



BEACONHOUSE NATIONAL UNIVERSITY

PRJ-F23/333

WASAIL

DEMAND FORECASTING SYSTEM

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**

Project Title	Wasail: Demand Forecasting System	
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

Prof. Dr. Khawaja Shafaat Ahmed Bazaz
Dean School of Computer and IT

Date: _____

Huda Sarfraz
Internal Supervisor

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.

GitHub Repository



Scan the QR code or go to

<https://github.com/yrrebeere/prj-403>

Table of Contents

1 Introduction to the Project.....	29
1.1 Existing Systems.....	31
1.2 Literature review.....	37
2 Requirement Analysis.....	40
2.1 Requirement Gathering and Fact Finding.....	42
2.1.1 Data Collection Technique.....	42
2.1.2 Issues Identified.....	42
2.1.3 Interview Insights.....	43
2.1.4 Potential Improvements & Proposed Solution.....	44
2.2 System Environment.....	44
2.3 Machine Learning Requirements.....	45
2.3.1 Data Collection.....	45
2.3.1.1 Local Vendor Dataset.....	45
2.3.1.2 Local Pharmacy Dataset.....	47
2.3.1.3 Corporación Favorita Grocery Sales Forecasting.....	49
2.3.1.4 Instacart Market Basket Analysis.....	50
2.3.1.5 Other Datasets for Further Exploration.....	51
2.3.2 Feature Engineering.....	51
2.3.3 Model Shortlisting.....	52
2.3.4 Model Training and Testing.....	54
2.3.5 Integration with User Interface.....	55
2.3.6 Recommendation Updates.....	55
2.3.7 Feedback and Learning.....	56
2.4 User Roles.....	56
2.5 User Stories.....	56
2.5.1 User Story: Placing the Order.....	56
2.5.2 User Story: Searching for a Product.....	57
2.6 Functional Requirements.....	58
2.6.1 Grocery Store and Vendor (FR1).....	59
FR1.1: Language Selection.....	59
FR1.2: Phone Registration.....	60
FR1.3: Phone Number Confirmation.....	61
FR1.4: Phone Number Exists.....	62
FR1.5: OTP Code Generation and Delivery.....	63
FR1.6: Login.....	64
FR1.7: Logout.....	65
FR1.8: Reset Password.....	66
FR1.9: View Profile.....	67

2.6.2 Grocery Store (FR2).....	68
FR2.9: Account Details.....	68
FR2.10: Search Product.....	69
FR2.11: Search Category.....	70
FR2.12: Search Vendor.....	71
FR2.13: Browse Category.....	72
FR2.14: View Vendor Profile.....	73
FR2.15: Add Vendor to Vendor List.....	74
FR2.16: Contact Vendor.....	75
FR2.17: View Products on the Vendor's Profile.....	76
FR2.18: View Searched Product.....	77
FR2.19: Select Products.....	78
FR2.20: Order Recommendation.....	79
FR2.21: Quantity Selection.....	80
FR2.22: Add to Cart.....	81
FR2.23: View Cart.....	82
FR2.24: Update Product Quantity in Cart.....	83
FR2.25: Remove Product from Cart.....	84
FR2.26: Clear Cart.....	85
FR2.27: Order Placement.....	86
FR2.28: View Orders.....	87
FR2.29: Order Tracking.....	88
FR2.30: View Vendor List.....	89
FR2.31: View Order History.....	90
FR2.32: Edit Profile.....	91
2.6.3 Vendor (FR3).....	92
FR3.9: Vendor Registration.....	92
FR3.10: Valid Password.....	93
FR3.11: Username Exists.....	94
FR3.12: Search Product in Inventory.....	95
FR3.13: Add Product to Inventory.....	96
FR3.14: All Products Search.....	97
FR3.15: Restrict Product Duplication.....	98
FR3.16: Remove Product.....	99
FR3.17: Edit Details of Product.....	100
FR3.18: View Inventory.....	101
FR3.19: View Current Orders.....	102
FR3.20: View Grocery Stores List.....	103
FR3.21: View Grocery Store Profile.....	104
FR3.22: View Grocery Store Current Order.....	105
FR3.23: View Orders History.....	106

FR3.24: Order Dispatch Tracking.....	107
FR3.25: Edit Profile.....	108
2.6.4 Admin Portal (FR4).....	109
FR4.1: Login.....	109
FR4.2: Add New User.....	110
FR4.3: Add New User Details.....	111
FR4.4: Edit User Profile.....	112
FR4.5: Delete User Profile.....	113
FR4.6: Search User.....	114
FR4.7: View Grocery Store's Profile.....	115
FR4.8: Disable Grocery Store's Profile.....	116
FR4.9: Search Grocery Store.....	117
FR4.10: View Vendor's Profile.....	118
FR4.11: Disable Vendor's Profile.....	119
FR4.12: Search Vendor.....	120
FR4.13: Select Grocery Store's ML Model.....	121
FR4.14: Display Analytics.....	122
FR4.15: View Grocery Store Count.....	123
FR4.16: View Vendor Count.....	124
FR4.17: View Category Count.....	125
FR4.18: View Product Count.....	126
FR4.19: Add Category.....	127
FR4.20: Update Category.....	128
FR4.21: Delete Category.....	129
FR4.22: Search Category.....	130
FR4.23: Add Product.....	131
FR4.24: Update Product.....	132
FR4.25: Delete Product.....	133
FR4.26: Search Product.....	134
2.7 Non-Functional Requirements.....	135
2.7.1 Performance.....	135
2.7.2 Security.....	135
2.7.3 Usability.....	135
2.7.4 Data Storage.....	135
2.7.5 Error Handling.....	135
2.7.6 Mobile Responsiveness.....	135
2.8 Future Improvements.....	135
3 Design.....	136
3.1 Development Tools.....	138
3.1.1 Programming Languages.....	138
3.1.2 Machine Learning Libraries.....	138

3.1.3 Web Development.....	138
3.1.4 Data Storage.....	138
3.1.5 Object Relational Mapper.....	138
3.1.6 Cloud Services.....	138
3.1.7 Mobile App Development.....	138
3.1.8 Version Control.....	139
3.1.9 Project Management.....	139
3.1.10 Collaboration.....	139
3.2 System Architecture.....	139
3.2.1 System Context Diagram.....	139
3.2.2 Container Diagram.....	140
3.2.3 Component Diagram.....	141
3.2.4 Class Diagram.....	142
3.3 Data Design.....	143
3.4 Sequence Diagrams.....	145
FR 2.19 Order Recommendation.....	145
FR 2.24 Order Tracking.....	147
FR 3.14 All Products Search.....	148
FR 1.4 OTP Code Generation and Delivery.....	149
FR 2.14 Add Vendor to Vendor List.....	150
3.5 API Design.....	151
3.6 Machine Learning Design.....	153
3.6.1 Introduction.....	153
3.6.2 Feature Engineering.....	153
3.6.2.1 Local Pharmacy Dataset.....	153
3.6.2.2 Corporación Favorita Grocery Sales Forecasting.....	155
3.6.3 Model Shortlisting.....	160
3.6.3.1 Random Forest.....	160
3.6.3.2 XGBoost.....	160
3.6.3.3 Prophet.....	160
3.6.3.4 Recurrent Neural Networks.....	161
3.6.3.5 LightGBM.....	161
3.6.3.6 N-BEATS.....	161
3.6.3.7 DeepAR.....	161
3.6.4 Neural Network Architecture.....	161
3.7 Deployment Design.....	162
3.7.1 Machine Learning.....	162
4 Implementation.....	164
4.1 App Development.....	166
4.1.1 Front End.....	166
4.1.1.1 Vendor App.....	166

4.1.1.1.1 Phase 1: Initial Front-End Design.....	166
4.1.1.1.2 Phase 2: App Development Progress.....	175
4.1.1.1.3 Phase 3: Front-End and Back-End Integration.....	181
4.1.1.1.4 Phase 4: Final Front-End Design.....	183
4.1.1.2 Grocery Store App.....	191
4.1.1.2.1 Phase 1: Initial Front-End Design.....	191
4.1.1.2.2 Phase 2: App Development Progress.....	196
4.1.1.2.3 Phase 3: Front-End and Back-End Integration.....	199
4.1.1.2.4 Phase 4: Final Front-End Design.....	202
4.1.1.3 Admin Portal.....	211
4.1.1.3.1 Phase 1: Initial Front-End Design.....	211
4.1.1.3.2 Phase 2: App Development Progress.....	215
4.1.1.3.3 Phase 3: Front-End and Back-End Integration.....	219
4.1.1.3.4 Phase 4: Final Front-End Design.....	222
4.1.2 Back End.....	231
4.1.2.1 Database Design.....	231
4.1.2.2 Spring Boot.....	234
4.1.2.3 Installation of Sequelize.....	234
4.1.2.4 User Model Implementation.....	236
4.1.2.5 Implementation of ENV.....	237
4.1.2.6 Vendor Model Implementation.....	239
4.1.2.7 Implementation of Controller.....	240
4.1.2.8 Implementation of Relations.....	243
4.1.2.9 Implementation of Routes.....	245
4.1.2.10 Implementation of Routes in App.....	245
4.1.2.11 Postman Testing.....	246
4.1.2.12 Implementation of Grocery Store Application.....	249
4.1.2.13 Implementation of Admin Portal.....	277
4.1.2.14 Deployment.....	286
4.1.2.15 Conclusion.....	290
4.1.3 Test Cases.....	291
4.1.3.1 Vendor App.....	291
Test Case 1: Language Selection.....	291
Test Case 2: Phone Registration.....	291
Test Case 3: Phone Number Confirmation.....	292
Test Case 4: Phone Number Exists.....	292
Test Case 5: OTP Code Generation and Delivery.....	292
Test Case 6: Login.....	293
Test Case 7: Logout.....	293
Test Case 8: Reset Password.....	293
Test Case 9: View Profile.....	294

Test Case 10: Vendor Registration.....	294
Test Case 11: Valid Password.....	294
Test Case 12: Username Exists.....	295
Test Case 13: Search Product in Inventory.....	295
Test Case 14: Add Product to Inventory.....	295
Test Case 15: All Product Search.....	296
Test Case 16: Restrict Product Duplication.....	296
Test Case 17: Remove Product.....	296
Test Case 18: Edit Details of Product.....	297
Test Case 19: View Inventory.....	298
Test Case 20: View Current Orders.....	298
Test Case 21: View Grocery Store List.....	298
Test Case 22: View Grocery Store Profile.....	299
Test Case 23: View Grocery Store Current Order.....	299
Test Case 24: View Order History.....	299
Test Case 25: Order Dispatch Tracking.....	300
Test Case 26: Edit Profile.....	300
4.1.3.2 Grocery Store App.....	301
Test Case 27: Language Selection.....	301
Test Case 28: Phone Registration.....	301
Test Case 29: Phone Number Confirmation.....	301
Test Case 30: Phone Number Exists.....	302
Test Case 31: OTP Code Generation and Delivery.....	302
Test Case 32: Login.....	302
Test Case 33: Logout.....	303
Test Case 34: Reset Password.....	303
Test Case 35: View Profile.....	303
Test Case 36: Account Details.....	304
Test Case 37: Valid Password.....	304
Test Case 38: Username Exists.....	304
Test Case 39: Search Product.....	305
Test Case 40: Search Category.....	305
Test Case 41: Search Vendor.....	306
Test Case 42: Browse Category.....	306
Test Case 43: View Vendor Profile.....	306
Test Case 44: Add Vendor to Vendor List.....	307
Test Case 45: Contact Vendor.....	307
Test Case 46: View Products on the Vendor's Profile.....	307
Test Case 47: View Searched Product.....	307
Test Case 48: Select Products.....	308
Test Case 49: Order Recommendation.....	308

Test Case 50: Quantity Selection.....	308
Test Case 51: Add to Cart.....	309
Test Case 52: View Cart.....	309
Test Case 53: Update Product in Cart.....	309
Test Case 54: Remove Product from Cart.....	310
Test Case 55: Clear Cart.....	310
Test Case 56: Order Placement.....	310
Test Case 57: View Orders.....	311
Test Case 58: Order tracking.....	311
Test Case 59: View Vendor List.....	311
Test Case 60: View Order History.....	311
Test Case 61: Edit Profile.....	312
4.1.3.3 Admin Portal.....	312
Test Case 62: Add New Admin.....	312
Test Case 63: Add New Admin Details.....	312
Test Case 64: Edit Admin Profile.....	313
Test Case 65: Delete Admin Profile.....	313
Test Case 66: Search Admin.....	313
Test Case 67: View Grocery Store Profile.....	314
Test Case 68: Disable Grocery Store Profile.....	314
Test Case 69: Search Grocery Store.....	314
Test Case 70: View Vendor Profile.....	315
Test Case 71: Disable Vendor Profile.....	315
Test Case 72: Search Vendor.....	315
Test Case 73: Select Grocery Store's ML Model.....	315
Test Case 74: Display Analytics.....	316
Test Case 75: View Grocery Store Count.....	316
Test Case 76: View Vendor Count.....	316
Test Case 77: View Category Count.....	317
Test Case 78: View Product Count.....	317
Test Case 79: Add Category.....	317
Test Case 80: Update Category.....	317
Test Case 81: Delete Category.....	318
Test Case 82: Search Category.....	318
Test Case 83: Add Product.....	318
Test Case 84: Update Product.....	318
Test Case 85: Delete Product.....	319
Test Case 86: Search Product.....	319
4.1.4 Test Case Grid.....	320
4.1.4.1 Vendor App.....	320
4.1.4.2 Grocery Store App.....	322

4.1.4.3 Admin Portal.....	324
4.2 Machine Learning.....	326
4.2.1 Training and Testing.....	326
4.2.1.1 Local Pharmacy Dataset.....	326
4.2.1.1.1 Random Forest.....	326
4.2.1.1.2 XGBoost.....	331
4.2.1.1.3 Prophet.....	332
4.2.1.1.4 Recurrent Neural Networks.....	336
4.2.1.1.5 Summary.....	338
4.2.1.2 Corporación Favorita Grocery Sales Forecasting.....	339
4.2.1.2.1 Random Forest.....	343
4.2.1.2.2 XGBoost.....	348
4.2.1.2.3 Prophet.....	349
4.2.1.2.4 Recurrent Neural Networks.....	351
4.2.1.2.5 LightGBM.....	355
4.2.1.2.6 Summary.....	359
4.2.1.3 Darts.....	360
4.2.1.3.1 Random Forest.....	361
4.2.1.3.2 XGBoost.....	370
4.2.1.3.3 Prophet.....	376
4.2.1.3.4 Recurrent Neural Networks.....	380
4.2.1.3.5 LightGBM.....	388
4.2.1.3.6 N-BEATS.....	391
4.2.1.3.7 Ensemble Learning.....	396
4.2.2 Deployment.....	398
5 Conclusion.....	406
6 References.....	408
7 Appendices.....	410
Appendix A: Z Mart Interview Transcript.....	410
Appendix B: Express Store Interview Transcript.....	411
Appendix C: Setting up the AI Server.....	412
Appendix D: Demo Videos.....	416
Vendor App Demo.....	416
Store App Demo.....	417
Admin Portal Demo.....	418
Appendix E: Posters.....	419
Appendix F: Testing and Quality Assurance.....	420

Table of Figures

Figure 1.1.1 Tajir Mobile Application.....	32
Figure 1.1.2 Jugnu Mobile Application.....	33
Figure 1.1.3 Retailo Mobile Application.....	34
Figure 1.1.4 Dastgyr Mobile Application.....	35
Figure 1.1.5 Bazaar Mobile Application.....	36
Figure 1.2.1 Shelf Engine Mobile Application.....	38
Figure 1.2.2 Guac Web Application.....	39
Figure 2.1.2 Jalal Son's (taken on 5th November 2023).....	43
Figure 2.2 System Environment.....	44
Figure 2.3.1.1 Local Vendor Dataset.....	46
Figure 2.3.1.2 Local Pharmacy Dataset.....	48
Figure 2.3.5 ML Integration with the User Interface.....	55
Figure 2.5.1 User Story: Placing the Order.....	57
Figure 2.5.2 User Story: Searching for a Product.....	58
Figure 2.6.1.1 Language Selection Prototype.....	59
Figure 2.6.1.2 Phone Registration Prototype.....	60
Figure 2.6.1.3 Phone Number Confirmation Prototype.....	61
Figure 2.6.1.4 Phone Number Exists Prototype.....	62
Figure 2.6.1.5 OTP Code Generation and Delivery Prototype.....	63
Figure 2.6.1.6 Login Prototype.....	64
Figure 2.6.1.7 Logout Prototype.....	65
Figure 2.6.1.8 Reset Password Prototype.....	66
Figure 2.6.1.9 View Profile Prototype.....	67
Figure 2.6.2.1 Account Details Prototype.....	68
Figure 2.6.2.2 Search Product Prototype.....	69
Figure 2.6.2.3 Search Category Prototype.....	70
Figure 2.6.2.4 Search Vendor Prototype.....	71
Figure 2.6.2.5 Browse Category Prototype.....	72
Figure 2.6.2.6 View Vendor Profile Prototype.....	73
Figure 2.6.2.7 Add Vendor to Vendor List Prototype.....	74
Figure 2.6.2.8 Contact Vendor Prototype.....	75
Figure 2.6.2.9 View Products on the Vendor's Profile Prototype.....	76
Figure 2.6.2.10 View Searched Product Prototype.....	77
Figure 2.6.2.11 Select Products Prototype.....	78
Figure 2.6.2.12 Order Recommendation Prototype.....	79
Figure 2.6.2.13 Quantity Selection Prototype.....	80
Figure 2.6.2.14 Add to Cart Prototype.....	81
Figure 2.6.2.15 View Cart Prototype.....	82

Figure 2.6.2.16 Update Product Quantity in Cart Prototype.....	83
Figure 2.6.2.17 Remove Product Prototype.....	84
Figure 2.6.2.18 Clear Cart Prototype.....	85
Figure 2.6.2.19 Order Placement Prototype.....	86
Figure 2.6.2.20 View Order Prototype.....	87
Figure 2.6.2.21 Order Tracking Prototype.....	88
Figure 2.6.2.22 View Vendor List Prototype.....	89
Figure 2.6.2.23 View Order History Prototype.....	90
Figure 2.6.2.24 Edit Profile Prototype.....	91
Figure 2.6.3.1 Vendor Registration Prototype.....	92
Figure 2.6.3.2 Valid Password Prototype.....	93
Figure 2.6.3.3 Username Exists Prototype.....	94
Figure 2.6.3.4 Search Product in Inventory Prototype.....	95
Figure 2.6.3.5 Add Product to Inventory Prototype.....	96
Figure 2.6.3.6 All Products Search Prototype.....	97
Figure 2.6.3.7 Restrict Product Duplication Prototype.....	98
Figure 2.6.3.8 Remove Product Prototype.....	99
Figure 2.6.3.9 Edit Details of Product Prototype.....	100
Figure 2.6.3.10 View Inventory Prototype.....	101
Figure 2.6.3.11 View Current Orders Prototype.....	102
Figure 2.6.3.12 View Grocery Stores List Prototype.....	103
Figure 2.6.3.13 View Grocery Store Profile Prototype.....	104
Figure 2.6.3.14 View Grocery Store Current Order Prototype.....	105
Figure 2.6.3.15 View Orders History Prototype.....	106
Figure 2.6.3.16 Order Dispatch Tracking Prototype.....	107
Figure 2.6.3.17 Edit Profile Prototype.....	108
Figure 2.6.4.1 User Login Prototype.....	109
Figure 2.6.4.2 Add New User Prototype.....	110
Figure 2.6.4.3 Add New User Details Prototype.....	111
Figure 2.6.4.4 Edit User Profile Prototype.....	112
Figure 2.6.4.5 Delete User Profile Prototype.....	113
Figure 2.6.4.6 Search User Prototype.....	114
Figure 2.6.4.7 View Grocery Store's Profile Prototype.....	115
Figure 2.6.4.8 Disable Grocery Store's Profile Prototype.....	116
Figure 2.6.4.9 Search Grocery Store Prototype.....	117
Figure 2.6.4.10 View Vendor's Profile Prototype.....	118
Figure 2.6.4.11 Disable Vendor's Profile Prototype.....	119
Figure 2.6.4.12 Search Vendor Prototype.....	120
Figure 2.6.4.13 Select Grocery Store's ML Model Prototype.....	121
Figure 2.6.4.14 Display Analytics Prototype.....	122
Figure 2.6.4.15 View Grocery Store Count Prototype.....	123

Figure 2.6.4.16 View Vendor Count Prototype.....	124
Figure 2.6.4.17 View Category Count Prototype.....	125
Figure 2.6.4.18 View Product Count Prototype.....	126
Figure 2.6.4.19 Add Category Prototype.....	127
Figure 2.6.4.20 Update Category Prototype.....	128
Figure 2.6.4.21 Delete Category Prototype.....	129
Figure 2.6.4.22 Search Category Prototype.....	130
Figure 2.6.4.23 Add Product Prototype.....	131
Figure 2.6.4.24 Update Product Prototype.....	132
Figure 2.6.4.25 Delete Product Prototype.....	133
Figure 2.6.4.26 Search Product Prototype.....	134
Figure 3.1.1 System Context Diagram.....	139
Figure 3.1.2 Container Diagram.....	140
Figure 3.1.3 Component Diagram.....	141
Figure 3.1.4 Class Diagram.....	142
Figure 3.2.1 Data Design.....	144
Figure 3.3.1 Order Recommendation Sequence Diagram.....	146
Figure 3.3.2 Order Tracking Sequence Diagram.....	147
Figure 3.3.3 All Products Search Sequence Diagram.....	148
Figure 3.3.4 OTP Code Generation and Delivery Sequence Diagram.....	149
Figure 3.3.5 Add Vendor to Vendor List Sequence Diagram.....	150
Figure 3.5 API Design.....	152
Figure 3.5.1.1 Combining monthly csv files into one dataset.....	153
Figure 3.5.1.2 Inspecting the dataset.....	154
Figure 3.5.1.3 Dataset after removing the aforementioned columns.....	154
Figure 3.5.1.4 Date is split in date and time.....	154
Figure 3.5.1.5 Aggregating daily sales data for each item.....	155
Figure 3.5.1.6 Filtering Panadol Tab rows.....	155
Figure 3.5.2.1 Holiday Events.....	156
Figure 3.5.2.2 Oil Prices.....	156
Figure 3.5.2.3 Store Information.....	156
Figure 3.5.2.4 Converting date feature to datetime data type.....	156
Figure 3.5.2.5 Visualising time series data.....	157
Figure 3.5.2.6 Joining holiday_events, oil, and stores.....	157
Figure 3.5.2.7 Training with holiday_events, oil, and stores features.....	157
Figure 3.5.2.8 Extracting weekday, month, and year.....	158
Figure 3.5.2.9 Changing transferred holidays to normal days.....	158
Figure 3.5.2.10 Missing values in oil prices.....	159
Figure 3.5.2.11 Fixing missing values in oil prices.....	159
Figure 3.5.3 Shelf Engine: Demand Forecasting Models.....	160
Figure 3.6.4 Neural network architecture drawn using NN SVG.....	162

Figure 3.7.1 Model Deployment Diagram.....	163
Figure 4.1.1.1.1.1 Phone Number, Confirmation, OTP.....	167
Figure 4.1.1.1.1.2 Registration.....	168
Figure 4.1.1.1.1.3 Login, Home, Order History, Search.....	169
Figure 4.1.1.1.1.4 Current Orders.....	170
Figure 4.1.1.1.1.5 Inventory.....	172
Figure 4.1.1.1.1.6 Menu.....	174
Figure 4.1.1.1.2.1 Gesture Detector usage for accessing Order History in the Home Page.....	175
Figure 4.1.1.1.2.2 Navigation Bar using a Stateful Widget to handle Navigation.....	176
Figure 4.1.1.1.2.3 English and Urdu .arb file snippets.....	176
Figure 4.1.1.1.2.4 App context carrying the language selected and using its respective locale.....	176
Figure 4.1.1.1.2.5 Declarations of localisation needed to internationalise a page.....	177
Figure 4.1.1.1.2.6 Overview of the app in Urdu.....	178
Figure 4.1.1.1.2.7 Language locale provided by the provider.....	178
Figure 4.1.1.1.2.8 Provider used for various things across the app other than locale.....	179
Figure 4.1.1.1.2.9 Login Page in the early stage being unresponsive on a different device.....	180
Figure 4.1.1.1.2.10 MediaQuery Implementation.....	181
Figure 4.1.1.1.3.1 Basic User CRUD Implementation.....	182
Figure 4.1.1.1.3.2 Imports were used for Integration.....	182
Figure 4.1.1.1.3.3 GET operation on the Product table for Inventory Management.....	182
Figure 4.1.1.1.3.4 JSON decoding and encoding.....	183
Figure 4.1.1.1.4.1 English and Urdu.....	184
Figure 4.1.1.1.4.2 Login and Confirmation.....	184
Figure 4.1.1.1.4.3 OTP and Registration.....	185
Figure 4.1.1.1.4.4 Error Message.....	186
Figure 4.1.1.1.4.5 Home Page, Current Orders, and Menu.....	187
Figure 4.1.1.1.4.6 Add Product, Edit Product, and Store List.....	188
Figure 4.1.1.1.4.7 Inventory, Search Inventory, and Search Product.....	189
Figure 4.1.1.1.4.8 Order History, Current Orders, Edit Profile, and Edit Number.....	190
Figure 4.1.1.2.1.1 Search Product.....	191
Figure 4.1.1.2.1.2 Search Results.....	192
Figure 4.1.1.2.1.3 Current Orders & Order History Sections.....	192
Figure 4.1.1.2.1.4 Current Orders.....	193
Figure 4.1.1.2.1.5 Order History.....	193
Figure 4.1.1.2.1.6 Languages, Phone Number, OTP.....	194
Figure 4.1.1.2.1.7 Login, Home, Orders.....	195
Figure 4.1.1.2.1.8 Results, Vendor Details, Current Orders.....	195
Figure 4.1.1.2.1.9 Home Page.....	196
Figure 4.1.1.2.2.1 Implementation of the SKU tiles.....	197
Figure 4.1.1.2.2.2 Implementation of the text fields in Update Page.....	198
Figure 4.1.1.2.2.3 Implementation of the text fields for input in the Add To Inventory Page.....	199

Figure 4.1.1.2.3.1 HTTP Function Vendor Profile.....	200
Figure 4.1.1.2.3.2 HTTP Function for Adding Vendor.....	200
Figure 4.1.1.2.3.3 HTTP Request for Current Orders.....	201
Figure 4.1.1.2.3.4 HTTP Request for Order History.....	202
Figure 4.1.1.2.4.1 Languages, Phone Number, Confirmation.....	203
Figure 4.1.1.2.4.2 OTP, Registration.....	203
Figure 4.1.1.2.4.3 Home, Search Results.....	204
Figure 4.1.1.2.4.4 Vendor Details.....	205
Figure 4.1.1.2.4.5 Vendor Details.....	206
Figure 4.1.1.2.4.6 Current Orders.....	207
Figure 4.1.1.2.4.7 Order History.....	208
Figure 4.1.1.2.4.8 Cart.....	209
Figure 4.1.1.2.4.9 Menu, Profile, Edit.....	210
Figure 4.1.1.2.4.10 Cart.....	211
Figure 4.1.1.3.1.1 User Management.....	212
Figure 4.1.1.3.1.2 Content Management (List Category).....	213
Figure 4.1.1.3.1.3 Content Management (Add Category).....	213
Figure 4.1.1.3.1.4 Content Management (List Product).....	214
Figure 4.1.1.3.1.5 Content Management (Add Product).....	214
Figure 4.1.1.3.1.6 Vendor Listing.....	214
Figure 4.1.1.3.1.7 Vendor Addition.....	215
Figure 4.1.1.3.1.8 Grocery Store Listing.....	215
Figure 4.1.1.3.1.9 Grocery Store Addition.....	215
Figure 4.1.1.3.2.1 Add,Delete,View.....	216
Figure 4.1.1.3.2.2 Create Category Function.....	216
Figure 4.1.1.3.2.3 Create Product Function.....	217
Figure 4.1.1.3.2.4 Vendor Addition Code Snippet.....	218
Figure 4.1.1.3.2.5 Grocery Store Addition Code Snippet.....	219
Figure 4.1.1.3.3.1 Fetching Categories.....	220
Figure 4.1.1.3.3.2 Fetching Products.....	220
Figure 4.1.1.3.3.3 Vendor Listing Code Snippet.....	221
Figure 4.1.1.3.3.4 Grocery Listing Snippet.....	222
Figure 4.1.1.3.4.1 Admin Management.....	223
Figure 4.1.1.3.4.2 Add New Admin.....	223
Figure 4.1.1.3.4.3 Update Admin.....	224
Figure 4.1.1.3.4.4 Search Admin.....	224
Figure 4.1.1.3.4.5 Store Management.....	225
Figure 4.1.1.3.4.6 Update Store.....	225
Figure 4.1.1.3.4.7 Update Store.....	226
Figure 4.1.1.3.4.8 Vendor Management.....	226
Figure 4.1.1.3.4.9 Search Vendor.....	227

Figure 4.1.1.3.4.10 Analytics.....	227
Figure 4.1.1.3.4.11 Category Management.....	228
Figure 4.1.1.3.4.12 Add Category.....	228
Figure 4.1.1.3.4.13 Update Category.....	229
Figure 4.1.1.3.4.14 Search Category.....	229
Figure 4.1.1.3.4.15 Product Management.....	230
Figure 4.1.1.3.4.16 Add Product.....	230
Figure 4.1.1.3.4.17 Update Product.....	230
Figure 4.1.1.3.4.18 Search Product.....	231
Figure 4.1.2.1.1 First iteration of the database model.....	232
Figure 4.1.2.1.2 Last iteration of database model.....	233
Figure 4.1.2.2 Implementation using Spring Boot.....	234
Figure 4.1.2.3.1 Installation of Sequelize.....	234
Figure 4.1.2.3.2 Installation of MySQL2.....	235
Figure 4.1.2.3.3 Installation of sequelize-cli.....	235
Figure 4.1.2.3.4 Dependencies installed for the project.....	235
Figure 4.1.2.3.5 Empty files created by sequelize.....	236
Figure 4.1.2.4 User table model created.....	237
Figure 4.1.2.5.1 .env file.....	237
Figure 4.1.2.5.2 Installation of .env file.....	237
Figure 4.1.2.5.3 Config file.....	238
Figure 4.1.2.5.4 Error handling in index file.....	238
Figure 4.1.2.5.5 Sync function.....	239
Figure 4.1.2.6 Implementation of Vendor model.....	240
Figure 4.1.2.7.1 Implementation of User controller.....	241
Figure 4.1.2.7.2 Implementation of Product controller.....	243
Figure 4.1.2.8.1 Importing the models in the index file.....	244
Figure 4.1.2.8.2 One to one relation.....	244
Figure 4.1.2.8.3 One to many relation.....	244
Figure 4.1.2.8.4 Many to many relation.....	245
Figure 4.1.2.9.1 Implementation of user routes.....	245
Figure 4.1.2.10 Implementation of main routes.....	246
Figure 4.1.2.11.1 Adding a user.....	246
Figure 4.1.2.11.2 Searching a product.....	247
Figure 4.1.2.11.3 Current Orders.....	248
Figure 4.1.2.11.4 Order History.....	249
Figure 4.1.2.12.1 Store Registration.....	250
Figure 4.1.2.12.2 Relation between Grocery Store and User Table.....	250
Figure 4.1.2.12.3 End Point Implemented for Store Registration.....	251
Figure 4.1.2.12.4 Postman Testing.....	251
Figure 4.1.2.12.5 Search Product.....	252

Figure 4.1.2.12.6 Search Product.....	252
Figure 4.1.2.12.7 Search Product.....	253
Figure 4.1.2.12.8 Search Product.....	253
Figure 4.1.2.12.9 End Point Implemented for Search Product.....	253
Figure 4.1.2.12.10 View Vendor List.....	254
Figure 4.1.2.12.11 End Point Implemented for Vendor List.....	254
Figure 4.1.2.12.12 Postman Testing.....	255
Figure 4.1.2.12.13 Product Category Link Class.....	256
Figure 4.1.2.12.14 Search Category.....	257
Figure 4.1.2.12.15 Search Category.....	257
Figure 4.1.2.12.16 Search Category.....	258
Figure 4.1.2.12.17 Search Category.....	258
Figure 4.1.2.12.18 Search Category.....	259
Figure 4.1.2.12.19 Postman Testing.....	260
Figure 4.1.2.12.20 View Vendor Profile.....	260
Figure 4.1.2.12.21 View Vendor Profile.....	261
Figure 4.1.2.12.22 View Vendor Profile.....	261
Figure 4.1.2.12.23 End Point Implemented for Vendor Profile.....	261
Figure 4.1.2.12.24 Postman Testing.....	262
Figure 4.1.2.12.25 Relation between the tables.....	263
Figure 4.1.2.12.26 Order History.....	263
Figure 4.1.2.12.27 End Point Implemented for Order History.....	263
Figure 4.1.2.12.28 Postman Testing.....	264
Figure 4.1.2.12.29 Current Orders.....	265
Figure 4.1.2.12.30 End Point Implemented for Current Orders.....	265
Figure 4.1.2.12.31 Postman Testing.....	266
Figure 4.1.2.12.32 Order Placement.....	267
Figure 4.1.2.12.33 End Point Implemented for Order Placement.....	267
Figure 4.1.2.12.34 Postman Testing.....	268
Figure 4.1.2.12.35 Installation of Multer.....	268
Figure 4.1.2.12.36 Multer Package.....	268
Figure 4.1.2.12.37 Uploads Folder for Images.....	269
Figure 4.1.2.12.38 Implementation of Get Product Image.....	269
Figure 4.1.2.12.39 End Point Implemented for Product Images.....	270
Figure 4.1.2.12.40 Postman Testing.....	270
Figure 4.1.2.12.41 Implementation of Get Category Image.....	270
Figure 4.1.2.12.42 End Point Implemented for Get Category Image.....	271
Figure 4.1.2.12.43 Postman Testing.....	271
Figure 4.1.2.12.44 Implementation of Get Store Image.....	271
Figure 4.1.2.12.45 End Point Implemented for Get Store Image.....	272
Figure 4.1.2.12.46 Postman Testing.....	272

Figure 4.1.2.12.47 Implementation of Get Vendor Image.....	272
Figure 4.1.2.12.48 End Point Implemented for Get Vendor Image.....	272
Figure 4.1.2.12.49 Postman Testing.....	273
Figure 4.1.2.12.50 Upload Product.....	273
Figure 4.1.2.12.51 End Point Implemented for Upload Product.....	273
Figure 4.1.2.12.52 Installation of Axios.....	274
Figure 4.1.2.12.53 Axios Package.....	274
Figure 4.1.2.12.54 Implementation of Weekly Prediction.....	275
Figure 4.1.2.12.55 End Point Implemented for Weekly Prediction.....	275
Figure 4.1.2.12.56 Postman Testing.....	275
Figure 4.1.2.12.57 Implementation of Monthly Prediction.....	276
Figure 4.1.2.12.58 End Point Implemented for Weekly Prediction.....	276
Figure 4.1.2.12.59 Postman Testing.....	276
Figure 4.1.2.13.1 Admin Model.....	277
Figure 4.1.2.13.2 Add Admin.....	278
Figure 4.1.2.13.3 Get All Admins.....	278
Figure 4.1.2.13.4 Get One Admin.....	278
Figure 4.1.2.13.5 Update Admin.....	278
Figure 4.1.2.13.6 Delete Admin.....	279
Figure 4.1.2.13.7 End Point implemented for Admin.....	279
Figure 4.1.2.13.8 Relation between admin and user table.....	279
Figure 4.1.2.13.9 Implementation of Add Category.....	280
Figure 4.1.2.13.10 End Point Implemented for Add Category.....	280
Figure 4.1.2.13.11 Implementation of Update Category.....	280
Figure 4.1.2.13.12 End Point Implemented for Update Category.....	280
Figure 4.1.2.13.13 Postman Testing.....	281
Figure 4.1.2.13.14 Implementation of Delete Category.....	281
Figure 4.1.2.13.15 End Point Implemented for Delete Category.....	281
Figure 4.1.2.13.16 Postman Testing.....	282
Figure 4.1.2.13.17 Implementation of Add Product.....	282
Figure 4.1.2.13.18 End Point Implemented for Add Product.....	282
Figure 4.1.2.13.19 Implementation of Update Product.....	283
Figure 4.1.2.13.20 End Point Implemented for Update Product.....	283
Figure 4.1.2.13.21 Postman Testing.....	283
Figure 4.1.2.13.22 Implementation of Delete Product.....	283
Figure 4.1.2.13.23 End Point Implemented for Delete Product.....	284
Figure 4.1.2.13.24 Postman Testing.....	284
Figure 4.1.2.13.25 Implementation of View Grocery Store Profile.....	284
Figure 4.1.2.13.26 End Point Implemented for Grocery Store Profile.....	284
Figure 4.1.2.13.27 Postman Testing.....	285
Figure 4.1.2.13.28 Implementation of Vendor Profile.....	285

Figure 4.1.2.13.29 End Point Implemented for View Vendor.....	285
Figure 4.1.2.13.30 Postman Testing.....	286
Figure 4.1.2.14.1 Database Cluster on Digital Ocean.....	286
Figure 4.1.2.14.2 Connection details for the Wasail database.....	287
Figure 4.1.2.14.3 Connection timed out error.....	287
Figure 4.1.2.14.4 Adding my computer's IP in trusted sources.....	288
Figure 4.1.2.14.5 Wasail database on MySQL Workbench.....	288
Figure 4.1.2.14.6 Adding all IP addresses as trusted sources.....	288
Figure 4.1.2.14.7 Backend system deployed on Digital Ocean.....	289
Figure 4.1.2.14.8 Deployment updated on each commit.....	289
Figure 4.1.2.14.9 All deployments on Digital Ocean.....	289
Figure 4.1.2.14.10 Frontend fetching data from the deployed backend system.....	290
Figure 4.2.1.1 Pharmacy Dataset Update Glossary.....	326
Figure 4.2.1.1.1.1 Importing necessary libraries.....	327
Figure 4.2.1.1.1.2 Loading the dataset and inspecting it.....	327
Figure 4.2.1.1.1.3 Shape of the dataframe.....	327
Figure 4.2.1.1.1.4 Separating features and label.....	327
Figure 4.2.1.1.1.5 Converting categorical attributes and inspecting the shape of the dataframe.....	328
Figure 4.2.1.1.1.6 Splitting data into training set and testing set.....	328
Figure 4.2.1.1.1.7 Creating a RF model and training it on the training set.....	328
Figure 4.2.1.1.1.8 Google Colab crashing.....	329
Figure 4.2.1.1.1.9 D5.csv used instead of D4.csv.....	329
Figure 4.2.1.1.1.10 Shape of the dataframe.....	330
Figure 4.2.1.1.1.11 Making predictions on the test set and calculating RMSE and RMSLE.....	330
Figure 4.2.1.1.1.12 Printing the RMSE and RMSLE values.....	330
Figure 4.2.1.1.1.13 Converting y_test to a normal np array.....	330
Figure 4.2.1.1.1.14 Actual values vs predicted values.....	331
Figure 4.2.1.1.2.1 Training a XGBoost model.....	331
Figure 4.2.1.1.2.2 Printing the RMSE and RMSLE values.....	331
Figure 4.2.1.1.2.3 Actual values vs predicted values.....	332
Figure 4.2.1.1.3.1 Importing necessary libraries.....	332
Figure 4.2.1.1.3.2 Renaming date and looseqty columns for Prophet.....	332
Figure 4.2.1.1.3.3 Creating and training the Prophet model.....	333
Figure 4.2.1.1.3.4 Defining time period for predictions.....	333
Figure 4.2.1.1.3.5 Printing the RMSE and RMSLE values.....	333
Figure 4.2.1.1.3.6 Prophet model's plot.....	334
Figure 4.2.1.1.3.7 Unique products in the Pharmacy dataset.....	334
Figure 4.2.1.1.3.8 Finding the top 10 products.....	335
Figure 4.2.1.1.3.9 Training models for the top 10 products.....	335
Figure 4.2.1.1.3.10 Printing the RMSE and RMSLE values.....	336
Figure 4.2.1.1.4.1 Creating a univariate dataframe.....	336

Figure 4.2.1.1.4.2 Generating sequences for training the model.....	336
Figure 4.2.1.1.4.3 Creating and training the LSTM model.....	337
Figure 4.2.1.1.4.4 Printing the RMSE and RMSLE values.....	337
Figure 4.2.1.1.4.5 Actual values vs predicted values.....	337
Figure 4.2.1.1.4.6 Creating and training the GRU model.....	338
Figure 4.2.1.1.4.7 Printing the RMSE and RMSLE values.....	338
Figure 4.2.1.1.4.8 Actual values vs predicted values.....	338
Table 4.2.1.1.5 Model Summary.....	339
Figure 4.2.1.2.1.1 Kaggle crashing.....	339
Figure 4.2.1.2.1.2 Google Colab crashing.....	340
Figure 4.2.1.2.1.3 Combining train and test dataset to apply get_dummies together.....	341
Figure 4.2.1.2.1.4 Information about the updated train dataset.....	341
Figure 4.2.1.2.1.5 Batch Training.....	342
Figure 4.2.1.2.1.6 Generating predictions, fixing them, and creating a submission file.....	342
Figure 4.2.1.2.1.7 Store Sales - Time Series Forecasting Leaderboard.....	343
Figure 4.2.1.2.2.1 Loading the dataset and inspecting it.....	343
Figure 4.2.1.2.2.2 Converting categorical attributes using get_dummies.....	344
Figure 4.2.1.2.2.3 Setting batch size and calculating number of batches.....	344
Figure 4.2.1.2.2.4 Using batch training to train the RF model.....	344
Figure 4.2.1.2.2.5 Printing the RMSE and RMSLE values.....	345
Figure 4.2.1.2.2.6 Plotting 100 values from actual and predicted values.....	345
Figure 4.2.1.2.2.7 Actual values vs predicted values.....	345
Figure 4.2.1.2.2.8 Batch size.....	346
Figure 4.2.1.2.2.9 Total number of batches.....	346
Figure 4.2.1.2.2.10 Printing the RMSE and RMSLE values.....	346
Figure 4.2.1.2.2.11 Actual values vs predicted values.....	346
Figure 4.2.1.2.2.12 Batch size.....	347
Figure 4.2.1.2.2.13 Total number of the batches.....	347
Figure 4.2.1.2.2.14 Printing the RMSE and RMSLE values.....	347
Figure 4.2.1.2.2.15 Actual values vs predicted values.....	347
Figure 4.2.1.2.2.16 CPU usage by Random Forest.....	348
Figure 4.2.1.2.3.1 Creating a XGBoost model.....	348
Figure 4.2.1.2.3.2 Kernel died error.....	348
Figure 4.2.1.2.3.3 Batch size.....	348
Figure 4.2.1.2.3.4 Printing the RMSE and RMSLE values.....	348
Figure 4.2.1.2.3.5 Actual values vs predicted values.....	349
Figure 4.2.1.2.3.6 CPU usage by XGBoost.....	349
Figure 4.2.1.2.4.1 Only sales for beverages (family = 3) at store number 1.....	350
Figure 4.2.1.2.4.2 Printing the RMSE and RMSLE values.....	350
Figure 4.2.1.2.4.3 Prophet model's plot.....	351
Figure 4.2.1.2.5.1 Importing libraries and loading the dataset.....	351

Figure 4.2.1.2.5.2 Converting the data frame into training sequences.....	352
Figure 4.2.1.2.5.3 Splitting the data in training, validation, and test sets.....	352
Figure 4.2.1.2.5.4 Importing libraries and initialising our model.....	353
Figure 4.2.1.2.5.5 Compiling and fitting the model.....	353
Figure 4.2.1.2.5.6 Lowest RMSE.....	353
Figure 4.2.1.2.5.7 Test predictions vs actual values plotted.....	354
Figure 4.2.1.2.5.8 GRU (Univariate).....	354
Figure 4.2.1.2.5.9 LSTM (Multivariate).....	355
Figure 4.2.1.2.6.1 Loading the dataset and inspecting it.....	355
Figure 4.2.1.2.6.2 Inspecting training features (X) for categorical attributes.....	356
Figure 4.2.1.2.6.3 Converting categorical attributes to numerical values.....	356
Figure 4.2.1.2.6.4 Inspecting the updated dataset.....	357
Figure 4.2.1.2.6.5 Splitting data into training set and testing set and setting batch size.....	357
Figure 4.2.1.2.6.6 Batch training LGBMRegressor.....	358
Figure 4.2.1.2.6.7 Generating predictions and fixing them.....	358
Figure 4.2.1.2.6.8 Printing the RMSE and RMSLE values.....	358
Figure 4.2.1.2.6.9 Actual values vs predicted values.....	359
Table 4.2.1.2.6 Model Summary.....	359
Figure 4.2.1.3.1.1 Darts.....	360
Figure 4.2.1.3.1.2 Forecasting Models.....	360
Figure 4.2.1.3.1.3 Random Forest Example.....	361
Figure 4.2.1.3.2.1 Loading the dataset and inspecting it.....	362
Figure 4.2.1.3.2.2 Creating a TimeSeries and splitting data into training set and validation set.....	362
Figure 4.2.1.3.2.3 Instantiating and training the Random Forest model.....	362
Figure 4.2.1.3.2.4 Random Forest producing a ValueError when instantiated without lags.....	362
Figure 4.2.1.3.2.5 Generating predictions, converting series to lists, and fixing them.....	363
Figure 4.2.1.3.2.6 Printing the RMSE and RMSLE values.....	363
Figure 4.2.1.3.2.7 Actual values vs predicted values.....	364
Figure 4.2.1.3.2.8 Loading the dataset and inspecting it.....	364
Figure 4.2.1.3.2.9 Getting unique combinations of stores and families and declaring variables.....	365
Figure 4.2.1.3.2.10 Iterating over each combination of store and family and creating a subset of data.....	365
Figure 4.2.1.3.2.11 Creating a TimeSeries and splitting data into training set and validation set.....	365
Figure 4.2.1.3.2.12 Instantiating, training, and saving the Random Forest model(s).....	366
Figure 4.2.1.3.2.13 Generating predictions, converting series to lists, and fixing them.....	366
Figure 4.2.1.3.2.14 Calculating RMSE and RMSLE, adding them to calculate average later, incrementing number of models, and printing the RMSE and RMSLE values.....	366
Figure 4.2.1.3.2.15 Generating a plot of actual values vs predicted values and saving it locally.....	366
Figure 4.2.1.3.2.16 Printing the RMSE and RMSLE values of each model as it is trained.....	367
Figure 4.2.1.3.2.17 Printing the average RMSE and RMSLE values.....	367
Figure 4.2.1.3.2.18 Untrained Random Forest model due to no data.....	368
Figure 4.2.1.3.2.19 Under-fitted Random Forest model due to very little data.....	368

Figure 4.2.1.3.2.20 Well-fitted Random Forest model due to abundant data.....	369
Figure 4.2.1.3.2.21 Not counting models with RMSE=0.....	369
Figure 4.2.1.3.2.22 Calculating and printing average RMSE and RMSLE.....	369
Figure 4.2.1.3.2.23 CPU usage by Random Forest.....	370
Figure 4.2.1.3.3.1 Instantiating and training the XGBoost model.....	370
Figure 4.2.1.3.3.2 Printing the average RMSE and RMSLE values.....	370
Figure 4.2.1.3.3.3 Untrained XGBoost model due to no data.....	371
Figure 4.2.1.3.3.4 Under-fitted XGBoost model due to very little data.....	371
Figure 4.2.1.3.3.5 Well-fitted XGBoost model due to abundant data.....	372
Figure 4.2.1.3.3.6 CPU usage by XGBoost.....	372
Table 4.2.1.3.3.7 Random Forest vs XGBoost.....	372
Figure 4.2.1.3.3.8 Loading the dataset and inspecting it.....	373
Figure 4.2.1.3.3.9 Declaring dictionaries.....	373
Figure 4.2.1.3.3.10 Training models and generating predictions.....	373
Figure 4.2.1.3.3.11 Loading the dataset.....	374
Figure 4.2.1.3.3.12 Inspecting the dataset.....	374
Figure 4.2.1.3.3.13 Nested Loop Approach.....	374
Figure 4.2.1.3.3.14 Inspecting the dataset manually.....	375
Figure 4.2.1.3.3.15 Dictionary Iteration Approach.....	375
Figure 4.2.1.3.3.16 Creating the submission csv.....	375
Figure 4.2.1.3.3.17 Submissions to Store Sales (Kaggle).....	376
Figure 4.2.1.3.3.18 Store Sales Leaderboard.....	376
Figure 4.2.1.3.4.1 Loading the dataset, updating index, and inspecting it.....	377
Figure 4.2.1.3.4.2 Count of NaN values in each df subset.....	377
Figure 4.2.1.3.4.3 Converting NaN values to 0.....	378
Figure 4.2.1.3.4.4 Printing the average RMSE and RMSLE values.....	378
Table 4.2.1.3.4.5 Random Forest vs XGBoost vs Prophet.....	378
Figure 4.2.1.3.4.6 Untrained Prophet model due to no data.....	379
Figure 4.2.1.3.4.7 Under-fitted Prophet model due to very little data.....	379
Figure 4.2.1.3.4.8 Well-fitted Prophet model due to abundant data.....	380
Figure 4.2.1.3.4.9 CPU usage by Prophet.....	380
Figure 4.2.1.3.5.1 Inspecting the dataframe.....	381
Figure 4.2.1.3.5.2 Instantiating the RNN model(s).....	381
Figure 4.2.1.3.5.3 Converting the datatype of sales to float32.....	381
Figure 4.2.1.3.5.4 Training RNN model(s).....	382
Figure 4.2.1.3.5.5 Untrained RNN model due to no data.....	382
Figure 4.2.1.3.5.6 Under-fitted RNN model due to very little data.....	383
Figure 4.2.1.3.5.7 Under-fitted RNN model despite adequate amount of data.....	384
Figure 4.2.1.3.5.8 CPU usage by RNN.....	384
Figure 4.2.1.3.5.9 Instantiating the RNN model(s).....	384
Figure 4.2.1.3.5.10 Training RNN model(s).....	385

Figure 4.2.1.3.5.11 Average RMSE and RMSLE.....	385
Table 4.2.1.3.5.12 Random Forest vs XGBoost vs Prophet vs RNN.....	386
Figure 4.2.1.3.5.13 Number of epochs set to 50.....	386
Figure 4.2.1.3.5.14 Average RMSE and RMSLE.....	386
Figure 4.2.1.3.5.15 Under-fitted RNN model (n_epochs=10).....	387
Figure 4.2.1.3.5.16 Under-fitted RNN model (n_epochs=50).....	387
Figure 4.2.1.3.5.18 GPU usage by RNN.....	388
Figure 4.2.1.3.6.1 Instantiating and training the LightGBM model(s).....	388
Figure 4.2.1.3.6.2 Printing the RMSE and RMSLE values.....	388
Figure 4.2.1.3.6.3 Well-fitted LightGBM model.....	389
Table 4.2.1.3.6.4 Random Forest vs XGBoost vs Prophet vs RNN vs LightGBM.....	389
Figure 4.2.1.3.6.5 Instantiating and training the LightGBM model(s) with lags=7.....	389
Figure 4.2.1.3.6.6 Printing the RMSE and RMSLE values.....	390
Figure 4.2.1.3.6.7 Instantiating and training the LightGBM model(s) with lags=30.....	390
Figure 4.2.1.3.6.8 Printing the RMSE and RMSLE values.....	390
Table 4.2.1.3.6.9 Lag comparison in LightGBM.....	390
Figure 4.2.1.3.6.10 CPU usage by LightGBM.....	391
Figure 4.2.1.3.7.1 Precision of matrix multiplications set to ‘medium’.....	391
Figure 4.2.1.3.7.2 Instantiating the N-BEATS model(s) with n_epochs=1.....	392
Figure 4.2.1.3.7.3 Training the N-BEATS model(s) using the GPU.....	392
Figure 4.2.1.3.7.4 Printing the RMSE and RMSLE values.....	392
Figure 4.2.1.3.7.5 N-BEATS model showing an increasing pattern.....	393
Figure 4.2.1.3.7.6 N-BEATS model showing a decreasing pattern.....	393
Table 4.2.1.3.7.7 Random Forest vs XGBoost vs Prophet vs RNN vs LightGBM vs N-BEATS.....	394
Figure 4.2.1.3.7.8 Instantiating the N-BEATS model(s) with n_epochs=10.....	394
Figure 4.2.1.3.7.9 Printing the RMSE and RMSLE values.....	394
Figure 4.2.1.3.7.10 N-BEATS model showing a decreasing pattern.....	395
Figure 4.2.1.3.7.11 N-BEATS model showing a decreasing pattern.....	395
Figure 4.2.1.3.7.12 Memory usage by N-BEATS.....	396
Figure 4.2.1.3.8.1 Models merged for Ensemble Learning.....	396
Figure 4.2.1.3.8.2 Creating two Time Series.....	397
Figure 4.2.1.3.8.3 Instantiating the models.....	397
Figure 4.2.1.3.8.4 Saving the lowest RMSE and RMSLE amongst the four models.....	397
Figure 4.2.1.3.8.5 Printing the RMSE and RMSLE values.....	397
Table 4.2.1.3.8.6 Random Forest vs XGBoost vs Prophet vs LightGBM vs Ensemble.....	398
Figure 4.2.2.1 pip freeze > requirements.txt.....	398
Figure 4.2.2.2 Label Encoder.....	399
Figure 4.2.2.3 Exporting all the encoders through Pickle.....	399
Figure 4.2.2.4 Exporting the model using Joblib.....	399
Figure 4.2.2.5 Hard coded values for the model.....	400
Figure 4.2.2.6 Importing encoders and encoding categorical attributes.....	400

Figure 4.2.2.7 Creating feature vector X and importing the model using Joblib.....	401
Figure 4.2.2.8 Generating predictions using the XGBoost model.....	401
Figure 4.2.2.9 Prediction generated by the model for ‘eggs’ on the hardcoded store and date.....	401
Figure 4.2.2.10 Prediction generated by the model for ‘dairy’ on the hardcoded store and date.....	402
Figure 4.2.2.11 Date Picker.....	402
Figure 4.2.2.12 Shortlisted categories in the drop down list.....	402
Figure 4.2.2.13 Converting date into day_of_week, day, month, and year.....	403
Figure 4.2.2.14 Filtering test dates.....	403
Figure 4.2.2.15 Filtering store number 1.....	403
Figure 4.2.2.16 Filtering shortlisted families.....	404
Figure 4.2.2.17 Inspecting the final dataset.....	404
Figure 4.2.2.18 Filtering data based on day, month, year, and family.....	404
Figure 4.2.2.19 Collecting holiday, oil pricing, promotion, and actual sales data.....	404
Figure 4.2.2.20 Flask Application.....	405
Figure 4.2.2.21 Deployment on Digital Ocean.....	405
Figure 7.1 Adjusting the VPN settings on FortiClient.....	412
Figure 7.2 Establishing the connection.....	413
Figure 7.3 Microsoft Remote Desktop application.....	413
Figure 7.4 Server’s specifications.....	414
Figure 7.5 Setting up a conda environment.....	415
Figure 7.6 GPU not detected.....	415
Figure 7.7 GPU detected.....	416

Abstract

This project entails designing and implementing a dual mobile application system to optimise inventory management and sales forecasting for grocery stores and vendors. The system includes two apps: one for vendors to manage inventory and orders, and one for grocery store owners to place orders and address stock issues. Initial research involved studying systems like Shelf Engine and Guac, interviewing grocery store owners in Lahore, and examining local apps like Tajir and Dastagyr. The frontend for both apps was built with Flutter, the admin portal with React, and the backend with NodeJS and ExpressJS, using Sequelize and MySQL. Machine learning models, initially developed with Scikit-Learn and TensorFlow and refined with the Darts library, were trained on the large 'Corporación Favorita Grocery Sales Forecasting' dataset using batch training, resulting in 1782 models for different products per store. An ensemble model of Random Forest, XGBoost, Prophet, and LightGBM achieved high prediction accuracy with an RMSE of 159. The models were served via a Flask API and integrated into the NodeJS backend, deployed on Digital Ocean. Future improvements include enhancing prediction accuracy with local data, exploring advanced transformers, and performance enhancements through local caching and Cloudflare integration. This project highlights the potential of machine learning in improving inventory management and sales forecasting in the grocery industry.

1 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 operability. 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.

1.1 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 1.1.1, 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.

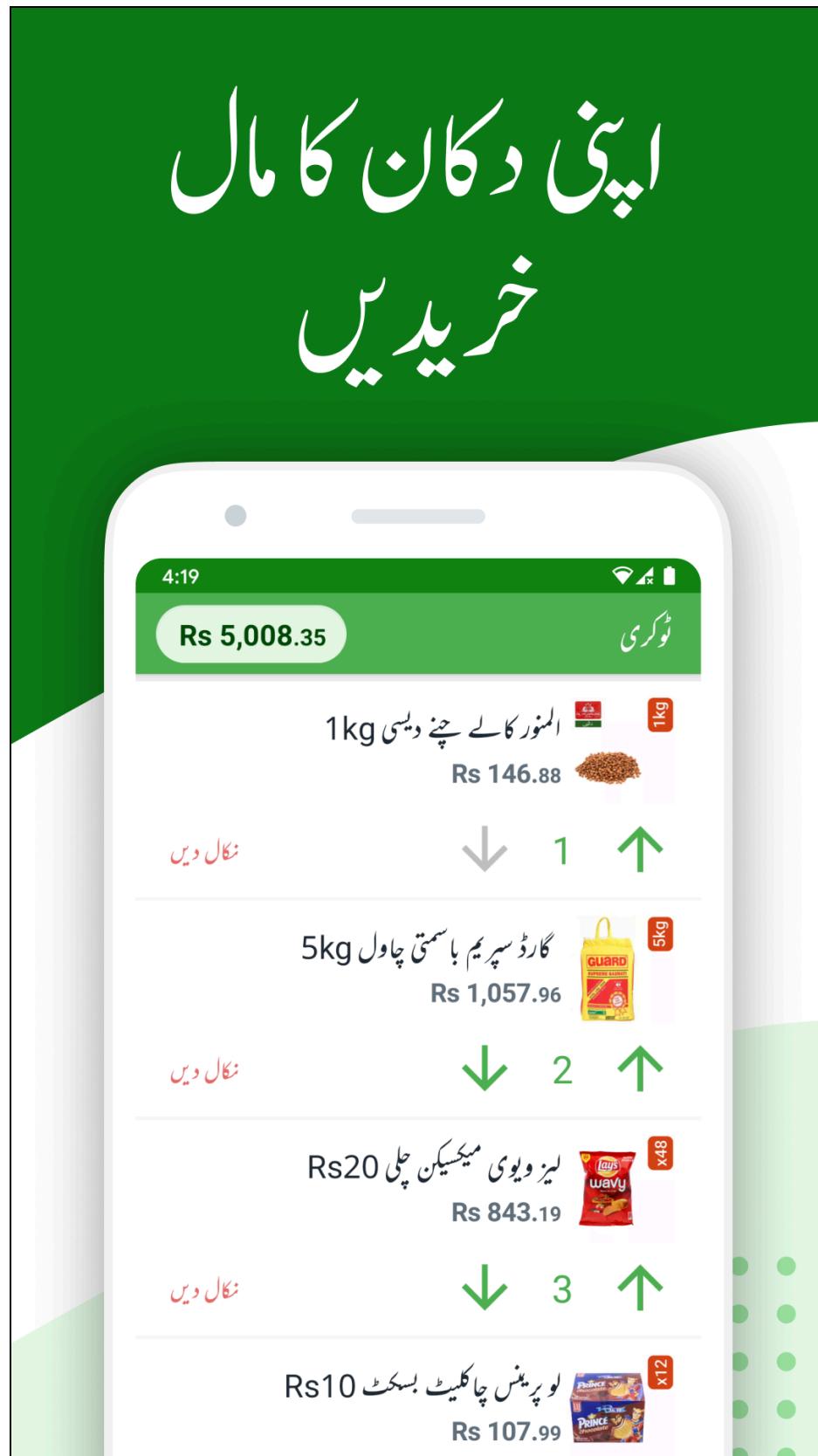


Figure 1.1.1 Tajir Mobile Application

Adding to this is Jugnu, a Business to Business (B2B) e-commerce platform founded in 2020. Jugnu, seen in Figure 1.1.2 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.

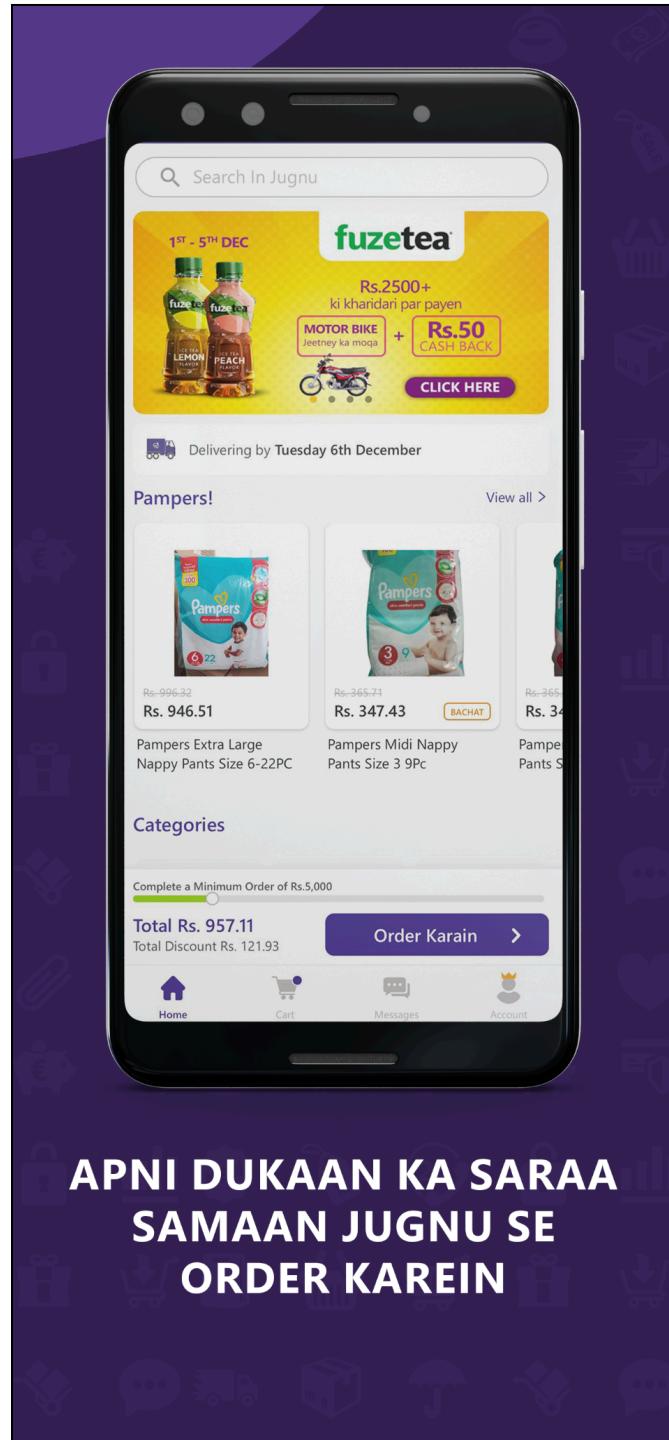


Figure 1.1.2 Jugnu Mobile Application

Below in Figure 1.1.3 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.

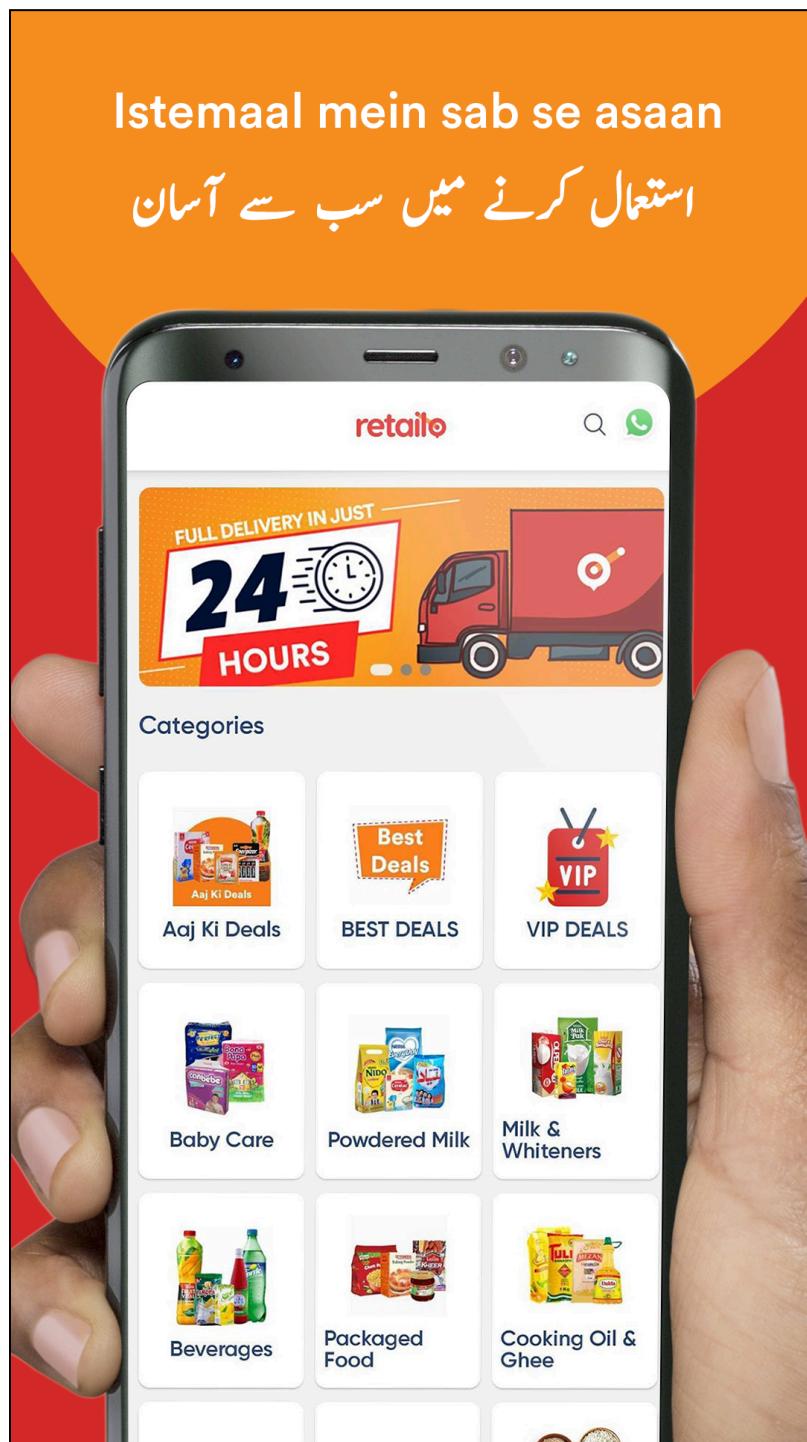


Figure 1.1.3 Retailo Mobile Application

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 1.1.4, and receive on-demand delivery.

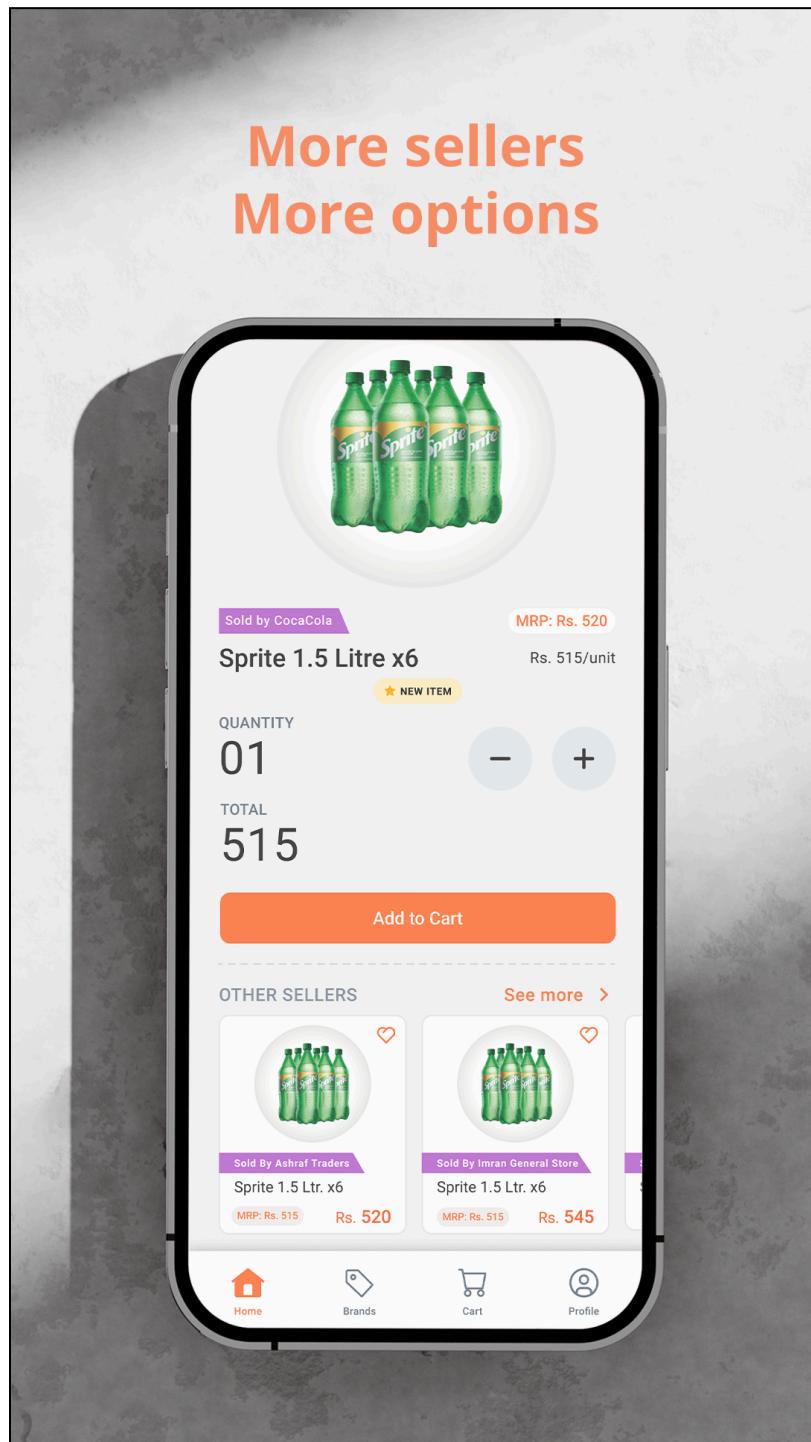


Figure 1.1.4 Dastgyr Mobile Application

Next is Bazaar in Figure 1.1.5 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.

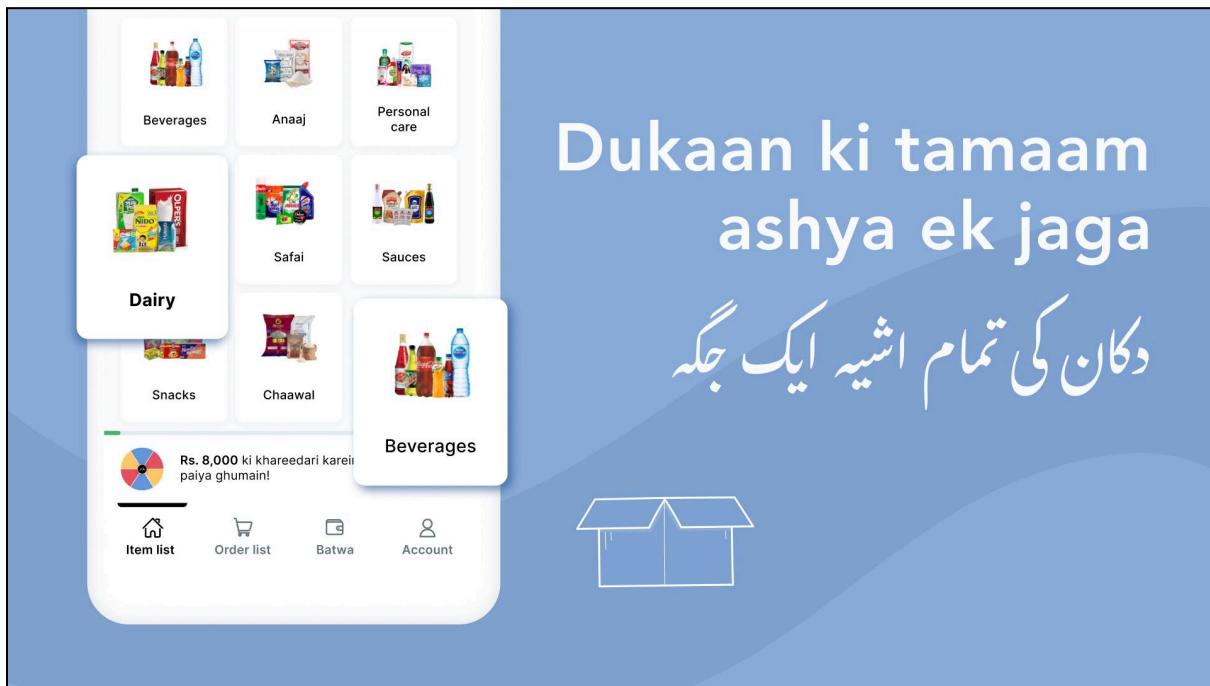


Figure 1.1.5 Bazaar Mobile Application

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.

1.2 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, there are US based platforms like Shelf Engine shown in Figure 1.2.1 and Guac shown in Figure 1.2.2, 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.

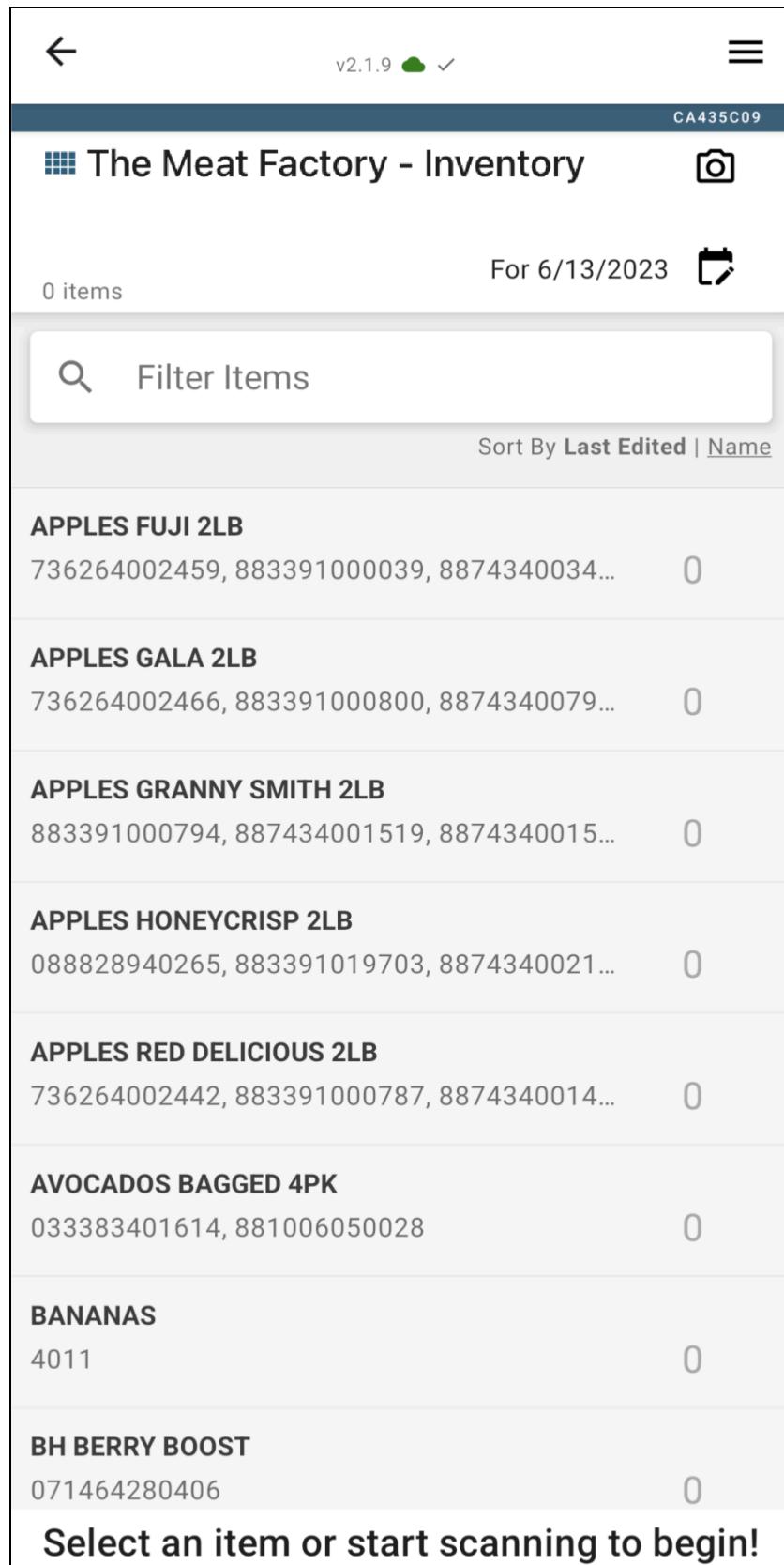


Figure 1.2.1 Shelf Engine Mobile Application

The screenshot shows the Guac web application interface. At the top, there's a navigation bar with icons for Predictions, Daily Prep, Orders (selected), Past Orders, and Analytics. Below the navigation is a search bar with filters for date (2-Jul - 5-Jul) and an 'EXPORT CSV' button.

The main area is titled 'Orders' and displays a table of items ordered. The table has columns for Category, Item, Current (Estimated), Order (Recommended), and Cost. The categories are Fresh and Refrigerated. The items listed are Apple (Fresh) and Guacamole (Refrigerated). The table includes a notes section with demand drivers and a summary at the bottom.

CATEGORY	ITEM	CURRENT (ESTIMATED)	ORDER (RECOMMENDED)	COST
Fresh	Apple	- 5 cases +	- 1 case +	£10.10
Refrigerated	Guacamole	- 3 boxes +	- 1 box +	\$7.30
Fresh	Watermelon	- 23 units +	- 8 units +	\$8.50

NOTES

- 50% Last week you ordered 16 litres
- > 50% The average order is 4 litres

DEMAND DRIVERS

- ↑ Start of school summer holidays
- ↓ Unseasonably bad weather (cold temperature)

TOTAL: \$102.20

PLACE ORDER

Superfresh
London, E1 6RU

Figure 1.2.2 Guac Web Application

2 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.

2.1 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.

2.1.1 Data Collection Technique

Interviews were used as they 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.

2.1.2 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 2.1.2, 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 2.1.2 Jalal Son's (taken on 5th November 2023)

In summary, an outdated ordering system 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

2.1.3 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.

In summary, 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.

2.1.4 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.

2.2 System Environment

Figure 2.2 provides a simple overview of our system by describing the main communication.

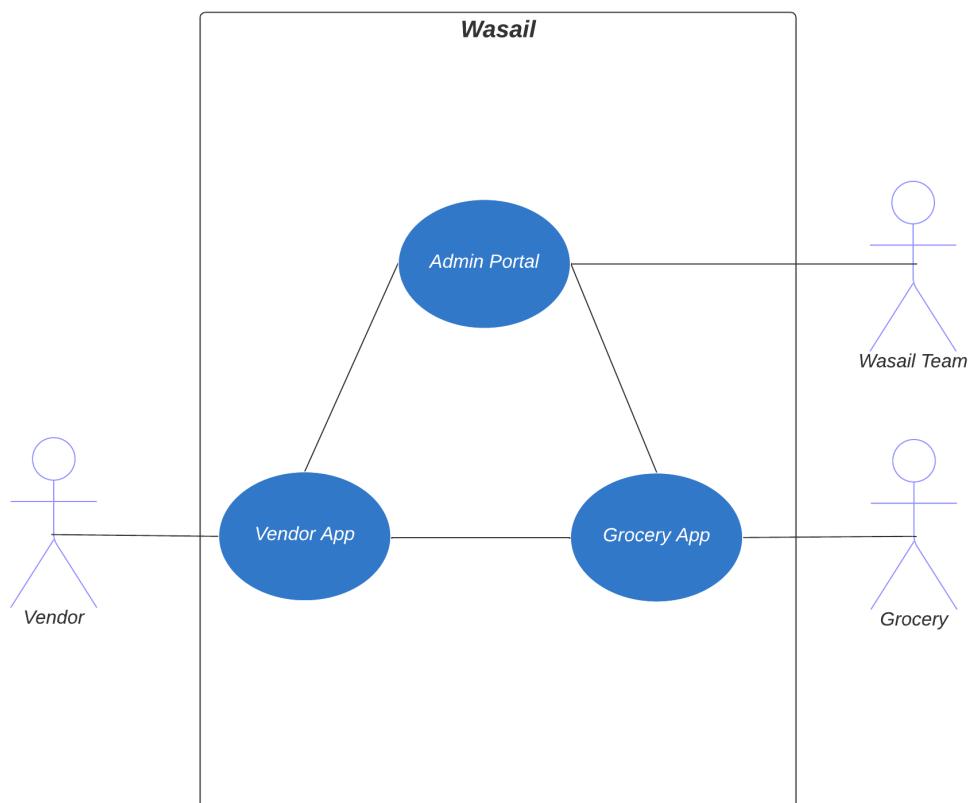


Figure 2.2 System Environment

2.3 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.”

2.3.1 Data Collection

In this we discuss four datasets out of which two were locally acquired and two were acquired from Kaggle.

2.3.1.1 Local Vendor Dataset

This dataset was collected from a local vendor in Lahore, Pakistan. The data, seen in Figure 2.3.1.1, 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.

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

Item ID	Item Description	Date	Qty Received	Item Cost	Actual Cost	Assembly Qty	Assembly (\$)	Adjust Qty	Adjust (\$)	Quantity Sold	Cost of Sales	Remaining Qty	Remaining Value
AAP001	AGAR AGAR-POWDER 25G	1/27/20	100.00	297.44	29,744.00			100.00	29,744.00			100.00	29,744.00
AAP001	AGAR AGAR POWDER 25G	1/29/20						100.00	29,744.00			100.00	29,744.00
ABC001	ABC SWEET SOY SAUCE 500ML											96.00	69,233.28
ABC001	ABC SWEET SOY SAUCE 500ML	7/24/20	96.00	721.18	69,233.28			24.00	17,308.32	72.00		51,924.96	
ABC001	ABC SWEET SOY SAUCE 500ML	7/25/20						24.00	17,308.32	72.00		51,924.96	
ABC001	ABC SWEET SOY SAUCE 500ML	7/25/20	120.00	710.25	85,230.00			72.00	51,924.96	120.00		85,230.00	
ABC001	ABC SWEET SOY SAUCE 500ML	8/8/20						36.00	25,569.00	84.00		59,681.00	
ABC001	ABC SWEET SOY SAUCE 500ML	8/11/20						84.00	59,681.00				
ABC001	ABC SWEET SOY SAUCE 500ML	8/23/20	72.00	672.60	48,427.20			72.00	48,427.20	72.00		48,427.20	
ABC001	ABC SWEET SOY SAUCE 500ML	8/25/20						72.00	48,427.20	72.00		48,427.20	
ABC001	ABC SWEET SOY SAUCE 500ML	9/7/20	72.00	672.60	48,427.20			48.00	32,284.80	24.00		16,142.40	
ABC001	ABC SWEET SOY SAUCE 500ML	9/12/20						48.00	32,284.80	312.00		221,860.80	
ABC001	ABC SWEET SOY SAUCE 500ML	9/27/20	288.00	714.30	205,718.40			60.00	41,857.20	252.00		180,003.60	
ABC001	ABC SWEET SOY SAUCE 500ML	11/3/20						60.00	42,958.00	192.00		137,145.60	
ABC001	ABC SWEET SOY SAUCE 500ML	11/21/20						80.00	85,716.00	72.00		51,429.60	
ABC001	ABC SWEET SOY SAUCE 500ML	11/24/20						88.00	48,572.40	4.00		2,857.20	
ABC001	ABC SWEET SOY SAUCE 500ML	11/31/21						4.00	2,857.20				
AC001	AMERI COLOR												
AC001	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			30.00	19,531.80	-30.00		-19,531.80	
ACTC02	ANTICA CANTINA CHILLI 200G												
ACTC02	ANTICA CANTINA CHILLI 200G	8/29/20											
ACTC02	ANTICA CANTINA CHILLI 200G	8/29/20	30.00	651.06	19,531.80			30.00	19,531.80	-30.00		-19,531.80	
ACTC03	ANTICA CANTINA N.CHEESE 200G												
ACTC03	ANTICA CANTINA N.CHEESE 200G	8/29/20											
ACTC03	ANTICA CANTINA N.CHEESE 200G	8/29/20	30.00	651.06	19,531.80			30.00	19,531.80	-30.00		-19,531.80	
ACTC04	ANTICA CANTINA BBQ 200G												
ACTC04	ANTICA CANTINA BBQ 200G	8/29/20											
ACTC04	ANTICA CANTINA BBQ 200G	8/29/20	30.00	651.06	19,531.80			30.00	19,531.80	-30.00		-19,531.80	
ACTD01	ANTICA CANTINA MILD SALSA 300G												
ACTD01	ANTICA CANTINA MILD SALSA 300G	8/29/20											
ACTD02	ANTICA CANTINA MED SALSA 300G	8/29/20	12.00	915.19	10,982.28			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			12.00	10,982.28	-12.00		-10,982.28	

Figure 2.3.1.1 Local Vendor Dataset

2.3.1.2 Local Pharmacy Dataset

This dataset was collected from a local pharmacy in Lahore, Pakistan. The data, seen in Figure 2.3.1.2, 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.

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

saleinvoiceno	CustomerRef	date	Indisdiscp:	flatdisc	misscharges	Invstestax	remarks	loosely	packqty	price	lendisdiscp:	packunits	batch	expiry	sales tax	itemname	deleting	customer	rwid
268401		4/1/23 9:07	0	0	0	0		1	0	175	0	1	10/12/24 00:00	0	CARE BABY WIPES(80)	10/4/2023	***CASH SALES CUSTOMER	52/334	
268402		4/1/23 9:28	0	0	0	0		1	0	105.2	3	10	10/12/24 00:00	0	ITEM NEBULIZER-10S	10/4/2023	***CASH SALES CUSTOMER	52/335	
268402		4/1/23 9:28	0	0	0	0		1	0	97.6	3	10	10/12/24 00:00	0	CLENLA INJ	10/4/2023	***CASH SALES CUSTOMER	52/336	
268402		4/1/23 9:28	0	0	0	0		1	0	23.69	0	1	10/12/25 00:00	0	NS 25ML AMP (OTSUBUKA)	10/4/2023	***CASH SALES CUSTOMER	52/337	
268403		4/1/23 9:30	0	0	0	0		1	0	30	0	100	10/22/27 00:00	0	10CC SHIFT DISFRINGE(UNI)(BM)	10/4/2023	***CASH SALES CUSTOMER	52/338	
268403		4/1/23 9:30	0	0	0	0		23	0	10	1	10	10/22/24 00:00	0	FACE MASK - PLY GREEN (FS(5)	10/4/2023	***CASH SALES CUSTOMER	52/339	
268405		4/1/23 9:32	1	0	0	0		4	0	25.33	0	30	10/12/24 00:00	0	DEXXOO 30MG CAP 30'S	10/4/2023	***CASH SALES CUSTOMER	52/340	
268406		4/1/23 9:42	0	3	0	0		1	0	17.9	1	10	10/12/24 00:00	0	MARK 40MG CAP 10'S XANS PHARMAY	10/4/2023	***CASH SALES CUSTOMER	52/341	
268406		4/1/23 9:46	0	3	0	0		1	0	138.61	0	1	11/12/25 00:00	0	LAXOBERON 120ML LIQ	10/4/2023	***CASH SALES CUSTOMER	52/342	
268407		4/1/23 10:04	5	0	0	0		10	0	4.43	0	100	10/4/25 00:00	0	LAXOBERON 5ML TAB	10/4/2023	***CASH SALES CUSTOMER	52/343	
268407		4/1/23 10:54	5	0	0	0		10	0	35	0	10	7/1/24 00:00	0	LIPTOR 10MG TAB(PARKE DAVIS)	10/4/2023	***CASH SALES CUSTOMER	52/344	
268408		4/1/23 10:57	0	0	0	0		1	0	64.52	0	20	9/1/24 00:00	0	ENTERO GERMINA INJ	10/4/2023	***CASH SALES CUSTOMER	52/345	
268409		4/1/23 11:04	0	0	0	0		4	0	22	0	10	4/1/24 00:00	0	PIZER G 152MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/346	
268409		4/1/23 11:04	0	0	0	0		1	0	22	0	10	12/1/24 00:00	0	PIZER G 152MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/347	
268410		4/1/23 11:20	10	0	0	0		60	0	25.06	20	..	11/1/24 00:00	0	DAMICHON 15GM CREAM	10/4/2023	***CASH SALES CUSTOMER	52/348	
268411		4/1/23 11:28	2	0	0	0		1	0	54.32	0	1	11/12/28 00:00	0	XYNOSINE 40DRGS 15ML(ADULT)	10/4/2023	***CASH SALES CUSTOMER	52/349	
268411		4/1/23 11:28	7	0	0	0		0	14	0	27.93	0	14	10/1/24 00:00	0	SANTE 60MG CAP	10/4/2023	***CASH SALES CUSTOMER	52/350
268412		4/1/23 11:34	7	0	0	0		0	15	0	27.93	0	10	10/1/24 00:00	0	E70-OD 150MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/351
268412		4/1/23 11:34	7	0	0	0		0	15	0	35	0	10	10/1/24 00:00	0	E70-OD 150MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/352
268412		4/1/23 11:34	7	0	0	0		0	15	0	35	0	30	10/1/24 00:00	0	NERVON TAB	10/4/2023	***CASH SALES CUSTOMER	52/353
268412		4/1/23 11:34	7	0	0	0		0	15	0	41	0	14	12/1/24 00:00	0	VONZAN 20MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/354
268412		4/1/23 11:39	7	0	0	0		0	15	0	41	0	14	10/1/24 00:00	0	KONCEPT 40MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/355
268412		4/1/23 11:39	7	0	0	0		0	15	0	41	0	14	10/1/24 00:00	0	ALP 0.25MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/356
268412		4/1/23 11:39	10	0	0	0		3	0	74.75	0	20	11/1/24 00:00	0	ICON 10MG CAP(FERZOSONS)	10/4/2023	***CASH SALES CUSTOMER	52/357	
268413		4/1/23 11:34	10	0	0	0		3	0	49.75	0	20	11/1/24 00:00	0	CUTIS 25MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/358	
268414		4/1/23 11:37	0	0	0	0		3	0	41.07	0	14	11/1/25 00:00	0	GABICA 15MG CAP	10/4/2023	***CASH SALES CUSTOMER	52/359	
268414		4/1/23 11:41	7	0	0	0		0	15	0	30	0	21	12/1/24 00:00	0	RONIROL 2MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/360
268415		4/1/23 11:41	7	0	0	0		0	15	0	41	0	14	10/1/24 00:00	0	RONIROL 2MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/361
268415		4/1/23 11:41	7	0	0	0		0	15	0	41	0	14	10/1/24 00:00	0	OCCO 50MG CAP	10/4/2023	***CASH SALES CUSTOMER	52/362
268415		4/1/23 11:41	7	0	0	0		0	15	0	40.02	0	20	11/1/24 00:00	0	O MAXFLOW 0.05MG CAP	10/4/2023	***CASH SALES CUSTOMER	52/363
268415		4/1/23 11:41	7	0	0	0		0	15	0	52	0	20	12/1/24 00:00	0	O MAXLOW 0.05MG CAP	10/4/2023	***CASH SALES CUSTOMER	52/364
268415		4/1/23 11:41	7	0	0	0		0	15	0	52	0	20	12/1/24 00:00	0	O VITRUM TAB(SEARLE)	10/4/2023	***CASH SALES CUSTOMER	52/365
268415		4/1/23 11:41	7	0	0	0		0	15	0	19.42	0	20	12/1/24 00:00	0	O ELAXINE 15MG TAB (METHAZAPINE)	10/4/2023	***CASH SALES CUSTOMER	52/366
268415		4/1/23 11:41	7	0	0	0		0	15	0	25	0	14	11/1/24 00:00	0	O NEIUXAN 25MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/367
268415		4/1/23 11:41	7	0	0	0		0	15	0	30	0	12/1/24 00:00	0	O SINNET EXTRA 25+10MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/368	
268415		4/1/23 11:41	7	0	0	0		0	15	0	30	0	12/1/25 00:00	0	O SINNET EXTRA 25+10MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/369	
268415		4/1/23 11:41	7	0	0	0		0	15	0	30	0	12/1/25 00:00	0	O LAMICIL 50MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/370	
268416		4/1/23 11:48	0	0	0	0		0	15	0	51.56	0	20	11/1/24 00:00	0	O TONOFLEX-P TAB (NEW)	10/4/2023	***CASH SALES CUSTOMER	52/371
268416		4/1/23 11:48	0	0	0	0		1	0	17.15	0	20	8/1/25 00:00	0	O NEIUXAN 0.5 TAB	10/4/2023	***CASH SALES CUSTOMER	52/372	
268416		4/1/23 11:48	0	0	0	0		10	0	8.12	0	20	8/1/25 00:00	0	O NEIUXAN 1MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/373	
268416		4/1/23 11:48	0	0	0	0		14	0	16.06	0	30	8/1/25 00:00	0	O NEIUXAN 1MG TAB NEW	10/4/2023	***CASH SALES CUSTOMER	52/374	
268416		4/1/23 11:48	0	0	0	0		14	0	9.36	0	50	9/1/25 00:00	0	O LEXOTAN 1MG TAB NEW	10/4/2023	***CASH SALES CUSTOMER	52/375	
268416		4/1/23 11:48	0	0	0	0		1	0	20	20	..	12/1/24 00:00	0	O MOVAX 2MG TAB (NEW)	10/4/2023	***CASH SALES CUSTOMER	52/376	
268416		4/1/23 11:48	0	0	0	0		1	0	6.96	10	30	9/1/24 00:00	0	O ZYRET 10MG TAB (30'S)	10/4/2023	***CASH SALES CUSTOMER	52/377	
268416		4/1/23 11:48	0	0	0	0		1	0	32.14	10	21	4/1/25 00:00	0	O RISER 40MG CAP (NEW (2'S))	10/4/2023	***CASH SALES CUSTOMER	52/378	
268417		4/1/23 11:49	5.5	2	0	0		14	0	27.75	0	20	12/1/24 00:00	0	O CITANEW 10MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/379	
268417		4/1/23 11:49	5.5	2	0	0		14	0	33.93	0	14	11/1/24 00:00	0	O GASICA 10MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/380	
268417		4/1/23 11:49	5.5	2	0	0		14	0	33.21	0	14	11/1/24 00:00	0	O SITA MET 50/100MG TAB	10/4/2023	***CASH SALES CUSTOMER	52/381	
268417		4/1/23 11:49	5.5	2	0	0		14	0	8.7	0	20	11/1/24 00:00	0	O NISET TAB	10/4/2023	***CASH SALES CUSTOMER	52/382	
268418		4/1/23 11:54	10	0	0	0		7	0	28.57	0	14	11/1/24 00:00	0	O MONTIKA 0MG TAB(SAMI)	10/4/2023	***CASH SALES CUSTOMER	52/383	
268418		4/1/23 11:54	10	0	0	0		7	0	13.75	0	30	12/1/25 00:00	0	O LORIN 150MG TAB(3'S)	10/4/2023	***CASH SALES CUSTOMER	52/384	
268418		4/1/23 11:54	10	0	0	0		7	0	48.26	0	10	12/1/25 00:00	0	O KLARICID 250MG TAB (ABBOTT)	10/4/2023	***CASH SALES CUSTOMER	52/385	
268418		4/1/23 11:54	10	0	0	0		7	0	23.81	0	100	12/1/25 00:00	0	O METHYCOBAL TAB	10/4/2023	***CASH SALES CUSTOMER	52/386	
268419		4/1/23 11:54	0	0	0	0		1	0	50	12	..	12/1/24 00:00	0	O KNIGHT RIDER CONDOM 3'S	10/4/2023	***CASH SALES CUSTOMER	52/387	

Figure 2.3.1.2 Local Pharmacy Dataset

2.3.1.3 [Corporación Favorita Grocery Sales Forecasting](#)

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 [\[25\]](#).

Datasets

The training data includes dates, store and item information, whether that item was being promoted, as well as the unit sales. Additional files include supplementary information that may be useful in building our models.

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

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.

2.3.1.4 [Instacart Market Basket Analysis](#)

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.

Datasets

The training data includes aisle, department, prior orders, orders, and the products.

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

2.3.1.5 Other Datasets for Further Exploration

- [Grupo Bimbo Inventory Demand](#)
- [Store Item Demand Forecasting Challenge](#)

2.3.2 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].

2.3.3 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.

2.3.4 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].

2.3.5 Integration with User Interface

The ML model will generate a recommended amount to order for each product as shown in Figure 2.3.5 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.

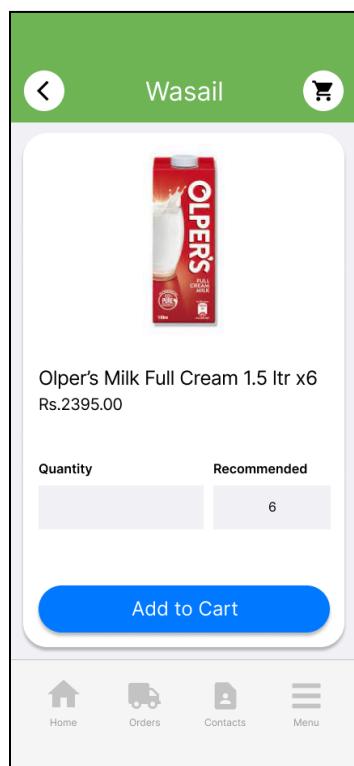


Figure 2.3.5 ML Integration with the User Interface

2.3.6 Recommendation Updates

To generate recommendations, the ML model will be provided with real-time data - product name, price, date, store's name and location, holidays, and oil prices.

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

2.4 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.

2.5 User Stories

2.5.1 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 2.5.1).

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.

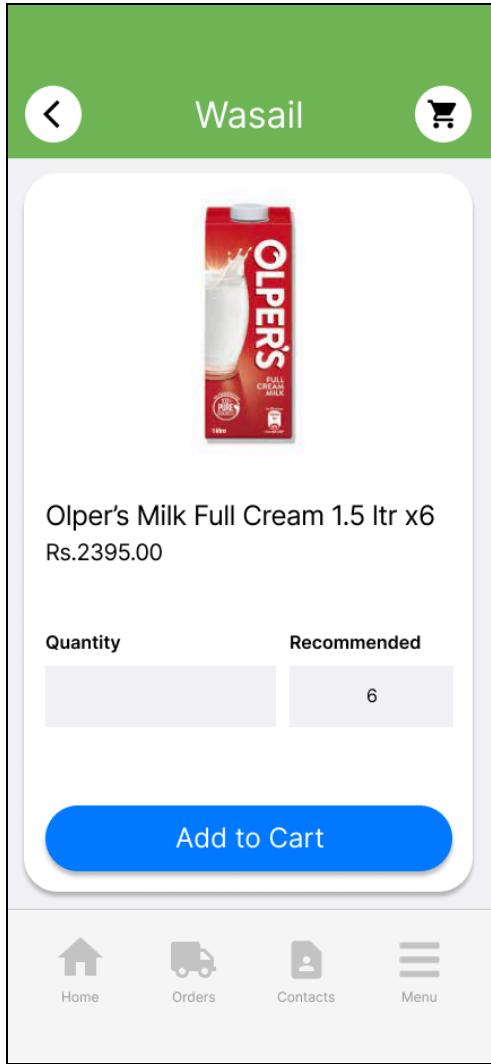


Figure 2.5.1 User Story: Placing the Order

2.5.2 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 2.5.2).

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)

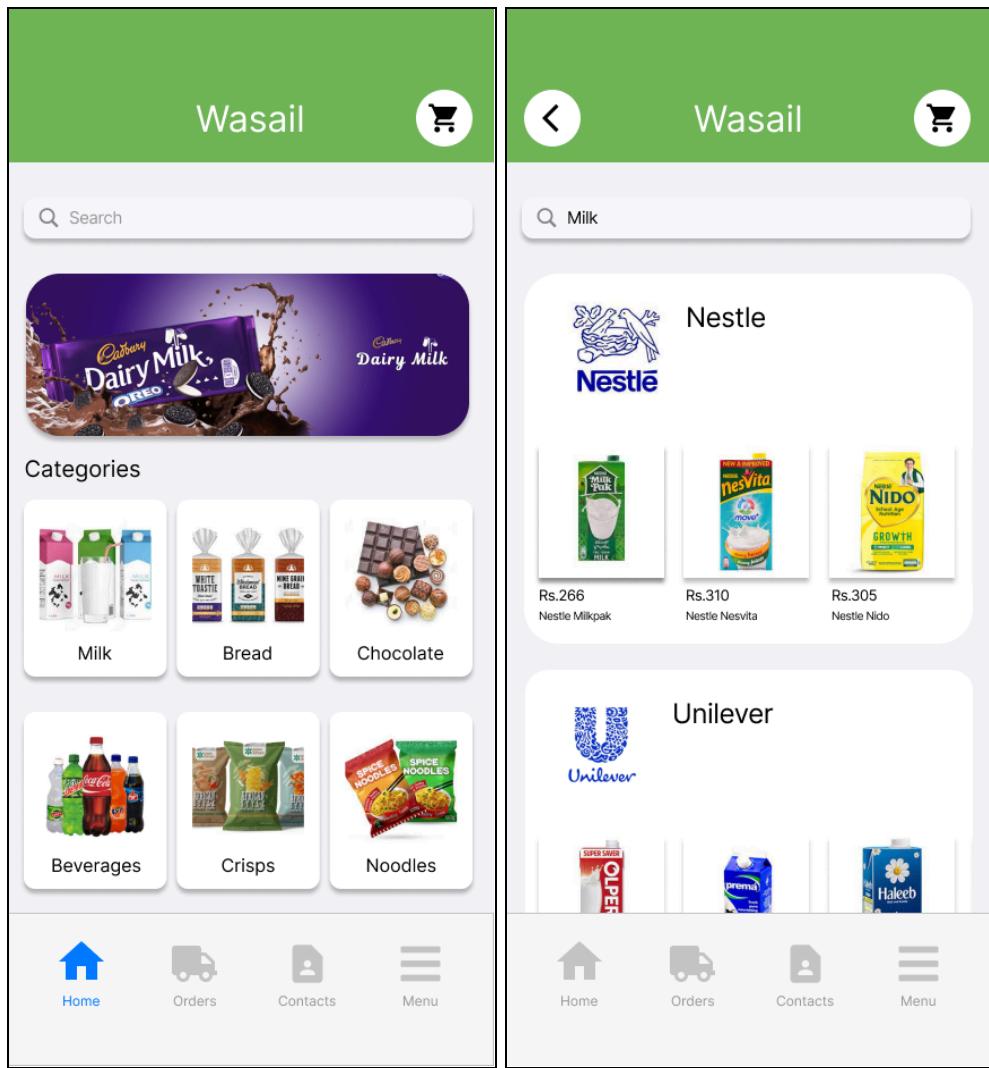


Figure 2.5.2 User Story: Searching for a Product

2.6 Functional Requirements

This section details functional requirements for:

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

2.6.1 Grocery Store and Vendor (FR1)

FR1.1: Language Selection

- **Description:** The system should allow the user to select a language (Figure 2.6.1.1).
- **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 and Urdu.
 3. The user's choice for language is saved.

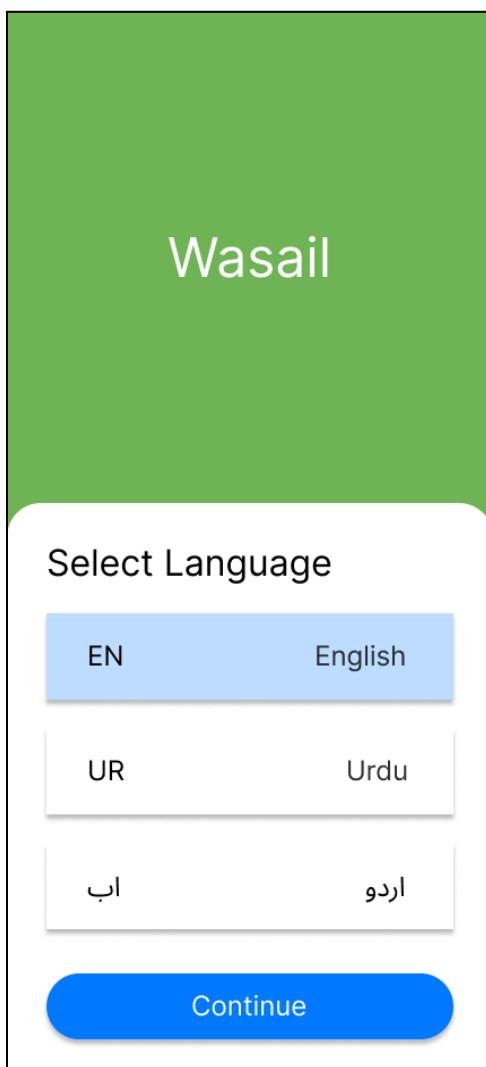


Figure 2.6.1.1 Language Selection Prototype

FR1.2: Phone Registration

- **Description:** The system should allow users to create an account using their phone number (Figure 2.6.1.2).
- **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.

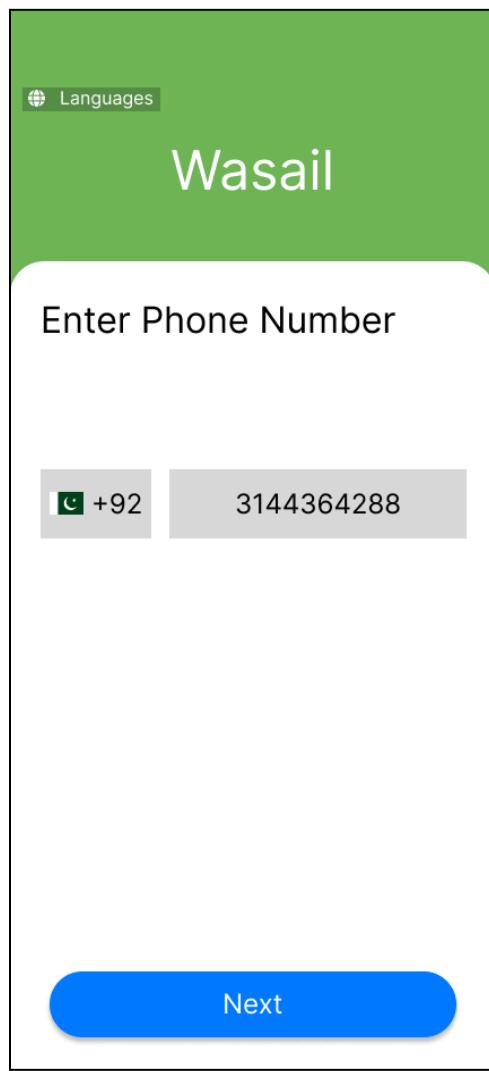


Figure 2.6.1.2 Phone Registration Prototype

FR1.3: Phone Number Confirmation

- **Description:** The system should allow users to confirm if they entered the number correctly (Figure 2.6.1.3).
- **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.

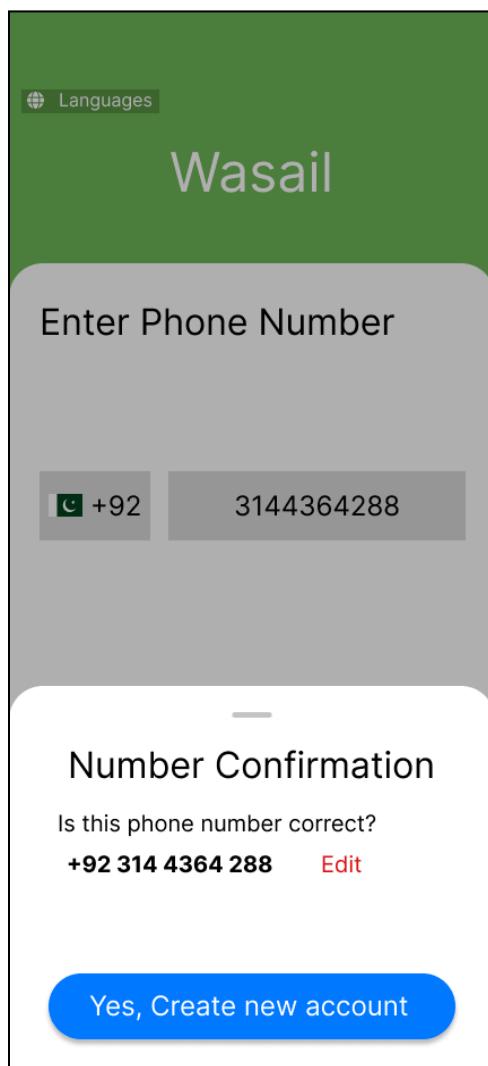


Figure 2.6.1.3 Phone Number Confirmation Prototype

FR1.4: Phone Number Exists

- **Description:** The system should check if the phone number exists in the database (Figure 2.6.1.4).
- **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.

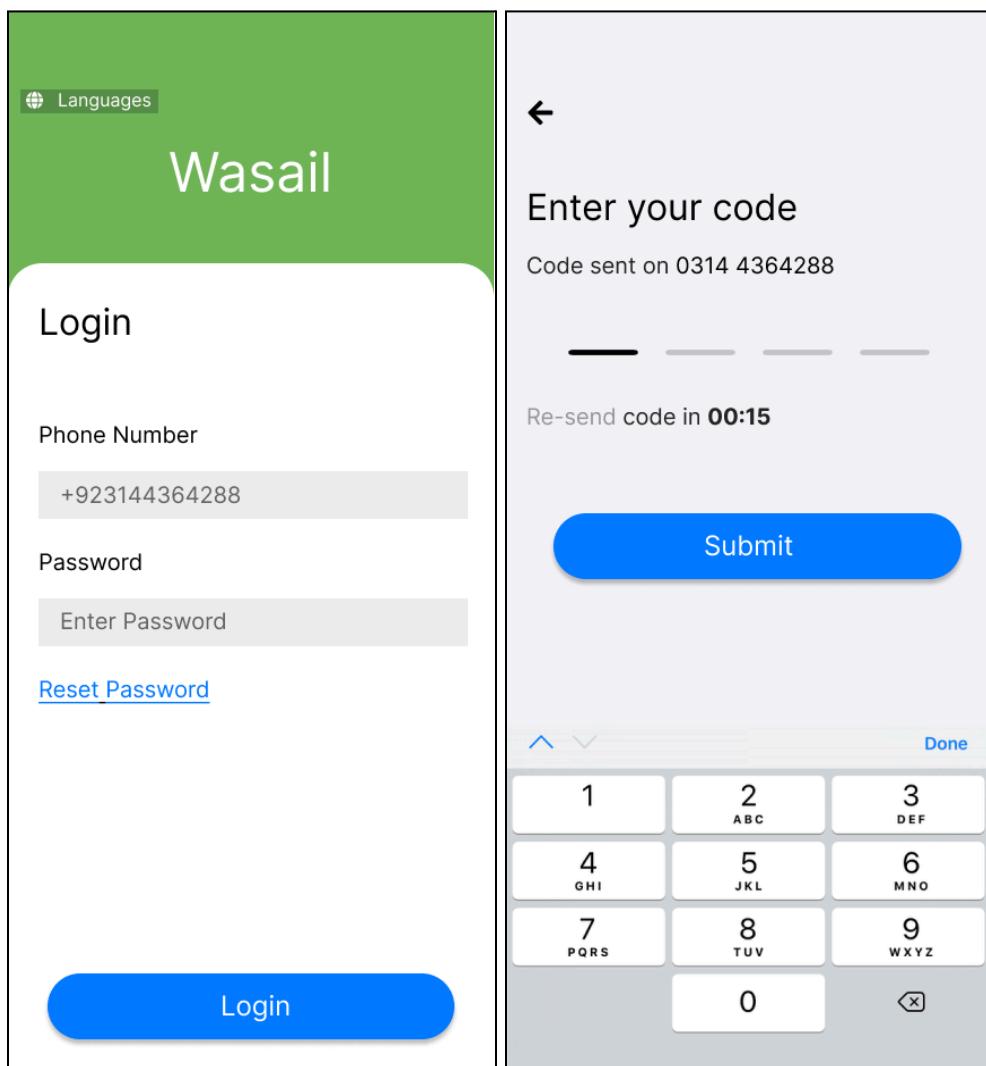


Figure 2.6.1.4 Phone Number Exists Prototype

FR1.5: OTP Code Generation and Delivery

- **Description:** The system should send an OTP code to the user's phone number (Figure 2.6.1.5).
- **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.

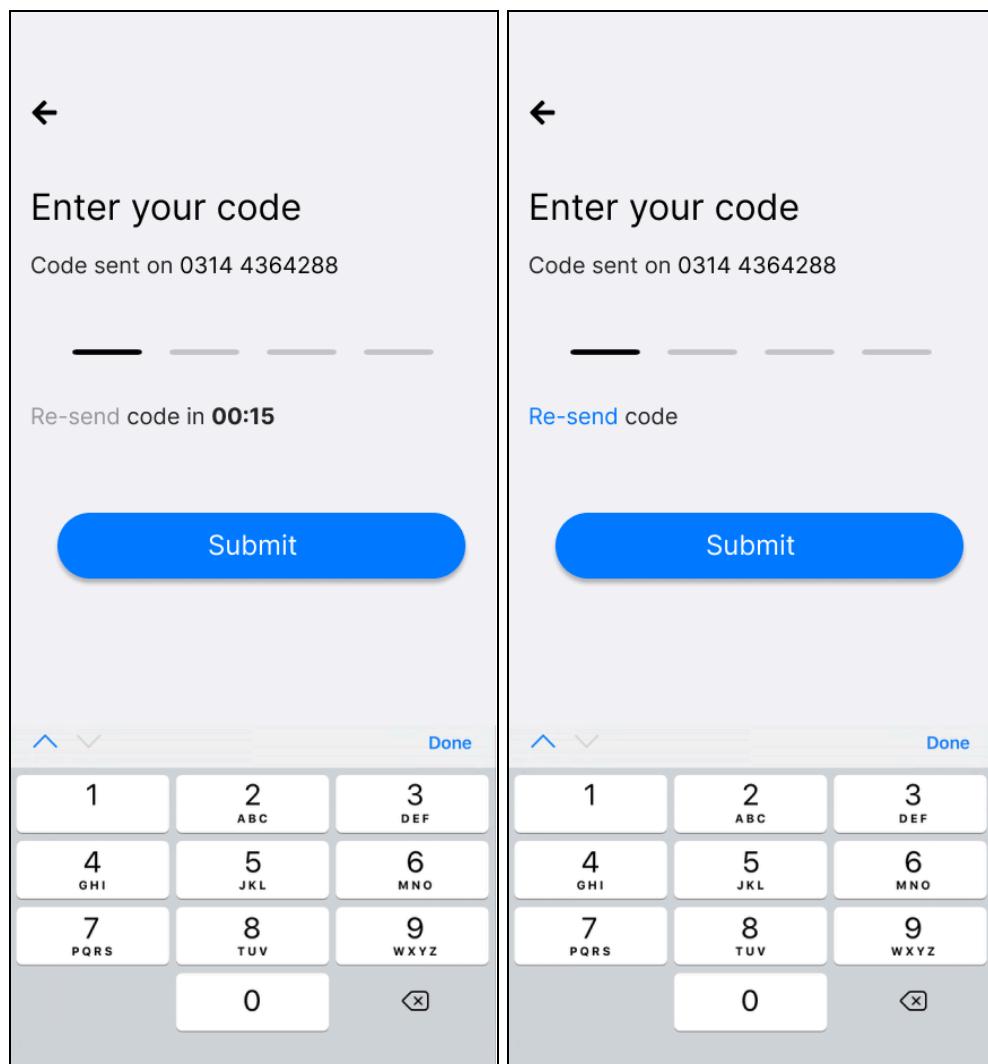


Figure 2.6.1.5 OTP Code Generation and Delivery Prototype

FR1.6: Login

- **Description:** The system should allow the registered users to log in using their credentials (Figure 2.6.1.6).
- **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.

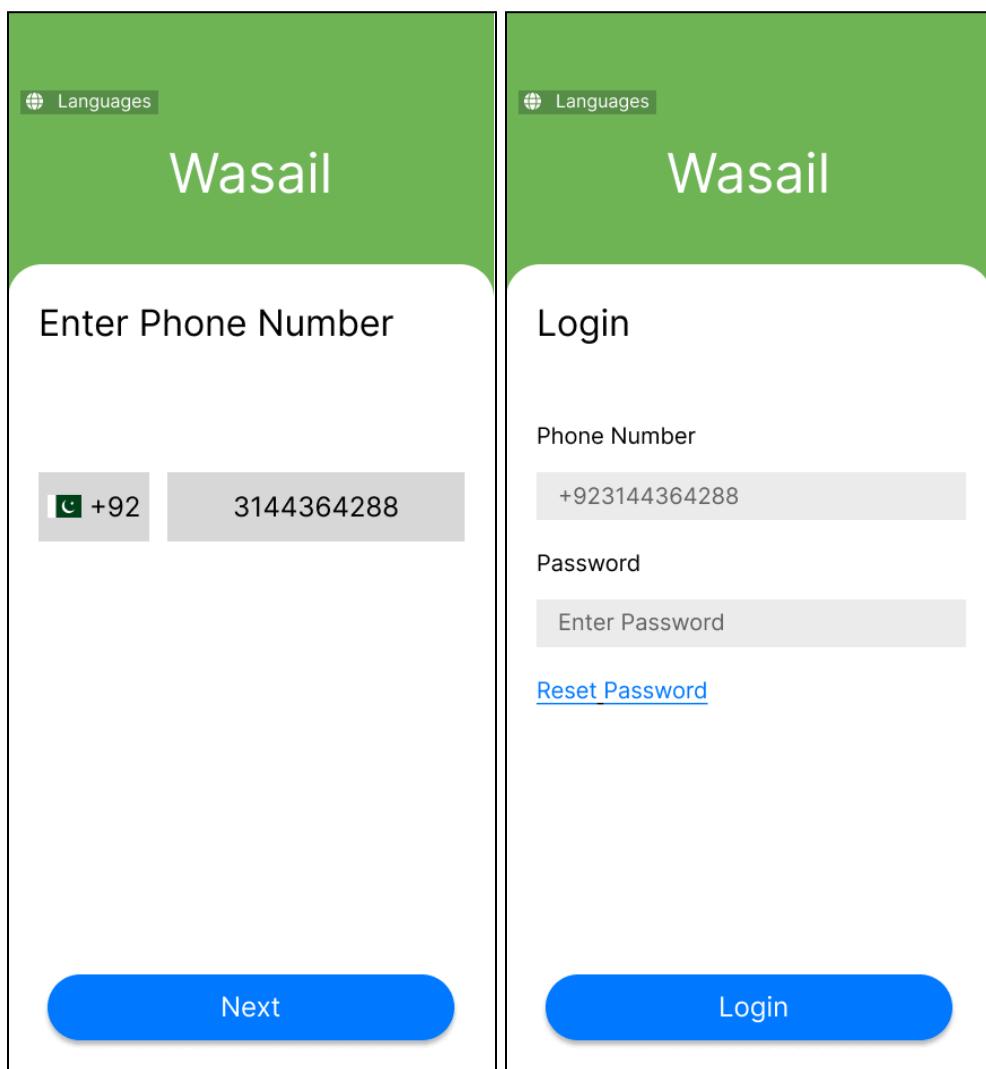


Figure 2.6.1.6 Login Prototype

FR1.7: Logout

- **Description:** The system should allow the user to logout (Figure 2.6.1.7).
- **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.

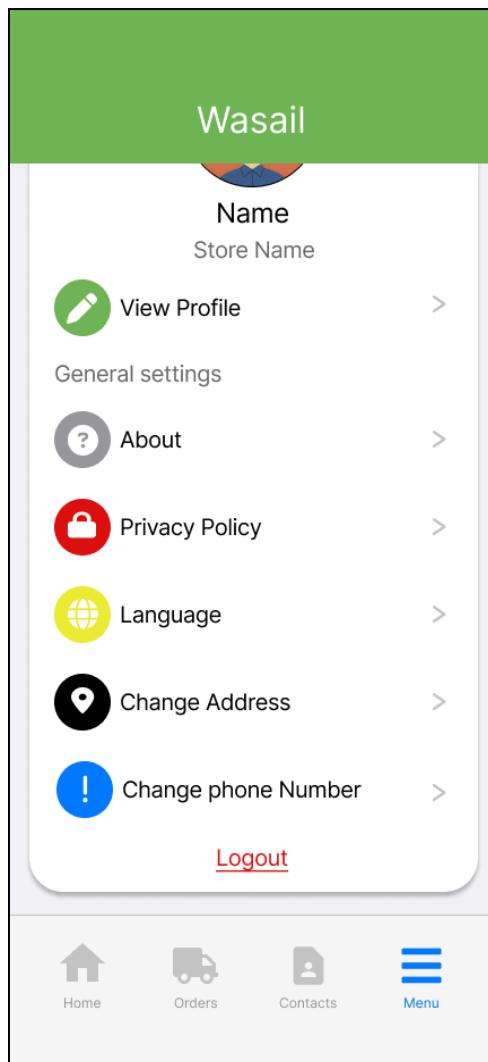


Figure 2.6.1.7 Logout Prototype

FR1.8: Reset Password

- **Description:** The system should allow the user to reset their password (Figure 2.6.1.8).
- **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. The user shall re enter their password for confirmation.
 3. After confirmation, the user shall save the password.
 4. The system updates the password in the database.

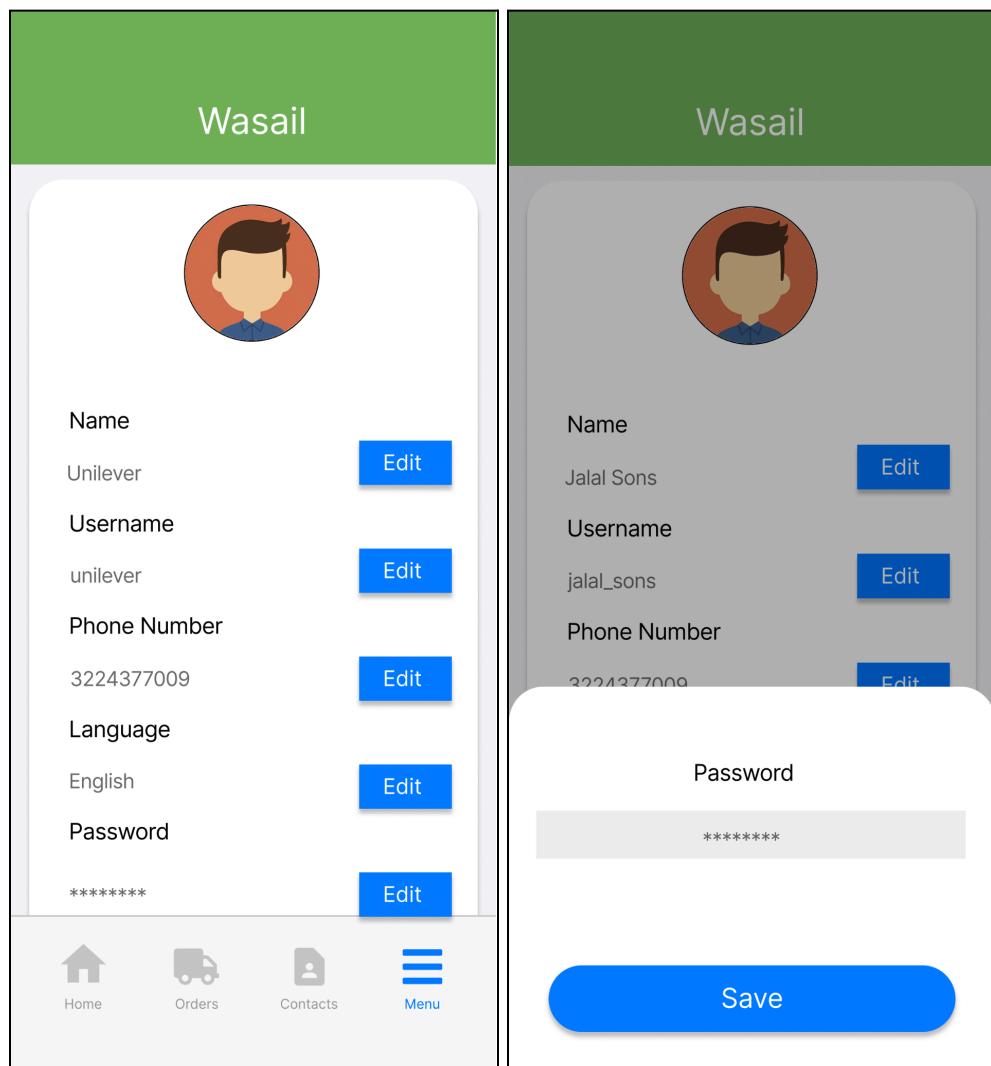


Figure 2.6.1.8 Reset Password Prototype

FR1.9: View Profile

- **Description:** The system should allow the user to view their own profile (Figure 2.6.1.9).
- **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.

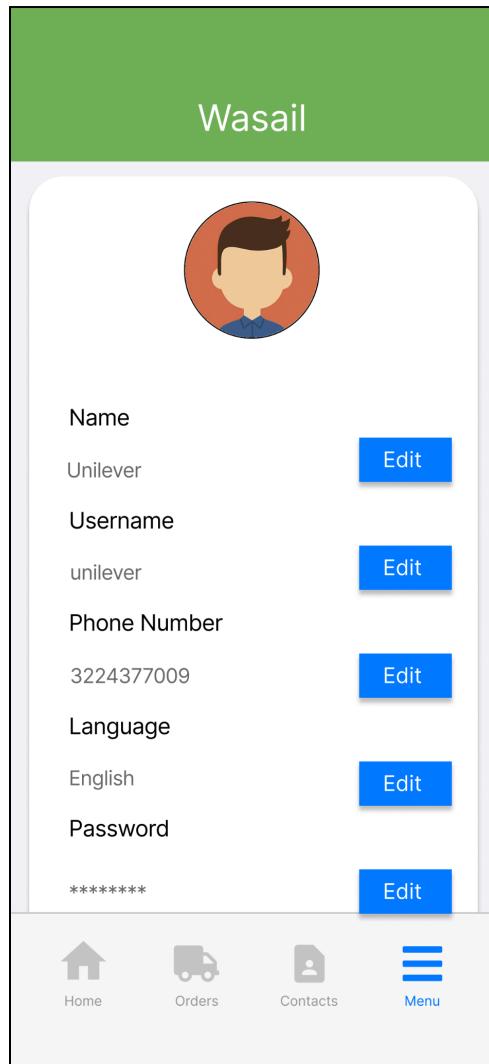
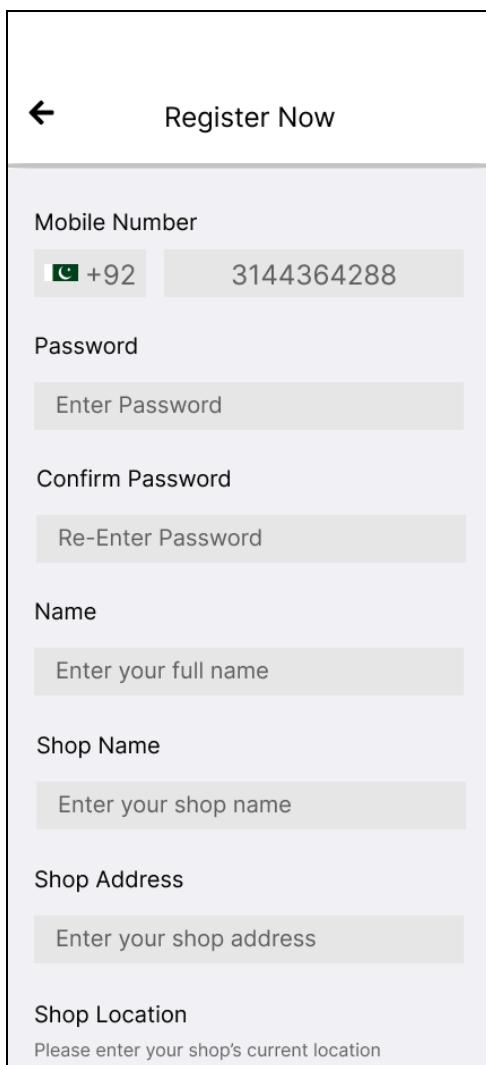


Figure 2.6.1.9 View Profile Prototype

2.6.2 Grocery Store (FR2)

FR2.9: Account Details

- **Description:** The system should allow the user to enter their account details (Figure 2.6.2.1).
- **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.



The image shows a mobile application prototype for account registration. The screen title is "Register Now". It includes fields for "Mobile Number" (with a +92 prefix and the number 3144364288), "Password" (with placeholder "Enter Password"), "Confirm Password" (with placeholder "Re-Enter Password"), "Name" (with placeholder "Enter your full name"), "Shop Name" (with placeholder "Enter your shop name"), "Shop Address" (with placeholder "Enter your shop address"), and "Shop Location" (with placeholder "Please enter your shop's current location"). A back arrow is visible at the top left.

Figure 2.6.2.1 Account Details Prototype

FR2.10: Search Product

- **Description:** The system should allow the users to search products based on product name and category (Figure 2.6.2.2).
- **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.

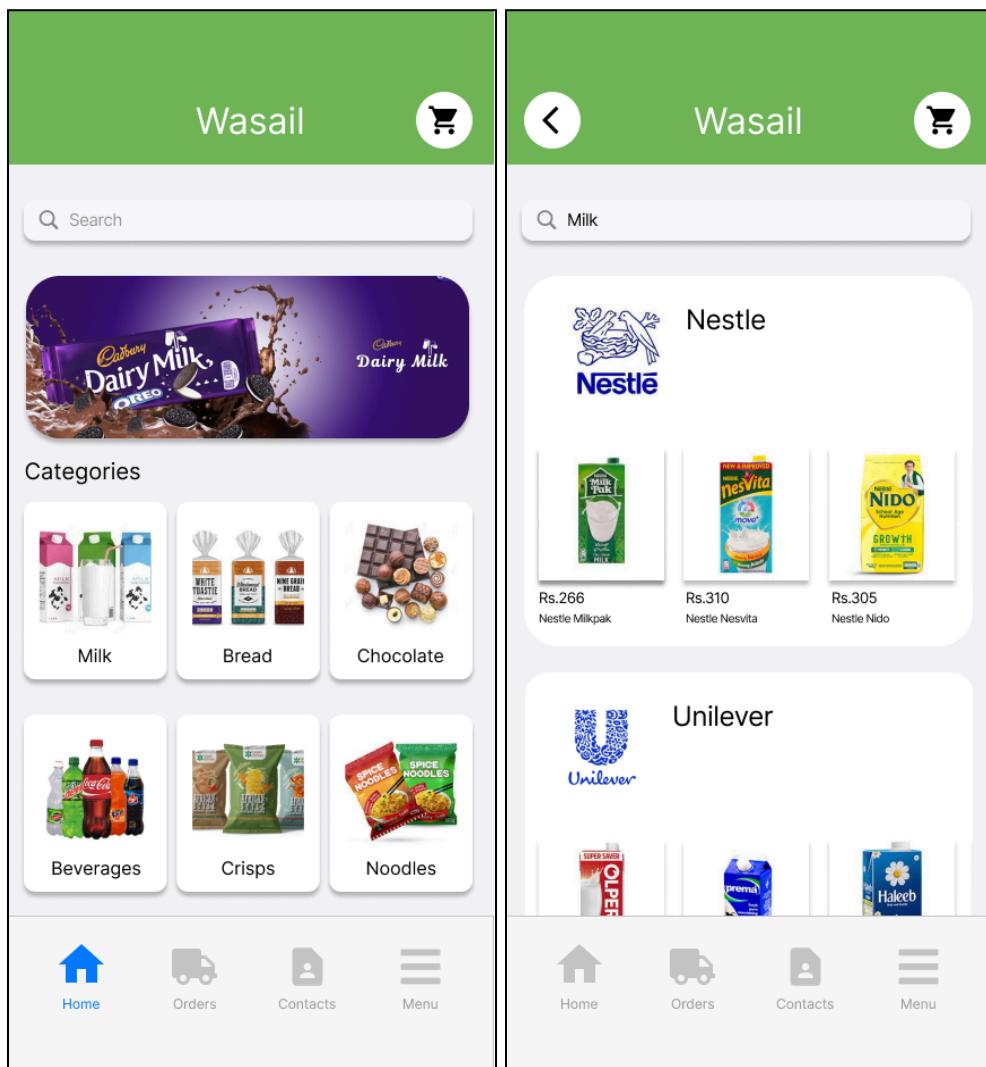


Figure 2.6.2.2 Search Product Prototype

FR2.11: Search Category

- **Description:** The system should allow the users to search categories based on their name (Figure 2.6.2.3).
- **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.

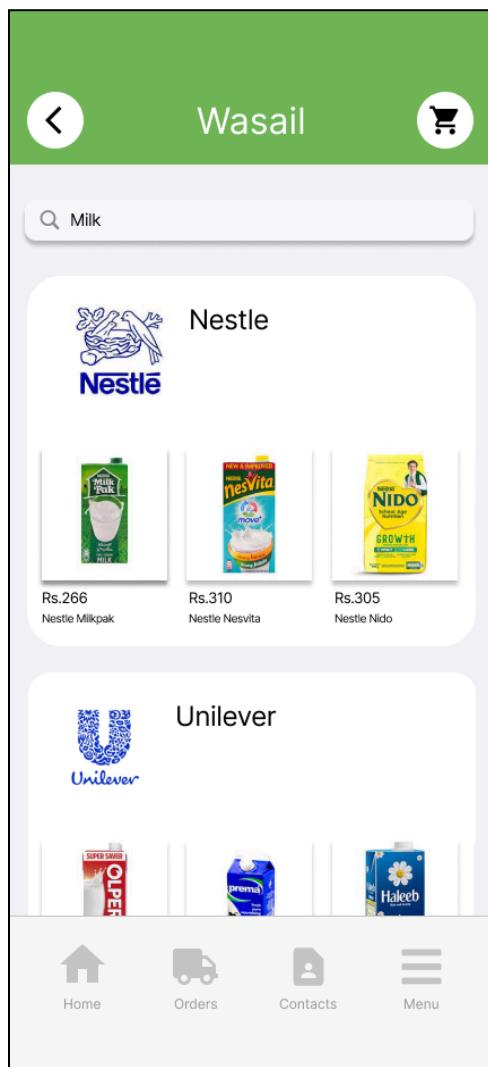


Figure 2.6.2.3 Search Category Prototype

FR2.12: Search Vendor

- **Description:** The system should allow the users to search vendors based on their name (Figure 2.6.2.4).
- **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.

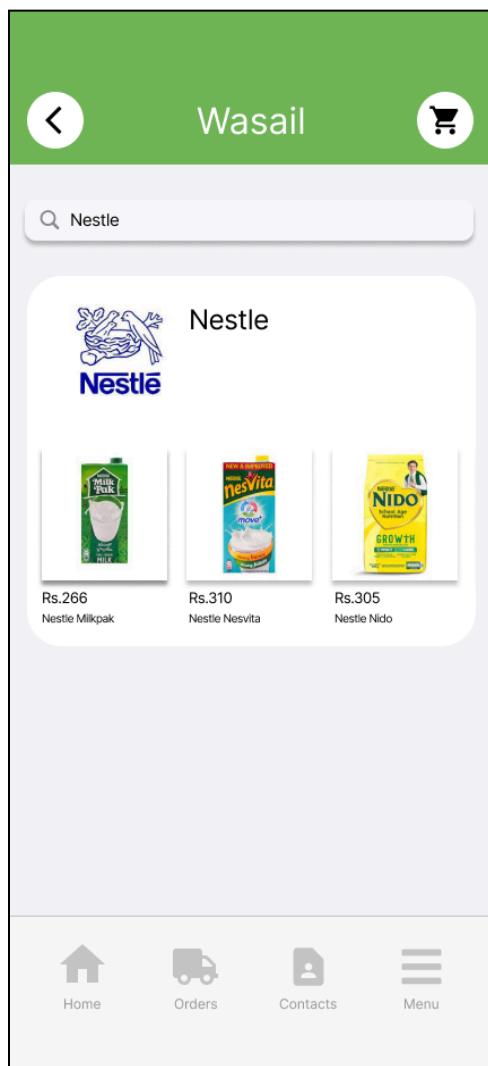


Figure 2.6.2.4 Search Vendor Prototype

FR2.13: Browse Category

- **Description:** The system should allow the user to select a category from the home page directly (Figure 2.6.2.5).
- **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.

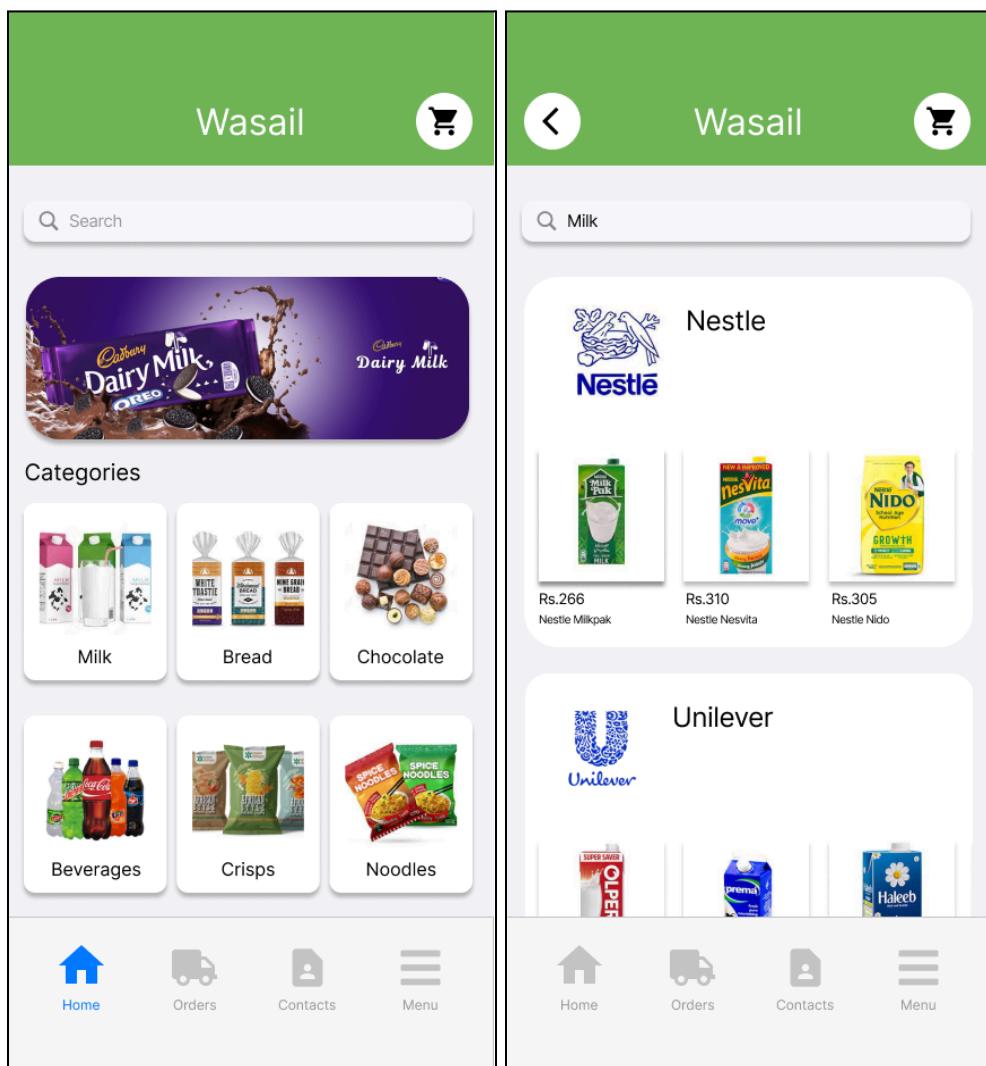


Figure 2.6.2.5 Browse Category Prototype

FR2.14: View Vendor Profile

- **Description:** The system should allow the user to view the vendor's profile (Figure 2.6.2.6).
- **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.

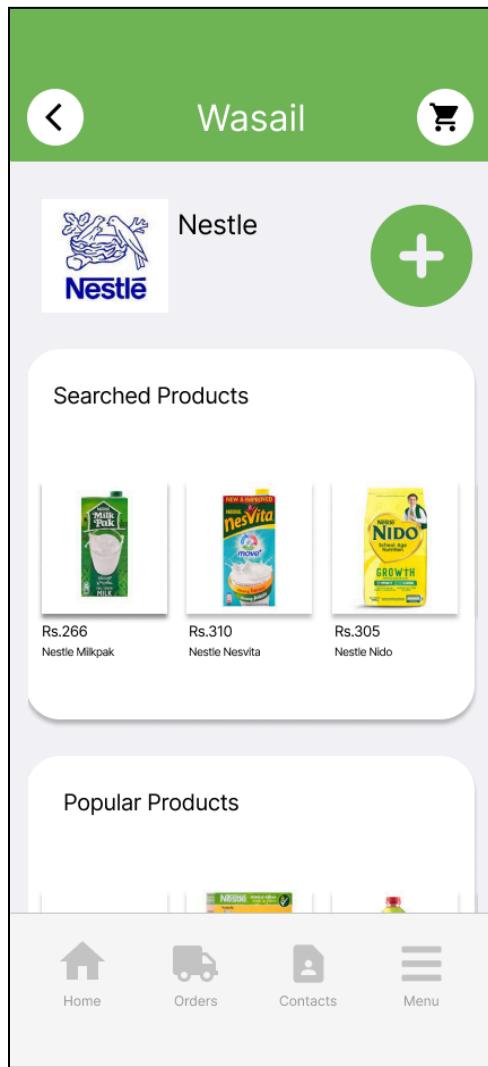


Figure 2.6.2.6 View Vendor Profile Prototype

FR2.15: Add Vendor to Vendor List

- **Description:** The system should allow the user to add the vendor to their vendor list (Figure 2.6.2.7).
- **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.

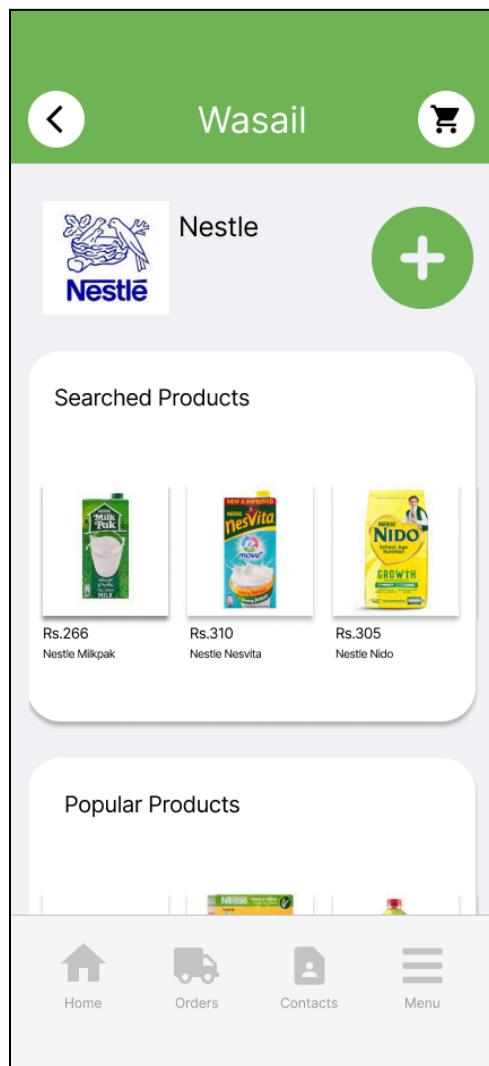


Figure 2.6.2.7 Add Vendor to Vendor List Prototype

FR2.16: Contact Vendor

- **Description:** The system should allow the user to contact the vendor (Figure 2.6.2.8).
- **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.

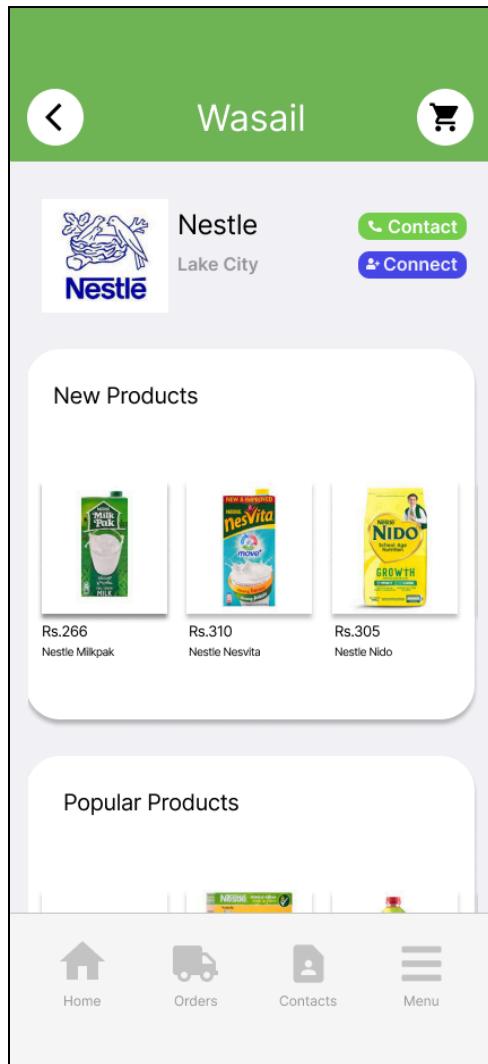


Figure 2.6.2.8 Contact Vendor Prototype

FR2.17: 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 2.6.2.9).
- **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.

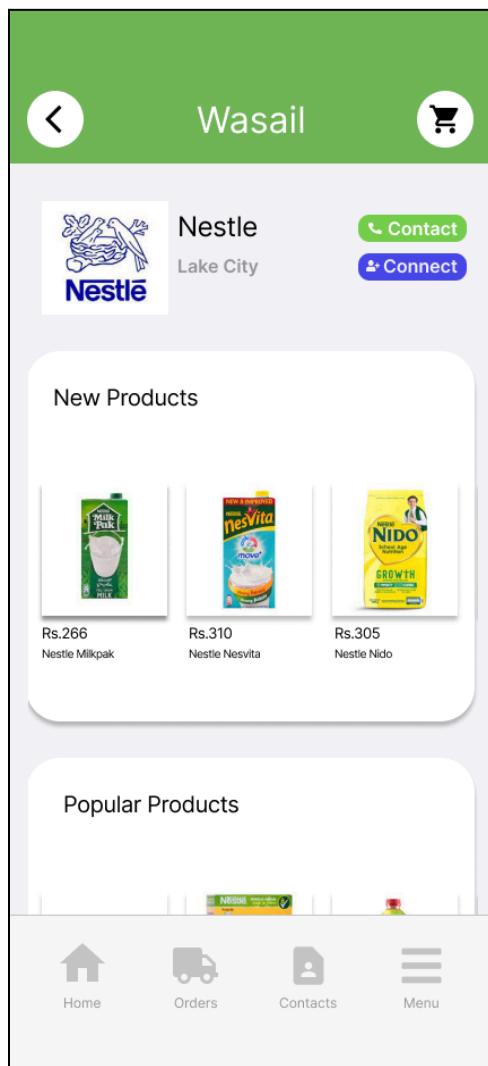


Figure 2.6.2.9 View Products on the Vendor's Profile Prototype

FR2.18: View Searched Product

- **Description:** The system should allow the user to view the searched product on the vendor's profile (Figure 2.6.2.10).
- **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.

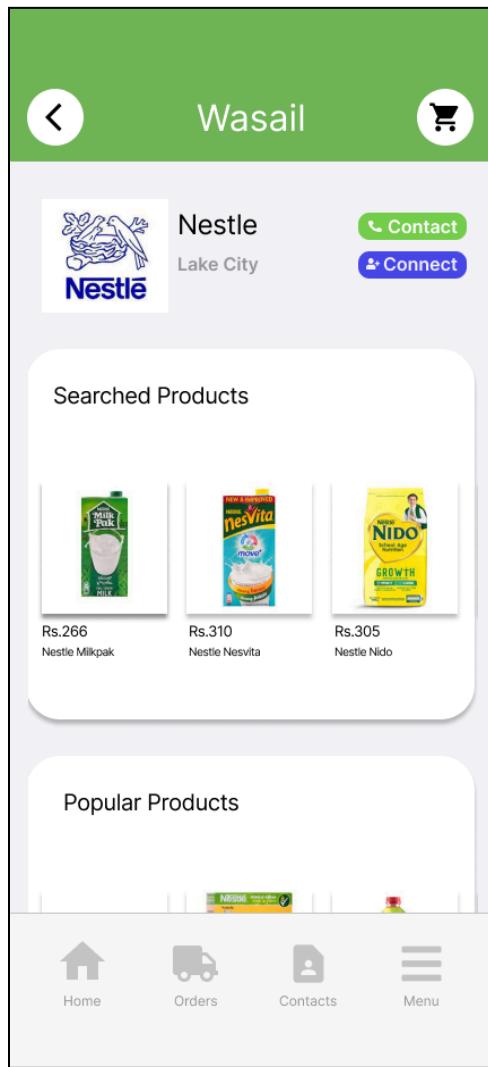


Figure 2.6.2.10 View Searched Product Prototype

FR2.19: Select Products

- **Description:** The system should allow the user to select the product that they want to order (Figure 2.6.2.11).
- **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.

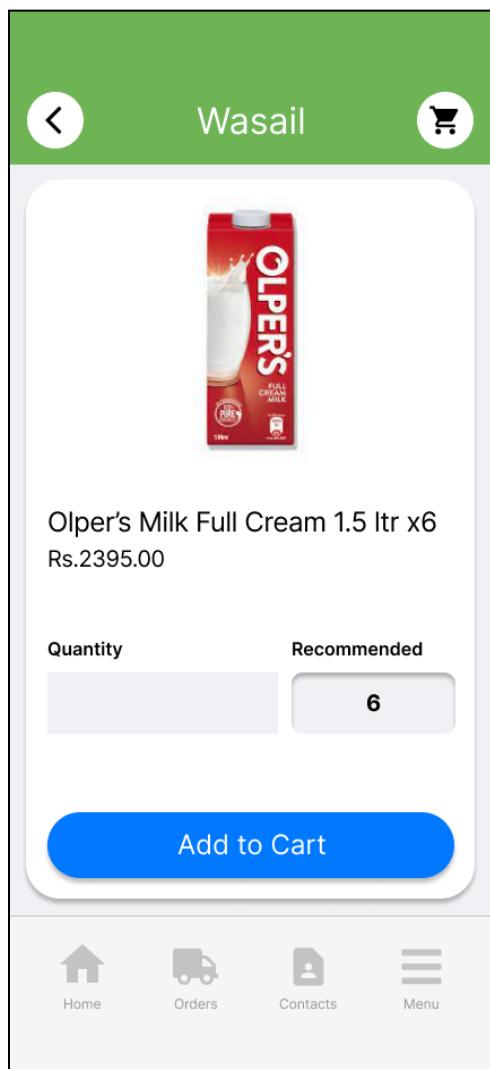


Figure 2.6.2.11 Select Products Prototype

FR2.20: Order Recommendation

Description: The system should allow the user to view the recommended amount to order that would ensure maximum profit (Figure 2.6.2.12).

- **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 and oil prices.

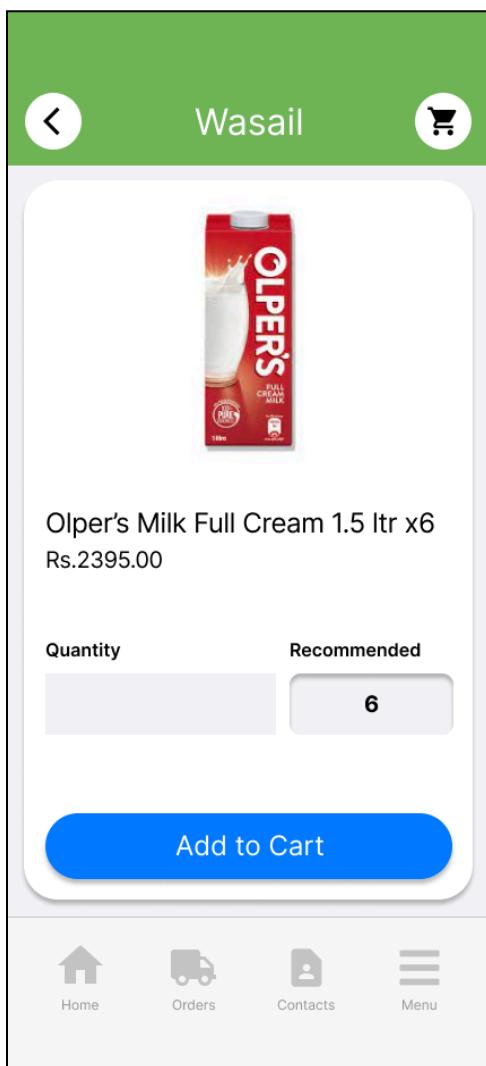


Figure 2.6.2.12 Order Recommendation Prototype

FR2.21: Quantity Selection

- **Description:** The system should allow the user to add their own amount to order (Figure 2.6.2.13).
- **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.

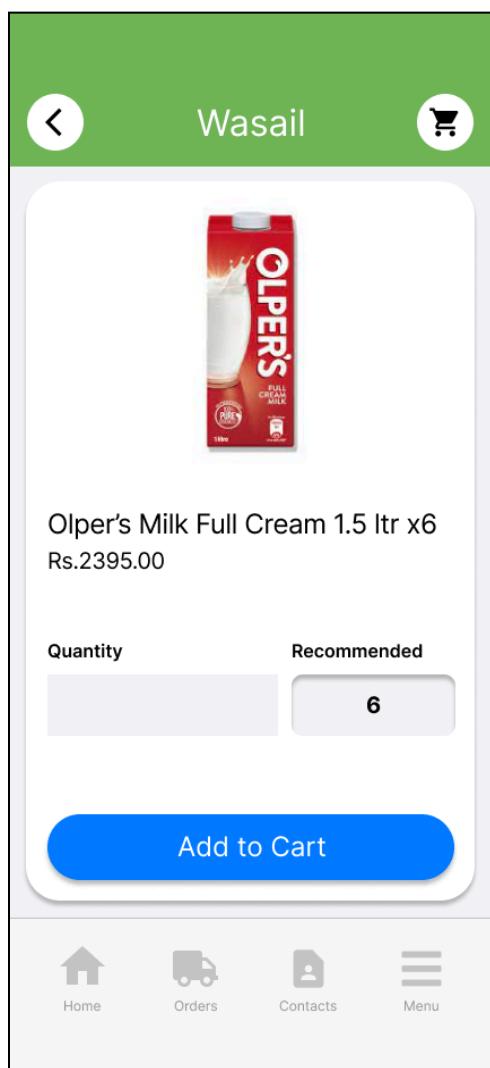


Figure 2.6.2.13 Quantity Selection Prototype

FR2.22: Add to Cart

- **Description:** The system should allow the user to add products to their cart in order to place an order (Figure 2.6.2.14).
- **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.

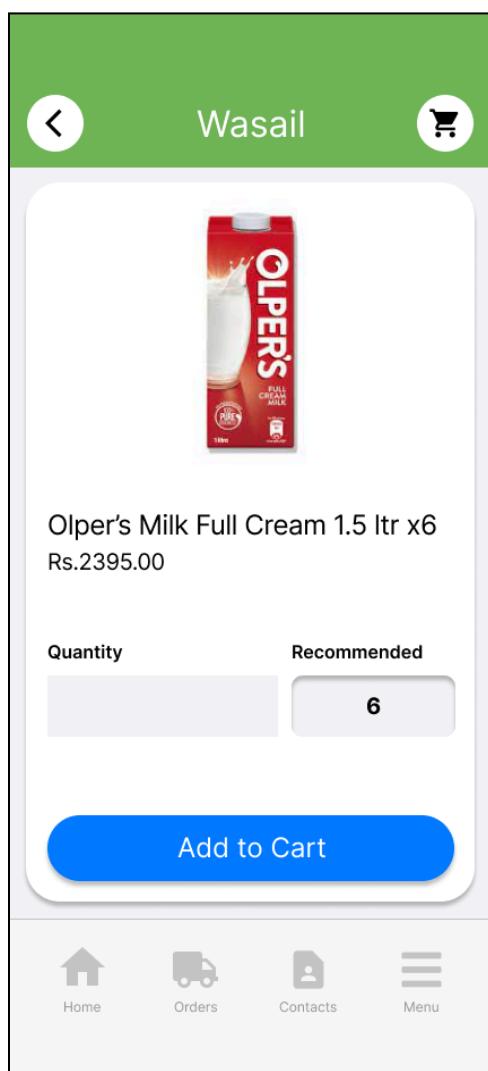


Figure 2.6.2.14 Add to Cart Prototype

FR2.23: View Cart

- **Description:** The system should allow the user to view their cart (Figure 2.6.2.15).
- **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.

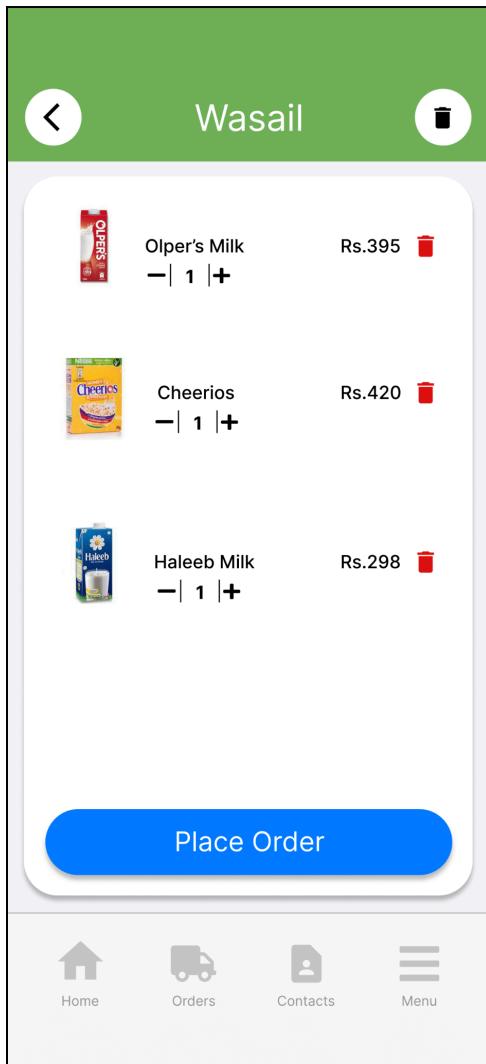


Figure 2.6.2.15 View Cart Prototype

FR2.24: Update Product Quantity in Cart

- **Description:** The system should allow the user to update the quantity of the product added in their cart (Figure 2.6.2.16).
- **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.

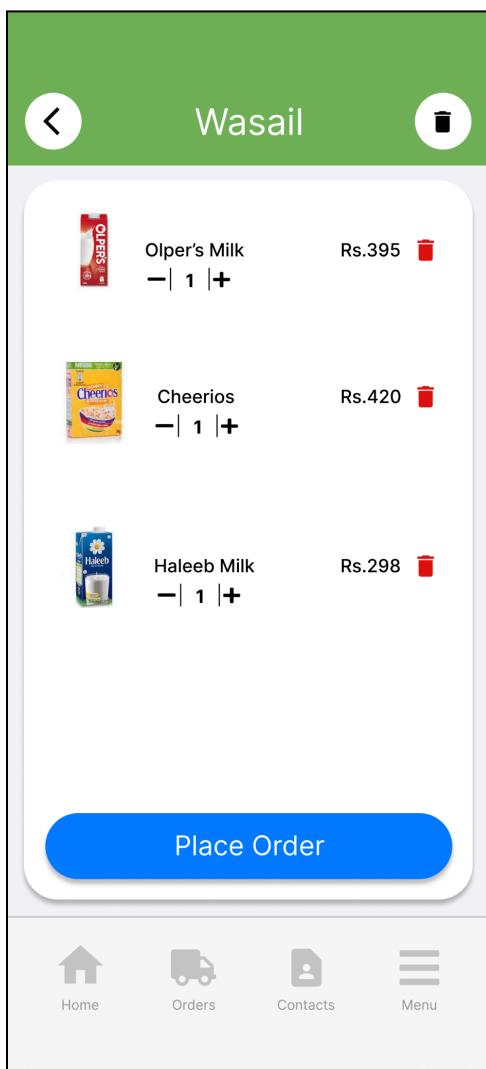


Figure 2.6.2.16 Update Product Quantity in Cart Prototype

FR2.25: Remove Product from Cart

- **Description:** The system should allow the user to remove one specific product (Figure 2.6.2.17).
- **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.

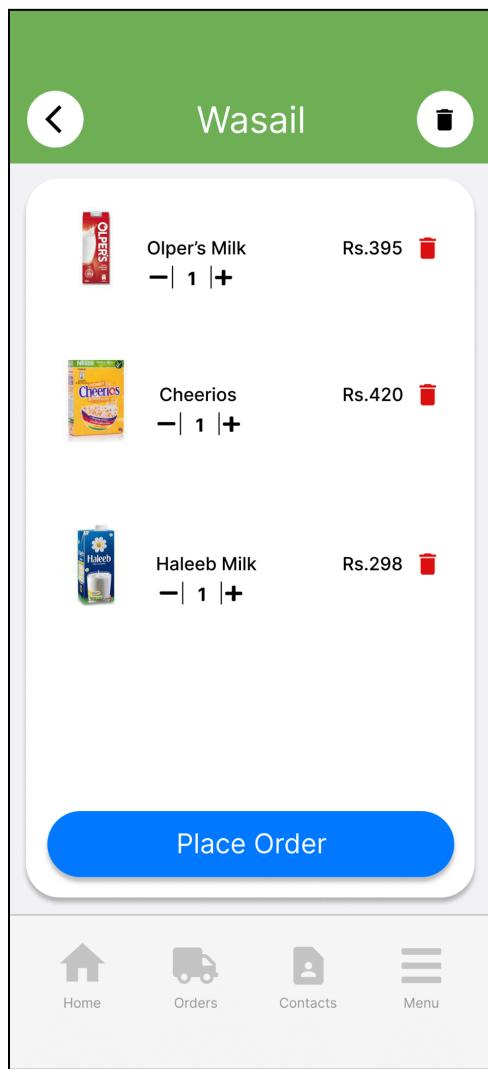


Figure 2.6.2.17 Remove Product Prototype

FR2.26: Clear Cart

- **Description:** The system should allow the user to clear the cart in order to remove all the products from it (Figure 2.6.2.18).
- **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.

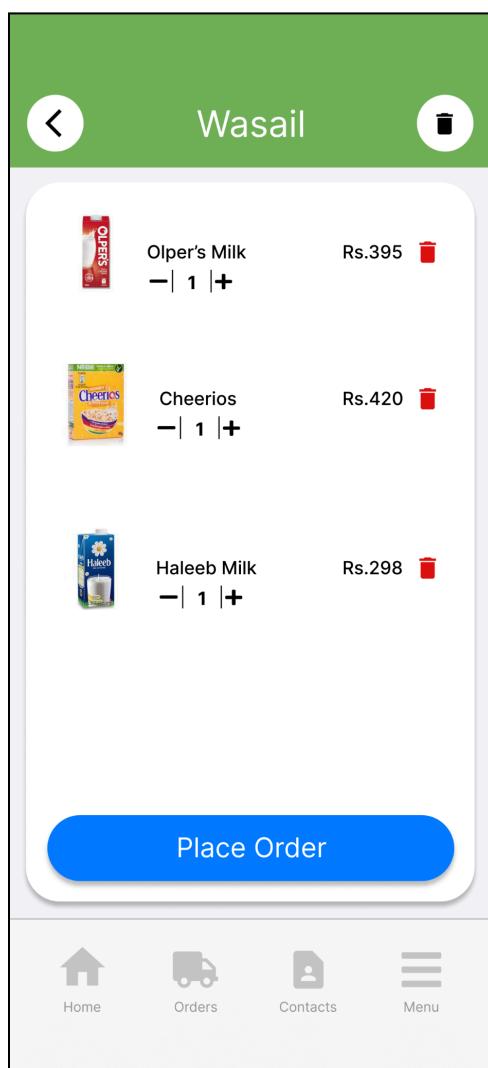


Figure 2.6.2.18 Clear Cart Prototype

FR2.27: Order Placement

- **Description:** The system should allow the user to place the order (Figure 2.6.2.19).
- **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.

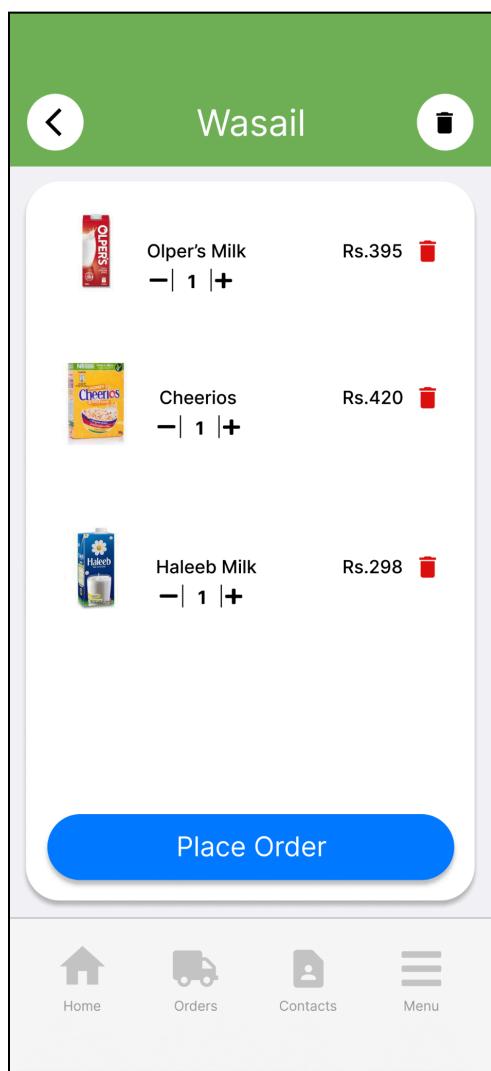


Figure 2.6.2.19 Order Placement Prototype

FR2.28: View Orders

- **Description:** The system should allow the users to view the current orders that they have placed (Figure 2.6.2.20).
- **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.

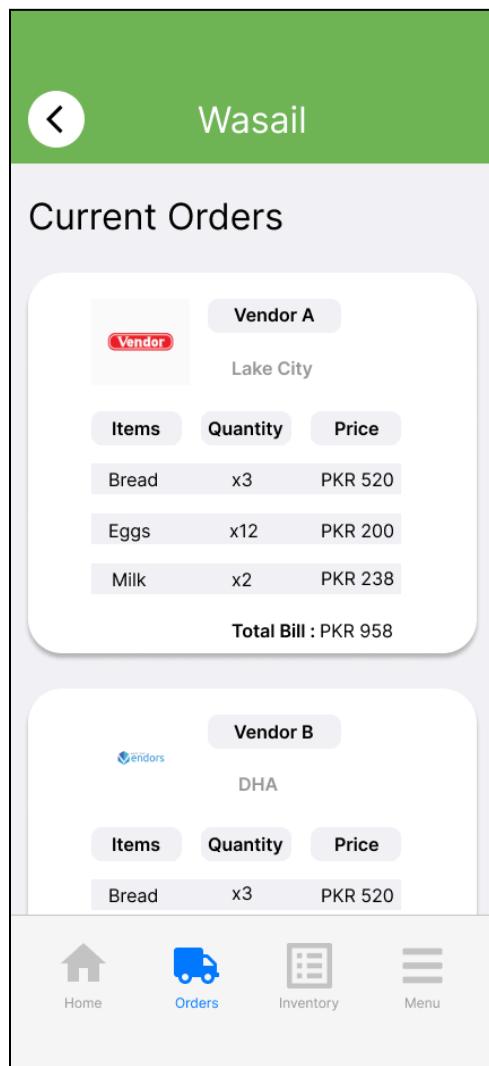


Figure 2.6.2.20 View Order Prototype

FR2.29: Order Tracking

- **Description:** The system should allow the user to track the order (Figure 2.6.2.21).
- **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

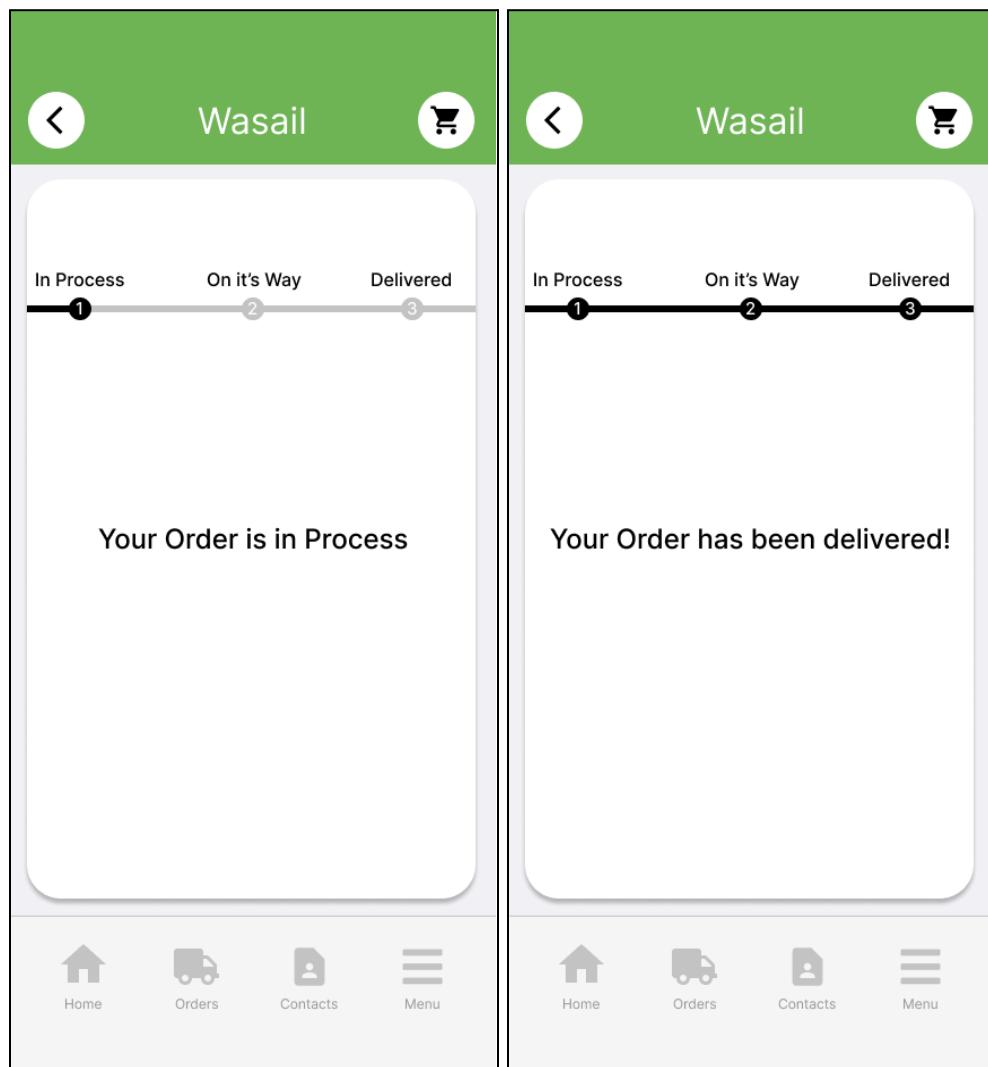


Figure 2.6.2.21 Order Tracking Prototype

FR2.30: View Vendor List

- **Description:** The system should allow the user to view the vendors in the vendor list (Figure 2.6.2.22).
- **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.

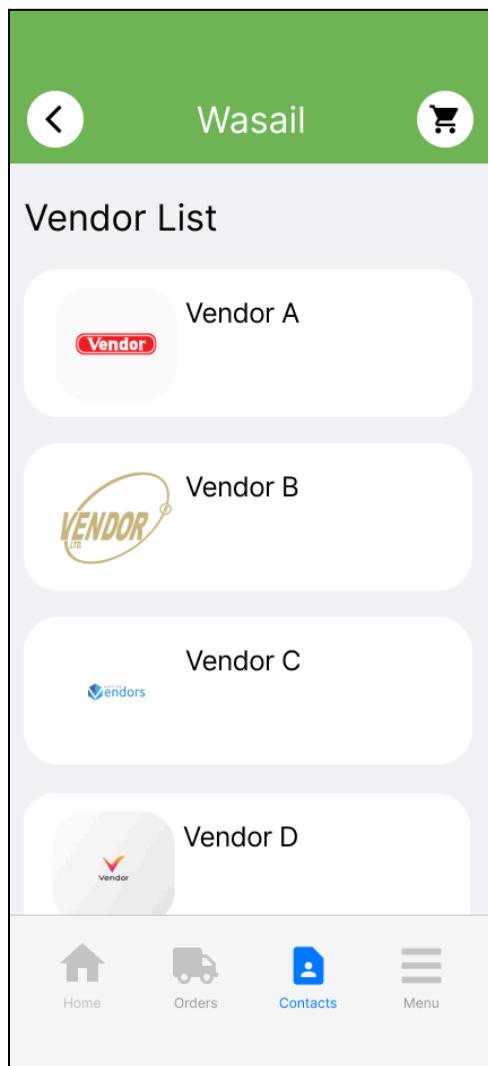


Figure 2.6.2.22 View Vendor List Prototype

FR2.31: View Order History

- **Description:** The system should allow the user to view their previous orders (Figure 2.6.2.23).
- **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.

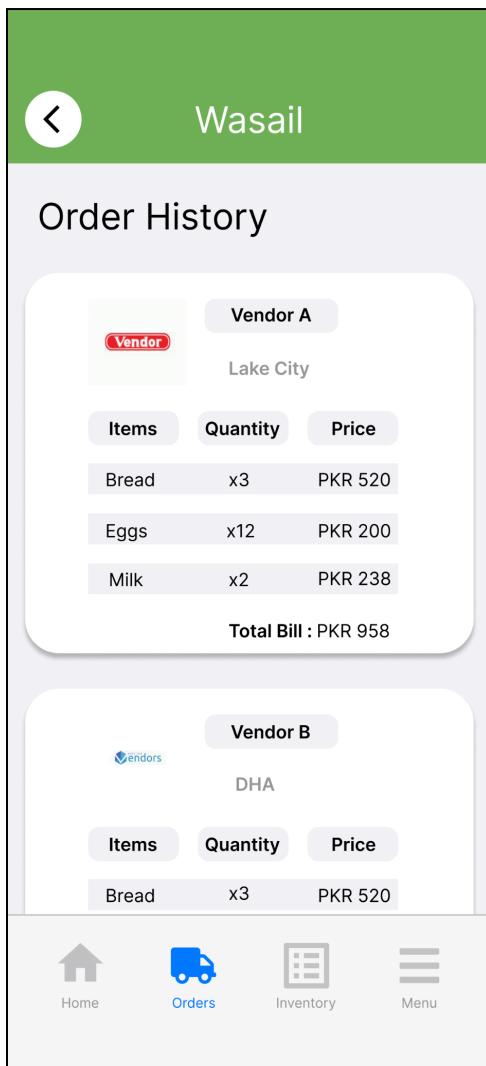


Figure 2.6.2.23 View Order History Prototype

FR2.32: Edit Profile

- **Description:** The system should allow the user to edit their profile details (Figure 2.6.2.24).
- **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.

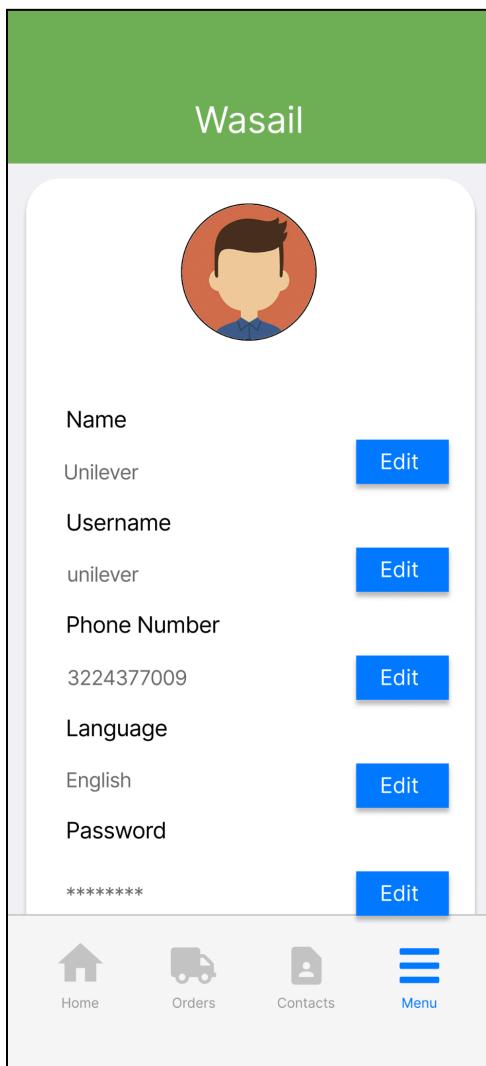
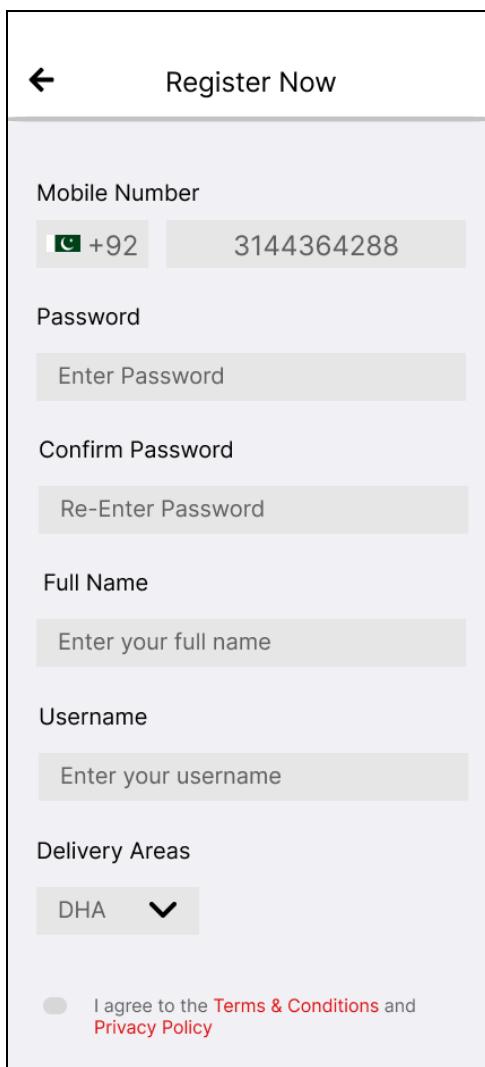


Figure 2.6.2.24 Edit Profile Prototype

2.6.3 Vendor (FR3)

FR3.9: Vendor Registration

- **Description:** The system should allow the user to enter their account details (Figure 2.6.3.1).
- **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.



The form is titled "Register Now" with a back arrow icon. It contains the following fields:

- Mobile Number: A field showing "+92 3144364288".
- Password: A field labeled "Enter Password".
- Confirm Password: A field labeled "Re-Enter Password".
- Full Name: A field labeled "Enter your full name".
- Username: A field labeled "Enter your username".
- Delivery Areas: A dropdown menu currently showing "DHA".
- A checkbox followed by the text "I agree to the [Terms & Conditions](#) and [Privacy Policy](#)".

Figure 2.6.3.1 Vendor Registration Prototype

FR3.10: Valid Password

- **Description:** The system should allow the user to enter a password (Figure 2.6.3.2).
- **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.

The screenshot shows a mobile application interface for registration. At the top, there is a back arrow icon and the text "Register Now". Below this, there are several input fields and labels:

- Mobile Number:** A field containing a green phone icon, "+92", and the number "3144364288".
- Password:** A field containing six asterisks ("*****"). Below it, a red error message states: "The password should be 8 characters long and should have at least one special character".
- Confirm Password:** A field labeled "Re-Enter Password".
- Full Name:** A field labeled "Enter your full name".
- Username:** A field labeled "Enter your username".
- Delivery Areas:** A dropdown menu currently showing "DHA".
- Agreement:** A checkbox followed by the text "I agree to the [Terms & Conditions](#) and [Privacy Policy](#)".

Figure 2.6.3.2 Valid Password Prototype

FR3.11: Username Exists

- **Description:** The system should check if the username exists in the database (Figure 2.6.3.3).
- **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.

The screenshot shows a mobile application's registration screen titled "Register Now". The form includes fields for "Mobile Number" (with an icon for +92 and the number 3144364288), "Password" (with placeholder "Enter Password"), "Confirm Password" (with placeholder "Re-Enter Password"), "Full Name" (with placeholder "Enter your full name"), "Username" (with the entry "faty" and a red error message "Username already exists."), and "Delivery Areas" (with "DHA" selected from a dropdown menu). At the bottom, there is a checkbox followed by the text "I agree to the [Terms & Conditions](#) and [Privacy Policy](#)".

Figure 2.6.3.3 Username Exists Prototype

FR3.12: Search Product in Inventory

- **Description:** The system should allow the user to search for a product from their inventory (Figure 2.6.3.4).
- **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.

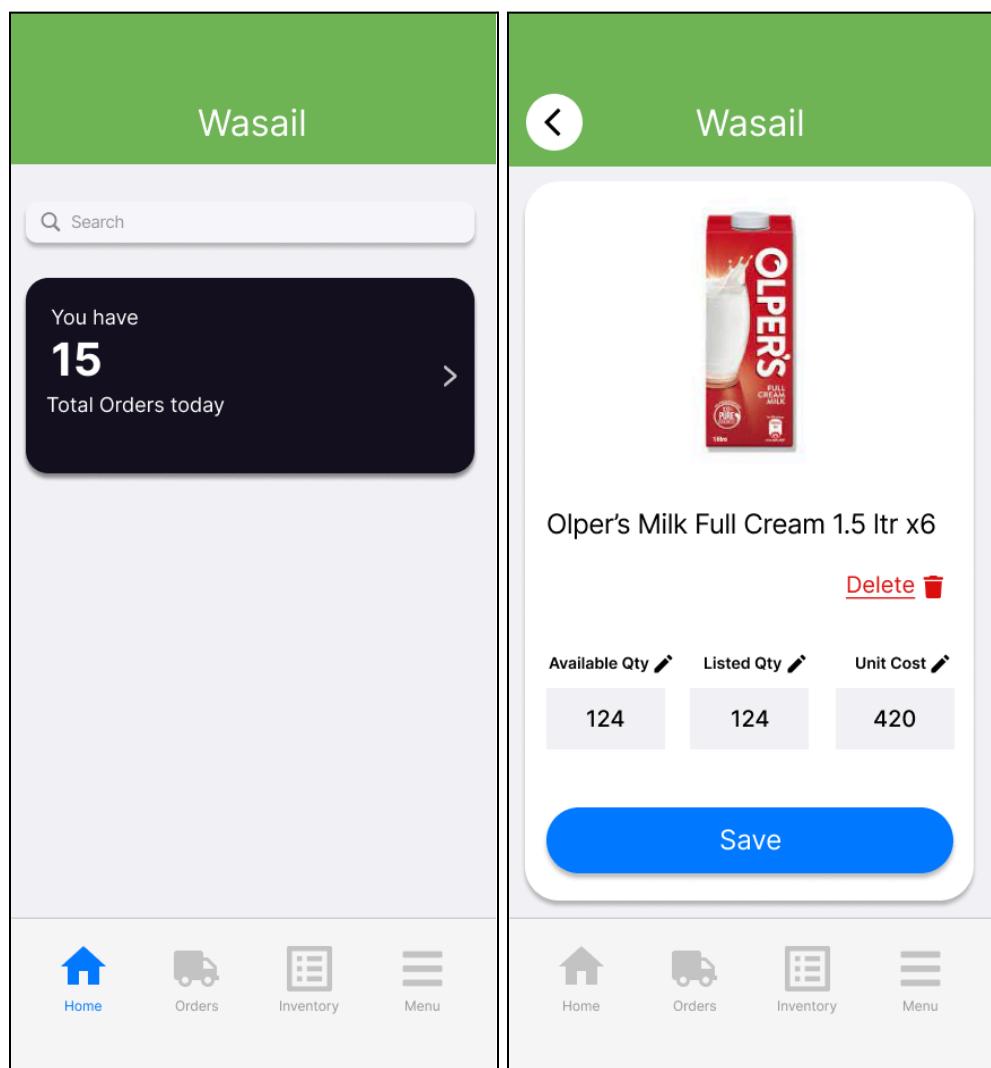


Figure 2.6.3.4 Search Product in Inventory Prototype

FR3.13: Add Product to Inventory

- **Description:** The system should allow the user to add a new product to their inventory (Figure 2.6.3.5).
- **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.

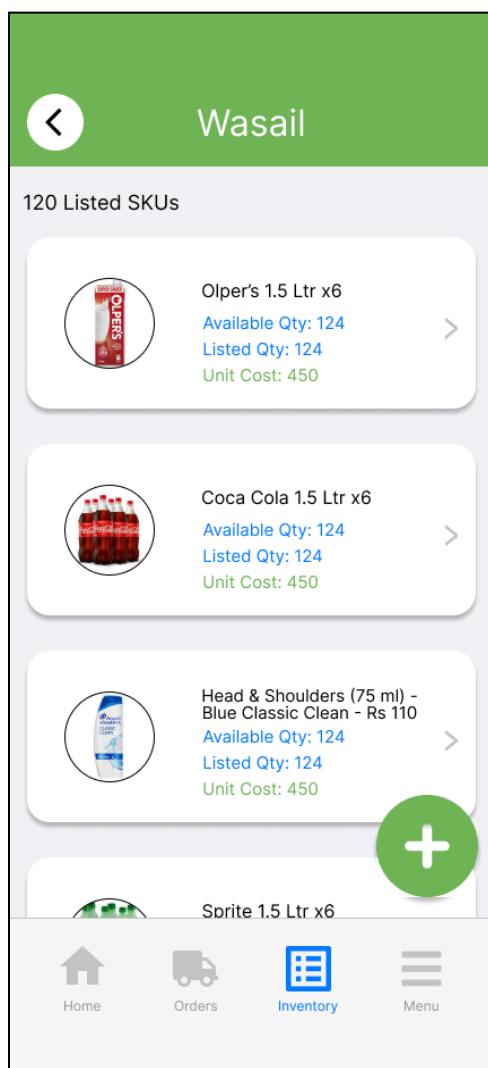


Figure 2.6.3.5 Add Product to Inventory Prototype

FR3.14: All Products Search

- **Description:** The system should allow the user to search all products from the system's product listing (Figure 2.6.3.6).
- **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.

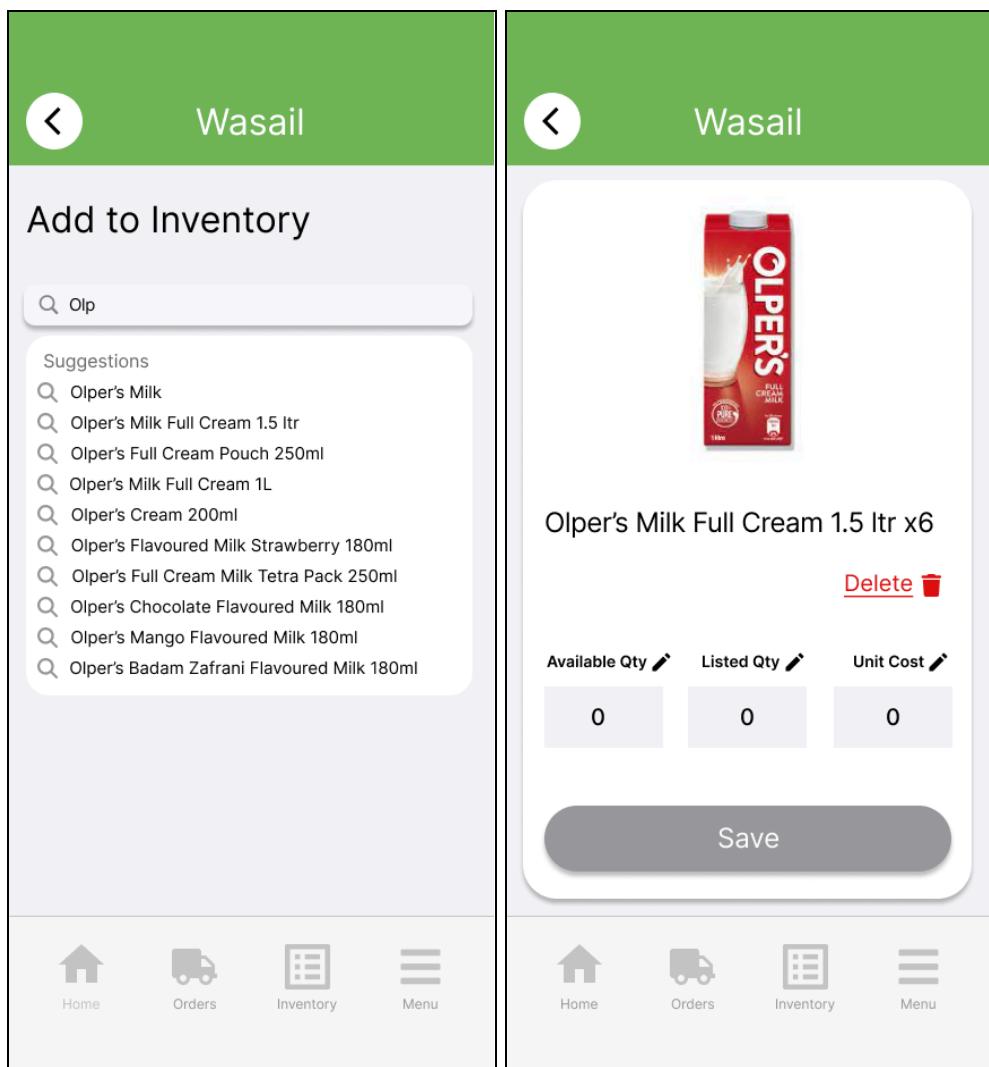


Figure 2.6.3.6 All Products Search Prototype

FR3.15: Restrict Product Duplication

- **Description:** The system should restrict duplication of products (Figure 2.6.3.7).
- **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.

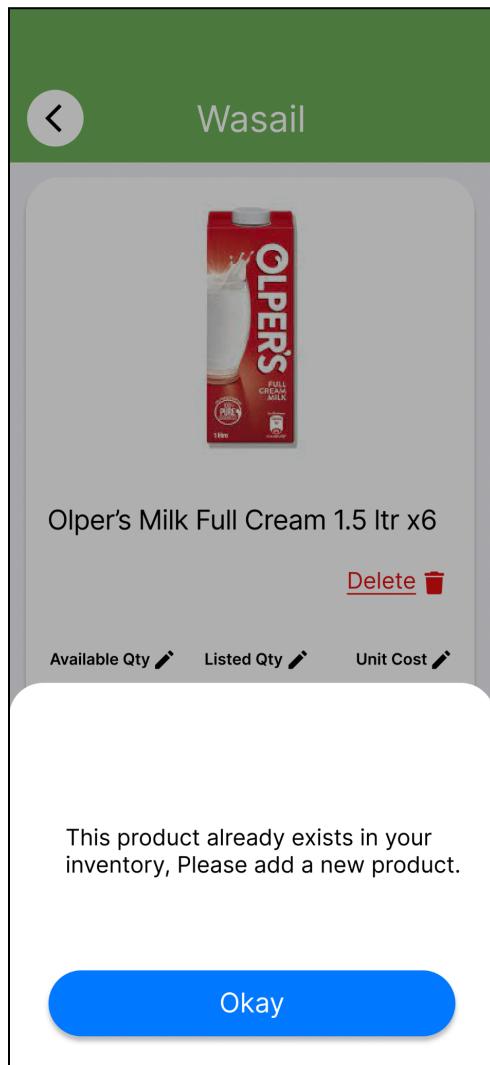


Figure 2.6.3.7 Restrict Product Duplication Prototype

FR3.16: Remove Product

- **Description:** The system should allow the user to remove a product from the inventory (Figure 2.6.3.8).
- **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.

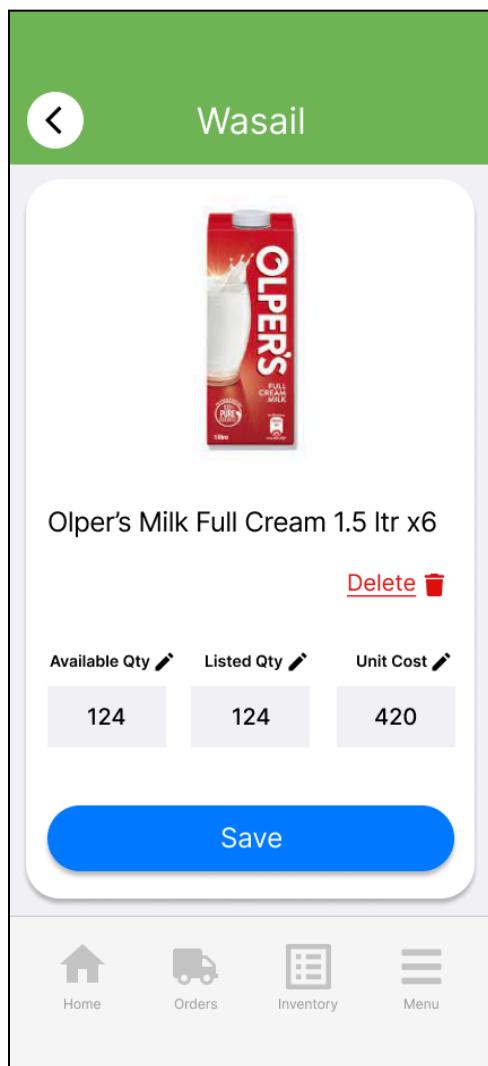


Figure 2.6.3.8 Remove Product Prototype

FR3.17: Edit Details of Product

- **Description:** The system should allow the user to edit the details of products already in their inventory (Figure 2.6.3.9).
- **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.

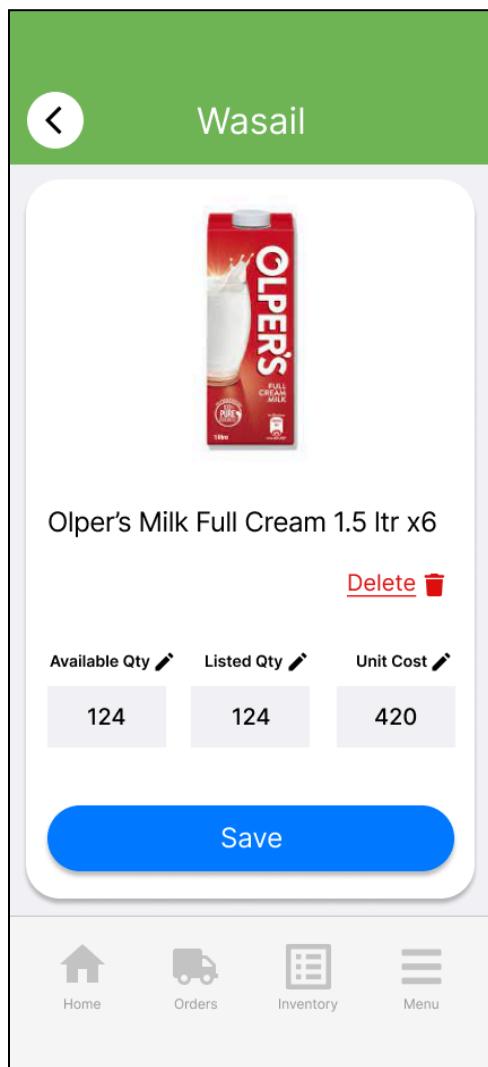


Figure 2.6.3.9 Edit Details of Product Prototype

FR3.18: View Inventory

- **Description:** The system should allow the user to view their inventory i.e. product listings (Figure 2.6.3.10).
- **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.

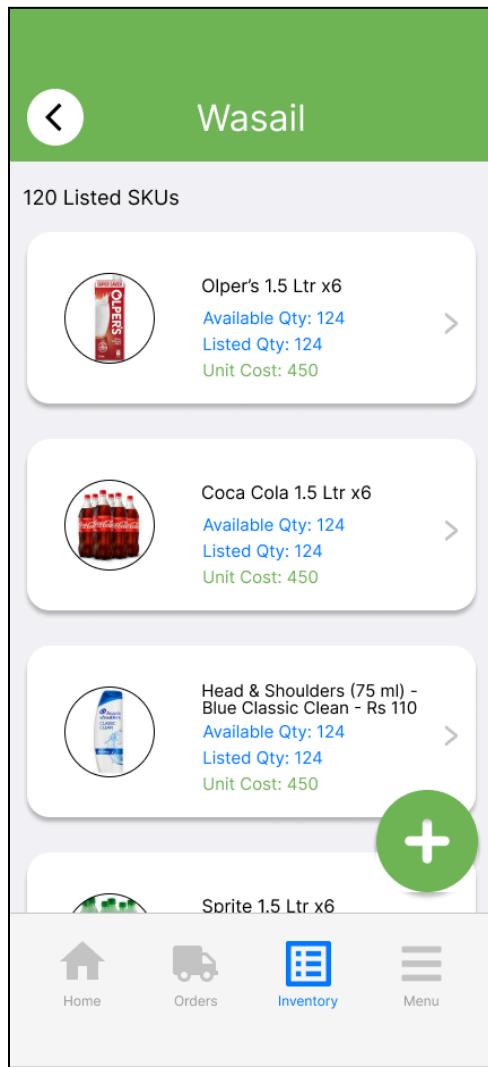


Figure 2.6.3.10 View Inventory Prototype

FR3.19: View Current Orders

- **Description:** The system should allow the user to view the total number of current orders from all grocery stores (Figure 2.6.3.11).
- **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.

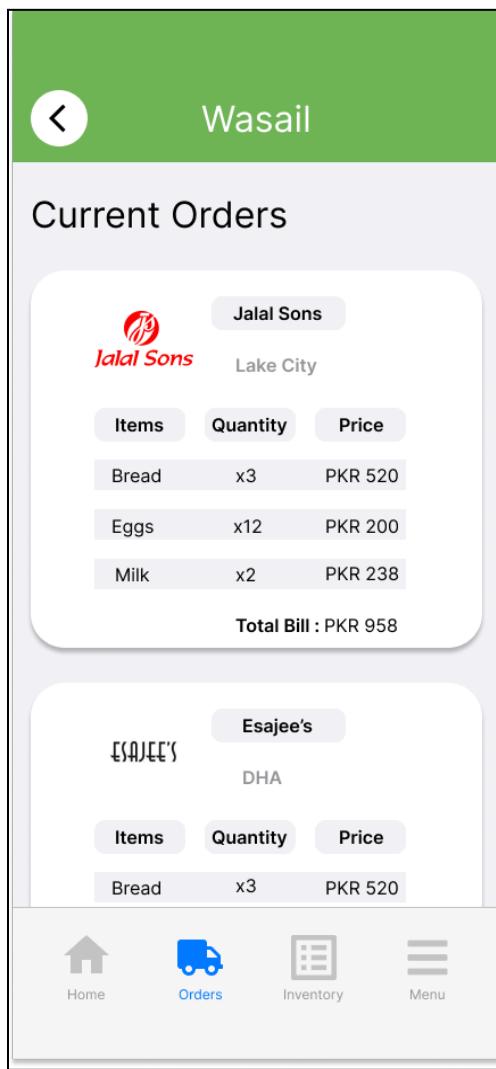


Figure 2.6.3.11 View Current Orders Prototype

FR3.20: View Grocery Stores List

- **Description:** The system should allow the user to view the grocery store list (Figure 2.6.3.12).
- **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.

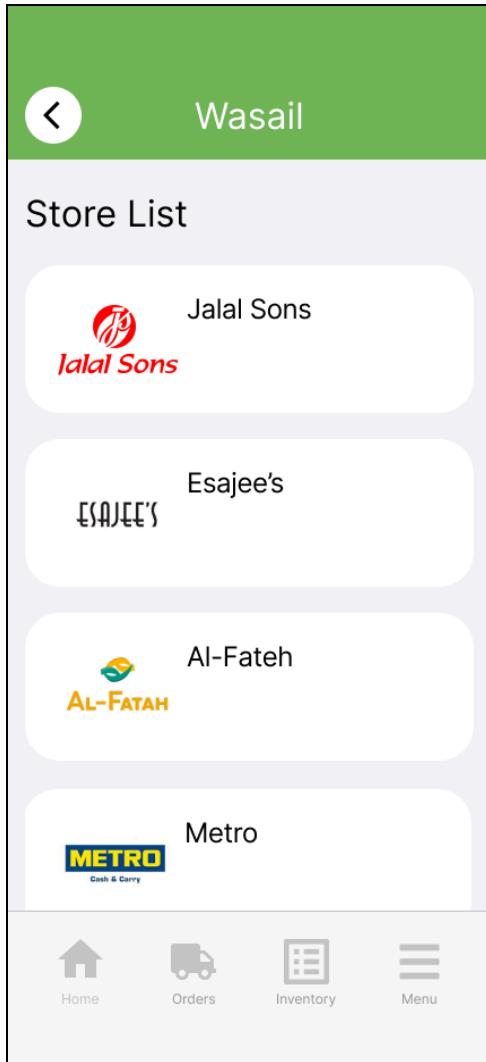


Figure 2.6.3.12 View Grocery Stores List Prototype

FR3.21: View Grocery Store Profile

- **Description:** The system should allow the user to view the grocery store profile (Figure 2.6.3.13).
- **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.

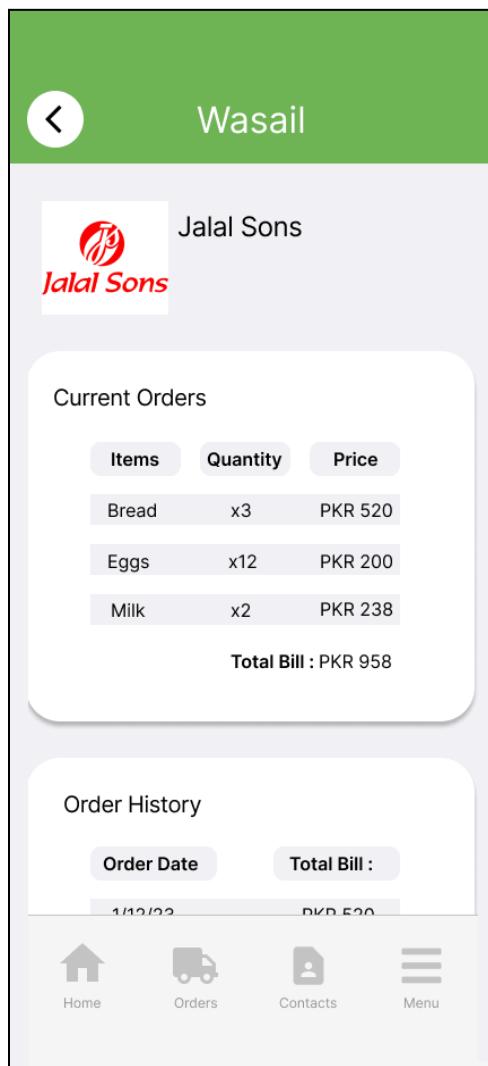


Figure 2.6.3.13 View Grocery Store Profile Prototype

FR3.22: View Grocery Store Current Order

- **Description:** The system should allow the user to view the current order placed by the grocery store (Figure 2.6.3.14).
- **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.

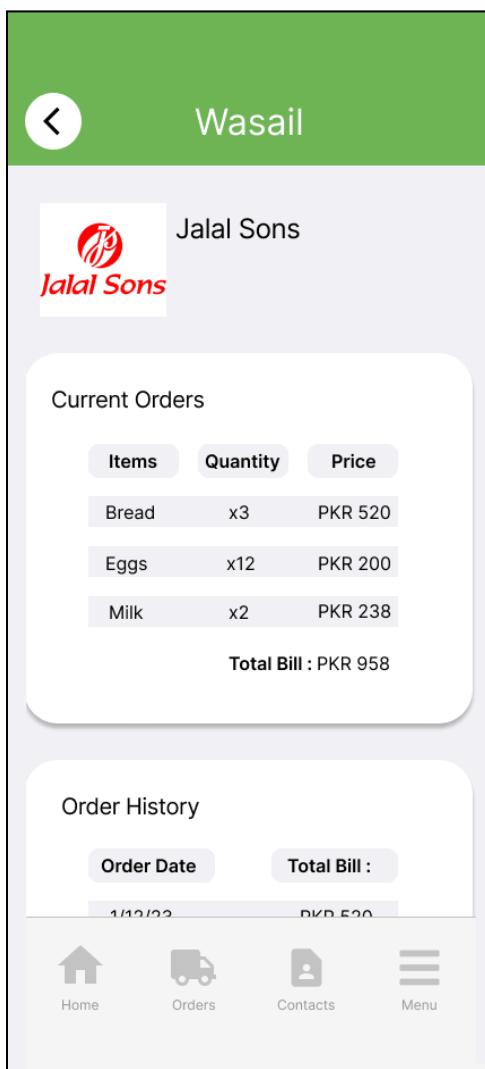


Figure 2.6.3.14 View Grocery Store Current Order Prototype

FR3.23: View Orders History

- **Description:** The system should allow the user to view the orders they have already completed delivering (Figure 2.6.3.15).
- **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.

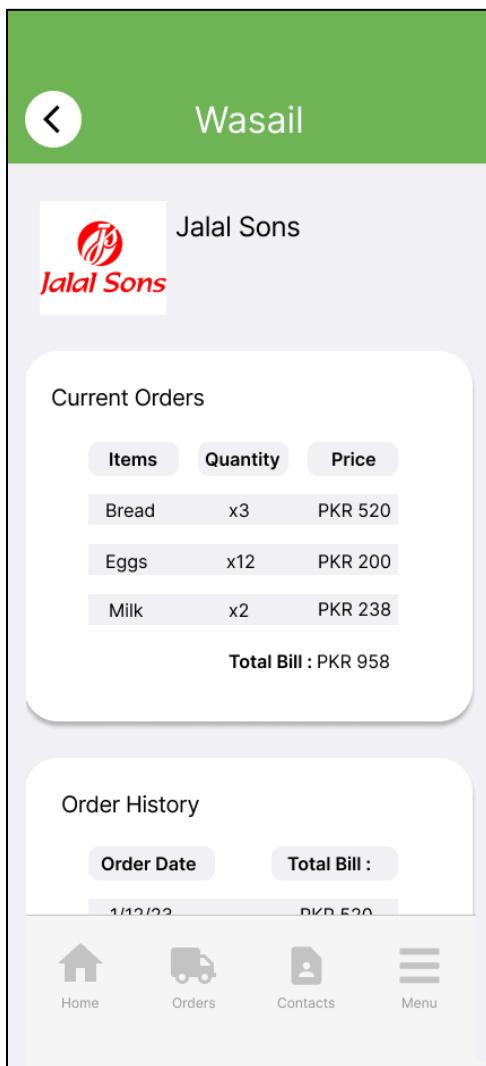


Figure 2.6.3.15 View Orders History Prototype

FR3.24: Order Dispatch Tracking

- **Description:** The system should allow the user to track the dispatched order (Figure 2.6.3.16).
- **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.

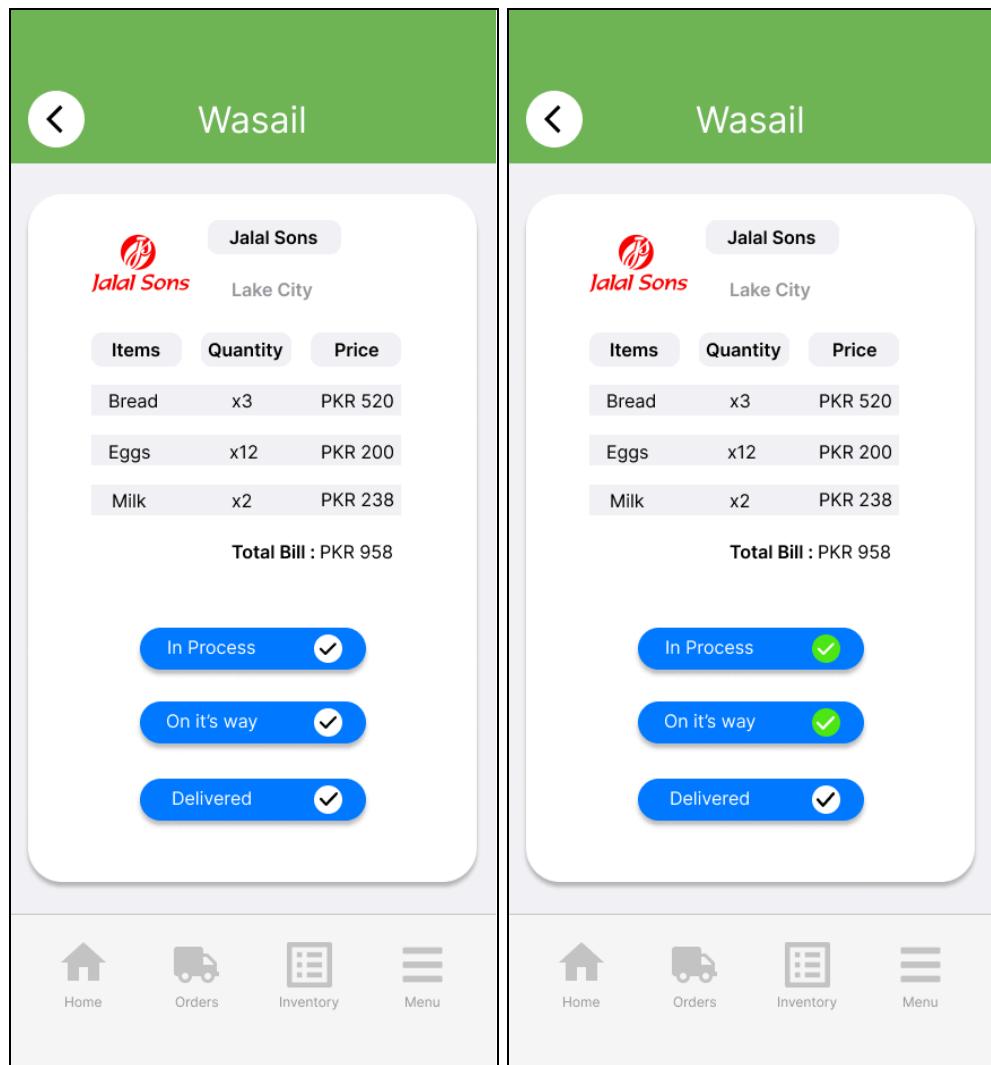


Figure 2.6.3.16 Order Dispatch Tracking Prototype

FR3.25: Edit Profile

- **Description:** The system should allow the user to edit their profile details (Figure 2.6.3.17).
- **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

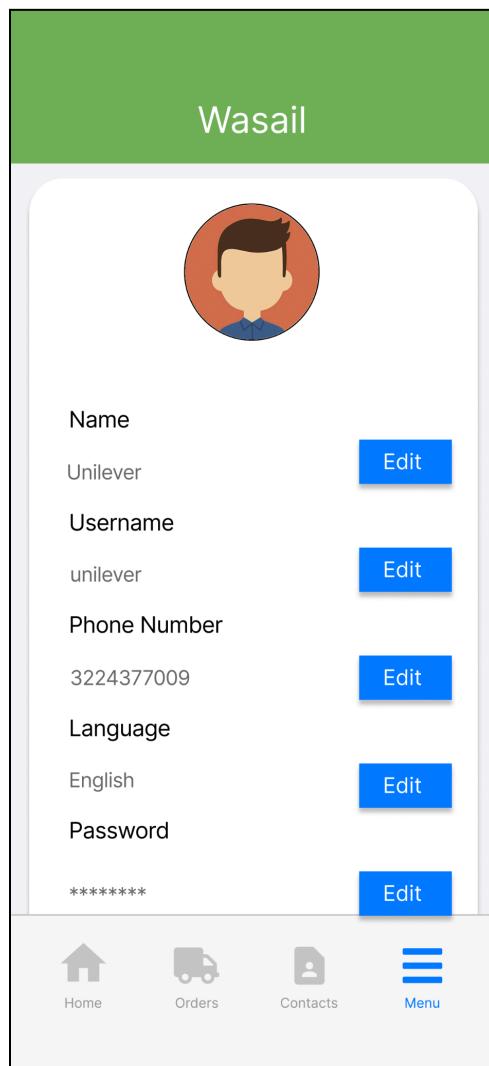
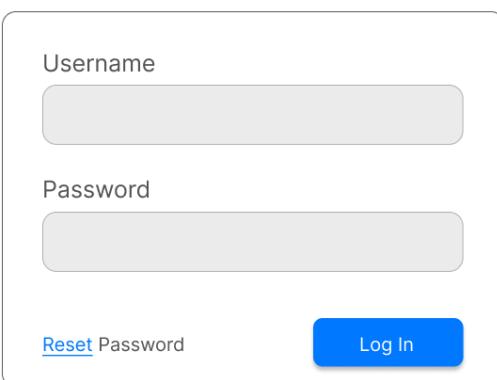


Figure 2.6.3.17 Edit Profile Prototype

2.6.4 Admin Portal (FR4)

FR4.1: Login

- **Description:** The system should allow the user to login using their credentials (Figure 2.6.4.1).
- **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.



The image shows a user login prototype for a system named WASAIL. At the top center, the word "WASAIL" is displayed in green capital letters. Below it is a rectangular form with rounded corners. The form contains two input fields: one labeled "Username" and another labeled "Password", both represented by light gray rectangular boxes. At the bottom left of the form is a blue link labeled "Reset Password". At the bottom right is a blue button labeled "Log In".

Figure 2.6.4.1 User Login Prototype

FR4.2: Add New User

- **Description:** The system should allow the existing user to add a new user (Figure 2.6.4.2).
- **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.

The screenshot shows the WASAIL application interface. On the left, there is a vertical sidebar with icons and labels for User Management, Grocery Management, Vendor Management, ML Configuration, Analytics, and Content Management. The main area is titled 'Users' and has a 'Add New' button. A success message 'New User Created. [Edit User](#)' is displayed. Below this is a search bar and a table listing users. The table has columns for Username, Name, and Email. The data is as follows:

Username	Name	Email
fatima	Fatima Ali	fatimaalitmizi12@gmail.com
irtaza	Irtaza Ahmad	yrrebeere@gmail.com
fizza	Fizza Adeel	fizza.adeel19@gmail.com
malaika	Malaika Sultan	malaikasultant@gmail.com

Figure 2.6.4.2 Add New User Prototype

FR4.3: Add New User Details

- **Description:** The system should allow the existing user to add details of the new user (Figure 2.6.4.3).
- **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.

The screenshot shows a web-based application interface titled 'WASAIL'. On the left, there is a vertical navigation menu with several items: 'User Management' (selected), 'Grocery Management', 'Vendor Management', 'ML Configuration', 'Analytics', and 'Content Management'. The main content area has a title 'Add New User' and a subtitle 'Create a brand new user and add them to this site.' Below this, there are four input fields: 'Username (required)' containing 'fatima', 'Email (required)' containing 'fatimaalitirmizi12@gmail.com', 'First Name' containing 'Fatima', and 'Last Name' containing 'Ali'. There is also a 'Password' field which is currently empty. At the bottom left is a blue button labeled 'Add New User'. To the right of the 'Add New User' button is a checkbox labeled 'Send User Notification' with the sub-instruction 'Send the new user an email about their account.'

Figure 2.6.4.3 Add New User Details Prototype

FR4.4: Edit User Profile

- **Description:** The system should allow the user to edit their profile (Figure 2.6.4.4).
- **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.

The screenshot shows the WASAIL application interface. On the left, there is a vertical sidebar with icons and labels for User Management, Grocery Management, Vendor Management, ML Configuration, Analytics, and Content Management. The main area is titled "Users" and has a "Add New" button. A success message "New User Created. [Edit User](#)" is displayed. Below this, there is a search bar and a table listing four users:

Username	Name	Email
fatima	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 2.6.4.4 Edit User Profile Prototype

FR4.5: Delete User Profile

- **Description:** The system should allow the user to delete their profile (Figure 2.6.4.5).
- **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.

The screenshot shows the WASAIL application interface. On the left, there is a sidebar with various management options: User Management (selected), Grocery Management, Vendor Management, ML Configuration, Analytics, and Content Management. The main area is titled "Users" and has a "Add New" button. A success message "New User Created. [Edit User](#)" is displayed. Below the message is a search bar and a "Search Users" button. A table lists four users with columns for Username, Name, and Email. Each user row includes "Edit" and "Delete" links.

Username	Name	Email
fatima	Fatima Ali	fatima.alitirmizi12@gmail.com
irtaza	Irtaza Ahmad	yrrebeere@gmail.com
fizza	Fizza Adeel	fizza.adeel19@gmail.com
malaika	Malaika Sultan	malaikasultant@gmail.com

Figure 2.6.4.5 Delete User Profile Prototype

FR4.6: Search User

- **Description:** The system should allow the user to search the admin on the admin page (Figure 2.6.4.6).
- **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.

The screenshot shows the WASAIL admin interface. On the left, there is a sidebar with the following menu items:

- User Management
- Grocery Management
- Vendor Management
- ML Configuration
- Analytics
- Content Management

The main area is titled "Users" and contains an "Add New" button. Below this, there is a search bar with the placeholder "Search Users" and a text input field containing "fatima". A table displays user information:

Username	Name	Email
fatima	Fatima Ali	fatimaalitirmizi12@gmail.com

Underneath the table, there are links for "Edit" and "Delete".

Figure 2.6.4.6 Search User Prototype

FR4.7: View Grocery Store's Profile

- **Description:** The system should allow the user to view the grocery store's profile (Figure 2.6.4.7).
- **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.

The screenshot shows a web-based application interface titled "Grocery Management". On the left, there is a vertical sidebar with the "WASAIL" logo at the top. Below it, several menu items are listed with corresponding icons: "User Management" (person icon), "Grocery Management" (store icon, highlighted in grey), "Vendor Management" (truck icon), "ML Configuration" (gears icon), "Analytics" (chart icon), and "Content Management" (list icon). To the right of the sidebar, there is a search bar with the placeholder "Search Users" and a "Search" button. The main content area displays a table with four rows, each representing a grocery store. The columns are labeled "Icons", "Name", "Phone Number", and "Actions". The data is as follows:

Icons	Name	Phone Number	Actions
	Jalal Sons	0300-9876543	Delete View
	Esajee's	0312-8766542	Delete View
	Al-Fateh	0314-6247966	Delete View
	Metro	0305-7654836	Delete View

Figure 2.6.4.7 View Grocery Store's Profile Prototype

FR4.8: Disable Grocery Store's Profile

- **Description:** The system should allow the user to delete the grocery store's profile (Figure 2.6.4.8).
- **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.

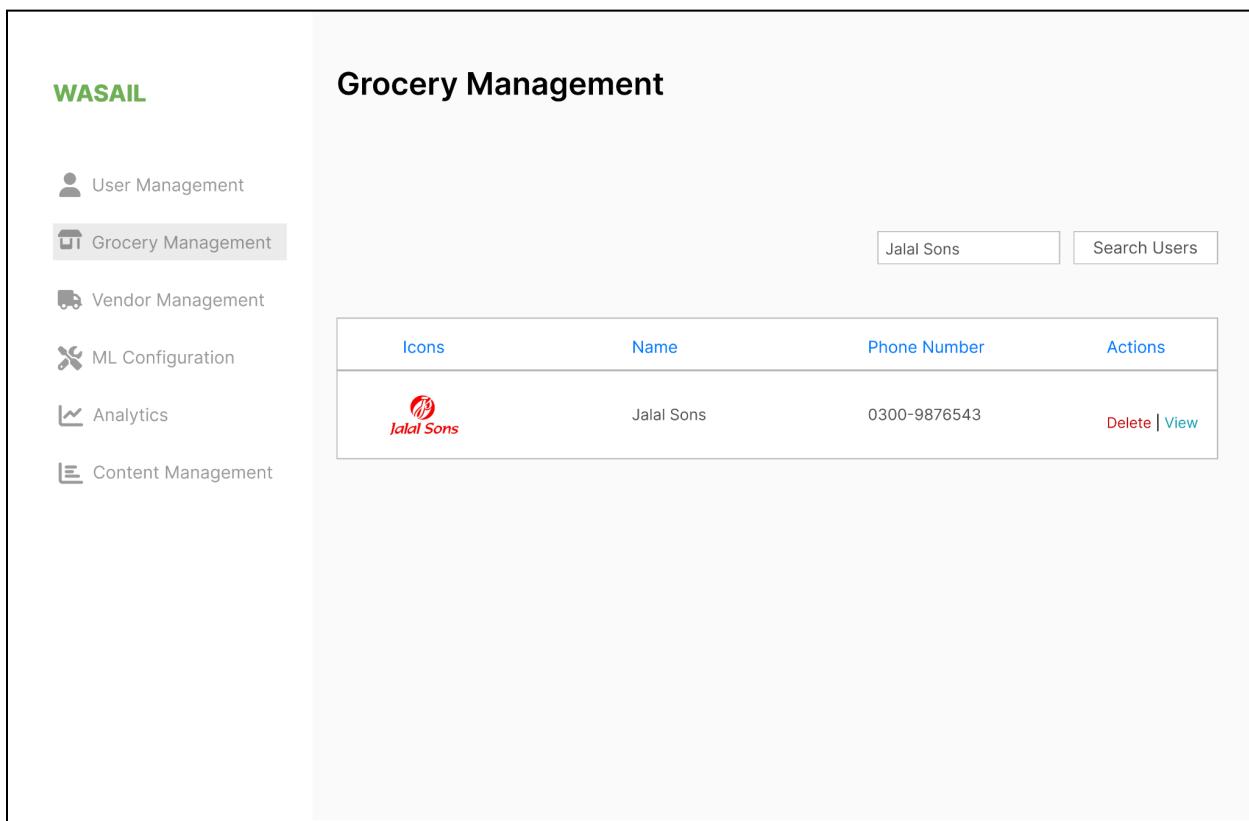
The screenshot shows a web-based application interface titled "Grocery Management". On the left, there is a sidebar with the "WASAIL" logo and several menu items: User Management, **Grocery Management** (which is currently selected), Vendor Management, ML Configuration, Analytics, and Content Management. The main content area is titled "Grocery Management" and displays a table of grocery store profiles. The table has columns for Icons, Name, Phone Number, and Actions. There are four entries in the table:

Icons	Name	Phone Number	Actions
	Jalal Sons	0300-9876543	Delete View
	Esajee's	0312-8766542	Delete View
	Al-Fateh	0314-6247966	Delete View
	Metro	0305-7654836	Delete View

Figure 2.6.4.8 Disable Grocery Store's Profile Prototype

FR4.9: Search Grocery Store

- **Description:** The system should allow the user to search the grocery store on the grocery store page (Figure 2.6.4.9).
- **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.



The screenshot shows a prototype of a grocery management system interface. On the left, there is a sidebar with the title "WASAIL" at the top. Below it are several menu items with corresponding icons: "User Management" (person icon), "Grocery Management" (grocery bag icon, highlighted in grey), "Vendor Management" (truck icon), "ML Configuration" (cogs icon), "Analytics" (chart icon), and "Content Management" (list icon). On the right, the main area has a title "Grocery Management". At the top right of this area is a search bar containing the text "Jalal Sons" and a "Search Users" button. Below the search bar is a table with four columns: "Icons", "Name", "Phone Number", and "Actions". The table contains one row with the following data: a logo for "Jalal Sons" (a red circle with a white design), the name "Jalal Sons", the phone number "0300-9876543", and a "Delete | View" button. The entire interface is contained within a large rectangular frame.

Figure 2.6.4.9 Search Grocery Store Prototype

FR4.10: View Vendor's Profile

- **Description:** The system should allow the user to view the vendor's profile (Figure 2.6.4.10).
- **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.

The screenshot shows the WASAIL application interface. On the left, there is a vertical sidebar with the following menu items: User Management, Grocery Management, **Vendor Management** (which is currently selected and highlighted in grey), ML Configuration, Analytics, and Content Management. The main content area is titled "Vendor Management". At the top right of this area, there is a search bar with the placeholder "Search Users" and a "Search" button. Below the search bar is a table listing four vendors. The table has columns for Icons, Name, Phone Number, and Actions. Each vendor row contains a small icon representing the vendor, the vendor's name, their phone number, and two buttons labeled "Delete" and "View".

Icons	Name	Phone Number	Actions
	Vendor A	0300-9876543	Delete View
	Vendor B	0312-8766542	Delete View
	Vendor C	0314-6247966	Delete View
	Vendor D	0305-7654836	Delete View

Figure 2.6.4.10 View Vendor's Profile Prototype

FR4.11: Disable Vendor's Profile

- **Description:** The system should allow the user to delete the vendor's profile (Figure 2.6.4.11).
- **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.

The screenshot shows the WASAIL application interface. On the left is a vertical sidebar with navigation links: User Management, Grocery Management, **Vendor Management** (which is currently selected and highlighted in grey), ML Configuration, Analytics, and Content Management. The main area is titled "Vendor Management". It features a search bar with placeholder text "Search Users" and a table listing four vendor profiles. The table columns are "Icons", "Name", "Phone Number", and "Actions". Each row contains an icon representing the vendor, the vendor's name, their phone number, and two action buttons: "Delete" and "View".

Icons	Name	Phone Number	Actions
	Vendor A	0300-9876543	Delete View
	Vendor B	0312-8766542	Delete View
	Vendor C	0314-6247966	Delete View
	Vendor D	0305-7654836	Delete View

Figure 2.6.4.11 Disable Vendor's Profile Prototype

FR4.12: Search Vendor

- **Description:** The system should allow the user to search the vendor on the vendor page (Figure 2.6.4.12).
- **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.

The screenshot shows a prototype of the WASAIL application's Vendor Management module. On the left, there is a sidebar with the following menu items: User Management, Grocery Management, **Vendor Management** (which is currently selected and highlighted in grey), ML Configuration, Analytics, and Content Management. The main content area is titled "Vendor Management". At the top right of this area, there is a search bar containing the placeholder text "Vendor A" and a "Search Users" button. Below the search bar is a table with the following columns: Icons, Name, Phone Number, and Actions. The table contains one row with the following data: a red "Vendor" icon, the name "Vendor A", the phone number "0300-9876543", and a "Delete | View" link in red text. The entire interface is framed by a thick black border.

Figure 2.6.4.12 Search Vendor Prototype

FR4.13: Select Grocery Store's ML Model

- **Description:** The system should allow the user to select the ML model for the grocery store (Figure 2.6.4.13).
- **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.

WASAIL

ML Configuration

User Management Grocery Management Vendor Management Search Users

ML Configuration Analytics Content Management

Icons	Name	Model	RMSLE
Jalal Sons	Jalal Sons	Jalal Sons Model A	0.23
Esajee's	Esajee's	Esajee's Model B	0.5
Al-Fatah	Al-Fateh	Al-Fateh Model A	1.2
METRO	Metro	Metro Model C	0.95

Figure 2.6.4.13 Select Grocery Store's ML Model Prototype

FR4.14: Display Analytics

- **Description:** The system should display the analytics to the user (Figure 2.6.4.14).
- **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.

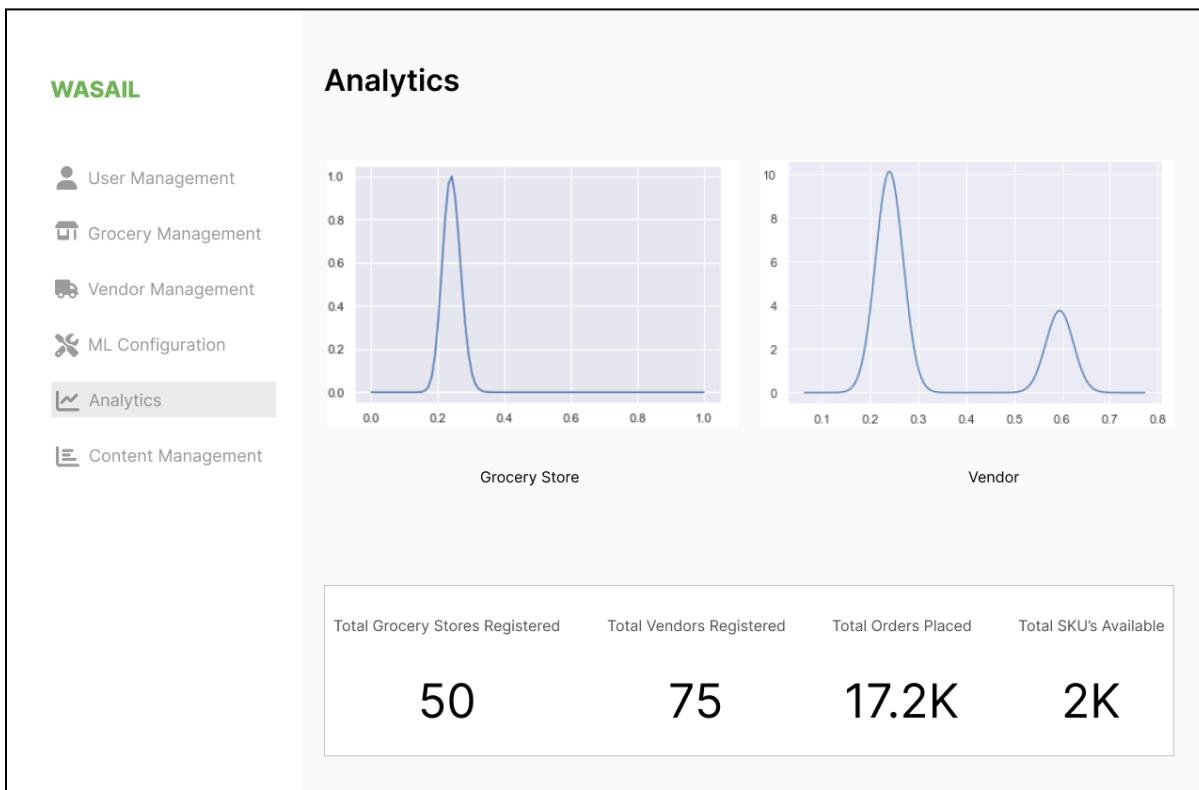


Figure 2.6.4.14 Display Analytics Prototype

FR4.15: View Grocery Store Count

- **Description:** The system should allow the user to view the total number of grocery stores that have been registered (Figure 2.6.4.15).
- **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.

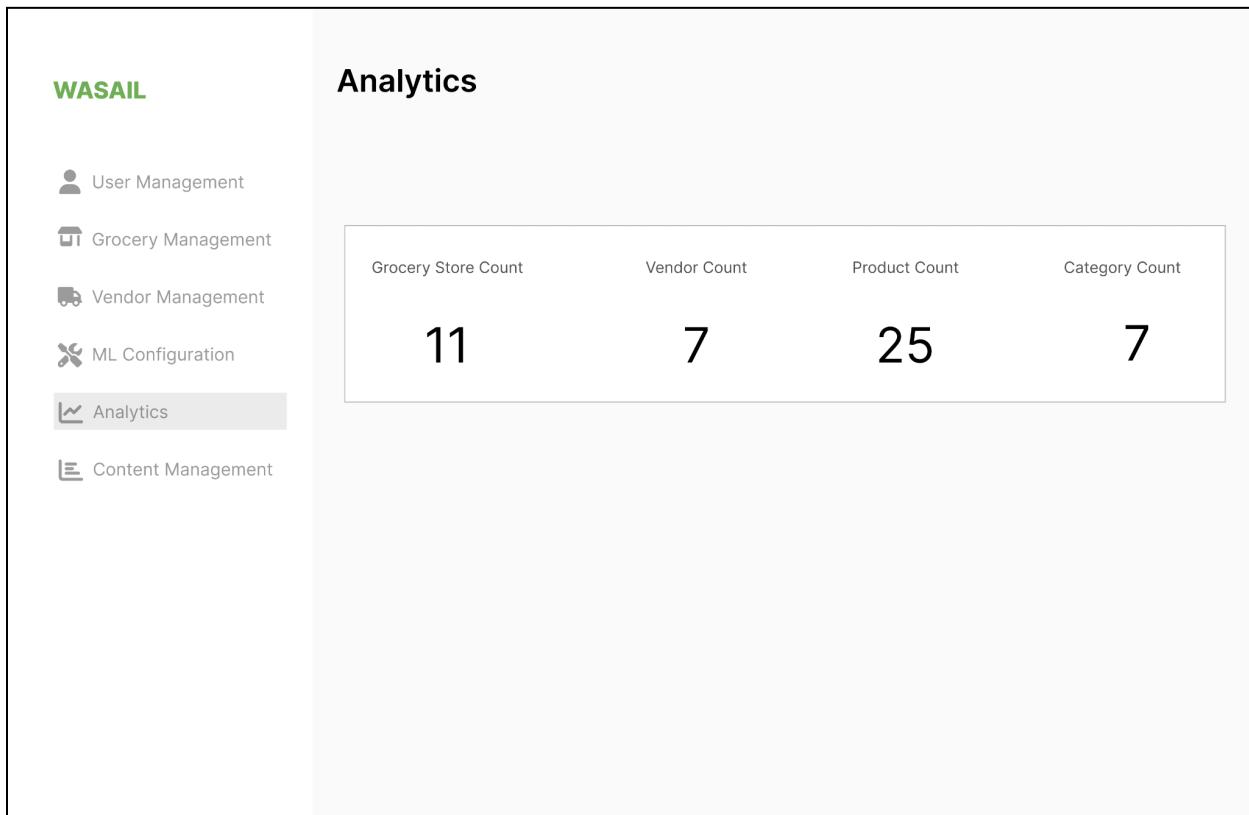


Figure 2.6.4.15 View Grocery Store Count Prototype

FR4.16: View Vendor Count

- **Description:** The system should allow the user to view the total number of vendors that have been registered (Figure 2.6.4.16).
- **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.

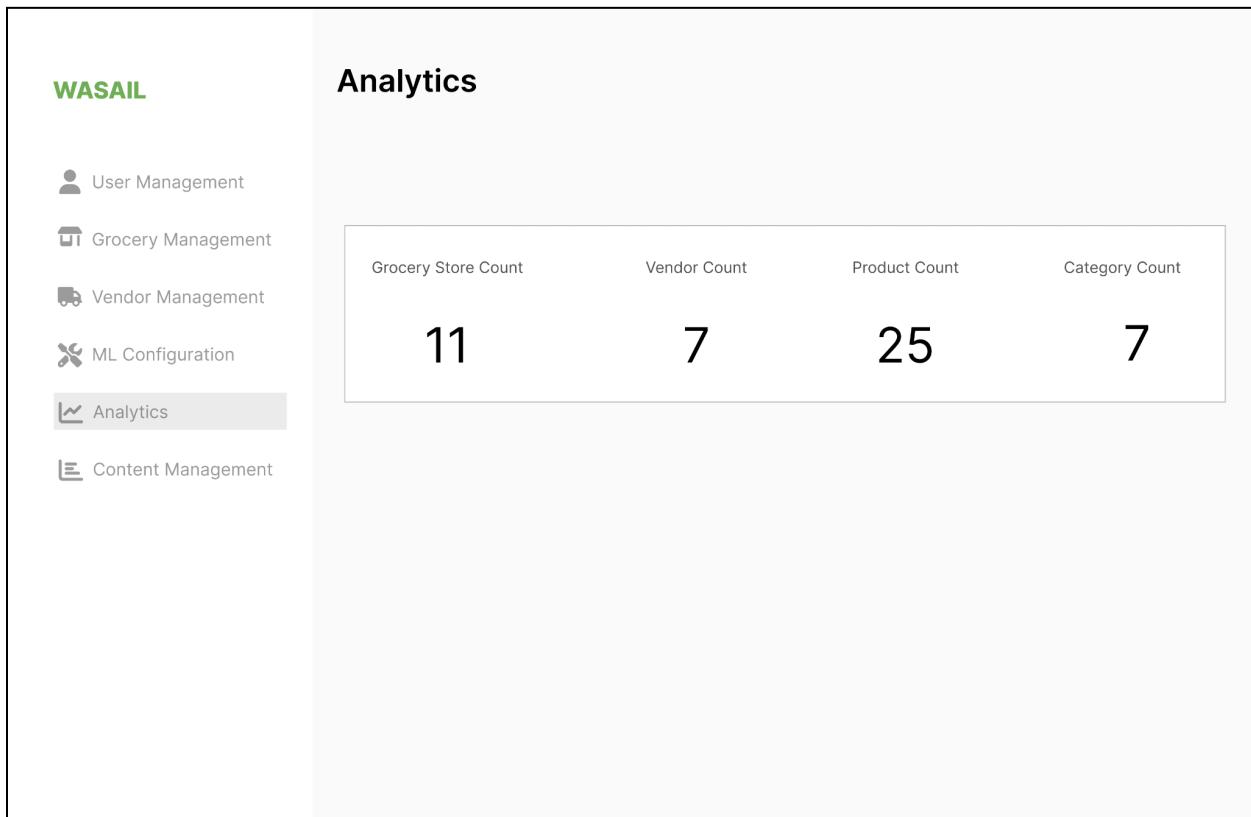


Figure 2.6.4.16 View Vendor Count Prototype

FR4.17: View Category Count

- **Description:** The system should allow the user to view the total number of categories in the database (Figure 2.6.4.17).
- **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.

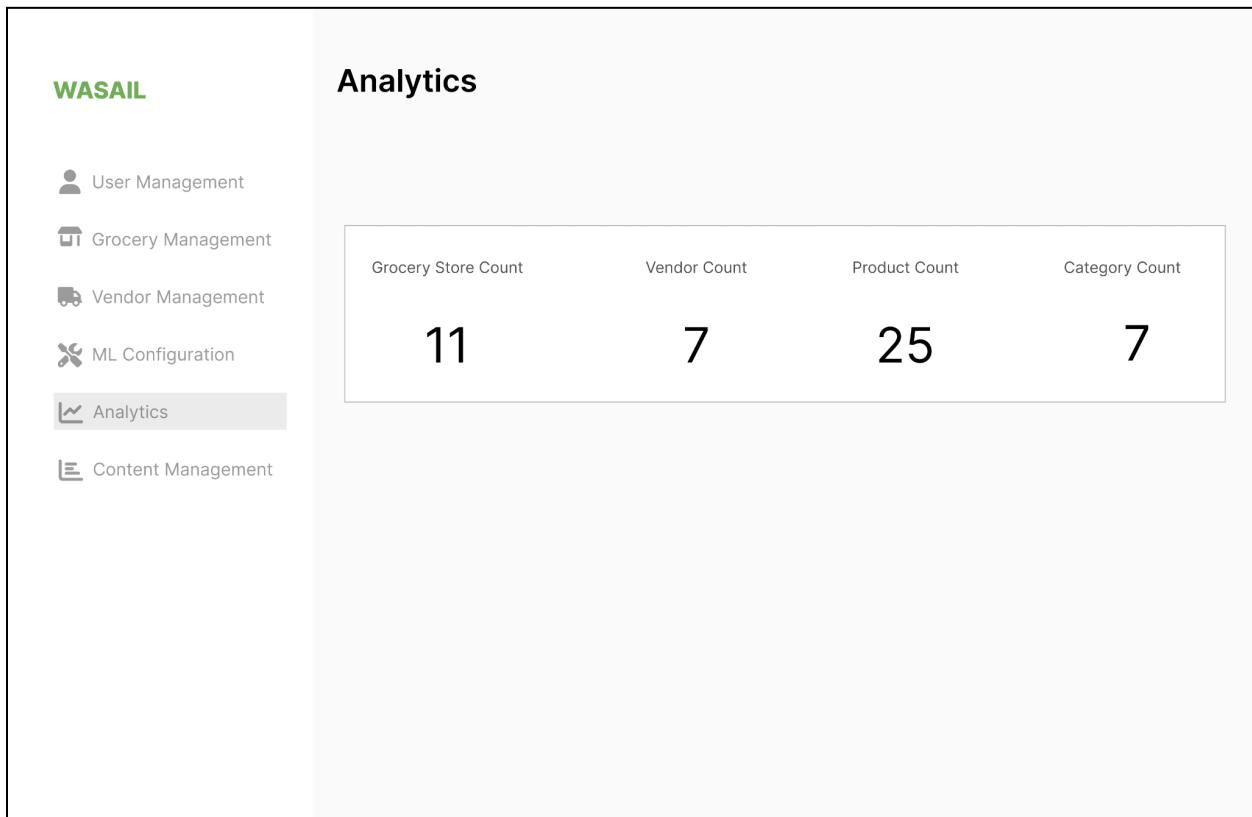


Figure 2.6.4.17 View Category Count Prototype

FR4.18: View Product Count

- **Description:** The system should allow the user to view the total number of products in the database (Figure 2.6.4.18).
- **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.

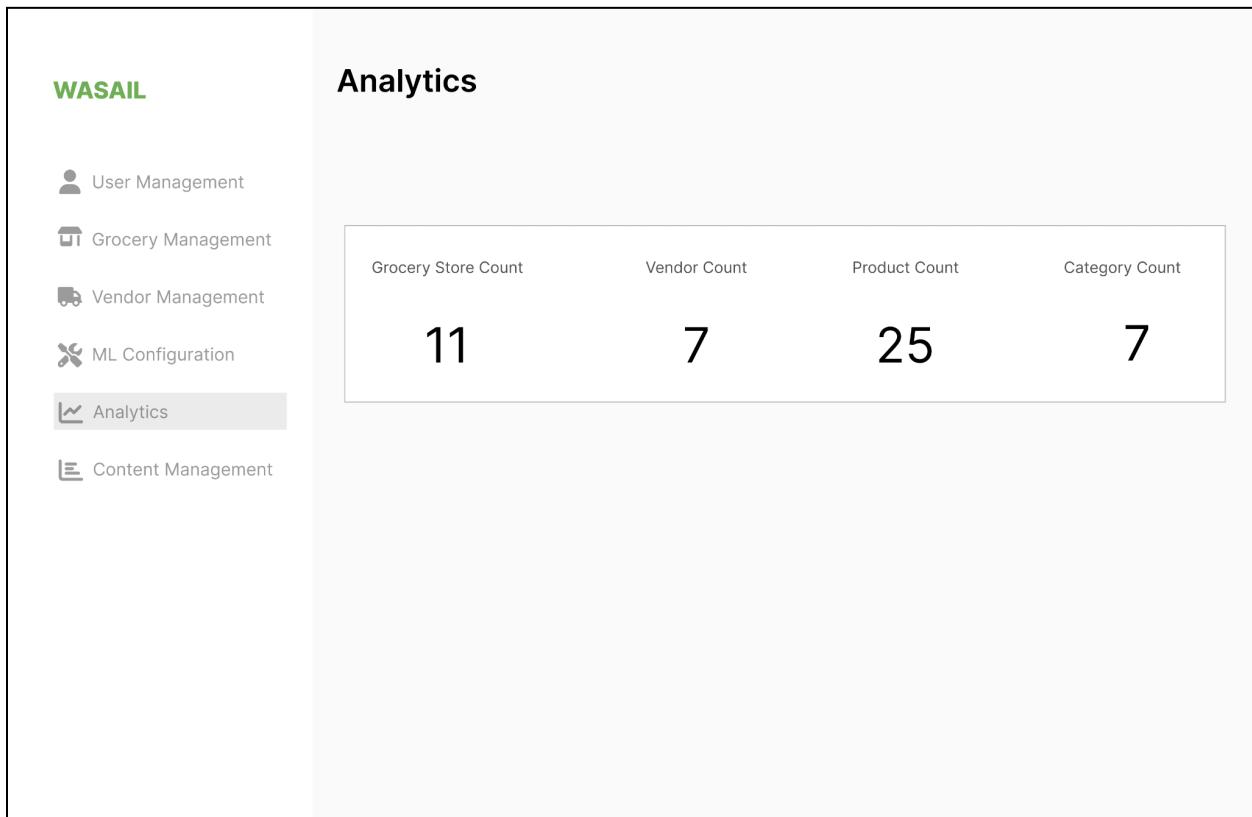


Figure 2.6.4.18 View Product Count Prototype

FR4.19: Add Category

- **Description:** The system should allow the user to add a category of products (Figure 2.6.4.19).
- **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.

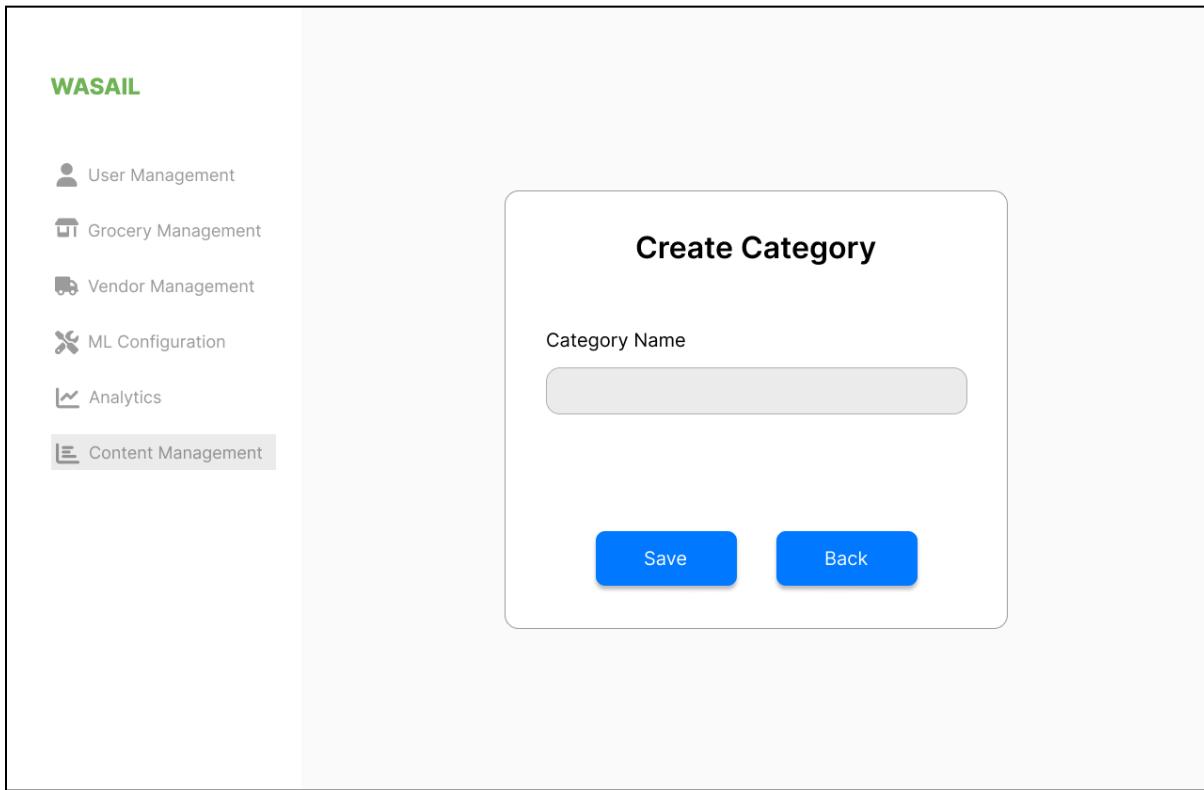


Figure 2.6.4.19 Add Category Prototype

FR4.20: Update Category

- **Description:** The system should allow the user to update the category (Figure 2.6.4.20).
- **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.

The screenshot shows the 'List Category' screen of the WASAIL application. On the left, there is a sidebar with the following menu items:

- User Management
- Grocery Management
- Vendor Management
- ML Configuration
- Analytics
- Content Management

The 'Content Management' item is highlighted with a grey background. The main area is titled 'List Category' and contains a table with the following data:

Category Name	Actions
Dairy	Update Delete
Vegetables	Update Delete
Snacks	Update Delete
Meat	Update Delete

Below the table, there is a blue button labeled 'Add Category' and a link labeled 'List Product'.

Figure 2.6.4.20 Update Category Prototype

FR4.21: Delete Category

- **Description:** The system should allow the user to delete the category (Figure 2.6.4.21).
- **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.

The screenshot shows the 'List Category' page of the WASAIL application. On the left, there is a sidebar with the following menu items:

- User Management
- Grocery Management
- Vendor Management
- ML Configuration
- Analytics
- Content Management

The 'Content Management' item is highlighted with a grey background. The main area is titled 'List Category' and contains a table with the following data:

Category Name	Actions
Dairy	Update Delete
Vegetables	Update Delete
Snacks	Update Delete
Meat	Update Delete

Below the table, there is a blue link labeled 'List Product'.

Figure 2.6.4.21 Delete Category Prototype

FR4.22: Search Category

- **Description:** The system should allow the user to search the category on the category page (Figure 2.6.4.22).
- **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.

The prototype interface for the 'List Category' page of the WASAIL system. On the left, there is a sidebar with the 'WASAIL' logo and links to User Management, Grocery Management, Vendor Management, ML Configuration, Analytics, and Content Management. The main area has a title 'List Category' and a blue 'Add Category' button. Below it is a search bar with the word 'Dairy' in the input field and a 'Search Category' button. A table lists a single category entry: 'Category Name' (Dairy) and 'Actions' (Update | Delete).

Category Name	Actions
Dairy	Update Delete

Figure 2.6.4.22 Search Category Prototype

FR4.23: Add Product

- **Description:** The system should allow the user to add product details (Figure 2.6.4.23).
- **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.

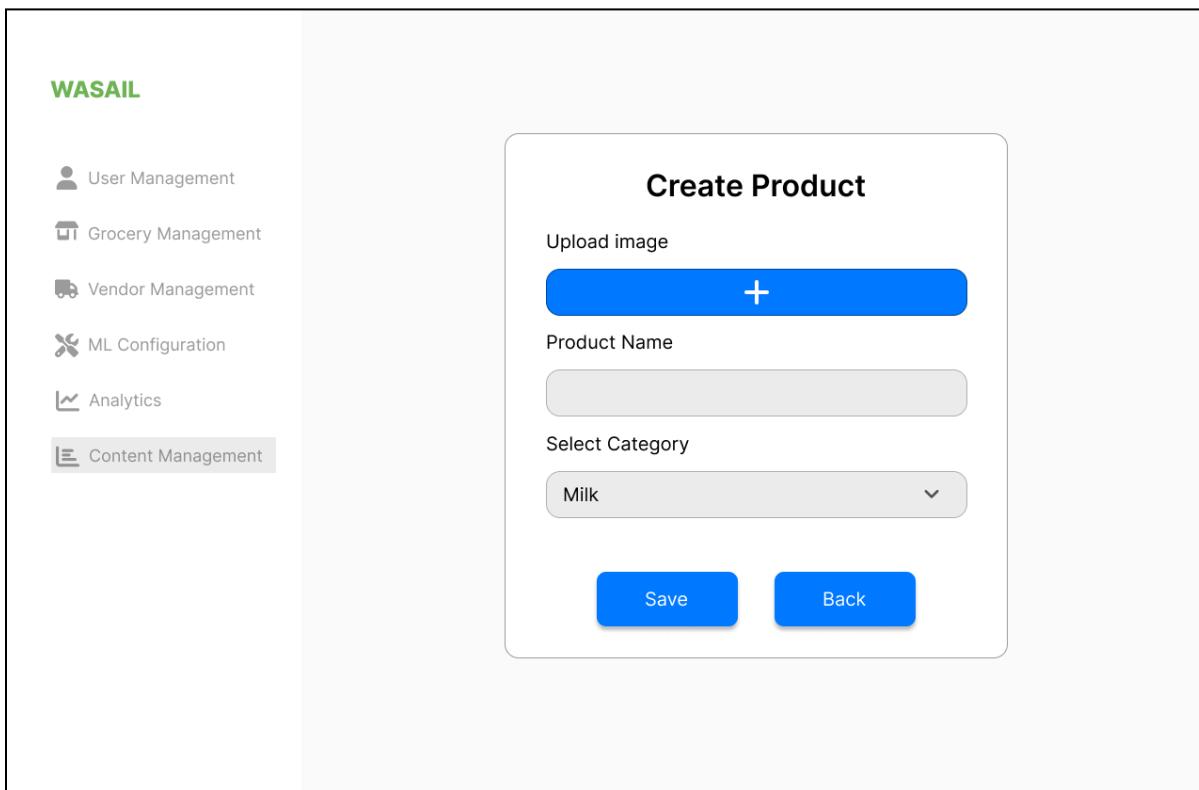


Figure 2.6.4.23 Add Product Prototype

FR4.24: Update Product

- **Description:** The system should allow the user to update the product (Figure 2.6.4.24).
- **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.

WASAIL

List Product

Add Product

User Management

Grocery Management

Vendor Management

ML Configuration

Analytics

Content Management

Image	Product Name	Category	Actions
	Olper's Milk Pack	Dairy	Update Delete
	Carrots	Vegetable	Update Delete
	Cookies	Snacks	Update Delete
	Minced Beef	Meat	Update Delete

[List Category](#)

Figure 2.6.4.24 Update Product Prototype

FR4.25: Delete Product

- **Description:** The system should allow the user to delete the product (Figure 2.6.4.25).
- **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.

The screenshot shows the 'List Product' page of the WASAIL application. On the left, there is a sidebar with the following menu items:

- User Management
- Grocery Management
- Vendor Management
- ML Configuration
- Analytics
- Content Management

The 'Content Management' item is highlighted with a grey background. The main area is titled 'List Product' and contains a blue 'Add Product' button. Below it is a search bar with a placeholder and a 'Search Product' button. A table displays four products:

Image	Product Name	Category	Actions
	Olper's Milk Pack	Dairy	Update Delete
	Carrots	Vegetable	Update Delete
	Cookies	Snacks	Update Delete
	Minced Beef	Meat	Update Delete

At the bottom of the table, there is a link labeled 'List Category'.

Figure 2.6.4.25 Delete Product Prototype

FR4.26: Search Product

- **Description:** The system should allow the user to search the product on the product page (Figure 2.6.4.26).
- **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.

The screenshot shows a web-based application interface for 'WASAIL'. On the left, there is a vertical sidebar with icons and labels for User Management, Grocery Management, Vendor Management, ML Configuration, Analytics, and Content Management. The main area is titled 'List Product' and contains a blue 'Add Product' button. Below it is a search bar with the placeholder 'Search Product' and a text input field containing 'Olper's Milk'. A table lists products with columns for Image, Product Name, Category, and Actions. One row is visible, showing an image of a milk carton labeled 'Olper's Milk Pack', the category 'Dairy', and an 'Actions' column with 'Update | Delete' links.

Image	Product Name	Category	Actions
	Olper's Milk Pack	Dairy	Update Delete

Figure 2.6.4.26 Search Product Prototype

2.7 Non-Functional Requirements

2.7.1 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.

2.7.2 Security

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

2.7.3 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.

2.7.4 Data Storage

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

2.7.5 Error Handling

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

2.7.6 Mobile Responsiveness

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

2.8 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

3 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.

3.1 Development Tools

The following tools were used while developing our project.

3.1.1 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.

3.1.2 Machine Learning Libraries

- Scikit-Learn
- Prophet
- TensorFlow
- PyTorch
- Darts

3.1.3 Web Development

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

3.1.4 Data Storage

- MySQL

3.1.5 Object Relational Mapper

- Sequelize

3.1.6 Cloud Services

- Digital Ocean

3.1.7 Mobile App Development

- Flutter

3.1.8 Version Control

- GitHub

3.1.9 Project Management

- Jira

3.1.10 Collaboration

- Discord

3.2 System Architecture

3.2.1 System Context Diagram

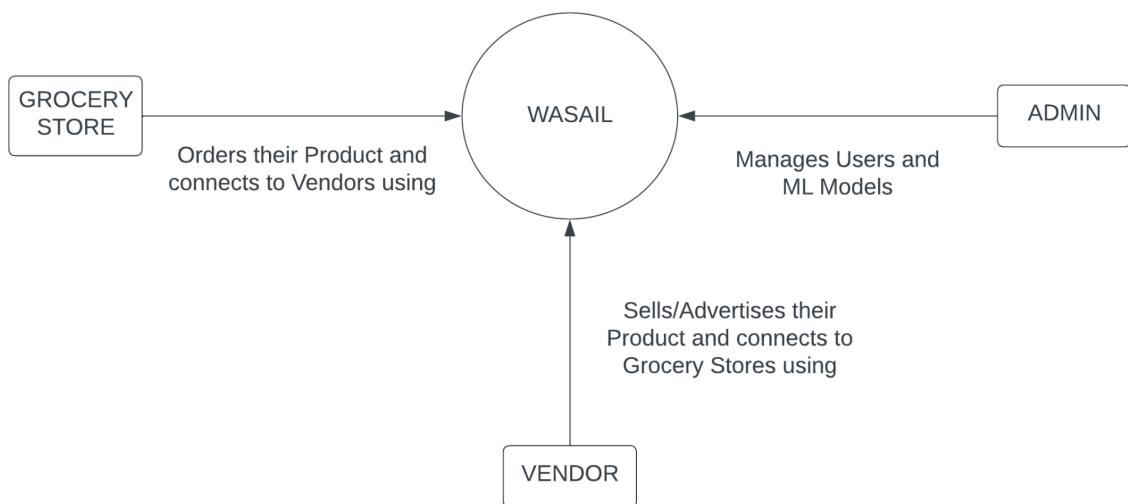


Figure 3.1.1 System Context Diagram

3.2.2 Container Diagram

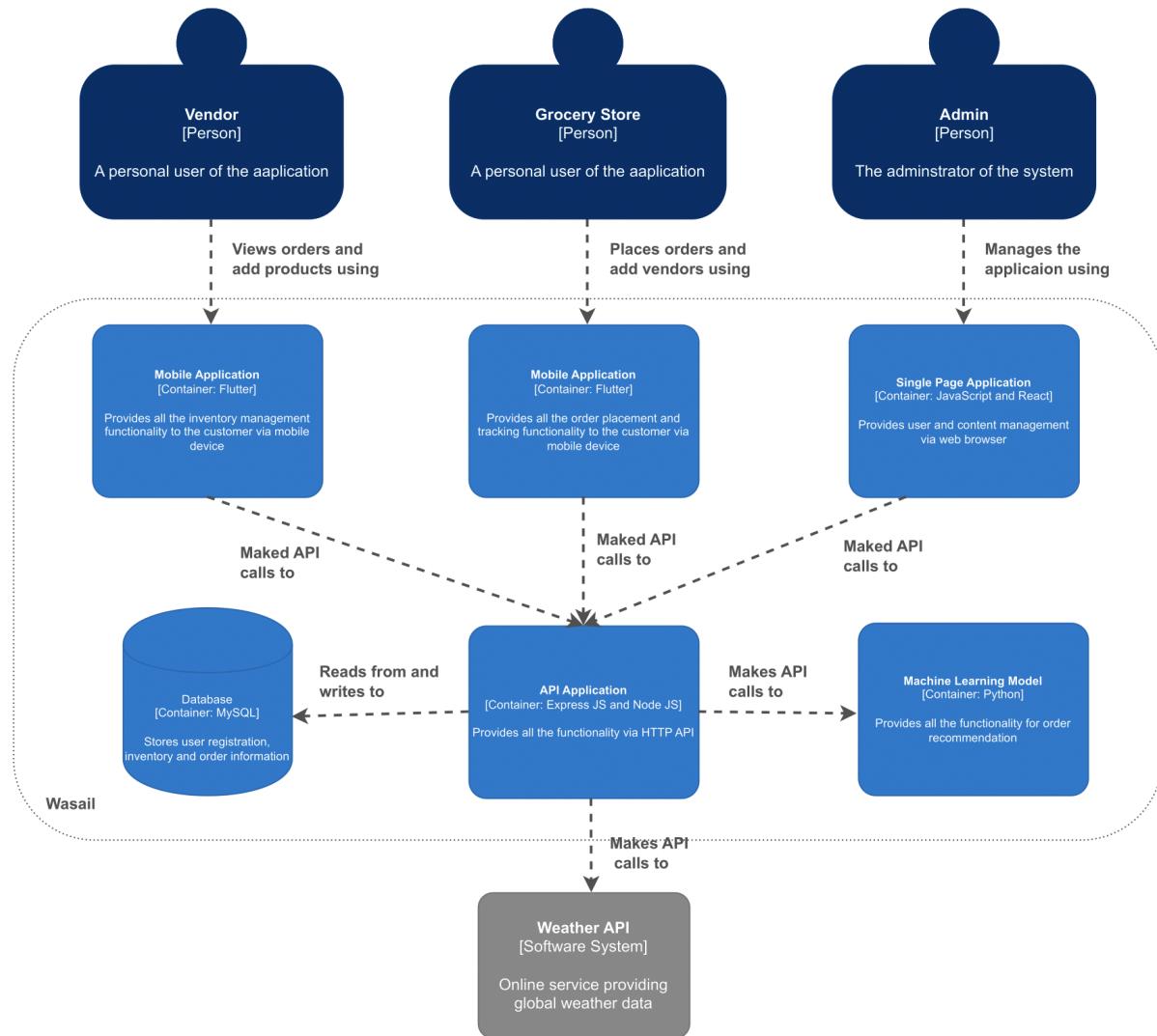


Figure 3.1.2 Container Diagram

3.2.3 Component Diagram

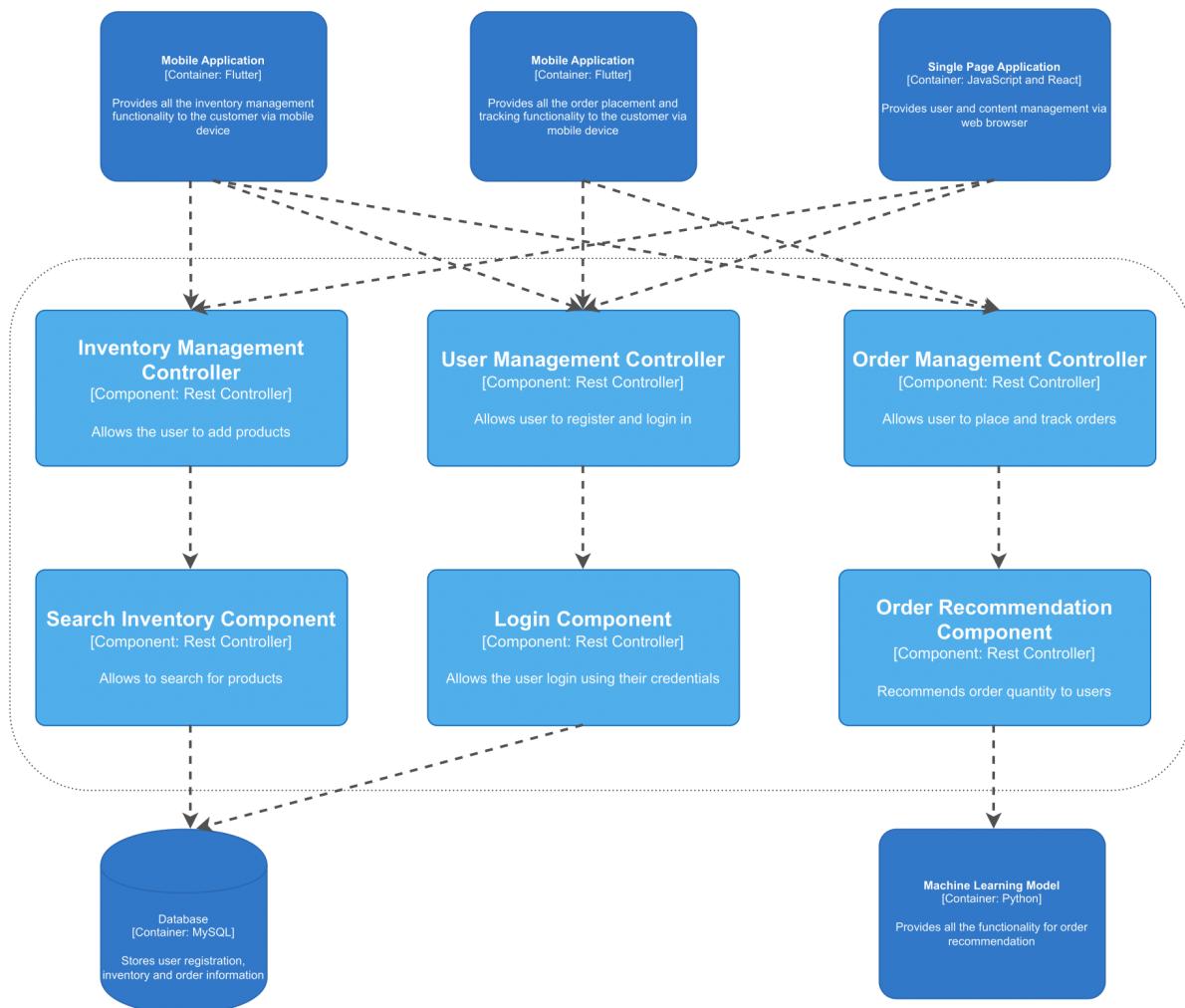


Figure 3.1.3 Component Diagram

3.2.4 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.

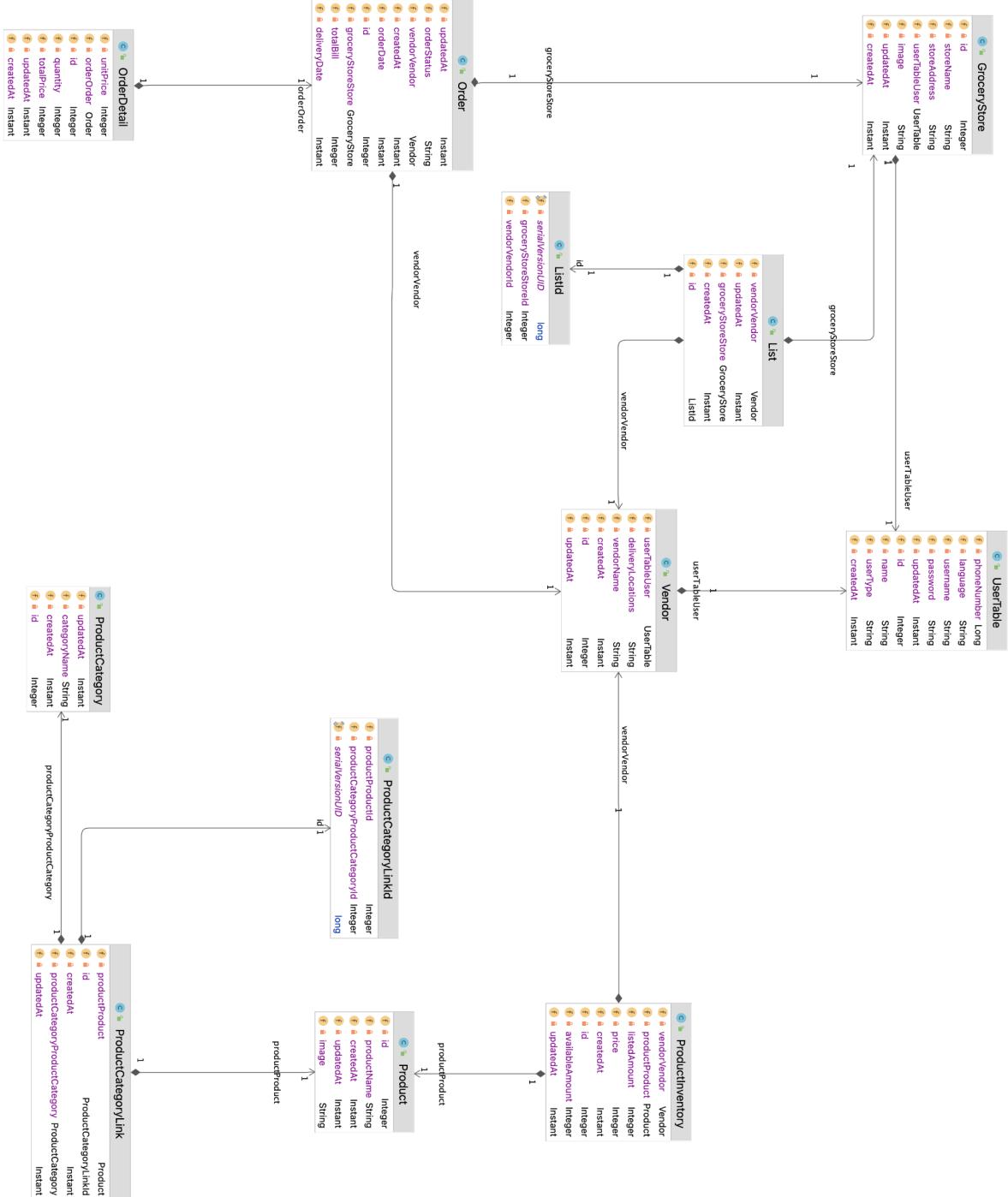


Figure 3.1.4 Class Diagram

3.3 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.

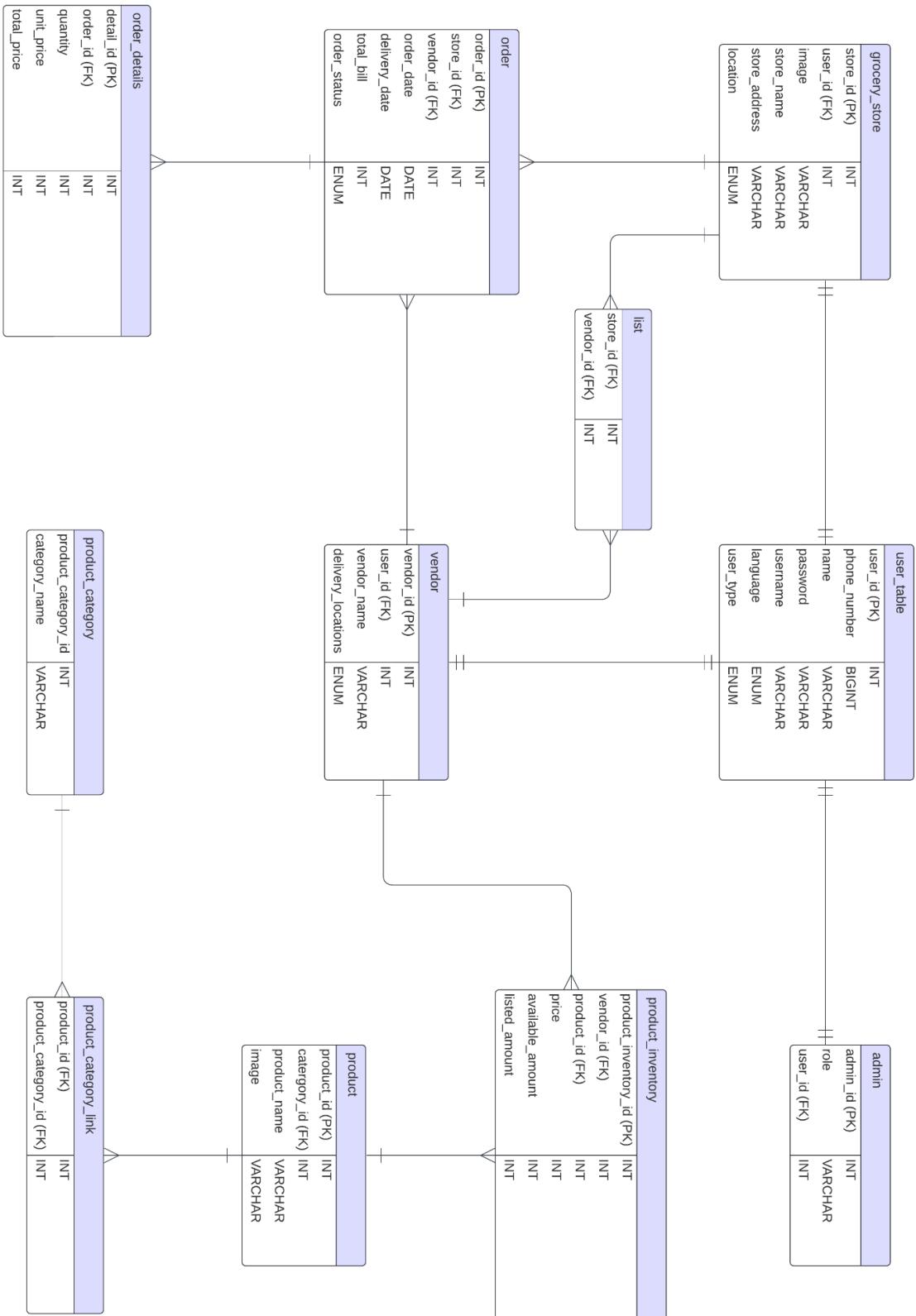


Figure 3.2.1 Data Design

3.4 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

The sequence diagram (Figure 3.3.1) illustrates the system recommending order amounts to a grocery store based on real-time data and external API inputs.

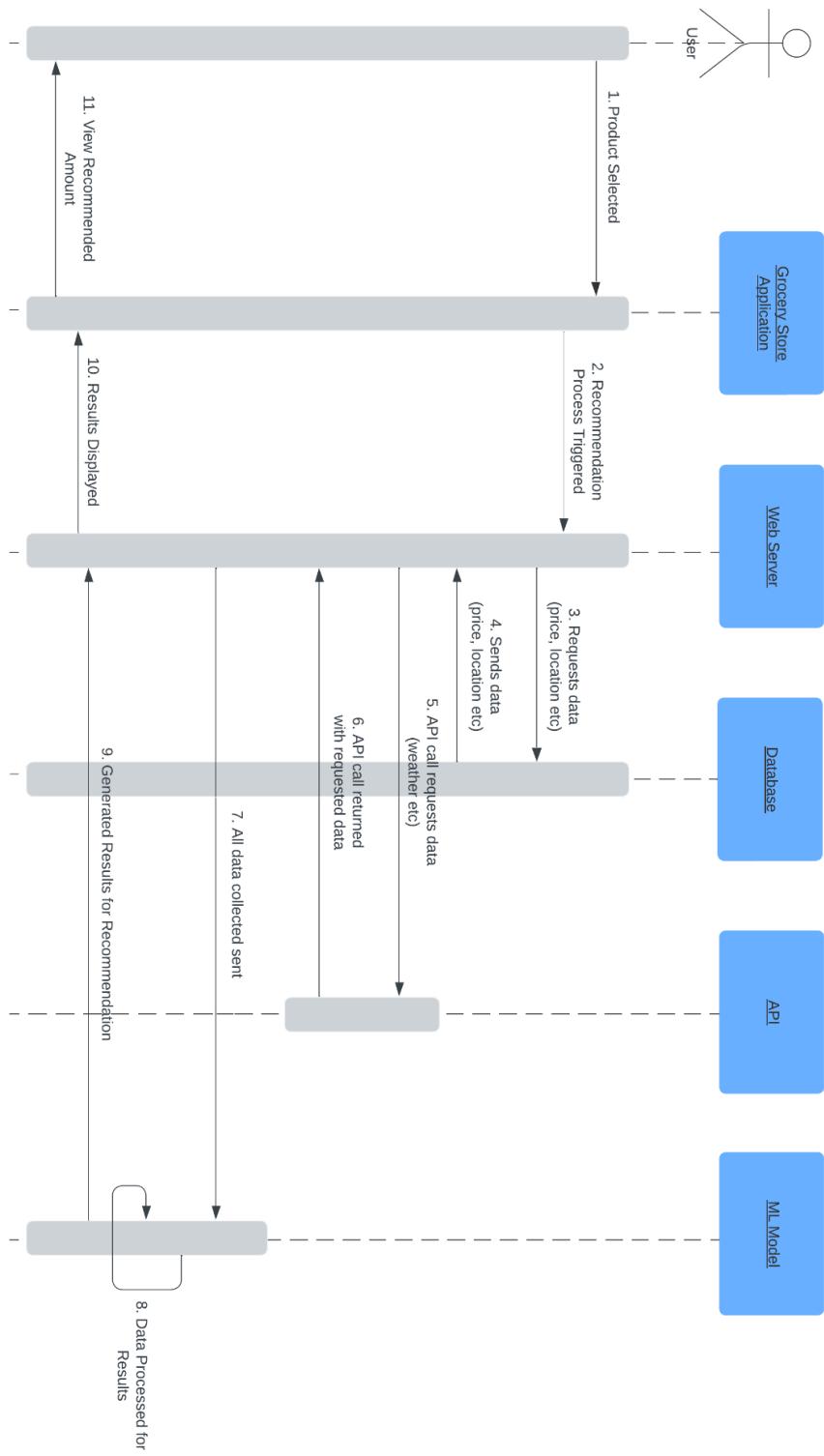


Figure 3.3.1 Order Recommendation Sequence Diagram

FR 2.24 Order Tracking

The sequence diagram (Figure 3.3.2) represents the logical flow of actions for a user to track their orders, from selection to delivery notification.

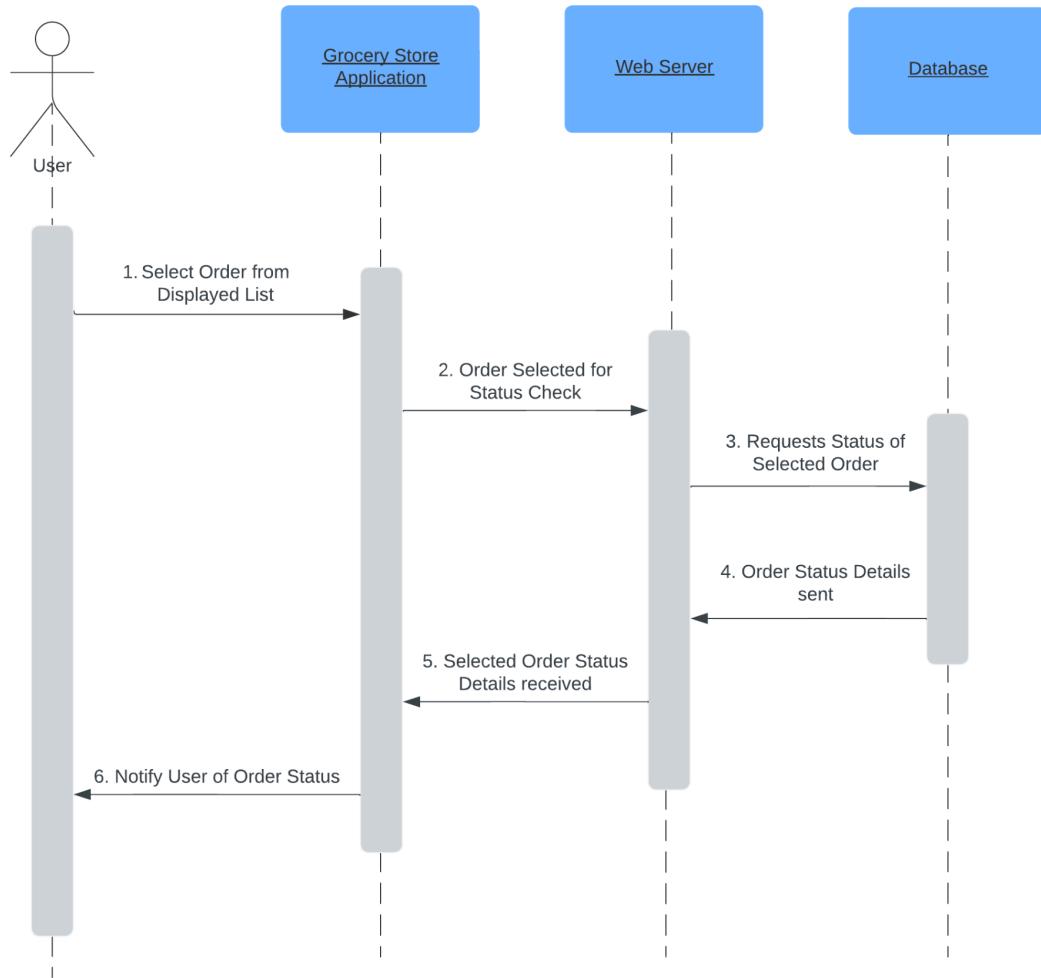


Figure 3.3.2 Order Tracking Sequence Diagram

FR 3.14 All Products Search

The sequence diagram (Figure 3.3.3) depicts the practical steps for a vendor to search, select, and add products to their inventory.

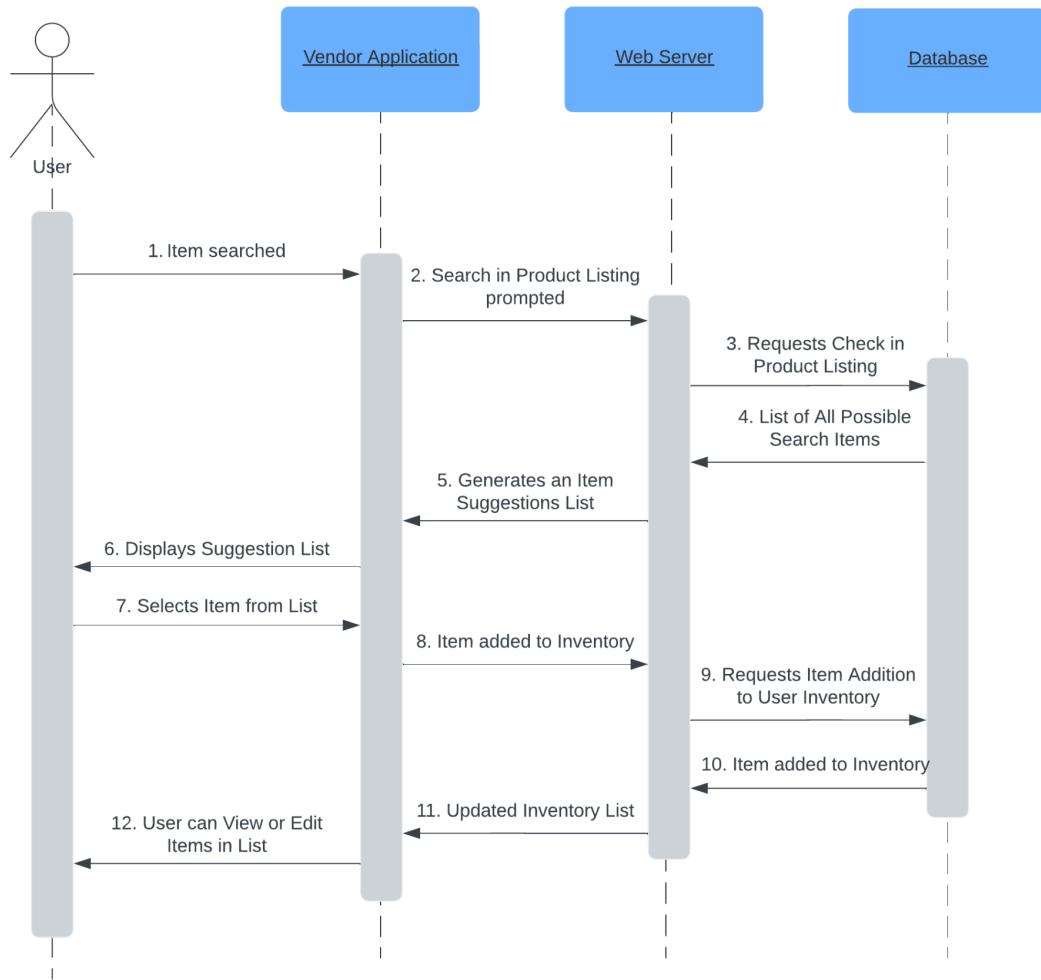


Figure 3.3.3 All Products Search Sequence Diagram

FR 1.4 OTP Code Generation and Delivery

The sequence diagram (Figure 3.3.4) illustrates the secure process of generating and delivering OTP codes for user verification.

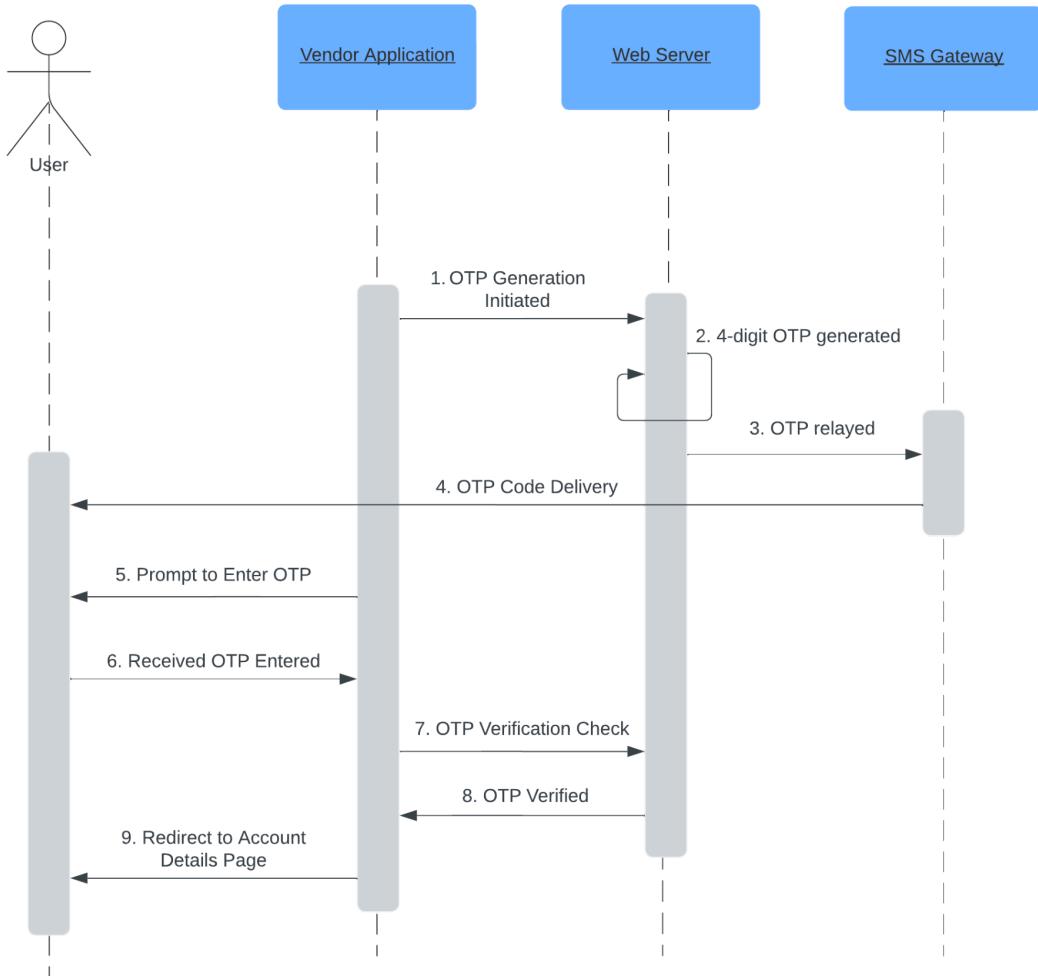


Figure 3.3.4 OTP Code Generation and Delivery Sequence Diagram

FR 2.14 Add Vendor to Vendor List

The sequence diagram (Figure 3.3.5) represents the user-friendly process of a grocery store adding a vendor to its list for efficient management.

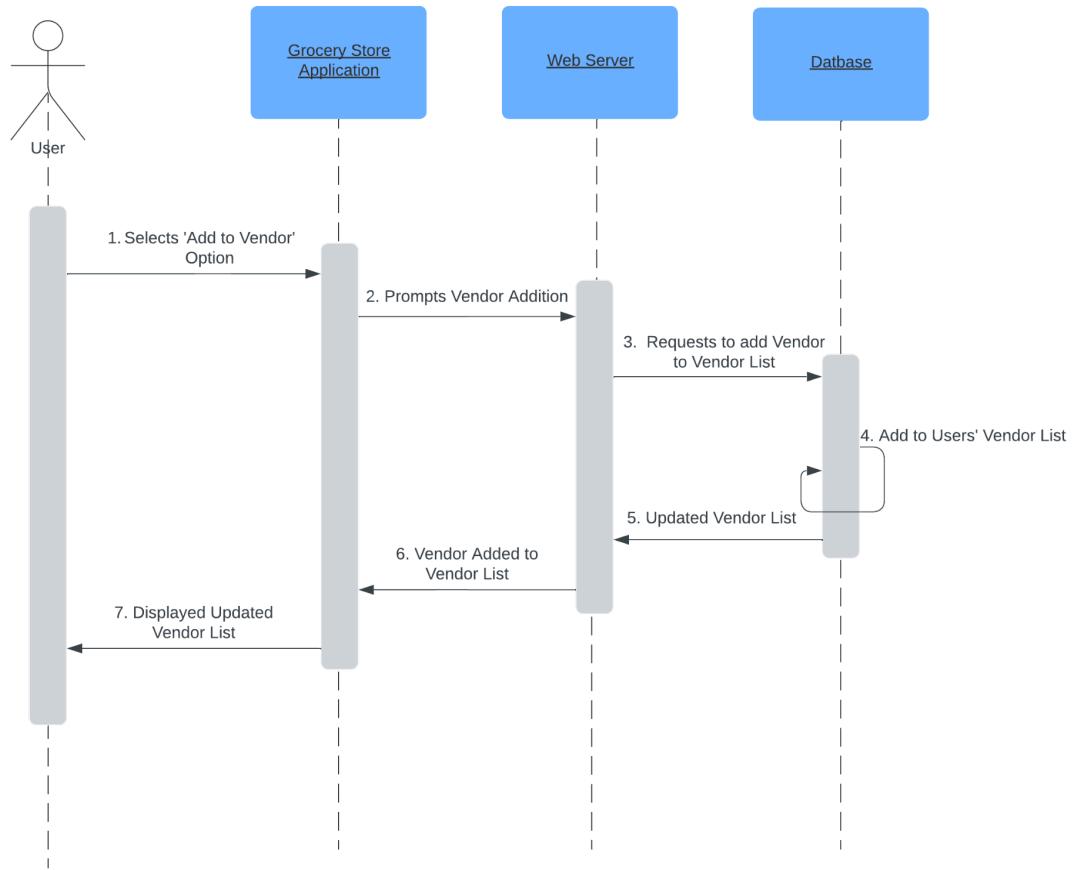


Figure 3.3.5 Add Vendor to Vendor List Sequence Diagram

3.5 API Design

The API Design (Figure 3.5) 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 app.

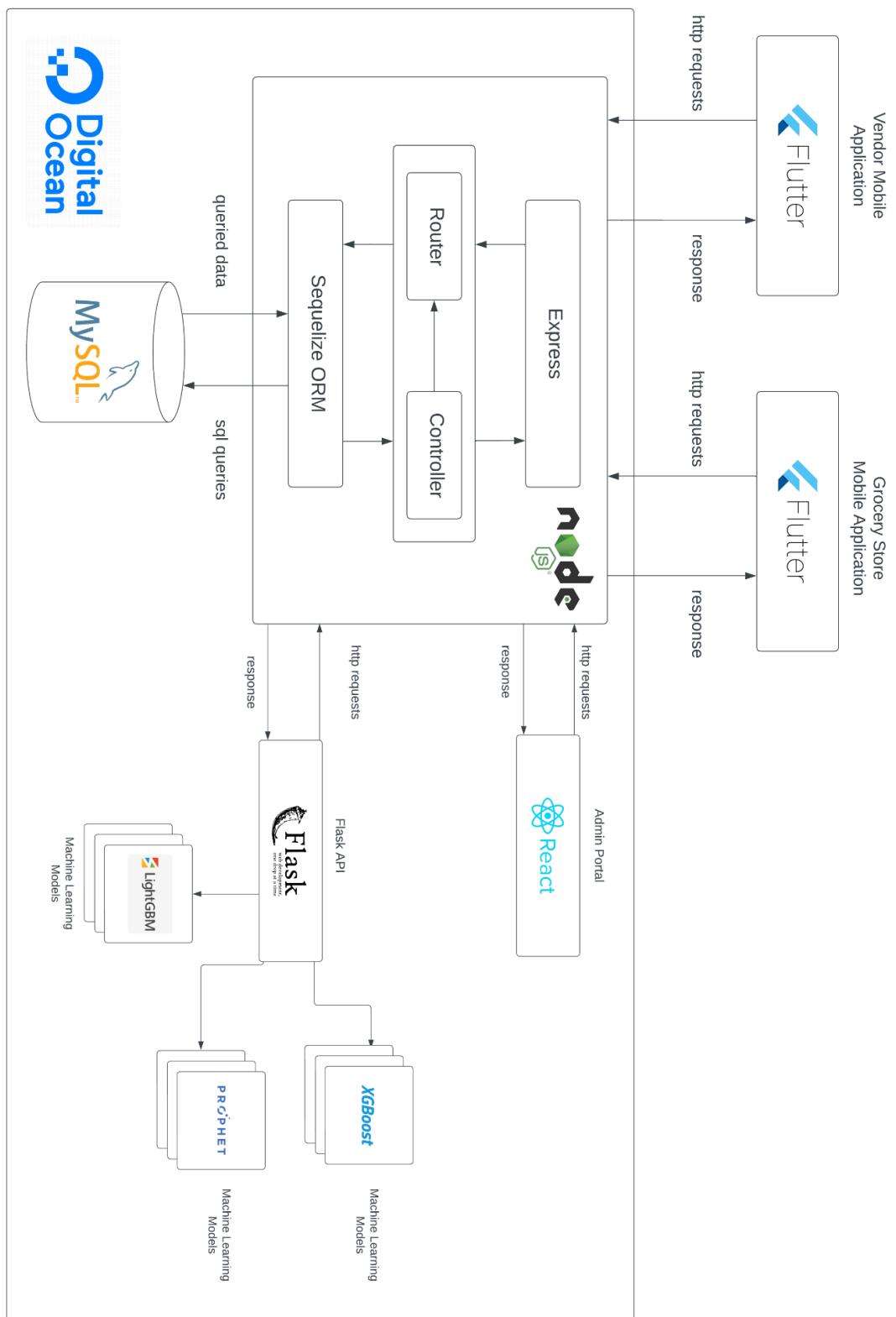


Figure 3.5 API Design

3.6 Machine Learning Design

3.6.1 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.

3.6.2 Feature Engineering

3.6.2.1 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 3.5.1.1 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).

saleinvcode	CustomerRef	date	invdiscperc	flatdisc	miscecharges	invsalestax	remarks	looseqty	packqty	price	itemdiscperc	packunits	batch	expiry	salestar	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 3.5.1.2 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 3.5.1.3 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()		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 3.5.1.4 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 3.5.1.5 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 3.5.1.6 Filtering Panadol Tab rows

3.6.2.2 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 3.5.2.1 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 3.5.2.2 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 3.5.2.3 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 3.5.2.4 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)
```

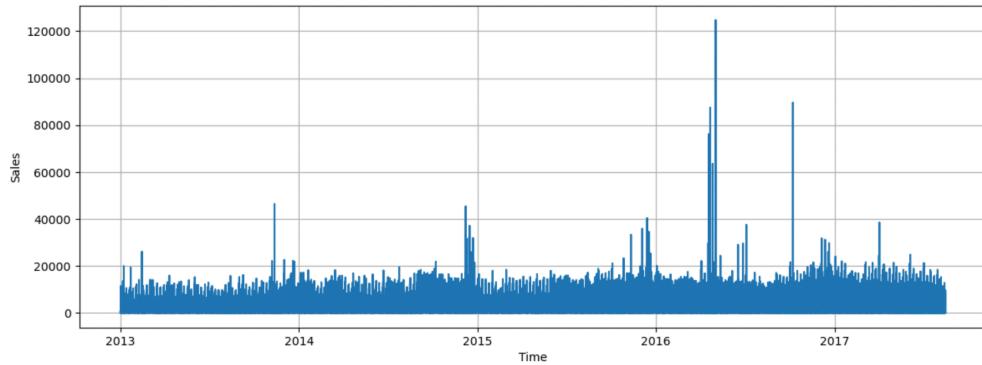


Figure 3.5.2.5 Visualising time series data

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 3.5.2.6 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  dc01iwico  city  state  typestores  cluster
0  0  2013-01-01        1  AUTOMOTIVE    0.0       0  Holiday  National  Ecuador  Primer dia del año  False    NaN  Quito  Pichincha      D  13
1  1  2013-01-01        1  BABY CARE    0.0       0  Holiday  National  Ecuador  Primer dia del año  False    NaN  Quito  Pichincha      D  13
2  2  2013-01-01        1    BEAUTY     0.0       0  Holiday  National  Ecuador  Primer dia del año  False    NaN  Quito  Pichincha      D  13
3  3  2013-01-01        1  BEVERAGES    0.0       0  Holiday  National  Ecuador  Primer dia del año  False    NaN  Quito  Pichincha      D  13
4  4  2013-01-01        1     BOOKS     0.0       0  Holiday  National  Ecuador  Primer dia del año  False    NaN  Quito  Pichincha      D  13
```

Figure 3.5.2.7 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()
```

	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 año	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 año	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 año	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 año	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 año	False	NaN	Quito	Pichincha	D	13	2	1	2013

Figure 3.5.2.8 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	count
NDay	26730
Holiday	1782
Name: count, dtype: int64	

Figure 3.5.2.9 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)
```

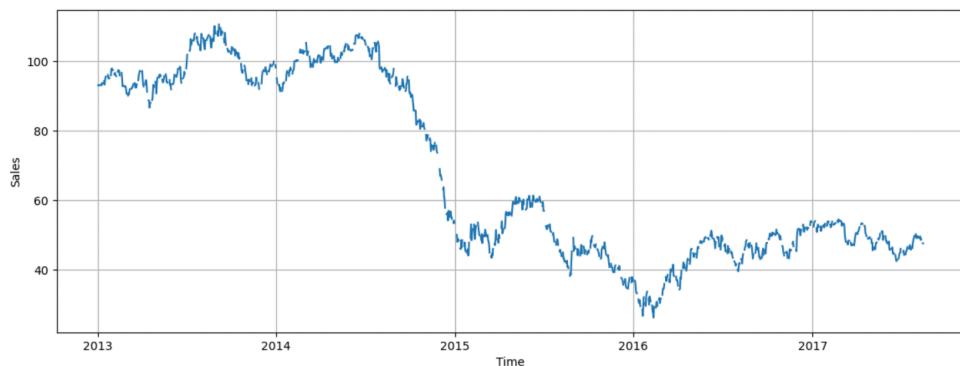


Figure 3.5.2.10 Missing values in oil prices

```
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)
```

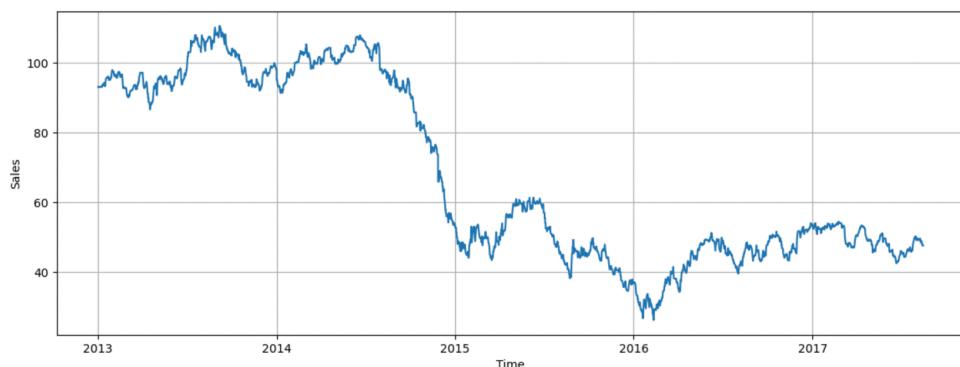


Figure 3.5.2.11 Fixing missing values in oil prices

Based on info seen '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.

3.6.3 Model Shortlisting

The slide features a circular profile picture of a woman in the top left corner. The title "Suite of demand forecasting models" is centered at the top in a large, bold, white font. Below the title is a stylized icon of a neural network or a series of connected dots. A bulleted list of models follows:

- Autoregressive models
- Prophet
- ML: Random Forest, XGBoost, ...
- RNNs, LSTMs, Transformers
- Hierarchical Forecasting
- Bayesian Hierarchical Forecasting

In the bottom right corner of the slide area, there is a small scatter plot showing data points (black dots) and a fitted red line, representing a time series forecast. At the bottom of the slide, the author's name "Shawn L. Ramirez, PhD" is followed by a LinkedIn icon and the handle "slramz". To the right, an email address "shawn@shelfengine.com" is provided.

Figure 3.5.3 Shelf Engine: Demand Forecasting Models

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.

3.6.3.1 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.

3.6.3.2 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.

3.6.3.3 Prophet

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.

3.6.3.4 Recurrent Neural Networks

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.

Next, we will implement models that leverage recent advancements to further enhance prediction accuracy and address specific challenges in the time series domain.

3.6.3.5 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.

3.6.3.6 N-BEATS

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

3.6.3.7 DeepAR

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.

3.6.4 Neural Network Architecture

We created a sequential model (Figure 3.6.4), 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 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.

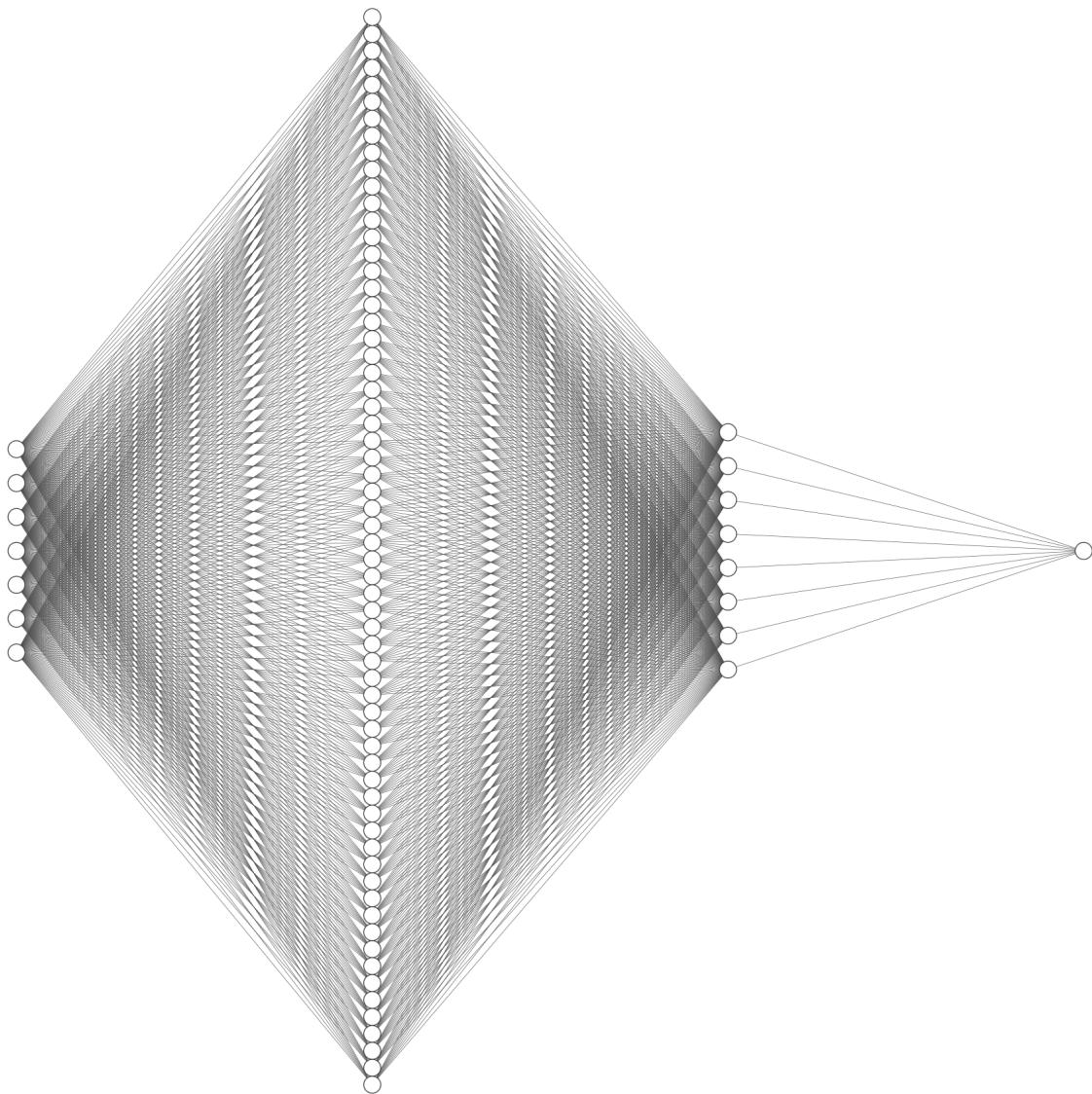


Figure 3.6.4 Neural network architecture drawn using [NN SVG](#)

3.7 Deployment Design

3.7.1 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 3.7.1).

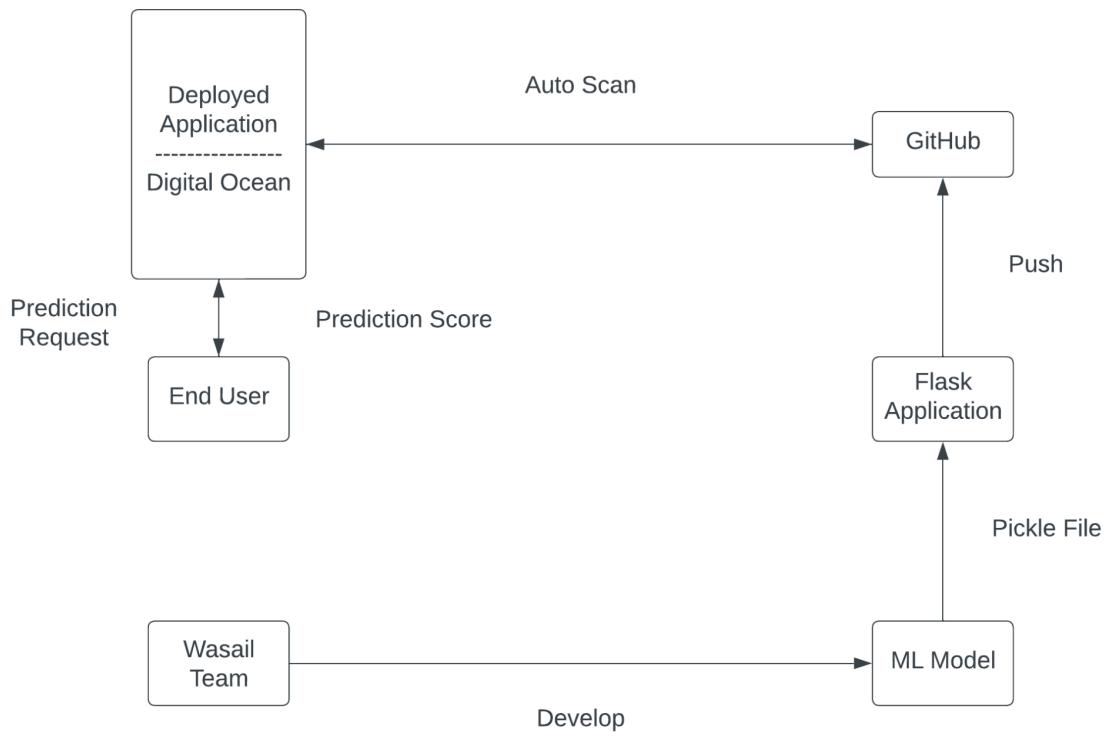


Figure 3.7.1 Model Deployment Diagram

4 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.

4.1 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.

4.1.1 Front End

4.1.1.1 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.

4.1.1.1.1 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.

As seen in the Figure 4.1.1.1.1 up until the OTP process is the phone confirmation and checking process whereby new users are distinguished from existing ones.

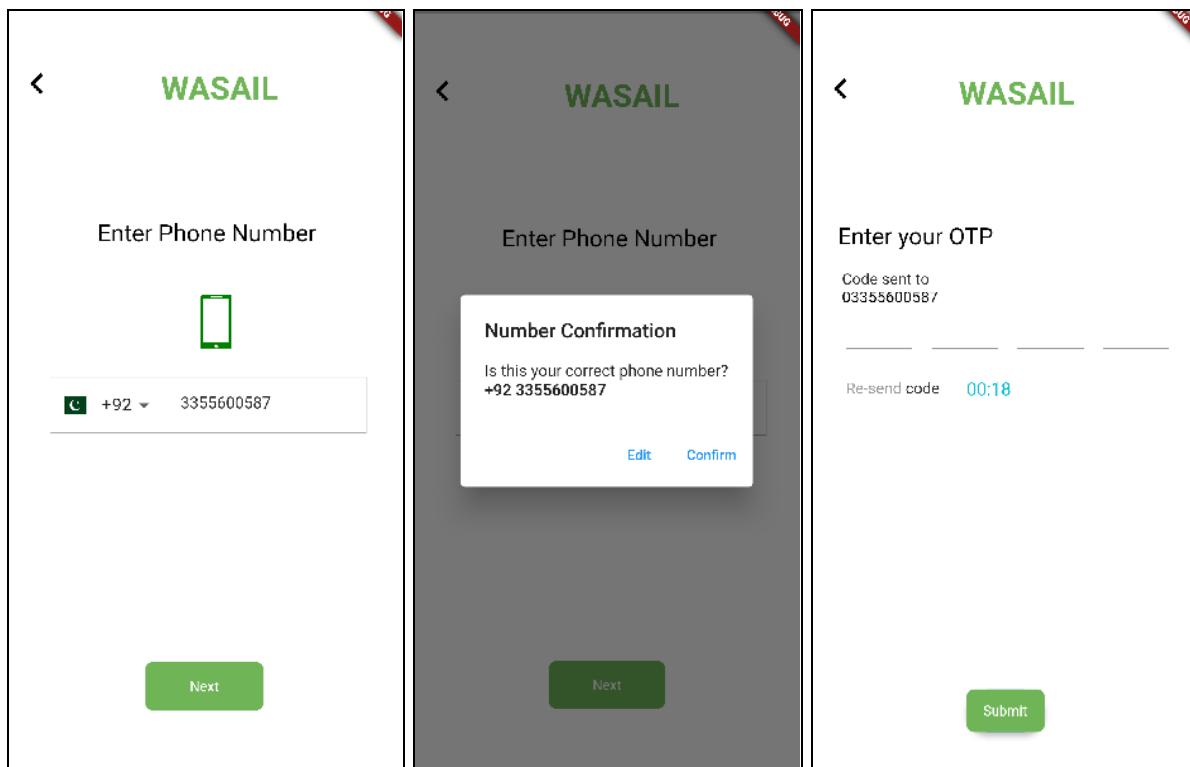


Figure 4.1.1.1.1 Phone Number, Confirmation, OTP

User Registration (Figure 4.1.1.1.2) : 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.

Phone Number
+92 3355600587

Password
Malaika123!

Confirm Password
Malaika123

Full Name
Malaika Sultan

Username
malaikasul

Delivery Areas
Gulberg ▾

I agree to the [Terms and Conditions](#) and [Privacy Policy](#)

Register

Phone Number
+92 3355600587

Password

Confirm Password

Full Name
Malaika Sultan

Username
malaikasul

Delivery Areas
Gulberg ▾

I agree to the [Terms and Conditions](#) and [Privacy Policy](#)

Register

Phone Number
+92 3355600587

Password

Confirm Password

Full Name
Malaika Sultan

Username
malaikast

Delivery Areas
Gulberg ▾

I agree to the [Terms and Conditions](#) and [Privacy Policy](#)

Register

Figure 4.1.1.1.1.2 Registration

Home Page (Figure 4.1.1.1.3): 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.

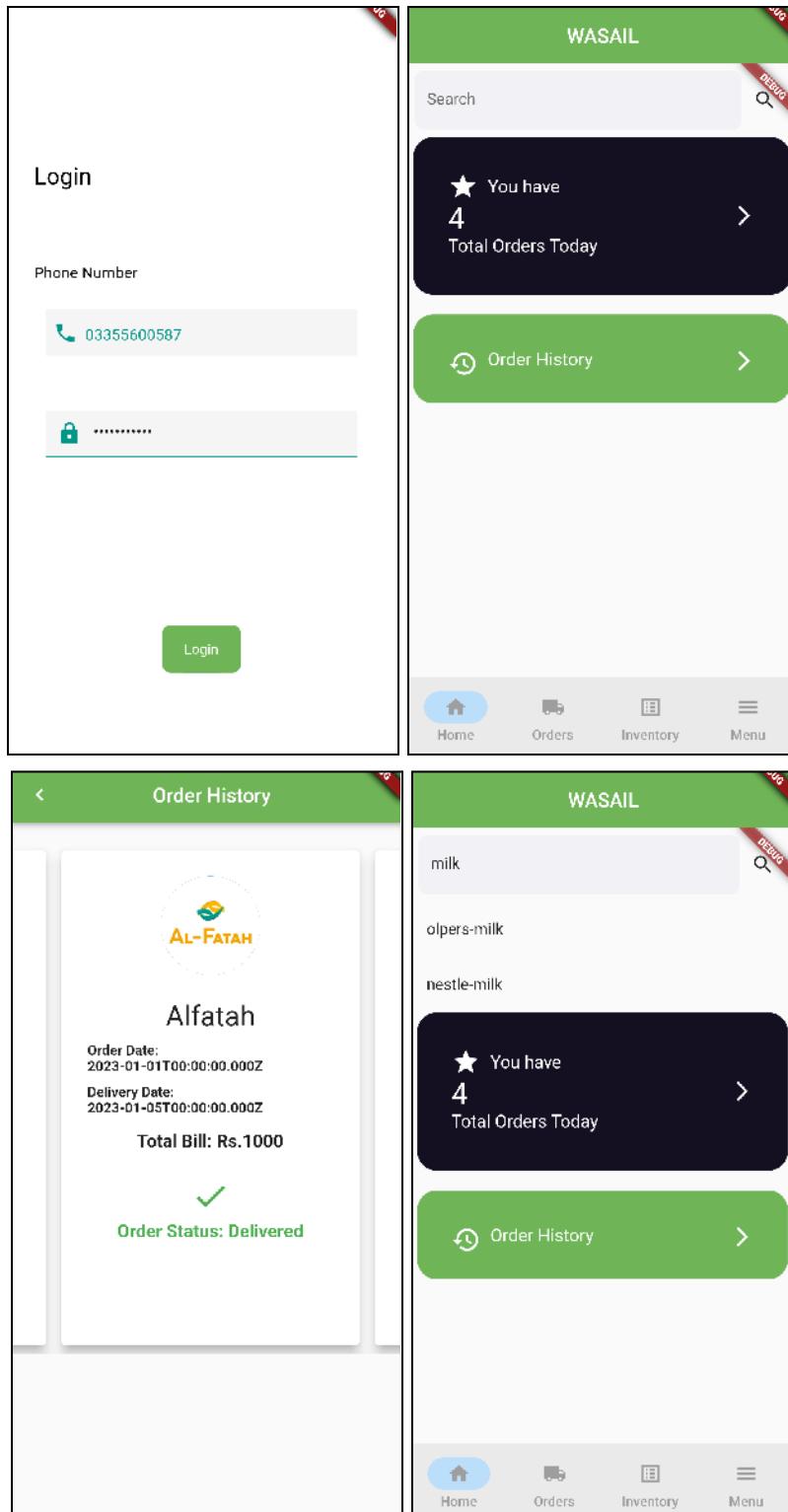


Figure 4.1.1.1.1.3 Login, Home, Order History, Search

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 4.1.1.1.1.4): Tracked all current orders, condition ‘In Process’ or ‘On Its Way’ for order tracking and management.

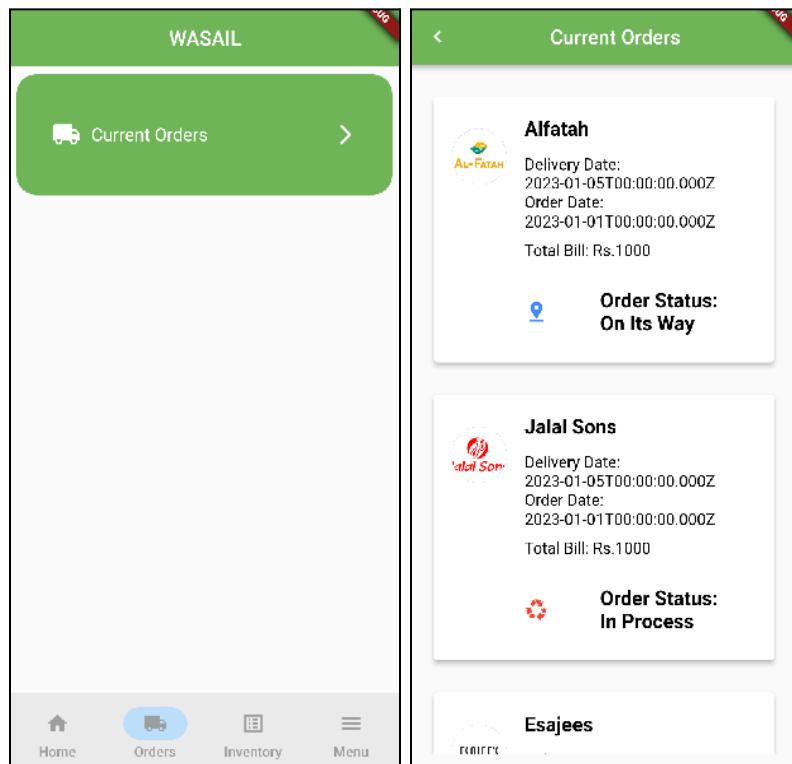


Figure 4.1.1.1.4 Current Orders

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 4.1.1.1.5): 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 4.1.1.1.5. 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.

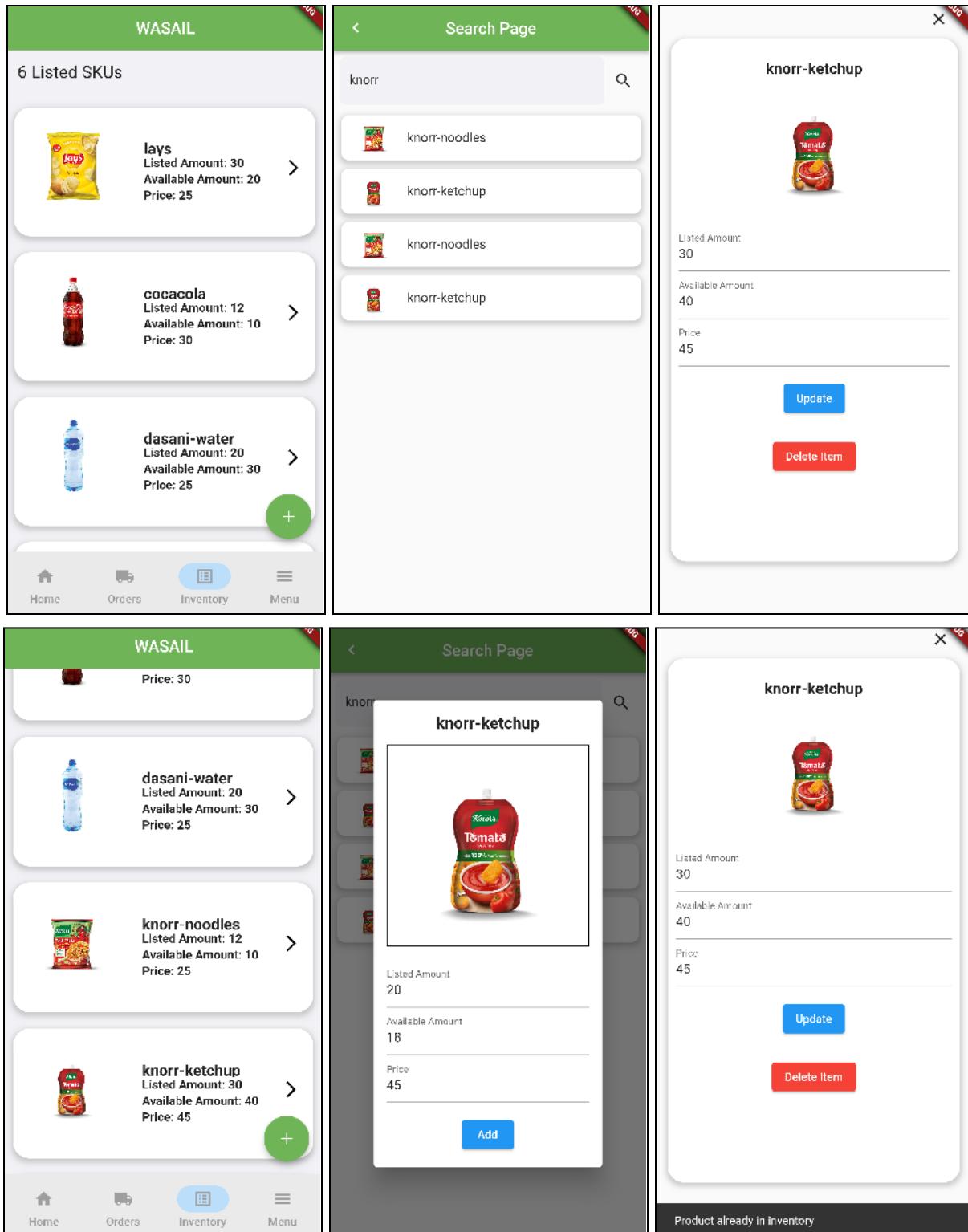


Figure 4.1.1.1.5 Inventory

Menu Page (Figure 4.1.1.1.6): 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.

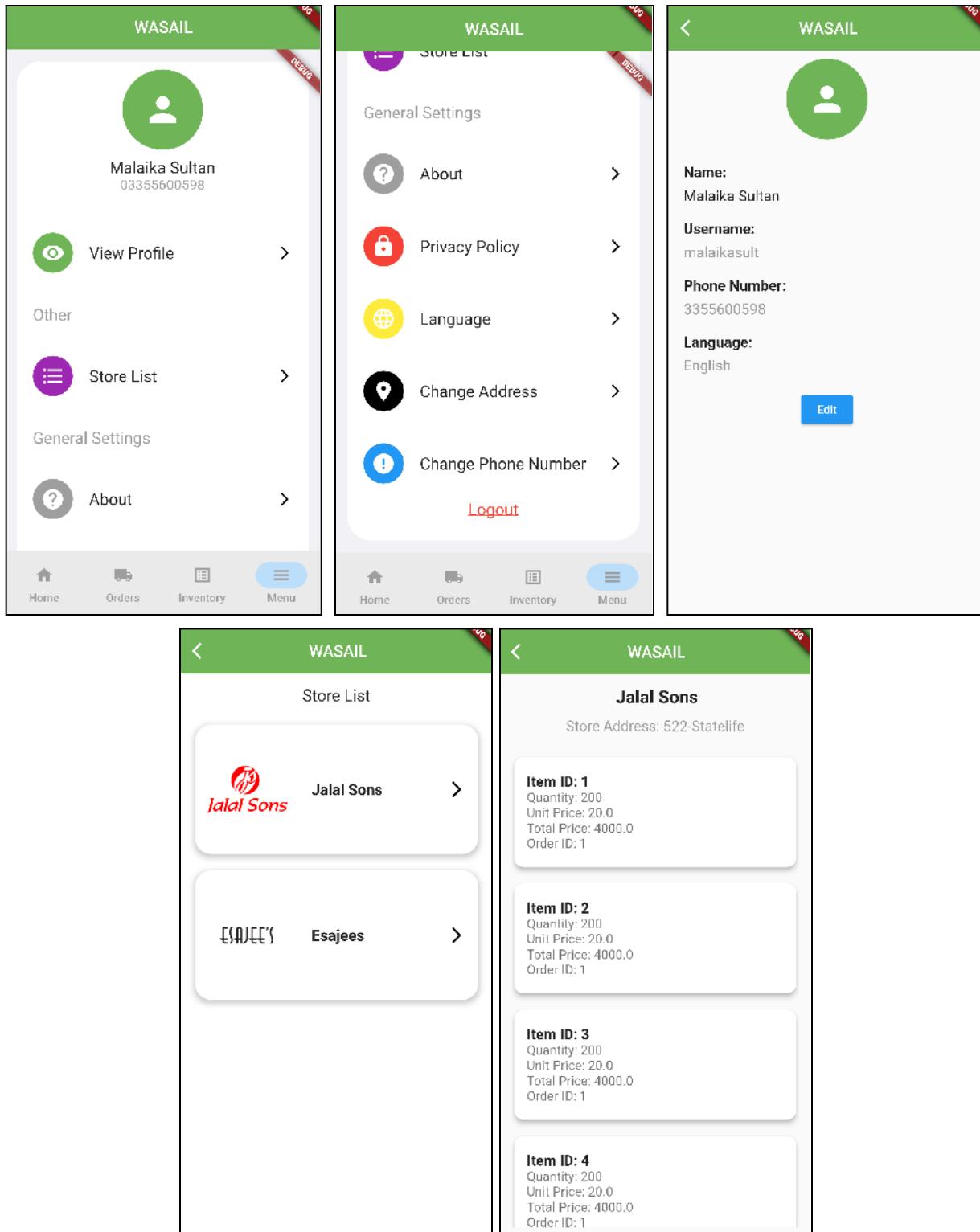


Figure 4.1.1.1.6 Menu

4.1.1.1.2 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.

Gesture-Based Interactions: Enhanced user engagement with gestures for dynamic language selection across all pages using Flutter's GestureDetector as shown in Figure 4.1.1.2.1.

Stateful Widgets: Achieved seamless transitions from language selection to user registration and login processes.

```
GestureDetector(
  onTap: () {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => OrderHistoryPage(vendorId),
      ), // MaterialPageRoute
    );
  },
},
```

Figure 4.1.1.2.1 Gesture Detector usage for accessing Order History in the Home Page

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

```

class NavBar extends StatefulWidget {
  final int selectedIndex;

  const NavBar({Key? key, this.selectedIndex = 0}) : super(key: key);

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

class _NavBarState extends State<NavBar> {
  late int index;

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

```

Figure 4.1.1.1.2.2 Navigation Bar using a Stateful Widget to handle Navigation

<pre> "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", </pre>	<pre> "app_name": "وسائل", "select_lang": "زبان منتخب کریں", "continue_button": "جاری رہے", "enter_number": "فون نمبر درج کریں", "welcome_msg": "وصیل میں خوش آمدید", "next_button": "اگلے مرحلے پر", "login": "لاگ ان" </pre>
--	--

Figure 4.1.1.1.2.3 English and Urdu .arb file snippets

```

Positioned(
  left: screenWidth * 0.1,
  top: screenHeight * 0.25,
  child: Text(
    AppLocalizations.of(context)!.login,
    style: TextStyle(
      color: Colors.black,

```

Figure 4.1.1.1.2.4 App context carrying the language selected and using its respective locale

```

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 4.1.1.1.2.5 Declarations of localisation needed to internationalise a page

Language Selection, Internationalisation, and State Management (Figure 4.1.1.1.2.6): 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.

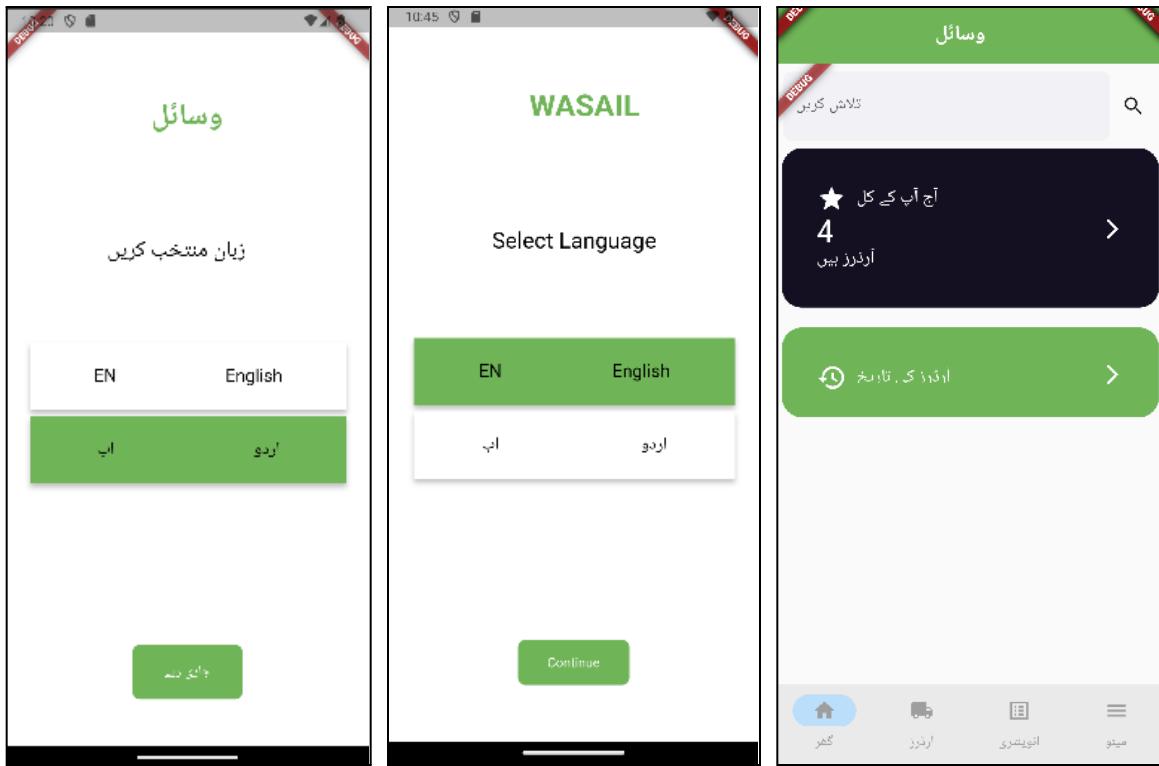


Figure 4.1.1.1.2.6 Overview of the app in Urdu

```
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(
      debugShowCheckedModeBanner: false,
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: AppLocalizations.supportedLocales,
```

Figure 4.1.1.1.2.7 Language locale provided by the provider

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 4.1.1.1.2.8.

```

class MyVendorApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider<LanguageProvider>(
          create: (context) => LanguageProvider(),
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<UserIdProvider>(
          create: (_) => UserIdProvider()
        ), // ChangeNotifierProvider
        ChangeNotifierProvider<PhoneNumberProvider>(
          create: (context) => PhoneNumberProvider(),
        ), // ChangeNotifierProvider
      ],
    );
  }
}

```

Figure 4.1.1.1.2.8 Provider used for various things across the app other than locale

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.

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 4.1.1.1.2.9 to 4.1.1.1.2.10): 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.

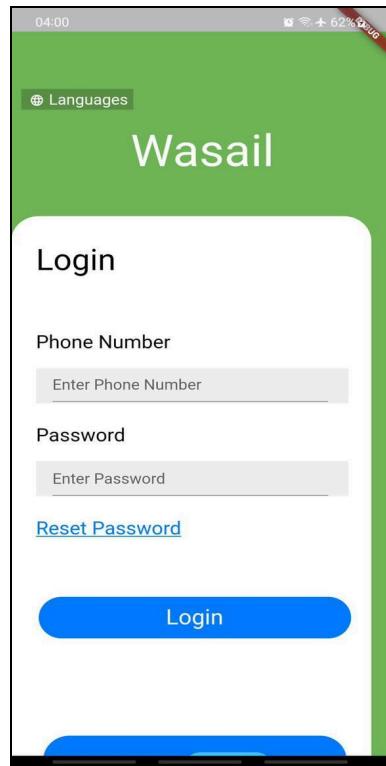


Figure 4.1.1.1.2.9 Login Page in the early stage being unresponsive on a different device

```

Widget build(BuildContext context) {
  double screenWidth = MediaQuery.of(context).size.width;
  double screenHeight = MediaQuery.of(context).size.height;
  return Consumer<LanguageProvider>(
    builder: (context, languageProvider, child) {
      return MaterialApp(
        debugShowCheckedModeBanner: false,
        localizationsDelegates: AppLocalizations.localizationsDelegates,
        supportedLocales: AppLocalizations.supportedLocales,
        locale: languageProvider.selectedLocale,
        builder: (context, child) {
          return Directionality(
            textDirection: TextDirection.ltr,
            child: child!,
          ); // Directionality
    },
  ),
}

```

Figure 4.1.1.2.10 MediaQuery Implementation

4.1.1.1.3 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.

To become familiar, we initially implemented a basic User CRUD application in Flutter as shown in Figure 4.1.1.1.3.1.

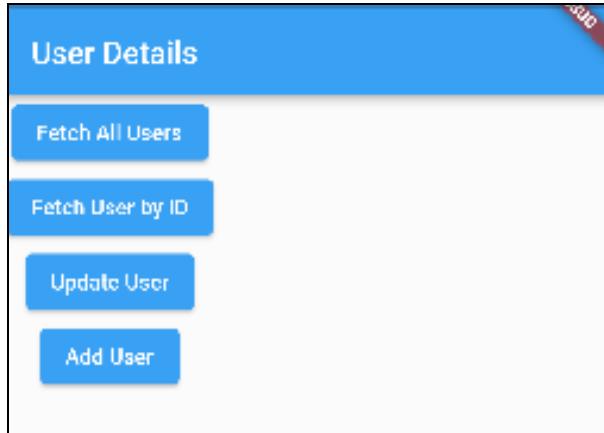


Figure 4.1.1.3.1 Basic User CRUD Implementation

Integrating required use of imports, libraries of Flutter, as seen in Figure 4.1.1.3.2 below, which were used in JSON encoding and decoding as well as HTTP protocol functions; GET, UPDATE, etc.

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

Figure 4.1.1.3.2 Imports were used for Integration

The operations looked as below in Figure 4.1.1.3.3 where Uri are the endpoints where information is being handled in the backend/database.

```
Future<void> _fetchAndDisplayProducts(String query) async {
  final response = await http.get(
    Uri.parse('https://sea-lion-app-wbl8m.ondigitalocean.app/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 4.1.1.3.3 GET operation on the Product table for Inventory Management

The tables involved JSON decoding and encoding which had syntax as shown in the Figures below for two different tables. Figure 4.1.1.3.4 is decoding an item from inventory and encoding a product into the product inventory.

```

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

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 4.1.1.3.4 JSON decoding and encoding

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.

4.1.1.4 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.

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 4.1.1.4.1 to Figure 4.1.1.4.3.



Figure 4.1.1.1.4.1 English and Urdu

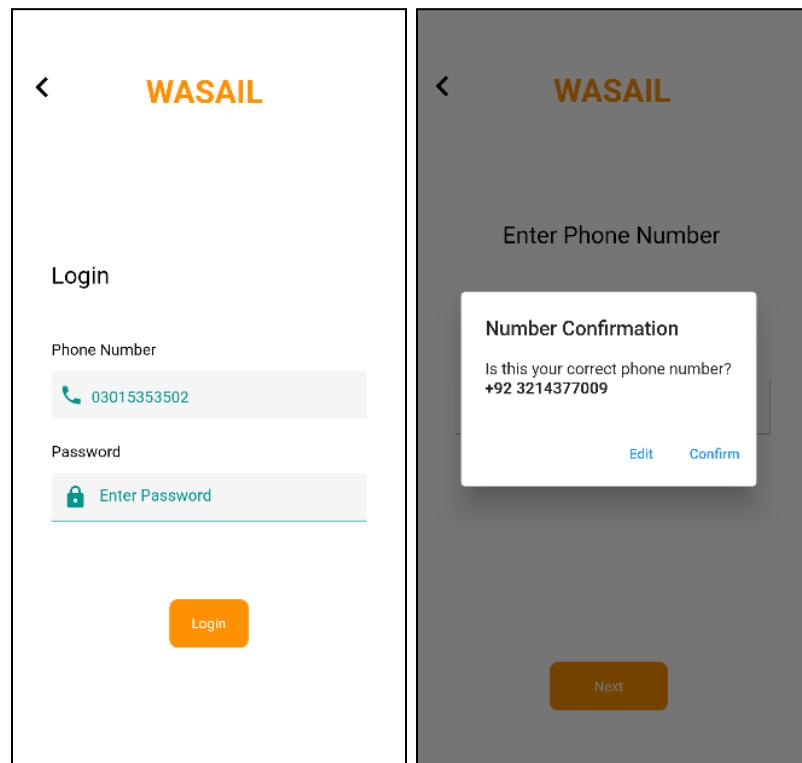


Figure 4.1.1.1.4.2 Login and Confirmation

The image consists of two side-by-side screenshots of a mobile application interface. Both screenshots feature a header with the word 'WASAIL' in orange and a back arrow icon.

Left Screenshot (OTP Screen):

- Header: WASAIL
- Text: Enter your OTP
- Text: Code sent to +92 3214377008
- Text: _____
- Text: _____
- Text: _____
- Text: _____
- Text: Re-send code 00:22
- Text: Submit

Right Screenshot (Registration Details Screen):

- Header: WASAIL
- Text: Enter Registration Details
- Text: Phone Number +92 3214377008
- Text: Password _____
- Text: Confirm Password _____ ✓
- Text: Full Name _____
- Text: Username _____
- Text: Delivery Areas Dha ▾
- Text: I agree to the [Terms and Conditions](#) and [Privacy Policy](#)

Figure 4.1.1.1.4.3 OTP and Registration

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 unauthorised access across different app functionalities as shown in the below Figure 4.1.1.1.4.4.

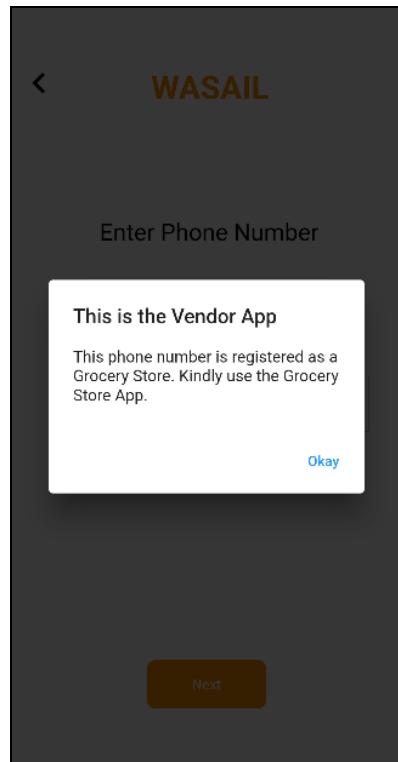


Figure 4.1.1.4.4 Error Message

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 4.1.1.4.5.

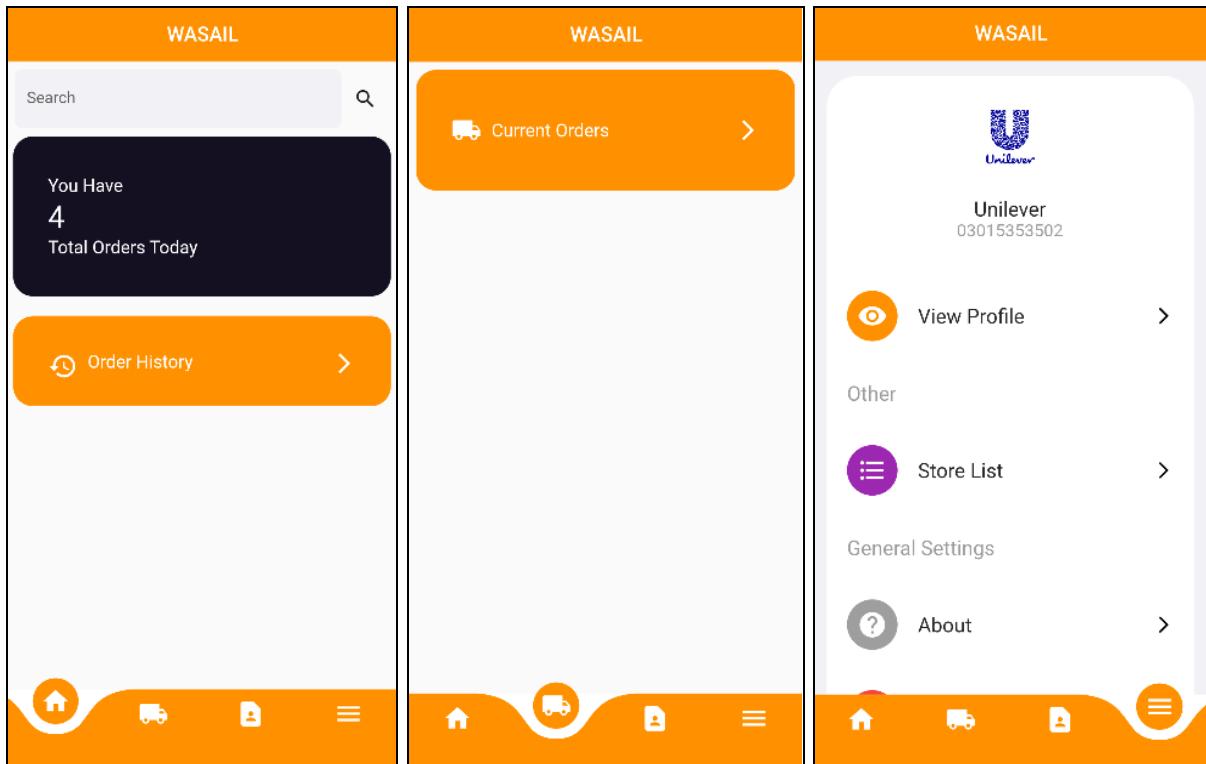


Figure 4.1.1.1.4.5 Home Page, Current Orders, and Menu

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 below Figure 4.1.1.1.4.6.

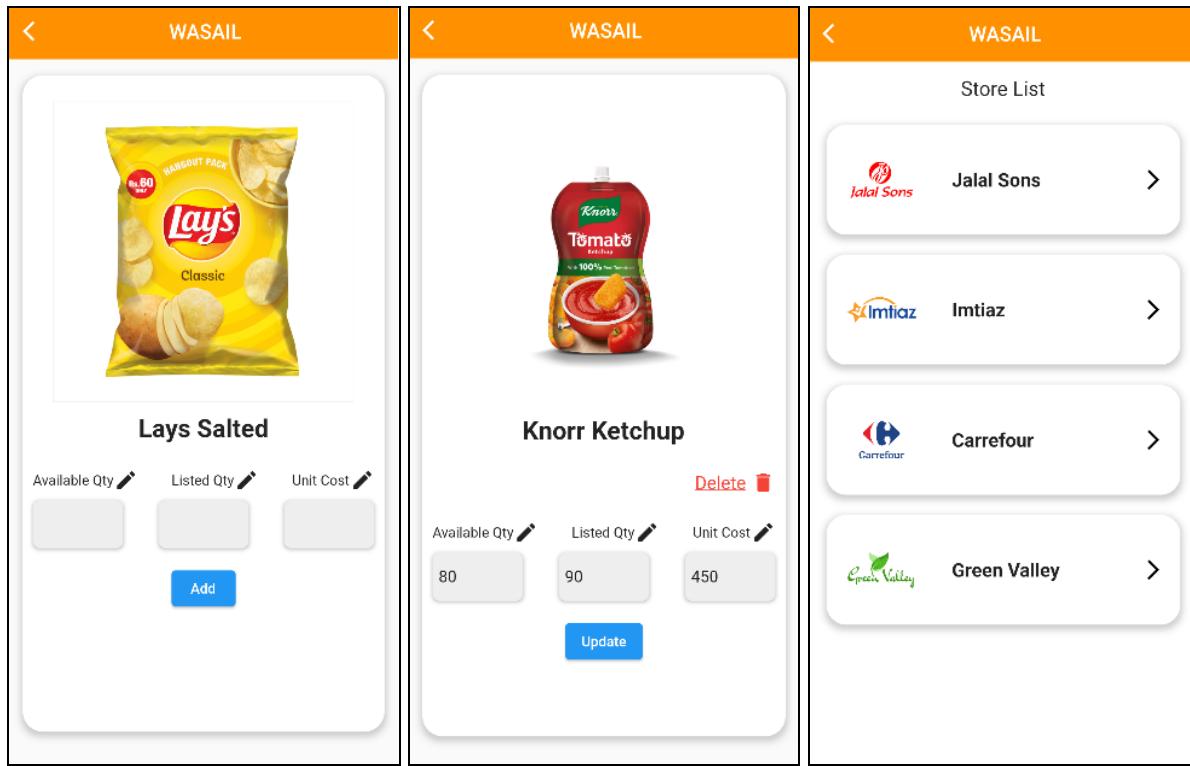


Figure 4.1.1.1.4.6 Add Product, Edit Product, and Store List

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 4.1.1.1.4.7.

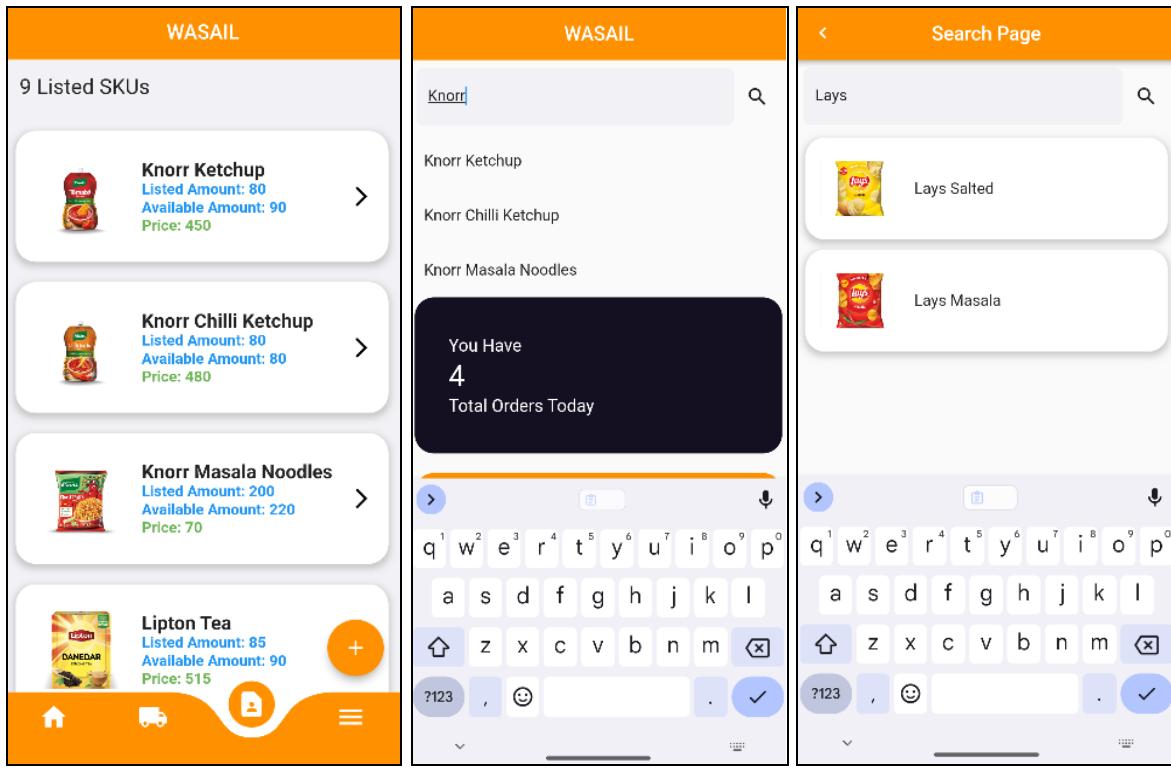


Figure 4.1.1.1.4.7 Inventory, Search Inventory, and Search Product

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 be seen in below Figure 4.1.1.1.4.8.

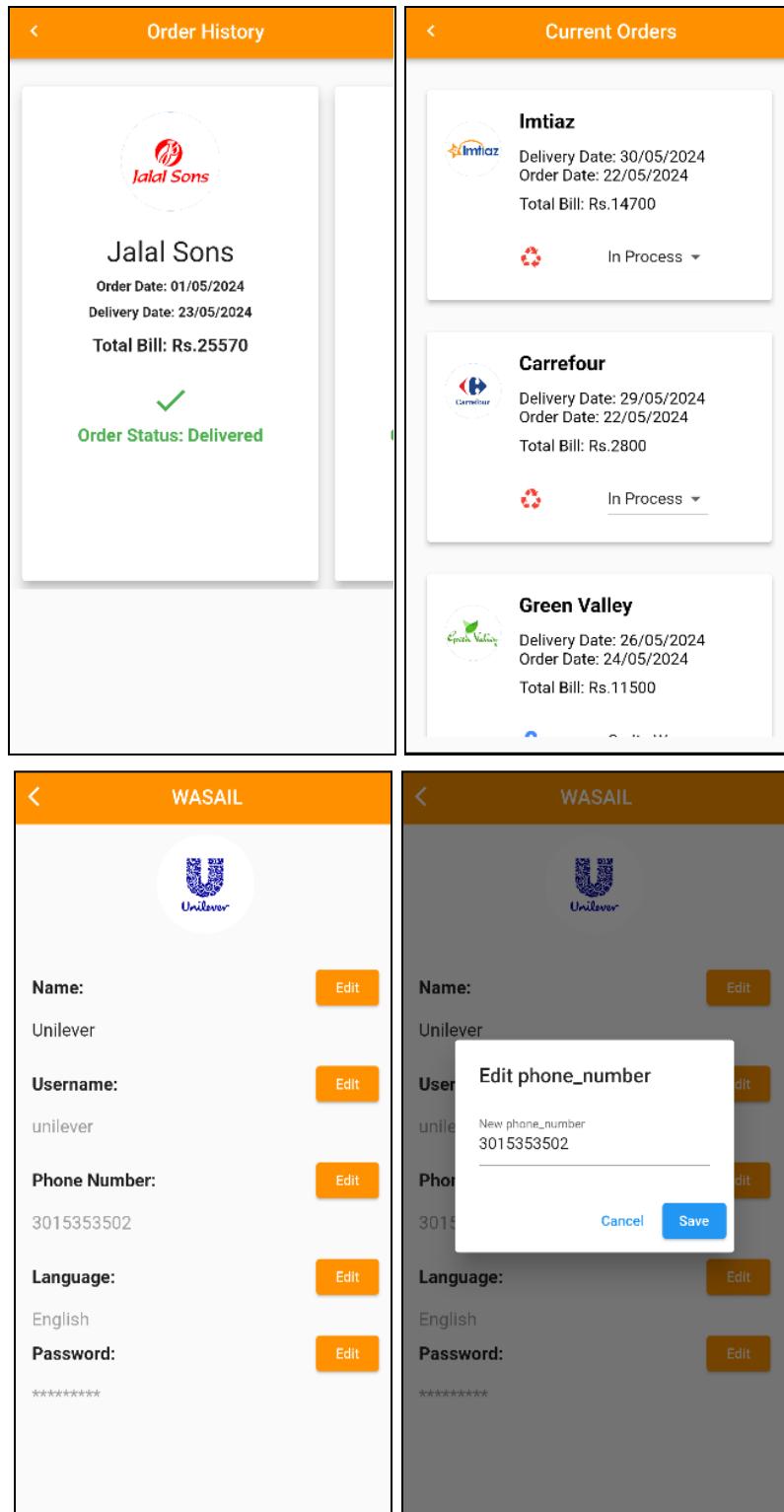


Figure 4.1.1.1.4.8 Order History, Current Orders, Edit Profile, and Edit Number

4.1.1.2 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 utilising Flutter, we aimed to streamline the development process and ensure a consistent user experience across different devices for grocery store owners.

4.1.1.2.1 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.

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 below Figure 4.1.1.2.1.1 and Figure 4.1.1.2.1.2,

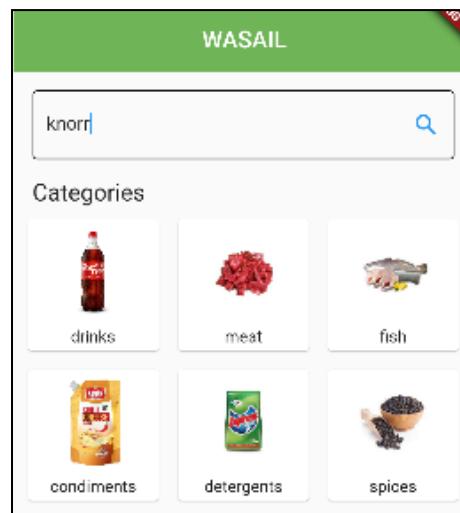


Figure 4.1.1.2.1.1 Search Product

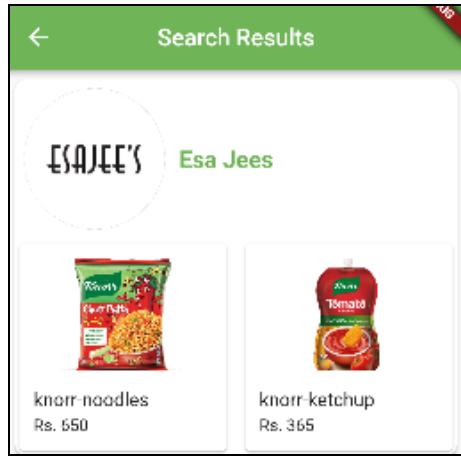


Figure 4.1.1.2.1.2 Search Results

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 4.1.1.2.1.3. 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.

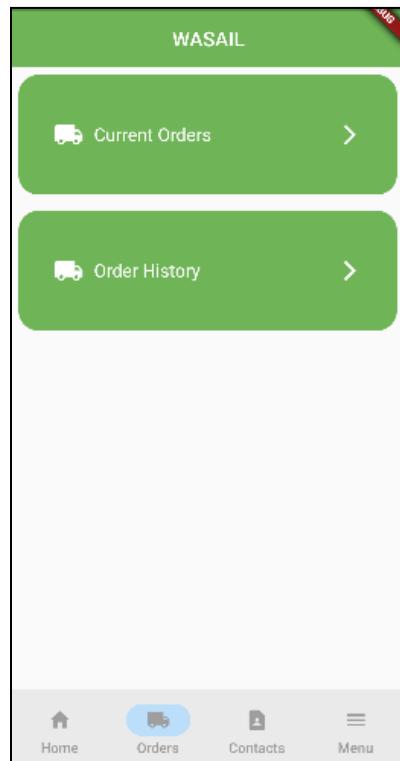


Figure 4.1.1.2.1.3 Current Orders & Order History Sections

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 4.1.1.2.1.4.

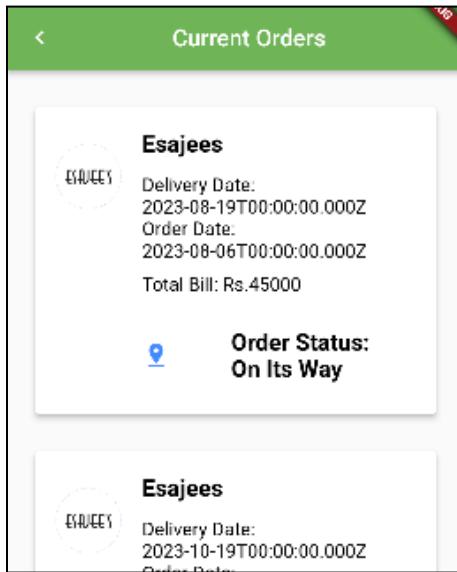


Figure 4.1.1.2.1.4 Current Orders

Subsequently, the page dynamically constructed a user-friendly interface, Figure 4.1.1.2.1.5, 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.

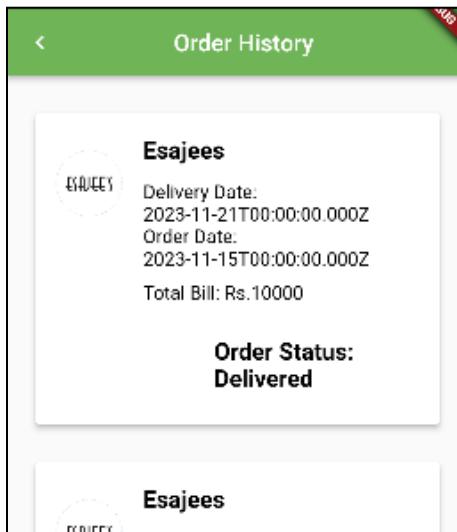


Figure 4.1.1.2.1.5 Order History

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 below Figures 4.1.1.2.1.6 to Figure 4.1.1.2.1.8.

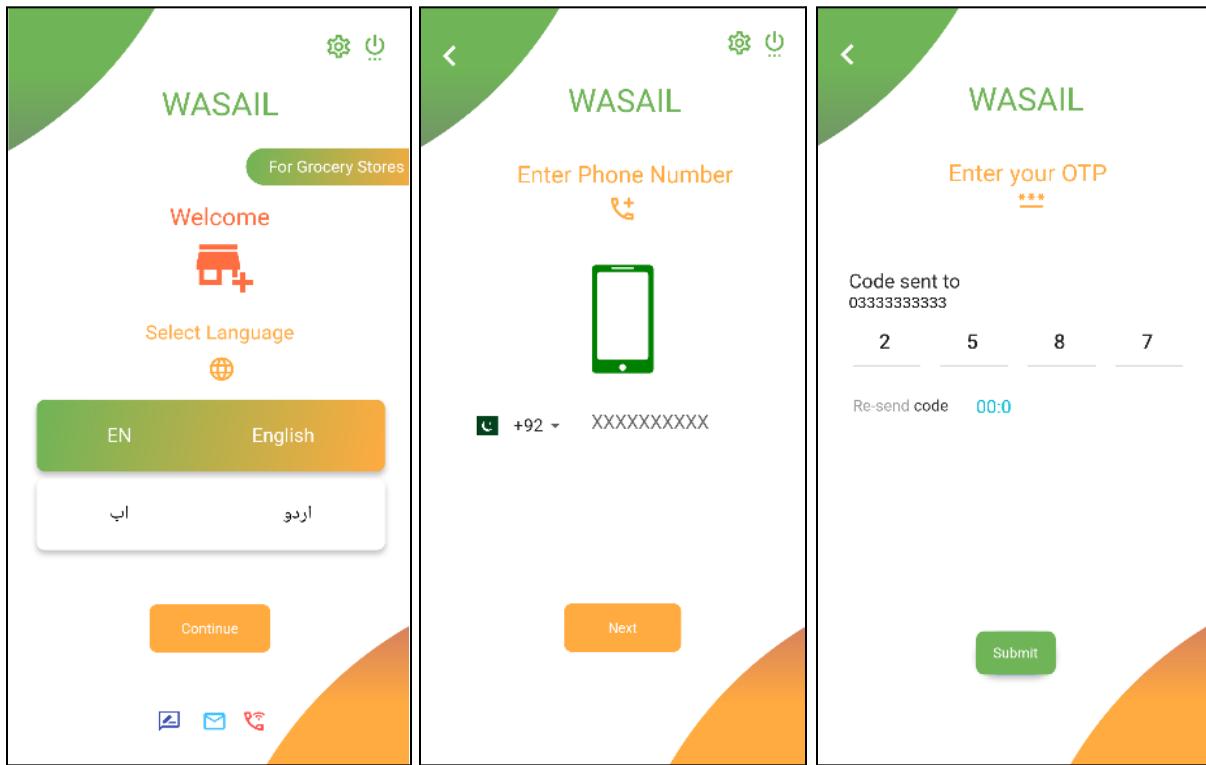


Figure 4.1.1.2.1.6 Languages, Phone Number, OTP

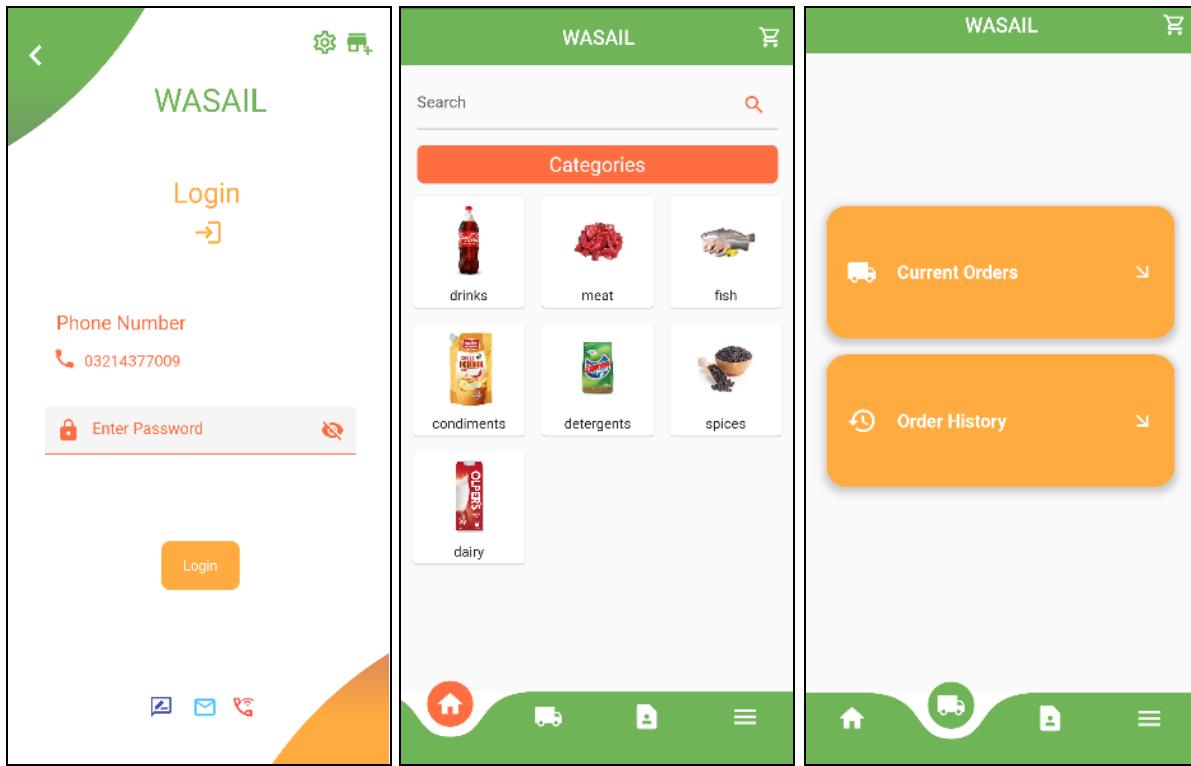


Figure 4.1.1.2.1.7 Login, Home, Orders

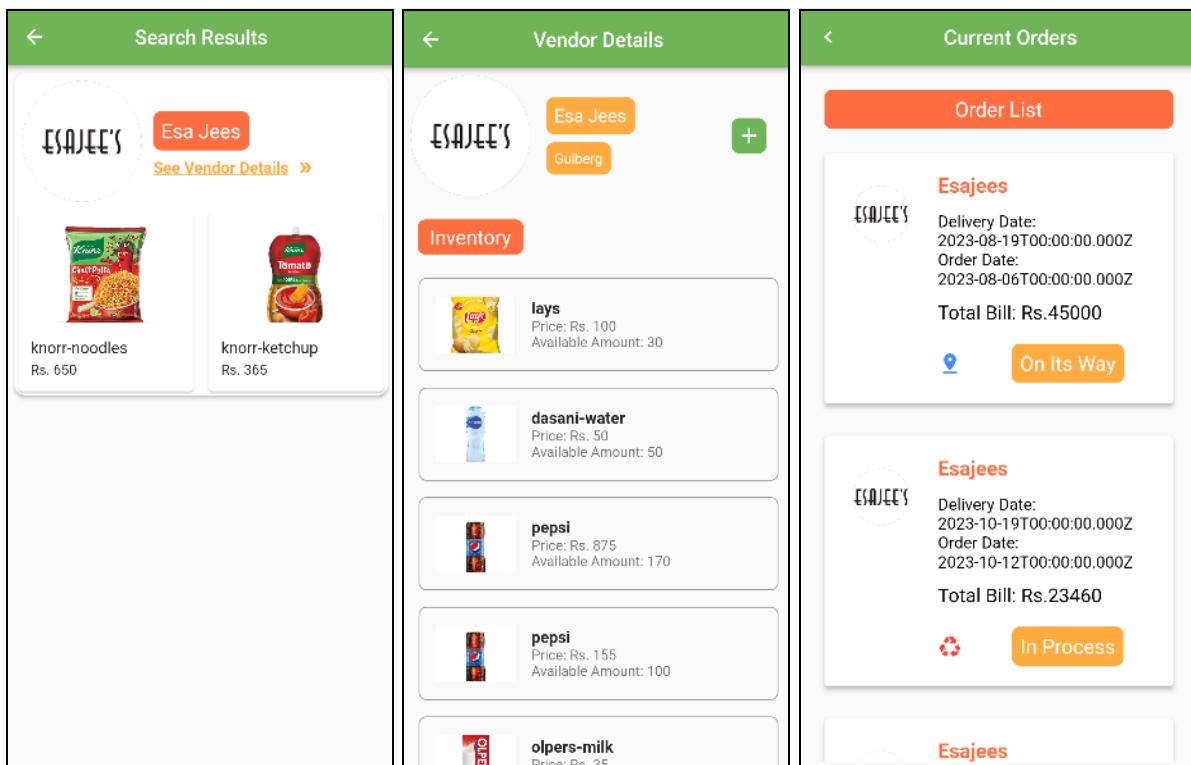


Figure 4.1.1.2.1.8 Results, Vendor Details, Current Orders

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 4.1.1.2.1.9.

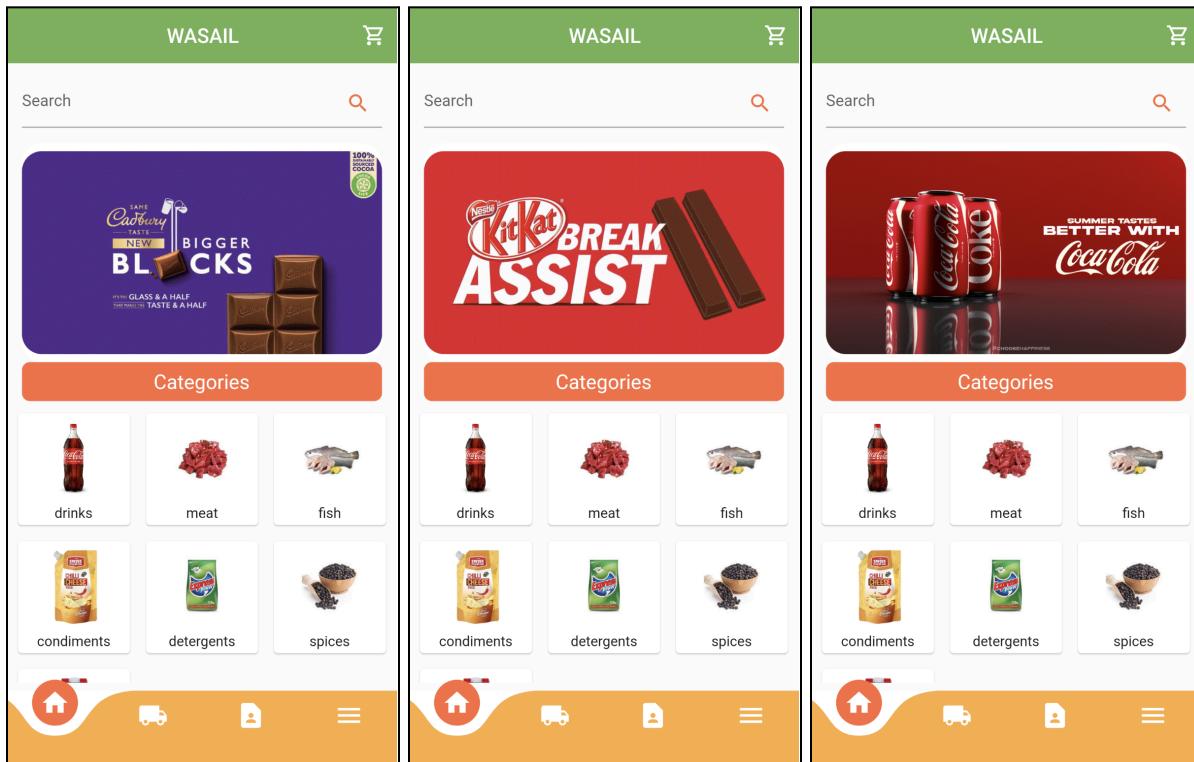


Figure 4.1.1.2.1.9 Home Page

4.1.1.2.2 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.

Seen below in Figure 4.1.1.2.2.1 is the Flutter Implementation of the SKU tiles shown above in Inventory List.

```

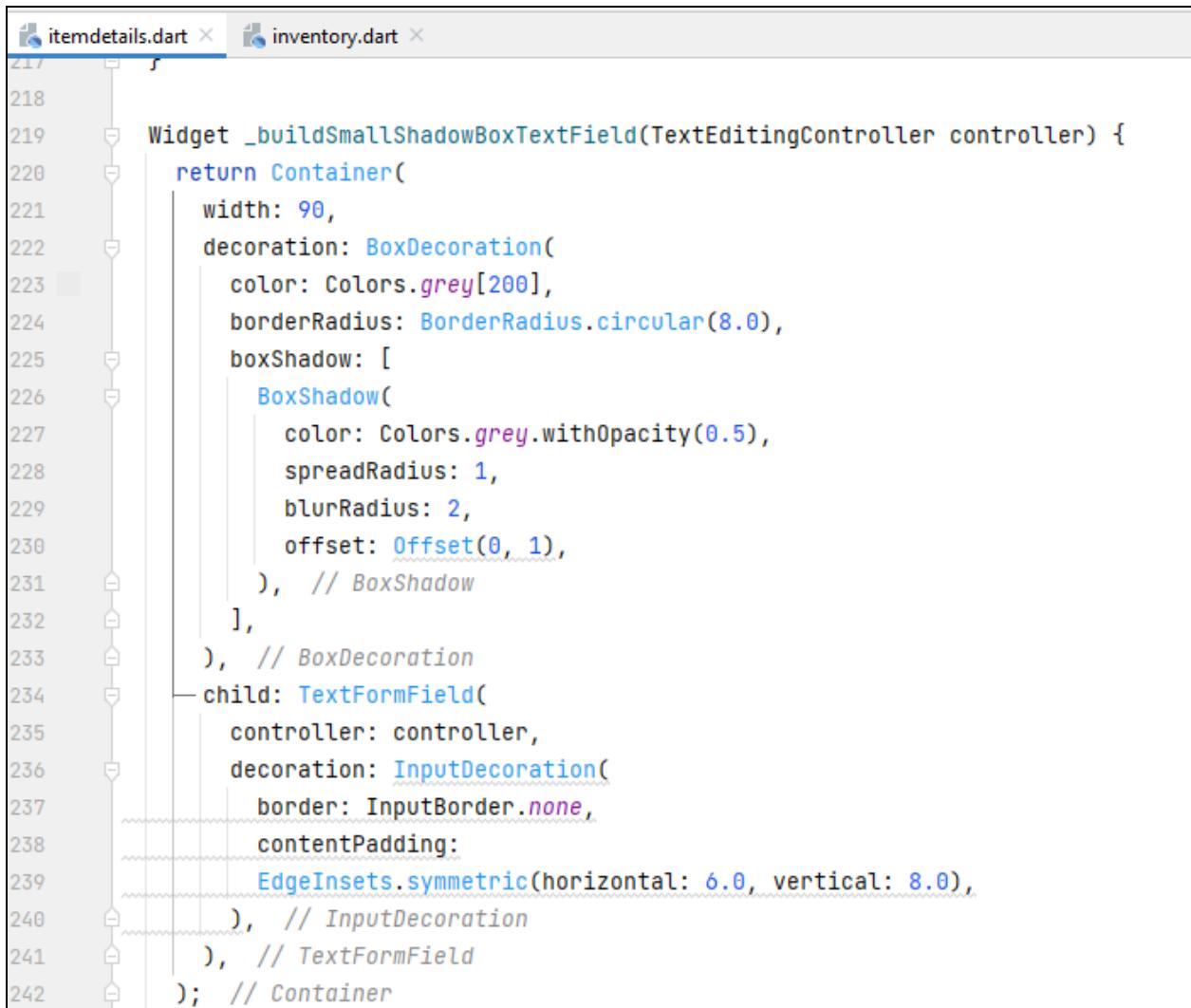
Expanded( // Use Expanded to allow the text to take available space
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Text(
        "${productItem.name}",
        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 19),
        maxLines: 2, // Limit to 2 lines
        overflow: TextOverflow.ellipsis, // Handle overflow with ellipsis
      ), // Text
      Text(
        "${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 4.1.1.2.2.1 Implementation of the SKU tiles

Seen below in Figure 4.1.1.2.2.2 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.



```
itemdetails.dart × inventory.dart ×
217
218
219     Widget _buildSmallShadowBoxTextField(TextEditingController controller) {
220         return Container(
221             width: 90,
222             decoration: BoxDecoration(
223                 color: Colors.grey[200],
224                 borderRadius: BorderRadius.circular(8.0),
225                 boxShadow: [
226                     BoxShadow(
227                         color: Colors.grey.withOpacity(0.5),
228                         spreadRadius: 1,
229                         blurRadius: 2,
230                         offset: Offset(0, 1),
231                     ), // BoxShadow
232                 ],
233             ), // BoxDecoration
234             child: TextFormField(
235                 controller: controller,
236                 decoration: InputDecoration(
237                     border: InputBorder.none,
238                     contentPadding:
239                         EdgeInsets.symmetric(horizontal: 6.0, vertical: 8.0),
240                 ), // InputDecoration
241             ), // TextFormField
242         ); // Container
```

Figure 4.1.1.2.2.2 Implementation of the text fields in Update Page

Seen below in Figure 4.1.1.2.2.3 is the Flutter Implementation of the text fields for input shown above in the Add To Inventory Page.

```

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 4.1.1.2.2.3 Implementation of the text fields for input in the Add To Inventory Page

4.1.1.2.3 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 utilisation of HTTP protocols for JSON data exchange. This integration was crucial in bringing the app to life and demonstrating its core features effectively.

The front-end and back-end integration for fetching vendor profiles as seen below was done through HTTP protocol as usual, the code snippet for it is attached above, Figure 4.1.1.2.3.1.

```

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 4.1.1.2.3.1 HTTP Function Vendor Profile

The HTTP function adding a vendor to ‘Vendor List’ implemented below can be seen in the attached Figure 4.1.1.2.3.2 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.

```

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 4.1.1.2.3.2 HTTP Function for Adding Vendor

As seen in Figure 4.1.1.2.3.3, 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 4.1.1.2.3.3, 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.

```

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 4.1.1.2.3.3 HTTP Request for Current Orders

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 4.1.1.2.3.4 below. Following a successful response, the page processes the received data, extracting vital details such as order dates, delivery dates, total bills, and order statuses.

```

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 4.1.1.2.3.4 HTTP Request for Order History

4.1.1.2.4 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.

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 4.1.1.2.4.1 and Figure 4.1.1.2.4.2.

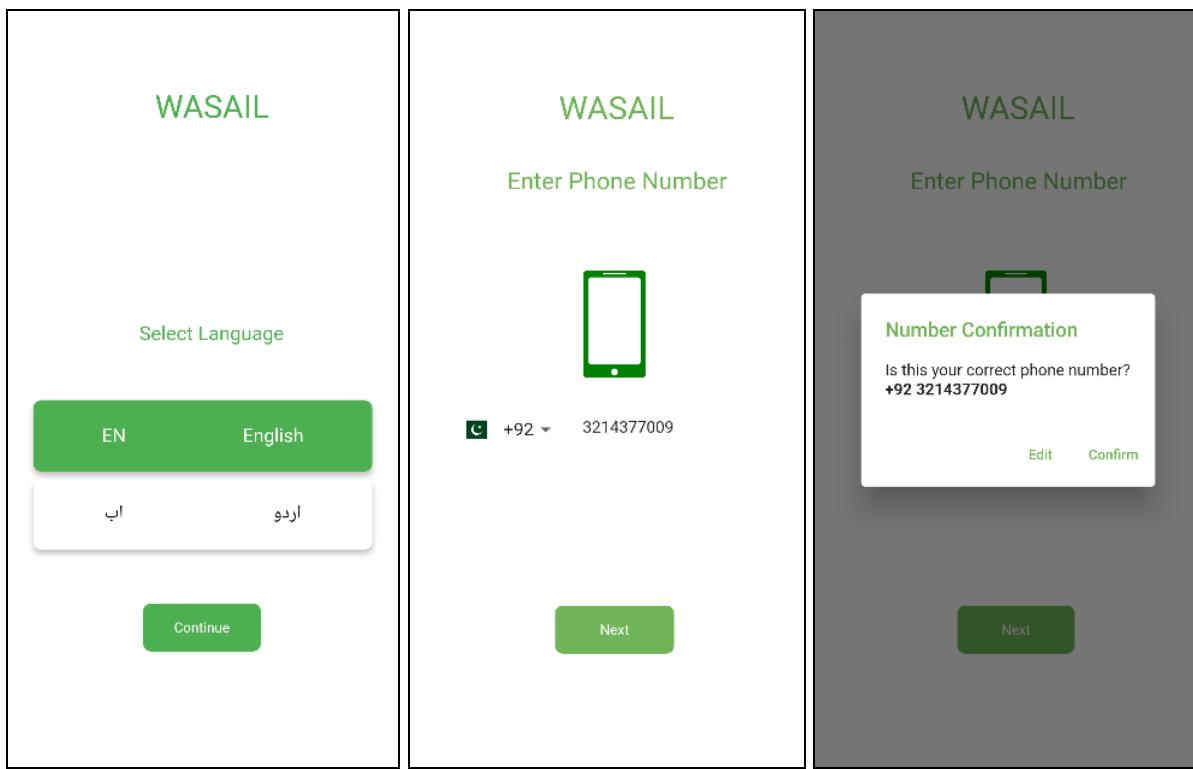


Figure 4.1.1.2.4.1 Languages, Phone Number, Confirmation

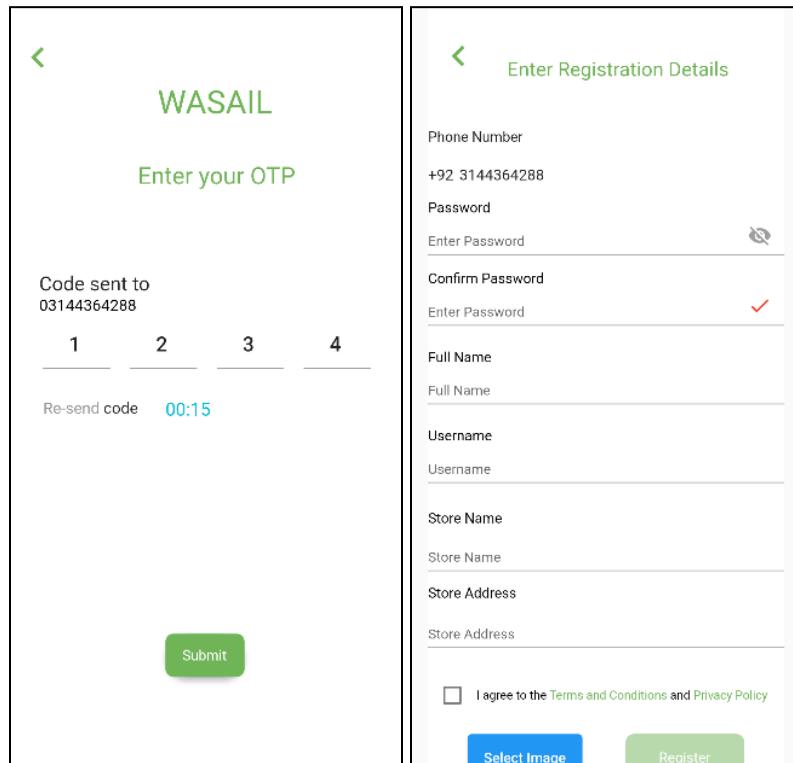


Figure 4.1.1.2.4.2 OTP, Registration

After successfully registering, new users are directed to the login screen. Entering the correct credentials will then grant access to the Homepage 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 4.1.1.2.4.3. 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 4.1.1.2.4.3 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 4.1.1.2.4.3. 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.

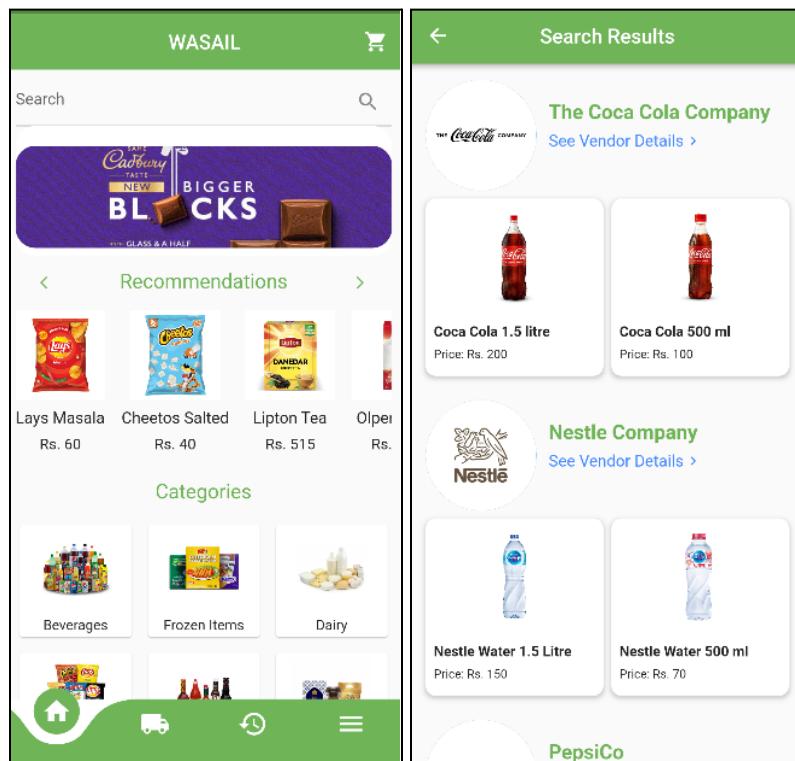


Figure 4.1.1.2.4.3 Home, Search Results

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 below Figure 4.1.1.2.4.4.

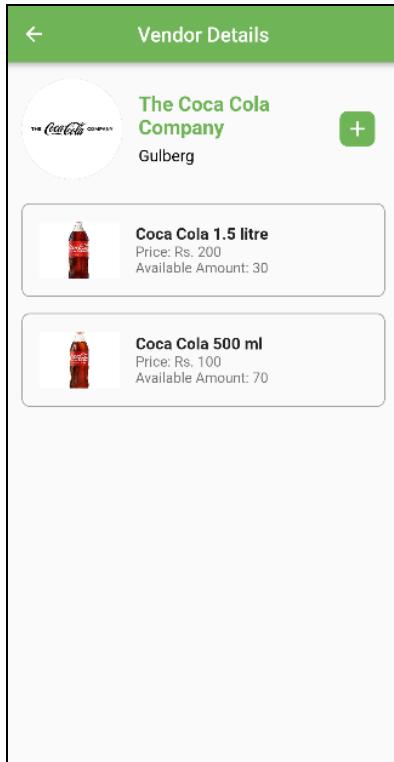


Figure 4.1.1.2.4.4 Vendor Details

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 4.1.1.2.4.5.

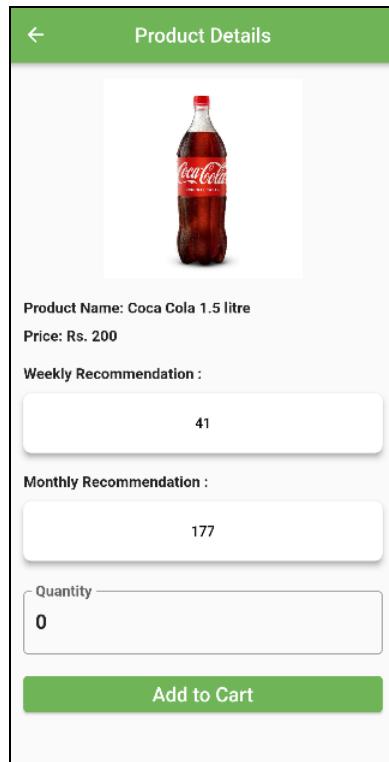


Figure 4.1.1.2.4.5 Vendor Details

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 below Figure 4.1.1.2.4.6.

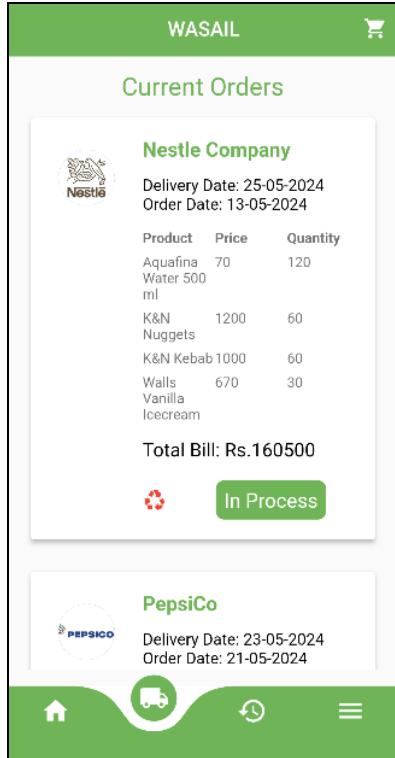


Figure 4.1.1.2.4.6 Current Orders

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 4.1.1.2.4.7.

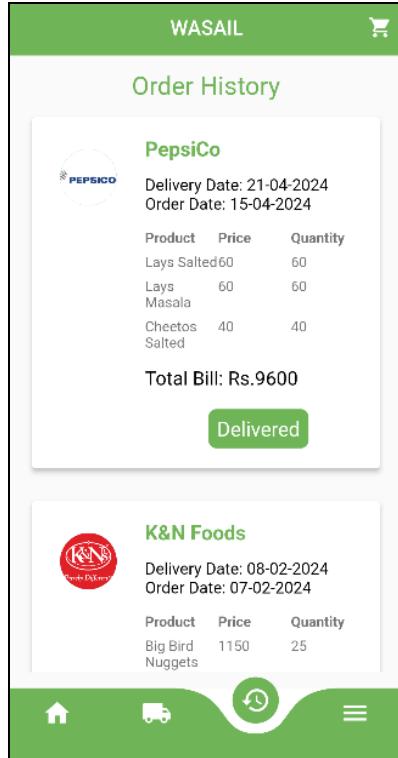


Figure 4.1.1.2.4.7 Order History

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 4.1.1.2.4.8. 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.

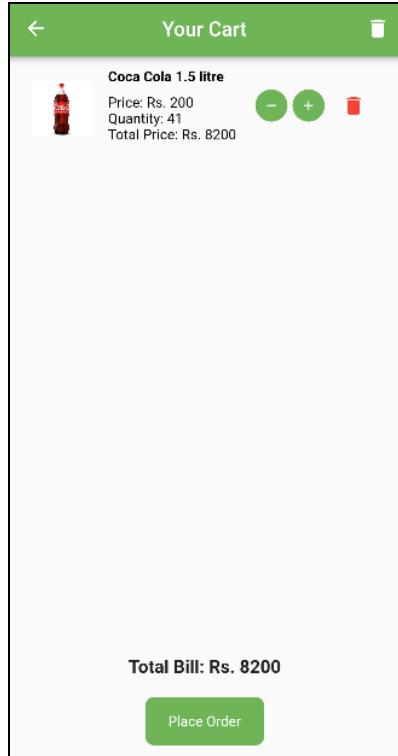


Figure 4.1.1.2.4.8 Cart

Finally ,On the menupage Figure 4.1.1.2.4.9 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 4.1.1.2.4.9. 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 4.1.1.2.4.10. This functionality provides users with control over their store and account management, ensuring a personalised and efficient experience within the platform.

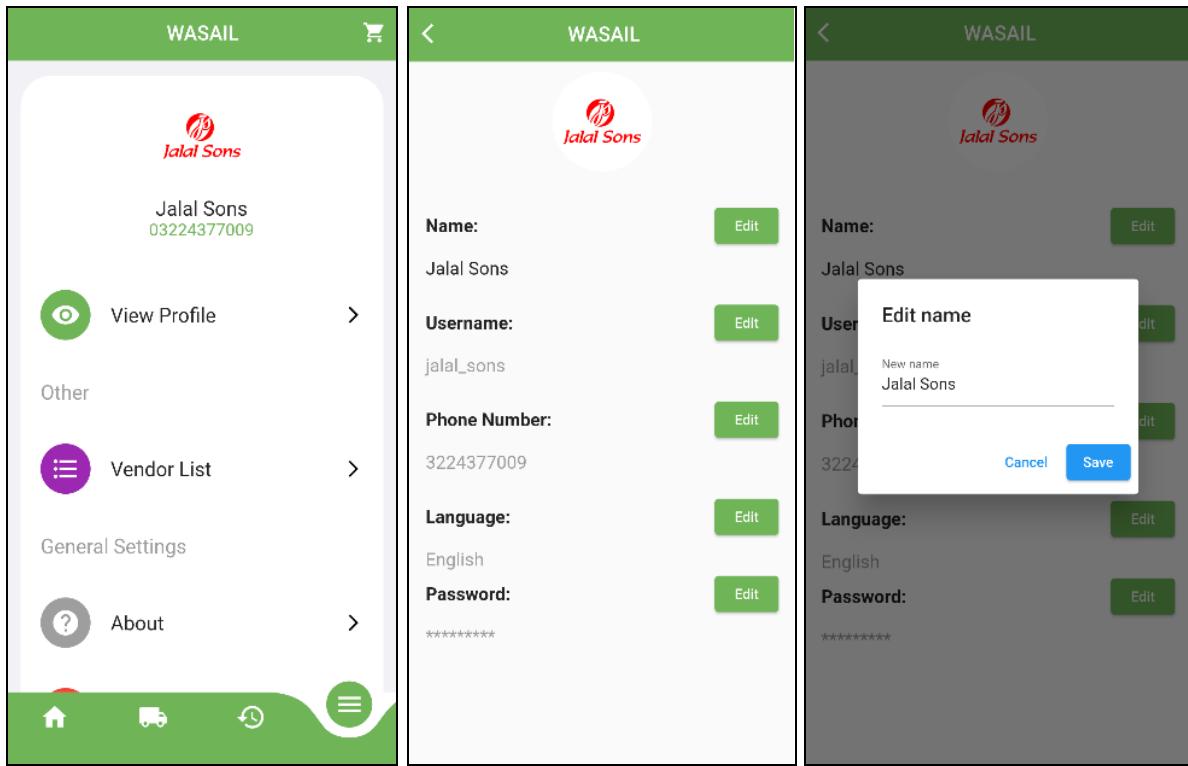


Figure 4.1.1.2.4.9 Menu, Profile, Edit

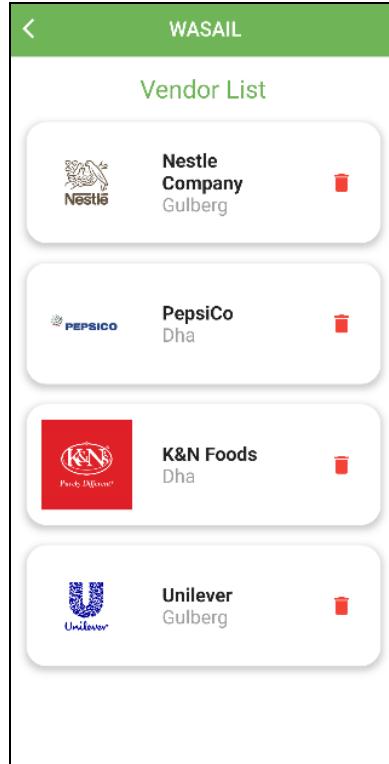


Figure 4.1.1.2.4.10 Cart

4.1.1.3 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.

4.1.1.3.1 Phase 1: Initial Front-End Design

Since the admin portal is intended for use by the Wasail team, it serves as a centralised 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 analyse 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.

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 4.1.1.3.1.1.

Users	Vendors	Groceries	Analytics	Machine Learning	Content
Users					
Email	Role	Options			
fatima@gmail.com	Administrator	Edit	Delete	View	
fizza@gmail.com	Moderator	Edit	Delete	View	
irtaza@gmail.com	Editor	Edit	Delete	View	
malaika@gmail.com	Viewer	Edit	Delete	View	

Figure 4.1.1.3.1.1 User Management

We created a list category page for our admin portal, maintaining consistency in the navbar design across the portal as shown in the below Figure 4.1.1.3.1.2.

The screenshot shows a web application interface for managing categories. At the top, there is a blue navigation bar with tabs: Users, Vendors, Groceries, Analytics, Machine Learning, and Content Management. The 'Content Management' tab is active. Below the navigation bar, the title 'List Category' is displayed, followed by a blue 'Add Category' button. A table lists various categories with their names in the first column and three buttons ('Edit', 'Delete', 'View') in the second column. The categories listed are Drink, Meat, Fish, Condiments, Detergents, Spices, and Dairy.

Category Name	Options
Drink	Edit Delete View
Meat	Edit Delete View
Fish	Edit Delete View
Condiments	Edit Delete View
Detergents	Edit Delete View
Spices	Edit Delete View
Dairy	Edit Delete View

[List Products](#)

Figure 4.1.1.3.1.2 Content Management (List 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 4.1.1.3.1.3.

The screenshot shows a 'Create Category' form. It features a title 'Create Category' at the top, followed by a text input field labeled 'Enter category name'. Below the input field are two buttons: 'Save' and 'Back'. The entire form is enclosed in a black-bordered box.

Figure 4.1.1.3.1.3 Content Management (Add Category)

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 4.1.1.3.1.4.

List Products					
Add Product					
Image	Product Name	Options			
	Lays	Edit	Delete	View	
	Coca Cola	Edit	Delete	View	
	Dasani Water	Edit	Delete	View	
	Pepsi	Edit	Delete	View	
	Pepsi	Edit	Delete	View	
	Olpers Milk	Edit	Delete	View	

Figure 4.1.1.3.1.4 Content Management (List 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 4.1.1.3.1.5.

Add Product					
<input type="text" value="Enter product name"/>					
<input type="file" value="Upload Image"/>					
Save Back					

Figure 4.1.1.3.1.5 Content Management (Add Product)

On the website, the vendor listing was displayed as shown in Figure 4.1.1.3.1.6 below.

Vendors					
Add Vendor					
Icon	Vendor Name	Delivery Location	Options		
	P&G	Gulberg	Edit	Delete	View

Figure 4.1.1.3.1.6 Vendor Listing

On the website, the vendor addition was displayed as shown in Figure 4.1.1.3.1.7 below.

Figure 4.1.1.3.1.7 Vendor Addition

On the website, the grocery store listing was displayed as shown in Figure 4.1.1.3.1.8 below.

Stores				
	Icon	Store Name	Store Address	Options
		Esajees	Shop 175, Y Block Market, DHA Phase 3	<button>Edit</button> <button>Delete</button> <button>View</button>
		Jalal Sons	Shop 5, H Block Market, DHA Phase 2	<button>Edit</button> <button>Delete</button> <button>View</button>
		Alfatah	Shop 30, A Block Market, State Life	<button>Edit</button> <button>Delete</button> <button>View</button>
	Store Icon	My Store	DHA Phase 3	<button>Edit</button> <button>Delete</button> <button>View</button>

Figure 4.1.1.3.1.8 Grocery Store Listing

On the website, the grocery store addition was displayed as shown in Figure 4.1.1.3.1.9 below.

Figure 4.1.1.3.1.9 Grocery Store Addition

4.1.1.3.2 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.

We implemented functionality for adding, viewing, and deleting data by incorporating corresponding buttons into the admin portal as seen in Figure 4.1.1.3.2.1.

```

{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> &nbsp;
      <Link to={`/`} className="btn btn-danger" style={{ marginLeft: '5px' }}>Delete</Link> &nbsp;
      <Link to={`/`} className="btn btn-primary" style={{ marginLeft: '5px' }}>View</Link> &nbsp;
    </td>
  </tr>
))}
```

Figure 4.1.1.3.2.1 Add,Delete,View

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 4.1.1.3.2.2.

```

const AddCategory = ({ addCategory }) => {
  const [categoryName, setCategoryName] = useState(initialState: '');
  const location = useLocation();

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

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

Figure 4.1.1.3.2.2 Create Category 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 Figure 4.1.1.3.2.3.

```

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 4.1.1.3.2.3 Create Product Function

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. Below is Figure 4.1.1.3.2.4 showing the vendor addition form.

```

3 usages  ± malaikasulant
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  ± malaikasulant
const isActive = () :boolean  => location.pathname === '/content-management';

1 usage  ± malaikasulant
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 4.1.1.3.2.4 Vendor 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 4.1.1.3.2.5 is the code snippet for the grocery store 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 4.1.1.3.2.5 Grocery Store Addition Code Snippet

4.1.1.3.3 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 utilising HTTP protocols for JSON data exchange. This integration was essential in bringing the portal to life and effectively demonstrating its core features.

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 4.1.1.3.3.1.

```

const ListContentManagementComponent = ({} ) => {
  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 4.1.1.3.3.1 Fetching Categories

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 4.1.1.3.3.2.

```

3 usages ± malaikasulant +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 ± malaikasulant +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 4.1.1.3.3.2 Fetching Products

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 4.1.1.3.3.3 provided below illustrates how the vendor list is rendered.

```

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 4.1.1.3.3.3 Vendor Listing Code 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. Below is Figure 4.1.1.3.3.4 showcasing how the grocery store list is rendered.

```

3 usages  ± malaikasulant
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  ± malaikasulant
        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 4.1.1.3.3.4 Grocery Listing Snippet

4.1.1.3.4 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.

In the admin page, Users who are members of the Wasail team, are listed in a tabular form as shown in Figure 4.1.1.3.4.1. 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 4.1.1.3.4.2. Similarly, we can also delete users or update their information as needed as shown in Figure 4.1.1.3.4.3, 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.

WASAIL		Admin Management					
	Admins						Add Admins
	Grocery Stores						Search users <input type="text"/> Search
	Vendors	Name	Username	Role	Email	Phone Number	Options
	Analytics	Fizza Adeel	fizza	Moderator	f2020-336@bnu.edu.pk	3218829929	<button>Update</button> <button>Delete</button>
	Categories	Fatima Ali	fatima	Editor	f2020-718@bnu.edu.pk	3224766880	<button>Update</button> <button>Delete</button>
	Products	Malaika Sultan	malaika	Viewer	f2020-661@bnu.edu.pk	3057877887	<button>Update</button> <button>Delete</button>
		Irtaza Ahmed	irtaza	Viewer	f2020-153@bnu.edu.pk	3225454503	<button>Update</button> <button>Delete</button>

Figure 4.1.1.3.4.1 Admin Management

Add New Admin

Phone Number	<input type="text"/>
Name	<input type="text"/>
Password	<input type="text"/>
Username	<input type="text"/>
Language	<input type="text"/> Select Language ▾
User Type	<input type="text"/>
Role	<input type="text"/> Select Role ▾
Email	<input type="text"/>

Figure 4.1.1.3.4.2 Add New Admin

Update Admin	
Admin Role	<input type="text" value="Editor"/>
Admin Email	<input type="text" value="f2020-718@bnu.edu.pk"/>
Phone Number	<input type="text" value="3224766880"/>
Name	<input type="text" value="Fatima Ali"/>
Password	<input type="text" value="....."/>
Username	<input type="text" value="fatima"/>
Language	<input type="text" value="English"/>
User Type	<input type="text" value="Admin"/>

Figure 4.1.1.3.4.3 Update 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 4.1.1.3.4.4. This capability streamlines user management tasks, making it easier to find and manage admins within the Wasail team.

WASAIL		Admin Management																	
		Add Admins <div style="float: right;"> <input type="text" value="Editor"/> <input type="button" value="Search"/> </div>																	
		<table border="1"> <thead> <tr> <th>Name</th><th>Username</th><th>Role</th><th>Email</th><th>Phone Number</th><th>Options</th></tr> </thead> <tbody> <tr> <td>Fatima Ali</td><td>fatima</td><td>Editor</td><td>f2020-718@bnu.edu.pk</td><td>3224766880</td><td> <input type="button" value="Update"/> <input type="button" value="Delete"/> </td></tr> </tbody> </table>						Name	Username	Role	Email	Phone Number	Options	Fatima Ali	fatima	Editor	f2020-718@bnu.edu.pk	3224766880	<input type="button" value="Update"/> <input type="button" value="Delete"/>
Name	Username	Role	Email	Phone Number	Options														
Fatima Ali	fatima	Editor	f2020-718@bnu.edu.pk	3224766880	<input type="button" value="Update"/> <input type="button" value="Delete"/>														
	Admins																		
	Grocery Stores																		
	Vendors																		
	Analytics																		
	Categories																		
	Products																		

Figure 4.1.1.3.4.4 Search Admin

In store management, all the stores stored in the database are presented in a structured tabular format as shown in Figure 4.1.1.3.4.5. 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 4.1.1.3.4.6.

WASAIL		Store Management					
	Admins					Search store	Search
↳	Grocery Stores	Image	Store Name	Store Address	Model	Options	
↳	Vendors		Esajees	Shop 175, Y Block Market, DHA Phase 3	LightGBM	<button>Update</button>	<button>Delete</button>
↳	Analytics		Jalal Sons	Shop 5, H Block Market, DHA Phase 2	XGBoost	<button>Update</button>	<button>Delete</button>
↳	Categories		Alfatah	Shop 30, A Block Market, State Life	Prophet	<button>Update</button>	<button>Delete</button>
↳	Products		Imtiaz	Shop 301, Lalik Chowk, DHA Phase 3	LightGBM	<button>Update</button>	<button>Delete</button>
			Carrefour	Shop 29, Main Boulevard, Gulberg II	Prophet	<button>Update</button>	<button>Delete</button>

Figure 4.1.1.3.4.5 Store Management

Update Store

Store Name	<input type="text" value="Esajees"/>
Store Address	<input type="text" value="Shop 175, Y Block Market, DHA Phase 3"/>
Model	<input type="text" value="LightGBM"/>
	<button>Back</button> <button>Update Store</button>

Figure 4.1.1.3.4.6 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 4.1.1.3.4.7.

The screenshot shows the 'Store Management' section of the WASAIL application. On the left, there is a sidebar with navigation links: Admins, Grocery Stores, Vendors, Analytics, Categories, and Products. The main area is titled 'Store Management' and contains a table with two rows. The table has columns for 'Image', 'Store Name', 'Store Address', 'Model', and 'Options'. The first row corresponds to 'Shop 301, Lalik Chowk, DHA Phase 3' with 'LightGBM' as the model. The second row corresponds to 'Shop 4, D Block Market, State Life' with 'Prophet' as the model. Each row includes 'Update' and 'Delete' buttons. A search bar at the top right contains the text 'imtiaz'.

Image	Store Name	Store Address	Model	Options
	Imtiaz	Shop 301, Lalik Chowk, DHA Phase 3	LightGBM	<button>Update</button> <button>Delete</button>
	Imtiaz	Shop 4, D Block Market, State Life	Prophet	<button>Update</button> <button>Delete</button>

Figure 4.1.1.3.4.7 Update Store

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 4.1.1.3.4.8.

The screenshot shows the 'Vendor Management' section of the WASAIL application. On the left, there is a sidebar with navigation links: Admins, Grocery Stores, Vendors, Analytics, Categories, and Products. The main area is titled 'Vendor Management' and contains a table with five rows. The table has columns for 'Image', 'Vendor Name', 'Delivery Locations', and 'Options'. The vendors listed are 'The Coca Cola Company' (Gulberg), 'Nestle Company' (Gulberg), 'PepsiCo' (Dha), 'K&N Foods' (Dha), and 'Big Bird Foods' (State Life). Each row includes a 'Delete' button. A search bar at the top right contains the placeholder text 'Search vendor'.

Image	Vendor Name	Delivery Locations	Options
	The Coca Cola Company	Gulberg	<button>Delete</button>
	Nestle Company	Gulberg	<button>Delete</button>
	PepsiCo	Dha	<button>Delete</button>
	K&N Foods	Dha	<button>Delete</button>
	Big Bird Foods	State Life	<button>Delete</button>

Figure 4.1.1.3.4.8 Vendor Management

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 4.1.1.3.4.9.

The screenshot shows the WASAIL vendor management interface. On the left, there is a sidebar with navigation links: Admins, Grocery Stores, Vendors, Analytics, Categories, and Products. The main area is titled "Vendor Management" and contains a search bar with a magnifying glass icon and a "Search" button. Below the search bar is a table with four columns: Image, Vendor Name, Delivery Locations, and Options. The table has one row showing "K&N Foods" with delivery locations "Dha" and a red "Delete" button.

Figure 4.1.1.3.4.9 Search Vendor

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 below Figure 4.1.1.3.4.10.

The screenshot shows the WASAIL analytics interface. On the left, there is a sidebar with navigation links: Admins, Grocery Stores, Vendors, Analytics, Categories, and Products. The main area is titled "Analytics" and displays four large numbers representing counts: 11 for Grocery Store Count, 7 for Vendor Count, 25 for Product Count, and 7 for Category Count.

Figure 4.1.1.3.4.10 Analytics

In category management, we have comprehensive functionality for managing categories efficiently as shown in Figure 4.1.1.3.4.11. Users can add new categories to the system (Figure 4.1.1.3.4.12), update existing categories (Figure 4.1.1.3.4.13) or delete categories as needed.

WASAIL		Category Management		
	Admins			Add Category
	Grocery Stores			Search category <input type="text"/> Search
	Vendors	Image	Category Name	Options
	Analytics		Beverages	<button>Update</button> <button>Delete</button>
	Categories		Frozen Items	<button>Update</button> <button>Delete</button>
	Products		Dairy	<button>Update</button> <button>Delete</button>
			Edibles	<button>Update</button> <button>Delete</button>
			Condiments	<button>Update</button> <button>Delete</button>

Figure 4.1.1.3.4.11 Category Management

Add Category

Category Name	<input type="text"/>
Category Image	<input type="file"/> Choose File No file chosen

[Back](#) [Add Category](#)

Figure 4.1.1.3.4.12 Add Category

Update Category

Category Name	<input type="text" value="Beverages"/>
Category Image	<input type="button" value="Choose File"/> No file chosen
Back Update Category	

Figure 4.1.1.3.4.13 Update 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 4.1.1.3.4.14.

WASAIL		Category Management								
✉ Admins ↳ Grocery Stores ▢ Vendors 📊 Analytics 📁 Categories 📦 Products		Add Category <div style="margin-top: 10px;"> <input type="text" value="Dairy"/> 🔍 Search </div> <table border="1" style="width: 100%; border-collapse: collapse; margin-top: 10px;"> <thead> <tr> <th style="width: 20%;">Image</th> <th style="width: 40%;">Category Name</th> <th style="width: 40%;">Options</th> </tr> </thead> <tbody> <tr> <td style="text-align: center; padding: 10px;"> </td><td style="text-align: center; padding: 10px;">Dairy</td><td style="text-align: center; padding: 10px;"> Update Delete </td></tr> </tbody> </table>			Image	Category Name	Options		Dairy	Update Delete
Image	Category Name	Options								
	Dairy	Update Delete								

Figure 4.1.1.3.4.14 Search Category

In product management, users have extensive capabilities for managing products within the system as shown in Figure 4.1.1.3.4.15. This includes the ability to add new products (Figure 4.1.1.3.4.16), update existing product information (Figure 4.1.1.3.4.17), and delete products if necessary.

WASAIL

Product Management

Image	Product Name	Options
	Coca Cola 1.5 litre	<button>Update</button> <button>Delete</button>
	Coca Cola 500 ml	<button>Update</button> <button>Delete</button>
	Nestle Water 1.5 Litre	<button>Update</button> <button>Delete</button>
	Nestle Water 500 ml	<button>Update</button> <button>Delete</button>
	Aquafina Water 500 ml	<button>Update</button> <button>Delete</button>

Figure 4.1.1.3.4.15 Product Management

Add Product

Product Name:

Product Image: No file chosen

Back Add Product

Figure 4.1.1.3.4.16 Add Product

Update Product

Product Name: Nestle Water 1.5 Litre

Product Image: No file chosen

Back Update Product

Figure 4.1.1.3.4.17 Update 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 4.1.1.3.4.18.

The screenshot shows the WASAIL Product Management interface. On the left, there is a sidebar with navigation links: Admins, Grocery Stores, Vendors, Analytics, Categories, and Products. The main area is titled "Product Management" and contains a search bar with the text "coca" and a "Search" button. Below the search bar is a table with two rows of product data.

Image	Product Name	Options
	Coca Cola 1.5 litre	<button>Update</button> <button>Delete</button>
	Coca Cola 500 ml	<button>Update</button> <button>Delete</button>

Figure 4.1.1.3.4.18 Search Product

4.1.2 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.

4.1.2.1 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 are 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 4.1.2.1.1 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.

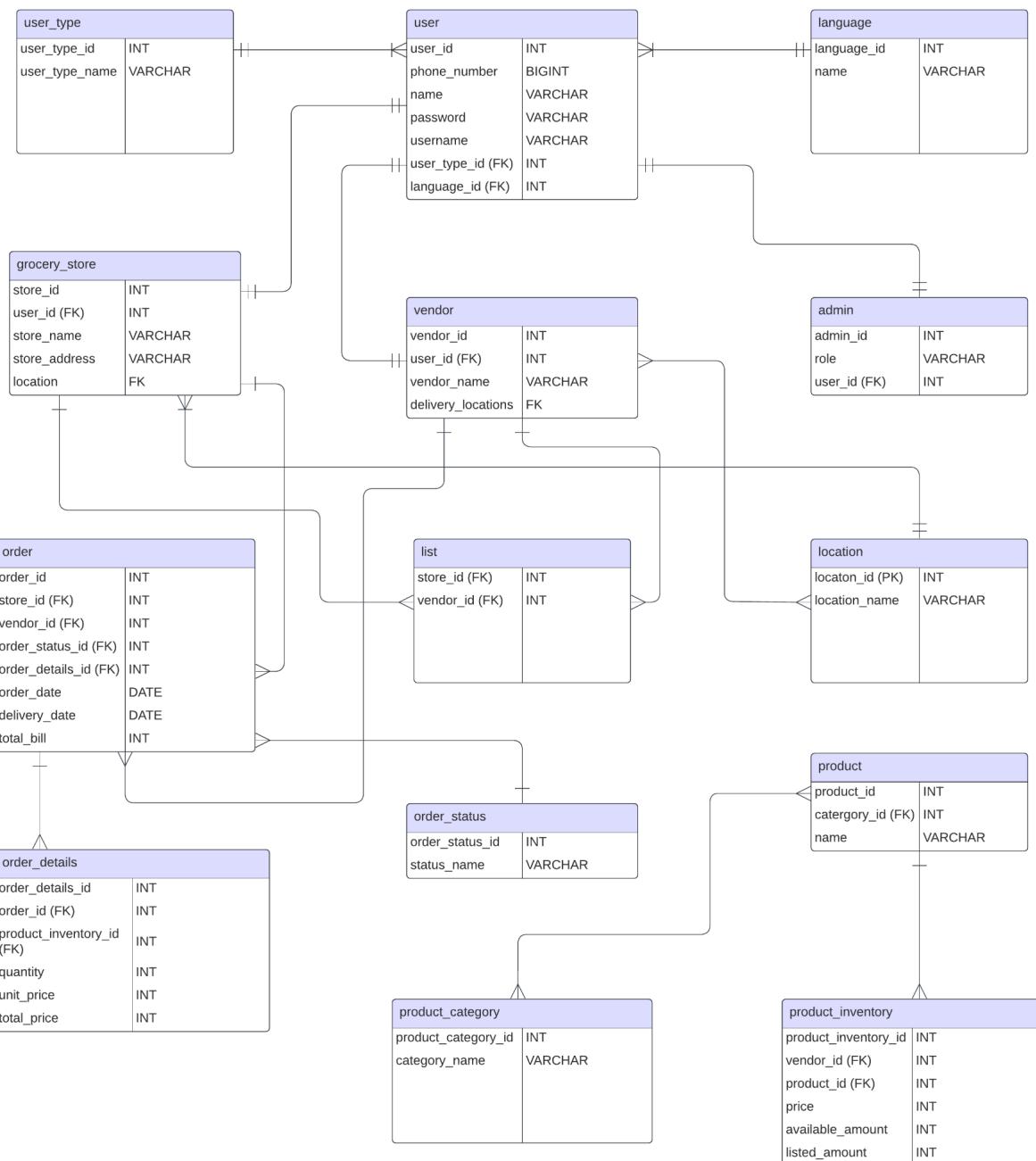


Figure 4.1.2.1.1 First iteration of the database model

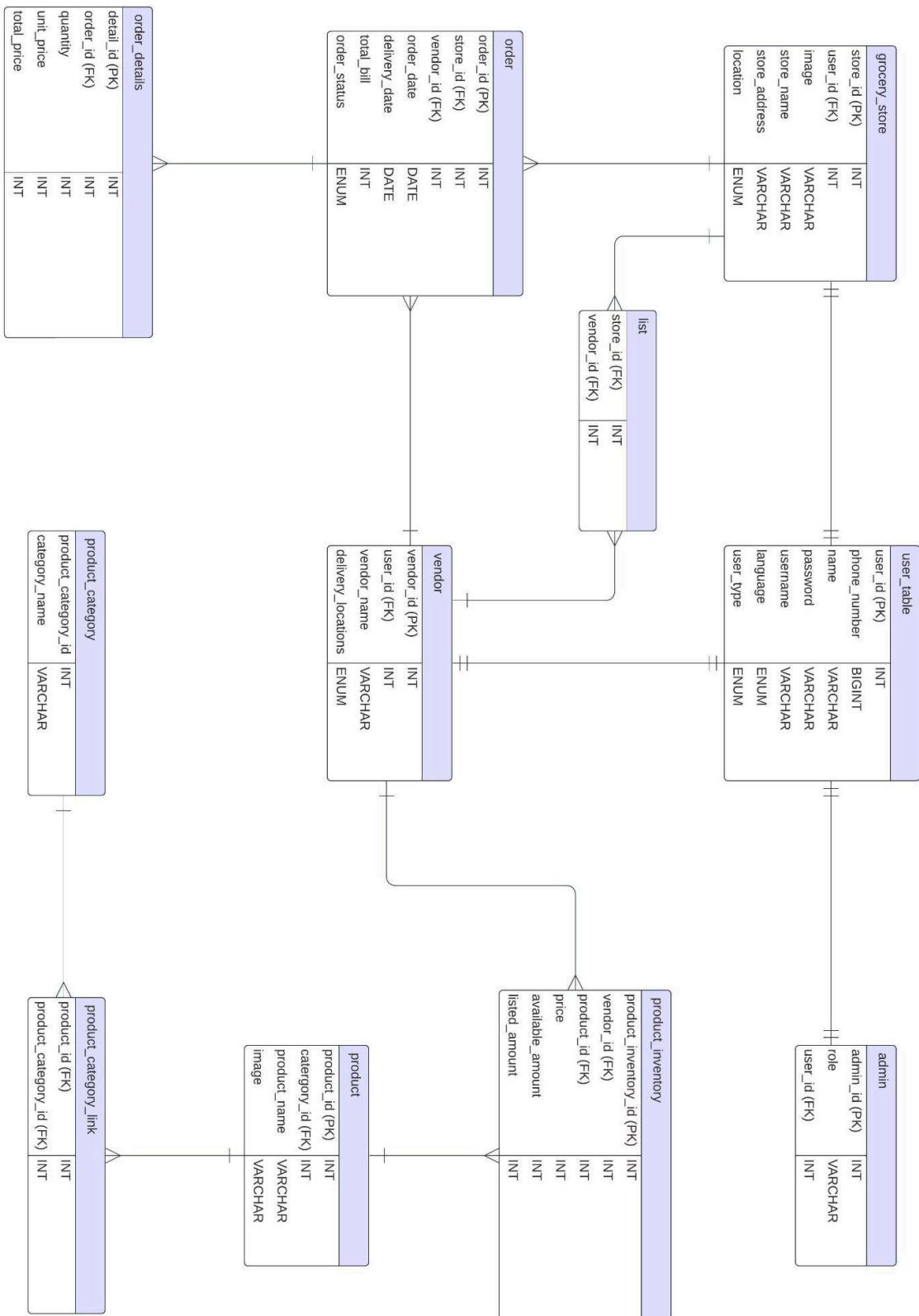


Figure 4.1.2.1.2 Last iteration of database model

4.1.2.2 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 4.1.2.2 shows the front end of the implementation with all the CRUD operations working. We have uploaded the Spring Boot project on GitHub.

The screenshot shows a web application interface. At the top, there is a green header bar with the word "Wasail". Below it, the main title is "Grocery Stores". Underneath the title is a green button labeled "Add Grocery Store". The main content area contains a table with the following data:

Grocery Store Id	Name	Store Name	Store Address	Mobile Number	Shop Location	Actions
5	Fizza Adeel	Jalal Sons	963X+7CM, Block M 7 Lake City	03214356782	Lake City	<button>Edit</button> <button>Delete</button>

Figure 4.1.2.2 Implementation using Spring Boot

4.1.2.3 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 4.1.2.3.1 shows the installation of sequelize. Figure 4.1.2.3.2 shows the installation of MySQL2.

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

Figure 4.1.2.3.1 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 4.1.2.3.2 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.

Figure 4.1.2.3.3 shows the installation of sequelize cli which is another library needed to make the models work with sequelize.

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

Figure 4.1.2.3.3 Installation of sequelize-cli

Figure 4.1.2.3.4 shows all the dependencies that were added in the package.json file in the project.

```
"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 4.1.2.3.4 Dependencies installed for the project

Figure 4.1.2.3.5 shows all the empty folders created after installing sequelize. The ones that were created included mainly the models folder and the config folder.

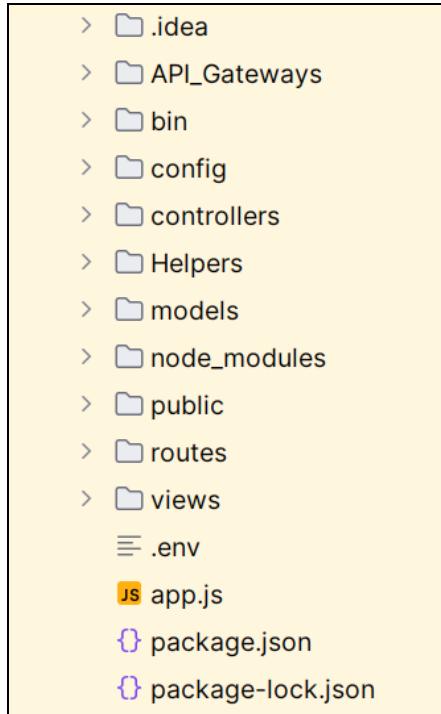


Figure 4.1.2.3.5 Empty files created by sequelize

4.1.2.4 User Model Implementation

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 4.1.2.4, 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.

```
'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,
        },
    },
}
```

Figure 4.1.2.4 User table model created

4.1.2.5 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. As seen in figure 4.1.2.5.1 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.

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

Figure 4.1.2.5.1 .env file

Then using the [npmenv](#) the dot env package was installed as seen in Figure 4.1.2.5.2.

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

Figure 4.1.2.5.2 Installation of .env file

By following the tutorial, we then set up the configuration file as seen in Figure 4.1.2.5.3.

```

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 4.1.2.5.3 Config 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. Figure 4.1.2.5.4 shows the implementation for this.

```

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 4.1.2.5.4 Error handling in index file

As shown in Figure 4.1.2.5.5, the sync function was used to help us connect with the database and sync all the models that were made with the database.

```
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 4.1.2.5.5 Sync function

4.1.2.6 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 4.1.2.6 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 4.1.2.6 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](#)

4.1.2.7 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 4.1.2.7.1 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

1usage  ± 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).json({ 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 4.1.2.7.1 Implementation of User controller

Figure 4.1.2.7.2 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.

```

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 4.1.2.7.2 Implementation of Product controller

4.1.2.8 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 4.1.2.8.1.

```

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 4.1.2.8.1 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 4.1.2.1.2 So we had to implement the three types in three different ways. The three ways have been shown in Figure 4.1.2.8.2, 4.1.2.8.3 and 4.1.2.8.4 respectively. 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 4.1.2.1.2.

```

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

```

Figure 4.1.2.8.2 One to one relation

The relation between product and product inventory is a one to many relation as shown in Figure 4.1.2.1.2.

```

db.producthasMany(db.product_inventory);
db.product_inventory.belongsTo(db.product);

```

Figure 4.1.2.8.3 One 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.

```

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

```

Figure 4.1.2.8.4 Many to many relation

4.1.2.9 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 4.1.2.9.1 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 4.1.2.9.1 Implementation of user routes

Figure 4.1.2.9.2 shows all the routes or the end points defined for the products in the product routes file.

```

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 4.1.2.9.2 Implementation of product routes

4.1.2.10 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 4.1.2.10 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 4.1.2.10 Implementation of main routes

4.1.2.11 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 4.1.2.11.1 shows adding a user through postman.

The screenshot shows a Postman interface for a POST request to `localhost:3000/api/user_table/adduser`. The 'Body' tab is selected, showing a JSON payload:

```

1
2   ...
3     "phone_number":3228483782,
4     "name": "Amna Khan",
5     "password": "lahore@321",
6     "username": "Amna",
7     "language": "English",
8     "user_type": "Vendor"
9
10

```

Below the body, the response is shown in JSON format:

```

1 {
2   "user_id": 5,
3   "phone_number": 3228483782,
4   "name": "Amna Khan",
5   "password": "lahore@321",
6   "username": "Amna",
7   "language": "English",
8   "user_type": "Vendor",
9   "updatedAt": "2024-01-16T04:12:24.190Z",
10  "createdAt": "2024-01-16T04:12:24.190Z"
11

```

Figure 4.1.2.11.1 Adding a user

Figure 4.1.2.11.2 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 4.1.2.11.2 the word water has been searched and 2 products containing the word water have been returned from the database.

```

GET      localhost:3000/api/product/searchproduct/water
Params   Authorization   Headers (6)   Body   Pre-request Script   Tests   Settings
none   form-data   x-www-form-urlencoded   raw   binary   GraphQL
This request does not have a body
Body   Cookies   Headers (7)   Test Results
Pretty   Raw   Preview   Visualize   JSON
1 [
2   {
3     "product_id": 3,
4     "product_name": "dasani-water",
5     "image": "Assets/Images/Products/dasani-water.png",
6     "createdAt": "2023-12-17T12:34:56.000Z",
7     "updatedAt": "2023-12-19T12:34:56.000Z"
8   },
9   {
10    "product_id": 11,
11    "product_name": "aquafina-water",
12    "image": "Assets/Images/Products/aquafina-water.png",
13    "createdAt": "2023-12-17T12:34:56.000Z",
14    "updatedAt": "2023-12-19T12:34:56.000Z"
15  }
]

```

Figure 4.1.2.11.2 Searching a product

Figure 4.1.2.11.3 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.

```

GET      localhost:3000/api/order/search/1
Params   Authorization   Headers (6)   Body   Pre-request Script   Tests   Settings
Body   Cookies   Headers (7)   Test Results
Pretty   Raw   Preview   Visualize   JSON
1 [
2   {
3     "order_id": 1,
4     "order_date": "2023-08-06T00:00:00.000Z",
5     "delivery_date": "2023-08-19T00:00:00.000Z",
6     "total_bill": 45000,
7     "order_status": "On Its Way",
8     "createdAt": "2023-12-17T12:34:56.000Z",
9     "updatedAt": "2023-12-19T12:34:56.000Z",
10    "groceryStoreStoreId": 1,
11    "vendorVendorId": 1
12  }
]

```

```

14      "order_id": 2,
15      "order_date": "2023-10-12T00:00:00.000Z",
16      "delivery_date": "2023-10-19T00:00:00.000Z",
17      "total_bill": 23460,
18      "order_status": "In Process",
19      "createdAt": "2023-12-17T12:34:56.000Z",
20      "updatedAt": "2023-12-19T12:34:56.000Z",
21      "groceryStoreStoreId": 1,
22      "vendorVendorId": 1
23    },
24  {
25      "order_id": 3,
26      "order_date": "2023-10-13T00:00:00.000Z",
27      "delivery_date": "2023-10-25T00:00:00.000Z",
28      "total_bill": 2100,
29      "order_status": "In Process",
30      "createdAt": "2023-12-17T12:34:56.000Z",
31      "updatedAt": "2023-12-19T12:34:56.000Z",
32      "groceryStoreStoreId": 1,
33
34
35
36      "order_id": 5,
37      "order_date": "2023-10-07T00:00:00.000Z",
38      "delivery_date": "2023-10-08T00:00:00.000Z",
39      "total_bill": 5050,
40      "order_status": "On Its Way",
41      "createdAt": "2023-12-17T12:34:56.000Z",
42      "updatedAt": "2023-12-19T12:34:56.000Z",
43      "groceryStoreStoreId": 2,
44      "vendorVendorId": 1
45    },
46  {
47      "order_id": 6,
48      "order_date": "2023-08-06T00:00:00.000Z",
49      "delivery_date": "2023-08-19T00:00:00.000Z",
50      "total_bill": 1935,
51      "order_status": "In Process",
52      "createdAt": "2023-12-17T12:34:56.000Z",
53      "updatedAt": "2023-12-19T12:34:56.000Z",
54      "groceryStoreStoreId": 2,
55
56
57

```

Figure 4.1.2.11.3 Current Orders

Similarly, Figure 4.1.2.11.4 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.

GET		localhost:3000/api/order/orderhistory/1					
Params	Authorization	Headers (6)	Body	Pre-request Script	Tests	Settings	
Body	Cookies	Headers (7)	Test Results				
Pretty	Raw	Preview	Visualize	JSON			
1 [
2 {							
3 "order_id": 4,							
4 "order_date": "2023-11-15T00:00:00.000Z",							
5 "delivery_date": "2023-11-21T00:00:00.000Z",							
6 "total_bill": 10000,							
7 "order_status": "Delivered",							
8 "createdAt": "2023-12-17T12:34:56.000Z",							
9 "updatedAt": "2023-12-19T12:34:56.000Z",							
10 "groceryStoreStoreId": 1,							
11 "vendorVendorId": 1							
12 },							

```

14 |     "order_id": 7,
15 |     "order_date": "2024-01-01T00:00:00.000Z",
16 |     "delivery_date": "2023-01-05T00:00:00.000Z",
17 |     "total_bill": 1935,
18 |     "order_status": "Delivered",
19 |     "createdAt": "2023-12-17T12:34:56.000Z",
20 |     "updatedAt": "2023-12-19T12:34:56.000Z",
21 |     "groceryStoreStoreId": 2,
22 |     "vendorVendorId": 1
23 |
24 |
25 |     "order_id": 8,
26 |     "order_date": "2024-01-03T00:00:00.000Z",
27 |     "delivery_date": "2023-01-08T00:00:00.000Z",
28 |     "total_bill": 1935,
29 |     "order_status": "Delivered",
30 |     "createdAt": "2023-12-17T12:34:56.000Z",
31 |     "updatedAt": "2023-12-19T12:34:56.000Z",
32 |     "groceryStoreStoreId": 2,
33 |
34 |
35 |   {
36 |     "order_id": 11,
37 |     "order_date": "2024-01-03T00:00:00.000Z",
38 |     "delivery_date": "2024-01-07T00:00:00.000Z",
39 |     "total_bill": 1400,
40 |     "order_status": "Delivered",
41 |     "createdAt": "2023-12-17T12:34:56.000Z",
42 |     "updatedAt": "2023-12-19T12:34:56.000Z",
43 |     "groceryStoreStoreId": 3,
44 |     "vendorVendorId": 1
45 |
46 ]

```

Figure 4.1.2.11.4 Order History

4.1.2.12 Implementation of Grocery Store Application

We started the implementation for FYP II with the implementation of our second mobile application i.e the Grocery Store Application. Firstly, we started the work with handling the user management section of the database. The process started with implementing the registration for the Grocery Store as shown in Figure 4.1.2.12.1. Since there is a one to one relation between the grocery store and the vendor, as shown in Figure 4.1.2.12.2, we are first ensuring that a new user is being made. Once the user has been created successfully, then the store is created, saving the rest of the information which mainly includes the store name, the store image and the store address. Furthermore, if it fails to create the user or the store then it informs us through an error message so errors are being handled as well.

```

const addStoreRegistration = async (req, res) => {
  try {
    const userInfo = {
      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(userInfo);

    if (user) {
      const storeInfo = {
        // user_id: user.id,
        store_name: req.body.store_name,
        image: req.body.image,
        store_address: req.body.store_address,
        user_table_user_id: req.body.user_table_user_id
      };
    }
  }

  const store = await Store.create(storeInfo);

  res.status(200).json({ user, store });
} else {
  res.status(400).json({ error: 'Failed to create user.' });
}
} catch (error) {
  console.error('Error creating user and vendor:', error);
  res.status(500).json({ error: 'Internal Server Error' });
}
};


```

Figure 4.1.2.12.1 Store Registration

```

//user_table-grocery_store
db.user_tablehasOne(db.grocery_store);
db.grocery_store.belongsTo(db.user_table);

```

Figure 4.1.2.12.2 Relation between Grocery Store and User Table

Lastly the end point was made in the registration routes for the front end as shown in Figure 4.1.2.12.3.

```
router.post('/addstoreregistration', registrationController.addStoreRegistration)
```

Figure 4.1.2.12.3 End Point Implemented for Store Registration

Finally, after the implementation of the store registration, it was important to ensure that it was properly working. So it was tested through postman as shown in Figure 4.1.2.12.4.

The screenshot shows the Postman application interface. The top bar indicates a POST method and the URL `localhost:3000/api/registration/storeregistration`. Below the URL, there are tabs for Params, Authorization, Headers (8), Body (selected), Pre-request Script, Tests, and Settings. Under the Body tab, the content type is set to JSON. The body content is a JSON object with the following structure:

```
1 {
2   "phone_number": "3214377009",
3   "name": "Pizza Adeel",
4   "password": "password@123",
5   "username": "fizza",
6   "language": "English",
7   "user_type": "Grocery Store",
8   "store_name": "Alfatah",
9   "image": "Grocery Store",
10  "store_address": "Gulberg",
11  "createdAt": "2023-12-17T12:34:56.000Z",
12  "updatedAt": "2023-12-19T12:34:56.000Z"
13 }
```

Figure 4.1.2.12.4 Postman Testing

Moreover, another important functional requirement which was the search for products in the grocery store application is being explained as well. The purpose of the functional requirement was to search for the product name and then in return it should display all the vendors that are selling that product so that the grocery store can select which vendor to order that product from. So the logic that was used was that first the product name would be searched in the product table as shown in Figure 4.1.2.12.5.

```

const searchProductInStore = async (req, res) => {
  try {
    const product_name = req.params.product_name;

    if (!product_name) {
      return res.status(400).json({ error: 'Product name is required.' });
    }

    // Step 1: Search Products by Name
    const products = await Product.findAll({
      where: {
        product_name: {
          [Op.iLike]: `%%${product_name}%%`,
        },
      },
    });

    if (!products || products.length === 0) {
      return res.status(404).json({ error: 'No products found for the given name.' });
    }
  }
}

```

Figure 4.1.2.12.5 Search Product

Then, once that product is found its corresponding product id is extracted. Then, those product ids are used in the product inventories table to find its corresponding product inventories as shown in Figure 4.1.2.12.6.

```

const productIds = products.map((product) => product.product_id);

// Step 3: Search Product Inventory by Product IDs
const productInventories = await db.product_inventory.findAll({
  where: {
    product_product_id: {
      [Op.in]: productIds,
    },
  },
});

if (!productInventories || productInventories.length === 0) {
  return res.status(404).json({ error: 'No product inventory found for the given products.' });
}

```

Figure 4.1.2.12.6 Search Product

Then, using those product inventory ids their corresponding vendor id is found. Then, that vendor id is used to get all the information of that particular vendor from the vendors table as shown in Figure 4.1.2.12.7.

```

const vendorIds = productInventories.map((inventory) => inventory.vendor_vendor_id);

// Step 5: Search Vendors by Vendor IDs
const vendors = await db.vendor.findAll({
    where: {
        vendor_id: {
            [Op.in]: vendorIds,
        },
    },
});

if (!vendors || vendors.length === 0) {
    return res.status(404).json({ error: 'No vendors found for the given products.' });
}

```

Figure 4.1.2.12.7 Search Product

Finally, all the products, the product inventories, and the vendor were returned and error handling was done as shown in Figure 4.1.2.12.8.

```

res.status(200).json({
    products,
    productInventories,
    vendors,
});
} catch (error) {
    console.error('Error searching product details:', error);
    res.status(500).json({ error: 'Internal Server Error' });
}
};

```

Figure 4.1.2.12.8 Search Product

Its endpoint was also defined in the routes folder as shown in Figure 4.1.2.12.9.

```
router.get('/searchproductinstore/:product_name', productController.searchProductInStore)
```

Figure 4.1.2.12.9 End Point Implemented for Search Product

The functional requirement for the vendor list being displayed in the grocery store application was implemented as shown in Figure 4.1.2.12.10 so that the grocery store owner can view all the vendors they have in their vendor list.

```

const viewVendorList = async (req, res) => {
  try {
    const store_id = req.params.grocery_store_store_id;

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

    const associatedVendors = await db.list.findAll({
      where: { grocery_store_store_id: store_id },
    });

    const vendorIds = associatedVendors.map((vendor) => vendor.vendor_vendor_id);

    const vendors = await db.vendor.findAll({
      where: { vendor_id: vendorIds },
    });

    res.status(200).json(vendors);
  } catch (error) {
    console.error('Error viewing vendor list:', error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
};

```

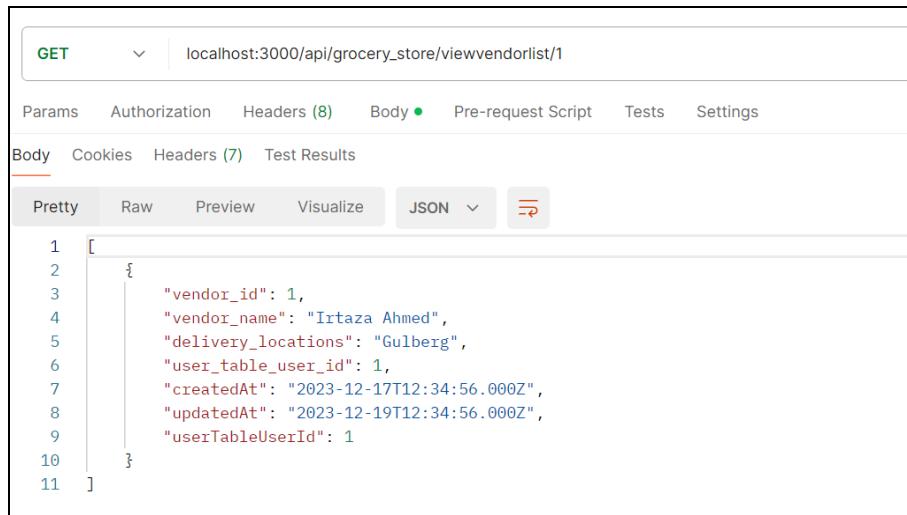
Figure 4.1.2.12.10 View Vendor List

The logic that was used behind the implementation was that the grocery store id would be first used, using which all the corresponding vendor ids would be retrieved from the pivot table called the ‘lists’ table. Then those corresponding vendor ids would be used and data related to the vendor would be fetched from the vendor table. Finally the fetched data would be sent back. Its endpoint was also defined in routes as shown in Figure 4.1.2.12.11.

```
router.get('/viewvendorlist/:grocery_store_store_id', grocerystoreController.viewVendorList)
```

Figure 4.1.2.12.11 End Point Implemented for Vendor List

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.12.



The screenshot shows a Postman interface with a GET request to `localhost:3000/api/grocery_store/viewvendorlist/1`. The response body is displayed in JSON format:

```
1 [  
2 {  
3   "vendor_id": 1,  
4   "vendor_name": "Irtaza Ahmed",  
5   "delivery_locations": "Gulberg",  
6   "user_table_user_id": 1,  
7   "createdAt": "2023-12-17T12:34:56.000Z",  
8   "updatedAt": "2023-12-19T12:34:56.000Z",  
9   "userTableUserId": 1  
10 }  
11 ]
```

Figure 4.1.2.12.12 Postman Testing

Furthermore, another important functional requirement was covered which was the search for categories in the grocery store application. The purpose of the functional requirement was to search for the category name and then in return it would show all the products in that category along with the vendors that are selling those specific products. So in order to implement this first we had to implement the model class for product category link as shown in Figure 4.1.2.12.13, as it is the pivot table for product and product categories.

```

module.exports = (sequelize, Datatypes)=>{
  return sequelize.define('product_category_link',{
    product_product_id: {
      type: Datatypes.INTEGER,
      primaryKey: true,
      references: {
        model: 'products',
        key: 'product_id'
      },
      allowNull: false
    },
    product_category_product_category_id: {
      type: Datatypes.INTEGER,
      primaryKey: true,
      references: {
        model: 'product_catgories',
        key: 'product_category_id'
      },
      allowNull: false
    },
    },{
      underscored:true
    })
}

```

Figure 4.1.2.12.13 Product Category Link Class

It mainly had two foreign keys, one for the products table and the other for the product categories table.

Then the function was implemented in the product category controller. So the logic that was used was that first the category name would be searched in the product category table as shown in Figure 4.1.2.12.14.

```

const searchCategoryInStore = async (req, res) => {
  try {
    const category_name = req.params.category_name;

    if (!category_name) {
      return res.status(400).json({ error: 'Category name is required.' });
    }

    const categories = await Category.findAll({
      where: {
        category_name: {
          [Op.like]: `%%${category_name}%%`,
        },
      },
    });

    if (!categories || categories.length === 0) {
      return res.status(404).json({ error: 'No categories found for the given name.' });
    }
  }
}

```

Figure 4.1.2.12.14 Search Category

Then, once that category is found its corresponding category id is extracted. Then, those category ids are used in the pivot table i.e the product category link table to find its corresponding product ids as shown in Figure 4.1.2.12.15.

```

const categoryIds = categories.map((category) => category.product_category_id);

const productCategoryLinks = await db.product_category_link.findAll({
  where: {
    product_category_product_category_id: {
      [Op.in]: categoryIds,
    },
  },
});

if (!productCategoryLinks || productCategoryLinks.length === 0) {
  return res.status(404).json({ error: 'No product-category links found for the given categories.' });
}

const productIds = productCategoryLinks.map((link) => link.product_product_id);

```

Figure 4.1.2.12.15 Search Category

Then, those product ids are used in the product table to find the products so that information about those products can be sent later. Then it is searching product inventories by product ids as shown in Figure 4.1.2.12.16.

```

const products = await db.product.findAll({
    where: {
        product_id: {
            [Op.in]: productIds,
        },
    },
});

if (!products || products.length === 0) {
    return res.status(404).json({ error: 'No products found for the given categories.' });
}

const productInventories = await db.product_inventory.findAll({
    where: {
        product_product_id: {
            [Op.in]: productIds,
        },
    },
});

if (!productInventories || productInventories.length === 0) {
    return res.status(404).json({ error: 'No product inventory found for the given products.' });
}

```

Figure 4.1.2.12.16 Search Category

Then, using those product inventory ids their corresponding vendor id is found. Then, that vendor id is used to get all the information of that particular vendor from the vendors table as shown in Figure 4.1.2.12.17.

```

const vendorIds = productInventories.map((inventory) => inventory.vendor_vendor_id);

const vendors = await db.vendor.findAll({
    where: {
        vendor_id: {
            [Op.in]: vendorIds,
        },
    },
});

if (!vendors || vendors.length === 0) {
    return res.status(404).json({ error: 'No vendors found for the given products.' });
}

```

Figure 4.1.2.12.17 Search Category

Finally, all the categories information, products, the product inventories, and the vendor were returned and error handling was done as shown in Figure 4.1.2.12.18.

```

        res.status(200).json({
            categories,
            products,
            productInventories,
            vendors,
        });
    } catch (error) {
        console.error('Error searching category details:', error);
        res.status(500).json({ error: 'Internal Server Error' });
    }
}

```

Figure 4.1.2.12.18 Search Category

Finally it was tested through postman as shown in Figure 4.1.2.12.19.

The screenshot shows a Postman interface with a GET request to `localhost:3000/api/product_category/searchcategoryinstore/drinks`. The response body is displayed in Pretty JSON format, showing the following data:

```

1  {
2      "categories": [
3          {
4              "product_category_id": 1,
5              "category_name": "drinks",
6              "image": "Assets/Images/Products/lays-salted.png",
7              "createdAt": "2023-12-17T12:34:56.000Z",
8              "updatedAt": "2023-12-19T12:34:56.000Z"
9          }
10         ],
11     "products": [
12         {
13             "product_id": 1,
14             "product_name": "lays",
15             "image": "Assets/Images/Products/lays-salted.png",
16             "createdAt": "2023-12-17T12:34:56.000Z",
17             "updatedAt": "2023-12-19T12:34:56.000Z"
18         },
19         {
20             "product_id": 2,
21             "product_name": "cocacola",
22             "image": "Assets/Images/Products/cocacola.jpg",
23             "createdAt": "2023-12-17T12:34:56.000Z",
24             "updatedAt": "2023-12-19T12:34:56.000Z"
25         }
26     ]
27 }

```

```

47   {
48     "product_inventory_id": 2,
49     "price": 50,
50     "available_amount": 50,
51     "listed_amount": 35,
52     "vendor_vendor_id": 1,
53     "product_product_id": 3,
54     "createdAt": "2023-12-19T12:34:56.000Z",
55     "updatedAt": "2023-12-19T12:34:56.000Z",
56     "productProductId": 3,
57     "vendorVendorId": 1
58   },
59 ]
60
61
62
63
64
65
66
67
68
69
70
71

```

Figure 4.1.2.12.19 Postman Testing

Another important functional requirement that was covered was the view vendor profile. The vendor profile would have the information about the vendor, and his inventories or the products that the vendor is selling to the grocery stores. So the logic that was used was that first the vendor id would be used to retrieve all the details of that vendor from the vendor table. This is being shown in Figure 4.1.2.12.20.

```

const vendorProfile = async (req, res) => {
  try {
    const vendor_id = req.params.vendor_id;

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

    const vendor = await Vendor.findOne({
      where: {
        vendor_id: vendor_id,
      },
    });

    if (!vendor) {
      return res.status(404).json({ error: 'Vendor not found for the given vendor ID.' });
    }
  }
}

```

Figure 4.1.2.12.20 View Vendor Profile

Next, using that vendor id, its corresponding product inventories are looked up from the product inventory table so that all the information related to its product inventories could be sent back as shown in Figure 4.1.2.12.21.

```

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

```

Figure 4.1.2.12.21 View Vendor Profile

Then, as shown in Figure 4.1.2.12.22, it looks up all the corresponding product ids in the product inventory table and uses those ids to fetch information about the products from the products table so that it can send back product information as well. It then sends all the information (which includes vendor, product inventory and product information) back and incase of an error it sends back a message.

```

const productIds = productInventories.map((inventory) => inventory.product_product_id);

const products = await db.product.findAll({
    where: {
        product_id: productIds,
    },
});

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


```

Figure 4.1.2.12.22 View Vendor Profile

Its end point was also defined in routes as shown in figure 4.1.2.12.23.

```
router.get('/vendorprofile/:vendor_id', vendorController.vendorProfile)
```

Figure 4.1.2.12.23 End Point Implemented for Vendor Profile

Finally it was tested through postman as shown in Figure 4.1.2.12.24.

```

GET      | localhost:3000/api/vendor/vendorprofile/1
Params   Authorization Headers (8) Body • Pre-request Script Tests Settings
Body   Cookies Headers (7) Test Results
Pretty Raw Preview Visualize JSON ▾
1 {
2   "vendor": {
3     "vendor_id": 1,
4     "vendor_name": "Irtaza Ahmed",
5     "delivery_locations": "Gulberg",
6     "image": "Assets/Images/Stores/esajees.png",
7     "user_table_user_id": 1,
8     "createdAt": "2023-12-17T12:34:56.000Z",
9     "updatedAt": "2023-12-19T12:34:56.000Z",
10    "userTableUserId": 1
11  },
122  "products": [
123    {
124      "product_id": 1,
125      "product_name": "lays",
126      "image": "Assets/Images/Products/lays-salted.png",
127      "createdAt": "2023-12-17T12:34:56.000Z",
128      "updatedAt": "2023-12-19T12:34:56.000Z"
129    },
130    {
131      "product_id": 3,
132      "product_name": "dasani-water",
133      "image": "Assets/Images/Products/dasani-water.png",
134      "createdAt": "2023-12-17T12:34:56.000Z",
135      "updatedAt": "2023-12-19T12:34:56.000Z"
136    }
25   {
26     "product_inventory_id": 2,
27     "price": 50,
28     "available_amount": 50,
29     "listed_amount": 35,
30     "vendor_vendor_id": 1,
31     "product_product_id": 3,
32     "createdAt": "2023-12-19T12:34:56.000Z",
33     "updatedAt": "2023-12-19T12:34:56.000Z",
34     "productProductId": 3,
35     "vendorVendorId": 1
36   },

```

Figure 4.1.2.12.24 Postman Testing

We also covered some order management related functional requirements for the grocery store application. We started off with first making some changes to the database. We realised that a relation between the product inventory table and the order details table, which was previously missing, is imperative for order management, so a one-to-many relation between the two tables was implemented as shown in Figure 4.1.2.12.25.

```

db.product_inventoryhasMany(db.order_detail);
db.order_detail.belongsTo(db.product_inventory);

```

Figure 4.1.2.12.25 Relation between the tables

Furthermore, another important functional requirement that was covered included order history which is basically all the previous orders that have been placed by the grocery store to different vendors. Figure 4.1.2.12.26 shows its implementation.

```

const storeOrderHistory = async (req, res) => {
  try {
    const store_id = req.params.grocery_store_store_id;

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

    const orders = await Order.findAll({
      where: {
        grocery_store_store_id: store_id,
        order_status: {
          [Op.in]: ['Delivered'],
        },
      },
    });

    res.status(200).send(orders);
  } catch (error) {
    console.error('Error searching products by store:', error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
};

```

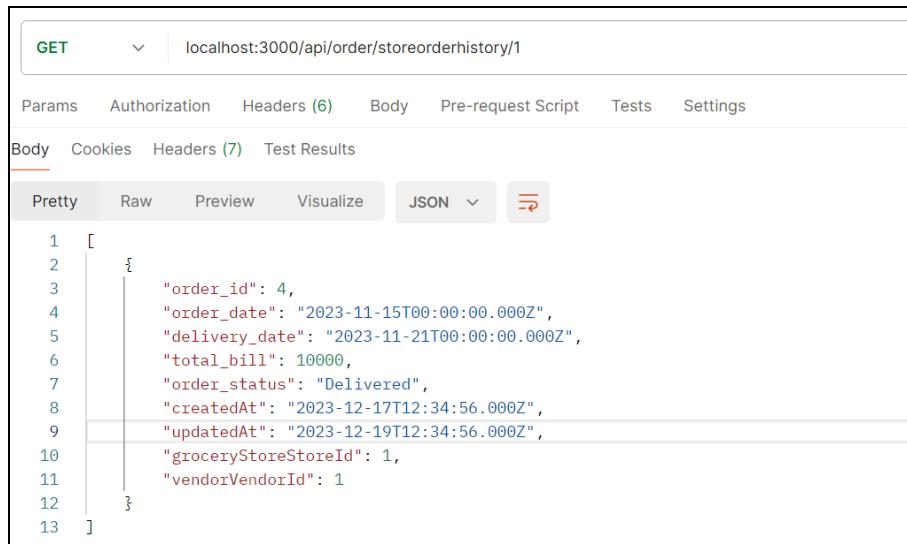
Figure 4.1.2.12.26 Order History

The logic that was used for this particular functional requirement was that by using the store id, all its corresponding orders were looked up in the orders table. Once those orders were found in the table, its order status would be checked and if that status was ‘delivered’ then all such orders would be returned. Its endpoint was also defined in routes as shown in Figure 4.1.2.12.27.

```
router.get('/storeorderhistory/:grocery_store_store_id', orderController.storeOrderHistory)
```

Figure 4.1.2.12.27 End Point Implemented for Order History

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.28.



The screenshot shows a Postman interface with a GET request to `localhost:3000/api/order/storeorderhistory/1`. The response is displayed in a JSONpretty format, showing a single order object:

```
1 [  
2 {  
3   "order_id": 4,  
4   "order_date": "2023-11-15T00:00:00.000Z",  
5   "delivery_date": "2023-11-21T00:00:00.000Z",  
6   "total_bill": 10000,  
7   "order_status": "Delivered",  
8   "createdAt": "2023-12-17T12:34:56.000Z",  
9   "updatedAt": "2023-12-19T12:34:56.000Z",  
10  "groceryStoreStoreId": 1,  
11  "vendorVendorId": 1  
12 }  
13 ]
```

Figure 4.1.2.12.28 Postman Testing

Similar to order history, current orders, which was also a functional requirement for the grocery store, was implemented. The purpose of this requirement was to display all the orders to the grocery stores which are yet to be delivered so that they can keep a track of them. Figure 4.1.2.12.29 shows the implementation of current orders.

```

const storeCurrentOrders = async (req, res) => {
  try {
    const store_id = req.params.grocery_store_store_id;

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

    const orders = await Order.findAll({
      where: {
        grocery_store_store_id: store_id,
        order_status: {
          [Op.in]: ['In Process', 'On Its Way'],
        },
      },
    });

    res.status(200).send(orders);
  } catch (error) {
    console.error('Error searching products by store:', error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
};

```

Figure 4.1.2.12.29 Current Orders

The logic implemented here was similar to order history, the only difference being that instead of order status being ‘delivered’ it was now either ‘in process’ or ‘on its way’. Its end point has also been defined in route as shown in Figure 4.1.2.12.30.

```
router.get('/storecurrentorder/:grocery_store_store_id', orderController.storeCurrentOrders)
```

Figure 4.1.2.12.30 End Point Implemented for Current Orders

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.31.

```

GET      localhost:3000/api/order/storecurrentorder/1

Params   Authorization   Headers (6)   Body   Pre-request Script   Tests   Settings
Body   Cookies   Headers (7)   Test Results
Pretty   Raw   Preview   Visualize   JSON   🔍

{
  "order_id": 1,
  "order_date": "2023-08-06T00:00:00.000Z",
  "delivery_date": "2023-08-19T00:00:00.000Z",
  "total_bill": 45000,
  "order_status": "On Its Way",
  "createdAt": "2023-12-17T12:34:56.000Z",
  "updatedAt": "2023-12-19T12:34:56.000Z",
  "groceryStoreStoreId": 1,
  "vendorVendorId": 1
},
{
  "order_id": 2,
  "order_date": "2023-10-12T00:00:00.000Z",
  "delivery_date": "2023-10-19T00:00:00.000Z",
  "total_bill": 23460,
  "order_status": "In Process",
  "createdAt": "2023-12-17T12:34:56.000Z",
  "updatedAt": "2023-12-19T12:34:56.000Z"
}

```

Figure 4.1.2.12.31 Postman Testing

Moreover, another functional requirement that was covered included order placement. The purpose of this requirement was that grocery stores can place an order to vendors through the mobile application instead of calling the vendors and then placing the order. In this way the grocery store will be able to keep track of all the items that it has ordered and there would be less chances of miscommunication as well. The implementation of this requirement has been shown in Figure 4.1.2.12.32.

```

const orderPlacement = async (req, res) => {
  try {
    const data = {
      order_date: req.body.order_date,
      delivery_date: req.body.delivery_date,
      total_bill: req.body.total_bill,
      order_status: req.body.order_status,
      grocery_store_store_id: req.body.grocery_store_store_id,
      vendor_vendor_id: req.body.vendor_vendor_id
    };

    const order = await Order.create(data);

    if (order) {
      const detailInfo = {
        quantity: req.body.quantity,
        unit_price: req.body.unit_price,
        total_price: req.body.total_price,
        order_order_id: req.body.order_order_id,
        product_inventory_product_inventory_id: req.body.product_inventory_product_inventory_id
      };
    }
  } catch (error) {
    res.status(500).json({ error: 'Internal Server Error' });
  }
};

```

```

        const detail = await Detail.create(detailInfo);

        res.status(200).json({ order, detail });
    } else {
        res.status(400).json({ error: 'Failed to create order.' });
    }
} catch (error) {
    console.error('Error creating order and order details:', error);
    res.status(500).json({ error: 'Internal Server Error' });
}
};


```

Figure 4.1.2.12.32 Order Placement

The logic implemented here is that when the user presses the ‘place order’ button on the front end, an order is created in the ‘Orders’ table and all information related to it for example order date, delivery date, total bill etc is stored in the orders table. Once an order has been created then a row in the order details table is created for every product that has been ordered. For example, if there are a total of three products in the order, then three individual rows would be created for those three products in the order details table, and each would have its own quantity and price etc. Figure 4.1.2.12.33 shows the end point that has been implemented in the routes folder.

```
router.post('/orderplacement', orderController.orderPlacement)
```

Figure 4.1.2.12.33 End Point Implemented for Order Placement

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.34.

```

2   "order_date": "2023-08-06T00:00:00.000Z",
3   "delivery_date": "2023-08-19T00:00:00.000Z",
4   "total_bill": 6000,
5   "order_status": "On Its Way",
6   "createdAt": "2023-12-17T12:34:56.000Z",
7   "updatedAt": "2023-12-19T12:34:56.000Z",
8   "groceryStoreStoreId": 1,
9   "vendorVendorId": 1,
10
11   "quantity": 5,
12   "unit_price": 80,
13   "total_price": 500,
14   "order_order_id": 1,
15   "product_inventory_product_inventory_id": 1,
16   "createdAt": "2023-12-17T12:34:56.000Z",
17   "updatedAt": "2023-12-19T12:34:56.000Z",
18   "orderOrderId": 1,
19   "productInventoryProductInventoryId": 1

```

Figure 4.1.2.12.34 Postman Testing

We worked on handling images on the backend as well. Previously, images were stored in a file inside the application on flutter, which made the applications (both vendor and grocery store) extremely heavy which would mean that they would require a lot of memory space on your mobile. So we decided to shift the images on the server's file system and each image file would be saved inside its designated directory. To start this off, we first needed to install multer which is middleware primarily used for uploading files. Figure 4.1.2.12.35 shows the installation of multer in NodeJS.

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

Figure 4.1.2.12.35 Installation of Multer

This was downloaded by reading the documentation from [npm](#)'s website. Next Figure 4.1.2.12.36 shows the package installed for multer in the packages folder.

```
"multer": "^1.4.5-lts.1",
```

Figure 4.1.2.12.36 Multer Package

Now before writing the code for images, all the images were shifted to a folder in NodeJs called 'uploads' and subfolders were made for all the images in order to organise them properly. Figure 4.1.2.12.37 shows the upload folder.

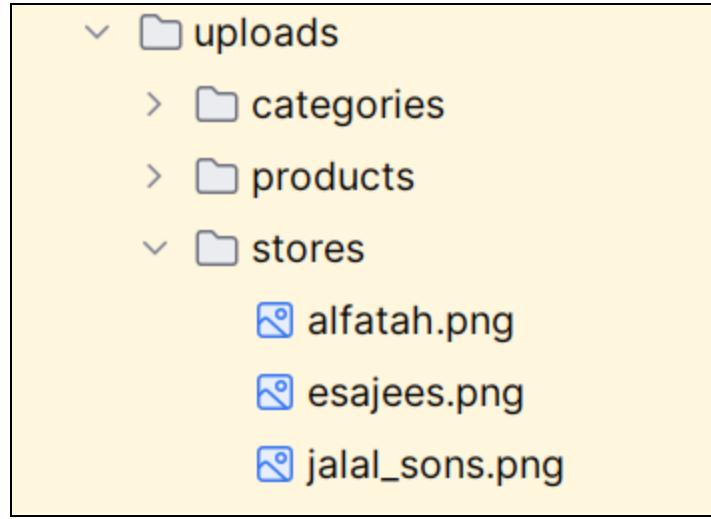


Figure 4.1.2.12.37 Uploads Folder for Images

Next, the function for fetching the images was implemented. Firstly, ‘Get Product Image’ was implemented that was going to deal with fetching images from the products folder, inside uploads. Figure 4.1.2.12.38 shows the implementation for this particular function.

```
const getProductImage = (req, res) => {
  const filename = req.params.filename;
  const imagePath = path.join(__dirname, '../uploads/products', filename);

  fs.readFile(imagePath, (err, data) => {
    if (err) {
      return res.status(404).json({ error: 'Image not found.' });
    }

    res.setHeader('Content-Type', 'image/png');
    res.send(data);
  });
};
```

Figure 4.1.2.12.38 Implementation of Get Product Image

The logic that was used to implement the function was that first we will get the filename from the request parameters. Then we will construct the path to the image file based on the filename and the upload directory. We will then use Node.js fs module to read the image file. If the image file is found, we set the appropriate content type header based on the image type and send the image data back to the client. If the image file is not found, we return an error message. Figure 4.1.2.12.39 shows its endpoint defined in the routes folder.

```
router.get('/products/:filename', imageController.getProductImage);
```

Figure 4.1.2.12.39 End Point Implemented for Product Images

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.40.

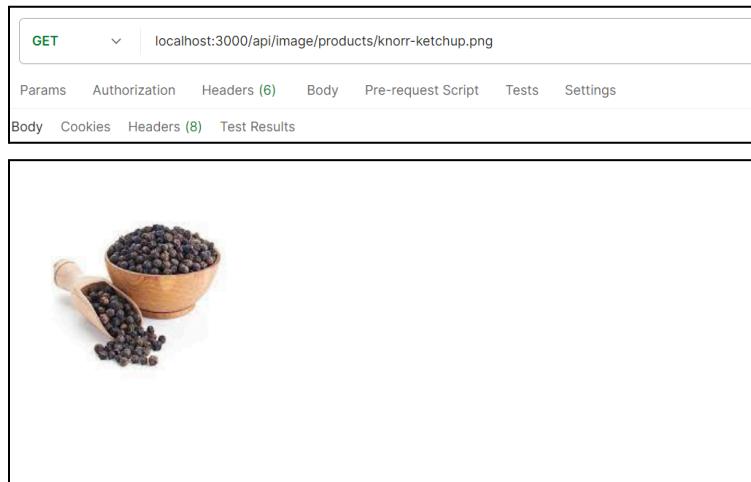


Figure 4.1.2.12.40 Postman Testing

Similar to this function, ‘Get Category Images’ was also implemented. Figure 4.1.2.12.41 shows the implementation of the aforementioned function.

```
const getCategoryImage = (req, res) => {
  const filename = req.params.filename;
  const imagePath = path.join(__dirname, '../uploads/categories', filename);

  fs.readFile(imagePath, (err, data) => {
    if (err) {
      return res.status(404).json({ error: 'Image not found.' });
    }

    res.setHeader('Content-Type', 'image/png');
    res.send(data);
  });
};
```

Figure 4.1.2.12.41 Implementation of Get Category Image

The logic implemented for this function is also similar to the get product images function, the only difference being that now it is fetching images from a different folder i.e the categories folder inside the uploads folder, because it is only supposed to display images related to categories. Figure 4.1.2.12.42 shows the end point defined for this particular function in the routes folder.

```
router.get('/categories/:filename', imageController.getCategoryImage);
```

Figure 4.1.2.12.42 End Point Implemented for Get Category Image

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.43.

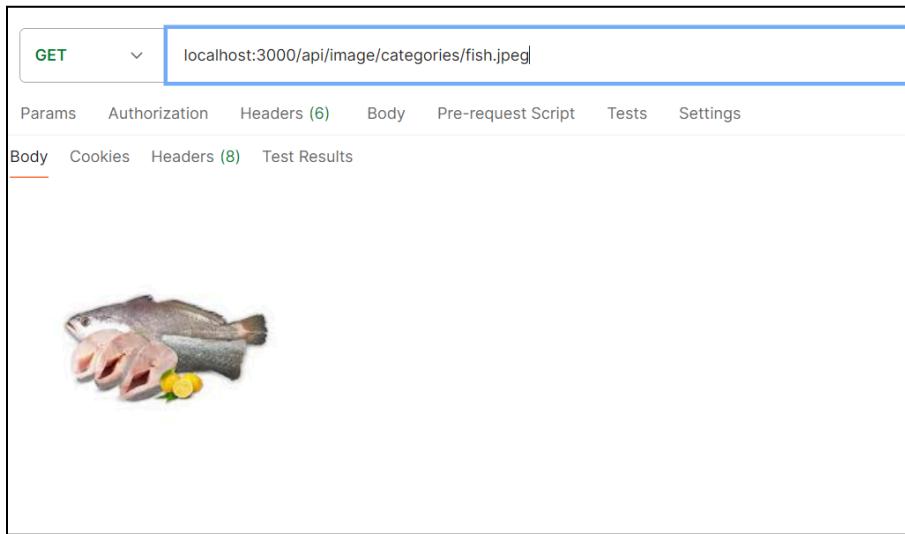


Figure 4.1.2.12.43 Postman Testing

Similarly, another function for store images was implemented. Implementation for it was done using the same logic, the only difference being that instead of the categories folder, stores folder was used to access store images. Figure 4.1.2.12.44 shows the implementation of ‘Get Store Image’.

```
const getStoreImage = (req, res) => {
  const filename = req.params.filename;
  const imagePath = path.join(__dirname, '../uploads/stores', filename);

  fs.readFile(imagePath, (err, data) => {
    if (err) {
      return res.status(404).json({ error: 'Image not found.' });
    }

    res.setHeader('Content-Type', 'image/png');
    res.send(data);
  });
};
```

Figure 4.1.2.12.44 Implementation of Get Store Image

Its endpoint was also defined in the routes folder as shown in Figure 4.1.2.12.45.

```
router.get('/stores/:filename', imageController.getStoreImage);
```

Figure 4.1.2.12.45 End Point Implemented for Get Store Image

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.46.

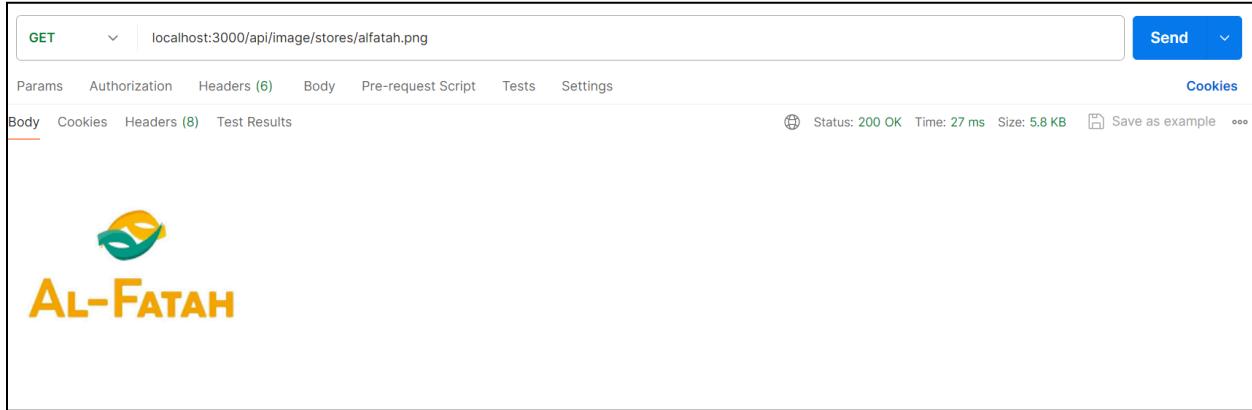
A screenshot of the Postman application interface. The top bar shows 'GET' selected, the URL 'localhost:3000/api/image/stores/alfatah.png', and a 'Send' button. Below the URL bar are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Headers (8)' tab is currently active. On the right side, status information is displayed: 'Status: 200 OK', 'Time: 27 ms', 'Size: 5.8 KB', and options to 'Save as example' and 'Copy'. The main body area shows the image file, which is the logo for 'AL-FATAH' featuring a stylized orange and green design above the word 'AL-FATAH' in yellow capital letters.

Figure 4.1.2.12.46 Postman Testing

Lastly, another function was implemented, but this time it was implemented for vendors. Figure 4.1.2.12.47 shows the implementation of 'Get Vendor Image'.

```
const getVendorImage = (req, res) => {
  const filename = req.params.filename;
  const imagePath = path.join(__dirname, '../uploads/vendors', filename);

  fs.readFile(imagePath, (err, data) => {
    if (err) {
      return res.status(404).json({ error: 'Image not found.' });
    }

    res.setHeader('Content-Type', 'image/png');
    res.send(data);
  });
};
```

Figure 4.1.2.12.47 Implementation of Get Vendor Image

Its endpoint was also defined in the routes folder as shown in Figure 4.1.2.12.48.

```
router.get('/vendors/:filename', imageController.getVendorImage);
```

Figure 4.1.2.12.48 End Point Implemented for Get Vendor Image

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.49.

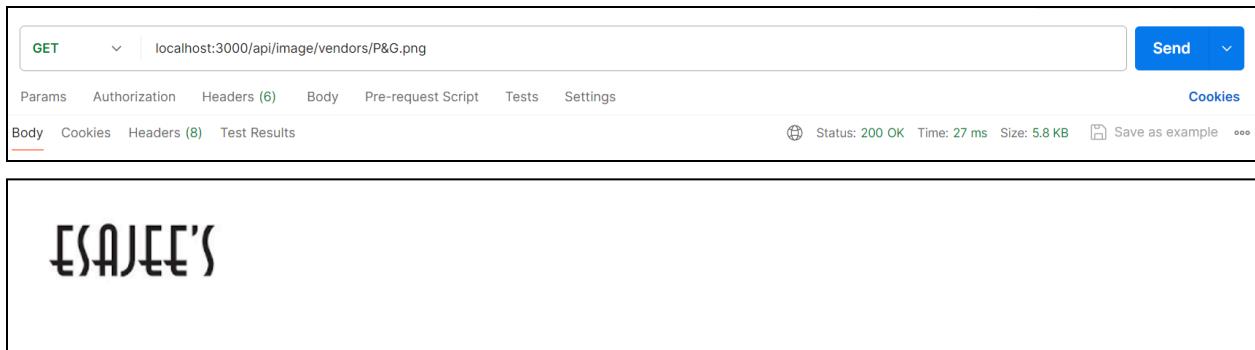


Figure 4.1.2.12.49 Postman Testing

```
const uploadProductImage = (req, res) => {

    const productName = req.body.product_name || req.query.product_name;

    if (!productName) {
        return res.status(400).json({ error: 'Product name is required.' });
    }

    upload.single('file')(req, res, (err) => {
        if (err) {
            return res.status(400).json({ error: 'Error uploading image.' });
        }

        if (!req.file) {
            return res.status(400).json({ error: 'No image uploaded.' });
        }

        const filename = `${productName}${path.extname(req.file.originalname)}`;
        const productImagePath = path.join(__dirname, '../uploads/products', filename);
    });
}
```

```
    fs.rename(req.file.path, productImagePath, (err) => {
        if (err) {
            return res.status(500).json({ error: 'Failed to move image to products folder.' });
        }

        res.status(200).json({ filename });
    });
};
```

Figure 4.1.2.12.50 Upload Product

```
router.post('/uploadproduct', imageController.uploadProductImage);
```

Figure 4.1.2.12.51 End Point Implemented for Upload Product

Now, for the order recommendation functional requirement, we needed to connect the flask API to the front end so that the predictions that are being generated by the machine learning models can be sent to the grocery store mobile application for the user to see. In order to do that, we decided to connect the front end of the mobile application with the flask API via NodeJS. In order to start this off, we first needed to install axios which is basically a javascript library used to make HTTP requests from node. Js. Figure 4.1.2.12.52 shows the installation of axios in our project.

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

Figure 4.1.2.12.52 Installation of Axios

This was downloaded by reading the documentation from [npm](#)'s website. Next Figure 4.1.2.12.53 shows the package installed for multer in the packages folder.

```
"axios": "^1.6.8",
```

Figure 4.1.2.12.53 Axios Package

Next, in order to implement this, we first made a new controller for predictions called ‘prediction Controller’ in our controllers folder. Since we had two types of predictions in our mobile application, a weekly prediction and a monthly prediction, we first decided to implement the weekly prediction. The logic that was used for the implementing this function was that first the name of the model (that is being used for that store to generate its predictions), the store number and the product number (as we need to generate the prediction based on a specific store and product) are being passed through the parameters. It then constructs the URL with the given model, store number and product number. It then sends a GET request to the specified URL using axios. Then if the request is successful, it returns the response back to the front end. Various error scenarios have been handled as well. Figure 4.1.2.12.54 shows the implementation of the weekly prediction function.

```

const db = require('../models');
const { Sequelize } = require('sequelize');
const axios = require('axios');

//usage ± fizzaadeel
const sendWeeklyPrediction = async (req, res) => {
  try {
    const { model, store_number, product_number} = req.params;

    if (!store_number || !product_number) {
      return res.status(400).json({ error: 'Store ID and product ID are required.' });
    }

    const url = `https://hammerhead-app-6m6td.ondigitalocean.app/get-weekly-prediction/${model}/${store_number}/${product_number}`

    const response = await axios.get(url);

    res.status(200).json(response.data);
  } catch (error) {
    console.error('Error sending weekly prediction request:', error);
  }
};

if (error.response) {
  res.status(error.response.status).json({ error: error.response.data });
} else if (error.request) {
  res.status(500).json({ error: 'No response received from prediction service.' });
} else {
  res.status(500).json({ error: 'Error in sending request to prediction service.' });
}
};

```

Figure 4.1.2.12.54 Implementation of Weekly Prediction

Figure 4.1.2.12.55 shows the endpoint defined in the routes folder.

```
router.get('/sendweeklyprediction/:model/:store_number/:product_number', predictionController.sendWeeklyPrediction);
```

Figure 4.1.2.12.55 End Point Implemented for Weekly Prediction

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.56. Here we have used ‘xgb’ as our model, ‘8’ as our store number which represents Jalal Sons and ‘2’ as our product number which represents a Coca Cola 500 ml bottle.

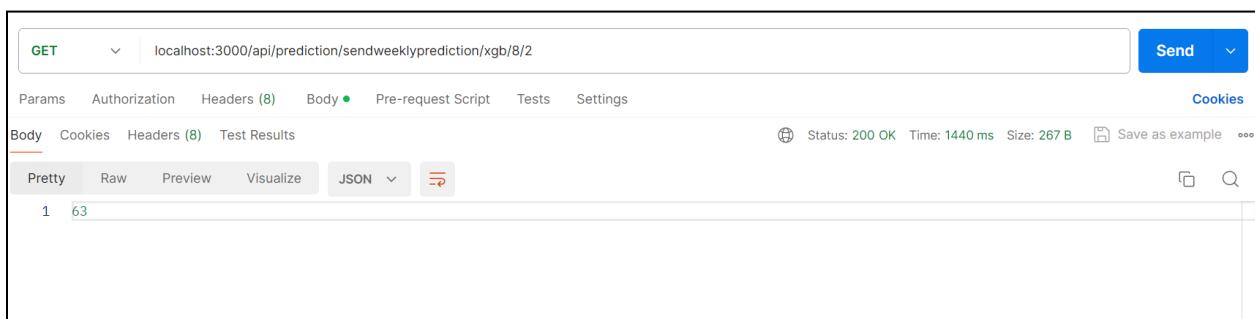


Figure 4.1.2.12.56 Postman Testing

Figure 4.1.2.12.57 shows the implementation of Monthly Prediction which was implemented using the same logic.

```
const sendMonthlyPrediction = async (req, res) => {
  try {
    const { model, store_number, product_number } = req.params;

    if (!store_number || !product_number) {
      return res.status(400).json({ error: 'Store ID and product ID are required.' });
    }

    const url = `https://hammerhead-app-6m6td.ondigitalocean.app/get-monthly-prediction/${model}/${store_number}/${product_number}`;

    const response = await axios.get(url);

    res.status(200).json(response.data);
  } catch (error) {
    console.error('Error sending monthly prediction request:', error);

    if (error.response) {
      res.status(error.response.status).json({ error: error.response.data });
    } else if (error.request) {
      res.status(500).json({ error: 'No response received from prediction service.' });
    } else {
      res.status(500).json({ error: 'Error in sending request to prediction service.' });
    }
  }
}
```

Figure 4.1.2.12.57 Implementation of Monthly Prediction

Figure 4.1.2.12.58 shows the endpoint defined in the routes folder.

```
router.get('/sendmonthlyprediction/:model/:store_number/:product_number', predictionController.sendMonthlyPrediction);
```

Figure 4.1.2.12.58 End Point Implemented for Weekly Prediction

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.12.59. Here, in order to draw a comparison we used the exact same values for model, store number and product number, the only difference being that this is a call for monthly prediction instead of a weekly prediction.

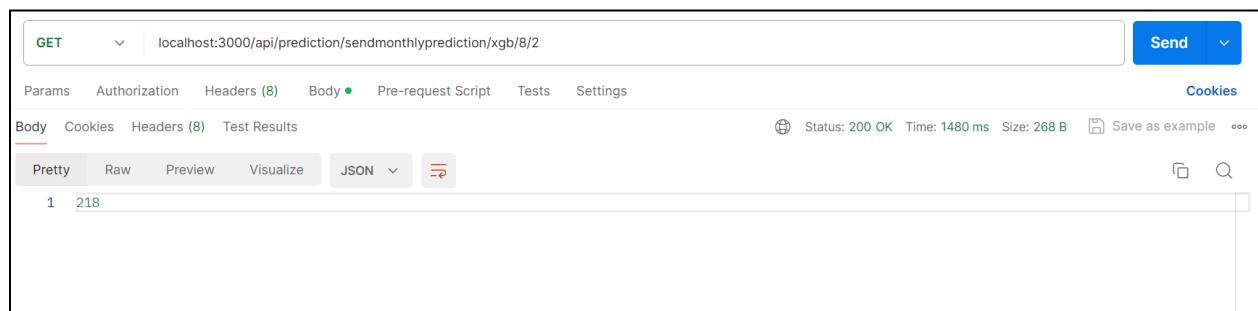


Figure 4.1.2.12.59 Postman Testing

4.1.2.13 Implementation of Admin Portal

We started with creating the admin on the backend for the admin portal after the applications were completed. We first worked on its model class, similar to all the model classes previously made, as shown in Figure 4.1.2.13.1.

```
module.exports = (sequelize, Datatypes)=>{
    return sequelize.define('admin',{
        admin_id:{
            type: Datatypes.INTEGER,
            primaryKey: true,
            autoIncrement: true
        },
        role:{
            type: Datatypes.STRING,
            isAlpha: true,
            notNull: true,
        },
        user_table_user_id: {
            type: Datatypes.INTEGER,
            references: {
                model: 'user_tables',
                key: 'user_id'
            },
            allowNull: false
        },
    },{
        underscored:true
    })
}
```

Figure 4.1.2.13.1 Admin Model

Next, All the CRUD functions for admin were created in its controller as shown in Figure 4.1.2.13.2, 4.1.2.13.3, 4.1.2.13.4, 4.1.2.13.5, and 4.1.2.13.6 respectively.

```

const db = require('../models')
const { Op } = require("sequelize");
const Admin = db.admin

no usages
const addAdmin = async (req, res) => {

    let data = {
        role: req.body.role,
        user_table_user_id: req.body.user_table_user_id
    }

    const admin = await Admin.create(data)
    res.status(201).send(admin)
}

```

Figure 4.1.2.13.2 Add Admin

```

const getAllAdmins = async (req, res) => {

    const admins = await Admin.findAll({})
    res.status(200).send(admins)

}

```

Figure 4.1.2.13.3 Get All Admins

```

const getOneAdmin = async (req, res) => {

    let admin_id = req.params.admin_id
    let admin = await Admin.findOne({ where: { admin_id: admin_id }})
    res.status(200).send(admin)

}

```

Figure 4.1.2.13.4 Get One Admin

```

const updateAdmin = async (req, res) => {

    let admin_id = req.params.admin_id
    const admin = await Admin.update(req.body, { where: { admin_id: admin_id }})
    res.status(200).send(admin)

}

```

Figure 4.1.2.13.5 Update Admin

```

const deleteAdmin = async (req, res) => {

  let admin_id = req.params.admin_id
  await Admin.destroy({ where: { admin_id: admin_id } })
  res.status(200).send('Admin is deleted !')

}

module.exports = {
  addVendor,
  getAllVendors,
  getOneVendor,
  updateVendor,
  deleteVendor
}

```

Figure 4.1.2.13.6 Delete Admin

Next, all the end points for the functions shown above were implemented in the routes folder as shown in Figure 4.1.2.13.7.

```

var router = require('express').Router();

const adminController = require('../controllers/adminController')

/* GET users listing. */
// router.get('/', function(req, res, next) {
//   res.send('respond with a resource');
// });

router.post('/addadmin',adminController.addAdmin)
router.get('/alladmins',adminController.getAllAdmins)
router.get('/:admin_id', adminController.getOneAdmin)
router.put('/:admin_id', adminController.updateAdmin)
router.delete('/:_id', adminController.deleteAdmin)
module.exports = router;

```

Figure 4.1.2.13.7 End Point implemented for Admin

Lastly, its relation with the user table was established in the index file of the models folder. It has a one to one relation with the user table (the same as grocery store and vendor). Figure 4.1.2.13.8 shows the relation.

```

db.user_table.hasOne(db.admin);
db.admin.belongsTo(db.user_table);

```

Figure 4.1.2.13.8 Relation between admin and user table

We then worked on the implementation of the functional requirements for the admin portal. We started off with implementing the “Add Category” first. Figure 4.1.2.13.9 shows the implementation of this particular functional requirement in the Product Category Controller.

```
const addProductCategory = async (req, res) => {

  let data = {
    category_name: req.body.category_name,
    image: req.body.image
  }

  const category = await Category.create(data)
  res.status(200).send(category)
}
```

Figure 4.1.2.13.9 Implementation of Add Category

The user which in this case is the admin would have the ability to add product category information which would include the category name and its image. Figure 4.1.2.13.10 shows its end point defined in the routes folder.

```
router.post('/addproductcategory', productcategoryController.addProductCategory)
```

Figure 4.1.2.13.10 End Point Implemented for Add Category

The functional requirement that was covered next was update category. The admin can update any information he wants about the product’s category. Figure 4.1.2.13.11 shows the implementation of ‘Update Category’.

```
const updateProductCategory = async (req, res) => {

  let product_category_id = req.params.product_category_id
  const category = await Category.update(req.body, { where: { product_category_id: product_category_id } })
  res.status(200).send(category)
}
```

Figure 4.1.2.13.11 Implementation of Update Category

Similar to the previous functional requirement, the end point for this was also defined in the routes folder as shown on Figure 4.1.2.13.12.

```
router.put('/:product_category_id', productcategoryController.updateProductCategory)
```

Figure 4.1.2.13.12 End Point Implemented for Update Category

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.13.13.

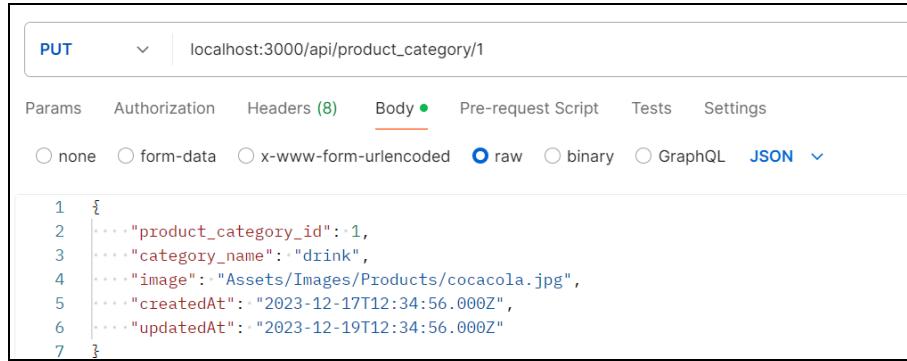


Figure 4.1.2.13.13 Postman Testing

Next, the functional requirement that was implemented was delete category. Figure 4.1.2.13.14 shows the implementation of ‘Delete Category’.

```
const deleteProductCategory = async (req, res) => {

  let product_category_id = req.params.product_category_id
  await Category.destroy({ where: { product_category_id: product_category_id } })
  res.status(200).send('Product Category is deleted !')

}
```

Figure 4.1.2.13.14 Implementation of Delete Category

The admin also has the ability to delete a category in case it is not being utilised. So by using the category id they can delete the category. Figure 4.1.2.13.15 shows the implementation of its end point in the routes folder.

```
router.delete('/:product_category_id', productcategoryController.deleteProductCategory)
```

Figure 4.1.2.13.15 End Point Implemented for Delete Category

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.13.16.

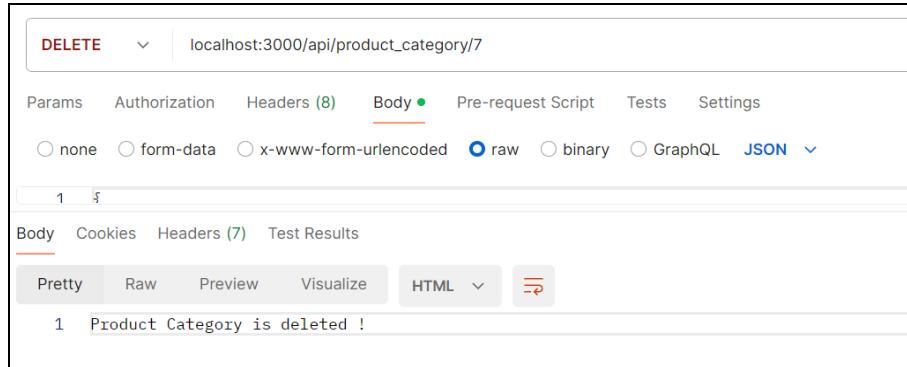


Figure 4.1.2.13.16 Postman Testing

Next, we implemented another functional requirement called “Add Product”. Figure 4.1.2.13.17 shows the implementation of this particular functional requirement in the Product Controller.

```
const addProduct = async (req, res) => {

  let data = {
    product_name: req.body.product_name,
    image: req.body.image,
  }

  const product = await Product.create(data)
  res.status(200).send(product)
}
```

Figure 4.1.2.13.17 Implementation of Add Product

The user which in this case is the admin would have the ability to add product information which would include the product name and its image. Figure 4.1.2.13.18 shows its end point defined in the routes folder.

```
router.post('/addproduct', productController.addProduct)
```

Figure 4.1.2.13.18 End Point Implemented for Add Product

The functional requirement that was covered next was to update the product. The admin can update any information about the product. Figure 4.1.2.13.19 shows the implementation of ‘Update Product’.

```

const updateProduct = async (req, res) => {

  let product_id = req.params.product_id
  const product = await Product.update(req.body, { where: { product_id: product_id }})
  res.status(200).send(product)

}

```

Figure 4.1.2.13.19 Implementation of Update Product

Similar to the previous functional requirement, the end point for this was also defined in the routes folder as shown on Figure 4.1.2.13.20.

```
router.put('/:product_id', productController.updateProduct)
```

Figure 4.1.2.13.20 End Point Implemented for Update Product

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.13.21.

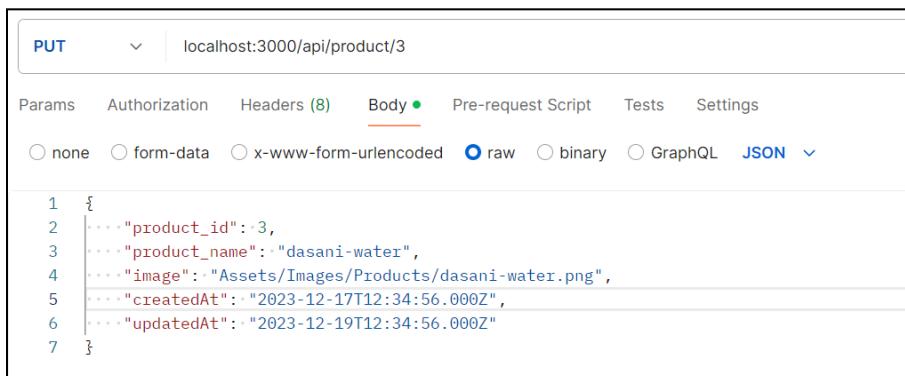


Figure 4.1.2.13.21 Postman Testing

Next, the functional requirement that was implemented was delete product. Figure 4.1.2.13.22 shows the implementation of 'Delete Product'.

```

const deleteProduct = async (req, res) => {

  let product_id = req.params.product_id
  await Product.destroy({ where: { product_id: product_id } })
  res.status(200).send('Product is deleted !')

}

```

Figure 4.1.2.13.22 Implementation of Delete Product

The admin also has the ability to delete a product in case it is not being utilised. So by using the product id they can delete the product. Figure 4.1.2.13.23 shows the implementation of its end point in the routes folder.

```
router.delete('/:product_id', productController.deleteProduct)
```

Figure 4.1.2.13.23 End Point Implemented for Delete Product

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.13.24.

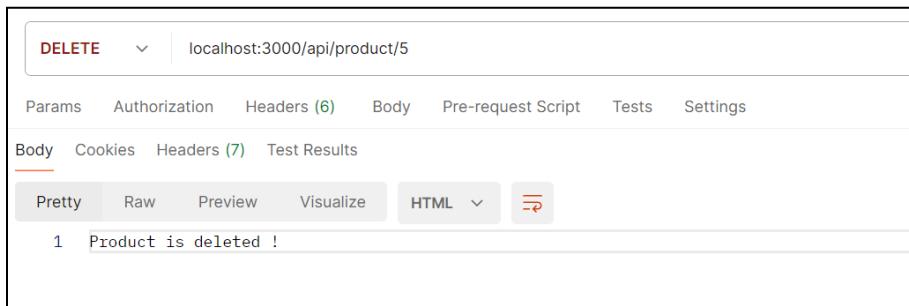


Figure 4.1.2.13.24 Postman Testing

We then worked on the implementation of the “View Grocery Store Profile”. Figure 4.1.2.13.25 shows the implementation of this particular functional requirement in the Grocery Store Controller.

```
const getOneStore = async (req, res) => {

  let store_id = req.params.store_id
  let store = await Store.findOne({ where: { store_id: store_id }})
  res.status(200).send(store)

}
```

Figure 4.1.2.13.25 Implementation of View Grocery Store Profile

The user, which in this case is the admin, would have the ability to view the grocery store’s profile which would include their name, store’s name, address etc. Figure 4.1.2.13.26 shows its end point defined in the routes folder.

```
router.get('/:store_id', grocerystoreController.getOneStore)
```

Figure 4.1.2.13.26 End Point Implemented for Grocery Store Profile

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.13.27.

```

1  {
2    "store_id": 1,
3    "store_name": "Esajees",
4    "image": "stores/esajees.png",
5    "store_address": "Shop 175, Y Block Market, DHA Phase 3",
6    "user_table_user_id": 1,
7    "createdAt": "2023-12-17T12:34:56.000Z",
8    "updatedAt": "2023-12-19T12:34:56.000Z",
9    "userTableUserId": 1
10 }

```

Figure 4.1.2.13.27 Postman Testing

Next, we implemented the “View Vendor Profile”. Figure 4.1.2.13.28 shows the implementation of this particular functional requirement in the Vendor Controller.

```

const getOneVendor = async (req, res) => {

  let vendor_id = req.params.vendor_id
  let vendor = await Vendor.findOne({ where: { vendor_id: vendor_id } })
  res.status(200).send(vendor)

}

```

Figure 4.1.2.13.28 Implementation of Vendor Profile

The user, which in this case is the admin, would have the ability to view the vendor’s profile which would include their name etc. Figure 4.1.2.13.29 shows its end point defined in the routes folder.

```
router.get('/:vendor_id', vendorController.getOneVendor)
```

Figure 4.1.2.13.29 End Point Implemented for View Vendor

To ensure that it is working perfectly, it was tested through Postman as shown in Figure 4.1.2.13.30.

```

1 {
2   "vendor_id": 1,
3   "vendor_name": "P&G",
4   "delivery_locations": "Gulberg",
5   "image": "Assets/Images/Stores/esajees.png",
6   "user_table_user_id": 1,
7   "createdAt": "2023-12-17T12:34:56.000Z",
8   "updatedAt": "2023-12-19T12:34:56.000Z",
9   "userTableUserId": 1
10 }

```

Figure 4.1.2.13.30 Postman Testing

4.1.2.14 Deployment

We deployed our database (MySQL) and backend system (Node.js) on Digital Ocean. We started off by creating a database cluster with datacenter region as New York, MySQL database engine, and shared cpu plan: 1 GB RAM / 1vCPU / 10 GB Disk (Figure 4.1.2.14.1).



Figure 4.1.2.14.1 Database Cluster on Digital Ocean

Next, a database named ‘wasail’ was created and connection details (Figure 4.1.2.14.2) were used on backend (node.js) to create tables in the database using Sequelize.

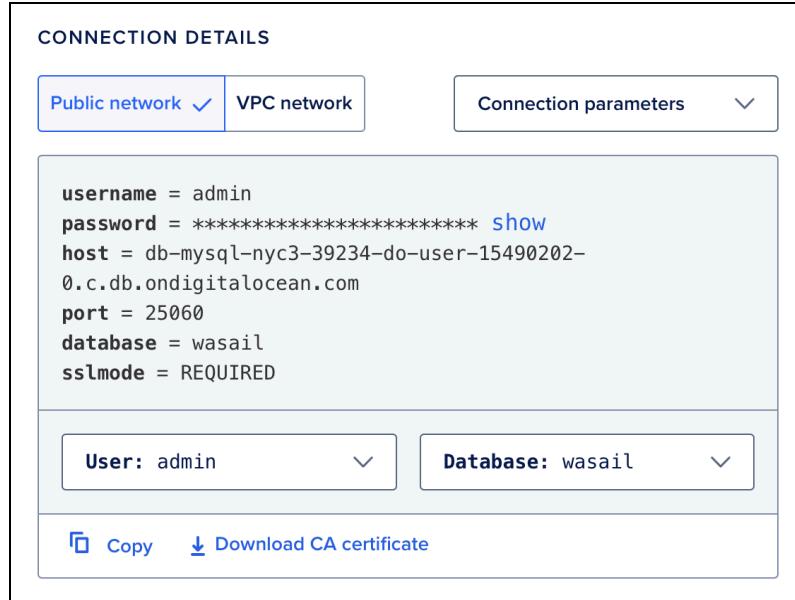


Figure 4.1.2.14.2 Connection details for the Wasail database

Upon entering the connection details and running the backend system, we got a Connection Timed Out error (Figure 4.1.2.14.3).

```
/usr/local/bin/node /Users/yrrebeere/Documents/University/8th Semester/Final Year Project/prj-403/BackEnd/Node.js/Wasail/bin/www
{
  db: {
    username: 'admin',
    password: '',
    database: 'wasail',
    host: 'db-mysql-nyc3-39234-do-user-15490202-0.c.db.ondigitalocean.com'
  }
}
listen on port 4000
Unable to connect to the database ConnectionError [SequelizeConnectionError]: connect ETIMEDOUT
```

Figure 4.1.2.14.3 Connection timed out error

After some research, we found out this occurs when the database's firewall doesn't allow you to connect to the database from your local machine. In order to fix this, we added the computer's IP as a trusted source to the database cluster (Figure 4.1.2.14.4).

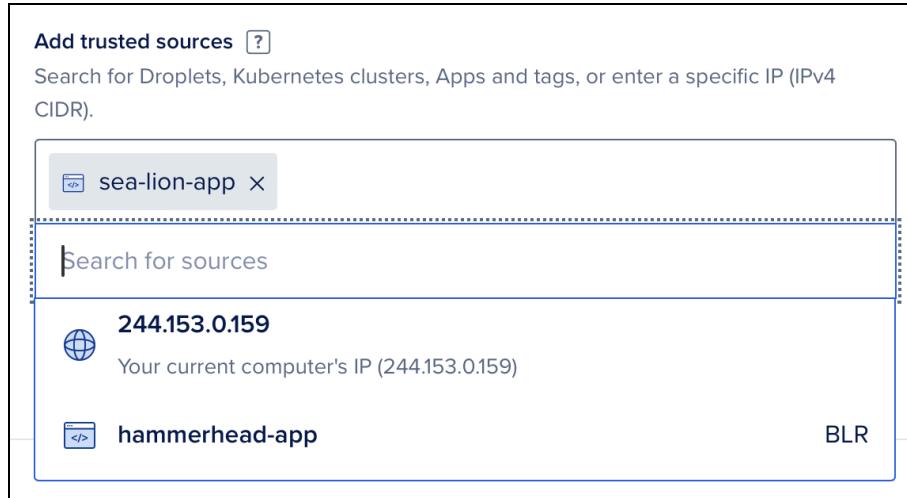


Figure 4.1.2.14.4 Adding my computer's IP in trusted sources

Once the computer's IP address was added, the database was accessed (Figure 4.1.2.14.5) and we were able to create the tables using Sequelize.

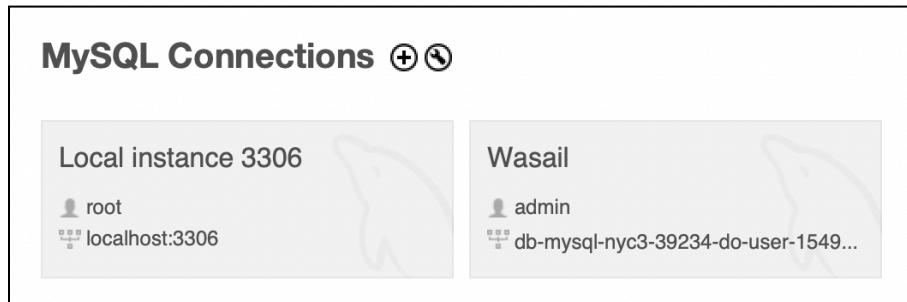


Figure 4.1.2.14.5 Wasail database on MySQL Workbench

However, the database was only accessible through this one computer. So, we had to add the IP address of every machine that was going to be used to access the database using the node.js system locally (Figure 4.1.2.14.6).

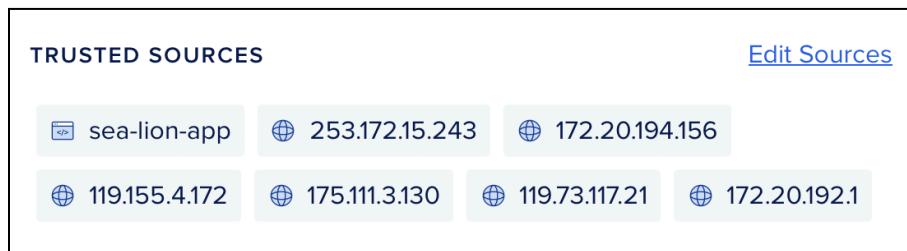


Figure 4.1.2.14.6 Adding all IP addresses as trusted sources

Next, we deployed the backend system (Node.js) as an application (sea-lion-app) on Digital Ocean (Figure 4.1.2.14.7). The application is deployed on the Bangalore server (as it is the closest) with the basic plan: 1 GB RAM / 1 vCPU / 40 GB Bandwidth.

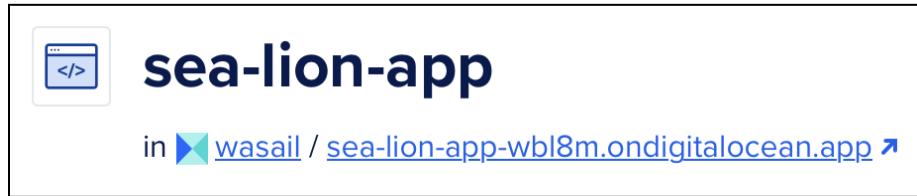


Figure 4.1.2.14.7 Backend system deployed on Digital Ocean

The deployment is directly connected to the GitHub repository. Meaning, everytime someone pushes a commit, the deployment is updated (Figure 4.1.2.14.8)

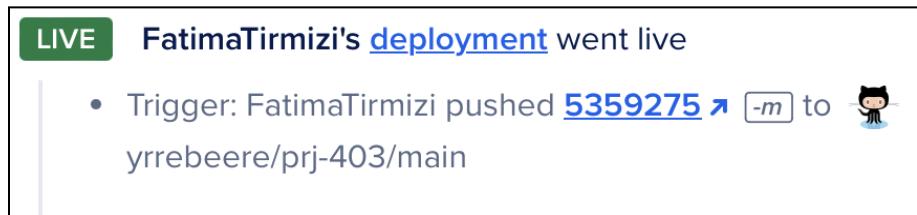
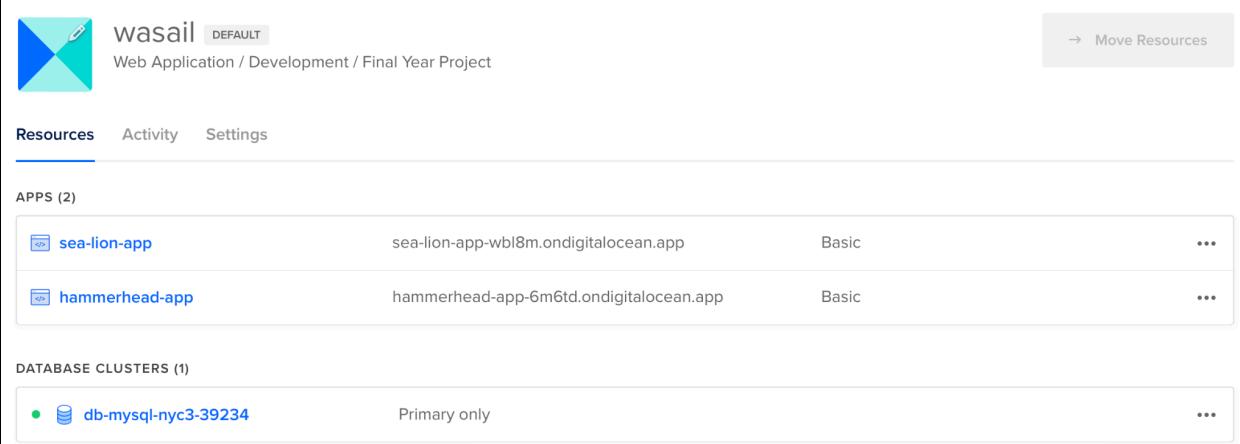


Figure 4.1.2.14.8 Deployment updated on each commit

Now, we have 3 deployments on Digital Ocean: 2 Applications and 1 Database (Figure 4.1.2.14.9). The Flask API to demo the machine learning model from FYP I (hammerhead-app), the Node.js backend system (sea-lion-app), and the MySQL database (db-mysql-nyc3-39234).



The screenshot shows the Digital Ocean Wasail interface. At the top, it displays the project name "Wasail" with a "DEFAULT" tag, a "Move Resources" button, and tabs for "Resources", "Activity", and "Settings". Under the "Resources" tab, there are sections for "APPS (2)" and "DATABASE CLUSTERS (1)".

Resource Type	Name	URL	Plan	Actions
APP	sea-lion-app	sea-lion-app-wbl8m.ondigitalocean.app	Basic	...
APP	hammerhead-app	hammerhead-app-6m6td.ondigitalocean.app	Basic	...
DATABASE CLUSTER	db-mysql-nyc3-39234	Primary only		...

Figure 4.1.2.14.9 All deployments on Digital Ocean

The deployed system was tested on both frontend and backend (using Postman). Wherever on the Flutter applications the data was being fetched from <http://10.0.2.2:3000> (localhost), now it is being fetched from <https://sea-lion-app-wbl8m.ondigitalocean.app> (Figure 4.1.2.14.10).

```

Future<void> _fetchAndDisplayProductInventories(String vendorId) async {
    final response = await http.get(
        Uri.parse('https://sea-lion-app-wbl8m.ondigitalocean.app/api/product_inventory/search/$vendorId'),
    );

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

```

Figure 4.1.2.14.10 Frontend fetching data from the deployed backend system

4.1.2.15 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.

4.1.3 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.

4.1.3.1 Vendor App

Test Case 1: Language Selection

Test Scenario: The user wants to select their preferred language.

Preconditions: The user has not specified their preferred language.

Test Steps:

1. Open the Vendor Application.
2. Navigate to the language selection option.
3. Select English as the preferred language.

Test Data:

1. English.
- **Expected Result:** The app's text is displayed in English. English is saved as the preferred language.
 - **Alternate Flow:** User selects Urdu as the preferred language.

Test Data:

1. Urdu.
- **Expected Result:** The app's text is displayed in Urdu. Urdu is saved as the preferred language.

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.:

Test Steps:

1. The user enters their phone number.

Test Data:

1. 03224377009.
- **Expected Result:** The system should display a confirmation screen.
 - **Alternate Flow:** None

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.

Test Steps:

1. The user selects the option to confirm their number.
- **Expected Result:** The user should be able to confirm their phone number.
- **Alternate Flow:** The user can select the edit option given by the system.
- **Expected Result:** The user should be able to edit their phone number, in case it is incorrect.

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.

Test Steps:

1. The user has confirmed their phone number.
- **Expected Result:** The phone number exists, the user should be directed to the login screen.
- **Alternate Flow:** The phone number does not exist.
- **Expected Result:** The user should be directed to the registration screen.

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.

Test Steps:

1. The system can confirm the phone number of the user.
 2. The system generates an OTP code and sends it via SMS.
 3. The user waits for 60 seconds to allow resending OTP.
 4. The system resends another OTP after the timer closes.
 5. The system verifies the validity of the OTP code by comparing it to the generated code sent to the given phone number.
 6. Upon verification, the user shall be directed to the account details page.
- **Expected Result:** The phone number exists, the user should be directed to the login screen.
 - **Alternate Flow:** The phone number does not exist.
 - **Expected Result:** The user should be directed to the registration screen.

Test Case 6: Login

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

Preconditions: The user is on the login page

Test Steps:

1. The user enters their password.

Test Data:

1. password@123.
- **Expected Result:** The system should log the user in.
 - **Alternate Flow:** User enters an incorrect password.

Test Data:

1. passwd@123.
- **Expected Result:** The system asks for a re-entry.

Test Case 7: Logout

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

Preconditions: The user is logged in.

Test Steps:

1. The system should allow the user to view the logout option
 2. The user clicks on the logout option.
- **Expected Result:** The user is logged out of the system.
 - **Alternate Flow:** None

Test Case 8: Reset Password

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

Preconditions: The user is registered.

Test Steps:

1. The system should give the user the option for resetting their password.

Test Data:

1. Old Password: password@123.
 2. New Password: university_321
- **Expected Result:** The user resets their password.
 - **Alternate Flow:** None

Test Case 9: View Profile

Test Scenario: The system should allow the user to view their profile.

Preconditions: The user is on their profile.

Test Steps:

1. The user navigates to the menu option to have details displayed to him including his profile.
 2. The system retrieves the user's profile details and displays them to him.
- **Expected Result:** The user can see their profile.
 - **Alternate Flow:** None

Test Case 10: Vendor Registration

Test Scenario: The user should enter their details.

Preconditions: The user is on the registration page.

Test Steps:

1. The user enters their username.
 2. The user enters their password.
 3. The user selects the delivery area.
 4. The user uploads their picture.
- **Expected Result:** The user is registered.
 - **Alternate Flow:** None

Test Case 11: Valid Password

Test Scenario: The user should enter a password.

Preconditions: The user is on the registration page.

Test Steps:

1. The user enters a password.

Test Data:

1. password@123.
- **Expected Result:** The system accepts the password.
 - **Alternate Flow:** The user enters an invalid password.

Test Data:

1. lahore5.
- **Expected Result:** The system should indicate that it is an invalid password and ask for re-entry.

Test Case 12: Username Exists

Test Scenario: The user should enter a username.

Preconditions: The user is on the registration page.

Test Steps:

1. The user enters their username.
- **Expected Result:** The username does not exist in the database, hence the system accepts the username.
- **Alternate Flow:** The user enters a username that exists in the database.
- **Expected Result:** The system should indicate that it exists and ask for re-entry.

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.

Test Steps:

1. The system directs the user to the homepage where the search bar is.
2. The user can search the name of the product from their own inventory.
3. The system fetches results for the product that has been searched.

Test Data:

1. Lays.
- **Expected Result:** The system displays the product the user has searched for.
 - **Alternate Flow:** The product does not exist in their inventory.

Test Data:

1. Doritos.
- **Expected Result:** The system does not display the product the user has searched for.

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.

Test Steps:

1. The user searches for a product by entering its name to add to their inventory.
2. The user adds details after selecting a product namely price, listed amount, and available amount.
3. The system confirms the addition to the user inventory.

Test Data:

1. Coca Cola.
- **Expected Result:** The system adds the product in the user's inventory.
 - **Alternate Flow:** None.

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.

Test Steps:

1. The user enters a product name.

Test Data:

1. Water.

- **Expected Result:** All products containing the search term should be displayed.
- **Alternate Flow:** The product does not exist in the database.

Test Data:

1. Corn.

- **Expected Result:** No results should be displayed.

Test Case 16: Restrict Product Duplication

Test Scenario: The system should restrict duplication of products.

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

Test Steps:

1. The user clicks on the product to add to their inventory.
2. The system shows an alert to inform of duplication.

Test Data:

1. Lays.

- **Expected Result:** The product should not be added to the inventory.
- **Alternate Flow:** None.

Test Case 17: Remove Product

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

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

Test Steps:

1. The user enters a product name.
2. The user presses the delete option displayed.

Test Data:

1. Lays.

- **Expected Result:** The system should delete the product from the user's inventory and it should not be displayed to the user.
- **Alternate Flow:** None.

Test Case 18: 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.

Test Steps:

3. The system displays the product inventory page for the user.
4. The system gives the user the option to edit the product details: listed amount, available amount, and price.
5. The user edits the product details.
6. The system saves the edited product details.

Test Data:

1. Selected Product: Coca Cola
 - a. Old details:
 - i. Price: 120
 - ii. Listed: 28
 - iii. Available: 23
 2. Selected Product: Coca Cola
 - a. Old details:
 - i. Price: 120
 - ii. Listed: 35
 - iii. Available: 28
- **Expected Result:** The details should be saved and updated in the inventory.
 - **Alternate Flow:** None.

Test Case 19: View Inventory

Test Scenario: The system should allow the user to view their inventory.

Preconditions: The user is logged in and on the inventory page.

Test Steps:

1. The user has clicked on the inventory page and can see their inventory displayed by the system.
- **Expected Result:** The system should display the user's inventory with details.
- **Alternate Flow:** None.

Test Case 20: 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 current orders page.

Test Steps:

1. The user has clicked on the current orders page and can see their orders from the stores displayed by the system.
- **Expected Result:** The system should display the current orders of the user from all stores that are yet to be delivered.
- **Alternate Flow:** None.

Test Case 21: View Grocery Store List

Test Scenario: The user wants to view the grocery store list.

Preconditions: The user is logged in.

Test Steps:

1. The user clicks on the "Store List" section in the menu.
- **Expected Result:** The system should display a list of grocery stores.
- **Alternate Flow:** None.

Test Case 22: 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.

Test Steps:

1. The user clicks on a Grocery Store in “Store List”.

Test Data:

1. Jalal Sons.

- **Expected Result:** The system should display all details of the grocery store namely name, image, and address.
- **Alternate Flow:** None.

Test Case 23: View Grocery Store Current Order

Test Scenario: The system should allow the user to view the current order placed by the grocery store.

Preconditions: The user is on the grocery store’s profile page.

Test Steps:

1. The user navigates to the grocery store's list from the menu.
 2. The user clicks on the store's profile to view its current orders.
- **Expected Result:** The system should display a list of all current orders of that grocery store to the user on the grocery store’s profile.
 - **Alternate Flow:** None.

Test Case 24: View Order History

Test Scenario: The system should enable the user to view the orders they have delivered.

Preconditions: The user is on the orders history page.

Test Steps:

1. The user has clicked on the order history page and can see their delivered orders.
- **Expected Result:** The system should display all the orders they have delivered to different grocery stores.
 - **Alternate Flow:** None.

Test Case 25: Order Dispatch Tracking

Test Scenario: The system should allow the user to track the order.

Preconditions: The user is on the current orders page.

Test Steps:

1. The user selects the option ‘In Process’.

Test Data:

1. In Process.

- **Expected Result:** The system displays the order as ‘in process’ and on the current orders page.
- **Alternate Flow:** The user selects the option ‘On Its Way’.

Test Data:

1. On Its Way.

- **Expected Result:** The system displays the order as ‘On Its Way’ and on the current orders page.
- **Alternate Flow:** The user selects the option ‘Delivered’.

Test Data:

1. Delivered.

- **Expected Result:** The system displays the order as ‘Delivered’ and on the order history page.

Test Case 26: Edit Profile

Test Scenario: The system should allow the user to edit their profile details.

Preconditions: The user is on their profile.

Test Steps:

1. The system displays the profile details for the user.
2. The system gives the user the option to edit profile details.
3. The user can edit the profile details.
4. The system saves the edit to the profile.

Test Data:

1. Old Name: Unilever
2. New Name: Uni-lever

- **Expected Result:** The system should display the profile according to the edited version and should update the information in the database.
- **Alternate Flow:** None.

4.1.3.2 Grocery Store App

Test Case 27: Language Selection

Test Scenario: The user wants to select their preferred language.

Preconditions: The user has not specified their preferred language.

Test Steps:

1. Open the Grocery Store Application.
2. Navigate to the language selection option.
3. Select English as the preferred language.

Test Data:

1. English.
- **Expected Result:** The app's text is displayed in English. English is saved as the preferred language.
 - **Alternate Flow:** User selects Urdu as the preferred language.

Test Data:

1. Urdu.
- **Expected Result:** The app's text is displayed in Urdu. Urdu is saved as the preferred language.

Test Case 28: Phone Registration

Test Scenario: The user wants to create an account using their phone number.

Preconditions: The user has opened the app.

Test Steps:

1. The user enters their phone number.

Test Data:

1. 03224377009.
- **Expected Result:** The system should display a confirmation screen.
 - **Alternate Flow:** None

Test Case 29: Phone Number Confirmation

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

Preconditions: The user has entered their phone number.

Test Steps:

1. The user selects the option to confirm their number.
- **Expected Result:** The user should be able to confirm their phone number.
 - **Alternate Flow:** The user can select the edit option given by the system.
 - **Expected Result:** The user should be able to edit their phone number, in case it is incorrect.

Test Case 30: 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.

Test Steps:

1. The user has confirmed their phone number.
- **Expected Result:** The phone number exists, the user should be directed to the login screen.
- **Alternate Flow:** The phone number does not exist.
- **Expected Result:** The user should be directed to the registration screen.

Test Case 31: 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.

Test Steps:

1. The system can confirm the phone number of the user.
 2. The system generates an OTP code and sends it via SMS.
 3. The user waits for 60 seconds to allow resending OTP.
 4. The system resends another OTP after the timer closes.
 5. The system verifies the validity of the OTP code by comparing it to the generated code sent to the given phone number.
 6. Upon verification, the user shall be directed to the account details page.
- **Expected Result:** The phone number exists, the user should be directed to the login screen.
 - **Alternate Flow:** The phone number does not exist.
 - **Expected Result:** The user should be directed to the registration screen.

Test Case 32: Login

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

Preconditions: The user is on the login page

Test Steps:

1. The user enters their password.

Test Data:

1. password@123.
- **Expected Result:** The system should log the user in.
 - **Alternate Flow:** User enters an incorrect password.

Test Data:

1. password@123.
- **Expected Result:** The system asks for a re-entry.

Test Case 33: Logout

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

Preconditions: The user is logged in.

Test Steps:

1. The system should allow the user to view the logout option
 2. The user clicks on the logout option.
- **Expected Result:** The user is logged out of the system.
 - **Alternate Flow:** None

Test Case 34: Reset Password

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

Preconditions: The user is registered.

Test Steps:

1. The system should give the user the option for resetting their password.

Test Data:

1. Old Password: password@123.
2. New Password: university_321

- **Expected Result:** The user resets their password.
- **Alternate Flow:** None

Test Case 35: View Profile

Test Scenario: The system should allow the user to view their profile.

Preconditions: The user is on their profile.

Test Steps:

1. The user navigates to the menu option to have details displayed to him including his profile.
 2. The system retrieves the user's profile details and displays it to them.
- **Expected Result:** The user can see their profile.
 - **Alternate Flow:** None

Test Case 36: Account Details

Test Scenario: The user should enter their details.

Preconditions: The user is on the registration page.

Test Steps:

1. The user enters their username.
 2. The user enters their password.
 3. The user selects the delivery area.
 4. The user uploads their picture.
- **Expected Result:** The user is registered.
 - **Alternate Flow:** None

Test Case 37: Valid Password

Test Scenario: The user should enter a password.

Preconditions: The user is on the registration page.

Test Steps:

1. The user enters a password.

Test Data:

1. password@123.
- **Expected Result:** The system accepts the password.
 - **Alternate Flow:** The user enters an invalid password.

Test Data:

1. ahore5.
- **Expected Result:** The system should indicate that it is an invalid password and ask for re-entry.

Test Case 38: Username Exists

Test Scenario: The user should enter a username.

Preconditions: The user is on the registration page.

Test Steps:

1. The user enters their username.
- **Expected Result:** The username does not exist in the database, hence the system accepts the username.
 - **Alternate Flow:** The user enters a username that exists in the database.
 - **Expected Result:** The system should indicate that it exists and ask for re-entry.

Test Case 39: Search Product

Test Scenario: The system should allow the user to search for a product.

Preconditions: The user is logged in and on the home page.

Test Steps:

1. The system directs the user to the homepage where the search bar is.
2. The user can search the name of the product.
3. The system fetches results for the product that has been searched.

Test Data:

1. Lipton.
- **Expected Result:** The system displays the product the user has searched for along with the vendors that sell them.
 - **Alternate Flow:** The product does not exist.

Test Data:

1. Pringles.
- **Expected Result:** The system does not display the product the user has searched for.

Test Case 40: Search Category

Test Scenario: The system should allow the user to search for a category.

Preconditions: The user is logged in and on the home page.

Test Steps:

1. The system directs the user to the homepage where the search bar is.
2. The user can search the name of the category.
3. The system fetches results for the category that has been searched.

Test Data:

1. Tea and Coffee.
- **Expected Result:** The system displays all the products in that category to the user along with the vendors that sell them.
 - **Alternate Flow:** The category does not exist.

Test Data:

1. Confectionery.
- **Expected Result:** The system does not display anything.

Test Case 41: Search Vendor

Test Scenario: The system should allow the user to search for a vendor.

Preconditions: The user is logged in and on the home page.

Test Steps:

1. The system directs the user to the homepage where the search bar is.
2. The user can search the name of the vendor.
3. The system fetches results for the vendor that has been searched.

Test Data:

1. Unilever.
- **Expected Result:** The system displays the vendors along with the products that they sell.
 - **Alternate Flow:** The vendor does not exist.

Test Data:

1. Bisconni.
- **Expected Result:** The system does not display anything.

Test Case 42: Browse Category

Test Scenario: The system should allow the user to select a category from the home page directly.

Preconditions: The user is logged in and on the home page.

Test Steps:

1. The system directs the user to the homepage.
 2. The system displays all the categories under the ‘Category’ section.
 3. The user can select any category.
- **Expected Result:** The system displays all the products in that category to the user along with the vendors that sell them.
 - **Alternate Flow:** None.

Test Case 43: View Vendor Profile

Test Scenario: The user wants to view the profile of a specific vendor.

Preconditions: The user is on the vendor list page.

Test Steps:

1. The user clicks on a Vendor in “Vendor List”.

Test Data:

1. Unilever.
- **Expected Result:** The system should display all details of the vendor.
 - **Alternate Flow:** None

Test Case 44: Add Vendor to Vendor List

Test Scenario: The user wants to add a vendor to their list.

Preconditions: The user is on the vendor's profile.

Test Steps:

1. The user clicks on the add button.
- **Expected Result:** The system should add that vendor to the user's vendor list and display it in the list.
- **Alternate Flow:** None.

Test Case 45: Contact Vendor

Test Scenario: The user wants to contact the vendor.

Preconditions: The user is on the vendor's profile.

Test Steps:

1. The user clicks on the contact button.
- **Expected Result:** The system should contact the vendor.
- **Alternate Flow:** None.

Test Case 46: View Products on the Vendor's Profile

Test Scenario: The user wants to view products on the vendor's profile.

Preconditions: The user is on the vendor's profile.

Test Steps:

1. The user clicks on the vendor's profile.
 2. The products are being displayed to the user.
- **Expected Result:** The system should display the products that the vendor sells on the vendor's profile.
 - **Alternate Flow:** None.

Test Case 47: View Searched Product

Test Scenario: The user wants to view the searched product on the vendor's profile.

Preconditions: The user is on the vendor's profile.

Test Steps:

1. The user has searched for the product.
 2. The user clicks on the product being displayed.
 3. The user is on the vendor's profile.
- **Expected Result:** The system should display the searched product on the vendor's profile.
 - **Alternate Flow:** None.

Test Case 48: Select Products

Test Scenario: The system should allow the user to select the product that they want to order.

Preconditions: The user is on the vendor's profile.

Test Steps:

1. The user has clicked on the product they want to order.

Test Data:

1. Lipton Tea.
- **Expected Result:** The system should allow the user to select the product.
 - **Alternate Flow:** None.

Test Case 49: Order Recommendation

Test Scenario: The system should allow the user to view the recommended amount to order.

Preconditions: The user has selected the product.

Test Steps:

1. The user can see the weekly order recommendation for the product they have selected.

Test Data:

1. Lipton Tea.
- **Expected Result:** The system should display the weekly order recommendation.
 - **Alternate Flow:** The user can see the monthly order recommendation for the product they have selected.

Test Data:

1. Lipton Tea.
- **Expected Result:** The system should display the monthly order recommendation.

Test Case 50: Quantity Selection

Test Scenario: The system should allow the user to add their own amount to order.

Preconditions: The user has selected the product.

Test Steps:

1. The user has viewed the weekly and monthly recommendation.
2. The user can add the amount in the textbox named quantity.

Test Data:

1. Lipton Tea.
- **Expected Result:** The user adds the amount that they want to order.
 - **Alternate Flow:** None.

Test Case 51: Add to Cart

Test Scenario: The system should allow the user to add products to their cart in order to place an order.

Preconditions: The user has added the quantity of the product.

Test Steps:

1. The user has selected the ‘add to cart’ option.

Test Data:

1. Lipton Tea.

- **Expected Result:** The product has been added to the cart.
- **Alternate Flow:** None.

Test Case 52: View Cart

Test Scenario: The system should allow the user to view the cart.

Preconditions: The user has added a product to the cart.

Test Steps:

1. The user clicks on the cart button in order to view it.
- **Expected Result:** The system displays the cart for the user to view.
 - **Alternate Flow:** None.

Test Case 53: Update Product in Cart

Test Scenario: The system should allow the user to update the quantity of the product added in their cart.

Preconditions: The user has added a product to the cart.

Test Steps:

1. The user clicks on the ‘+’ button to update the quantity of the product.

Test Data:

1. Lipton Tea.

- **Expected Result:** The quantity of the product gets updated inside the cart.
- **Alternate Flow:** The user clicks on the ‘-’ button to update the quantity of the product.

Test Data:

1. Lipton Tea.

- **Expected Result:** The quantity of the product gets updated inside the cart.

Test Case 54: Remove Product from Cart

Test Scenario: The system should allow the user to remove one specific product from the cart.

Preconditions: The user has added a product to the cart.

Test Steps:

1. The user selects the delete option to remove a specific product they do not wish to order.

Test Data:

1. Lipton Tea.
- **Expected Result:** The product has been removed from the cart.
 - **Alternate Flow:** None.

Test Case 55: Clear Cart

Test Scenario: The system should allow the user to clear the cart in order to remove all the products from it.

Preconditions: The user has added products in the cart.

Test Steps:

1. The user selects the delete option to remove the product they do not wish to order.

Test Data:

1. Coca Cola
 2. Nescafe Coffee
 3. Lays Salted
- **Expected Result:** The products have been removed and the cart is empty.
 - **Alternate Flow:** None.

Test Case 56: Order Placement

Test Scenario: The system should allow the user to place the order.

Preconditions: The user has added products in the cart.

Test Steps:

1. The user selects the option to place the order.
- **Expected Result:** The order has been placed to the vendor.
 - **Alternate Flow:** None.

Test Case 57: View Orders

Test Scenario: The system should allow the user to view their current orders.

Preconditions: The user is logged in and on the current orders page.

Test Steps:

1. The user has clicked on the current orders page and can see the orders they have placed to vendors.
- **Expected Result:** The system should display the current orders of the user from that are yet to be delivered.
- **Alternate Flow:** None.

Test Case 58: Order tracking

Test Scenario: The system should allow the user to track the order.

Preconditions: The user is logged in and on the current orders page.

Test Steps:

1. The user can view the status of the order on the current orders page.
- **Expected Result:** The user can track the order by seeing its status.
- **Alternate Flow:** None.

Test Case 59: View Vendor List

Test Scenario: The user wants to view the vendor list.

Preconditions: The user is logged in.

Test Steps:

1. The user clicks on the "Vendor List" section in the menu.
- **Expected Result:** The system should display a list of the added vendors.
- **Alternate Flow:** None.

Test Case 60: View Order History

Test Scenario: The system should enable the user to view the orders that have been delivered.

Preconditions: The user is on the orders history page.

Test Steps:

1. The user has clicked on the order history page.
- **Expected Result:** The system should display all the orders they have been delivered.
- **Alternate Flow:** None.

Test Case 61: Edit Profile

Test Scenario: The system should allow the user to edit their profile details.

Preconditions: The user is on their profile.

Test Steps:

1. The system displays the profile details for the user.
2. The system gives the user the option to edit profile details.
3. The user can edit the profile details.
4. The system saves the edit to the profile.

Test Data:

1. Old Name: Jalal Sons
 2. New Name: Jalal_Sons
- **Expected Result:** The system should display the profile according to the edited version and should update the information in the database.
 - **Alternate Flow:** None.

4.1.3.3 Admin Portal

Test Case 62: Add New Admin

Test Scenario: The user wants to add a new admin.

Preconditions: The user is on the admin portal.

Test Steps:

1. The user clicks on the "Add Admin" button.
- **Expected Result:** The system should display the add admin page.
 - **Alternate Flow:** None.

Test Case 63: Add New Admin Details

Test Scenario: The system should allow the user to add details of the new admin.

Preconditions: The user is on the add new admin page.

Test Steps:

1. The user enters their phone number.
 2. The user enters their name.
 3. The user enters their password.
 4. The user enters their username.
 5. The user selects their language.
 6. The user enters their user type.
 7. The user selects their role.
 8. The user enters their email.
- **Expected Result:** A new admin is created.
 - **Alternate Flow:** None.

Test Case 64: Edit Admin Profile

Test Scenario: The system should allow the user to edit details of the admin.

Preconditions: The user is on the admin page.

Test Steps:

1. The system displays the update button next to each admin.
2. The system allows the user to update their information.
3. The system saves the updated information

Test Data:

1. Old Name: pizza
2. New Name: pizzaadeel
3. Old Admin Role: Moderator
4. New Admin Role: Viewer

- **Expected Result:** The system should display the new information and should update the information in the database.
- **Alternate Flow:** None.

Test Case 65: Delete Admin Profile

Test Scenario: The system should allow the user to delete an admin.

Preconditions: The user is on the admin page.

Test Steps:

1. The system displays the delete button next to each admin.
2. The user selects the delete option in order to delete an admin.

Test Data:

1. Pizza Adeel
- **Expected Result:** The system should delete the admin.
 - **Alternate Flow:** None.

Test Case 66: Search Admin

Test Scenario: The system should allow the user to search for an admin.

Preconditions: The user is on the admin page.

Test Steps:

1. The system displays a search bar.
2. The user searches for an admin using their username.

Test Data:

1. pizza

- **Expected Result:** The system displays the results according to the search.

- **Alternate Flow:** None.

Test Case 67: View Grocery Store Profile

Test Scenario: The system should allow the user to view the profile of a grocery store.

Preconditions: The user is on the store management page.

Test Steps:

1. The user clicks on the ‘grocery stores’ button on the side bar.
- **Expected Result:** The system displays information about the grocery stores that have been registered on the app.
- **Alternate Flow:** None.

Test Case 68: Disable Grocery Store Profile

Test Scenario: The system should allow the user to delete the profile of a grocery store.

Preconditions: The user is on the store management page.

Test Steps:

1. The user clicks on the delete button next to each grocery store.

Test Data:

1. Esajees
- **Expected Result:** The system should delete the grocery store and they should not be able to log in to their account on the application as well.
 - **Alternate Flow:** None.

Test Case 69: Search Grocery Store

Test Scenario: The system should allow the user to search for a grocery store.

Preconditions: The user is on the store management page.

Test Steps:

1. The system displays a search bar.
2. The user searches for a grocery store using their name.

Test Data:

1. Jalal Sons
- **Expected Result:** The system displays the results according to the search.
 - **Alternate Flow:** None.

Test Case 70: View Vendor Profile

Test Scenario: The system should allow the user to view the profile of a vendor.

Preconditions: The user is on the vendor management page.

Test Steps:

1. The user clicks on the ‘vendors’ button on the side bar.
- **Expected Result:** The system displays information about the vendors that have been registered on the app.
- **Alternate Flow:** None.

Test Case 71: Disable Vendor Profile

Test Scenario: The system should allow the user to delete the profile of a vendor.

Preconditions: The user is on the vendor management page.

Test Steps:

1. The user clicks on the delete button next to each vendor.

Test Data:

1. Nestle
- **Expected Result:** The system should delete the vendor and they should not be able to log in to their account on the application as well.
 - **Alternate Flow:** None.

Test Case 72: Search Vendor

Test Scenario: The system should allow the user to search for a vendor.

Preconditions: The user is on the vendor management page.

Test Steps:

1. The system displays a search bar.
2. The user searches for a vendor using their name.

Test Data:

1. PepsiCo
- **Expected Result:** The system displays the results according to the search.
 - **Alternate Flow:** None.

Test Case 73: Select Grocery Store’s ML Model

Test Scenario: The system should allow the user to select the ML model for the grocery store.

Preconditions: The user is on the store management page.

Test Steps:

1. The system displays an update option for the user to update the machine learning model of the grocery store.
2. The user can select any model from the options available.

Test Data:

1. LightGBM
 2. XGBoost
- **Expected Result:** The system displays the name of the ml model selected for the store.
 - **Alternate Flow:** None.

Test Case 74: Display Analytics

Test Scenario: The system should display the analytics to the user.

Preconditions: The user is on the analytics page.

Test Steps:

1. The user clicks on the ‘analytics’ button on the side bar.
- **Expected Result:** The system displays the analytics for the admin.
 - **Alternate Flow:** None.

Test Case 75: View Grocery Store Count

Test Scenario: The system should display a total count of the grocery stores that have registered.

Preconditions: The user is on the analytics page.

Test Steps:

1. The user clicks on the ‘analytics’ button on the side bar.
- **Expected Result:** The system displays the total number of grocery stores that are registered.
 - **Alternate Flow:** None.

Test Case 76: View Vendor Count

Test Scenario: The system should display a total count of the vendors that have registered.

Preconditions: The user is on the analytics page.

Test Steps:

1. The user clicks on the ‘analytics’ button on the side bar.
- **Expected Result:** The system displays the total number of vendors that are registered.
 - **Alternate Flow:** None.

Test Case 77: View Category Count

Test Scenario: The system should display a total count of the categories in the database.

Preconditions: The user is on the analytics page.

Test Steps:

1. The user clicks on the ‘analytics’ button on the side bar.
- **Expected Result:** The system displays the total number of categories.
- **Alternate Flow:** None.

Test Case 78: View Product Count

Test Scenario: The system should display a total count of the products in the database.

Preconditions: The user is on the analytics page.

Test Steps:

1. The user clicks on the ‘analytics’ button on the side bar.
- **Expected Result:** The system displays the total number of products.
- **Alternate Flow:** None.

Test Case 79: Add Category

Test Scenario: The system should allow the user to add a category.

Preconditions: The user is on the category management page.

Test Steps:

1. The user clicks on the ‘add category’ button.
 2. The system takes the user to the add category page.
 3. The user adds a name and an image for the category.
 4. The user clicks on the ‘add category’ button.
- **Expected Result:** The system displays the new category that has been added.
 - **Alternate Flow:** None.

Test Case 80: Update Category

Test Scenario: The system should allow the user to update a category.

Preconditions: The user is on the category management page.

Test Steps:

1. The user clicks on the ‘update’ button next to each category.
 2. The system takes the user to the update category page.
 3. The user updates either the name or the image for the category.
 4. The user clicks on the ‘update category’ button.
- **Expected Result:** The system displays the category that has been updated.

- **Alternate Flow:** None.

Test Case 81: Delete Category

Test Scenario: The system should allow the user to delete a category.

Preconditions: The user is on the category management page.

Test Steps:

1. The user clicks on the ‘delete’ button next to each category.
- **Expected Result:** The system deletes the category.
 - **Alternate Flow:** None.

Test Case 82: Search Category

Test Scenario: The system should allow the user to search for a category.

Preconditions: The user is on the category management page.

Test Steps:

1. The system displays a search bar.
 2. The user searches for a category using its name.
- **Expected Result:** The system displays the results according to the search.
 - **Alternate Flow:** None.

Test Case 83: Add Product

Test Scenario: The system should allow the user to add a product.

Preconditions: The user is on the product management page.

Test Steps:

1. The user clicks on the ‘add product’ button.
 2. The system takes the user to the add product page.
 3. The user adds a name and an image for the product.
 4. The user clicks on the ‘add product’ button.
- **Expected Result:** The system displays the new product that has been added.
 - **Alternate Flow:** None.

Test Case 84: Update Product

Test Scenario: The system should allow the user to update a product.

Preconditions: The user is on the product management page.

Test Steps:

1. The user clicks on the 'update' button next to each product.
 2. The system takes the user to the update product page.
 3. The user updates either the name or the image for the product.
 4. The user clicks on the 'update product' button.
- **Expected Result:** The system displays the product that has been updated.
 - **Alternate Flow:** None.

Test Case 85: Delete Product

Test Scenario: The system should allow the user to delete a product.

Preconditions: The user is on the product management page.

Test Steps:

1. The user clicks on the 'delete' button next to each product.
- **Expected Result:** The system deletes the product.
 - **Alternate Flow:** None.

Test Case 86: Search Product

Test Scenario: The system should allow the user to search for a product.

Preconditions: The user is on the product management page.

Test Steps:

3. The system displays a search bar.
 4. The user searches for a product using its name.
- **Expected Result:** The system displays the results according to the search.
 - **Alternate Flow:** None.

4.1.4 Test Case Grid

4.1.4.1 Vendor App

Test Case	Description	Status
1	Language Selection	✓
2	Phone Registration	✓
3	Phone Number Confirmation	✓
4	Phone Number Exists	✓
5	OTP Code Generation and Delivery	x
6	Login	✓
7	Logout	✓
8	Reset Password	✓
9	View Profile	✓
10	Vendor Registration	✓
11	Valid Password	✓
12	Username Exists	✓
13	Search Product in Inventory	✓
14	Add Product to Inventory	✓
15	All Product Search	✓
16	Restrict Product Duplication	✓
17	Remove Product	✓
18	Edit Details of Product	✓
19	View Inventory	✓
20	View Current Orders	✓
21	View Grocery Store List	✓

22	View Grocery Store Profile	✓
23	View Grocery Store Current Order	✓
24	View Order History	✓
25	Order Dispatch Tracking	✓
26	Edit Profile	✓

4.1.4.2 Grocery Store App

Test Case	Description	Status
27	Language Selection	✓
28	Phone Registration	✓
29	Phone Number Confirmation	✓
30	Phone Number Exists	✓
31	OTP Code Generation and Delivery	x
32	Login	✓
33	Logout	✓
34	Reset Password	✓
35	View Profile	✓
36	Account Details	✓
37	Valid Password	✓
38	Username Exists	✓
39	Search Product	✓
40	Search Category	✓
41	Search Vendor	✓
42	Browse Category	✓
43	View Vendor Profile	✓
44	Add Vendor to Vendor List	✓
45	Contact Vendor	x
46	View Products on the Vendor's Profile	✓
47	View Searched Product	✓
48	Select Products	✓

49	Order Recommendation	✓
50	Quantity Selection	✓
51	Add to Cart	✓
52	View Cart	✓
53	Update Product Quantity in Cart	✓
54	Remove Product	✓
55	Clear Cart	✓
56	Order Placement	✓
57	View Order	✓
58	Order Tracking	✓
59	View Vendor List	✓
60	View Order History	✓
61	Edit Profile	✓

4.1.4.3 Admin Portal

Test Case	Description	Status
62	Login	✓
63	Add New User	✓
64	Add New User Details	✓
65	Edit User Profile	✓
66	Delete User Profile	✓
67	Search User	✓
68	View Grocery Store's Profile	✓
69	Disable Grocery Store's Profile	✓
70	Search Grocery Store	✓
71	View Vendor's Profile	✓
72	Disable Vendor's Profile	✓
73	Search Vendor	✓
74	Select Grocery Store's ML Model	✓
75	Display Analytics	✓
76	View Grocery Store Count	✓
77	View Vendor Count	✓
78	View Category Count	✓
79	View Product Count	✓
80	Add Category	✓
81	Update Category	✓
82	Delete Category	✓
83	Search Category	✓

84	Add Product	✓
85	Update Product	✓
86	Delete Product	✓
87	Search Product	✓

Note: ‘✓’ represents that the functional requirement has been implemented while ‘x’ represents that it has not been implemented.

Detailed Testing and Quality Assurance attached in the Appendix.

4.2 Machine Learning

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

4.2.1 Training and Testing

We have mainly worked on 2 datasets, the local pharmacy dataset and the Corporación Favorita dataset. For the local pharmacy dataset, we have implemented different variations of 4 machine learning algorithms: Random Forest, XGBoost, Prophet, and Recurrent Neural Networks (LSTM and GRU). For the Corporación Favorita dataset, we have implemented 6 machine learning algorithms: Random Forest, XGBoost, Prophet, Recurrent Neural Networks (DeepAR), LightGBM, and N-BEATS.

4.2.1.1 Local Pharmacy Dataset

Initially, we did not have a grocery store dataset, 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 as mentioned in Figure 4.2.1.1.

Pharmacy Dataset Update Glossary

File	Description
D1	Combines monthly pharmacy data from CSV files into a single DataFrame and saves it as <code>D1.csv</code>
D2	Reads <code>D1.csv</code> , performs analysis (correlation), drops irrelevant columns, and saves the refined data as <code>D2.csv</code>
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 (<code>looseqty</code>) at the end, yielding <code>D3.csv</code>
D4	Reads <code>D3.csv</code> , converts the 'date' column to datetime format, aggregates sales data based on 'date' and 'itemname,' and saves the combined data as <code>D4.csv</code>
D5	Reads <code>D4.csv</code> , filters the data for 'itemname' equal to 'PANADOL TAB,' and saves the filtered data as <code>D5.csv</code>

Figure 4.2.1.1 Pharmacy Dataset Update Glossary

4.2.1.1.1 Random Forest

We start our exploration with Random Forest (RF). We import the necessary libraries (Figure 4.2.1.1.1), read a CSV file (D4.csv) into a pandas dataframe, and inspect the first 5 rows of the dataframe (Figure Figure 4.2.1.1.1.2).

```

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 4.2.1.1.1 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 4.2.1.1.2 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 4.2.1.1.3). ‘looseqty’ (sales) is the label and the other 5 columns are the features (Figure 4.2.1.1.4).

```
df.shape
```

```
(213056, 6)
```

Figure 4.2.1.1.3 Shape of the dataframe

```

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

```

Figure 4.2.1.1.4 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 4.2.1.1.5).

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

```
X.shape
```

```
(213056, 7477)
```

Figure 4.2.1.1.1.5 Converting categorical attributes and inspecting the shape of the dataframe

The dataset is split into training and testing sets using an 80-20 split ratio (Figure Figure 4.2.1.1.1.6). 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 4.2.1.1.1.6 Splitting data into training set and testing set

Finally, a RF Regressor model is created and trained using the training set (Figure 4.2.1.1.1.7). 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 4.2.1.1.1.7 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 4.2.1.1.1.8).

The screenshot shows a Google Colab interface with several code cells. The first cell contains:

```
[ ] df.shape
```

The output is:

```
{x} (213056, 6)
```

The second cell contains:

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

The third cell contains:

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

The fourth cell contains:

```
[ ] X.shape
```

The output is:

```
(213056, 7477)
```

The fifth cell contains:

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

The sixth cell contains:

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

The seventh cell contains:

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

The eighth cell contains:

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

A modal dialog box is displayed, stating: "Your session crashed after using all available RAM. If you are interested in access to high-RAM runtimes, you may want to check out [Colab Pro](#).", with a "View runtime logs" link and a close button.

At the bottom, it says "Connected to Python 3 Google Compute Engine backend".

Figure 4.2.1.1.8 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 4.2.1.1.9). This new dataset has 337 rows and the same 6 columns (Figure 4.2.1.1.10).

	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 4.2.1.1.9 D5.csv used instead of D4.csv

```
df.shape  
  
(337, 6)
```

Figure 4.2.1.1.10 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 4.2.1.1.11). 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 (Figure 4.2.1.1.12).

```
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 4.2.1.1.11 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 4.2.1.1.12 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 4.2.1.1.13).

```
y_test.shape  
  
(68,)  
  
y_test = np.array(y_test)
```

Figure 4.2.1.1.13 Converting `y_test` to a normal np array

Upon the graph being plotted, we realised how severely under fitted the model was (Figure 4.2.1.1.14). 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.

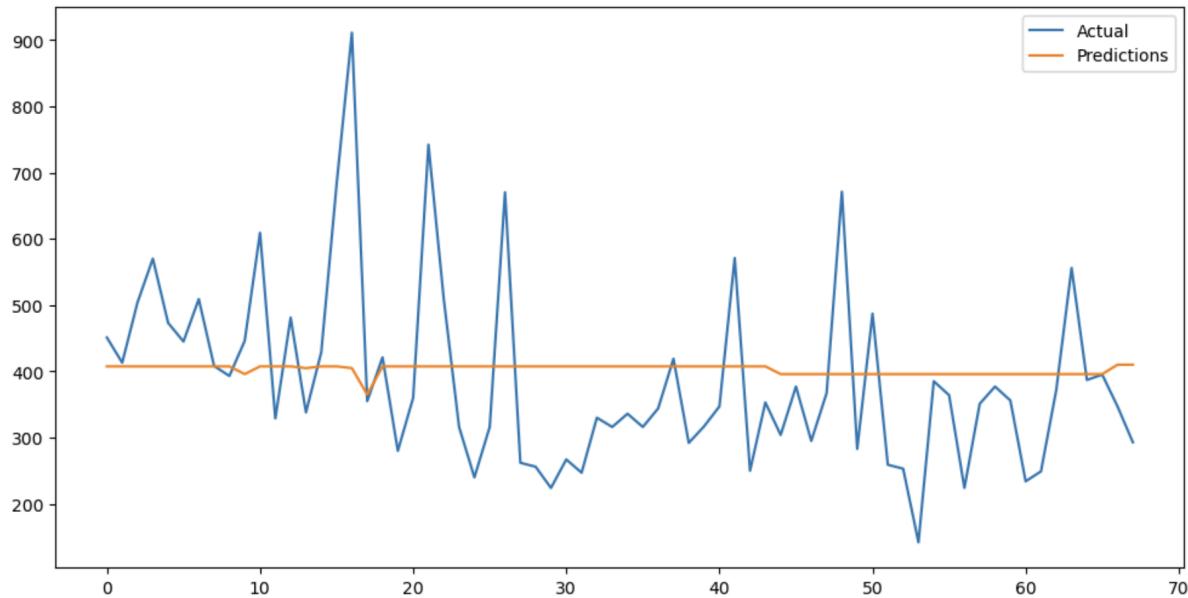


Figure 4.2.1.1.1.14 Actual values vs predicted values

4.2.1.1.2 XGBoost

Next, we experimented with a XGBoost model (Figure 4.2.1.1.2.1). 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 4.2.1.1.2.1 Training a XGBoost model

The model produced a RMSE of 145 and RMSLE of 0.371 (Figure 4.2.1.1.2.2) which are also reasonable on paper, however, we wanted to make sure that the model was not underfitting like the RF model.

```
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 4.2.1.1.2.2 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 4.2.1.1.2.3, is also underfitting.

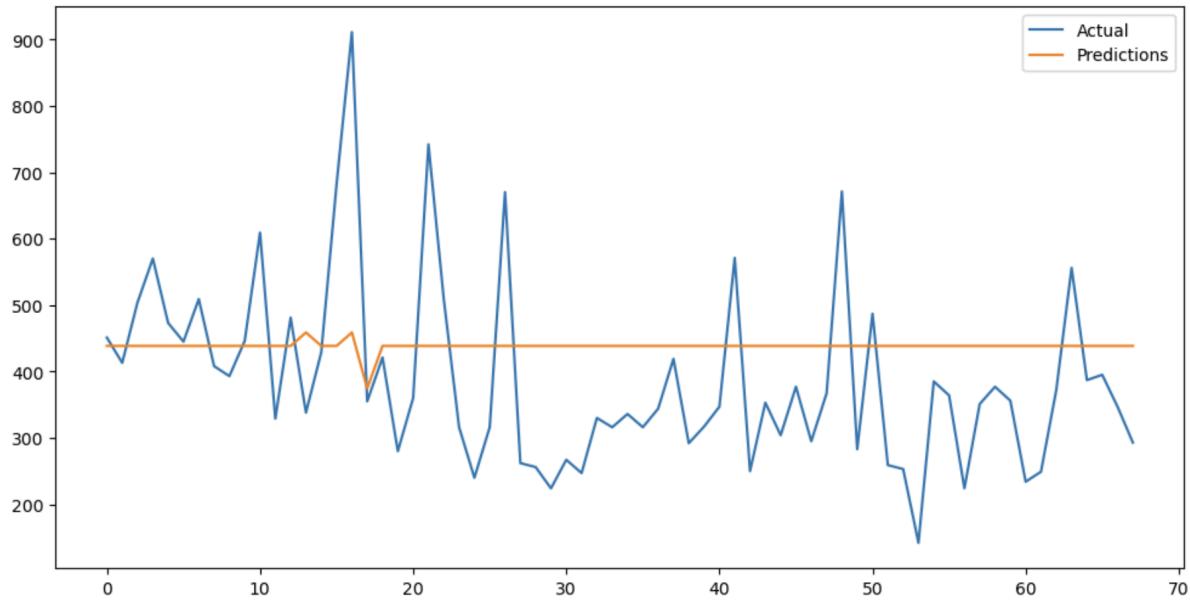


Figure 4.2.1.1.2.3 Actual values vs predicted values

4.2.1.1.3 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 4.2.1.1.3.1). 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 4.2.1.1.3.1 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 4.2.1.1.3.2).

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

Figure 4.2.1.1.3.2 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 4.2.1.1.3.3).

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

Figure 4.2.1.1.3.3 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 4.2.1.1.3.4).

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

Figure 4.2.1.1.3.4 Defining time period for predictions

The model produces an RMSE of 160 and RMSLE of 0.401 (Figure 4.2.1.1.3.5). This is slightly 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 4.2.1.1.3.6).

```
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 4.2.1.1.3.5 Printing the RMSE and RMSLE values

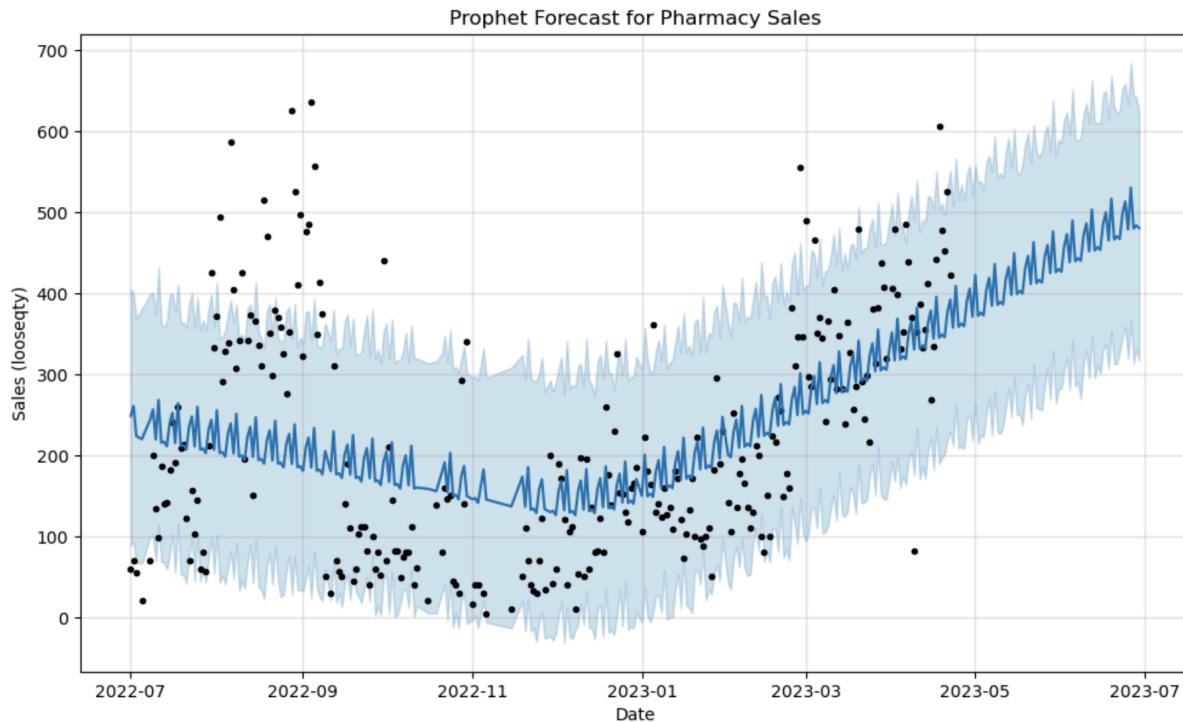


Figure 4.2.1.1.3.6 Prophet model's plot

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 4.2.1.1.3.7).

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

6053

Figure 4.2.1.1.3.7 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 4.2.1.1.3.8).

```

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 4.2.1.1.3.8 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 4.2.1.1.3.9). 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. 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 4.2.1.1.3.9 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 (Figure 4.2.1.1.3.10).

```
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 4.2.1.1.3.10 Printing the RMSE and RMSLE values

4.2.1.1.4 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 4.2.1.1.4.1).

```
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 4.2.1.1.4.1 Creating a univariate dataframe

Firstly, we create a function (Figure 4.2.1.1.4.2) 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 4.2.1.1.4.2 Generating sequences for training the model

We create a sequential model (Figure 4.2.1.1.4.3), 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 4.2.1.1.4.3 Creating and training the LSTM model

The model produced a RMSE of 139 and RMSLE of 0.330 (Figure 4.2.1.1.4.4) 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 4.2.1.1.4.5), 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 4.2.1.1.4.4 Printing the RMSE and RMSLE values

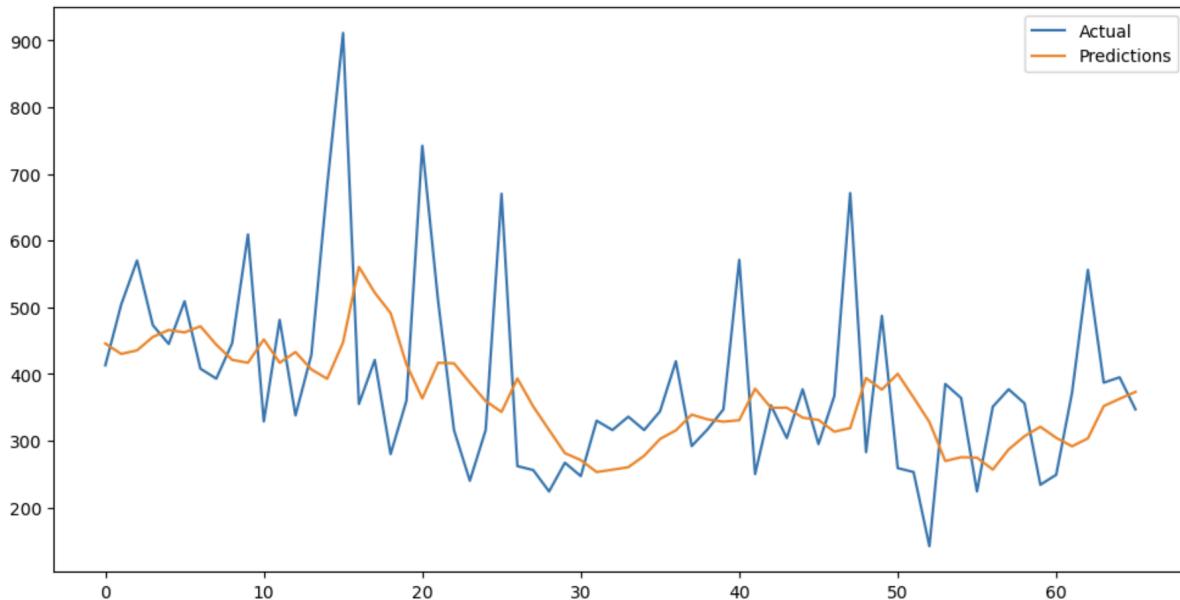


Figure 4.2.1.1.4.5 Actual values vs predicted values

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 4.2.1.1.4.6).

```

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 4.2.1.1.4.6 Creating and training the GRU model

The model produced a RMSE of 137 and RMSLE of 0.325 (Figure 4.2.1.1.4.7) which is actually slightly better than the LSTM model (we expected the performance to fall) and the plot (Figure 4.2.1.1.4.8) 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 4.2.1.1.4.7 Printing the RMSE and RMSLE values

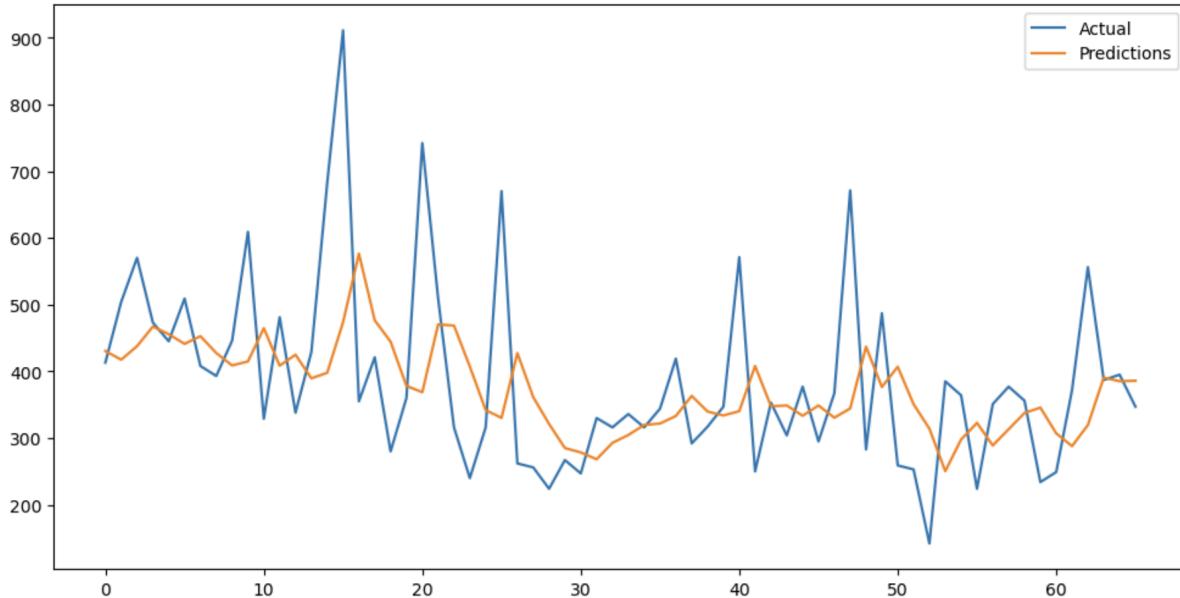


Figure 4.2.1.1.4.8 Actual values vs predicted values

4.2.1.1.5 Summary

Here is a side by side comparison (Table 4.2.1.1.5) of all the models trained and tested on the Pharmacy dataset (D5.csv) where RNN (GRU) showed the best performance whereas Prophet showed the worst.

Table 4.2.1.1.5 Model Summary

Model	RMSE	RMSLE
Random Forest	137	0.338
XGBoost	145	0.371
Prophet	160	0.401
RNN (LSTM)	139	0.330
RNN (GRU)	137	0.325

4.2.1.2 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 4.2.1.2.1.1) and Google Colab (Figure 4.2.1.2.1.2). As suggested by Ma'am Huda, we used batch training to overcome this issue.

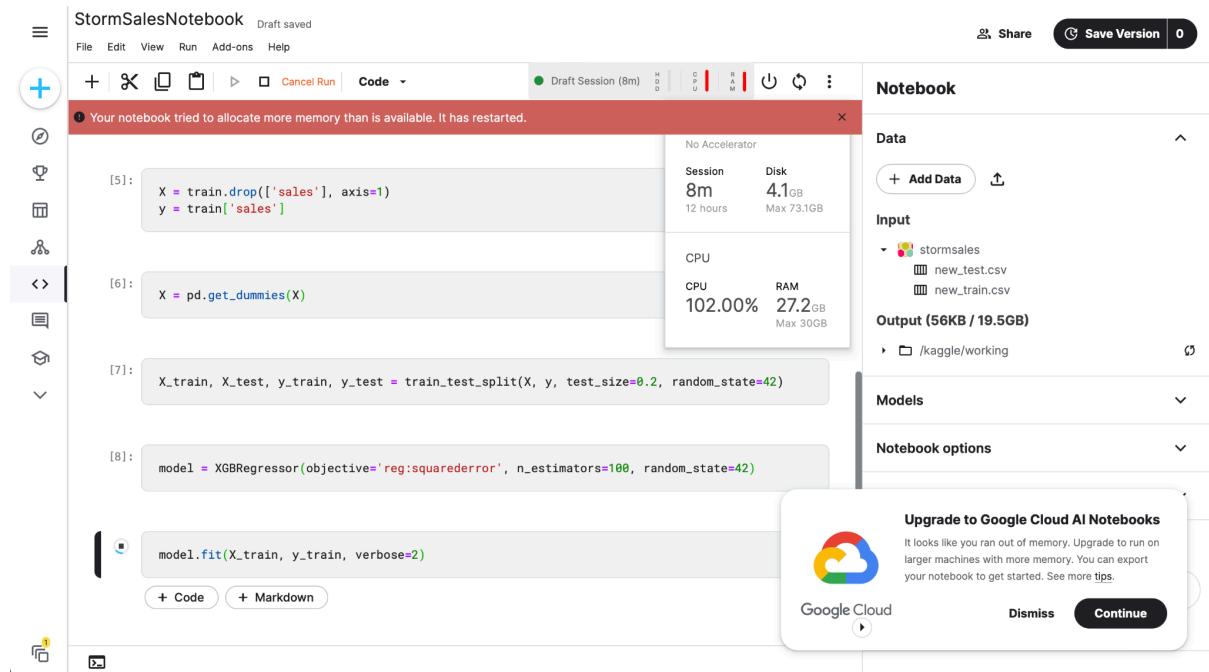


Figure 4.2.1.2.1.1 Kaggle crashing

The screenshot shows a Google Colab notebook titled 'M7.ipynb'. The code cell contains the following Python code:

```
[ ] train.head()
[ ] X = train.drop(['sales'], axis=1)
y = train['sales']
[ ] X = pd.get_dummies(X)
[ ] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
[ ] model = XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
[ ] model.fit(X_train, y_train)
```

A warning message at the bottom left states: "Your session crashed after using all available RAM. If you are interested in access to high-RAM runtimes, you may want to check out Colab Pro." A status bar at the bottom right indicates "Connected to Python 3 Google Compute Engine backend".

Figure 4.2.1.2.1.2 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 4.2.1.2.1.2. The feature engineering was done on the available datasets as mentioned in the Machine Learning Design 3.6.2.2. 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 4.2.1.2.1.3) 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 4.2.1.2.1.3 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 4.2.1.2.1.4).

```

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 4.2.1.2.1.4 Information about the updated train dataset

Considering the size of the dataset, batch training was used (Figure 4.2.1.2.1.5). 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 4.2.1.2.1.5 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 4.2.1.2.1.6). 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 4.2.1.2.1.6 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 4.2.1.2.1.7), producing an rmsle of 1.08120 on the unseen dataset and ranking 575th on the leaderboard.

Rank	User	Score	Entries	Days Ago
570	rajesh vashishtha	1.03039	2	1mo
571	ITESO (Oscar,Ricardo,Lili)	1.03067	1	15d
572	AngelXiang95	1.03680	2	1d
573	xinnHs	1.04788	4	1mo
574	hhoussam	1.07085	4	2mo
575	Irtaza Ahmed Khan	1.08120	1	2d
Your First Entry! Welcome to the leaderboard!				
576	kdekto	1.10311	1	2d
577	RAMA	1.11353	1	1mo
578	20231005	1.12556	8	2d
579	Pourea Ziasistani	1.13470	1	1mo
580	Jade0725	1.13811	5	5d

Figure 4.2.1.2.1.7 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.

4.2.1.2.1 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 4.2.1.2.2.1).

```
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 4.2.1.2.2.1 Loading the dataset and inspecting it

Pandas' get_dummies function is used to convert categorical attributes to numerical values (Figure 4.2.1.2.2.2). 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 4.2.1.2.2.2 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 4.2.1.2.2.3).

```
batch_size = 32

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

Figure 4.2.1.2.2.3 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 4.2.1.2.2.4). 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 4.2.1.2.2.4 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 4.2.1.2.2.4).

```

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 4.2.1.2.2.5 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 4.2.1.2.2.6) to find out the cause for such high RMSE and RMSLE. As seen in Figure 4.2.1.2.2.7, 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 4.2.1.2.2.6 Plotting 100 values from actual and predicted values

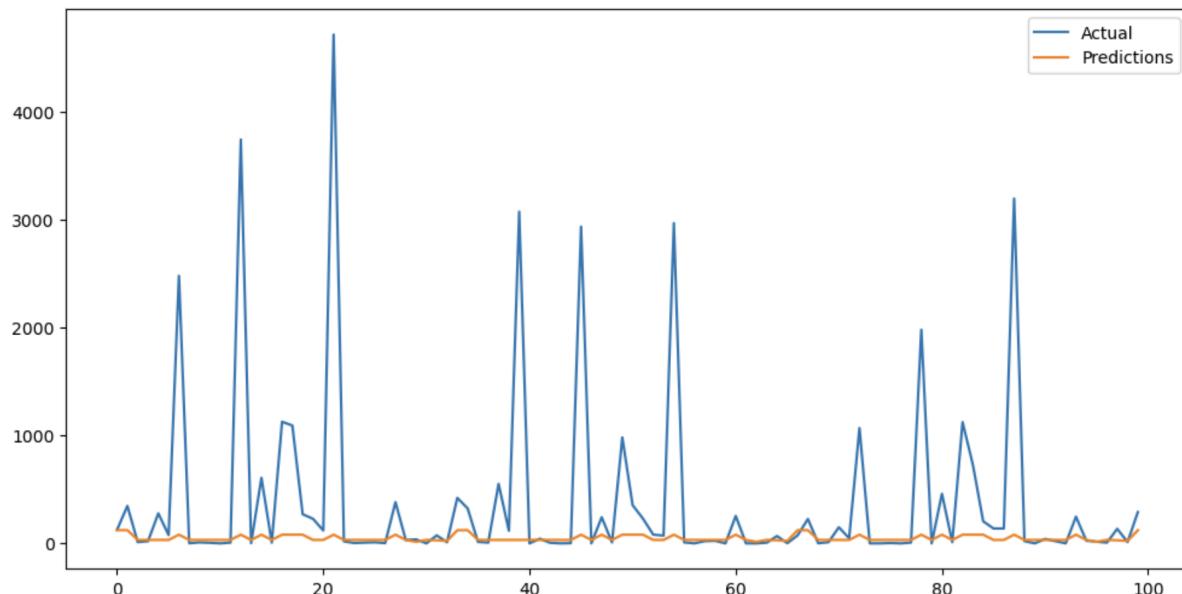


Figure 4.2.1.2.2.7 Actual values vs predicted values

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 4.2.1.2.2.8). With batch size set to 512, the model was trained on 4689 batches (Figure 4.2.1.2.2.9).

```
batch_size = 512
```

Figure 4.2.1.2.2.8 Batch size

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

Figure 4.2.1.2.2.9 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 (4.2.1.2.2.10). 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 4.2.1.2.2.11).

```
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 4.2.1.2.2.10 Printing the RMSE and RMSLE values

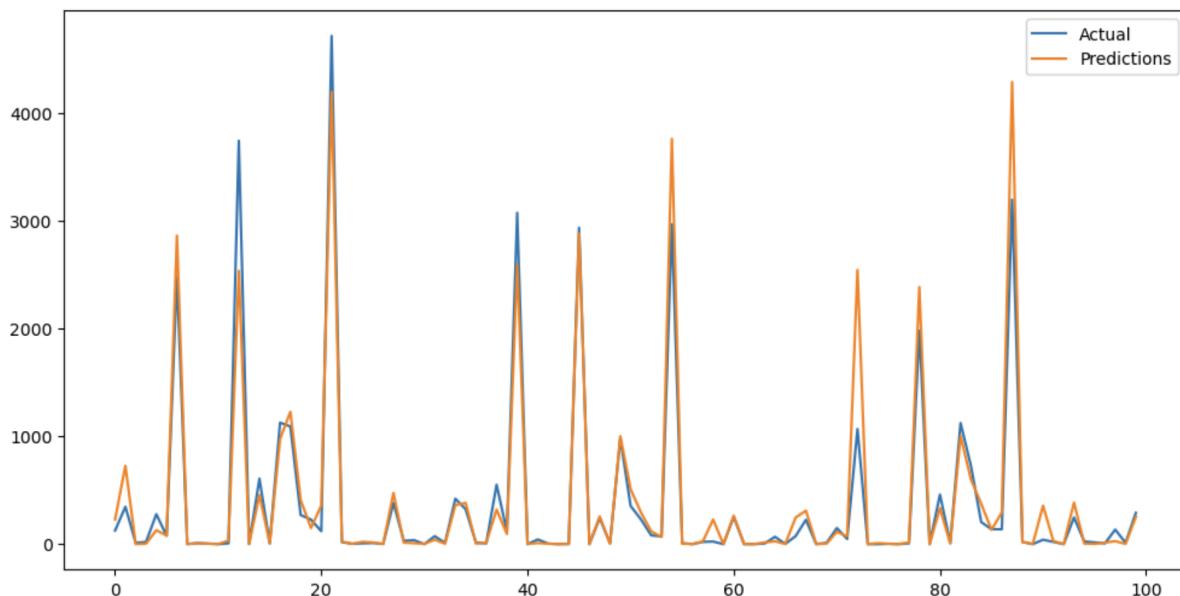


Figure 4.2.1.2.2.11 Actual values vs predicted values

Considering the improvement in results by increasing the batch size, we increased the batch size further (Figure 4.2.1.2.2.12). With a batch size of 1024, the model was trained on 2345 batches (Figure 4.2.1.2.2.13). However, there was no difference in the RMSE or RMSLE, as they stayed exactly the same at 975 and 1.618 respectively (Figure 4.2.1.2.2.14). The fit of the model as seen in Figure 4.2.1.2.2.15 is also identical.

```
batch_size = 1024
```

Figure 4.2.1.2.2.12 Batch size

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

Figure 4.2.1.2.2.13 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 4.2.1.2.2.14 Printing the RMSE and RMSLE values

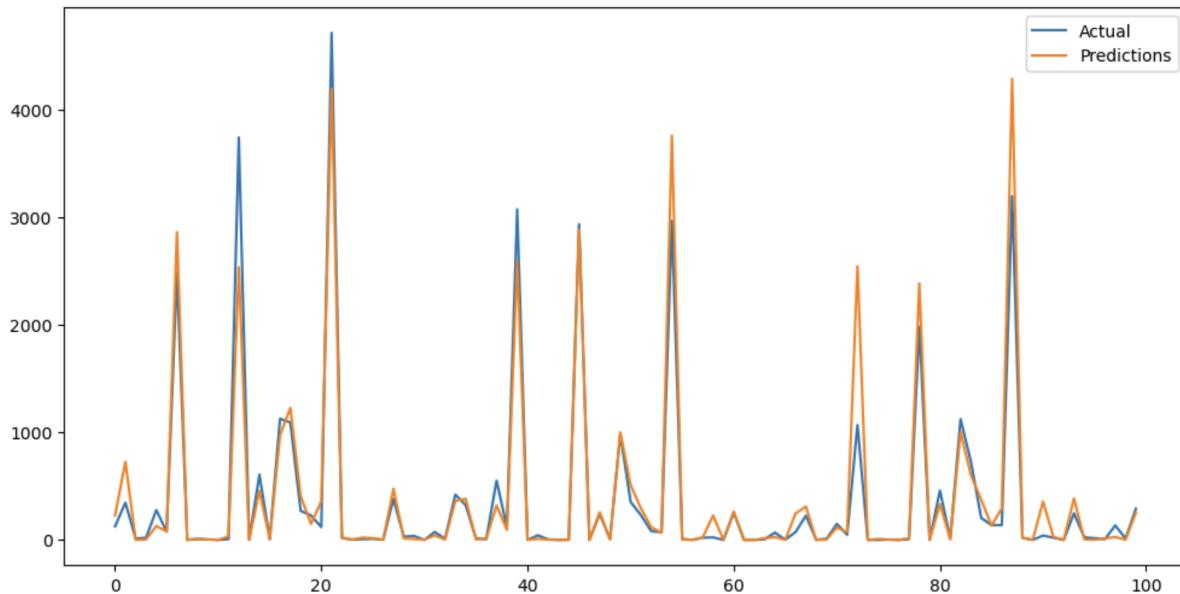


Figure 4.2.1.2.2.15 Actual values vs predicted values

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 4.2.1.2.2.16).

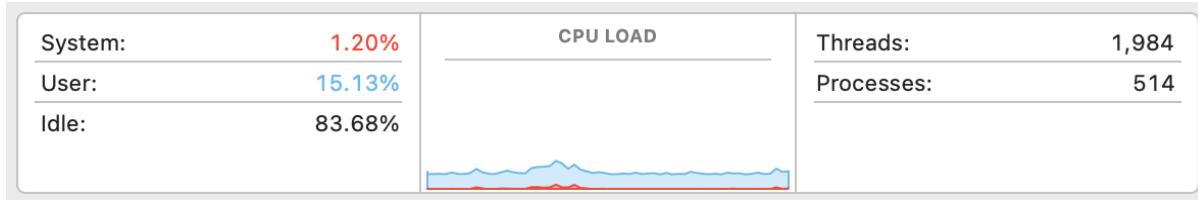


Figure 4.2.1.2.2.16 CPU usage by Random Forest

4.2.1.2.2 XGBoost

Next, we trained a XGBoost model (Figure 4.2.1.2.3.1) with the same configurations as the RF model, however, the kernel kept crashing (Figure 4.2.1.2.3.2).

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

Figure 4.2.1.2.3.1 Creating a XGBoost model



Figure 4.2.1.2.3.2 Kernel died error

Increasing the batch size to 512 solved the issue (Figure 4.2.1.2.3.3). 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 4.2.1.2.3.4). The model is not under-fitted and is able to generalise well on unseen as seen in Figure 4.2.1.2.3.5.

```
batch_size = 512
```

Figure 4.2.1.2.3.3 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 4.2.1.2.3.4 Printing the RMSE and RMSLE values

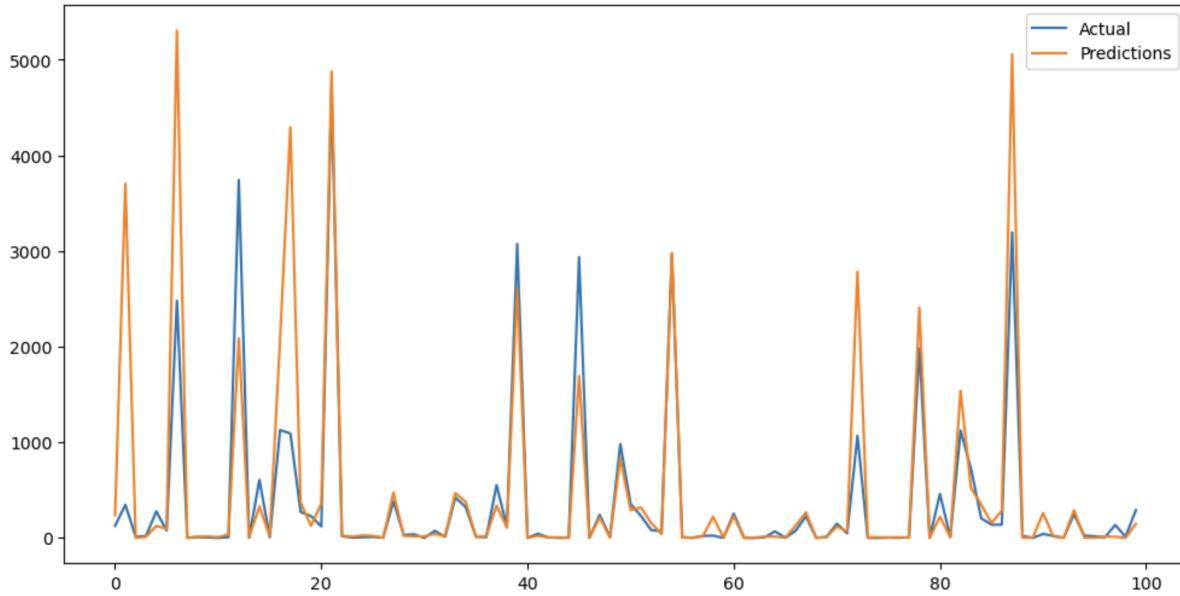


Figure 4.2.1.2.3.5 Actual values vs predicted values

The XGBoost trained significantly faster than the RF model. One of the reasons being that XGBoost utilised all the available resources while training (Figure 4.2.1.2.3.6). This was a key reason for XGBoost being used more than RF for experimentation during the project despite producing slightly worse results than RF.

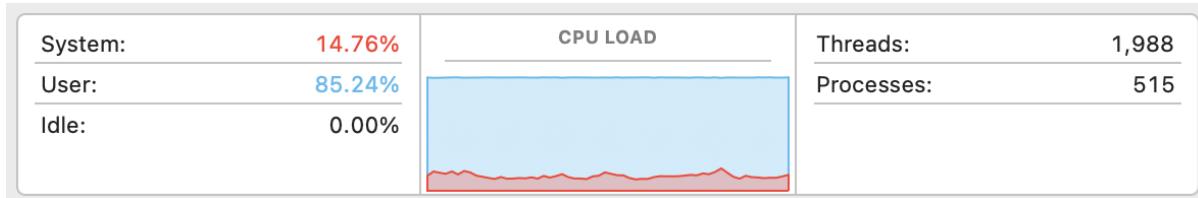


Figure 4.2.1.2.3.6 CPU usage by XGBoost

4.2.1.2.3 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 4.2.1.2.4.1). 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 4.2.1.2.4.1 Only sales for beverages (family = 3) at store number 1

The model produced a very low RMSE (653) and RMSLE (0.554) score (Figure 4.2.1.2.4.2), however, it cannot be compared with the evaluations earlier as we don't know if 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 4.2.1.2.4.3).

```
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 4.2.1.2.4.2 Printing the RMSE and RMSLE values

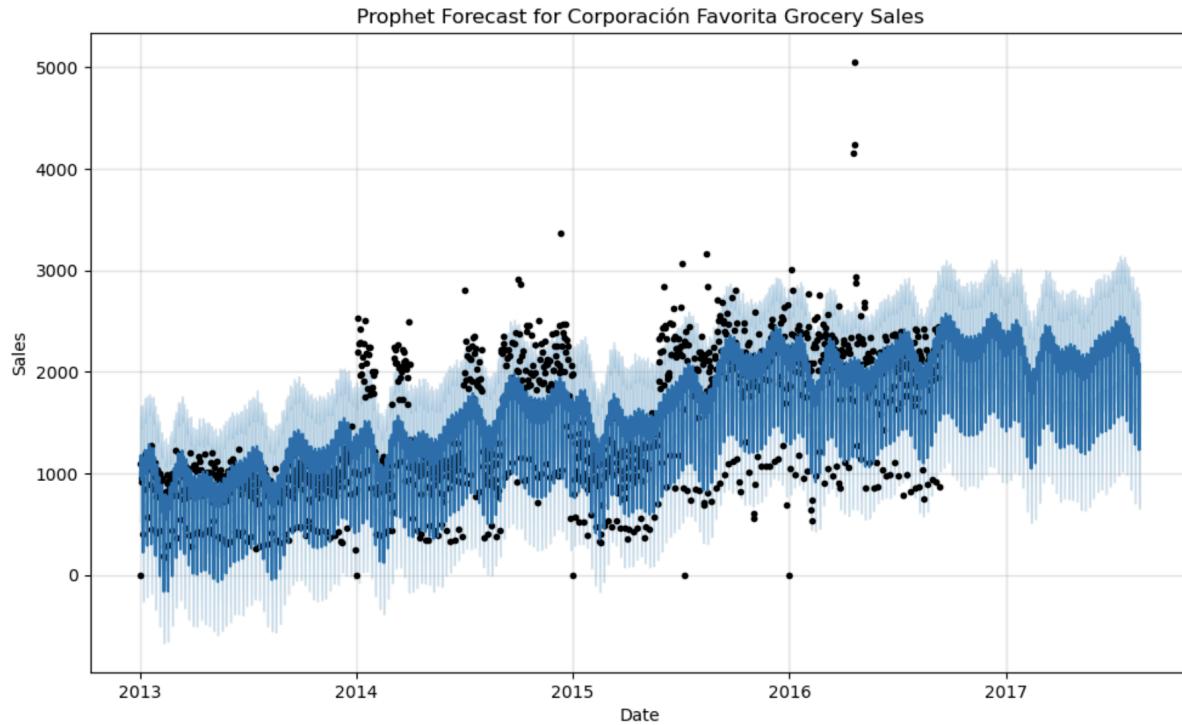


Figure 4.2.1.2.4.3 Prophet model's plot

4.2.1.2.4 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 4.2.1.2.5.1, 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 4.2.1.2.5.1 Importing libraries and loading the dataset

Firstly, we create a function (Figure 4.2.1.2.5.2) 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 4.2.1.2.5.2 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 4.2.1.2.5.3, 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 4.2.1.2.5.3 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 4.2.1.2.5.4), 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

```



```

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

```

Figure 4.2.1.2.5.4 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 4.2.1.2.5.5). 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 4.2.1.2.5.6).

```

: cp1 = ModelCheckpoint('modell/', save_best_only=True)

: modell.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 the legacy Keras optimizer instead, located at `tf.keras.optimizers.legacy.Adam`.

: modell.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:tensorflow:Assets written to: modell/assets
INFO:tensorflow:Assets written to: modell/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 4.2.1.2.5.5 Compiling and fitting the model

```

Epoch 10/10
2398/2401 [=====>.] - ETA: 0s - loss: 146996.0312 - root_mean_squared_error: 383.4006INFO:tensorflow:Assets written to: modell/assets
INFO:tensorflow:Assets written to: modell/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 4.2.1.2.5.6 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 4.2.1.2.5.7, 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>]

```

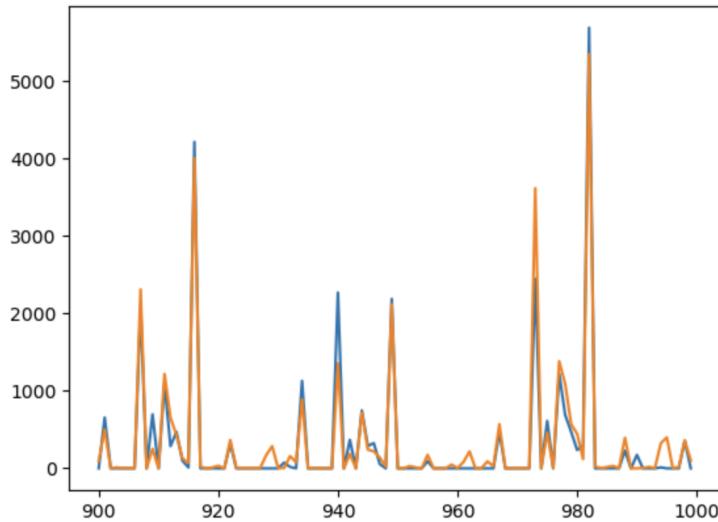


Figure 4.2.1.2.5.7 Test predictions vs actual values plotted

Shown in Figure 4.2.1.2.5.8, 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 4.2.1.2.5.8 GRU (Univariate)

Finally, the third model (Figure 4.2.1.2.5.9) 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 4.2.1.2.5.9 LSTM (Multivariate)

4.2.1.2.5 LightGBM

Next, we implemented a LightGBM model. We read the processed csv file (new_train.csv) into a pandas dataframe, and inspected the first 5 rows of the dataframe (Figure 4.2.1.2.6.1).

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

```
df.head()
```

	store_nbr	family	sales	onpromotion	typeholiday	dcoilwtico	city	state	typestores	cluster	day_of_week	day
0	1	AUTOMOTIVE	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1
1	1	BABY CARE	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1
2	1	BEAUTY	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1
3	1	BEVERAGES	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1
4	1	BOOKS	0.0	0	Holiday	93.14	Quito	Pichincha	D	13	2	1

Figure 4.2.1.2.6.1 Loading the dataset and inspecting it

The dataframe is converted into training features (X) and labels (y). Then, the training features are inspected for categorical attributes (Figure 4.2.1.2.6.2).

```

x = df.drop(['sales'], axis=1)
y = df['sales']

X.info()

```

#	Column	Dtype
0	store_nbr	int64
1	family	object
2	onpromotion	int64
3	typeholiday	object
4	dcoilwtico	float64
5	city	object
6	state	object
7	typestores	object
8	cluster	int64
9	day_of_week	int64
10	day	int64
11	month	int64
12	year	int64

dtypes: float64(1), int64(7), object(5)
memory usage: 297.6+ MB

Figure 4.2.1.2.6.2 Inspecting training features (X) for categorical attributes

Label Encoder is used to convert categorical attributes to numerical values (Figure 4.2.1.2.6.3). One of the biggest reasons for using Label Encoder instead of One-Hot Encoder (or Pandas' get_dummies function) is to reduce the computational requirements, as label encoding does not increase the dimensionality of the data as one-hot encoding does.

```

family_encoder = LabelEncoder()
typeholiday_encoder = LabelEncoder()
city_encoder = LabelEncoder()
state_encoder = LabelEncoder()
typestores_encoder = LabelEncoder()

X['family'] = family_encoder.fit_transform(X['family'])
X['typeholiday'] = typeholiday_encoder.fit_transform(X['typeholiday'])
X['city'] = city_encoder.fit_transform(X['city'])
X['state'] = state_encoder.fit_transform(X['state'])
X['typestores'] = typestores_encoder.fit_transform(X['typestores'])

```

Figure 4.2.1.2.6.3 Converting categorical attributes to numerical values

The first 5 rows of the dataframe are inspected to ensure the conversion took place as expected. As it can be seen in Figure 4.2.1.2.6.4, family, typeholiday, city, state, and typestores are converted to numerical values.

```
X.head()
```

	store_nbr	family	onpromotion	typeholiday	dcoilwtico	city	state	typestores	cluster	day_of_week	day	month	year
0	1	0	0	3	93.14	18	12	3	13	2	1	1	2013
1	1	1	0	3	93.14	18	12	3	13	2	1	1	2013
2	1	2	0	3	93.14	18	12	3	13	2	1	1	2013
3	1	3	0	3	93.14	18	12	3	13	2	1	1	2013
4	1	4	0	3	93.14	18	12	3	13	2	1	1	2013

Figure 4.2.1.2.6.4 Inspecting the updated dataset

The dataset is split into training and testing sets using an 80-20 split ratio (Figure 4.2.1.2.6.5). 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. The batch size is set to 1024 as during FYP I, we noticed that a model with a larger batch size (512 or more) did not only train faster but also avoided underfitting. As the performance (RMSE) for a batch size of 512 and 1024 stayed exactly the same, we chose 1024 as it trains faster.

```
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:]

batch_size = 1024

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

Figure 4.2.1.2.6.5 Splitting data into training set and testing set and setting batch size

Considering the size of the dataset, batch training was used. The dataset was divided in batches of 1024 samples. The total number of batches were calculated to cover the entire training dataset (which in this case were 2345). A LGBMRegressor model is instantiated and trained one batch at a time (Figure 4.2.1.2.6.6).

```

model = LGBMRegressor(objective='regression', 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")

```

Figure 4.2.1.2.6.6 Batch training LGBMRegressor

The predictions were generated on the LGBMRegressor model. All the slightly negative predictions were converted to zero to ensure calculating RMSLE is possible (Figure 4.2.1.2.6.7).

```

[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
Batch 2345/2345 completed

y_pred = model.predict(X_test)

y_pred = np.maximum(y_pred, 0)

```

Figure 4.2.1.2.6.7 Generating predictions and fixing them

The model produced a RMSE of 1085 and RMSLE of 2.441 as shown in Figure 4.2.1.2.6.8.

```

rmse = np.sqrt(mean_squared_error(y_test, y_pred))
rmsle = np.sqrt(mean_squared_error(np.log1p(y_test), np.log1p(y_pred)))

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

Root Mean Squared Error (RMSE): 1084.811390208004
Root Mean Squared Logarithmic Error (RMSLE): 2.4413069752162904

```

Figure 4.2.1.2.6.8 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 4.2.1.2.6.9). The model is not under-fitted which seemed to be a problem in some of the earlier models.

```

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()

```

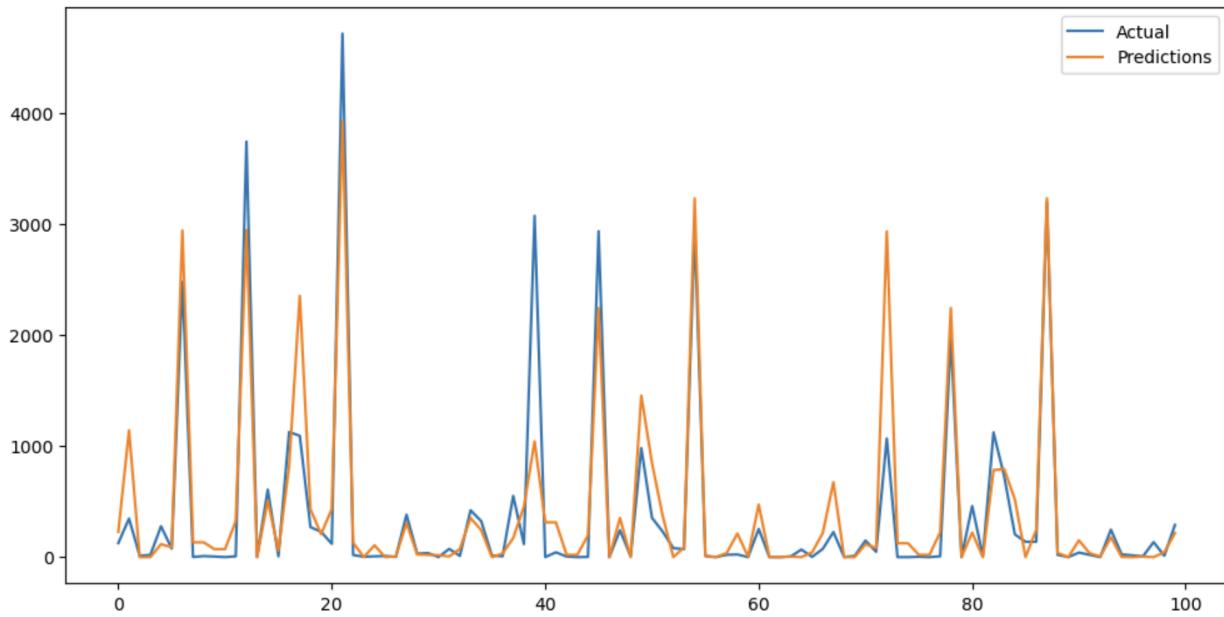


Figure 4.2.1.2.6.9 Actual values vs predicted values

4.2.1.2.6 Summary

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

Table 4.2.1.2.6 Model Summary

Model	RMSE	RMSLE
Random Forest	975	1.618
XGBoost	1119	1.462
Prophet*	653	0.554
LSTM	566	-
GRU	1071	-
LightGBM	1084	2.441

Prophet was only trained on a subset of the dataset (Figure 4.2.1.2.4.1).

4.2.1.3 Darts

Next, we used Darts (Figure 4.2.1.3.1.1) for creating our forecasting models. Darts has a variety of models available, from classic statistical models such as ARIMA to state of the art deep neural network models such as TFT.



Figure 4.2.1.3.1.1 Darts

Darts offers all the forecasting models we implemented (Random Forest, XGBoost, Prophet, and Recurrent Neural Networks) and all the forecasting models we plan on implementing (DeepAR, LightGBM, and N-BEATS). Figure 4.2.1.3.1.2 shows a breakdown of these models implemented in Darts.

Model	Sources	Target Series Support:	Covariates Support:	Probabilistic Forecasting:	Training & Forecasting on Multiple Series
		Univariate/ Multivariate	Past-observed/ Future-known/ Static	Sampled/ Distribution Parameters	
RandomForest		■ ■	■ ■ ■	■ ■	■
XGBModel		■ ■	■ ■ ■	■ ■	■
Prophet	Prophet repo	■ ■	■ ■ ■	■ ■	■
RNNModel (incl. LSTM and GRU); equivalent to DeepAR in its probabilistic version	DeepAR paper	■ ■	■ ■ ■	■ ■	■
LightGBMModel		■ ■	■ ■ ■	■ ■	■
NBEATSMModel	N-BEATS paper	■ ■	■ ■ ■	■ ■	■
TFTModel (Temporal Fusion Transformer)	TFT paper, PyTorch Forecasting	■ ■	■ ■ ■	■ ■	■

Figure 4.2.1.3.1.2 Forecasting Models

To get started with Darts, we tried to implement a Random Forest model. The implementation of Random Forest in Darts is wrapped around the Random Forest Regressor from Scikit-Learn which we had implemented, so it seemed like a suitable choice to start. Figure 4.2.1.3.1.3 shows an example of a Random Forest model implemented using Darts. Right away, you notice several new things compared to Scikit-Learn. TimeSeries are used to store the data instead of DataFrames or Lists. New parameters are

introduced such as lags, future covariates, past covariates etc. The predictions are not generated for an input of the data but a range of time steps after the end of the series.

```
>>> from darts.datasets import WeatherDataset
>>> from darts.models import RandomForest
>>> series = WeatherDataset().load()
>>> # predicting atmospheric pressure
>>> target = series['p (mbar)'][:100]
>>> # optionally, use past observed rainfall (pretending to be unknown beyond index 100)
>>> past_cov = series['rain (mm)'][:100]
>>> # optionally, use future temperatures (pretending this component is a forecast)
>>> future_cov = series['T (degC)'][:106]
>>> # random forest with 200 trees trained with MAE
>>> model = RandomForest(
>>>     lags=12,
>>>     lags_past_covariates=12,
>>>     lags_future_covariates=[0,1,2,3,4,5],
>>>     output_chunk_length=6,
>>>     n_estimators=200,
>>>     criterion="absolute_error",
>>> )
>>> model.fit(target, past_covariates=past_cov, future_covariates=future_cov)
>>> pred = model.predict(6)
>>> pred.values()
array([[1006.29805],
       [1006.23675],
       [1006.17325],
       [1006.10295],
       [1006.06505],
       [1006.05465]])
```

Figure 4.2.1.3.1.3 [Random Forest Example](#)

4.2.1.3.1 Random Forest

To get started with Darts, we created a new csv file (darts.csv) which contains data only for store_nbr = 1 and family = 3 (Beverages). We read this new csv file into a pandas dataframe and inspected the first 5 rows of the dataframe (Figure 4.2.1.3.2.1).

```
df = pd.read_csv("../Data/Kaggle/StoreSales/darts.csv", parse_dates=[ "date" ])
```

```
df.head()
```

	date	store_nbr	family	sales
0	2013-01-01	1	3	0.0
1	2013-01-02	1	3	1091.0
2	2013-01-03	1	3	919.0
3	2013-01-04	1	3	953.0
4	2013-01-05	1	3	1160.0

Figure 4.2.1.3.2.1 Loading the dataset and inspecting it

TimeSeries is the main class in Darts. We created a TimeSeries from the pandas dataframe using the function [from_dataframe](#) by Darts (Figure 4.2.1.3.2.2). The dataset is then split into training and validation sets using an 80-20 split ratio. As this is a time series forecasting problem, the first 80% is used for training and the last 20% is used for validation, instead of using a shuffled dataset.

```
ts = TimeSeries.from_dataframe(df, value_cols=[ "sales" ])
```

```
split_ratio = 0.8
training_size = int(len(ts) * split_ratio)
```

```
train = ts[:training_size]
val = ts[training_size:]
```

Figure 4.2.1.3.2.2 Creating a TimeSeries and splitting data into training set and validation set

A Random Forest model is instantiated. The model is passed two parameters: lags = 1 and output chunk length = 7 (Figure 4.2.1.3.2.3). Lags are the lagged target series values used to predict the next time step(s). Lags was passed as a parameter because the model gave a value error (Figure 4.2.1.3.2.4) upon instantiation without it.

```
model = RandomForest(lags=1, output_chunk_length=7)
model.fit(train)
```

```
RandomForest(lags=1, lags_past_covariates=None, lags_future_covariates=None, output_chunk_length=7, add_encoders=None, n_estimators=100, max_depth=None, multi_models=True, use_static_covariates=True)
```

Figure 4.2.1.3.2.3 Instantiating and training the Random Forest model

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

```
ValueError: At least one of `lags`, `lags_future_covariates` or `lags_past_covariates` must be not None.
```

Figure 4.2.1.3.2.4 Random Forest producing a ValueError when instantiated without lags

The predictions were generated on the Random Forest model for the length of the validation set, the number of time steps (days in our case) in the validation set. Both the validation series and the prediction series are converted to lists to make the plotting more convenient. All the slightly negative predictions were converted to zero to ensure calculating RMSLE is possible (Figure 4.2.1.3.2.5).

```
predictions = model.predict(n=len(val))

val = val.pd_series().tolist()
predictions = predictions.pd_series().tolist()

predictions = np.maximum(predictions, 0)
```

Figure 4.2.1.3.2.5 Generating predictions, converting series to lists, and fixing them

The model produced a RMSE of 768 and RMSLE of 0.604 as shown in Figure 4.2.1.3.2.6.

```
rmse = np.sqrt(mean_squared_error(val, predictions))
rmsle = np.sqrt(mean_squared_error(np.log1p(val), np.log1p(predictions)))

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

Root Mean Squared Error (RMSE): 768.4775402558267
Root Mean Squared Logarithmic Error (RMSLE): 0.6038746114762992
```

Figure 4.2.1.3.2.6 Printing the RMSE and RMSLE values

The graph was plotted for actual and predicted values of the validation set (Figure 4.2.1.3.2.7). The model is not under-fitted which seemed to be a problem in some of the earlier models.

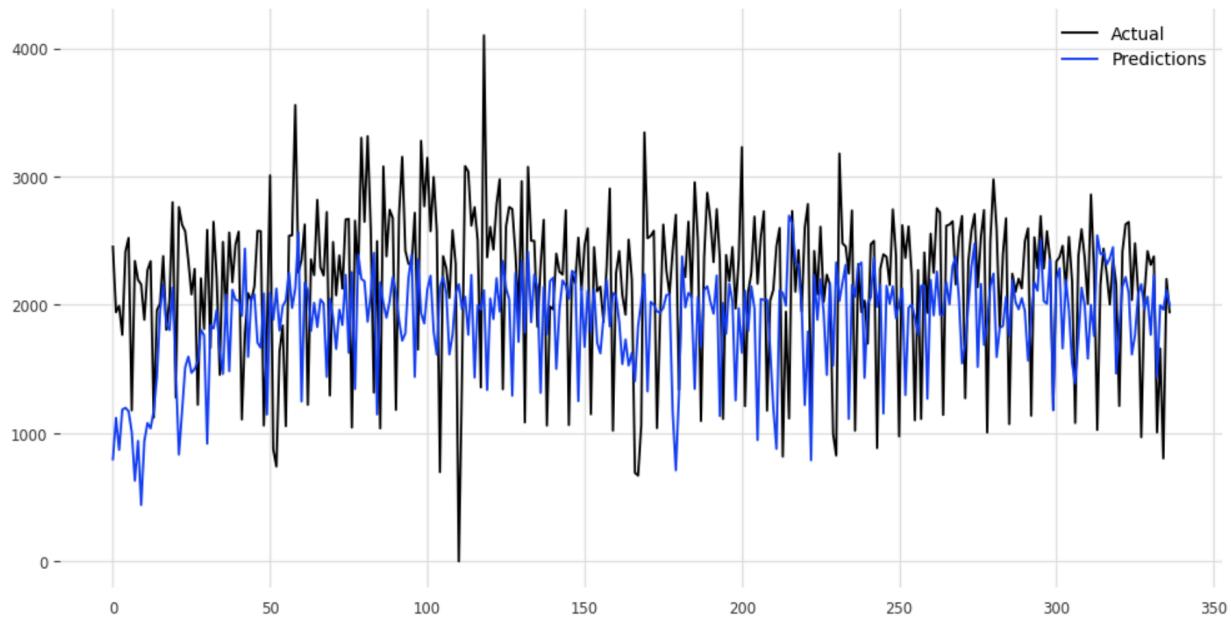


Figure 4.2.1.3.2.7 Actual values vs predicted values

Then, we created a new csv file (train_darts.csv) which contained the date column. The date column was converted into day_of_week, day, month, and year columns in earlier models. As we are training univariate time series forecasting models using Darts, the date column in the date format is necessary. We read this new csv file into a pandas dataframe and inspected the first 5 rows of the dataframe (Figure 4.2.1.3.2.8).

```
df = pd.read_csv("../Data/Kaggle/StoreSales/train_darts.csv", parse_dates=[ "date" ])
```

	date	store_nbr	family	sales	onpromotion	typeholiday	dcoilwtico	city	state	typestores	cluster	day_of_week	day	month
0	2013-01-01	1	0	0.0	0	3	93.14	18	12	3	13	2	1	1
1	2013-01-01	1	1	0.0	0	3	93.14	18	12	3	13	2	1	1
2	2013-01-01	1	2	0.0	0	3	93.14	18	12	3	13	2	1	1
3	2013-01-01	1	3	0.0	0	3	93.14	18	12	3	13	2	1	1
4	2013-01-01	1	4	0.0	0	3	93.14	18	12	3	13	2	1	1

Figure 4.2.1.3.2.8 Loading the dataset and inspecting it

Earlier, we trained multiseries models where one model was trained on the entire dataset (all stores and families). This time, we trained a separate model for each unique combination of store and family (product category). The Store Sales dataset contains data from 54 stores about 33 families. Meaning, we

trained 1782 Random Forest models. To allow the calculation of the average RMSE/RMSLE and save the trained models, the following variables were declared (Figure 4.2.1.3.2.9).

```
unique_combinations = df[['store_nbr', 'family']].drop_duplicates()

total_rmse = 0
total_rmsle = 0
num_models = 0

trained_models = {}
```

Figure 4.2.1.3.2.9 Getting unique combinations of stores and families and declaring variables

A for loop is used to iterate over each unique combination of store and family in the dataset. A filtered dataframe (df_subset) is created with only the data from one store and family (Figure 4.2.1.3.2.10). For example, in the first iteration, only the data for store_nbr = 1 and family = 0.

```
for index, row in unique_combinations.iterrows():
    store_nbr = row['store_nbr']
    family = row['family']

    print(f"\nTraining model for store {store_nbr} and family {family}...")

    df_subset = df[(df['store_nbr'] == store_nbr) & (df['family'] == family)]
```

Figure 4.2.1.3.2.10 Iterating over each combination of store and family and creating a subset of data

Then, the filtered dataset is converted into a TimeSeries using only the sales column (as it is a univariate model). The dataset is then split into training and validation sets using an 80-20 split ratio (Figure 4.2.1.3.2.11).

```
ts = TimeSeries.from_dataframe(df_subset, value_cols=["sales"])

split_ratio = 0.8
training_size = int(len(ts) * split_ratio)
train = ts[:training_size]
val = ts[training_size:]
```

Figure 4.2.1.3.2.11 Creating a TimeSeries and splitting data into training set and validation set

A Random Forest model is initialised with lag parameter set to 2 and fitted on the training data. The trained model is saved into the trained_models dictionary with a tuple (store_nbr, family) as the key. This tuple serves as an identifier for retrieving the corresponding model later. So, when we need to access a specific model later, we can retrieve it from the trained_models dictionary using the corresponding (store_nbr, family) tuple as the key (Figure 4.2.1.3.2.12).

```

model = RandomForest(lags=2)
model.fit(train)

trained_models[(store_nbr, family)] = model

```

Figure 4.2.1.3.2.12 Instantiating, training, and saving the Random Forest model(s)

The predictions were generated on the Random Forest model for the length of the validation set, the number of time steps (days in our case) in the validation set. Both the validation series and the prediction series are converted to lists to make the plotting more convenient. All the slightly negative predictions were converted to zero to ensure calculating RMSLE is possible (Figure 4.2.1.3.2.13).

```

predictions = model.predict(n=len(val))

val = val.pd_series().tolist()
predictions = predictions.pd_series().tolist()

predictions = np.maximum(predictions, 0)

```

Figure 4.2.1.3.2.13 Generating predictions, converting series to lists, and fixing them

RMSE and RMSLE are calculated between the validation data and predictions. The total_rmse, total_rmsle, and num_models variables are updated to allow us to calculate the average rmse/rmsle once all the models are trained. Then, RMSE and RMSLE for the current store and family are printed (Figure 4.2.1.3.2.14).

```

rmse = np.sqrt(mean_squared_error(val, predictions))
rmsle = np.sqrt(mean_squared_error(np.log1p(val), np.log1p(predictions)))

total_rmse += rmse
total_rmsle += rmsle
num_models += 1

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

```

Figure 4.2.1.3.2.14 Calculating RMSE and RMSLE, adding them to calculate average later, incrementing number of models, and printing the RMSE and RMSLE values

The plot for actual values vs predicted values is generated, saved as a PNG file, and closed to free up memory (Figure 4.2.1.3.2.15). The plots are saved instead of being shown in the notebook as the notebook increased the 100MB limit of GitHub.

```

plt.figure(figsize=(12, 6))
plt.plot(val, label='Actual')
plt.plot(predictions, label='Predictions')
plt.title(f"Model for store {store_nbr} and family {family}")
plt.legend()
plt.savefig(f"plots/M05.6_store{store_nbr}_family{family}.png")
plt.close()

```

Figure 4.2.1.3.2.15 Generating a plot of actual values vs predicted values and saving it locally

RMSE and RMSLE is printed for each model as it is trained (Figure 4.2.1.3.2.16). For example, the model for store 1 and family 0 produced a RMSE of 2.856 and RMSLE of 0.601.

```

Training model for store 1 and family 0...
Root Mean Squared Error (RMSE): 2.8560137175621243
Root Mean Squared Logarithmic Error (RMSLE): 0.6012867405294068

Training model for store 1 and family 1...
Root Mean Squared Error (RMSE): 0.0
Root Mean Squared Logarithmic Error (RMSLE): 0.0

Training model for store 1 and family 2...
Root Mean Squared Error (RMSE): 2.5138190765516484
Root Mean Squared Logarithmic Error (RMSLE): 0.5794790165175364

```

Figure 4.2.1.3.2.16 Printing the RMSE and RMSLE values of each model as it is trained

The average RMSE and RMSLE produced for the 1782 models was 220 and 0.749 respectively (Figure 4.2.1.3.2.17). This is a huge improvement over the results from the multiseries models trained (where LSTM produced the best results with a RMSE of 566).

```

avg_rmse = total_rmse / num_models
avg_rmsle = total_rmsle / num_models

print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

Average Root Mean Squared Error (RMSE) across all models: 219.98350599458203
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.7491666546394734

```

Figure 4.2.1.3.2.17 Printing the average RMSE and RMSLE values

Some combinations of the store and family did not have any data at all. Meaning, either this family was not available at the store or did not have any sales. As a result, the model is not fitted (Figure 4.2.1.3.2.18). The model produced an RMSE/RMSLE of 0 as shown in Figure 4.2.1.3.2.16 and was therefore removed from the calculation of the average RMSE/RMSLE in a later iteration.

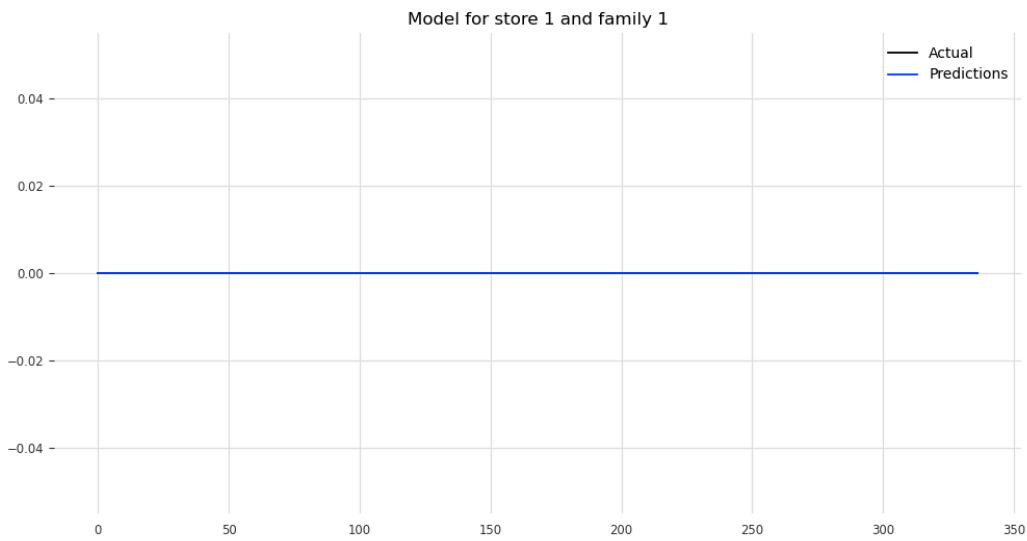


Figure 4.2.1.3.2.18 Untrained Random Forest model due to no data

Some combinations of the store and family had very little data. Meaning, this family did not have a lot of sales at this store. As a result, the model is under-fitted (Figure 4.2.1.3.2.19).

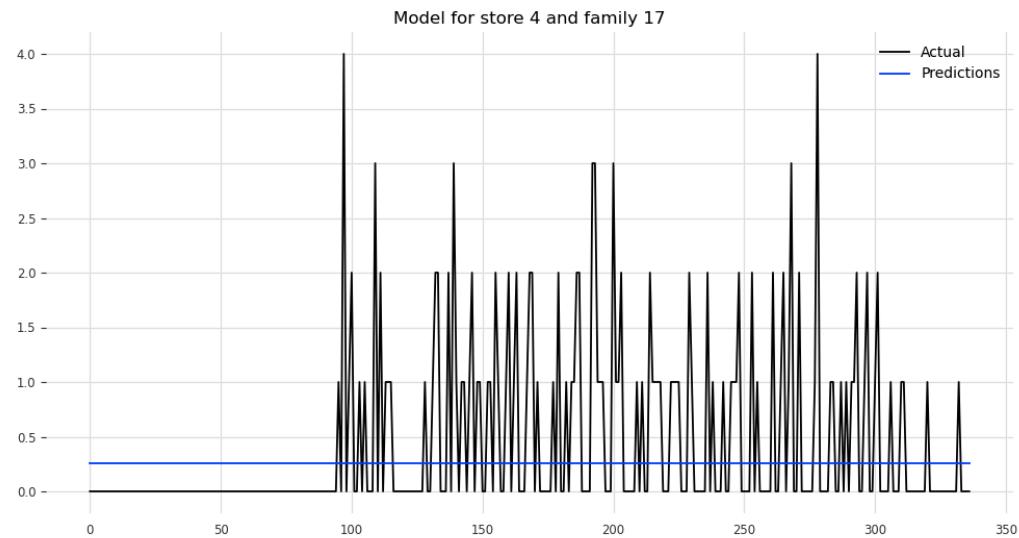


Figure 4.2.1.3.2.19 Under-fitted Random Forest model due to very little data

The combinations of the store and family that had a good amount of data produced a well-fitted model (Figure 4.2.1.3.2.20).

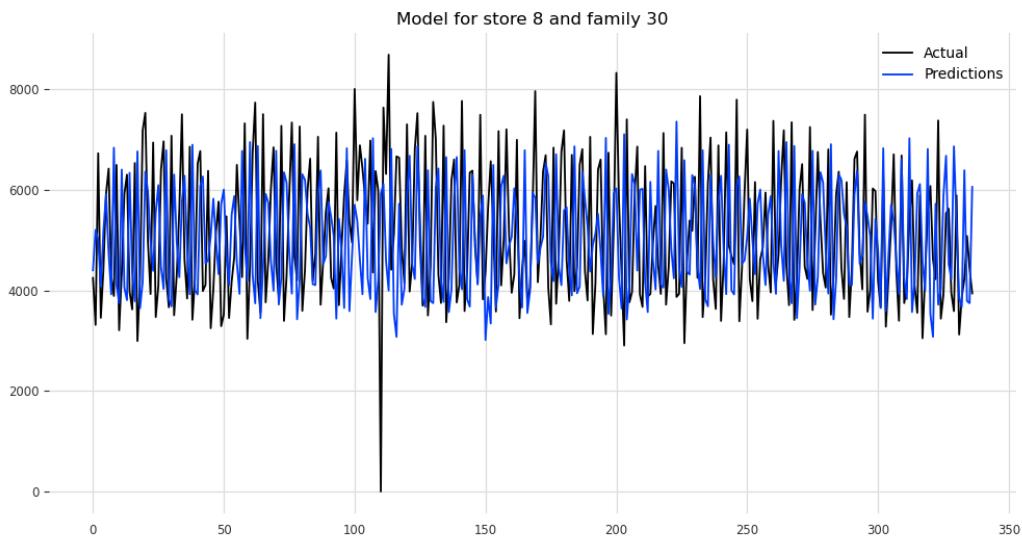


Figure 4.2.1.3.2.20 Well-fitted Random Forest model due to abundant data

As seen in Figure 4.2.1.3.2.16 and Figure 4.2.1.3.2.18, some models which were not fitted due to no data produced RMSE/RMSLE of 0. We suspected the low average RMSE/RMSLE might be due to these models so we excluded these models from the calculation by adding the following condition (Figure 4.2.1.3.2.21).

```
if rmse > 0:
    total_rmse += rmse
    total_rmsle += rmsle
    num_models += 1
```

Figure 4.2.1.3.2.21 Not counting models with RMSE=0

The RMSE (220 to 224) and RMSLE (0.749 to 0.771) only showed a slight increase meaning the results were not majorly affected by these unfitted models (4.2.1.3.2.22).

```
avg_rmse = total_rmse / num_models
avg_rmsle = total_rmsle / num_models
```

```
print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")
```

```
Average Root Mean Squared Error (RMSE) across all models: 224.14517415031307
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.7706459304868941
```

Figure 4.2.1.3.2.22 Calculating and printing average RMSE and RMSLE

Training took a lot of time, as all the available resources were not utilised (Figure 4.2.1.3.2.23).

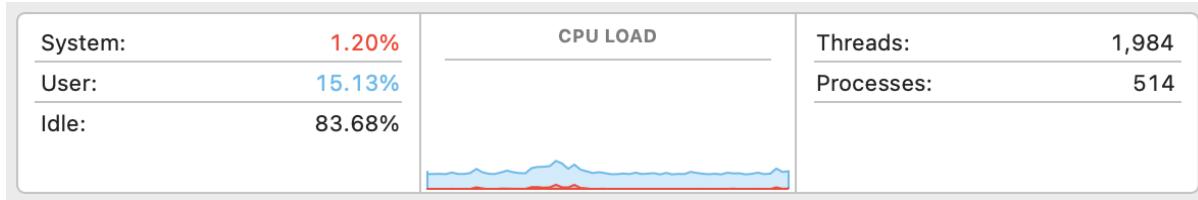


Figure 4.2.1.3.2.23 CPU usage by Random Forest

4.2.1.3.2 XGBoost

Next, we trained a XGBoost model (Figure 4.2.1.3.3.1) with the same configurations as the Random Forest model.

```
model = XGBModel(lags=2)
model.fit(train)
```

Figure 4.2.1.3.3.1 Instantiating and training the XGBoost model

The average RMSE and RMSLE produced for the XGBoost models was 232 and 0.763 respectively (Figure 4.2.1.3.3.2).

```
print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")
```

```
Average Root Mean Squared Error (RMSE) across all models: 231.8516885931285
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.763459383953906
```

Figure 4.2.1.3.3.2 Printing the average RMSE and RMSLE values

The same combinations of the store and family that were analysed for Random Forest were analysed for XGBoost as well. For store 1 and family 1, XGBoost did not fit as well due to no data (Figure 4.2.1.3.3.3).

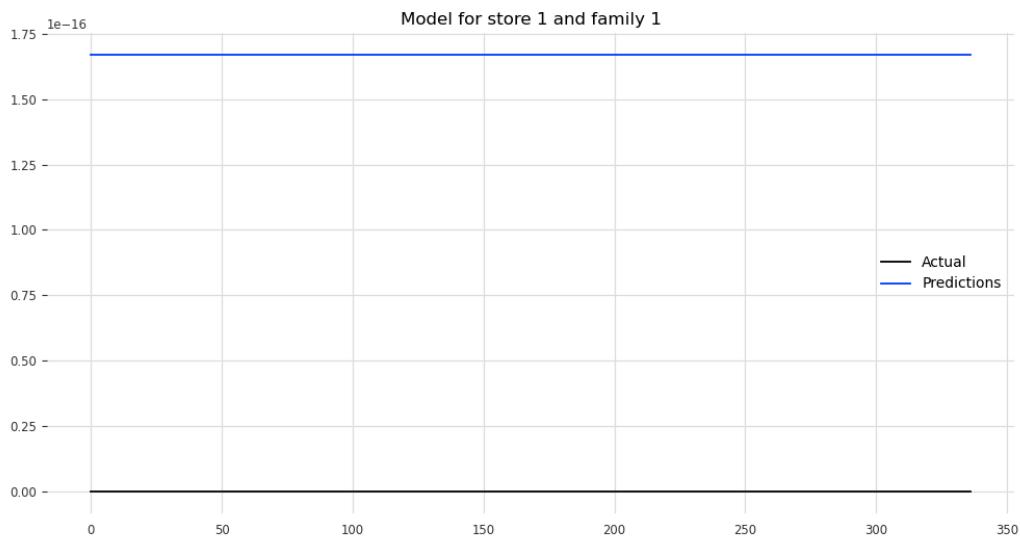


Figure 4.2.1.3.3.3 Untrained XGBoost model due to no data

Similar to Random Forest, XGBoost generated an under-fitted model for store 4 and family 17 due to very little data (Figure 4.2.1.3.3.4).

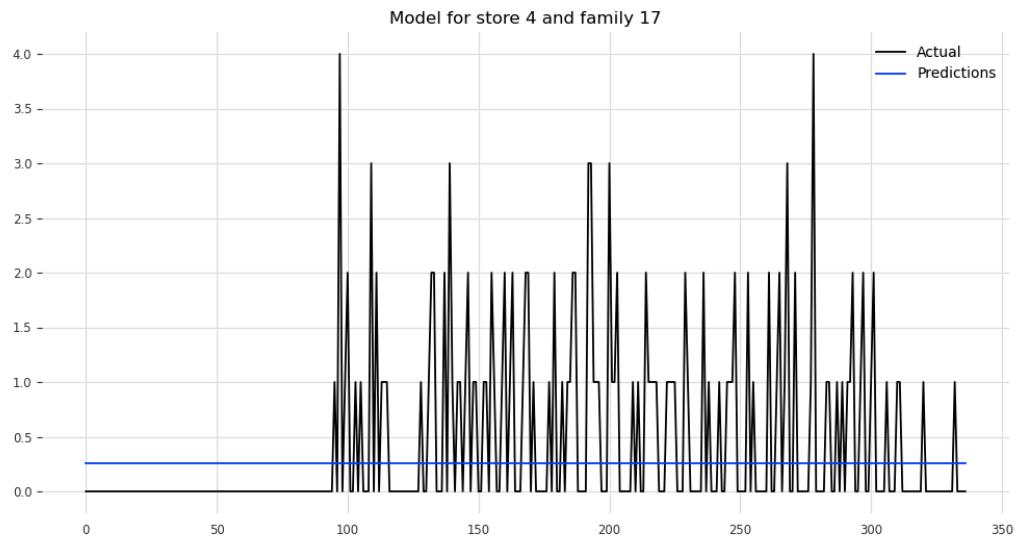


Figure 4.2.1.3.3.4 Under-fitted XGBoost model due to very little data

XGBoost produced a well-fitted model for store 8 and family 30 as a good amount of data was available (Figure 4.2.1.3.3.5).

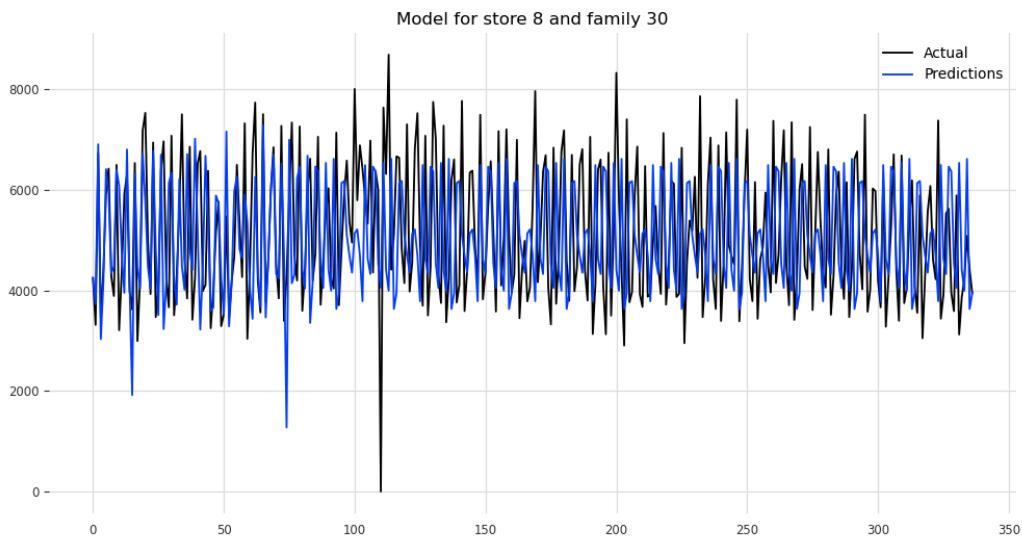


Figure 4.2.1.3.3.5 Well-fitted XGBoost model due to abundant data

The XGBoost models trained significantly faster than the Random Forest models. One of the reasons being that XGBoost utilised all the available resources while training (Figure 4.2.1.3.3.6).

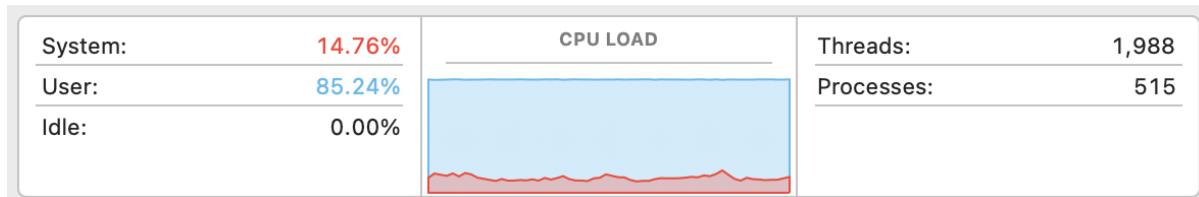


Figure 4.2.1.3.3.6 CPU usage by XGBoost

Despite XGBoost producing slightly worse results compared to Random Forest (Table 4.2.1.3.3.7), XGBoost was preferred for experimentation during the project due to its faster training time.

Table 4.2.1.3.3.7 Random Forest vs XGBoost

Model	RMSE	RMSLE
Random Forest	220	0.749
XGBoost	232	0.763

The implemented univariate single series Random Forest and XGBoost models using the Darts library were evaluated based on a validation set which is 20% of the training set. So, we decided to train the models on 100% of the training data and evaluate them on the unseen data by submitting the predictions in the Store Sales competition on Kaggle. XGBoost was used as the algorithm for this experiment as

XGBoost models train the fastest. As we are only creating univariate models, train_darts_univariate.csv is used to train the models (Figure 4.2.1.3.3.8).

```
train = pd.read_csv("../Data/Kaggle/StoreSales/train_darts_univariate.csv", parse_dates=[ "date" ])
```

```
train.head()
```

	date	store_nbr	family	sales
0	2013-01-01	1	0	0.0
1	2013-01-01	1	1	0.0
2	2013-01-01	1	2	0.0
3	2013-01-01	1	3	0.0
4	2013-01-01	1	4	0.0

Figure 4.2.1.3.3.8 Loading the dataset and inspecting it

Two empty dictionaries (trained_models and predictions) are created to store the models and the predictions from each model (Figure 4.2.1.3.3.9).

```
trained_models = {}
predictions = {}
```

Figure 4.2.1.3.3.9 Declaring dictionaries

Previously, we used a trained_models dictionary with a tuple (store_nbr, family) as the key. The tuple served as an identifier for retrieving the corresponding model later. To make things simpler, we have used the same logic for predictions as well. Predictions are generated for each model and saved in a dictionary with the tuple (store_nbr, family) as the key (Figure 4.2.1.3.3.10).

```
for index, row in unique_combinations.iterrows():
    store_nbr = row['store_nbr']
    family = row['family']

    print(f"\nTraining model for store {store_nbr} and family {family}...")
    train_subset = train[(train['store_nbr'] == store_nbr) & (train['family'] == family)]
    ts = TimeSeries.from_dataframe(train_subset, value_cols=["sales"])

    model = XGBModel(lags=2)
    model.fit(ts)

    trained_models[(store_nbr, family)] = model

    predictions[(store_nbr, family)] = model.predict(n=16)
    predictions[(store_nbr, family)] = predictions[(store_nbr, family)].pd_series().tolist()
    predictions[(store_nbr, family)] = np.maximum(predictions[(store_nbr, family)], 0)
```

Figure 4.2.1.3.3.10 Training models and generating predictions

We used the test set provided at the Store Sales competition to create test_darts_univariate.csv (Figure 4.2.1.3.3.11). This csv file contains the same columns as the training data used to train the models.

```
test = pd.read_csv("../Data/Kaggle/StoreSales/test_darts_univariate.csv", parse_dates=[ "date" ])
```

Figure 4.2.1.3.3.11 Loading the dataset

As seen in Figure 4.2.1.3.3.12, the dataset contains data for 16 time steps (days). The testset contains data for each store and family from 16th August 2017 to 31st August 2017. This is the reason the predictions were generated for 16 time steps (Figure 4.2.1.3.3.10).

```
test.head()
```

	id	date	store_nbr	family
0	3000888	2017-08-16	1	0
1	3000889	2017-08-16	1	1
2	3000890	2017-08-16	1	2
3	3000891	2017-08-16	1	3
4	3000892	2017-08-16	1	4

```
test.tail()
```

	id	date	store_nbr	family
28507	3029395	2017-08-31	9	28
28508	3029396	2017-08-31	9	29
28509	3029397	2017-08-31	9	30
28510	3029398	2017-08-31	9	31
28511	3029399	2017-08-31	9	32

Figure 4.2.1.3.3.12 Inspecting the dataset

In order to add the predictions to the test csv file, we implemented a nested loop approach (Figure 4.2.1.3.3.13). The outermost (1st) loop iterates for each day, then the 2nd loop iterates for each store, and lastly the 3rd loop iterates for each family. However, the results were not what was expected.

```
# for i in range(16):
#     for store_nbr in range(1, 55):
#         for family in range(33):
#             if (store_nbr, family) in predictions:
#                 print(f"store {store_nbr} family {family}: {predictions[(store_nbr, family)][i]}")
#                 test_sales.append(predictions[(store_nbr, family)][i])
```

Figure 4.2.1.3.3.13 Nested Loop Approach

Upon manual inspection of the dataset (Figure 4.2.1.3.3.14), we noticed the store numbers did not increase as we expected (1, 2, 3... 52, 53, 54). They increased in a different way (1, 10, 11... 7, 8, 9).

id	date	store_nbr	family
3000918	2017-08-16	1	30
3000919	2017-08-16	1	31
3000920	2017-08-16	1	32
3000921	2017-08-16	10	0
3000922	2017-08-16	10	1
3000923	2017-08-16	10	2

Figure 4.2.1.3.3.14 Inspecting the dataset manually

In order to achieve the ordering of the test set, we had to use a dictionary iteration approach (Figure 4.2.1.3.3.15). The outer loop iterates for each day. The inner loop iterates over each key-value pair in the predictions dictionary using predictions.items().

```
for i in range(16):
    for (store_nbr, family), prediction in predictions.items():
        print(f"Store {store_nbr}, Family {family} predictions: {prediction[i]}")
        test_sales.append(prediction[i])
```

Figure 4.2.1.3.3.15 Dictionary Iteration Approach

The predictions (test_sales) are added to the test dataframe as a sales column. All the extra columns are dropped as the submission should only have two columns, id and sales. A csv file is created (submission.csv) for the submission on Kaggle (Figure 4.2.1.3.3.16).

```
test['sales'] = test_sales

test.drop(columns=['date', 'store_nbr', 'family'], inplace=True)

test.to_csv('submission.csv', index=False)
```

Figure 4.2.1.3.3.16 Creating the submission csv

The submission produced a RMSLE of 0.55607 (Figure 4.2.1.3.3.17) which is a significant improvement over the best RMSLE achieved (1.08120) on the unseen data previously.

Submissions

All	Successful	Errors	Recent ▾
Submission and Description			Public Score ⓘ
	submission.csv	Complete · 5m ago · XGBModel	0.55607
	submission.csv	Complete · 4mo ago · XGBRegressor	1.08120

Figure 4.2.1.3.3.17 Submissions to Store Sales (Kaggle)

The latest submission (0.55607) has earned us 439th position on the competition's leaderboard (Figure 4.2.1.3.3.18). The first position on the leaderboard has a RMSLE of 0.37786.

Store Sales - Time Series Forecasting						
	Overview	Data	Code	Models	Discussion	Leaderboard
	435	skajsfe				0.54880 7 5d
	436	Shailadhd				0.54906 1 13d
	437	Caesar Eriksson				0.55134 2 2mo
	438	Patmos	 			0.55561 5 1mo
	439	Irtaza Ahmed Khan				0.55607 1 1m
 Your Best Entry! Your most recent submission scored 0.55607, which is an improvement of your previous score of 1.08120. Great job!						
	440	Charlotte2024				0.55813 5 1mo
	441	CharlotteMrs				0.55813 19 25d
	442	Denis Rukavishnikov				0.55996 1 23d
	443	Pituchai Mitpakdee				0.56446 1 2mo
	444	Sina Jabbari				0.57261 3 2mo
	445	Mahender ch				0.57426 7 18d

Figure 4.2.1.3.3.18 Store Sales Leaderboard

4.2.1.3.3 Prophet

Next, we implemented univariate single series Prophet models. To make things simpler, we created a new csv file (train_darts_univariate.csv) with only 4 columns (date, store_nbr, family, and sales) as we're

creating univariate models. Then, we set the date column as the index (Figure 4.2.1.3.4.1) as Prophet does not work with integer indices.

```
df = pd.read_csv("../Data/Kaggle/StoreSales/train_darts_univariate.csv", parse_dates=[ "date" ])
```

```
df.set_index('date', inplace=True)
```

```
df.head()
```

	store_nbr	family	sales
date			
2013-01-01	1	0	0.0
2013-01-01	1	1	0.0
2013-01-01	1	2	0.0
2013-01-01	1	3	0.0
2013-01-01	1	4	0.0

Figure 4.2.1.3.4.1 Loading the dataset, updating index, and inspecting it

We trained Prophet models with the same configurations as the Random Forest and XGBoost models. The models were fitted without any problems but gave an error upon the calculation of RMSE that there are NaN values in the validation set. This was strange as the same dataset was used to train the Random Forest and XGBoost models. We counted the number of NaN values and there was exactly 1 NaN value in each store and family combination (Figure 4.2.1.3.4.2).

```
Training model for store 52 and family 4...
There are 1 NaN values in the data.
Root Mean Squared Error (RMSE): 0.0
Root Mean Squared Logarithmic Error (RMSLE): 0.0

Training model for store 52 and family 5...
There are 1 NaN values in the data.
Root Mean Squared Error (RMSE): 501.25798358438607
Root Mean Squared Logarithmic Error (RMSLE): 3.9561522173747368

Training model for store 52 and family 6...
There are 1 NaN values in the data.
Root Mean Squared Error (RMSE): 7.544934426683811
Root Mean Squared Logarithmic Error (RMSLE): 1.4490694369067494
```

Figure 4.2.1.3.4.2 Count of NaN values in each df subset

Considering there was only 1 NaN value in each dataset, it was converted to 0 to run the program (Figure 4.2.1.3.4.3).

```

nan_indices = np.isnan(val)
nan_count = np.sum(nan_indices)

if nan_count > 0:
    print(f"There are {nan_count} NaN values in the data.")

val = np.nan_to_num(val, nan=0.0)

rmse = np.sqrt(mean_squared_error(val, predictions))
rmsle = np.sqrt(mean_squared_error(np.log1p(val), np.log1p(predictions)))

```

Figure 4.2.1.3.4.3 Converting NaN values to 0

The average RMSE and RMSLE produced for the 1782 models was 170 and 0.766 respectively (Figure 4.2.1.3.4.4).

```

print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

```

```

Average Root Mean Squared Error (RMSE) across all models: 169.72125261564008
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.7663137817858955

```

Figure 4.2.1.3.4.4 Printing the average RMSE and RMSLE values

This is a further improvement over the results from the Random Forest and XGBoost models (Table 4.2.1.3.4.5).

Table 4.2.1.3.4.5 Random Forest vs XGBoost vs Prophet

Model	RMSE	RMSLE
Random Forest	220	0.749
XGBoost	232	0.763
Prophet	170	0.766

The same combinations of the store and family that were analysed for Random Forest and XGBoost were analysed for Prophet as well. For store 1 and family 1, Prophet did not fit as well due to no data (Figure 4.2.1.3.4.6).

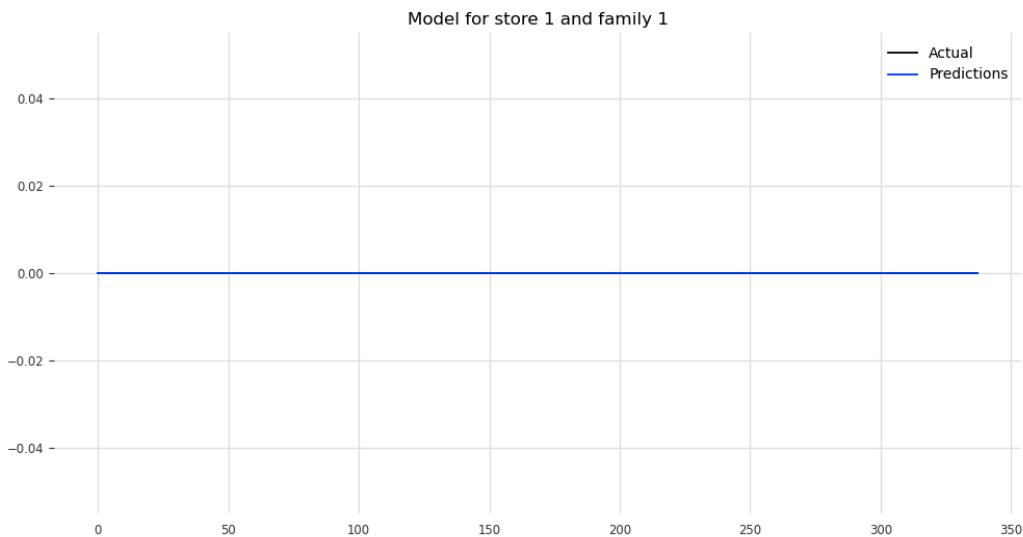


Figure 4.2.1.3.4.6 Untrained Prophet model due to no data

Similar to Random Forest and XGBoost, Prophet generated an under-fitted model for store 4 and family 17 due to very little data (Figure 4.2.1.3.4.7).

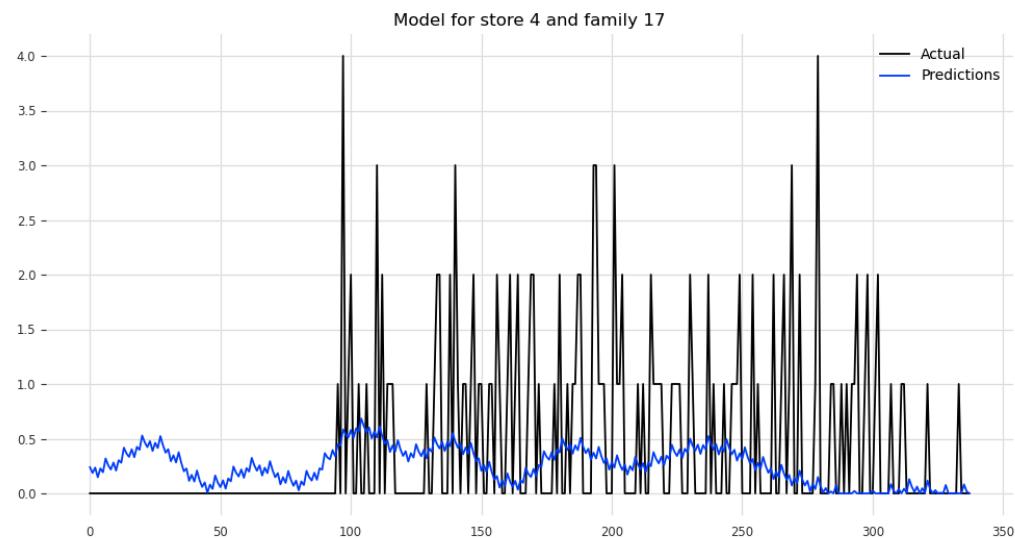


Figure 4.2.1.3.4.7 Under-fitted Prophet model due to very little data

Prophet produced a well-fitted model for store 8 and family 28 as a good amount of data was available (Figure 4.2.1.3.4.8).



Figure 4.2.1.3.4.8 Well-fitted Prophet model due to abundant data

The Prophet models trained slightly faster than the Random Forest models, but slower than the XGBoost models. One of the reasons being that the Prophet utilised more resources than Random Forest but less than XGBoost (Figure 4.2.1.3.4.9).

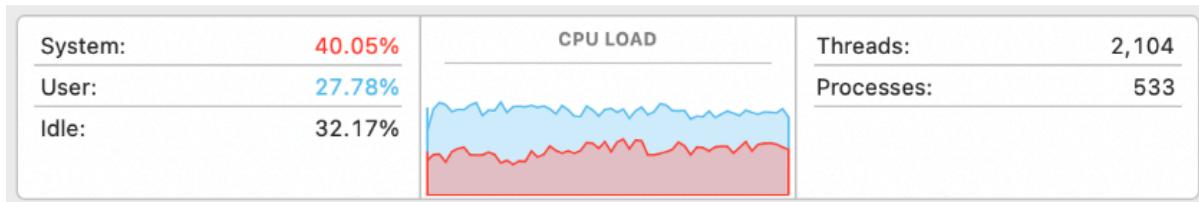


Figure 4.2.1.3.4.9 CPU usage by Prophet

4.2.1.3.4 Recurrent Neural Networks

Next, we tried implementing univariate single series Recurrent Neural Network (RNN) models. We continue to use `train_darts_univariate.csv`, which only has 4 columns as we are training univariate models, to reduce the computational load (Figure 4.2.1.3.5.1).

```
df.head()

  date  store_nbr  family  sales
0 2013-01-01      1       0    0.0
1 2013-01-01      1       1    0.0
2 2013-01-01      1       2    0.0
3 2013-01-01      1       3    0.0
4 2013-01-01      1       4    0.0
```

Figure 4.2.1.3.5.1 Inspecting the dataframe

The same steps were followed to train the RNN models as the previous models trained using Darts. Configurations specific to RNN (Figure 4.2.1.3.5.2) involved setting the variant to LSTM and the number of epochs to 10. The number of epochs was set to 10 as [DataScientist](#) suggested “Generally, a number of 11 epochs is ideal for training on most datasets.”

```
model = RNNModel(
    model="LSTM",
    input_chunk_length=7,
    training_length=14,
    n_epochs=10,
)
```

Figure 4.2.1.3.5.2 Instantiating the RNN model(s)

The model did not train producing the following error:

TypeError: Cannot convert a MPS Tensor to float64 dtype as the MPS framework doesn't support float64. Please use float32 instead.

Converting the datatype of sales from float64 to float32 (Figure 1.3) resolved the issue as suggested in this [GitHub](#) issue.

```
df['sales'] = df['sales'].astype('float32')
```

Figure 4.2.1.3.5.3 Converting the datatype of sales to float32

The models started to train (Figure 4.2.1.3.5.4). As each model ran 10 epochs, the training process took a lot of time and soon the laptop started to heat up. 1782 models had to be trained but we had to stop the training process after the first 180 models. The RMSE/RMSLE is calculated and the plot (actual vs predicted) is generated for each model as it is trained. So, we were able to analyse the first 180 models.

```

Training model for store 1 and family 2...
Training: |          | 0/? [00:00<?, ?it/s]
`Trainer.fit` stopped: `max_epochs=10` reached.
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
Predicting: |          | 0/? [00:00<?, ?it/s]
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

| Name      | Type       | Params
-----
0 | criterion | MSELoss   | 0
1 | train_metrics | MetricCollection | 0
2 | val_metrics | MetricCollection | 0
3 | rnn        | LSTM      | 2.8 K
4 | V          | Linear    | 26
-----
2.8 K   Trainable params
0       Non-trainable params
2.8 K   Total params
0.011   Total estimated model params size (MB)
Root Mean Squared Error (RMSE): 2.5408089397150624
Root Mean Squared Logarithmic Error (RMSLE): 0.5846747986406594

```

Figure 4.2.1.3.5.4 Training RNN model(s)

Some combinations of the store and family did not have any data at all. Meaning, either this family was not available at the store or did not have any sales. As a result, the model is not fitted (Figure 4.2.1.3.5.5).



Figure 4.2.1.3.5.5 Untrained RNN model due to no data

Some combinations of the store and family had very little data. Meaning, this family did not have a lot of sales at this store. As a result, the model is under-fitted (Figure 4.2.1.3.5.6).

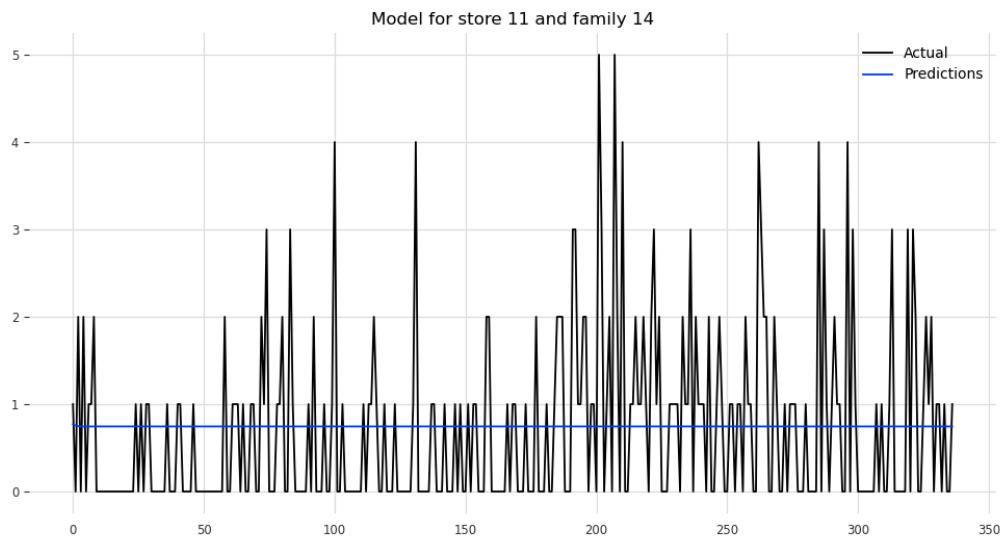


Figure 4.2.1.3.5.6 Under-fitted RNN model due to very little data

The performance shown in Figure 1.5 and Figure 1.6 was expected as Random Forest, XGBoost, and Prophet performed the same way. What came as a surprise was how RNN performed when there was an adequate amount of data available (Figure 4.2.1.3.5.7). Neither of the models previously trained were this under-fitted when this amount of data was available. We believe there could be 2 reasons for the under-fitted RNN models: the amount of data and the number of epochs. As the data is divided for each store and family, each subset contains only 1684 rows out of which 20% are reserved for validation. This is not enough to train a neural network. However, we want to increase the number of epochs to ensure it is not because of that.

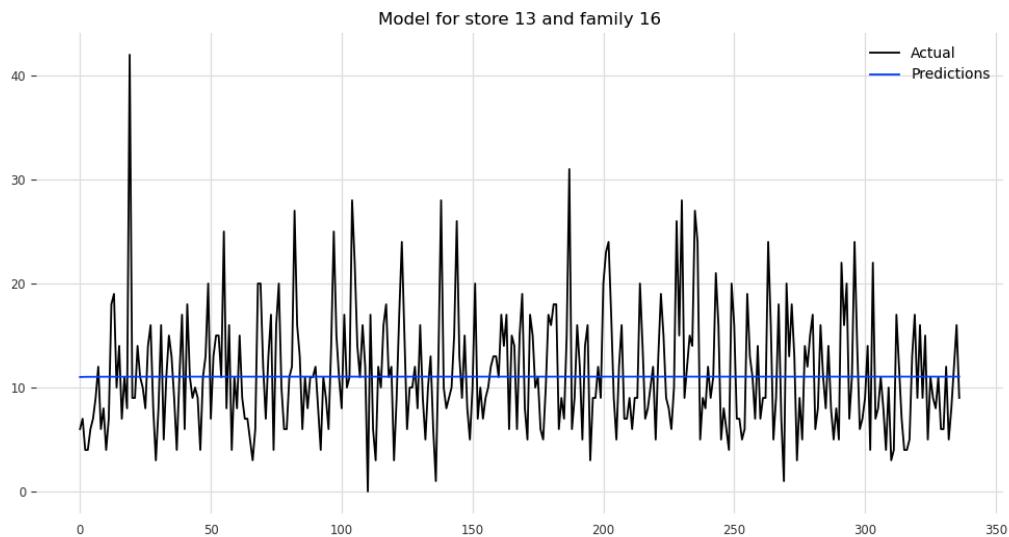


Figure 4.2.1.3.5.7 Under-fitted RNN model despite adequate amount of data

The RNN also trained slower due to the reason it did not fully utilise the available resources (Figure 4.2.1.3.5.8).

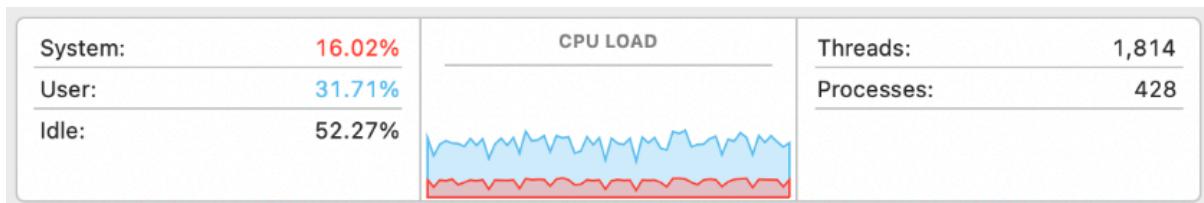


Figure 4.2.1.3.5.8 CPU usage by RNN

We tried implementing univariate single series RNN models, however, we were not able to complete the training due to computational constraints. So, we trained RNN models on the AI Server (The setup of the AI Server has been explained in Appendix C). The same steps were followed to train the RNN models as the previous models trained using Darts. Configurations specific to RNN (Figure 4.2.1.3.5.9) involved setting the variant to LSTM and the number of epochs to 10.

```
model = RNNModel(
    model="LSTM",
    input_chunk_length=7,
    training_length=14,
    n_epochs=10,
)
```

Figure 4.2.1.3.5.9 Instantiating the RNN model(s)

The models were trained (Figure 4.2.1.3.5.10) using the GPU (GPU available: True (cuda), used: True). The recommendation to set_float32_matmul_precision to ‘medium’ or ‘high’ was ignored while training

the RNN models. It was set to high while training the N-BEATS models and it did improve the performance.

```

Training model for store 1 and family 0...
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
You are using a CUDA device ('NVIDIA GeForce RTX 4070 Ti') that has Tensor Cores. To properly utilize them, you should set `torch.set_float32_matmul_precision('medium' | 'high')` which will trade-off precision for performance. For more details, read https://pytorch.org/docs/stable/generated/torch.set_float32_matmul_precision.html#torch.set_float32_matmul_precision
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

| Name           | Type            | Params
-----
0 | criterion     | MSELoss         | 0
1 | train_metrics | MetricCollection | 0
2 | val_metrics   | MetricCollection | 0
3 | rnn            | LSTM             | 2.8 K
4 | V              | Linear           | 26
-----
2.8 K      Trainable params
0          Non-trainable params
2.8 K      Total params
0.011     Total estimated model params size (MB)
Training: | 0/? [00:00<?, ?it/s]
`Trainer.fit` stopped: `max_epochs=10` reached.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Predicting: | 0/? [00:00<?, ?it/s]
Root Mean Squared Error (RMSE): 3.0360727407814148
Root Mean Squared Logarithmic Error (RMSLE): 0.6101217520438159

```

Figure 4.2.1.3.5.10 Training RNN model(s)

The average RMSE and RMSLE produced for the 1782 models was 508 and 1.984 respectively (Figure 4.2.1.3.5.11).

```

print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

Average Root Mean Squared Error (RMSE) across all models: 508.2327123630184
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 1.9837620175440323

```

Figure 4.2.1.3.5.11 Average RMSE and RMSLE

This is significantly worse as compared to Random Forest, XGBoost, and Prophet (Table 4.2.1.3.5.12).

Table 4.2.1.3.5.12 Random Forest vs XGBoost vs Prophet vs RNN

Model	RMSE	RMSLE
Random Forest	220	0.749
XGBoost	232	0.763
Prophet	170	0.766
Recurrent Neural Network	508	1.984

To ensure that the poor performance by the RNN is not due to less number of epochs, we increased the number of epochs to 50 (Figure 4.2.1.3.5.13). The models trained on the GPU for around 8 hours.

```
model = RNNModel(
    model="LSTM",
    input_chunk_length=7,
    training_length=14,
    n_epochs=50,
)
```

Figure 4.2.1.3.5.13 Number of epochs set to 50

It showed a minor improvement in the RMSE and RMSLE (Figure 4.2.1.3.5.14). From 508 to 495 and 1.984 to 1.434 respectively. Thus, proving that the poor performance was due to lack of sufficient data to train a neural network.

```
print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

Average Root Mean Squared Error (RMSE) across all models: 494.58386454750786
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 1.4339271610908098
```

Figure 4.2.1.3.5.14 Average RMSE and RMSLE

Here is the RNN model (n_epochs=10) for store 8 and family 30, a combination which had a good amount of data available (Figure 4.2.1.3.5.15).



Figure 4.2.1.3.5.15 Under-fitted RNN model (n_epochs=10)

Here is the RNN model ($n_epochs=50$) for the same combination showing how the increase in the number of epochs made little to no difference (Figure 4.2.1.3.5.16).

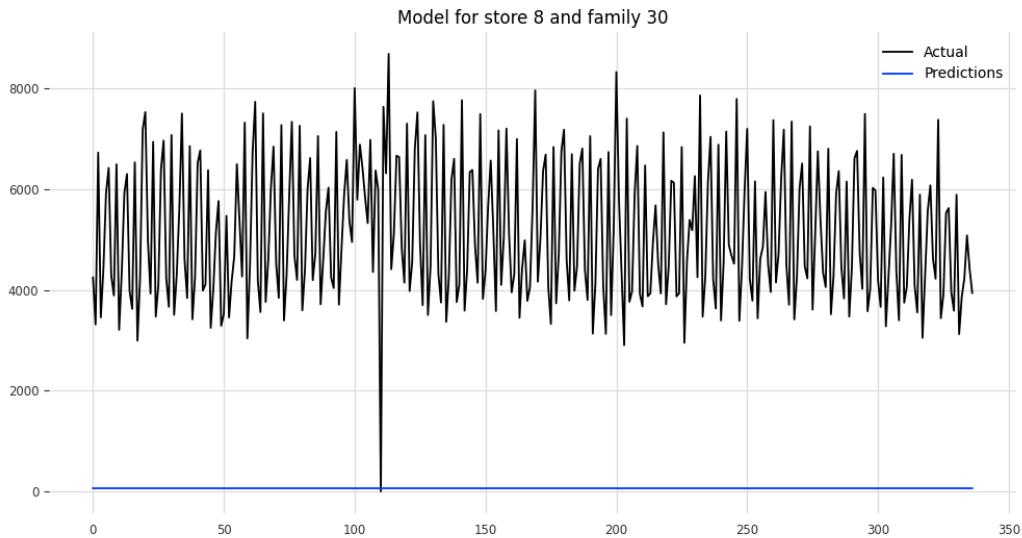


Figure 4.2.1.3.5.16 Under-fitted RNN model (n_epochs=50)

The models trained significantly faster on GPU as compared to CPU. However, training the RNN only utilised 20% of the GPU/CUDA (Figure 4.2.1.3.5.18). Upon research, we found out this is due to a small batch size (as we are training separate models for each store and product combination). Smaller batch sizes make GPU training inefficient.

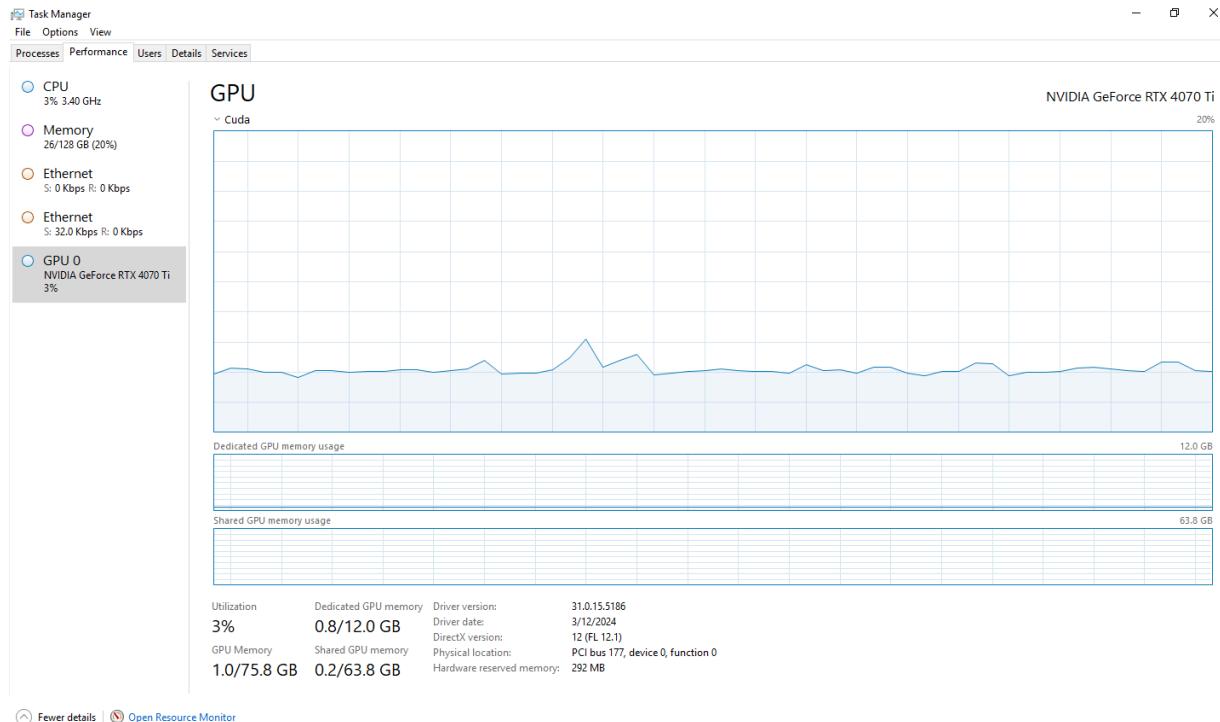


Figure 4.2.1.3.5.18 GPU usage by RNN

4.2.1.3.5 LightGBM

Next, we trained LightGBM models with different lags to capture daily, weekly, and monthly patterns. Firstly, we trained LightGBM models with lags=2 (Figure 4.2.1.3.6.1) to capture daily patterns. The same steps were followed to train the LightGBM models as the previous models trained using Darts.

```
model = LightGBMModel(lags=2)
model.fit(train)
```

Figure 4.2.1.3.6.1 Instantiating and training the LightGBM model(s)

The models produced an average RMSE of 216 and RMSLE of 0.706 across 1782 models (Figure 4.2.1.3.6.2).

```
print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

Average Root Mean Squared Error (RMSE) across all models: 216.45732056450802
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.7057847219887133
```

Figure 4.2.1.3.6.2 Printing the RMSE and RMSLE values

Here is a graph of predictions vs actual values that shows a well-fitted LightGBM model for store 8 and family 30 (Figure 4.2.1.3.6.3).



Figure 4.2.1.3.6.3 Well-fitted LightGBM model

Table 4.2.1.3.6.4 shows a comparison of the performance of the 5 algorithms we have used so far to train our forecasting models where Prophet produced the best RMSE and LightGBM produced the best RMSLE.

Table 4.2.1.3.6.4 Random Forest vs XGBoost vs Prophet vs RNN vs LightGBM

Model	RMSE	RMSLE
Random Forest	220	0.749
XGBoost	232	0.763
Prophet	170	0.766
Recurrent Neural Network	508	1.984
LightGBM	216	0.706

Then, we trained LightGBM models with lags=7 (Figure 4.2.1.3.6.5) to capture weekly patterns.

```
model = LightGBMModel(lags=7)
model.fit(train)
```

Figure 4.2.1.3.6.5 Instantiating and training the LightGBM model(s) with lags=7

The models produced an average RMSE of 224 and RMSLE of 0.757 across 1782 models (Figure 4.2.1.3.6.6).

```

print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

```

```

Average Root Mean Squared Error (RMSE) across all models: 224.3839850906194
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.7569359742406159

```

Figure 4.2.1.3.6.6 Printing the RMSE and RMSLE values

Lastly, we trained LightGBM models with lags=30 (Figure 4.2.1.3.6.7) to capture monthly patterns.

```

model = LightGBMModel(lags=30)
model.fit(train)

```

Figure 4.2.1.3.6.7 Instantiating and training the LightGBM model(s) with lags=30

The models produced an average RMSE of 227 and RMSLE of 0.731 across 1782 models (Figure 4.2.1.3.6.8).

```

print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

```

```

Average Root Mean Squared Error (RMSE) across all models: 226.54231068700406
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.7310139638439718

```

Figure 4.2.1.3.6.8 Printing the RMSE and RMSLE values

Table 4.2.1.3.6.9 shows a comparison of the effect of lag set on the average RMSE and RMSLE. Capturing daily patterns produced the best results, followed by weekly, and then monthly.

Table 4.2.1.3.6.9 Lag comparison in LightGBM

Lag	Pattern	RMSE	RMSLE
2	Daily	216	0.706
7	Weekly	224	0.757
30	Monthly	226	0.731

LightGBM models, despite being trained on the AI Server, used the CPU not GPU for training. However, they utilised a good amount of the resources available and therefore trained fast (Figure 4.2.1.3.6.10).

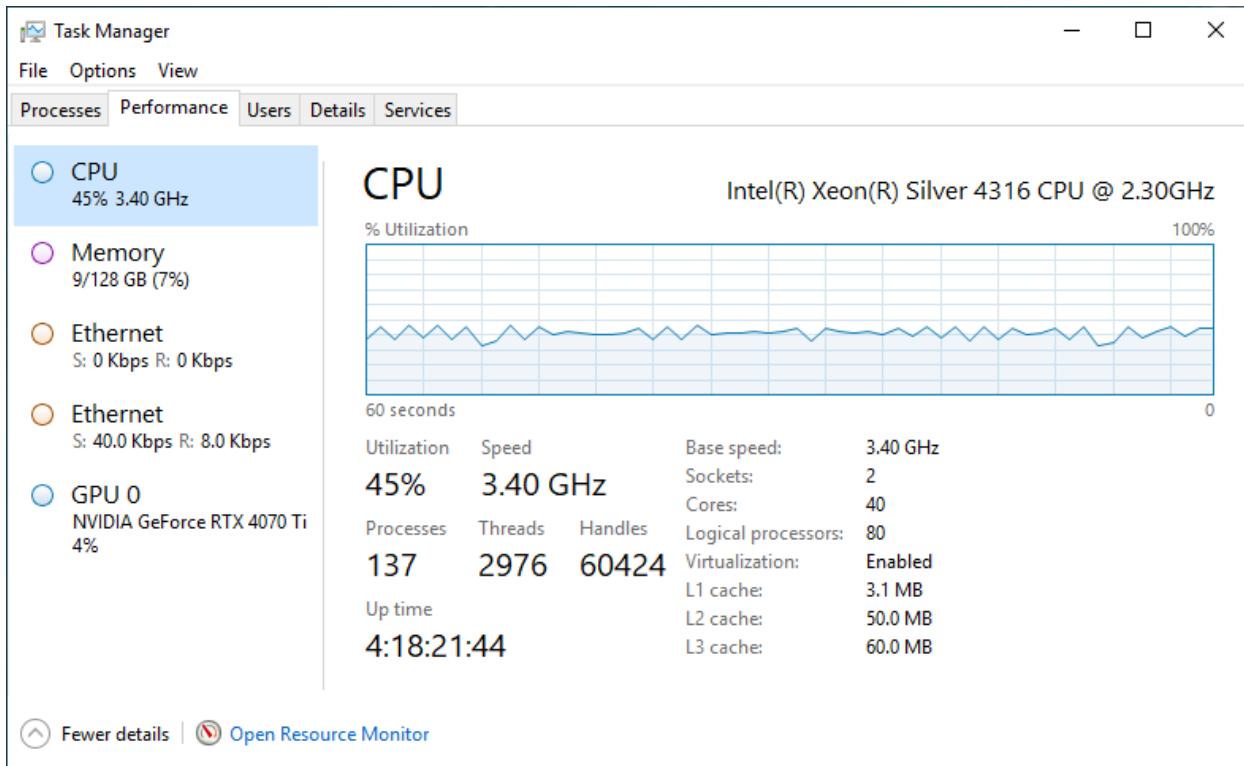


Figure 4.2.1.3.6.10 CPU usage by LightGBM

4.2.1.3.6 N-BEATS

Next, we wanted to train the N-BEATS models using Darts. We started off the training process by setting the precision of matrix multiplication to ‘medium’ (Figure 4.2.1.3.7.1). This was suggested while training the Recurrent Neural Networks as well, however, it was not used at the time. This was used to reduce the training time.

```
torch.set_float32_matmul_precision('medium')
```

Figure 4.2.1.3.7.1 Precision of matrix multiplications set to ‘medium’

The N-BEATS models were set up with input_chunk_length=7, output_chunk_length=7, and n_epochs=1 (Figure 4.2.1.3.7.2). The reason to set the number of epochs to 1 was to check if the training goes smoothly. Once that was insured, it was increased in a later iteration.

```

# pl_trainer_kwargs = {"accelerator": "cpu"}

model = NBEATSMModel(
    # pl_trainer_kwargs=pl_trainer_kwargs,
    input_chunk_length=7,
    output_chunk_length=7,
    n_epochs=1,
    activation='LeakyReLU'
)

```

Figure 4.2.1.3.7.2 Instantiating the N-BEATS model(s) with n_epochs=1

The models were trained on the GPU using CUDA (Figure 4.2.1.3.7.3). The training speed was slow as only 20% of the GPU was being utilised. We tried training the model using the CPU as shown by the commented out code in Figure 1.10, however, the models trained even slower on the CPU despite utilising more resources of the CPU. Therefore, GPU was used to train the models. Research showed that GPU training is inefficient when the batch size is small.

```

Training model for store 9 and family 27...
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

| Name           | Type            | Params
-----|-----|-----
0 | criterion     | MSELoss         | 0
1 | train_metrics | MetricCollection | 0
2 | val_metrics   | MetricCollection | 0
3 | stacks        | ModuleList      | 6.1 M
-----|-----|-----
6.1 M      Trainable params
1.3 K      Non-trainable params
6.1 M      Total params
24.249    Total estimated model params size (MB)
Training: |          | 0/? [00:00<?, ?it/s]
`Trainer.fit` stopped: `max_epochs=1` reached.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
Predicting: |          | 0/? [00:00<?, ?it/s]
Root Mean Squared Error (RMSE): 8.570587555887798
Root Mean Squared Logarithmic Error (RMSLE): 0.6894751792504655

```

Figure 4.2.1.3.7.3 Training the N-BEATS model(s) using the GPU

The models produced an average RMSE of 681,939 and RMSLE of 1.191 (Figure 4.2.1.3.7.4). This is the highest RMSE of any model we have trained to date.

```

print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

Average Root Mean Squared Error (RMSE) across all models: 681939.4291425318
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 1.1914348169161166

```

Figure 4.2.1.3.7.4 Printing the RMSE and RMSLE values

Upon plotting the predictions vs actual values graph, we noticed a strange trend. The predicted values start off well but either keep increasing (Figure 4.2.1.3.7.5) or keep decreasing (Figure 4.2.1.3.7.6). The models do not seem under fitted. The reason for why these trends are appearing is yet to be found.

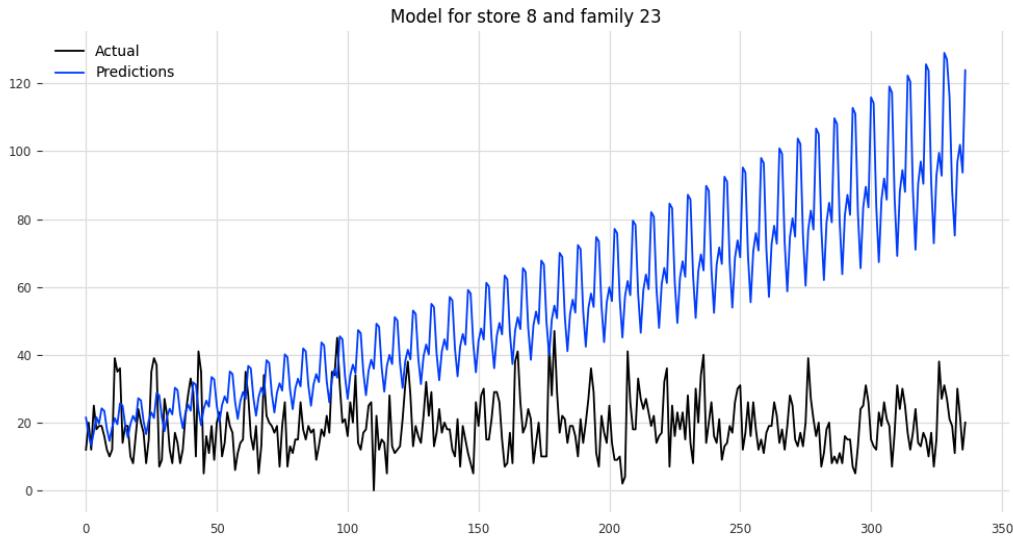


Figure 4.2.1.3.7.5 N-BEATS model showing an increasing pattern

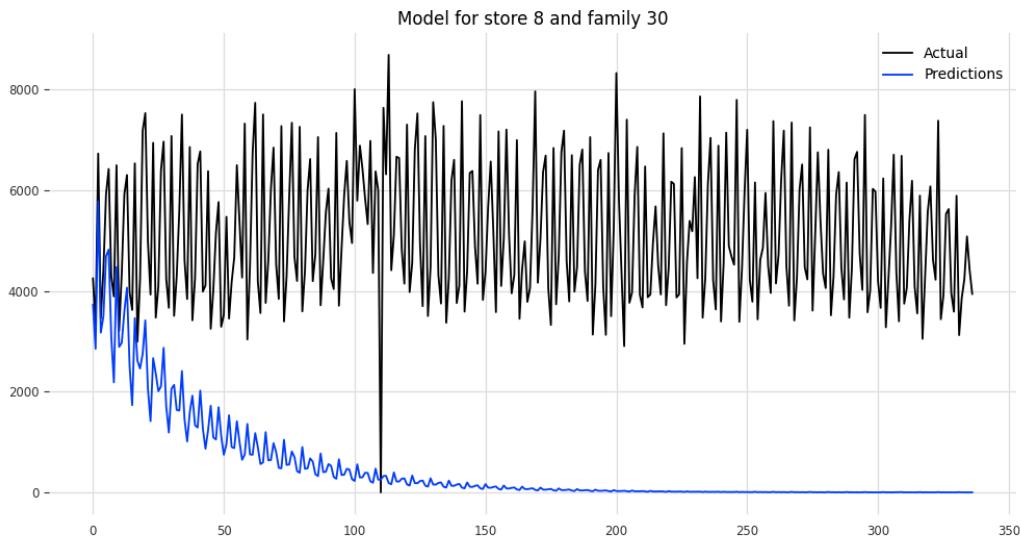


Figure 4.2.1.3.7.6 N-BEATS model showing a decreasing pattern

Table 4.2.1.3.7.7 shows a comparison of the performance of the 6 algorithms we have used so far to train our forecasting models where the drastic difference between the RMSE of N-BEATS and the remaining models can be seen.

Table 4.2.1.3.7.7 Random Forest vs XGBoost vs Prophet vs RNN vs LightGBM vs N-BEATS

Model	RMSE	RMSLE
Random Forest	220	0.749
XGBoost	232	0.763
Prophet	170	0.766
Recurrent Neural Network	508	1.984
LightGBM	216	0.706
N-BEATS	681,939	1.191

Once the N-BEATS model trained successfully with 1 epoch, we updated the parameters to input_chunk_length=90, output_chunk_length=7, and n_epochs=10 (Figure 4.2.1.3.7.8).

```
model = NBEATSMModel(
    input_chunk_length=90,
    output_chunk_length=7,
    n_epochs=10,
    activation='LeakyReLU'
)
```

Figure 4.2.1.3.7.8 Instantiating the N-BEATS model(s) with n_epochs=10

The models produced an average RMSE of 1,512,951,668 and RMSLE of 0.884 (Figure 4.2.1.3.7.9). Even though the RMSLE improved, the RMSE got significantly worse.

```
print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

Average Root Mean Squared Error (RMSE) across all models: 1512951668.430995
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.8842951907904852
```

Figure 4.2.1.3.7.9 Printing the RMSE and RMSLE values

Predictions vs actual values graphs were examined for the same combination of store and family (Figure 4.2.1.3.7.10 and Figure 4.2.1.3.7.11). The increasing and decreasing trends continue to exist.

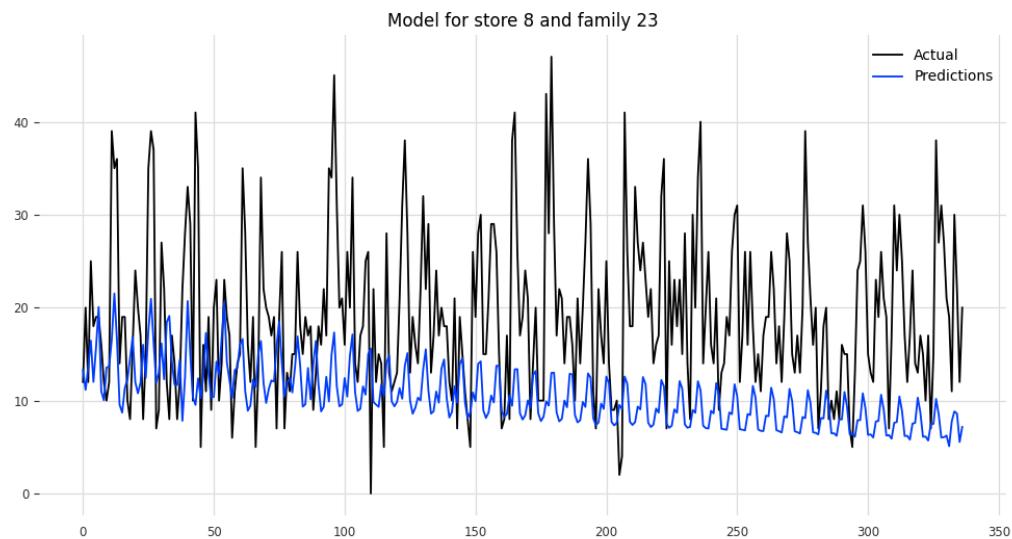


Figure 4.2.1.3.7.10 N-BEATS model showing a decreasing pattern

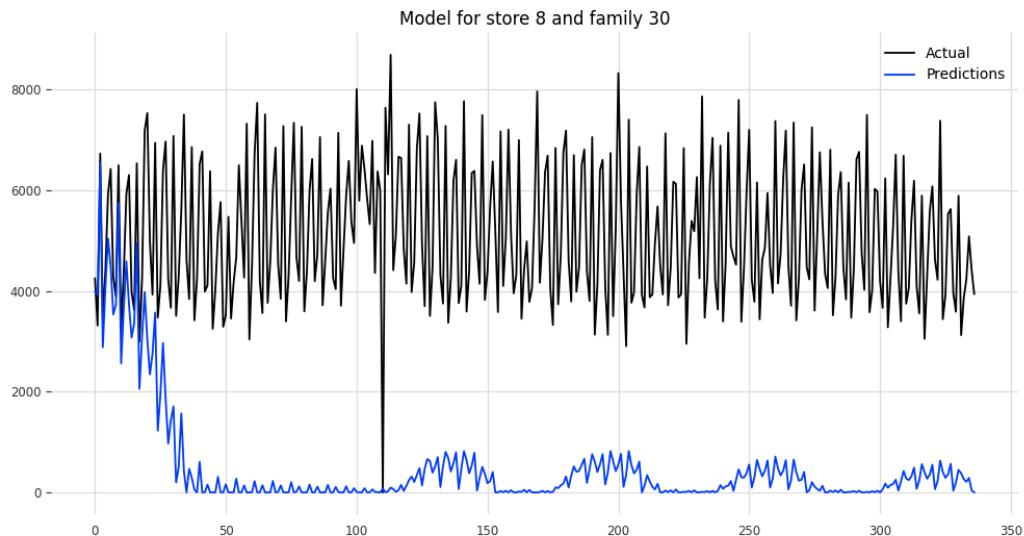


Figure 4.2.1.3.7.11 N-BEATS model showing a decreasing pattern

Training the N-BEATS models with 10 epochs utilised 122GB/95% of the memory available on the AI Server (Figure 4.2.1.3.7.12).

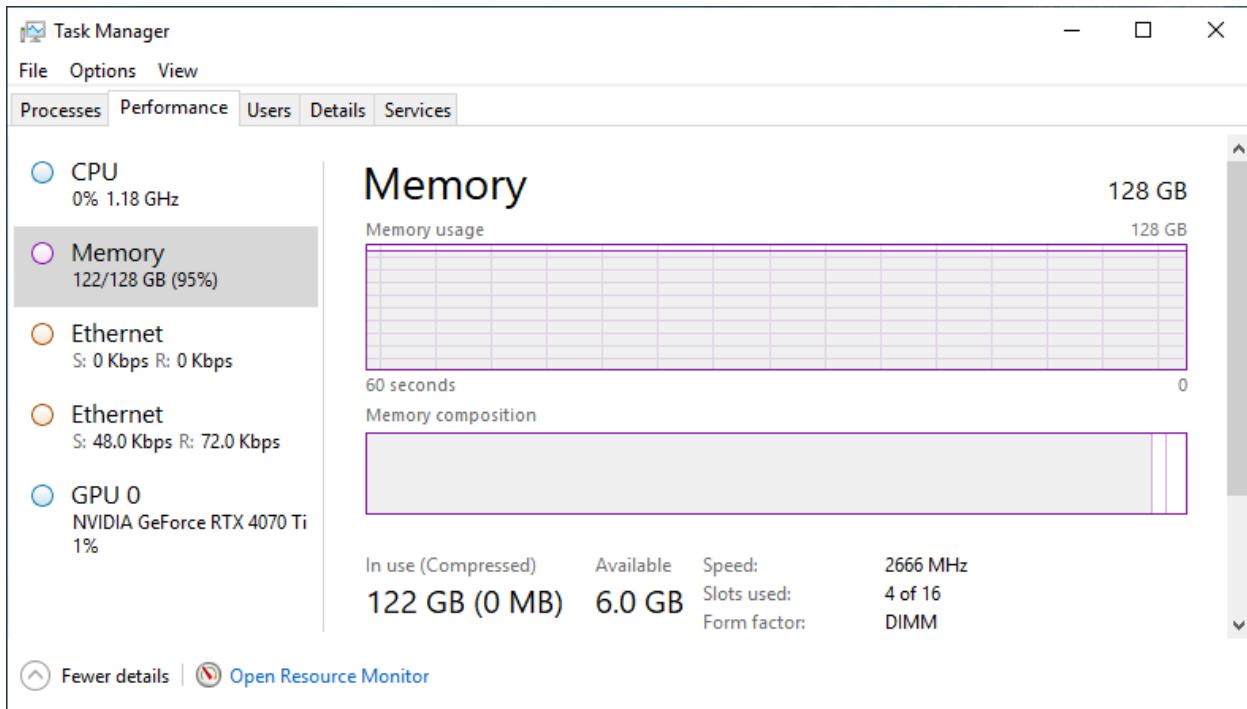


Figure 4.2.1.3.7.12 Memory usage by N-BEATS

4.2.1.3.7 Ensemble Learning

As seen in Table 4.2.1.3.7.7, the models that produced the most promising results out of the 6 models we have trained so far are Random Forest (RF), XGBoost (XGB), Prophet, and LightGBM (LGBM). Therefore, we decided to create an ensemble of these 4 models (Figure 4.2.1.3.8.1) and check the improvement in RMSE and RMSLE.

```
import pandas as pd
import numpy as np
from darts import TimeSeries
from darts.models import RandomForest
from darts.models import XGBModel
from darts.models import Prophet
from darts.models import LightGBMModel
from sklearn.metrics import mean_squared_error, mean_squared_log_error
```

Figure 4.2.1.3.8.1 Models merged for Ensemble Learning

We had to create two sets of time series, one for RF, XGB, and LGBM and one for Prophet (Figure 4.2.1.3.8.2). Prophet requires the date column to be set as the index of the dataset (and frequency of the time steps to be declared) whereas the rest do not.

```

df_subset = df[(df['store_nbr'] == store_nbr) & (df['family'] == family)]
df_subset_p = df_subset.copy()
df_subset_p.set_index('date', inplace=True)
ts = TimeSeries.from_dataframe(df_subset, value_cols=["sales"])
ts_p = TimeSeries.from_dataframe(df_subset_p, value_cols=["sales"], freq='d')

```

Figure 4.2.1.3.8.2 Creating two Time Series

The same steps were followed to train each model as they were trained individually earlier (Figure 4.2.1.3.8.3).

```

model_rf = RandomForest(lags=2)
model_xgb = XGBModel(lags=2)
model_p = Prophet()
model_lgbm = LightGBMModel(lags=2)

```

Figure 4.2.1.3.8.3 Instantiating the models

RMSE and RMSLE were calculated for each model for each combination of store and family. Then, the minimum values of RMSE and RMSLE were saved (Figure 4.2.1.3.8.4) to calculate the average RMSE and RMSLE later.

```

rmse = min(rmse_rf, rmse_xgb, rmse_p, rmse_lgbm)
rmsle = min(rmsle_rf, rmsle_xgb, rmsle_p, rmsle_lgbm)

```

Figure 4.2.1.3.8.4 Saving the lowest RMSE and RMSLE amongst the four models

The ensemble produced an average RMSE of 159 and RMSLE of 0.659 across 1782 models (Figure 4.2.1.3.8.5).

```

print(f"Average Root Mean Squared Error (RMSE) across all models: {avg_rmse}")
print(f"Average Root Mean Squared Logarithmic Error (RMSLE) across all models: {avg_rmsle}")

```

```

Average Root Mean Squared Error (RMSE) across all models: 159.49358307169695
Average Root Mean Squared Logarithmic Error (RMSLE) across all models: 0.6592078065208438

```

Figure 4.2.1.3.8.5 Printing the RMSE and RMSLE values

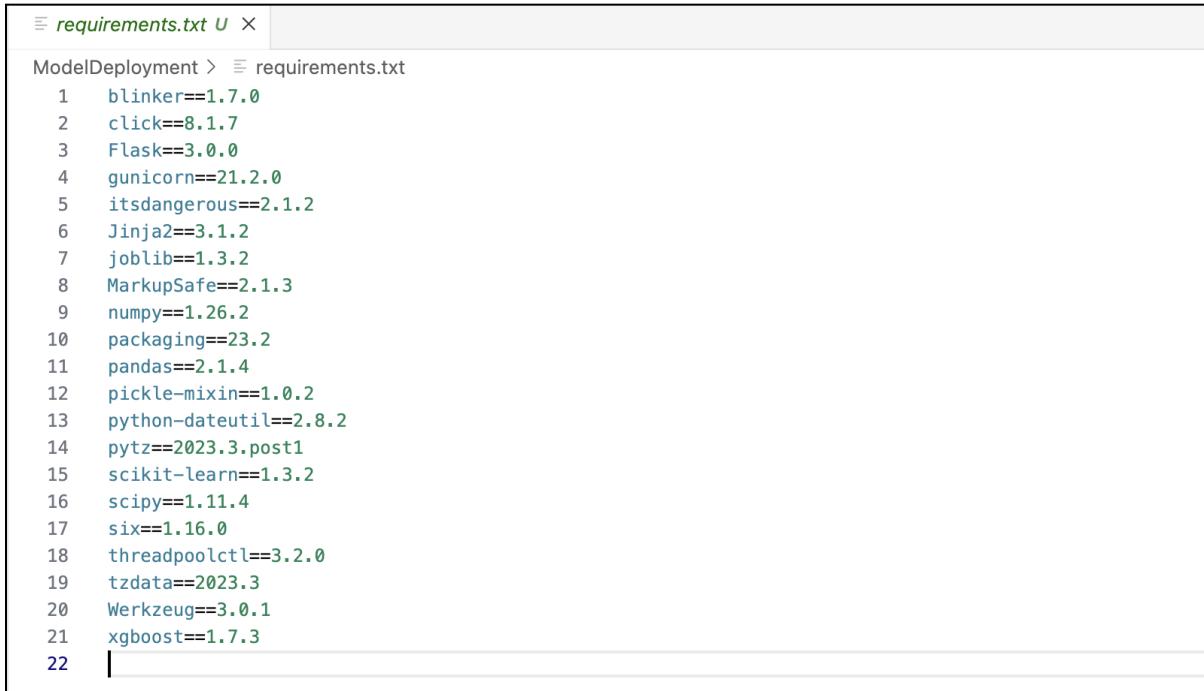
Table 4.2.1.3.8.6 shows the improvement in results shown by ensemble learning as compared to previous models.

Table 4.2.1.3.8.6 Random Forest vs XGBoost vs Prophet vs LightGBM vs Ensemble

Model	RMSE	RMSLE
Random Forest	220	0.749
XGBoost	232	0.763
Prophet	170	0.766
LightGBM	216	0.706
Ensemble	159	0.659

4.2.2 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
1  blinker==1.7.0
2  click==8.1.7
3  Flask==3.0.0
4  gunicorn==21.2.0
5  itsdangerous==2.1.2
6  Jinja2==3.1.2
7  joblib==1.3.2
8  MarkupSafe==2.1.3
9  numpy==1.26.2
10 packaging==23.2
11 pandas==2.1.4
12 pickle-mixin==1.0.2
13 python-dateutil==2.8.2
14 pytz==2023.3.post1
15 scikit-learn==1.3.2
16 scipy==1.11.4
17 six==1.16.0
18 threadpoolctl==3.2.0
19 tzdata==2023.3
20 Werkzeug==3.0.1
21 xgboost==1.7.3
22

```

Figure 4.2.2.1 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 4.2.2.2 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 4.2.2.3 Exporting all the encoders through Pickle

```
In [28]: dump(model, 'joblib/M10.joblib')

Out[28]: ['joblib/M10.joblib']
```

Figure 4.2.2.4 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 4.2.2.5 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 4.2.2.6 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 4.2.2.7 Creating feature vector X and importing the model using Joblib

```

prediction = model.predict(X)

return render_template(
    'index.html',
    prediction = prediction
)

```

Figure 4.2.2.8 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.

Demand Forecasting System

Family:

Prediction: [230.05856]

Figure 4.2.2.9 Prediction generated by the model for ‘eggs’ on the hardcoded store and date

Demand Forecasting System

Family: DAIRY

Predict Sales

Prediction: [435.4936]

Figure 4.2.2.10 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 4.2.2.11 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 4.2.2.12 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 4.2.2.13 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 4.2.2.14 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 4.2.2.15 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 4.2.2.16 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 4.2.2.17 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 4.2.2.18 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 4.2.2.19 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).

Demand Forecasting System

Family:

Select Date:

Predict Sales

Prediction: 875
Actual: 957

Figure 4.2.2.20 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/>

The screenshot shows the Digital Ocean dashboard interface. On the left, there's a sidebar with 'PROJECTS' containing a 'wasail' project and a '+ New Project' button. Below it is 'MANAGE' with options for 'Apps', 'Droplets', 'Functions', 'Kubernetes', and 'Volumes Block Storage'. The main area is titled 'hammerhead-app' and shows it's running in the 'wasail' project. It has 'Create' and 'Actions' buttons. Below that is an 'Overview' tab, which is currently selected, showing an 'APP HEALTH CHECK' section with a green checkmark and the word 'Available'. Other tabs include 'Insights', 'Activity', 'Runtime Logs', 'Console', and 'Settings'.

Figure 4.2.2.21 Deployment on Digital Ocean

5 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 existing systems like Shelf Engine and Guac which were our initial inspiration for this project. After studying their systems in detail in order to understand how they use machine learning to predict grocery store sales, we decided to do some market research by conducting interviews with grocery store owners around Lahore to understand the problems they face in their daily work life. Furthermore, we discovered mobile applications like Tajir, Dastagyr and many more that are already working in Pakistan to resolve vendor/ grocery store issues. All of this research led to the idea of developing two mobile applications, one for the vendors which would be used for managing their inventory, receiving orders from the grocery stores, and tracking orders placed by the grocery stores, and one for the grocery store owners which will help them place orders to the vendors, and resolve their overstocking or understocking issue by predicting sales for their products using machine learning. We also developed an admin portal to help us manage users and content more efficiently. Our extensive research helped us with the requirement gathering process. We initially collected data from a local vendor and a local pharmacy for our machine learning algorithms but due to insufficient number of samples we took the ‘Corporación Favorita Grocery Sales Forecasting’ dataset from Kaggle. Next, we documented our functional and non-functional requirements for the mobile applications and the admin portal as well as designed their prototypes on Figma for a better visual understanding. This helped us establish a clear roadmap for the development process. For the development part, we decided to develop the frontend of both the applications on Flutter, since it helps create apps for both Android and iOS, whereas the frontend for the admin portal was developed using React, and the backend was developed using NodeJS and ExpressJS with Sequelize as the ORM. We developed our database on MySQL. Initially, the machine learning and deep learning models were developed using Scikit-Learn and TensorFlow respectively. Finally, all the forecasting models were developed using Darts. While developing the frontend for our vendor application, we realised that the lack of responsiveness of the application is an issue because while developing it we were using the emulator to see the final results, but once we used it on our phones we realised that it does not cater to every screen size. So we made our vendor application responsive and ensured that the grocery store application was responsive as well. Moreover, we handled images on the backend. Previously, images were stored in a file inside the application on flutter, which made the applications (both vendor and grocery store) extremely heavy which meant that they required a lot of memory space on one’s mobile. So we shifted the images on the server’s file system and each image file was saved inside its designated directory. The aforementioned dataset contained 126 million samples because of which we were not able to train our models despite using services like kaggle and Google Colab. We finally used batch training which made the processing computationally feasible. We initially came up with two strategies. The first one was to train one model for the entire dataset and the second one was to train a model for each product per store which resulted in 1782 models in total. Since the second strategy required less computational power and gave a better accuracy, we decided to use the second approach. The machine learning and deep learning models we used included Random Forest, XGBoost, Prophet, Recurrent Neural Networks, LightGBM, and N-BEATS. Random Forest, XGBoost, Prophet, and LightGBM showed the most promising results. An ensemble model was created using these four algorithms that produced RMSE of 159 meaning if the model predicts that the grocery store will sell 10,000 items on a given day, the actual number of items that will be sold will typically be within the range

of 9841 to 10159 items ($10,000 \pm 159$). Then, a Flask API was developed which contained all the trained forecasting models for each registered store. This API is used to generate weekly and monthly predictions and communicate them to the backend system. The NodeJS backend system, MySQL database cluster, and the Flask API along with all the trained models were deployed on Digital Ocean. Due to limitations of disk size on the server, the deep learning models could not be deployed. In order to further enhance the user experience, local caching should be used on Flutter as it helps improve performance, reduce the load and network response times and enhance responsiveness, use Cloudflare with NodeJS to reduce response time by leveraging its global CDN, caching, DNS optimization, and performance-enhancing features. In order to make our prediction engine more suitable, we could train the forecasting models on data from local grocery stores and use transformers such as TFT. In summary, our project developed a system to address inventory management and sales forecasting for grocery stores and vendors. Through thorough research and design, we created two mobile apps and an admin portal using a robust tech stack. Despite challenges with data and responsiveness, we optimised our approach with scalable models and better performance management.

6 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. Riesegger, 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.

7 Appendices

Appendix A: Z Mart Interview Transcript

Interview conducted on 1st October 2023 at Z Mart, Falcon Complex, Lahore.

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.

Appendix B: Express Store Interview Transcript

Interview conducted on 1st October 2023 at Express Store, Gulberg 3, Lahore.

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.

Appendix C: Setting up the AI Server

In order to access the server remotely, we had to set up BNU's VPN. FortiClient was installed and the VPN settings were adjusted as guided by the ITRC team (Figure 7.1).

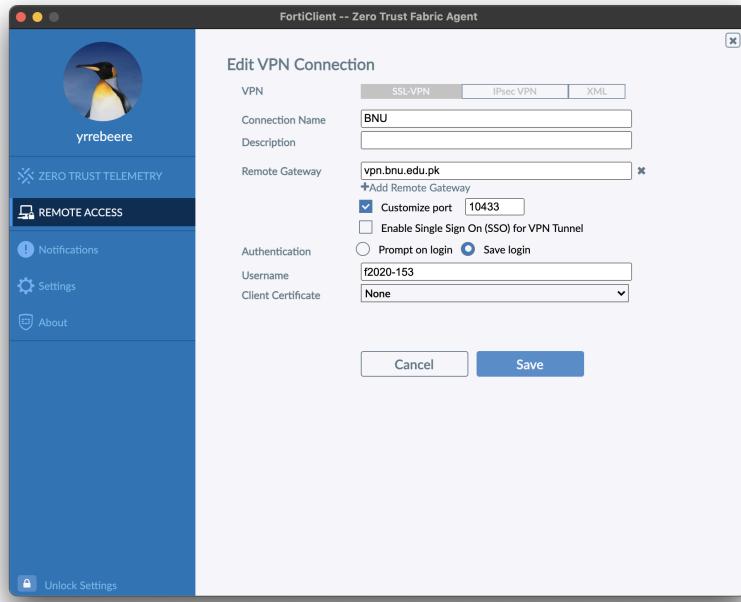


Figure 7.1 Adjusting the VPN settings on FortiClient

CMS login information (email and password) was used to establish the connection (Figure 7.2).

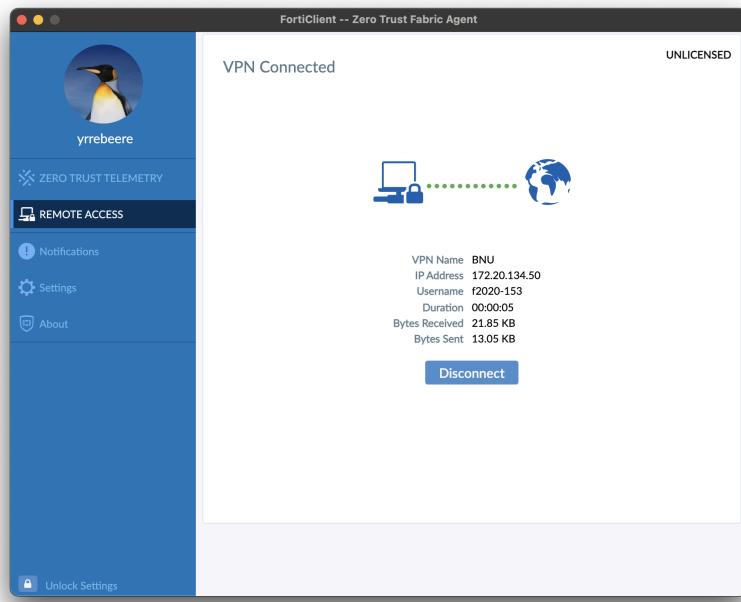


Figure 7.2 Establishing the connection

Once the vpn connection was established, Microsoft Remote Desktop application was used to establish remote desktop connection with the provided IP using Dr. Usman Nazir's credentials (Figure 7.3).

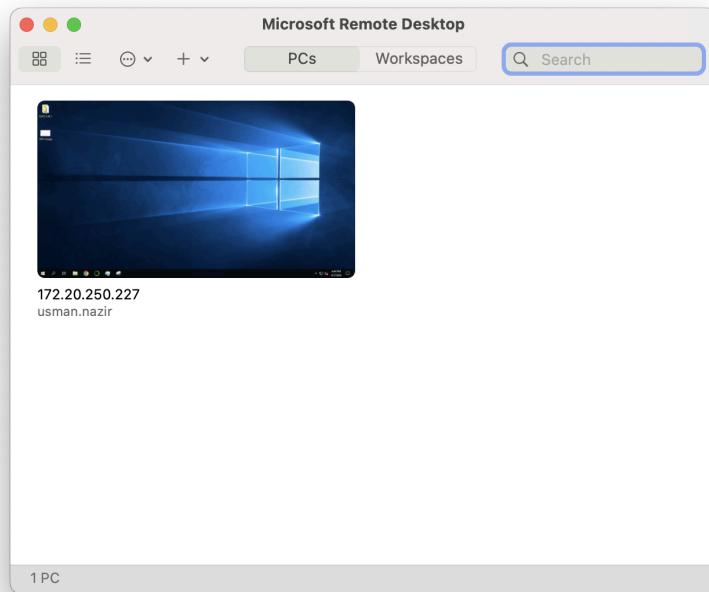


Figure 7.3 Microsoft Remote Desktop application

The server has an Intel Xeon Silver 4316 processor with 40 cores and 80 threads, 128GB physical memory, and NVIDIA GeForce RTX 4070 Ti graphics card (Figure 7.4).

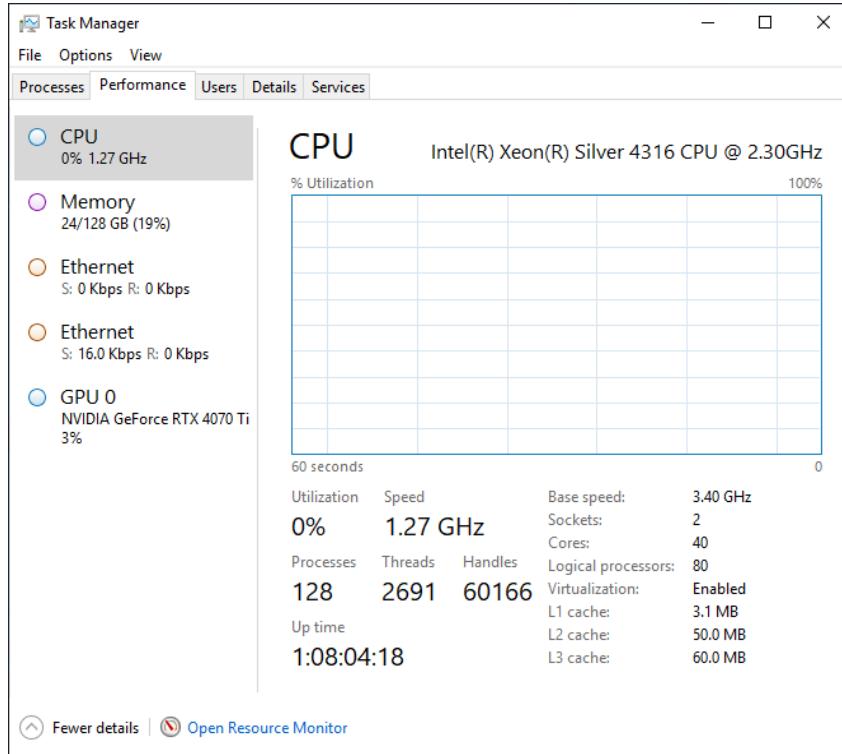


Figure 7.4 Server's specifications

We started off by setting up a new conda environment (wasail). Then, we installed all the libraries necessary to train our models including Darts, NumPy, Pandas, Pip, PyTorch, Scikit-Learn, TensorFlow etc (Figure 7.5).

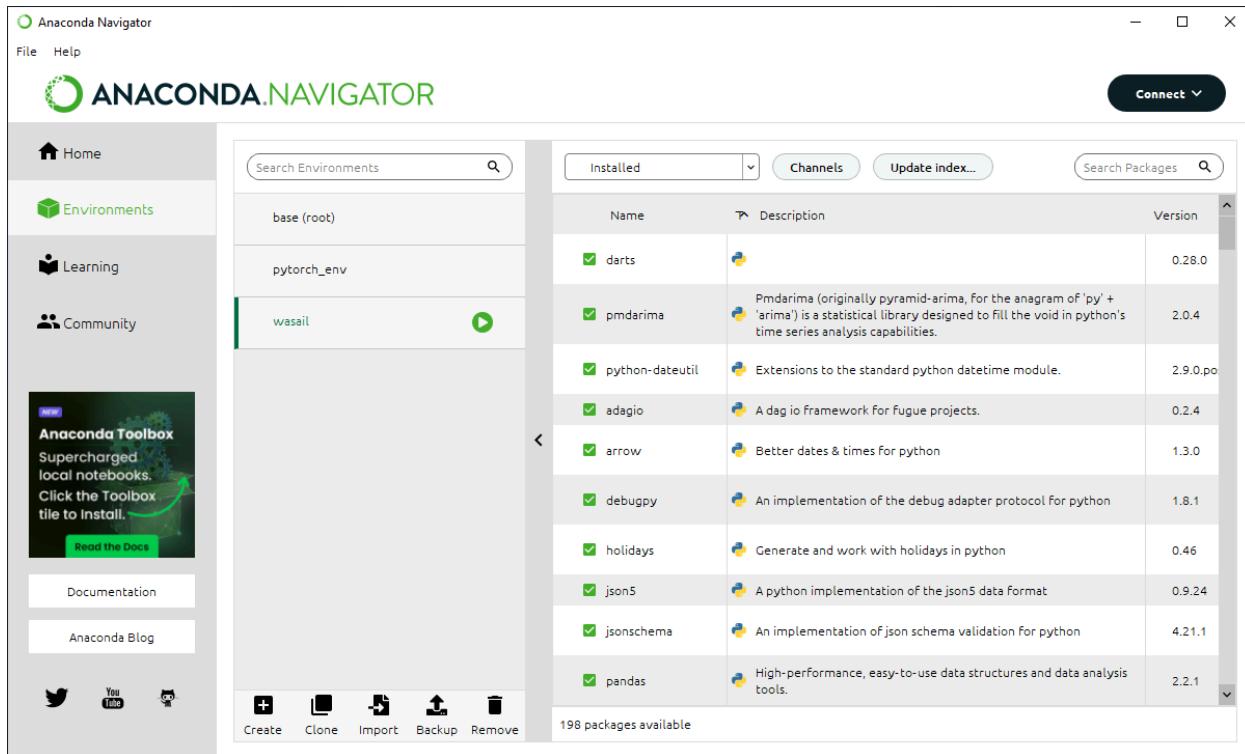


Figure 7.5 Setting up a conda environment

After setting up the conda environment, we tried to train the recurrent neural network. However, the system did not detect the GPU (GPU available: `False`, used: `False`). Upon research, we found out that in order to utilise GPUs for computation, you need CUDA. Our PyTorch environment did not have CUDA (Figure 7.6) and therefore did not utilise the GPU for training the models.

```
import torch
torch.cuda.is_available()
False
```

Figure 7.6 GPU not detected

The issue persisted after installing the latest version of CUDA. PyTorch did not support the latest version. Installing a previous version resolved the issue (Figure 7.7).

```
import torch  
  
torch.cuda.is_available()  
  
True  
  
torch.cuda.get_device_name(0)  
  
'NVIDIA GeForce RTX 4070 Ti'
```

Figure 7.7 GPU detected

Appendix D: Demo Videos

Vendor App Demo



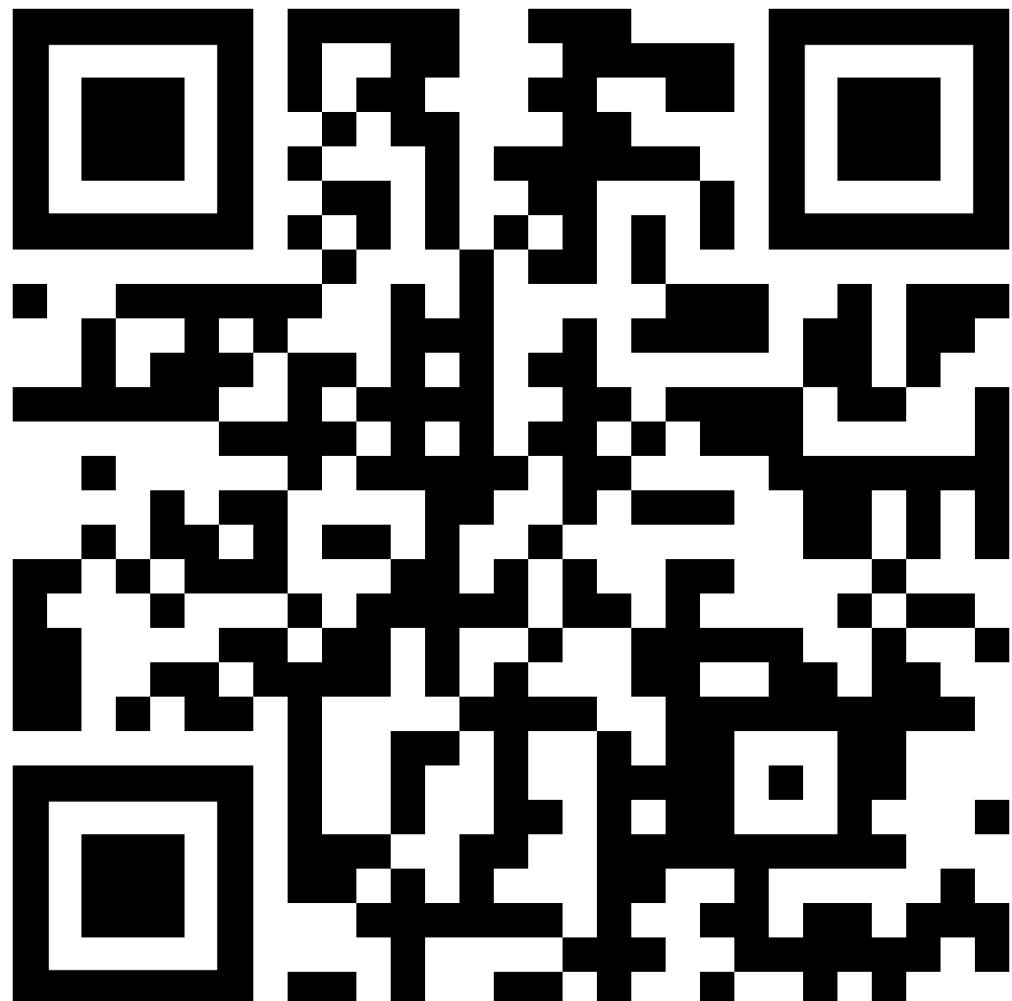
Scan the QR code or go to <https://youtu.be/0JWzBT7wbiI>

Store App Demo



Scan the QR code or go to <https://youtu.be/vqssCHEaSr8>

Admin Portal Demo



Scan the QR code or go to <https://youtu.be/t-o7tsfG9rY>

Appendix E: Posters



Scan the QR code or go to <https://github.com/yrrebeere/prj-403/tree/main/Documentation/Posters>

Appendix F: Testing and Quality Assurance



Scan the QR code or go to

<https://docs.google.com/spreadsheets/d/1fgMCJZCqgbgpCWwmZlrYROyJOGheAeRNdz71aFs9PbU/>