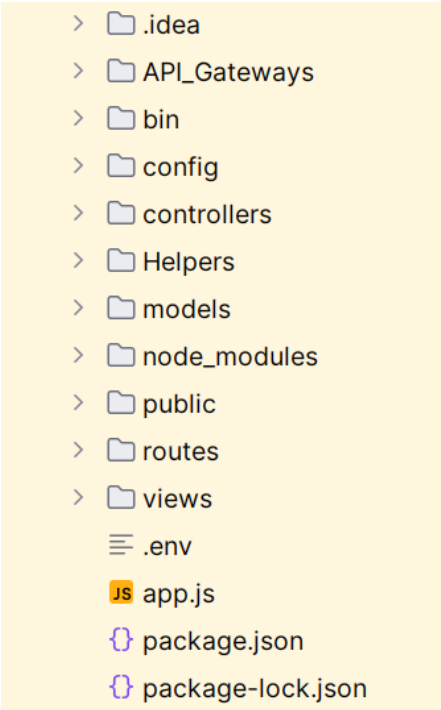


Figure 1.7 Empty files created by sequelize

Fig 1.7 shows all the empty folders created after installing sequelize. The ones that were created included mainly the models folder and the config folder.

User Model Implementation

```
'use strict';

module.exports = (sequelize, Datatypes)=>{
    return sequelize.define('user_table',{
        user_id:{
            type: Datatypes.INTEGER,
            primaryKey: true,
            autoIncrement: true
        },
        phone_number:{
            type: Datatypes.BIGINT,
            isNumeric: true,
            isInt: true,
            notNull: true,
        },
        name:{
            type: Datatypes.STRING,
            isAlpha: true,
            notNull: true,
        },

        password:{
            type: Datatypes.STRING,
            isAlphanumeric: true,
            notNull: true,
            len: [8,12],
        },
        username:{
            type: Datatypes.STRING,
            isAlphanumeric: true,
            notNull: true,
        },
        language:{
            type: Datatypes.ENUM('English', 'Urdu', 'Roman Urdu')
        },
        user_type:{
            type: Datatypes.ENUM('Admin', 'Grocery Store', 'Vendor')
        },
    },{
        underscored:true
    })
}
```

Figure 1.8 User table model created

Next, the first model was created using sequelize in the models folder. Since the user table was the first table created in the database diagram, the first model that was created was the user table as well. The model which had userId, phone number, name password, username, language and user type. The aforementioned documentation was referred to when defining the variables. Since we had to ensure that the variables language and user type do not have values other than the ones seen in figure 1.8, the data type used for them was ENUM which helped us define the fixed values for these specific variables. Moreover, some of the validation checks implemented on the variables for example isAlpha,

isAlphanumeric etc were also included for variables like username and password which were also implemented with the help of the documentation. Additionally, sequelize creates two variables 'Created At' and 'Updated At' on its own for all the tables. Since these are auto generated, these two variables have not been explicitly defined in the model class but have been given values in the database.

Implementation of ENV

Then MySQL Workbench along with MySQL server were installed for the database. My SQL workbench was set up by first setting up a username and a password for it. As we had worked with XAMPP and phpmyadmin before, setting up the workbench took some time but we were able to do it successfully in a couple of tries.

```
USERNAME=
PASSWORD=
DATABASE=
HOST=
NODE_ENV=
```

Figure 1.9 .env file

```
PS C:\Users\fizza\Documents\Final Year Project\prj-403\BackEnd\Node.js\Wasail> npm install dotenv
```

Figure 2.0 Installation of .env file

As seen in figure 1.9 a dot env file was created so that all the sensitive data that included the username, the password, the name of the database and the host can be kept separately outside of the main code. Then using the npmenv the dot env package was installed as seen in figure 2.0.

```
require('dotenv').config();

const username = process.env.USERNAME;
const password = process.env.PASSWORD;
const database = process.env.DATABASE;
// const host = process.env.HOST;
const host = "localhost";
const node_env = process.env.NODE_ENV;

const config = {
    dev : {
        db : {
            username,
            password,
            database,
            host
        }
    },
    test : {},
    prod : {}
}
module.exports = config[node_env];
```

Figure 2.1 Config file

By following the tutorial, we then set up the configuration file.

```
sequelize
    .authenticate()
    .then(() => {
      console.log('Connection has been established successfully.');
    })
    .catch(err => {
      console.error('Unable to connect to the database',err);
    });

module.exports = db;
```

Figure 2.2 Error handling in index file

In order to see the errors that occur, when it comes to connecting with the database, in the console, it is important to write this down otherwise finding the problem that has occurred would have been difficult .

```
const db = require('../models');
 fizzaadeel
db.sequelize.sync()
    .then(() =>{
      server.listen(port);
    })
    .catch(e => console.log(e));
server.on('error', onError);
server.on('listening', onListening);
```

Figure 2.3 Sync function

The sync function was used to help us connect with the database and sync all the models that were made with the database.

Vendor Model Implementation

After all this set up, the project was run to see if the tables are being made in the database. First a database was created with the name of 'wasail' then the project was run and at this point it was successful in creating the user table table in the database on My SQL workbench. After it was time to create relations between multiple tables in the database. In order to implement this, it was important to first implement another model, so we implemented the vendor model next. Its implementation was similar to that of the user table however it now had the user id as foreign key as well in order to create an association with it. Figure 2.4 shows the implementation of the vendor model.

```
'use strict';

module.exports = (sequelize, Datatypes)=>{
    return sequelize.define('vendor',{
        vendor_id:{
            type: Datatypes.INTEGER,
            primaryKey: true,
            autoIncrement: true
        },
        vendor_name:{
            type: Datatypes.STRING,
            isAlpha: true,
            notNull: true,
        },

        delivery_locations:{
            type: Datatypes.ENUM('Dha', 'Gulberg', 'State Life')
        },


        delivery_locations:{
            type: Datatypes.ENUM('Dha', 'Gulberg', 'State Life')
        },

        user_table_user_id: {
            type: Datatypes.INTEGER,
            references: {
                model: 'user_tables',
                key: 'user_id'
            },
            allowNull: false
        },
    },{
        underscored:true
    })
}
```

Figure 2.4 Implementation of vendor model

Next, it was time to implement the CRUD operations. For the implementation of the CRUD, the tutorial series that were being followed up until this point got complicated. So after going through multiple tutorials and not being successful after following them, the following tutorial along with the aforementioned sequelize documentation was followed in order to continue the implementation:

- Sequelize tutorial

Implementation of Controller

Next, the controller folder was set up, which had a separate controller for all the models. So a controller for the user table and for the vendor was implemented. Now the controller contains all the functionality. Figure 2.5 shows the user controller and some of the functionality that has been implemented.

```
const db = require('../models')
const { Op } = require("sequelize");
const User = db.user_table

1 usage   fizzaadeel +1
const addUser = async (req, res) => {

    let info = {
        phone_number: req.body.phone_number,
        name: req.body.name,
        password: req.body.password,
        username: req.body.username,
        language: req.body.language,
        user_type: req.body.user_type
    }

    const user = await User.create(info)
    res.status(201).send(user)
}


    const numberExists = async (req, res) => {
        try {
            let phone_number = req.params.phone_number;

            let user = await User.findOne({
                where: {
                    phone_number: {
                        [Op.eq]: phone_number,
                    },
                },
            });

            if(user == null) {
                res.status(200).json({ exists: false });
            }
            else {
                res.status(200).jsn({ exists: true, userId: user.user_id });
            }
        }
        catch (error) {
            console.error('Error checking phone number existence:', error);
            res.status(500).json({ error: 'Internal Server Error' });
        }
    };

     const usernameExists = async (req, res) => {
         try {
             let username = req.params.username;

             let user = await User.findOne({
                 where: {
                     username: {
                         [Op.eq]: username,
                     },
                 },
             });

             if(user == null) {
                 res.status(200).send(false)
             }
             else
                 res.status(200).send(true)
         }
         catch (error) {
             console.error('Error checking phone number existence:', error);
             res.status(500).json({ error: 'Internal Server Error' });
         }
     };
```

```
module.exports = {
    addUser,
    getAllUsers,
    getOneUser,
    updateUser,
    deleteUser,
    numberExists,
    usernameExists,
    userAuthentication
}
```

Figure 2.5 Implementation of user controller

```
const searchProduct = async (req, res) => {
    try {
        let product_name = req.params.product_name;
        if (!product_name) {
            return res.status(400).json({ error: 'Search term is required.' });
        }
        const product = await Product.findAll({
            where: {
                product_name: {
                    [Op.like]: `%${product_name}%`,
                },
            },
        });
        res.status(200).send(product)
    } catch (error) {
        console.error('Error searching products:', error);
        res.status(500).json({ error: 'Internal Server Error' });
    }
};
```

```
const searchProductInInventory = async (req, res) => {
    try {
        const vendor_id = req.params.vendor_vendor_id;
        let product_name = req.params.product_name;

        if (!vendor_id) {
            return res.status(400).json({ error: 'Vendor ID is required.' });
        }

        const associatedInventories = await db.product_inventory.findAll({
            where: { vendor_vendor_id: vendor_id },
        });

        const productInventoryIds = associatedInventories.map((inventory) => inventory.product_inventory_id);

        const products = await db.product.findAll({
            where: {
                product_id: productInventoryIds,
                product_name: {
                    [Op.like]: `%${product_name}%`,
                },
            },
        });
```

```
        res.status(200).json(products);
    } catch (error) {
        console.error('Error searching products in inventory by vendor ID:', error);
        res.status(500).json({ error: 'Internal Server Error' });
    }
};
```

Figure 2.5.1 Implementation of product controller

Figure 2.5.1 shows a part of the product controller. Here the two important functions search product in inventory and search product have been shown respectively. For example in the search product in inventory function the vendor id and the product name that is searched are being passed. It uses the vendor id and finds its corresponding product inventory id, then uses that product inventory id and finds its corresponding product id and based on the product id and the product name that has been searched fetches all the products and sends them as a response.

Implementation of Relations

After the implementation of the controller for both the user and the vendor, it was then time to implement the associations between the tables. Implementing the relations between the tables took some time. Even though the tutorial along with the documentation were being followed, we were faced with a lot of errors because of which we were unable to establish an association. However, after a couple of iterations, the relations or associations were established successfully. They are established in the index file inside the model folder. First, The models for which the relations need to be established are imported as shown in figure 2.6.

```
db.user_table = require('./usertable');
db.vendor = require('./vendor');
db.grocery_store = require('./grocerystore');
db.order = require('./order');
db.order_detail = require('./orderdetail');
db.product_category = require('./productcategory');
db.product = require('./product');
db.product_inventory = require('./productinventory');
db.list = require('./list');
console.log(config);
```

Figure 2.6 Importing the models in the index file

Then the relations are implemented. Since we had three different types of relations to implement which included one to one, one to many, and many to many as shown in our database model as well in figure 1.1. So we had to implement the three types in three different ways. The three ways have been shown in figure 2.7, 2.8 and 2.9 respectively.

```
db.user_table.hasOne(db.vendor);
db.vendor.belongsTo(db.user_table);
```

Figure 2.7 One to one relation

The relation between the vendor and the user table is a one to one relation as one user can only be one vendor and vice versa. This relation has been shown in figure 1.1.

```
db.product.hasMany(db.product_inventory);
db.product_inventory.belongsTo(db.product);
```

Figure 2.8 One to many relation

The relation between product and product inventory is a one to many relation as shown in figure 1.1.

```
db.grocery_store.belongsToMany(db.vendor, {through: 'lists'})
db.vendor.belongsToMany(db.grocery_store, {through: 'lists'})
```

Figure 2.8 Many to many relation

The relation between grocery store and vendor is a many to many relation. Since many to many relations can not be implemented directly, so instead it is being implemented through a pivot or an associative table called lists.

Implementation of Routes

After implementing the associations, we had to establish the endpoints or the routes, through which the front end will be able to connect with functions in the back end. The routes were again established separately for each controller, in the routes folder. Since the controller is exporting all the functions established in them, In order to establish the routes, first we need to import that specific controller. Then we need to properly define the endpoints so that they can be utilised by the front end. Figure 2.9 shows the routes established for the user table.

```
router.post('/adduser', usertableController.addUser)
router.get('/allusers', usertableController.getAllUsers)
router.get('/:user_id', usertableController.getOneUser)
router.put('/:user_id', usertableController.updateUser)
router.delete('/:user_id', usertableController.deleteUser)
router.get('/numberexists/:phone_number', usertableController.numberExists)
router.get('/usernameexists/:username', usertableController.usernameExists)
```

Figure 2.9 Implementation of user routes

```
router.post('/addproduct',productController.addProduct)
router.get('/allproducts',productController.getAllProducts)
router.get('/:product_id', productController.getOneProduct)
router.put('/:product_id', productController.updateProduct)
router.delete('/:product_id', productController.deleteProduct)
router.get('/searchproduct/:product_name', productController.searchProduct)
router.get('/searchproductininventory/:vendor_vendor_id/:product_name', productController.searchProductInInventory)
```

Figure 2.9.1 Implementation of product routes

Figure 2.9.1 shows all the routes or the end points defined for the products in the product routes file.

Implementation of Routes in App

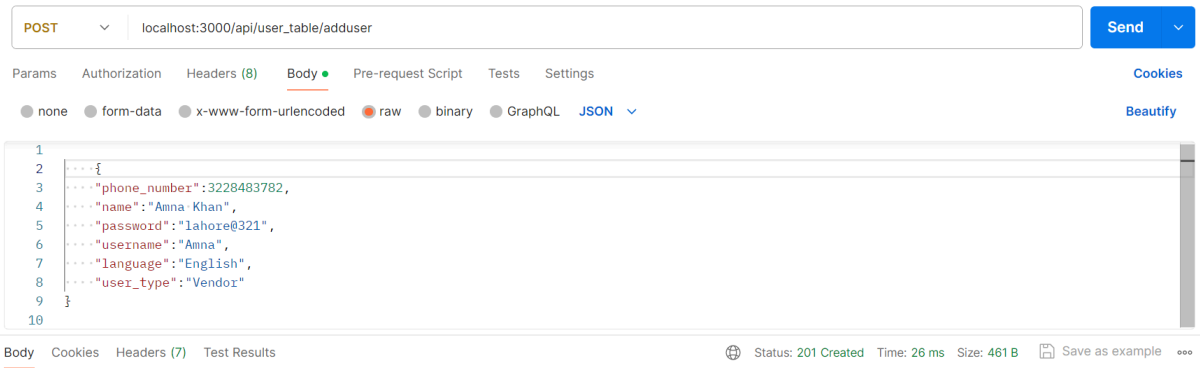
Lastly these routes are then exported to the app.js file. There, first we import the route files for each of them separately. Then these routes are concatenated with the main route and finally the app is exported. Figure 3.0 shows the main routes defined in app.js.

```
app.use('/api/user_table',usersRouter)
app.use('/api/vendor',vendorsRouter)
app.use('/api/grocery_store',storesRouter)
app.use('/api/order',ordersRouter)
app.use('/api/order_detail',detailsRouter)
app.use('/api/product_category',categoriesRouter)
app.use('/api/product',productsRouter)
app.use('/api/product_inventory',inventoriesRouter)
app.use('/api/registration',registrationsRouter)
```

Figure 3.0 Implementation of main routes

Postman Testing

The following screenshots are examples of some of the testing that was done while the implementation of backend was being carried out: Figure 3.1 shows adding a user through postman.



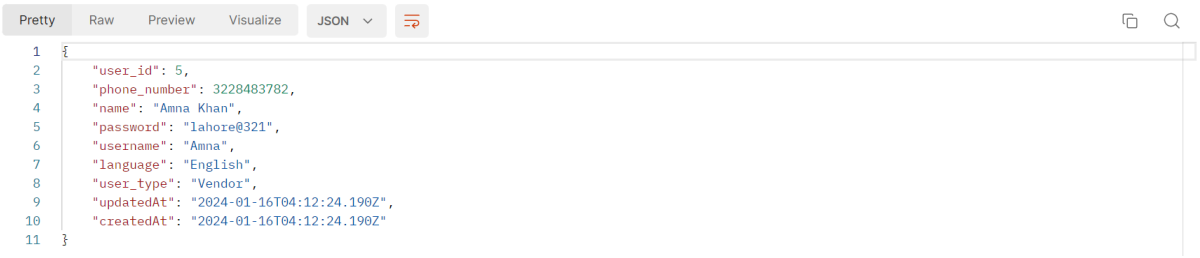


Figure 3.1 Adding a user

Figure 3.2 shows searching for a product. It is going to return all the products which contain the searched word in their name. For example in figure 3.2 the word water has been searched and 2 products containing the word water have been returned from the database.

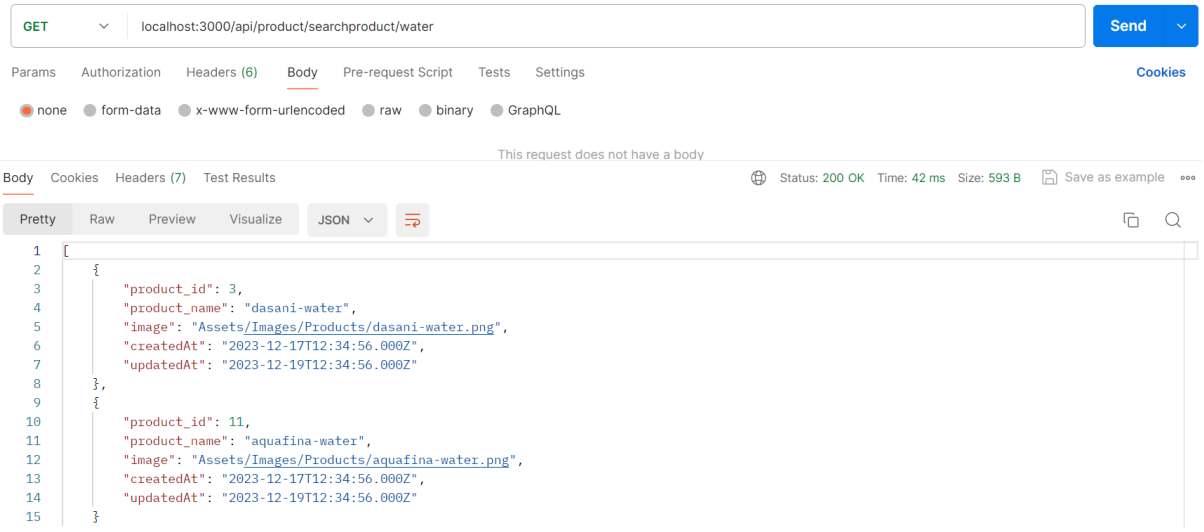


Figure 3.2 Searching a product

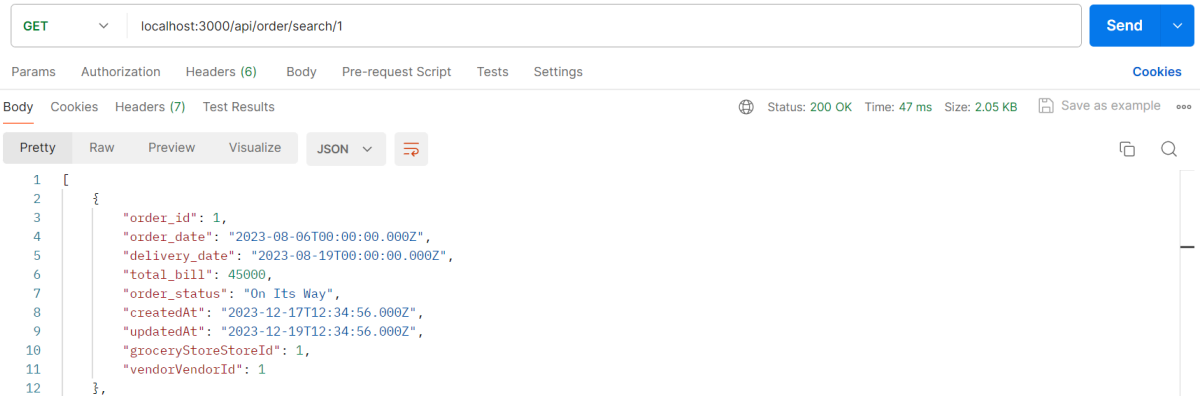
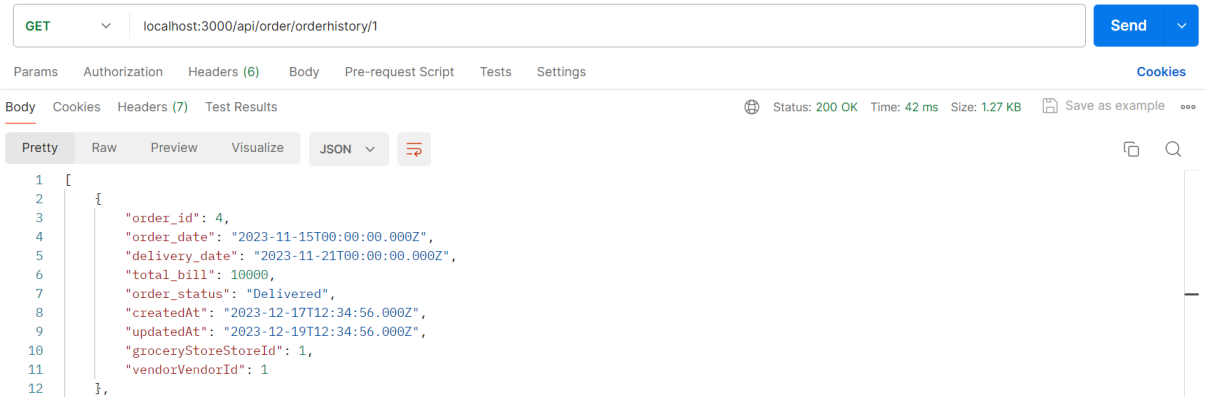


Figure 3.3 Current Orders

Figure 3.3 shows the current orders. We had three different ways to define the statuses of the orders. Those three ways included 'In Process', 'On Its Way', and 'Delivered'. So current orders show all the orders whose status was either In Process or On Its Way based on the vendor id. So the '1' has been passed as the vendor id and all its current orders have been displayed.



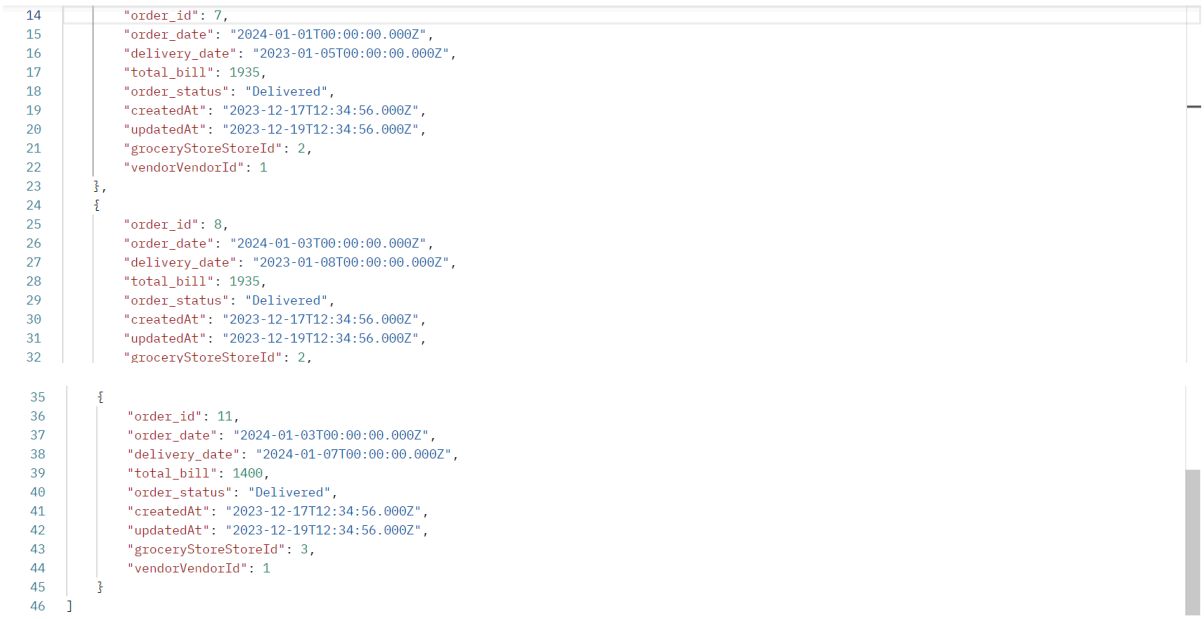


Figure 3.4 Order History

Similarly, figure 3.4 shows order history. So all the orders whose order status is going to be 'Delivered' in the database are going to be displayed for that specific vendor.

Conclusion

In this section, we have walked you through the entire process of the implementation of backend. Screenshots for the entire code of the back end could not be attached as the document would have become unreasonably lengthy. However, the process shown above has been implemented for all the tables for example implementation of the models, implementation of the CRUD operations as well as other functionality which included instances like search product, search inventory, order history, current orders and many more. Moreover, implementation of controllers, implementation of all the endpoints, implementation of relations between all the tables and testing everything through postman has also been done for each and every table in the database.

Test Cases

In this section, we report the test cases of the Vendor App from language selection to user interactions like phone registration, product management, and user profile handling. Through systematic test steps, diverse test data, and expected versus actual result comparisons, we evaluated the application's performance, responsiveness, and adherence to user expectations.

Test Case 1: Language Selection

Test Scenario: The user wants to select their preferred language.
Preconditions: The user has not specified their preferred language.
Postconditions: The user has specified their preferred language.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	Select preferred language	English	Text displayed in English. English is saved as the preferred language.	Text displayed in English. English is saved as the preferred language.	Pass
		Urdu	Text displayed in Urdu. Urdu is saved as the preferred language.	Text displayed in Urdu. Urdu is saved as the preferred language.	Pass
		Roman Urdu	Text displayed in Roman Urdu. Roman Urdu is saved as the preferred language.	Text displayed in English. English is saved as the preferred language.	Fail

Test Case 2: Phone Registration

Test Scenario: The user wants to create an account using their phone number.
Preconditions: The user has opened the app.
Postconditions: The user is directed to the phone number confirmation screen.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user enters their phone number.	3213439305	The system should display a confirmation screen.	The system displays a confirmation screen.	Pass

Test Case 3: Phone Number Confirmation

Test Scenario: The user wants to confirm their entered phone number.
Preconditions: The user has entered their phone number.

Postconditions: The phone confirmation dialogue box should open with options to edit and/or confirm.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user can select the edit option given by the system.	3213439305	The user should be able to edit their phone number, in case it is incorrect.	The user edits their phone number.	Pass
	The user selects the option to confirm to edit their number.	3213439306	The user should be able to confirm their phone number	The user confirms their phone number	Pass

Test Case 4: Phone Number Exists

Test Scenario: The user should be directed to either login or registration screen based on whether the phone number exists or not.
Preconditions: The user has confirmed their phone number.
Postconditions: The user is redirected to either the login or the registration page.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user has confirmed their phone number	3213439305 (exists)	The user should be directed to the login screen.	The login screen is displayed.	Pass
		3213439306 (doesn't exist)	The user should be directed to the registration screen.	The registration screen is displayed.	Pass

Test Case 5: OTP Code Generation and Delivery

Test Scenario: The system should send an OTP code to the user's phone number.
Preconditions: The user has confirmed their phone number.
Postconditions: The user has entered the OTP and is directed to the account details page.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system can confirm the phone	3355600598	The phone number should be confirmed by the system.	The phone number is confirmed by the system.	Pass

	number of the user.				
2	The system generates an OTP code and sends it via SMS		A 4-digit unique OTP code should be generated and sent.	A 4-digit unique OTP code is not generated and sent.	Fail
3	The user waits for 60 seconds to allow resending OTP.		A timer is displayed to the user, indicating the ability to resend OTP after 60 seconds.	A timer is displayed to the user, indicating the ability to resend OTP after 60 seconds.	Pass
4	The system resends another OTP after the timer closes.		A new OTP code is generated and sent after 60 seconds.	A new OTP code is not generated and sent after 60 seconds.	Fail
5	The system verifies the validity of the OTP code by comparing it to the generated code sent to the given phone number.		The system verifies the entered OTP with the generated code.	The system does not verify the entered OTP with the generated code.	Fail
6	Upon verification, the user shall be directed to the account details page.		The user is directed to the account details page upon successful OTP verification.	The user is not directed to the account details page due to unsuccessful OTP verification.	Fail

Test Case 6: Login

Test Scenario: Registered users want to log in using their credentials.

Preconditions: The user is on the login page.

Postconditions: The user logged in.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user enters their password	lahore123	The system, over an incorrect password, asks for a re-entry	The system asks for a re-entry.	Pass
		lahore@1	The system, over a correct password, should log the user in.	The user is logged in.	Pass

Test Case 7: Logout

Test Scenario: The system should allow the user to logout.

Preconditions: The user is logged in and on the menu.

Postconditions: The user is logged out.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system should allow the user to view the logout option when scrolling down in the menu.		The system gives the user the option to log out.	The system displays the option to log out.	Pass
2	The user clicks on the logout option.		The user should get logged out of the system.	The user is logged out of the system.	Pass

Test Case 8: Reset Password

Test Scenario: The system should allow the user to reset their password.

Preconditions: The user is registered.

Postconditions: The user's password is successfully reset.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system should give the user the		The system gives the user the option to reset their password.	The system gives the user the option to reset their password.	Fail

	option for resetting their password.				

Test Case 9: View Profile

Test Scenario: The system should allow the user to view their profile.
Preconditions: The user is on their profile.
Postconditions: The user can view their profile.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user navigates to the menu option to have details displayed to him including his profile.		The system displays a menu page where the profile details are visible.	The system displays a menu page where the profile details are visible.	Pass
2	The system retrieves the user's profile details and displays them to him.		The system should display the user's profile details; name, and profile picture	The system displays the user's profile details; name, and profile picture	Pass

Test Case 10: Vendor Registration

Test Scenario: The user should enter their details.
Preconditions: The user is on the registration page.
Postconditions: The user has successfully been registered.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	Enter username	fizza	The user already exists in the database, and asks for re-entry.	The system asks for re-entry.	Pass

		fizza123	The username does not exist in the database, hence the system accepts the username	The system accepts the username.	Pass

Test Case 11: Valid Password

Test Scenario: The user should enter a password.
Preconditions: The user is on the registration page.
Postconditions: The user has entered a valid password.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user enters password	123	The system should indicate that it is an invalid password and ask for re-entry.	The system does indicate that it is an invalid password and asks for re-entry.	Pass
		password	The system should indicate that it is an invalid password and ask for re-entry.	The system does indicate that it is an invalid password and asks for re-entry.	Pass
		lahore@1	The system should indicate that it is a valid password.	The system accepts the password.	Pass

Test Case 12: Username Exists

Test Scenario: The user should enter a username.
Preconditions: The user is on the registration page.
Postconditions: The user has entered a username.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user enters their username.	fizza	The username already exists in the database, hence the system asks for a re-entry.	The system asks for a re-entry.	Pass
		fizza123	The username does not exist in the database, hence the system accepts the username.	The system accepts the username.	Pass

Test Case 13: Search Product in Inventory

Test Scenario: The system should allow the user to search for a product from their inventory.
Preconditions: The user is logged in and on the home page.
Postconditions: The system retrieves the matching product's page.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system directs the user to the homepage where the search bar is		The user is on the home page	The user successfully navigates to the home page	Pass
2	The user can search product name from their own inventory	Olpers	The system allows the user to enter the search criteria	The system allows the user to enter the search criteria	Pass
3	The system fetches results for searched product by name	Olpers	The system retrieves product matching to name criteria	The system does not retrieve product matching to name criteria	Pass

Test Case 14: Add Product to Inventory

Test Scenario: The user wants to add a new product to their inventory.
Preconditions: The user is logged in and on the inventory page.
Postconditions: The user has added the product.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user searches for a product by entering its name	Coca Cola	All products under the name Coca Cola should be displayed.	The products under the name Coca Cola are displayed.	Pass
2	The user adds details after	Price: 120 Listed: 28 Available: 23	The details of the product, price, listed amount, and the	The details of the product, price, listed amount and the	Pass

	selecting a product from the listing namely price, listed amount, and available amount.		available amount should be added.	available amount are added.	
3	The system confirms the addition to the user inventory.		The product added to the inventory should be displayed under the SKUs list on the inventory page with its details	The product added to the inventory is displayed under the SKUs list on the inventory page with its details.	Pass

Test Case 15: All Product Search

Test Scenario: The user should be able to search for a product from the product listings.

Preconditions: The user is on the products search page.

Postconditions: The has searched the product.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user enters a product name.	milk	All products containing the word milk should be displayed.	All products containing the word milk are displayed.	Pass
		Lays	All products containing the word Lays should be displayed.	All products containing the word Lays are displayed.	Pass
		Coke	If the product does not exist in the database, no results should be displayed.	No results are displayed.	Pass

Test Case 16: Remove Product

Test Scenario: The user should remove a product from their inventory.

Preconditions: The user is on the product's page.

Postconditions: The user has removed a product.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user enters a product name.	Lays	Product should be displayed along with its description and delete option.	The product is displayed along with the delete option.	Pass
2	The user presses the delete option displayed.		The system should delete the product from the user's inventory.	The product is deleted from the user's inventory.	Pass

Test Case 17: Edit Details of Product

Test Scenario: The user wants to edit the details of a product in their inventory.

Preconditions: The user has added the product to their inventory.

Postconditions: The user has edited the product.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system displays the product inventory page for the user.	Coca Cola Lays lays water	All the products in vendor inventory should be displayed.	All the products in vendor inventory are displayed.	Pass
2	The system gives the user the option to edit the product details: listed amount, available amount, and price.	Selected Product: Coca Cola Old details: Price: 120 Listed: 28 Available: 23	The current details of the product, price, listed amount, and the available amount in the system should be visible when the edit dialog opens.	The current details of the product, price, listed amount, and the available amount in the system are visible when the edit dialog opens.	Pass
3	The user edits the product details over the old details.	Selected Product: Coca Cola New details: Price: 120	The product details should be editable over the current ones displayed.	The product details are editable over the current ones displayed.	Pass

		Listed: 35 Available: 28			
4	The system saves the edited product details.	Coca Cola Details: Price: 120 Listed: 35 Available: 28	The details should be saved and updated in the inventory.	The details get saved and updated in the inventory.	Pass

Test Case 18: View Inventory

Test Scenario: The system should allow the user to view their inventory, i.e., product listings.
Preconditions: The user is logged in and on the inventory page.
Postconditions: The user can view their inventory.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user has clicked on the inventory page and can see their inventory displayed by the system.		The system should display the user's inventory with details.	The system displays inventory to the user but with details.	Pass

Test Case 19: View Current Orders

Test Scenario: The system should allow the user to view their current orders, displaying relevant information for effective order management.
Preconditions: The user is logged in and on the orders page.
Postconditions: The user can view all the current orders received.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user has clicked on the orders page and can see their orders from the stores		The system should display the current orders of the user from all stores that are yet to be delivered.	The system displays the current orders of the user from all stores that are yet to be delivered.	Pass

	displayed by the system.				

Test Case 20: View Grocery Store List

Test Scenario: The user wants to view the grocery store list.
Preconditions: The user is logged in.
Postconditions: The grocery store list is displayed.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user clicks on the "Stores List" section in the menu.	Jalal Sons Esajees	The system should display a list of grocery stores.	The system displays a list of grocery stores.	Pass

Test Case 21: View Grocery Store Profile

Test Scenario: The user wants to view the profile of a specific grocery store.
Preconditions: The user is on the store list page.
Postconditions: The grocery store profile is displayed.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user clicks on a Grocery Store in “Store List”.	Jalal Sons	The system should display all details of the grocery store, namely name, image, and address.	The system displays all details of the grocery store, namely name, image, and address.	Pass

Test Case 22: View Grocery Store Current Order

Test Scenario: The system should allow the user to view the current order placed by the grocery store (profile).
Preconditions: The user is on the grocery store’s profile page.
Postconditions: The user has viewed the current order that they have received from the grocery store.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user navigates to the grocery store's list		The user should be on the grocery store's profile page.	.The user is on the grocery store's profile page.	Pass

	from the menu.				
2	The user clicks on the store's profile to view its current orders.	Jalal Sons: Items (1-4) with Order ID, Quantity, Unit Price, and Total Price Item 1: Quantity: 200 Unit Price 20 Total Price: 4000 Order ID: 1	The system should display a list of all current orders of that grocery store to the user with details such as quantity, unit price, total price, order id, and item id.	The system displays a list of all current orders of that grocery store to the user with details such as quantity, unit price, total price, order id, and item id.	Pass

Test Case 23: View Order History

Test Scenario: The system should enable the user to view the orders they have already completed delivering.
Preconditions: The user is on the orders history page.
Postconditions: The user can view the delivered orders.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user has clicked on the Home page's 'Order History' and can see their order history displayed by the system.		The system should display the order history with details; grocery store name, bill amount, order status, etc.	The system displays the order history with details; grocery store name, bill amount, order status, etc.	Pass

Test Case 24: Edit Profile

Test Scenario: The system should allow the user to edit their profile details.
Preconditions: The user is on their profile.
Postconditions: The user's profile is edited.

#	Test Steps	Test Data	Expected Result	Actual Result	Status

1	The system displays the profile details for the user.	Fizza Adeel	The system should display current details.	The system displays current details.	Pass
2	The system gives the user the option to edit profile details.		The system should give options to edit details.	The system gives options to edit details.	Pass
3	The user can edit the profile details over the old details.	Old: Fizza Adeel New: Fizza A	The system should allow editing of new details over displayed current ones.	The system allows editing of new details over displayed current ones.	Pass
4	The system saves the edit to the profile.	Fizza A	The system should update the information in the database.	The system updates the information in the database.	Pass

Test Case 25: Restrict Product Duplication

Test Scenario: The system should restrict duplication of products.

Preconditions: The user attempts to add an existing product to their inventory.

Postconditions: The system prevents the addition of duplicate products and alerts the user.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user after entering details clicks on the product to get added to their inventory.	Lays already exists, in the inventory, hence Lays	The product should not be added to the inventory.	The product is not added to the inventory.	Pass
2	The system shows an alert to	Lays	The system should display an alert to inform the user of	The system displays an alert to inform the user of duplication and	Pass

	inform of duplication.		duplication and instead direct toward the edit page of the added product.	instead directs toward the edit page of the added product.	

Machine Learning

Introduction

The entire process of training and testing the shortlisted models on the available datasets along with the deployment of the model on the cloud has been documented in detail.

Training and Testing

In FYP I, we have mainly worked on 2 datasets. The local pharmacy dataset and the Corporación Favorita dataset. For each of the dataset, we have implemented different variations of 4 machine learning algorithms: Random Forest, XGBoost, Prophet, and Recurrent Neural Networks (LSTM and GRU).

Local Pharmacy Dataset

Initially, we did not have grocery store datasets, so we started to explore our machine learning options on a local (we collected it ourselves) pharmacy's dataset. The dataset contains sales data for 1 year (300,000 rows). The dataset was cleaned and compiled to as mentioned in Figure 1.0.1.

Pharmacy Dataset Update Glossary

File	Description
D1	Combines monthly pharmacy data from CSV files into a single DataFrame and saves it as `D1.csv`
D2	Reads `D1.csv`, performs analysis (correlation), drops irrelevant columns, and saves the refined data as `D2.csv`
D3	Streamlines date-related columns by splitting them into distinct attributes (date and time), eliminating redundant data columns. Optimizes column order for enhanced dataset clarity, placing the label (`looseqty`) at the end, yielding `D3.csv`
D4	Reads `D3.csv`, converts the 'date' column to datetime format, aggregates sales data based on 'date' and 'itemname,' and saves the combined data as `D4.csv`
D5	Reads `D4.csv`, filters the data for 'itemname' equal to 'PANADOL TAB,' and saves the filtered data as `D5.csv`

Figure 1.0.1 Pharmacy Dataset Update Glossary

Random Forest

We start our exploration with Random Forest (RF). We import the necessary libraries Figure 1.1.1), read a CSV file (D4.csv) into a pandas dataframe, and inspect the first 5 rows of the dataframe (Figure 1.1.2).

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_squared_log_error
```

Figure 1.1.1 Importing necessary libraries

```
df = pd.read_csv('../../Data/Pharmacy/D4.csv')
```

```
df.head()
```

	date	itemname	packunits	expiry	price	looseqty
0	01/07/2022	10CC SHIFA D/SYRINGE(UNJT)(BM)	100	12/12/24	30.00	6
1	01/07/2022	1CC BD SYRINGE	100	12/12/24	30.00	1
2	01/07/2022	3CC SYRINGE INJEKT	100	3/1/24	15.00	3
3	01/07/2022	ACCU CHECK LANCET (CHINA)	200	12/12/24	3.00	50
4	01/07/2022	ACDERMIN GEL	1	5/1/23	278.44	1

Figure 1.1.2 Loading the dataset and inspecting it

The dataframe contains aggregated sales (based on dates) of the products. It has 213,056 rows and 6 columns (Figure 1.1.3). 'looseqtyl' (sales) is the label and the other 5 columns are the features (Figure 1.1.4).

```
df.shape
```

```
(213056, 6)
```

Figure 1.1.3 Shape of the dataframe

```
X = df[['date', 'itemname', 'packunits', 'expiry', 'price']]
y = df['looseqty']
```

Figure 1.1.4 Separating features and label

Pandas' get_dummies function is used to convert categorical attributes to numerical values. This function converts each variable in as many 0/1 variables as there are different values. Boolean columns are generated for date, itemname, and expiry. As a new column is created for each date (both date and expiry), it increases the dimensionality of the dataset tremendously (Figure 1.1.5).

```
X = pd.get_dummies(X)
```

```
X.shape
```

```
(213056, 7477)
```

Figure 1.1.5 Converting categorical attributes using get_dummies and inspecting the shape of the dataframe after that

The dataset is split into training and testing sets using an 80-20 split ratio (Figure 1.1.6). As this is a time series forecasting problem, the first 80% is used for training and the last 20% is used for testing, instead of using a shuffled dataset.

```
split_index = int(0.8 * len(df))
```

```
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]
```

Figure 1.1.6 Splitting data into training set and testing set

Finally, a RF Regressor model is created and trained using the training set (Figure 1.1.7). However, the model never completed training (despite being given a sufficient amount of time to train).

```
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```
rf_model.fit(X_train, y_train)
```

Figure 1.1.7 Creating a RF model and training it on the training set

Google Colab was used then, however, it also crashed as soon as the notebook reached training (Figure 1.1.8).

Figure 1.1.8 Google Colab crashing

We believe this is due to the dimensionality of the data. So, in order to overcome it, we reduced the scope of our dataset to only one product Panadol Tab (Figure 1.2.1). This new dataset has 337 rows and the same 6 columns (Figure 1.2.2).

```
df = pd.read_csv('../../Data/Pharmacy/D5.csv')
```

```
df.head()
```

	date	itemname	packunits	expiry	price	looseqty
0	01/07/2022	PANADOL TAB	200	4/25/24	1.70	60
1	02/07/2022	PANADOL TAB	200	4/25/24	1.70	70
2	03/07/2022	PANADOL TAB	200	4/25/24	1.70	55
3	05/07/2022	PANADOL TAB	200	4/25/24	1.45	20
4	08/07/2022	PANADOL TAB	200	4/25/24	1.70	70

Figure 1.2.1 D5.csv used instead of D4.csv

```
df.shape
```

```
(337, 6)
```

Figure 1.2.2 Shape of the dataframe

The same steps are performed with the new dataset to split into training set and testing set, create a RF model, and train it on the training set. Once the training is complete, the model is used to make predictions on the testing set (Figure 1.2.3). The predictions are evaluated using Root Mean Squared Error (RMSE) and Root Mean Squared Logarithmic Error (RMSLE). The model produced a RMSE of 137 and RMSLE of 0.338.

```
y_pred = rf_model.predict(X_test)
```

```
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
rmsle = np.sqrt(mean_squared_log_error(y_test, y_pred))
```

Figure 1.2.3 Making predictions on the test set and calculating RMSE and RMSLE

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 137.2606889869655
Root Mean Squared Logarithmic Error (RMSLE): 0.3384119387902401
```

Figure 1.2.4 Printing the RMSE and RMSLE values

Both RMSE and RMSLE showed reasonable performance by the model. We wanted to confirm that by plotting a graph of the actual values and the predicted values. While doing so, it generated an error about y_test and y_pred not being of the same dimensionality. This was resolved by converting y_test into a normal np array (Figure 1.2.5).

```
y_test.shape
```

```
(68,)
```

```
y_test = np.array(y_test)
```

Figure 1.2.5 Converting y_test to a normal np array

Upon the graph being plotted, we realised how severely under fitted the model was (Figure 1.2.6). This is due to the model not being able to learn the pattern in the data. Which is due to the fact that there are only 337 rows in the dataset. Out of which, only 269 rows are used to train the model.

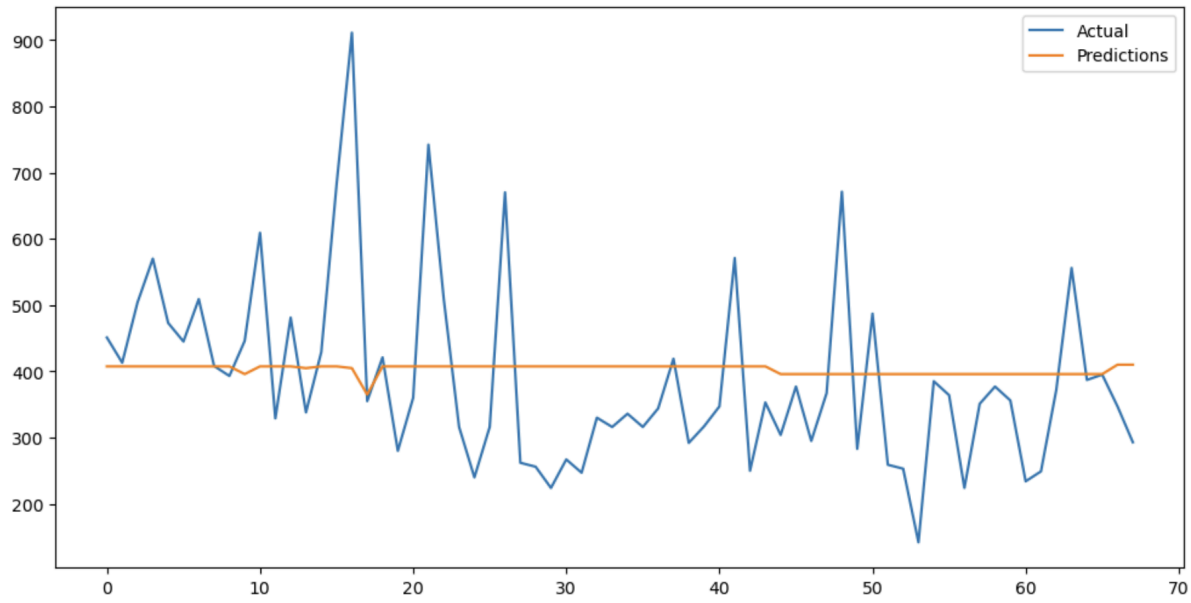


Figure 1.2.6 Actual values vs predicted values

XGBoost

Next, we experimented with a XGBoost model (Figure 1.3.1). The same steps were followed to train the XGBoost model that were performed to train the RF model.

```
xgboost_model = XGBRegressor(objective='reg:squarederror', random_state=42)
```

```
xgboost_model.fit(X_train, y_train)
```

Figure 1.3.1 Training a XGBoost model

The model produced a RMSE of 145 and RMSLE of 0.371 which are also reasonable on paper, however, we wanted to make sure that the model was not underfitting like the RF.

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 145.19829304207062
Root Mean Squared Logarithmic Error (RMSLE): 0.37083087932694353
```

Figure 1.3.2 Printing the RMSE and RMSLE values

To check whether the model is underfitting or not, we plotted the actual values and the predicted values again. The XGBoost model, as seen in Figure 1.3.3, is also underfitting.

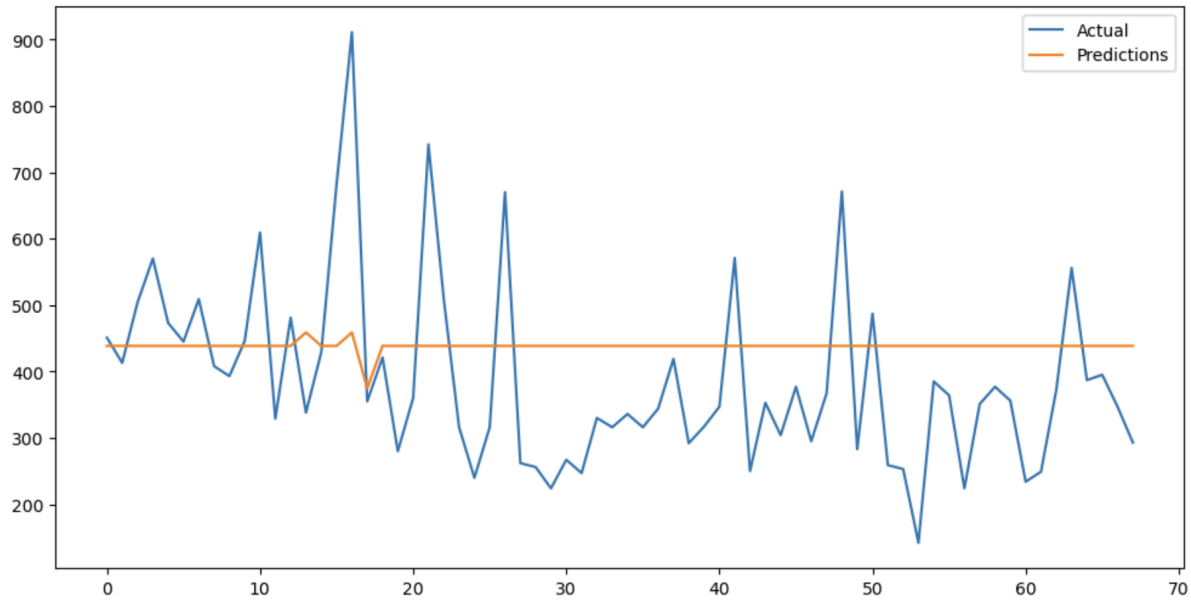


Figure 1.3.3 Actual values vs predicted values

Prophet

After the ensemble learning models failed to perform well on the dataset, we wanted to try a different kind of model. So, we used Prophet by Facebook. We started off by importing the necessary libraries (Figure 1.4.1). Installing and running Prophet was an extremely difficult task but we were finally able to make it work.

```
import pandas as pd
import numpy as np
from prophet import Prophet
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from math import sqrt
```

Figure 1.4.1 Importing necessary libraries

To match the column names expected by Prophet, the 'date' column is renamed to 'ds' and 'looseqty' column is renamed to 'y' (Figure 1.4.2).

```
df.rename(columns={'date': 'ds', 'looseqty': 'y'}, inplace=True)
```

Figure 1.4.2 Renaming date and looseqty columns for Prophet

Then, we create and train the Prophet model on only date (ds) and looseqty (y) as Prophet is a univariate model (Figure 1.4.3).

```
model = Prophet()
model.fit(train)
```

Figure 1.4.3 Creating and training the Prophet model

Then, we create a dataframe with future dates for which predictions will be made. In Prophet, you define the next number of days you want predictions for. Here we define the length of the test array as the number of days we want predictions for (Figure 1.4.4).

```
future = model.make_future_dataframe(periods=len(test))
```

```
forecast = model.predict(future)
```

Figure 1.4.4 Defining time period for predictions

The model produces an RMSE of 160 and RMSLE of 0.401 (Figure 1.4.5). This is worse than the scores from RF and XGBoost models, however, plotting the model shows it is decently fitted, unlike the RF and XGBoost models which were very underfitted (Figure 1.4.6).

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 160.64514907950073
Root Mean Squared Logarithmic Error (RMSLE): 0.4014388777975418
```

Figure 1.4.5 Printing the RMSE and RMSLE values

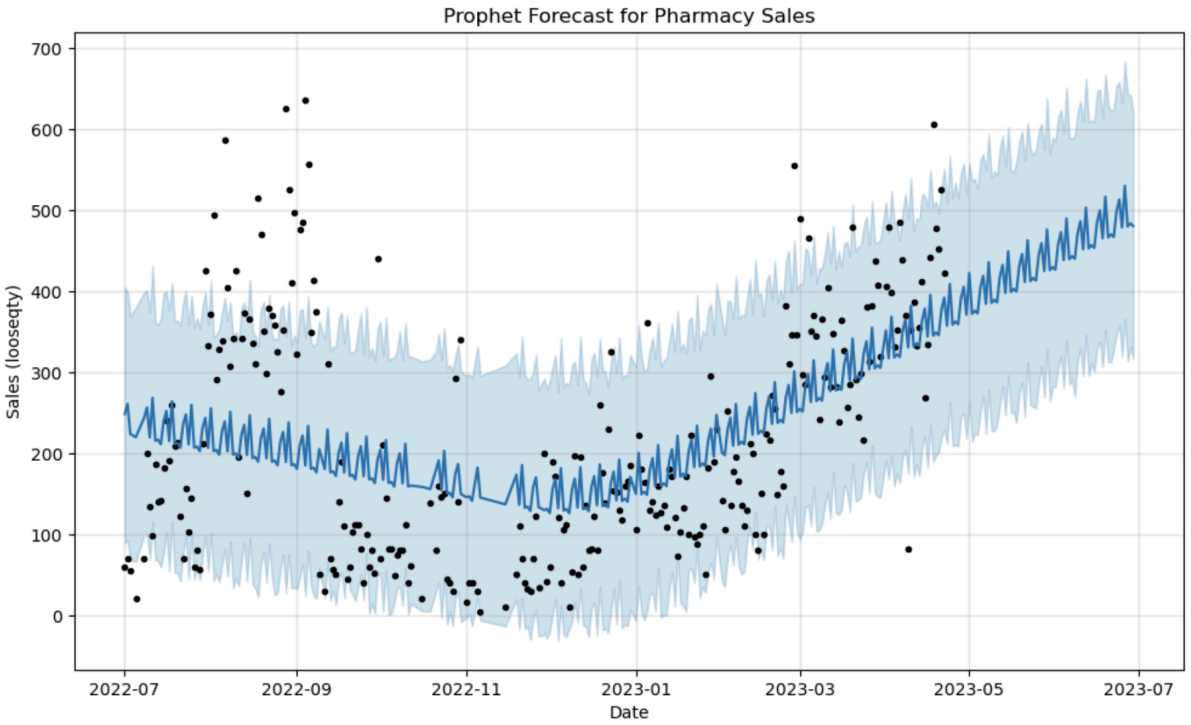


Figure 1.4.6 Prophet model's plot

After the success with Prophet on the Panadol dataset (D5.csv), we wanted to try it for other products as well. However, Prophet creates a different model for each time series (product) and this dataset contains 6053 different products (Figure 1.5.1).

```
unique_products = df['itemname'].unique()
unique_products.size
```

```
6053
```

Figure 1.5.1 Unique products in the Pharmacy dataset

It was not computationally feasible to train 6053 different models. So, we decided to train models for the top 10 products, based on the number of total sales (Figure 1.5.2).

```
total_sales_by_product = df.groupby('itemname')['y'].sum().reset_index()
```

```
top_10_products = total_sales_by_product.nlargest(10, 'y')['itemname'].tolist()
```

```
top_10_products
```

```
['PANADOL TAB',
 'LOPRIN 75MG TAB',
 "SURBEX Z TAB(30'S)",
 'GLUCOPHAGE 500MG TAB',
 'FACE MASK  3 PLY  GREEN  RS(5)',
 "DISPRIN 300MG TAB (600'S)",
 "CALPOL TAB (200'S)",
 'PANADOL EXTRA TAB',
 'METHYCOBAL TAB',
 'NUBEROL FORTE TAB']
```

Figure 1.5.2 Finding the top 10 products

Initially, there were errors while training the model for some of the products. To continue training the rest, try and except were implemented (Figure 1.5.3). Upon further inspection, it was found that the error stemmed from the model predicting slightly negative values and rmsle cannot be calculated for negative values. All the negative predicted values were converted to zero (Figure 1.5.3). We were able to train models for all of the top 10 products.

```
for product in top_10_products:
    product_data = df[df['itemname'] == product]

    try:
        model = Prophet()
        model.fit(product_data)

        future = model.make_future_dataframe(periods=len(product_data))

        forecast = model.predict(future)

        models[product] = model
        predictions[product] = forecast

        actual_values = product_data['y'].values
        predicted_values = forecast.tail(len(product_data))['yhat'].values

        predicted_values = np.maximum(predicted_values, 0)

        rmse = np.sqrt(mean_squared_error(actual_values, predicted_values))
        rmsle = np.sqrt(mean_squared_log_error(actual_values, predicted_values))

        rmse_scores[product] = rmse
        rmsle_scores[product] = rmsle

        print(f"\nRMSE for {product}: {rmse}")
        print(f"RMSLE for {product}: {rmsle}")

    except Exception as e:
        print(f"\nError processing {product}: {e}\n")
```

Figure 1.5.3 Training models for the top 10 products

The average RMSE for the top 10 products was 96 and the average RMSLE for the top 10 products was 1.385.

```
print(f"Average RMSE for the top 10 products: {average_rmse}")
print(f"Average RMSLE for the top 10 products: {average_rmsle}")
```

```
Average RMSE for the top 10 products: 96.95625655059175
Average RMSLE for the top 10 products: 1.3850059626898534
```

Figure 1.5.4 Printing the RMSE and RMSLE values

Recurrent Neural Networks

Lastly, we used a univariate LSTM model. We started off by setting the date as the index and keeping on looseqty (which is our label) in our dataset (Figure 1.6.1).

```
df['date'] = pd.to_datetime(df['date'], format='%d/%m/%Y')
df.set_index('date', inplace=True)
```

```
target_variable = 'looseqty'
df = df[[target_variable]]
```

Figure 1.6.1 Creating a univariate dataframe

Firstly, we create a function (Figure 1.6.2) to convert our dataframe into sequences. The sequence length (which is set to 10) is the number of previous time steps to consider as input features. The function iterates through the data and creates a list of lists containing the values of the previous 10 rows as input features. It then gets the value of the time step immediately following the window of previous time steps. We created a new dataframe which only has the sales (looseqty) column (as this is a univariate model) and used this dataframe to create training sequences for our LSTM model.

```
def create_sequences(data, seq_length):
    sequences = []
    for i in range(len(data) - seq_length):
        seq = data[i:i + seq_length]
        sequences.append(seq)
    return np.array(sequences)
```

```
sequence_length = 10
```

```
sequences = create_sequences(df_scaled, sequence_length)
```

Figure 1.6.2 Generating sequences for training the model

We create a sequential model (Figure 1.6.3), a model with a linear stack of layers. The first layer of the model is an LSTM layer with 50 neurons, where we pass our training sequences of length 10 and only 1 feature (looseqty) per time step. The second layer is the output layer with a single neuron as it is a regression task.

```
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(X_train.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

```
model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test), verbose=2)
```

Figure 1.6.3 Creating and training the LSTM model

The model produced a RMSE of 139 and RMSLE of 0.330 (Figure 1.6.4) which is similar to the evaluations on RF and XGBoost models, however, upon the creation of a plot of the actual values and the predicted values (Figure 1.6.5), it can be clearly seen that the model is not under-fitted.

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 139.43244021165586
Root Mean Squared Logarithmic Error (RMSLE): 0.33028218057622366
```

Figure 1.6.4 Printing the RMSE and RMSLE values

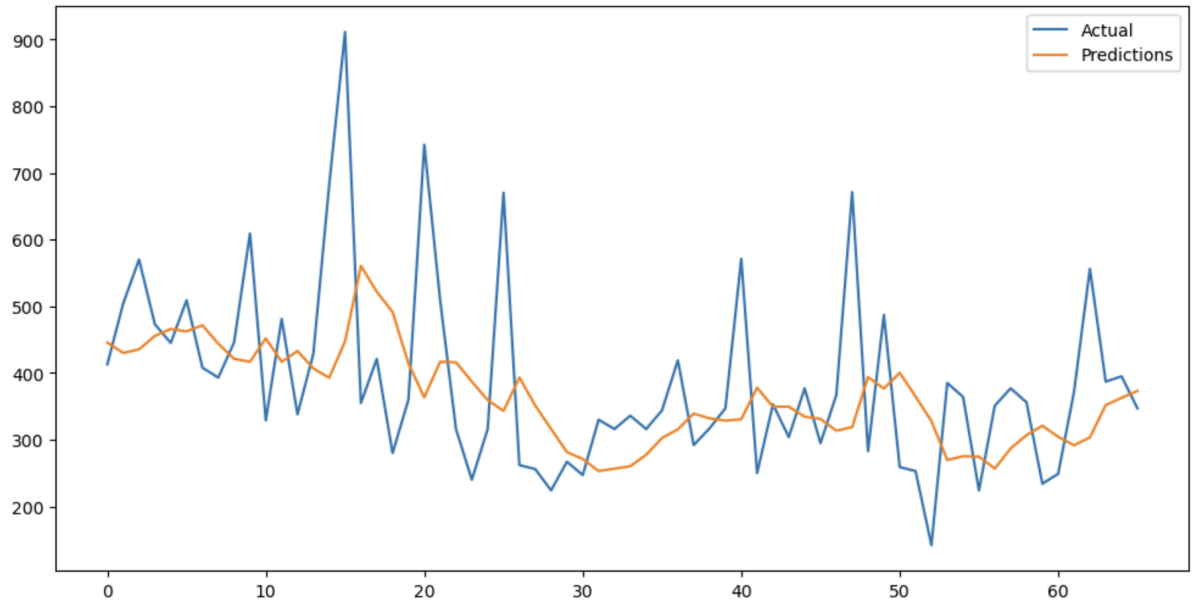


Figure 1.6.5 Actual values vs predicted values

Considering the success with LSTM, we used its variation GRU that achieves similar results while requiring lesser computation. All the same steps were followed to train the GRU model that were performed to train the LSTM model (Figure 1.7.1).

```
model = Sequential()
model.add(GRU(50, activation='relu', input_shape=(X_train.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Figure 1.7.1 Creating and training the GRU model

The model produced a RMSE of 137 and RMSLE of 0.325 (Figure 1.7.2) which is actually slightly better than the LSTM model (we expected the performance to fall) and the plot (Figure 1.7.3) also shows that the model is not under-fitted.

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 137.2803728457543
Root Mean Squared Logarithmic Error (RMSLE): 0.32572063927800055
```

Figure 1.7.2 Printing the RMSE and RMSLE values

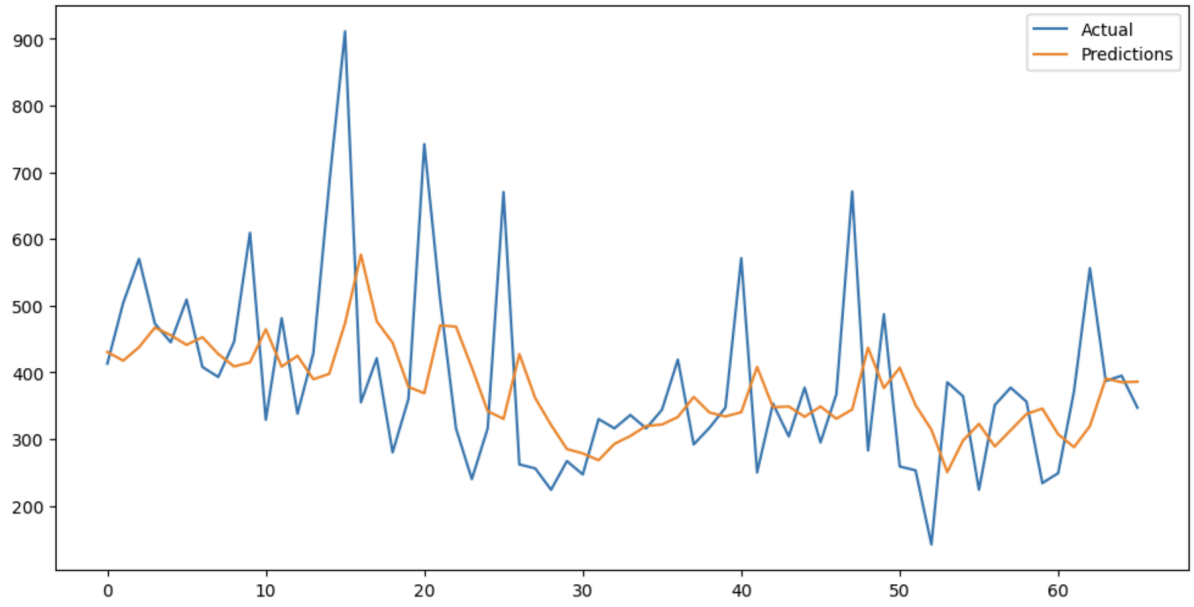


Figure 1.7.3 Actual values vs predicted values

Summary

Here is a side by side comparison of all the models trained and tested on the Pharmacy dataset (D5.csv).

Model	RMSE	RMSLE
Random Forest	137	0.338
XGBoost	145	0.371

Prophet	160	0.401
LSTM	139	0.330
GRU	137	0.325

Figure 1.8.1 Model Summary

Corporación Favorita Grocery Sales Forecasting

Initially, we were unable to train models on the Corporación Favorita dataset due to its sheer size (3+ million rows), despite utilising services like Kaggle (Figure 2.1.1) and Google Colab (Figure 2.1.2). As suggested by Ma'am Huda, we used batch training to overcome this issue.

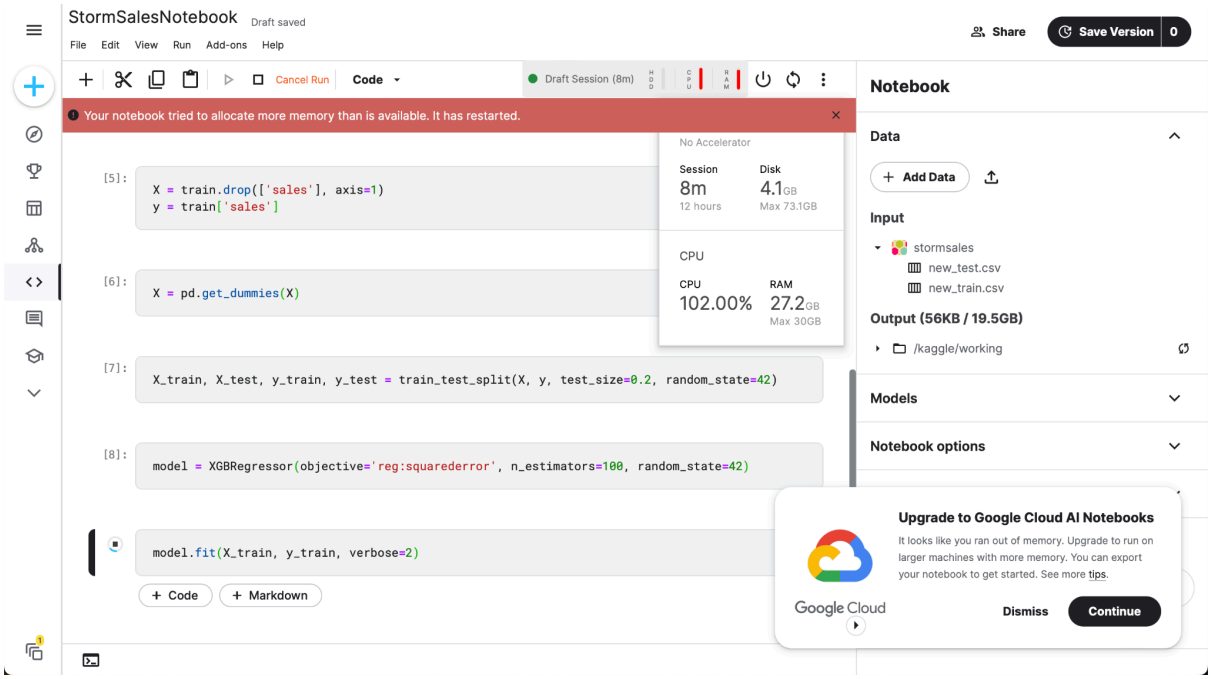


Figure 2.1.1 Kaggle crashing

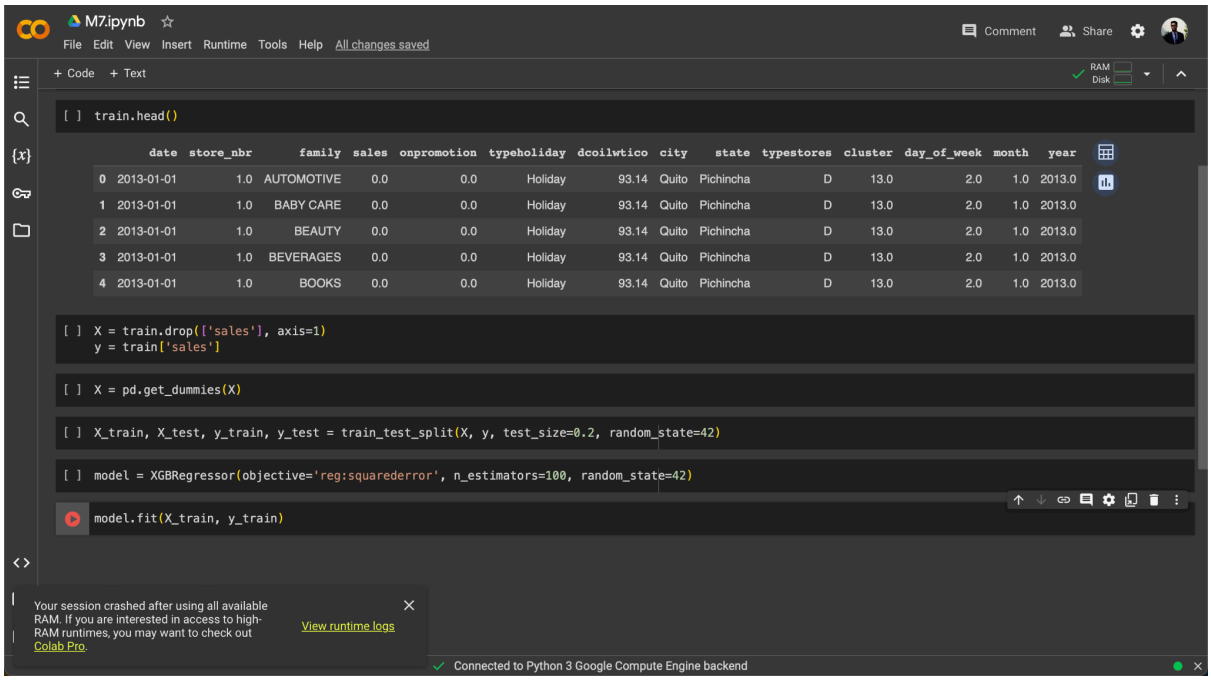


Figure 2.1.2 Google Colab crashing

During the initial exploration of the Corporación Favorita dataset, we kept the date feature as seen in the train.head() in Figure 2.1.2. The preprocessing was done on the available datasets as mentioned in the Design Document. While making predictions on the unseen dataset we found an issue. We were using the Pandas' get_dummies function to convert our categorical attributes to numerical values. This function converts each variable in as many 0/1 variables as there are different values. The boolean columns generated for family, typeholiday, city, state, and typestore worked as their values stay consistent across the train and test data, however, the date column generated error as the model was not trained on the dates (columns) from the test data. To overcome this, we combined both the datasets (Figure 2.1.3) and then applied the get_dummies function (a temporary fix as it would not work on any dates other than the ones in the train and test data). The data was split back into test and train after being transformed.

```
In [3]: combined = pd.concat([train, test], ignore_index=True)

In [4]: combined = pd.get_dummies(combined)

In [5]: first_range = (0, 3000887)
        second_range = (3000888, combined['id'].max())

In [6]: first_range = (0, 3000887)
        second_range = (3000888, combined['id'].max())

In [7]: train = combined[(combined['id'] >= first_range[0]) & (combined['id'] <= first_range[1])]
        test = combined[(combined['id'] >= second_range[0]) & (combined['id'] <= second_range[1])]
```

Figure 2.1.3 Combining train and test dataset to apply get_dummies together

The function increased the dimensionality of the data, making the training even slower. The train data, containing 1782 boolean columns, occupied 5.2 GB in the memory (Figure 2.1.4).

In [8]:
```
train.head()
```

Out[8]:

	id	store_nbr	sales	onpromotion	dcoilwtico	cluster	day_of_week	month	year	date_2013-01-01	...	state_Pastaza	state_Pich
0	0	1	0.0	0	93.14	13	2	1	2013	True	...	False	
1	1	1	0.0	0	93.14	13	2	1	2013	True	...	False	
2	2	1	0.0	0	93.14	13	2	1	2013	True	...	False	
3	3	1	0.0	0	93.14	13	2	1	2013	True	...	False	
4	4	1	0.0	0	93.14	13	2	1	2013	True	...	False	

5 rows × 1791 columns

In [9]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3000888 entries, 0 to 3000887
Columns: 1791 entries, id to typestores_E
dtypes: bool(1782), float64(2), int64(7)
memory usage: 5.2 GB
```

Figure 2.1.4 Information about the updated train dataset

Considering the size of the dataset, batch training was used (Figure 2.1.4). The dataset was divided in batches of 10,000 samples. The total number of batches were calculated to cover the entire training dataset (which in this case were 301). Lastly, the model is trained one batch at a time.

In [15]:
```
batch_size = 10000
```

In [16]:
```
num_batches = len(X) // batch_size + 1
```

In [17]:
```
model = XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
```

In [18]:
```
for i in range(num_batches):
    start_idx = i * batch_size
    end_idx = (i + 1) * batch_size

    X_batch = X.iloc[start_idx:end_idx]
    y_batch = y.iloc[start_idx:end_idx]

    model.fit(X_batch, y_batch)

    print(f"Batch {i + 1}/{num_batches} completed")
```

```
Batch 1/301 completed
Batch 2/301 completed
Batch 3/301 completed
Batch 4/301 completed
Batch 5/301 completed
```

Figure 2.1.4 Batch Training

The predictions were generated on the XGBRegressor model. All the slightly negative predictions were converted to zero to ensure calculating rmsle is possible (Figure 2.1.5). A submission csv file was created for the ongoing Store Sales competition on Kaggle.

```
In [19]: predictions = model.predict(test_data)

In [20]: submission = pd.DataFrame({'id': test['id'], 'sales': predictions})

In [21]: submission['sales'] = submission['sales'].apply(lambda x: max(0, x))

In [22]: submission.to_csv('../../Data/Kaggle/StoreSales/submission.csv', index=False)
```

Figure 2.1.5 Generating predictions, fixing them, and creating a submission file

We submitted the predictions generated using the XGBRegressor model in the Store Sales competition on Kaggle (Figure 2.1.6), producing an rmsle of 1.08120 on the unseen dataset and ranking 575th on the leaderboard.

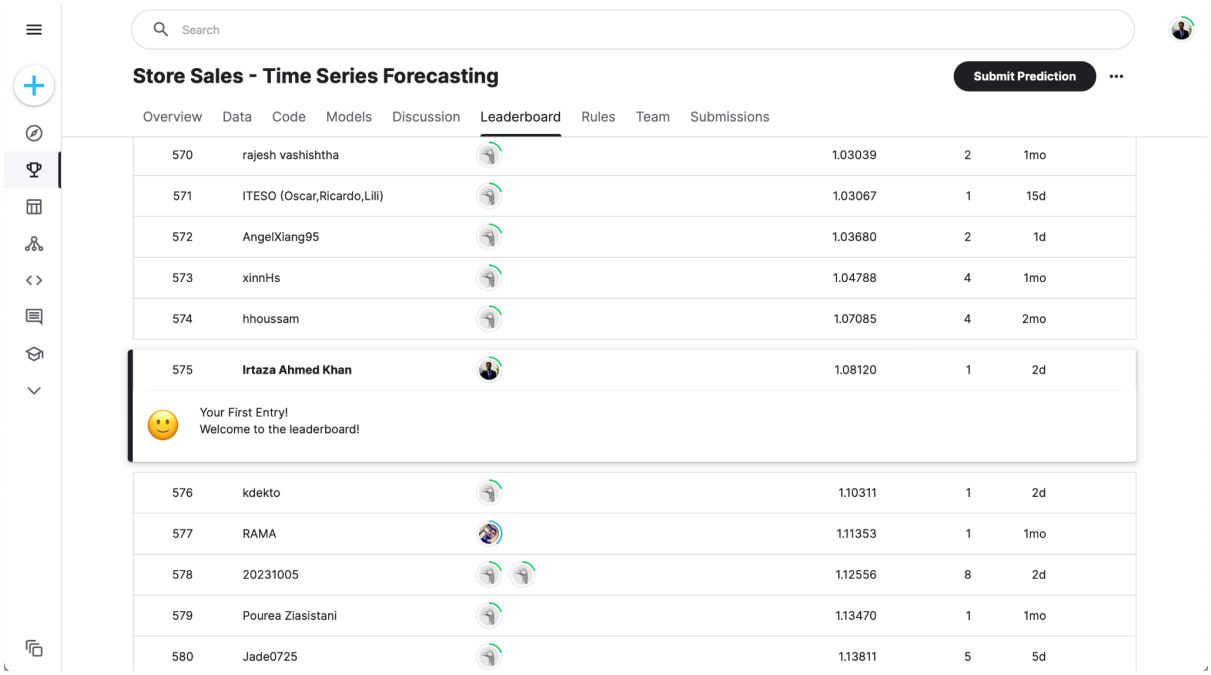


Figure 2.1.6 Store Sales - Time Series Forecasting Leaderboard

To overcome the issue of increased dimensionality, initially the date column was dropped and 4 separate features were generated from it (day_of_week, day, month, and year) and eventually get_dummies (one hot encoding) was replaced with Label Encoder to further reduce the dimensionality. However, Label Encoder provides an arbitrary order to categorical values which negatively affects the performance of the model.

Random Forest

Just like with the Pharmacy dataset, we started our implementation with Random Forest (RF). We read the processed CSV file (new_train.csv) into a pandas dataframe, and inspected the first 5 rows of the dataframe (Figure 2.2.1).

```
df = pd.read_csv("../../Data/Kaggle/StoreSales/new_train.csv")
```

```
df.head()
```

family	sales	onpromotion	typeholiday	dcoilwtico	city	state	typestores	cluster	day_of_week	day	month	year
AUTOMOTIVE	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
BABY CARE	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
BEAUTY	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
BEVERAGES	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013
BOOKS	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1	1	2013

Figure 2.2.1 Loading the dataset and inspecting it

Pandas' get_dummies function is used to convert categorical attributes to numerical values (Figure 2.2.2). Boolean columns are generated for family, typeholiday, city, state, and typestores. As we did not use this function on date (we split it into day of week, day, month, and year) this time, it did not increase the dimensionality to the same extent it did in the pharmacy dataset.

```
X = pd.get_dummies(X)
```

```
X.shape
```

```
(3000888, 90)
```

Figure 2.2.2 Converting categorical attributes using get_dummies

Batch of size 32 is a rule of thumb and considered a good initial choice so the batch size was set to 32 and the number of batches were calculated (Figure 2.2.3).

```
batch_size = 32
```

```
num_batches = len(X_train) // batch_size + 1
```

Figure 2.2.3 Setting batch size and calculating number of batches

We created and trained a RF model. Considering the size of the dataset, batch training was used (Figure 2.2.4). The dataset was divided in batches of 32 samples. The total number of batches were calculated to

cover the entire training dataset (which in this case were 75023). Lastly, the model is trained one batch at a time.

```
model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```
for i in range(num_batches):
    start_idx = i * batch_size
    end_idx = (i + 1) * batch_size

    X_batch = X_train.iloc[start_idx:end_idx]
    y_batch = y_train.iloc[start_idx:end_idx]

    model.fit(X_batch, y_batch)

    print(f"Batch {i + 1}/{num_batches} completed")
```

```
Batch 1/75023 completed
```

Figure 2.2.4 Using batch training to train the RF model

It took an unreasonable amount of time to complete the training. The model produced a significantly high RMSE of 1418 and RMSLE of 2.283 (Figure 2.2.5)

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 1418.1705197469257
Root Mean Squared Logarithmic Error (RMSLE): 2.2832200118427144
```

Figure 2.2.5 Printing the RMSE and RMSLE values

The graph was plotted for 100 (for better visibility) actual and predicted values from the middle of the test set (Figure 2.2.6) to find out the cause for such high RMSE and RMSLE. As seen in Figure 2.2.7, the model is severely under-fitted.

```
plt.figure(figsize=(12, 6))
plt.plot(y_test[300100:300200], label='Actual')
plt.plot(y_pred[300100:300200], label='Predictions')
plt.legend()
plt.show()
```

Figure 2.2.6 Plotting 100 values from actual and predicted values

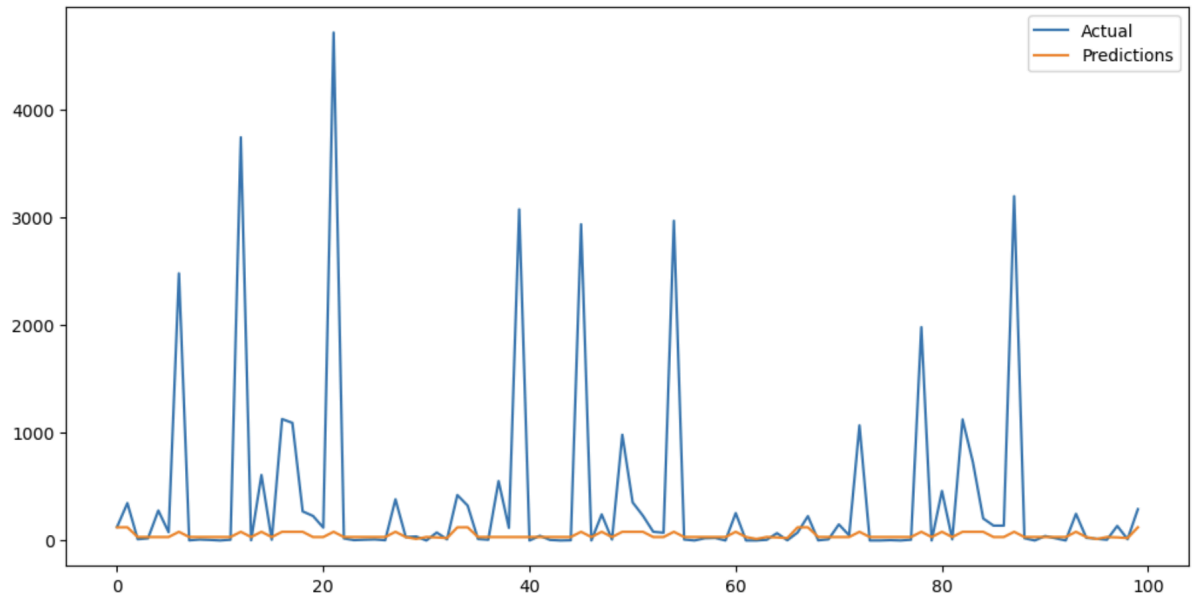


Figure 2.2.7 Actual values vs predicted values

While training the XGBoost, the kernel kept crashing with the batch size of 32. Upon increasing the batch size, XGBoost showed much better performance than RF so we trained another RF model with increased batch size (Figure 2.3.1). With batch size set to 512, the model was trained on 4689 batches (Figure 2.3.2).

```
batch_size = 512
```

Figure 2.3.1 Batch size

```
Batch 1/4689 completed
Batch 2/4689 completed
Batch 3/4689 completed
Batch 4/4689 completed
Batch 5/4689 completed
```

Figure 2.3.2 Total number of batches

The model did not only train faster but it also performed significantly better with a RMSE of 975 and RMSLE of 1.618 (Figure 2.3.3). To ensure the under-fitting problem was resolved, we plotted a graph of the 100 actual and predicted values and as it can be seen in Figure 2.3.4, the model is very well fitted (Figure 2.3.4).

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 975.9726172221381
Root Mean Squared Logarithmic Error (RMSLE): 1.6175720470454031
```

Figure 2.3.3 Printing the RMSE and RMSLE values

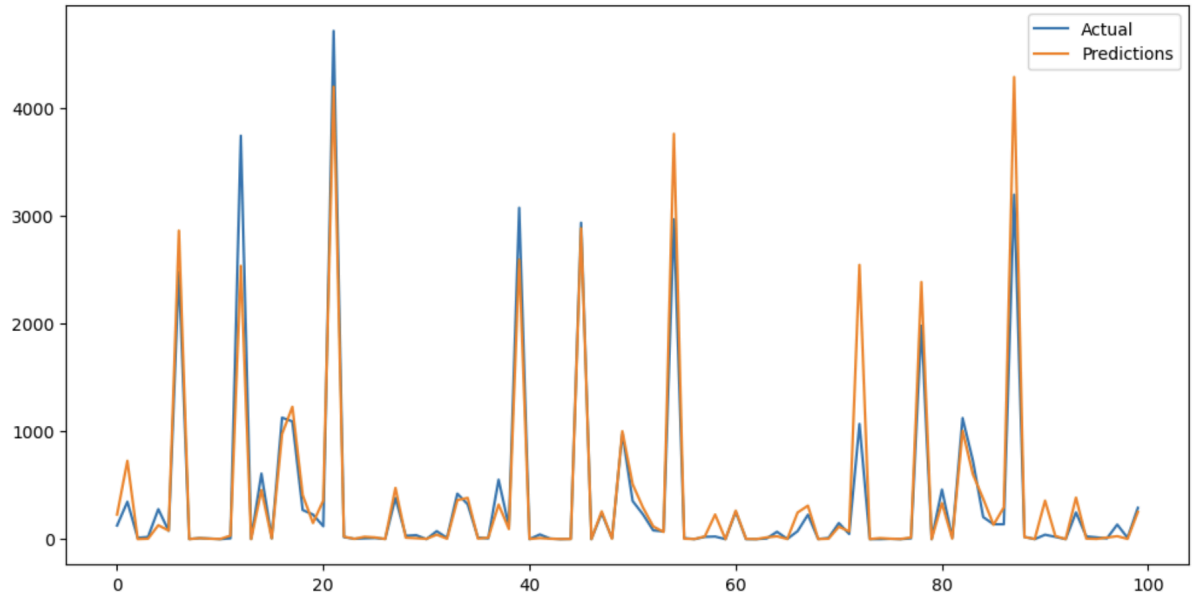


Figure 2.3.4 Actual values vs predicted values

Considering the improvement in results by increasing the batch size, we increased the batch size further (Figure 2.4.1). With a batch size of 1024, the model was trained on 2345 batches (Figure 2.4.2). However, there was no difference in the RMSE or RMSLE, as they stayed exactly the same at 975 and 1.618 respectively (Figure 2.4.3). The fit of the model as seen in Figure 2.4.4 is also identical.

```
batch_size = 1024
```

Figure 2.4.1 Batch size

```
Batch 1/2345 completed
Batch 2/2345 completed
Batch 3/2345 completed
Batch 4/2345 completed
Batch 5/2345 completed
```

Figure 2.4.2 Total number of the batches

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 975.9726172221381
Root Mean Squared Logarithmic Error (RMSLE): 1.6175720470454031
```

Figure 2.4.3 Printing the RMSE and RMSLE values

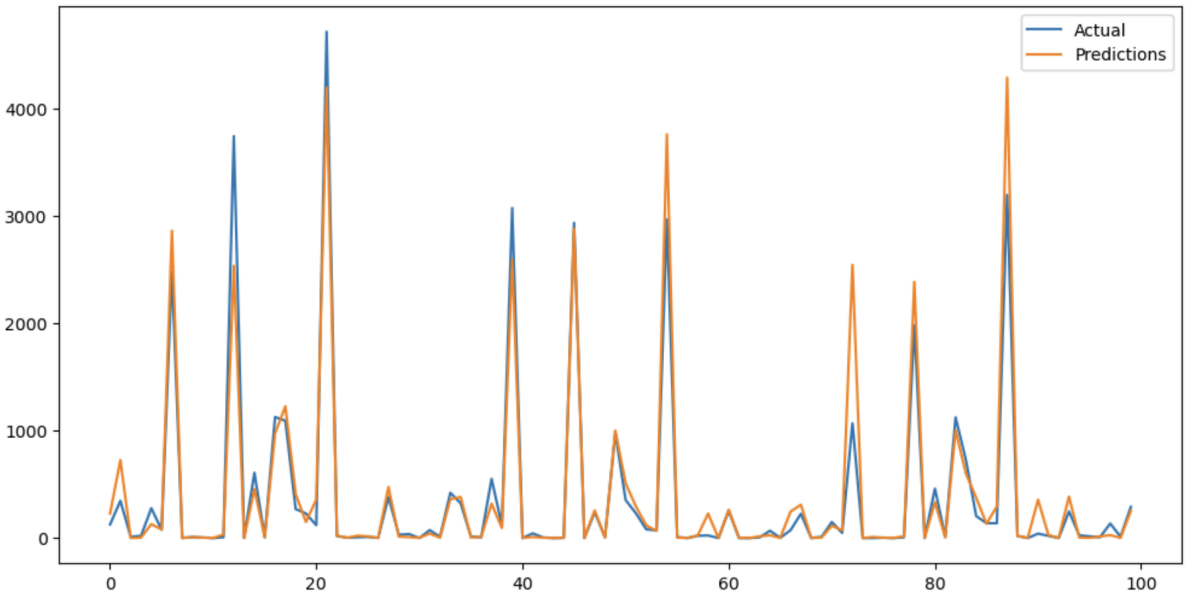


Figure 2.4.4 Actual values vs predicted values

Random Forest took a lot of time to train, even with a batch size of 1024, as it did not utilise all the available resources (Figure 2.4.5).

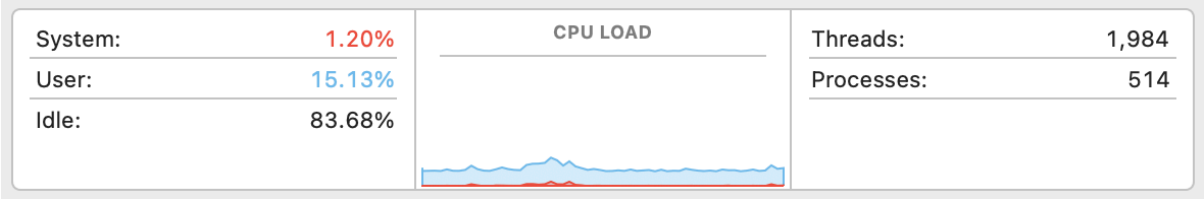


Figure 2.4.5 CPU usage

XGBoost

Next, we trained a XGBoost model (Figure 2.5.1) with the same configurations as the RF model, however, the kernel kept crashing (Figure 2.5.2).

```
model = XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
```

Figure 2.5.1 Creating a XGBoost model

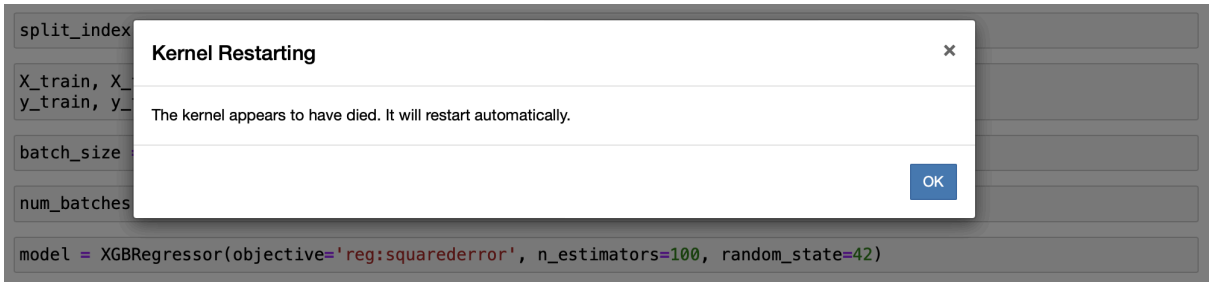


Figure 2.5.2 Kernel died error

Increasing the batch size to 512 solved the issue (Figure 2.5.3). The model performed significantly better (with an RMSE of 1119 AND RMSLE 1.462) than the RF model (with 32 batch size) but when the RF model was also trained with a batch size of 512, it outperformed XGBoost (Figure 2.5.4). The model is not under-fitted and is able to generalise well on unseen as seen in Figure 2.5.5.

```
batch_size = 512
```

Figure 2.5.3 Batch size

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 1119.766051910334
Root Mean Squared Logarithmic Error (RMSLE): 1.4624035138159117
```

Figure 2.5.4 Printing the RMSE and RMSLE values

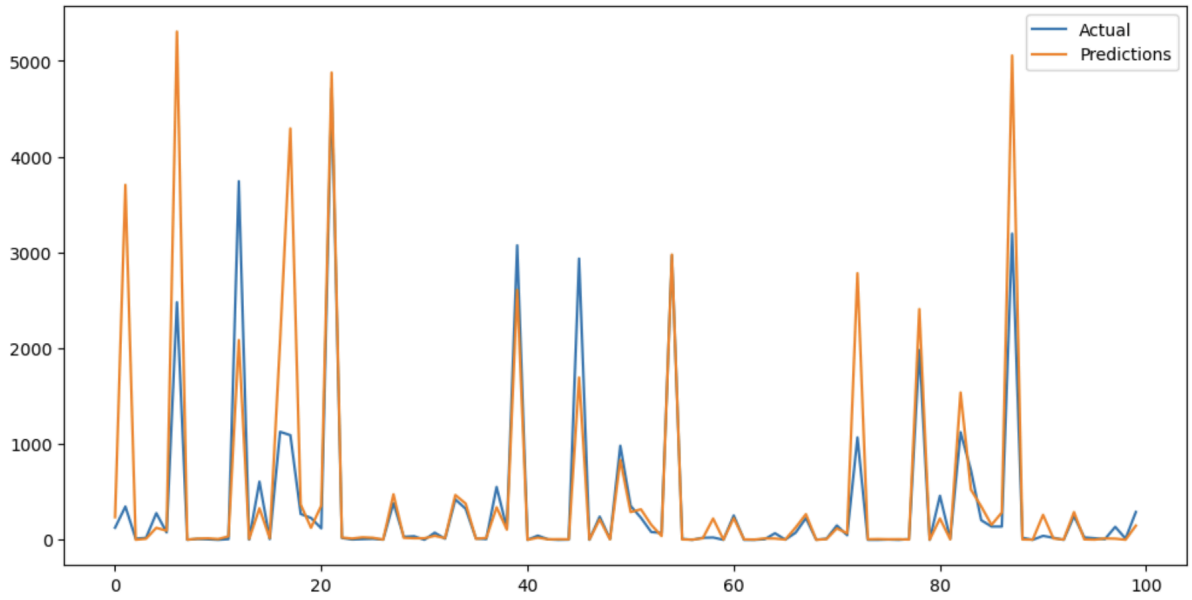


Figure 2.5.5 Actual values vs predicted values

The XGBoost trained significantly faster than the RF model. One of the reasons being that XGBoost utilised all the available resources while training (Figure 2.5.6). This was a key reason for XGBoost being used more than RF for experimentation during the project despite producing slightly worse results than RF.

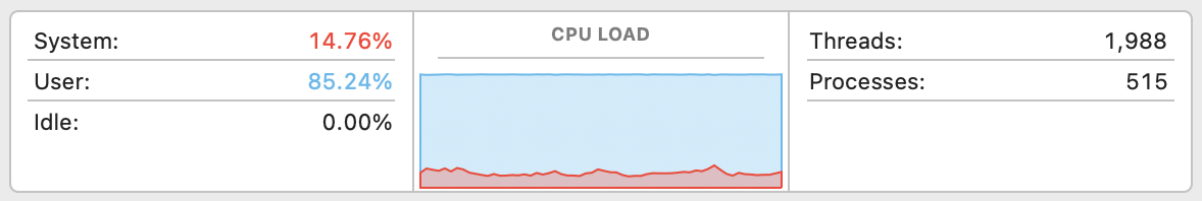


Figure 2.5.6 CPU usage

Prophet

Prophet works best with time series that have several seasons of historical data which makes Prophet a very good fit for this dataset as the dataset contains historical data for several years. As Prophet creates a different model for each time series, in this case store and product, it would create (number of stores * number of unique products) models which was computationally not infeasible. So, we created one model for store number = 1 and family = 3 (Beverages) (Figure 2.6.1). The model was trained only on date and sales as it is a univariate model.

```
df = df[(df['store_nbr'] == 1) & (df['family'] == 3)]
```

```
df.head()
```

id	date	store_nbr	family	sales	onpromotion	typeholiday	dcoilwtico	city	state	typestores	cluster	day_of_week	day
3	2013-01-01	1	3	0.0	0	3	93.14000	18	12	3	13	2	1
1785	2013-01-02	1	3	1091.0	0	4	93.14000	18	12	3	13	3	2
3567	2013-01-03	1	3	919.0	0	4	92.97000	18	12	3	13	4	3
5349	2013-01-04	1	3	953.0	0	4	93.12000	18	12	3	13	5	4
7131	2013-01-05	1	3	1160.0	0	4	93.12009	18	12	3	13	6	5

Figure 2.6.1 Only sales for beverages (family = 3) at store number 1

The model produced a very low RMSE (653) and RMSLE (0.554) score (Figure 2.6.2), however, it cannot be compared with the evaluations earlier as we don't know if the Prophet will perform just as well on other families and stores. The plot of the model does show the model's ability to fit well on such time series (Figure 2.6.3).

```
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"Root Mean Squared Logarithmic Error (RMSLE): {rmsle}")
```

```
Root Mean Squared Error (RMSE): 653.0936302929073
Root Mean Squared Logarithmic Error (RMSLE): 0.5537333343983055
```

Figure 2.6.2 Printing the RMSE and RMSLE values

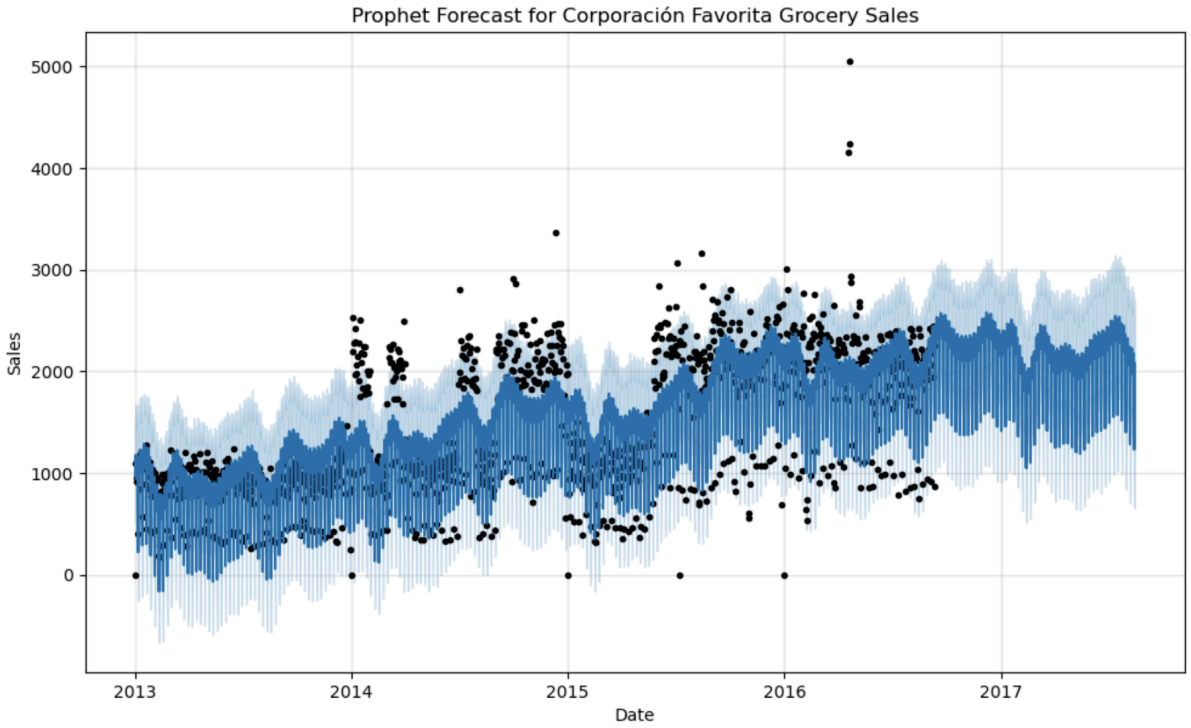


Figure 2.6.3 Prophet model's plot

Recurrent Neural Networks

We implemented 3 recurrent neural network (RNN) models for the Corporación Favorita dataset: LSTM (Univariate), GRU (Univariate), and LSTM (Multivariate). We started with the univariate LSTM model. In Figure 2.7.1, we import the necessary libraries and then read a CSV file (processed training dataset) into a pandas dataframe.

```
import tensorflow as tf
import pandas as pd
import numpy as np
```

```
df = pd.read_csv("../../Data/Kaggle/StoreSales/processed_train_v2.csv")
```

Figure 2.7.1 Importing libraries and loading the dataset

Firstly, we create a function (Figure 2.7.2) to convert our dataframe into X (training sequences) and y (labels) numpy arrays. Numpy arrays are used to make it easier to manipulate the data. The window_size (which is set to 7) is the number of previous time steps to consider as input features. The function iterates through the data, excluding the last 7 rows, and creates a list of lists containing the values of the previous 7 rows as input features. It then gets the value of the time step immediately following the window of previous time steps. We created a new dataframe which only has the sales column (as this is a univariate model) and used this dataframe to create training sequences for our LSTM (Univariate) model.

```
def df_to_X_y(df, window_size=7):
  df_as_np = df.to_numpy()
  X = []
  y = []
  for i in range(len(df_as_np)-window_size):
    row = [[a] for a in df_as_np[i:i+window_size]]
    X.append(row)
    label = df_as_np[i+window_size]
    y.append(label)
  return np.array(X), np.array(y)
```

```
WINDOW_SIZE = 7
```

```
X1, y1 = df_to_X_y(sales, WINDOW_SIZE)
```

Figure 2.7.2 Converting the data frame into training sequences

Then, we split the data into training (80%), validation (10%), and test sets (10%). Array slicing operations are used, as shown in Figure 2.7.3, to ensure the model is trained on the first 80%, validated on the next 10%, and finally tested on the last 10%.

```
X1.shape, y1.shape
```

```
((3000881, 7, 1), (3000881,))
```

```
X_train1, y_train1 = X1[:2400710], y1[:2400710]
X_val1, y_val1 = X1[2400710:2700799], y1[2400710:2700799]
X_test1, y_test1 = X1[2700799:], y1[2700799:]
```

```
X_train1.shape, y_train1.shape, X_val1.shape, y_val1.shape, X_test1.shape, y_test1.shape
```

```
((2400710, 7, 1),
 (2400710,),
 (300089, 7, 1),
 (300089,),
 (300082, 7, 1),
 (300082,))
```

Figure 2.7.3 Splitting the data in training, validation, and test sets

Next, we import various components from TensorFlow Keras specific to the training of the model. We create a sequential model (Figure 2.7.4), a model with a linear stack of layers. The first layer of the model is an InputLayer, where we pass our training sequences of length 7 and only 1 feature (sales) per time step as it is a univariate model. The second layer is an LSTM layer with 64 neurons. The third layer is a

Dense layer with 8 ReLUs. Lastly, the output layer is a dense layer with a single ReLU neuron as it's a regression task.

```
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import *
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.metrics import RootMeanSquaredError
from tensorflow.keras.optimizers import Adam
```

```
model1 = Sequential()
model1.add(InputLayer((7, 1)))
model1.add(LSTM(64))
model1.add(Dense(8, 'relu'))
model1.add(Dense(1, 'relu'))
```

Figure 2.7.4 Importing libraries and initialising our model

Then, a ModelCheckpoint callback is created and is configured to save the best model during training based on the validation loss (Figure 2.7.5). The chosen loss function is Mean Squared Error, the optimizer is Adam (with a learning rate of 0.0001), and the metric for evaluation is Root Mean Squared Error. The training is performed with a batch size of 1000, over 10 epochs, and the ModelCheckpoint callback (cp1) is specified to save the best model. The lowest RMSE on validation set (566) is achieved in the 10th epoch (Figure 2.7.6).

```
cp1 = ModelCheckpoint('model1/', save_best_only=True)
```

```
model1.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=0.0001), metrics=[RootMeanSquaredError()])
```

```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use th
e legacy Keras optimizer instead, located at `tf.keras.optimizers.legacy.Adam`.
```

```
model1.fit(X_train1, y_train1, validation_data=(X_val1, y_val1), batch_size=1000, epochs=10, callbacks=[cp1])
```

```
Epoch 1/10
2397/2401 [============================>.] - ETA: 0s - loss: 316139.6875 - root_mean_squared_error: 562.2630INFO:te
nsorflow:Assets written to: model1/assets
INFO:tensorflow:Assets written to: model1/assets
2401/2401 [==============================] - 28s 12ms/step - loss: 316046.9375 - root_mean_squared_error: 562.1805
- val_loss: 659186.1875 - val_root_mean_squared_error: 811.9028
```

Figure 2.7.5 Compiling and fitting the model

```
Epoch 10/10
2398/2401 [============================>.] - ETA: 0s - loss: 146996.0312 - root_mean_squared_error: 383.4006INFO:te
nsorflow:Assets written to: model1/assets
INFO:tensorflow:Assets written to: model1/assets
2401/2401 [==============================] - 27s 11ms/step - loss: 146947.9375 - root_mean_squared_error: 383.3379
- val_loss: 321167.2188 - val_root_mean_squared_error: 566.7162
```

Figure 2.7.6 Lowest RMSE

Finally, we use the trained model to make predictions on the test set (data it has never seen before). The predictions are obtained by calling the predict method on the model, and flatten() is used to convert the predictions into a 1D array. Then, Matplotlib is used to plot the test predictions and actual values. For better clarity, only hundred data points (900 to 1000) are visualised. As it can be seen in Figure 2.7.7, the

model does a very good job at making predictions as the predicted values are very close to the actual values.

```
test_predictions = model1.predict(X_test1).flatten()
test_results = pd.DataFrame(data={'Test Predictions':test_predictions, 'Actuals':y_test1})
```

```
9378/9378 [==============================] - 5s 485us/step
```

```
plt.plot(test_results['Test Predictions'][900:1000])
plt.plot(test_results['Actuals'][900:1000])
```

```
[<matplotlib.lines.Line2D at 0x324147250>]
```

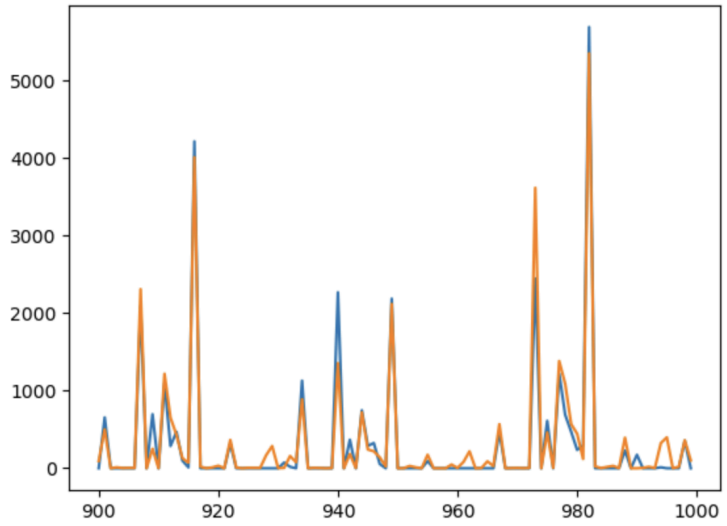


Figure 2.7.7 Test predictions vs actual values plotted

Shown in Figure 2.7.8, the second model follows a similar architecture as the first model with the main difference being the use of a GRU layer instead of an LSTM layer. The model showed similar performance to the LSTM model while taking less time to train.

```
model2 = Sequential()
model2.add(InputLayer((7, 1)))
model2.add(GRU(64))
model2.add(Dense(8, 'relu'))
model2.add(Dense(1, 'relu'))
```

Figure 2.7.8 GRU (Univariate)

Finally, the third model (Figure 2.7.9) follows a similar architecture as the previous models, but it has a different input shape, where each time step has 14 features.

```
model3 = Sequential()
model3.add(InputLayer((7, 14)))
model3.add(LSTM(64))
model3.add(Dense(8, 'relu'))
model3.add(Dense(1, 'relu'))
```

Figure 2.7.9 LSTM (Multivariate)

```
Epoch 10/10
74999/75023 [============================>.] - ETA: 0s - loss: 2.4119 - root_mean_squared_error: 1.5530INFO:tensorf
low:Assets written to: model3/assets
INFO:tensorflow:Assets written to: model3/assets
75023/75023 [==============================] - 109s 1ms/step - loss: 2.4115 - root_mean_squared_error: 1.5529 - val
_loss: 1.8977 - val_root_mean_squared_error: 1.3776
```

Figure 2.7.10 Lowest RMSE on LSTM (Multivariate)

Summary

Here is a side by side comparison of all the models trained and tested on the Corporación Favorita dataset.

Model	Batch Size	RMSE	RMSLE
Random Forest	32	1418	2.283
Random Forest	512	975	1.618
Random Forest	1024	975	1.618
XGBoost	512	1119	1.462
Prophet*	-	653	0.554
LSTM	1000	566	-
GRU	1000	1071	-

Figure 2.8.1 Model Summary

Prophet was only trained on a subset of the dataset (Figure 2.6.1).

Deployment

For the purpose of giving a demo of the ML model, we created a web application using Flask, a Python web framework (recommended by our external advisor). The development initially started on GitHub Codespaces, but was shifted to VS Code due to server issues. A Conda environment was set up and required packages were installed (Figure 3.1).

requirements.txt U

ModelDeployment > requirements.txt

```
blinker==1.7.0
click==8.1.7
Flask==3.0.0
gunicorn==21.2.0
itsdangerous==2.1.2
Jinja2==3.1.2
joblib==1.3.2
MarkupSafe==2.1.3
numpy==1.26.2
packaging==23.2
pandas==2.1.4
pickle-mixin==1.0.2
python-dateutil==2.8.2
pytz==2023.3.post1
scikit-learn==1.3.2
scipy==1.11.4
six==1.16.0
threadpoolctl==3.2.0
tzdata==2023.3
Werkzeug==3.0.1
xgboost==1.7.3
```

Figure 3.1 pip freeze > requirements.txt

To reduce the computational requirements, Label Encoder was used for each categorical attribute (Figure 3.2).

In [10]:
```
family_encoder = LabelEncoder()
typeholiday_encoder = LabelEncoder()
city_encoder = LabelEncoder()
state_encoder = LabelEncoder()
typestores_encoder = LabelEncoder()
```

In [11]:
```
X['family_encoded'] = family_encoder.fit_transform(X['family'])
X['typeholiday_encoded'] = typeholiday_encoder.fit_transform(X['typeholiday'])
X['city_encoded'] = city_encoder.fit_transform(X['city'])
X['state_encoded'] = state_encoder.fit_transform(X['state'])
X['typestores_encoded'] = typestores_encoder.fit_transform(X['typestores'])
```

Figure 3.2 Label Encoder

Then, the encoder for each attribute was exported using Pickle (Figure 3.3) and the model was exported using Joblib (Figure 3.4). Initially, Pickle was used for both however the model did not work and upon research we found out that Joblib is more suitable for exporting XGBoost models.

```
In [12]:
with open('pickle/family_encoder.pkl', 'wb') as file:
    pickle.dump(family_encoder, file)

with open('pickle/typeholiday_encoder.pkl', 'wb') as file:
    pickle.dump(typeholiday_encoder, file)

with open('pickle/city_encoder.pkl', 'wb') as file:
    pickle.dump(city_encoder, file)

with open('pickle/state_encoder.pkl', 'wb') as file:
    pickle.dump(state_encoder, file)

with open('pickle/typestores_encoder.pkl', 'wb') as file:
    pickle.dump(typestores_encoder, file)
```

Figure 3.3 Exporting all the encoders through Pickle

```
In [28]:
dump(model, 'joblib/M10.joblib')

Out[28]: ['joblib/M10.joblib']
```

Figure 3.4 Exporting the model using Joblib

To check whether the model was working after being deployed locally, a set of hard coded values (Figure 3.5) were given to the model. The values were in the form as they would be input in the actual deployment. The only attribute that was not hard coded was the family (category of the product). This deployment made predictions for all the listed products, but only for one specific (hard coded) date. Functionality to dynamically manipulate variables was added later.

```
data = {
    'store_nbr': 1,
    'family_encoded': "",
    'onpromotion': 0,
    'typeholiday_encoded': "Holiday",
    'dcoilwtico': 46.8,
    'city_encoded': "Quito",
    'state_encoded': "Pichincha",
    'typestores_encoded': "D",
    'cluster': 13,
    'day_of_week': 3,
    'day': 16,
    'month': 8,
    'year': 2017
}
```

Figure 3.5 Hard coded values for the model

The encoders that were exported earlier were imported on Flask using Pickle. Each encoder was used to transform the value of their respective attribute (Figure 3.6).

```
if request.method == 'POST':
    df['family_encoded'] = request.form.get('family_encoded')

with open('models/family_encoder.pkl', 'rb') as file:
    family_encoder = pickle.load(file)

with open('models/typeholiday_encoder.pkl', 'rb') as file:
    typeholiday_encoder = pickle.load(file)

with open('models/city_encoder.pkl', 'rb') as file:
    city_encoder = pickle.load(file)

with open('models/state_encoder.pkl', 'rb') as file:
    state_encoder = pickle.load(file)

with open('models/typestores_encoder.pkl', 'rb') as file:
    typestores_encoder = pickle.load(file)

df['family_encoded'] = family_encoder.transform([df['family_encoded'].iloc[0]])[0]
df['typeholiday_encoded'] = typeholiday_encoder.transform([df['typeholiday_encoded'].iloc[0]])[0]
df['city_encoded'] = city_encoder.transform([df['city_encoded'].iloc[0]])[0]
df['state_encoded'] = state_encoder.transform([df['state_encoded'].iloc[0]])[0]
df['typestores_encoded'] = typestores_encoder.transform([df['typestores_encoded'].iloc[0]])[0]
```

Figure 3.6 Importing encoders and encoding categorical attributes

Once all the attributes were in a form that can be fed to the model, the model was loaded using Joblib and given the feature vector X (Figure 3.7). The model produced a prediction which was then sent to the homepage where it was displayed (Figure 3.8).

```
features = [
    'store_nbr',
    'onpromotion',
    'dcoilwtico',
    'cluster',
    'day_of_week',
    'day',
    'month',
    'year',
    'family_encoded',
    'typeholiday_encoded',
    'city_encoded',
    'state_encoded',
    'typestores_encoded',
]

X = df[features]

model = load('models/M10.joblib')
```

Figure 3.7 Creating feature vector X and importing the model using Joblib

```
prediction = model.predict(X)

return render_template(
    'index.html',
    prediction = prediction
)
```

Figure 3.8 Generating predictions using the XGBoost model

Figure 3.9 and 3.10 show a simple interface, which was improved later. The interface shows a dropdown list of all the listed products. The user can select the family (product category) and the model generates predicted sales for that family for the specified (hardcoded) store and the specified (hard coded) date.

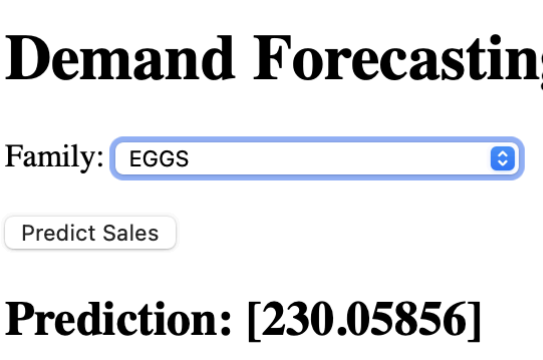


Figure 3.9 Prediction generated by the model for ‘eggs’ on the hardcoded store and date

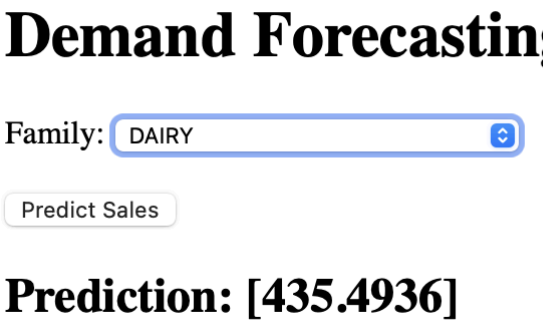


Figure 3.10 Prediction generated by the model for ‘dairy’ on the hardcoded store and date

For the demo, the users were given control over the selections dynamically, the data for holiday, oil pricing, and promotion was fetched in accordance to the date selected, and a side-by-side comparison was presented of the predicted sales and actual sales.

Bootstrap was used to improve the user experience visually and add functionality (Date Picker) as shown in Figure 3.11. The dates allowed to be selected are the dates from the test set. This was done so that we can show predicted values and actual values.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.9.0/js/bootstrap-datepicker.min.js"></script>
<script>
    $(document).ready(function(){
        $('.datepicker').datepicker({
            format: 'yyyy-mm-dd',
            startDate: '2016-09-14',
            endDate: '2017-08-14',
            autoclose: true
        });
    });
</script>
```

Figure 3.11 Date Picker

As discussed with Ma'am Huda, only 5 families (Beverages, Dairy, Frozen Foods, Meats, and Seafood) were selected for the demo (Figure 3.12).

```
<select class="form-control" id="family" name="family" required>
    <option value="BEVERAGES">BEVERAGES</option>
    <option value="DAIRY">DAIRY</option>
    <option value="FROZEN FOODS">FROZEN FOODS</option>
    <option value="MEATS">MEATS</option>
    <option value="SEAFOOD">SEAFOOD</option>
</select>
```

Figure 3.12 Shortlisted categories in the drop down list

To allow the user to select the date, the conversion from date to the extracted features we trained the model on had to be performed live (Figure 3.13).

```
if request.method == 'POST':
    df['date'] = request.form.get('datepicker')
    df['date'] = pd.to_datetime(df['date'])
    df['day_of_week'] = df['date'].dt.day_of_week
    df['day_of_week'] = df['day_of_week']+1
    df['day'] = df['date'].dt.day
    df['month'] = df['date'].dt.month
    df['year'] = df['date'].dt.year
```

Figure 3.13 Converting date into day_of_week, day, month, and year

In order to make it possible to fetch the holiday and oil price data for the selected date, a filtered dataset was created. The filter dataset only included dates from the test set, last 20% of the training data (Figure 3.14)

```
split_index = int(0.8 * train.shape[0])
```

```
filtered_train = train.iloc[split_index:]
```

```
filtered_train.shape
```

```
(600178, 10)
```

Figure 3.14 Filtering test dates

As discussed, the user was not allowed to select the store. So, for the sake of the demo, we selected store number 1 (Figure 3.15).

```
filtered_train = filtered_train[filtered_train['store_nbr'] == 1]
```

```
filtered_train.shape
```

```
(11088, 10)
```

```
filtered_train = filtered_train.drop(['store_nbr'], axis=1)
```

Figure 3.15 Filtering store number 1

The data was filtered for the shortlisted families (Figure 3.16) and inspected to ensure there were no issues.

```
filtered_train = filtered_train[(filtered_train['family'] == 'BEVERAGES') | (filtered_train['family'] == 'DAIRY')
```

Figure 3.16 Filtering shortlisted families

```
filtered_train.head()
```

	date	family	sales	onpromotion	typeholiday	dcoilwtico	day	month	year
2402139	2016-09-13	BEVERAGES	1942.000	0	NDay	44.91	13	9	2016
2402144	2016-09-13	DAIRY	703.000	0	NDay	44.91	13	9	2016
2402147	2016-09-13	FROZEN FOODS	95.000	0	NDay	44.91	13	9	2016
2402160	2016-09-13	MEATS	288.823	0	NDay	44.91	13	9	2016
2402168	2016-09-13	SEAFOOD	34.034	0	NDay	44.91	13	9	2016

```
filtered_train.tail()
```

	date	family	sales	onpromotion	typeholiday	dcoilwtico	day	month	year
2999109	2017-08-15	BEVERAGES	1942.000	11	Holiday	47.57	15	8	2017
2999114	2017-08-15	DAIRY	602.000	19	Holiday	47.57	15	8	2017
2999117	2017-08-15	FROZEN FOODS	89.000	1	Holiday	47.57	15	8	2017
2999130	2017-08-15	MEATS	274.176	0	Holiday	47.57	15	8	2017
2999138	2017-08-15	SEAFOOD	22.487	0	Holiday	47.57	15	8	2017

Figure 3.17 Inspecting the final dataset

In order to access the holiday, oil pricing, promotion, and actual sales data for the selected date and family, the filtered dataset is searched for the selected day, month, year, and family (Figure 3.18). As it would always return only one row, the data for holiday, oil pricing, and promotion is filled in the feature vector that is going to be used for the prediction. Actual sales are stored in a variable that is passed on to the html template (Figure 3.19).

```
filtered_df = pd.read_csv("data/filtered_train.csv")
filtered_df = filtered_df[(filtered_df['day'] == int(df['day'])) & (filtered_df['month'] == int(df['month'])) & (filtered_df['year'] == int(df['year']))]
filtered_df = filtered_df[(filtered_df['family'] == str(df['family'].iloc[0]))]
```

Figure 3.18 Filtering data based on day, month, year, and family

```
df['typeholiday'] = filtered_df['typeholiday'].iloc[0]
df['dcoilwtico'] = filtered_df['dcoilwtico'].iloc[0]
df['onpromotion'] = filtered_df['onpromotion'].iloc[0]
actual = round(filtered_df['sales'].iloc[0])
```

Figure 3.19 Collecting holiday, oil pricing, promotion, and actual sales data

The feature vector is passed on to the model as explained earlier and the prediction generated by the model along with the actual sales are sent to the home page where they are displayed (Figure 3.20).

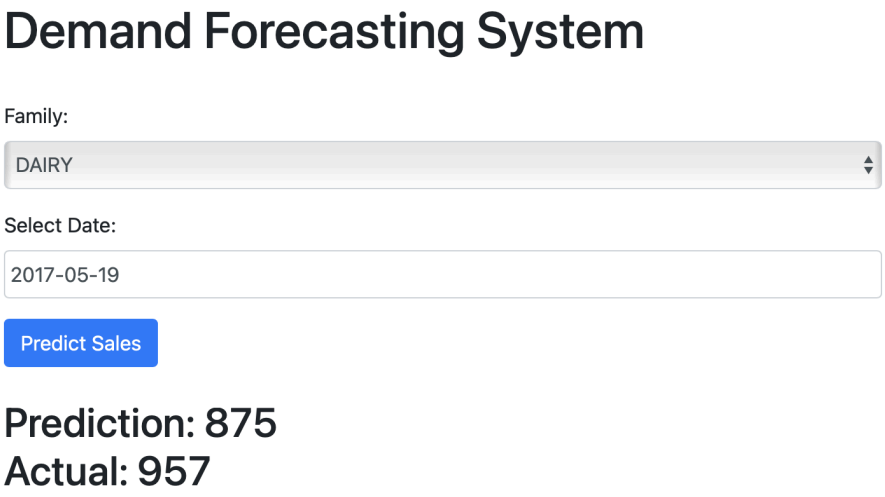


Figure 3.20 Flask Application

This concluded the development of the Flask application. The Flask application was then deployed on Digital Ocean (Figure 3.21). The demo of the ML model was given live on the cloud during the final presentation.

Deployment: https://hammerhead-app-6m6td.ondigitalocean.app/

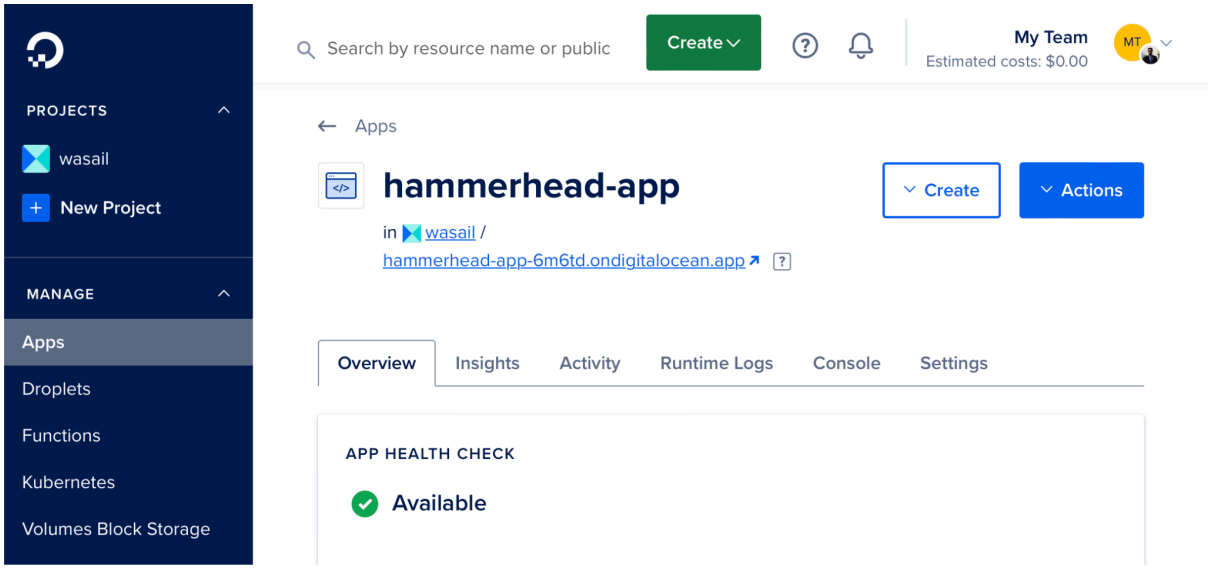


Figure 3.21 Deployment on Digital Ocean

Conclusion

In conclusion, this final year project report encapsulates the comprehensive process of developing our system, from initial requirement analysis through to design and implementation. We first started with looking into the existing systems by doing market research and reviewing literature which inspired us to create our mobile applications for both the vendor and the grocery stores. Then, gathering requirements helped lay a solid foundation for our stakeholders' needs, and by ensuring that every functional and nonfunctional requirement was clearly identified and documented, we established a clear roadmap for the development, aligning the project objectives with user expectations. The design section translated these requirements into a detailed blueprint, outlining the system architecture through C4 diagrams, going into details of the data design, delving into the specifics of the API design, and exploring the nuances of machine learning design. Finally, the implementation section brought the designs to life through rigorous coding, integration, and testing processes. Each component was developed with attention to detail, and extensive testing ensured that the system met all specified requirements and performed reliably under various conditions.

References

[1] L. X. Lu and J. M. Swaminathan, "Advances in Supply Chain Management," SSRN Electronic Journal, January 2013. DOI: 10.2139/ssrn.2758860.

[2] H. A. Khawaja, "What are the problems faced by the retail sector in Pakistan?" Online Blog Article, 2021.

[3] G. Ondiek, "Inventory management automation and performance of supermarkets in western Kenya," 2015.

[4] A. Sharma, "AI and Robotics for Reducing Waste in the Food Supply Chain: Systematic Literature Review, Theoretical Framework and Research Agenda," 2020.

[5] S. K. Panda and S. N. Mohanty, "Time Series Forecasting and Modeling of Food Demand Supply Chain Based on Regressors Analysis," 2023.

[6] Y. Liu, "Grocery Sales Forecasting," 2022.

[7] R. A. Carbonneau, R. Vahidov, K. Laframboise, "Machine Learning-Based Demand Forecasting in Supply Chains," 2007.

[8] S. Gull, I. Sarwar, Dr. W. Anwar, R. Rashid, "Smart eNose Food Waste Management System," 2021.

[9] J. Parfitt, M. Barthel, S. Macnaughton, "Food Waste within Food Supply Chains: Quantification and Potential for Change to 2050," 2010.

[10] L. Rieseegger, A. Hübner, "Reducing Food Waste at Retail Stores—An Explorative Study," 2022.

[11] A. M. Nascimento, F. S. Meirelles, "Applying Artificial Intelligence to Reduce Food Waste in Small Grocery Stores," 2022.

[12] M. Ulrich, H. Jahnke, R. Langrock, R. Pesch, R. Senge, "Distributional regression for demand forecasting in e-grocery," 2021.

[13] E. Gul, A. Lim, J. Xu, "Retail Store Layout Optimization for Maximum Product Visibility," 2021.

[14] N. Nassibi, H. Fasihuddin, L. Hsairi, "A Proposed Demand Forecasting Model by Using Machine Learning for the Food Industry," 2023.

[15] C.-H. Wang, Y.-W. Gu, "Sales Forecasting, Market Analysis, and Performance Assessment for US Retail Firms: A Business Analytics Perspective," 2022.

[16] V. Prabhakar, D. Sayiner, U. Chakraborty, T. Nguyen, M. A. Lanham, "Demand forecasting for a large grocery chain in Ecuador."

[17] J. H. Friedman, ‘‘Greedy function approximation: A gradient boosting machine,’’ Ann. Statist., vol. 29, no. 5, pp. 1189–1232, Oct. 2001.

[18] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, and T.-Y. Liu, ‘‘LightGBM: A highly efficient gradient boosting decision tree,’’ in Proc. Adv. Neural Inf. Process. Syst., vol. 30, 2017, pp. 3146–3154.

[19] T. Chen and C. Guestrin, ‘‘XGBoost: A scalable tree boosting system,’’ in Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, New York, NY, USA, Aug. 2016, pp. 785–794.

[20] A. V. Dorogush, V. Ershov, and A. Gulin, ‘‘CatBoost: Gradient boosting with categorical features support,’’ 2018, arXiv:1810.11363.

[21] S. Hochreiter and J. Schmidhuber, ‘‘Long short-term memory,’’ Neural Comput., vol. 9, no. 8, pp. 1735–1780, 1997

[22] H. Hewamalage, C. Bergmeir, and K. Bandara, ‘‘Recurrent neural networks for time series forecasting: Current status and future directions,’’ Int. J. Forecasting, vol. 37, no. 1, pp. 388–427, 2021.

[23] H. Xu and C. Y. Wang, ‘‘Demand prediction of chain supermarkets based on LSTM neural network,’’ China Logistics Purchasing, vol. 3, pp. 42–43, 2021

[24] J. Kim and N. Moon, ‘‘BiLSTM model based on multivariate time series data in multiple field for forecasting trading area,’’ J. Ambient Intell. Hum. Comput., pp. 1–10, Jul. 2019

[25] G. Kechyn, L. Yu, Y. Zang, S. Kechyn, "Sales Forecasting using WaveNet within the Framework of the Kaggle Competition," arXiv:1803.04037v1 [cs.LG], Mar. 11, 2018.

Appendices

Appendix A: Interview Transcripts

Z Mart

Q.1: How many of the items are imported?
A: Very less, out of 100 percent only 20 percent of the items are imported.

Q.2: What do you think can be solved if your work gets digitised?
A: It's easier to work using the software systems as they can scan items, 2 hours of work gets done in 15 minutes.

Q.3 Are there any issues you face with the software?
A: Not really. Nowadays, iPos software is being used in markets. If we encounter any issue, we call them and they fix it.

Q.4: Have you faced any issues with fake products?
A: Yes, Mobilers. Companies like Shangrila, Lipton, Supreme and Delmonte, their sales come from the company itself. For other companies, wholesalers in Shah Almi get their stuff from the companies in large quantities and get a discount. There are a few things in them that are fake, usually 'bubbles', 'tulsi', 'bombay' etc.

Q.5: How do you place your orders, do you have automatic systems or do you see for yourself if the items are less?
A: We observe how many items get sold in a day and for the next few days we buy the same amount.

Q.6: How does the order for the imported items get placed?
A: We get imported items for less quantities so they do not get wasted. There is 'Monthly Auditing' done too. If a certain item does not get sold before the expiry then we decrease the quantity for the next time. If the items have 3-4 months until their expiry, we sell them for discounted prices. Quantity of canned foods with a shelf life of 2-3 years is large therefore we put it on discount so it gets sold and fresh items get placed and for the next time we order a lesser quantity. We guess the quantity of items on a daily basis, our ordering is based on the sales of items from the previous day mostly.

Q.7: How does the ordering differ on special occasions?
A: On special occasions like eid, we buy more items than usual. For example, if we buy 4-5 breads each day, we would buy 20-25 breads on these occasions.

Q.8: How often do you update inventory?
A: Whatever comes from the company (vendor) gets added to the inventory first and expired items are removed from the inventory.

Q.9: Is there something that takes a lot of time to do that needs to be improved?

A: The iPos system does everything but sometimes if we forget to add something, something gets stolen, or 20 items are on the shelf but 24 are in the inventory, these types of issues do occur so we have to manually audit them.

Q.10: If you want to sell some new product, how do you find it?

A: The companies (vendors) ask us to display the new products and if it gets sold then they get paid. Otherwise, if the products do not get sold they take it back.

Express Store

Q.1: How do you run the grocery store and what significant issues do you confront?

A: People have expectations, therefore when imports ceased and demand for a product called stevia increased, we requested sellers (vendors) for it so we could continue to sell it. People once assumed that our store would carry items that were unavailable at other supermarkets, but they no longer do so because imports are either scarce or non-existent.

Q.2: How do you verify identification of personnel delivering your orders?

A: It's challenging to verify the authenticity because there have been, and still are, instances where individuals falsely claim to represent these organisations, selling stock with products nearing their expiry dates. Within the community, these individuals are commonly referred to as 'mobilers.'

Q.3: How do you manage imported goods?

A: Imported items have dealers here as well and they keep a follow-up as well. Nowadays there's an issue with the budget, since inflation the stock purchasing has decreased significantly, assuming that before 10 items were being purchased now just 3-4 items are being purchased. Then there are also a few things that get restricted by some stores just for themselves.

Q.4: Are there any issues regarding the current software that is being used in the store?

A: If electricity goes away then we are unable to use the software therefore we have to write down the purchases on a piece of paper which sometimes could get lost and then later we have to manually enter the data. There is no backup, for 8 hours there is no electricity, and sometimes if someone (buyer) is in a hurry then nothing gets entered into the system.