BEACONHOUSE NATIONAL UNIVERSITY

Wasail

PRJ-F23/333

IMPLEMENTATION DOCUMENT

EXTERNAL SUPERVISOR

Hamza Zafar

INTERNAL SUPERVISOR

Huda Sarfraz

GROUP MEMBERS

Fatima Ali Tirmizi	**F2020-718**
Fizza Adeel	**F2020-336**
Irtaza Ahmed Khan	**F2020-153**
Malaika Sultan	**F2020-661**

SCHOOL OF COMPUTER & IT

Table of Contents

Introduction

The following implementation document gives an outline of all the work done in the first phase of our final year project. It starts with explaining the development of our vendor mobile application including both front end and back end. The development of the front end started from designing the application's prototype in Figma all the way to implementing it on Flutter. Under backend, the process that started from making the database design, implementing it on MySQL to developing REST APIs on Node JS has been discussed. The test cases for all the functional requirements are also included in the document. The machine learning (ML) section shows the implementation of the shortlisted ML models for each of the available datasets, development of the Flask application, and deployment on the cloud.

App Development

This section discusses the entire cycle of developing the vendor mobile application. It has been further divided into three sections including the front end, the backend and the test cases.

Front End

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

Phase 1: Initial Front-End Design

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

User Registration (Figure 1.1 to 1.6): Designed a secure and efficient registration process, ensuring user privacy and information collection.

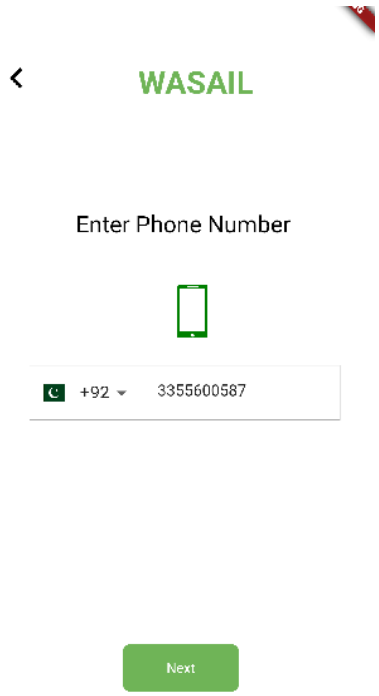


Figure 1.1

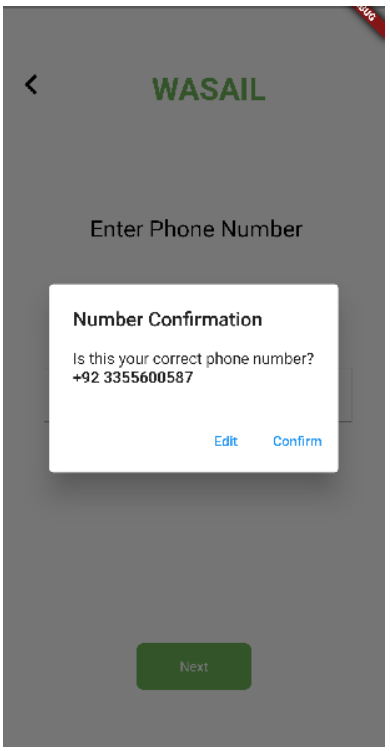


Figure 1.2

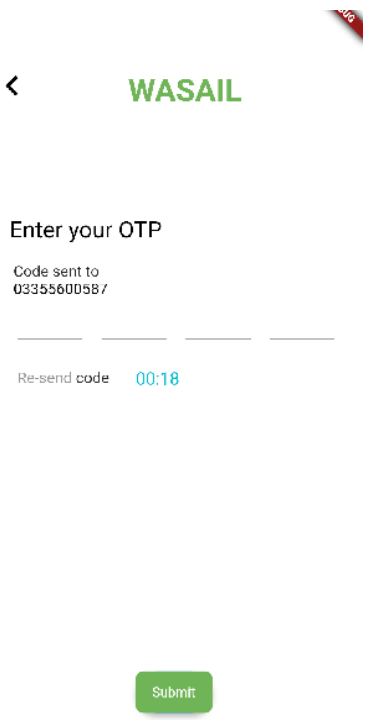


Figure 1.3

As seen in the figures above, up until the OTP process is the phone confirmation and checking process whereby new users are distinguished from existing ones. New users' phones are verified and then registered as will be seen in the sequence below. The registration comprises several validation checks including unique username validation, special characters password, password confirmation, and all filled-out fields for information required regarding vendors. Post successful registration are users taken to the login page.

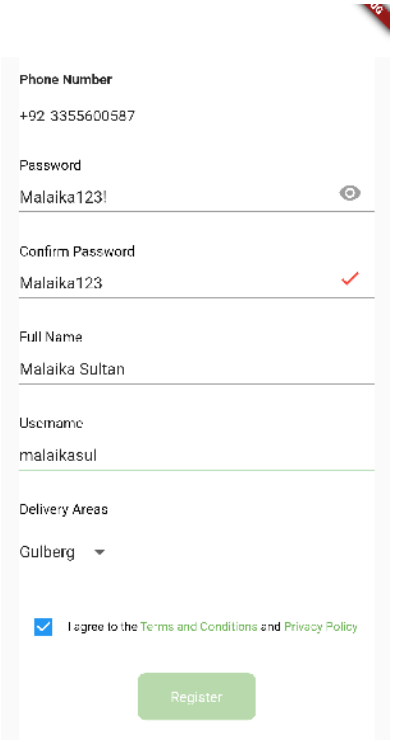


Figure 1.4

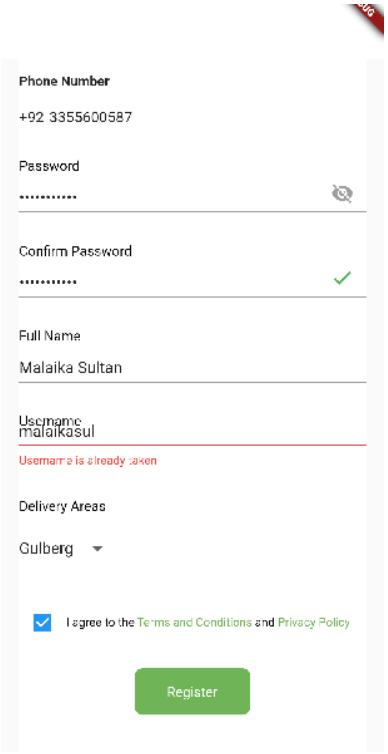


Figure 1.5

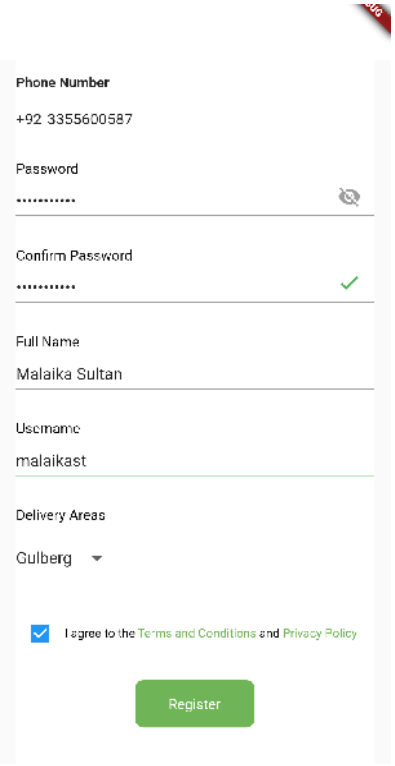


Figure 1.6

Home Page (Figure 1.7 to 1.10): Implemented a Home Page with intuitive navigation elements and key information display for an optimal user experience.

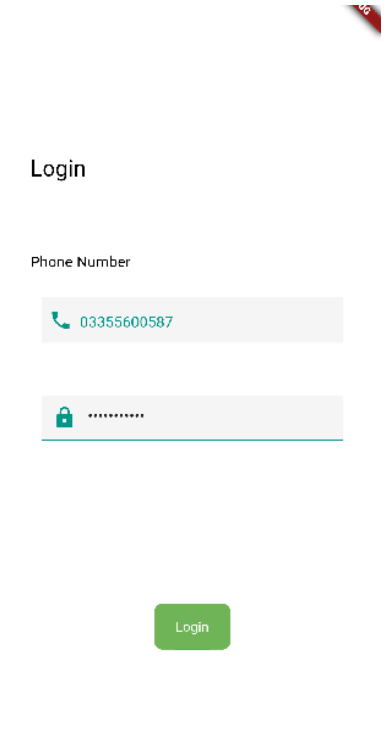


Figure 1.7

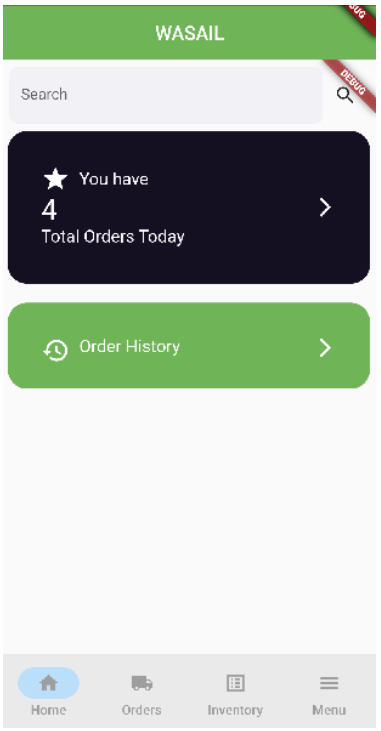


Figure 1.8

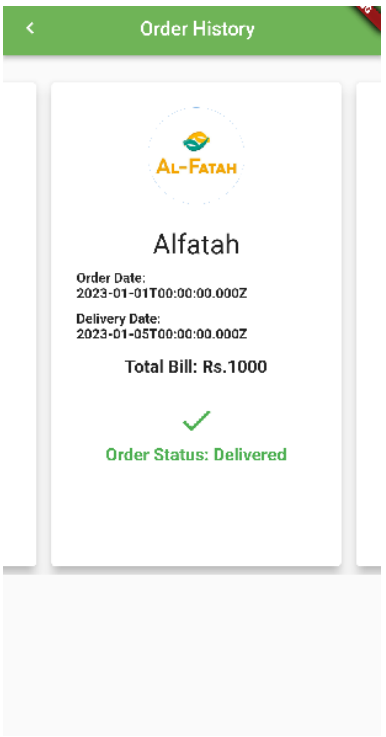


Figure 1.9

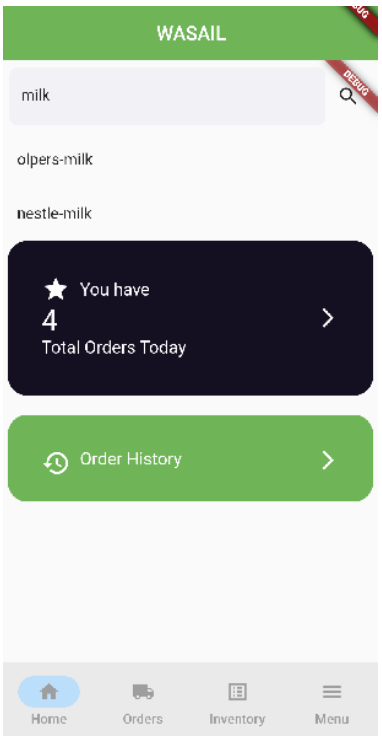
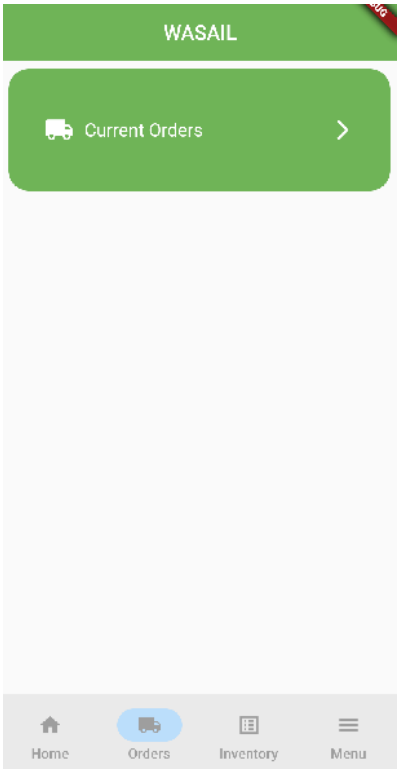


Figure 1.10

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. Furthermore, the search bar on the Home Page allows for searching in Vendors' own inventory i.e. items already present since the vendor will later have an extensive list of SKUs added, this would in quick search.

Orders Page (Figure 1.10 to 1.11): Tracked all current orders, condition 'In Process' or 'On Its Way' for order tracking and management.



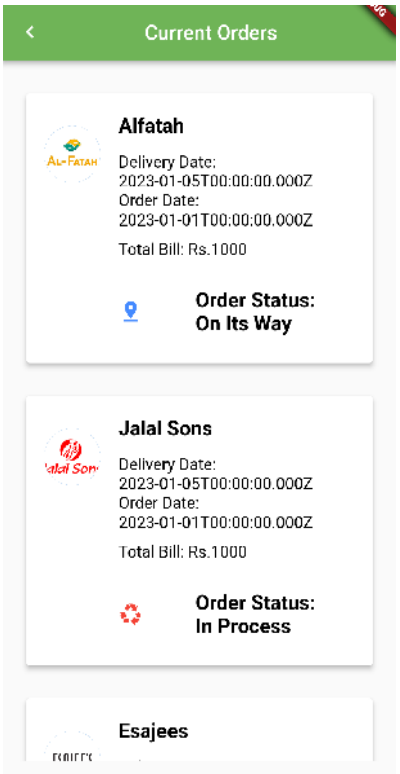


Figure 1.10

Figure 1.11

In the Orders Page, the user who is a vendor can track their current orders, which were earlier mentioned. An order will comprise the grocery store that placed the order, its order, and tentative delivery dates, as the order is still either 'In Process' or 'On Its Way', and the total bill of that order.

Inventory Page (Figure 1.12 to 1.17): Showcased listed SKUs with detailed information and provided convenient options for managing inventory.

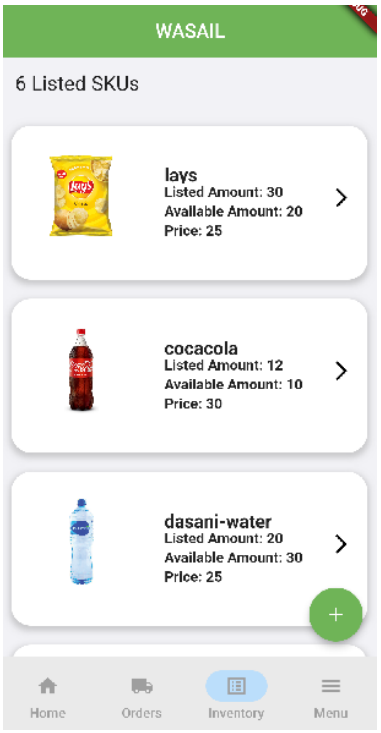


Figure 1.12

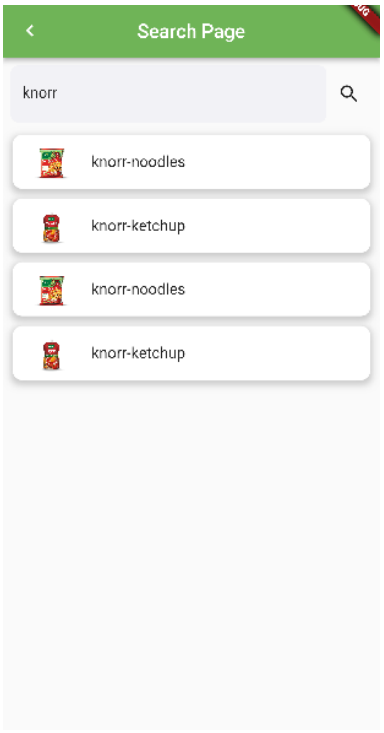


Figure 1.13

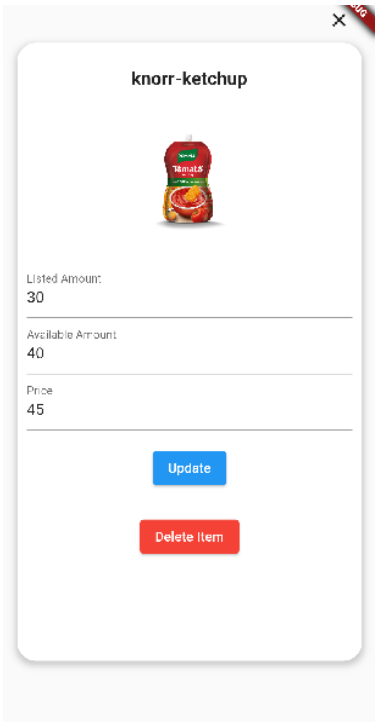


Figure 1.14

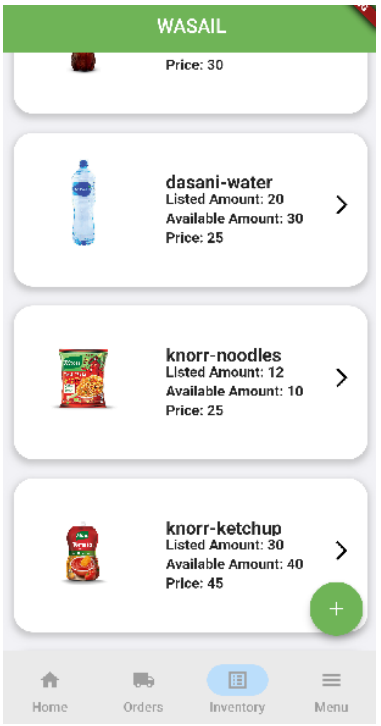


Figure 1.15

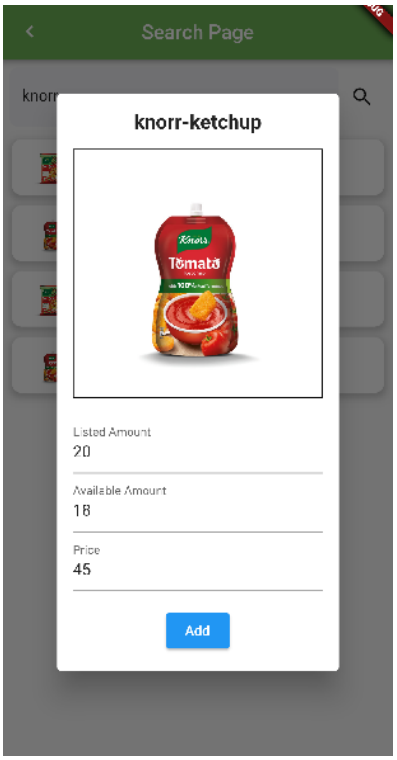


Figure 1.16

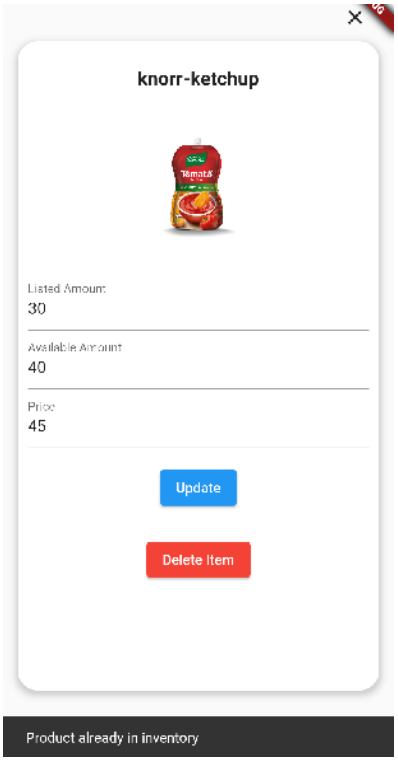


Figure 1.17

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

Menu Page (Figure 1.18 to 1.22): Created a structured layout for managing user profiles, organising the store list and app settings.

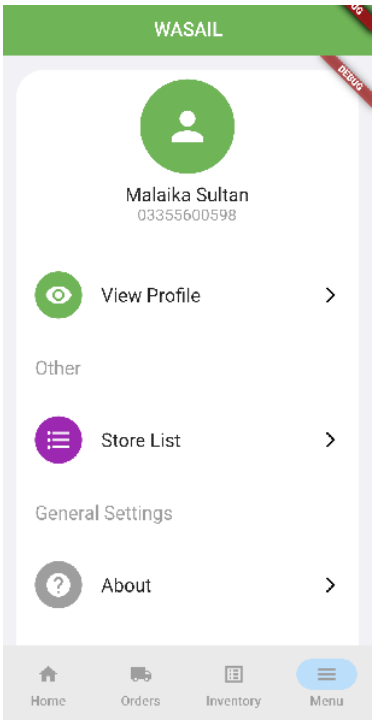


Figure 1.18

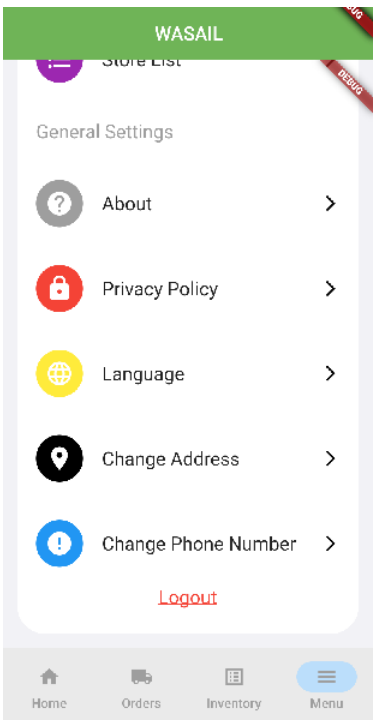


Figure 1.19

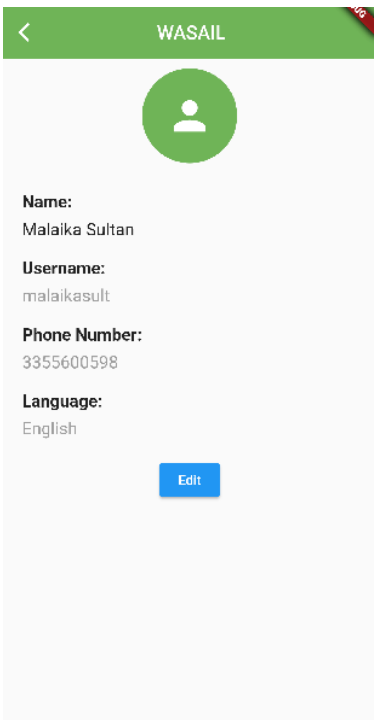


Figure 1.20

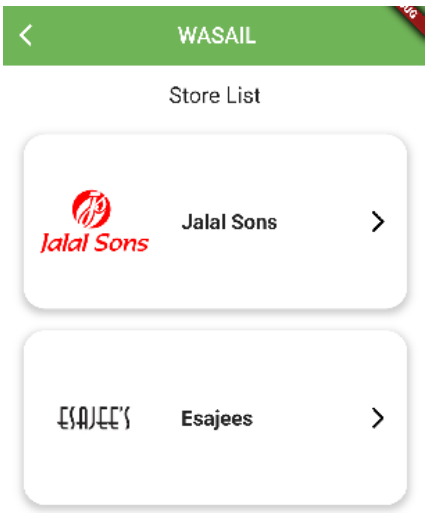


Figure 1.21

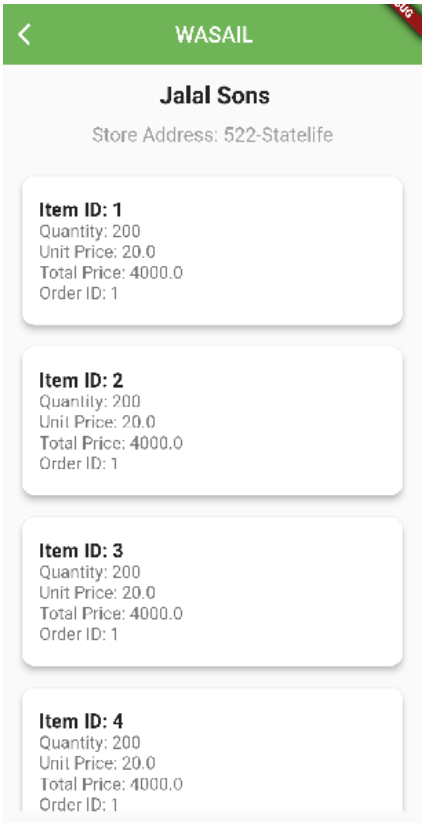


Figure 1.22

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

Phase 2: Mobile 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.

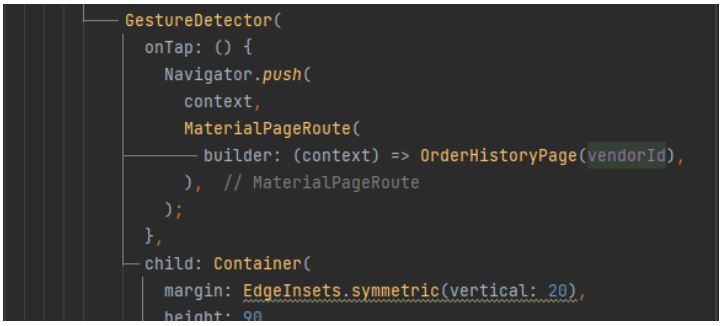


Figure 1.23 showing Gesture Detector Usage for accessing Order History in the Home Page

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

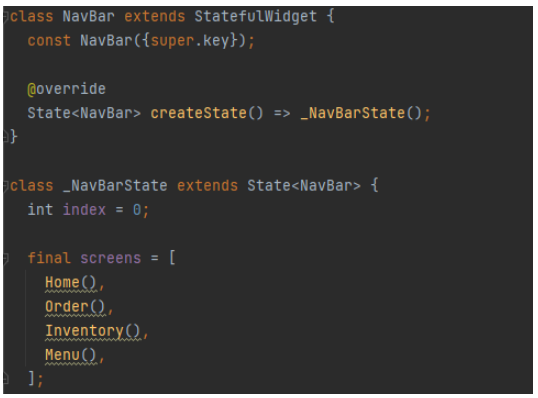


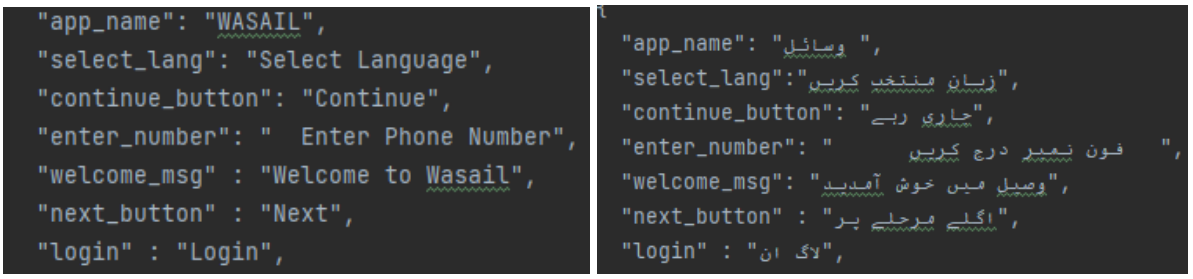
Figure 1.24 shows the navigation bar using a stateful widget to handle navigation.

Phase 3: Front-End Enhancement for User Experience

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

- **Language Selection, Internationalisation, and State Management (Figure 1.25 to 1.31):** Enabled users to navigate the app in their preferred language using Flutter's localization features.

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



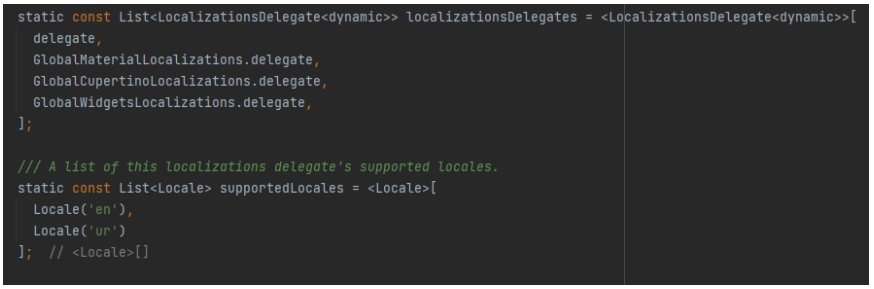
```
"app_name": "WASAIL",
"select_lang": "Select Language",
"continue_button": "Continue",
"enter_number": "  Enter Phone Number",
"welcome_msg" : "Welcome to Wasail",
"next_button" : "Next",
"login" : "Login",
```

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

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

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

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



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

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

Figure 1.28 shows declarations of localisation needed to internationalise a page

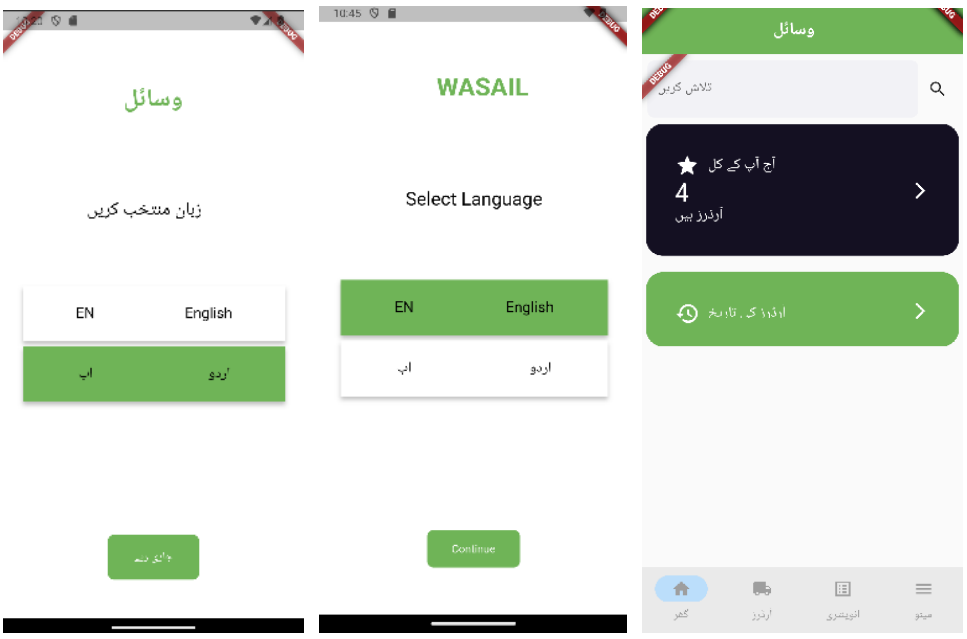


Figure 1.29 shows an overview of the app in Urdu

Finally, 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’

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

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

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

    return MaterialApp(

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

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

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

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

- **Media Query and Responsiveness (Figure 1.32 to 1.33):** Allows the application to handle responsiveness across various devices.

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.

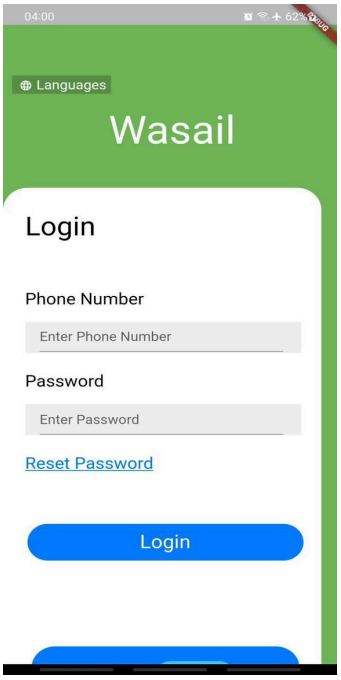


Figure 1.32 shows the Login Page in the early stage being unresponsive on a different device

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 below demonstrates:

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

Figure 1.33 shows the code of MediaQuery implementation

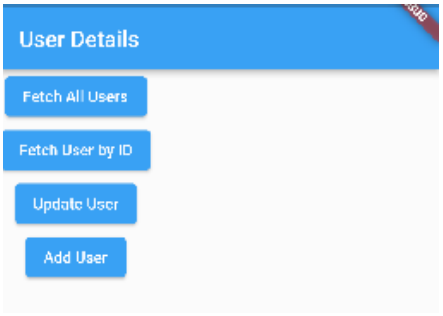
- **Visual Personalization - Dark-Bright Mode:** Conducted experiments with dark-bright mode functionality.

We briefly experimented with theme variation as shown in one of the weekly submissions by having a dark-light mode toggle in the app, however for FYP I, we decided to leave it out.

Phase 4: Front-End and Back-End Integration

In the final 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.



Integrating required use of imports, libraries of Flutter, as seen in Figure 1.34 below, which were used in JSON encoding and decoding as well as HTTP protocol functions; GET, UPDATE, etc.

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

Figure 1.34 shows which imports were used for integration

The operations looked as below in Figure 1.35 where Uri are the endpoints where information is being handled in the backend/database.

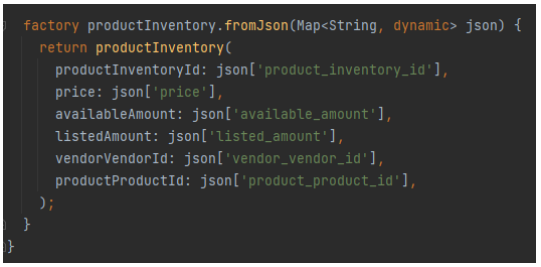
```
Future<void> _fetchAndDisplayProducts(String query) async {
  final response = await http.get(
    Uri.parse('http://10.0.2.2:3000/api/product/searchproduct/$query'),
  );

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

Figure 1.35 shows a 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 1.36 is decoding an item from inventory and Figure 1.37 is 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'],
  );
}
}
```

Figure 1.36



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

Figure 1.37

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.

Back End

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

Database Design

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

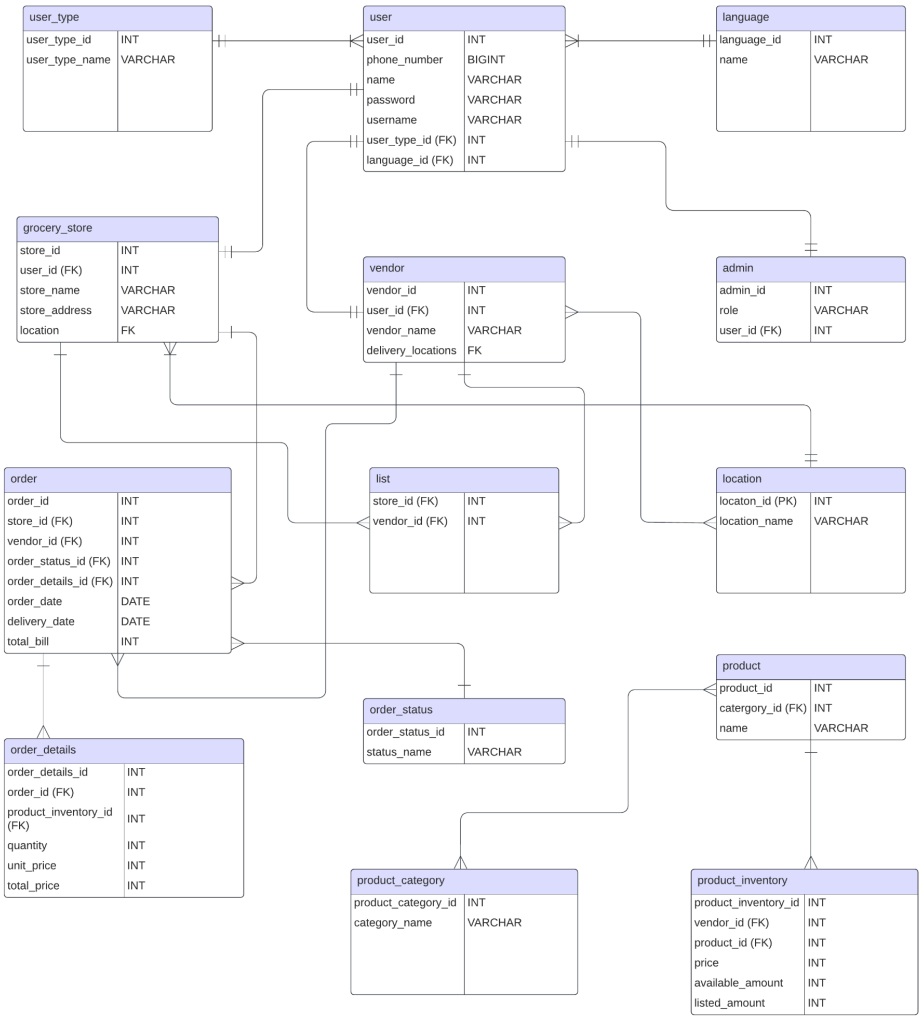


Figure 1.0 First iteration of the database model

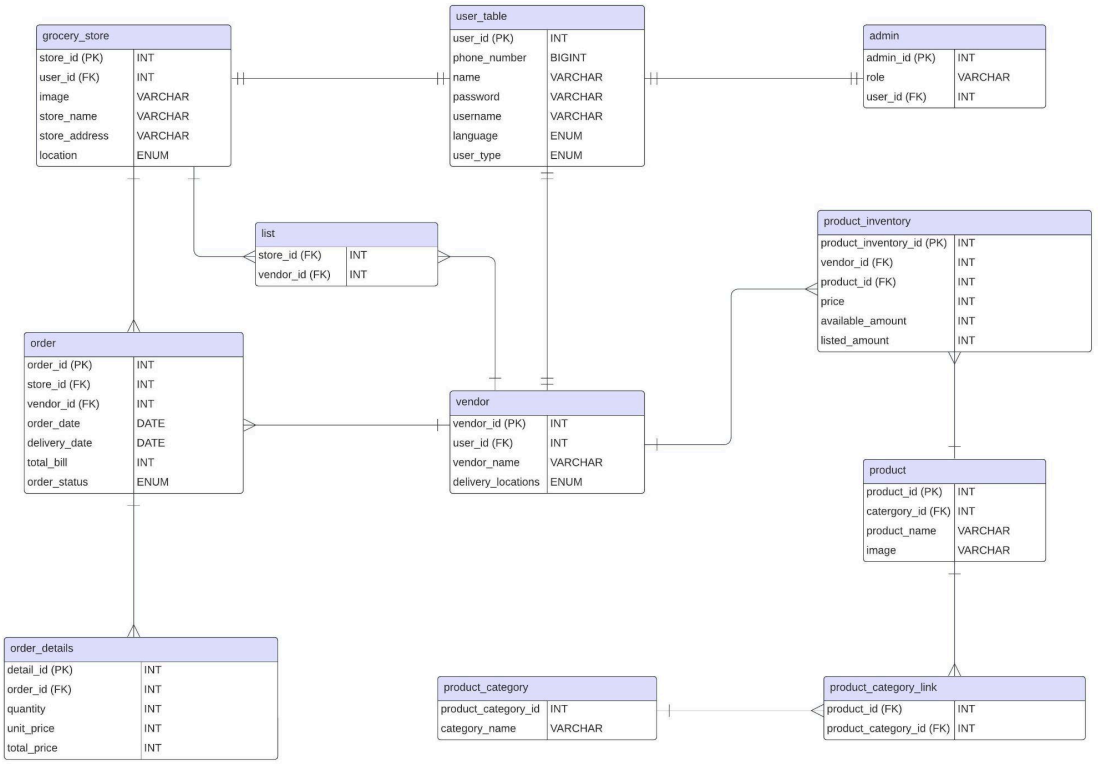


Fig 1.1 Last iteration of database model

Spring Boot

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

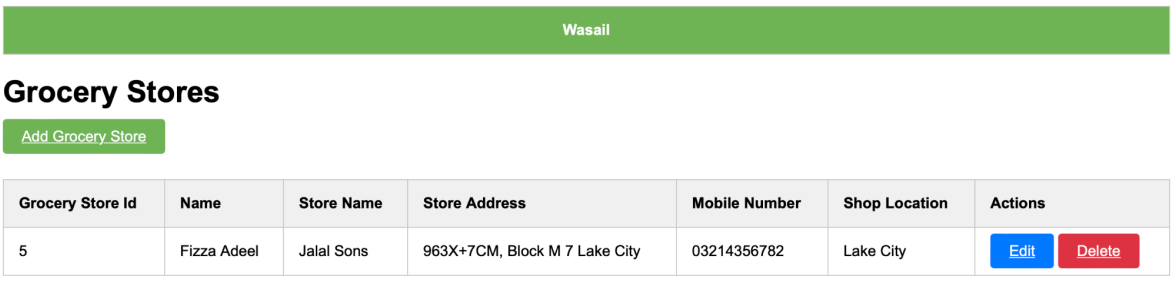


Figure 1.2 Implementation using Spring Boot

Installation of Sequelize

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

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

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

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

Figure 1.3 Installation of Sequelize

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

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

Figure 1.4 Installation of MySQL2

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

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

Figure 1.5 Installation of sequelize-cli

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

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

Figure 1.6 Dependencies installed for the project

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

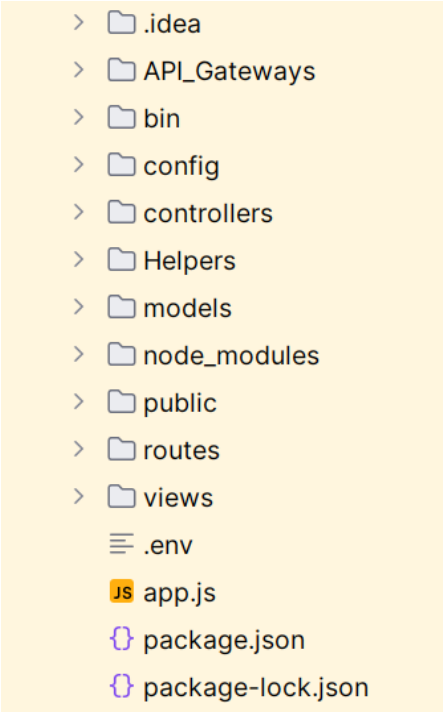


Figure 1.7 Empty files created by sequelize

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

User Model Implementation

```
'use strict';

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

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

Figure 1.8 User table model created

Next, the first model was created using sequelize in the models folder. Since the user table was the first table created in the database diagram, the first model that was created was the user table as well. The model which had userId, phone number, name password, username, language and user type. The aforementioned documentation was referred to when defining the variables. Since we had to ensure that the variables language and user type do not have values other than the ones seen in figure 1.8, the data type used for them was ENUM which helped us define the fixed values for these specific variables. Moreover, some of the validation checks implemented on the variables for example isAlpha, isAlphanumeric etc were also included for variables like username and password which were also implemented with the help of the documentation. Additionally, sequelize creates two variables ‘Created At’ and ‘Updated At’ on its own for all the tables. Since these are auto generated, these two variables have not been explicitly defined in the model class but have been given values in the database.

Implementation of ENV

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

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

Figure 1.9 .env file

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

Figure 2.0 Installation of .env file

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

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

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

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

Figure 2.1 Config file

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

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

module.exports = db;
```

Figure 2.2 Error handling in index file

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

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

Figure 2.3 Sync function

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

Vendor Model Implementation

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

```
'use strict';

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

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


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

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

Figure 2.4 Implementation of vendor model

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

- Sequelize tutorial

Implementation of Controller

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

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

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

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

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


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

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

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

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

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

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

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

Figure 2.5 Implementation of user controller

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

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

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

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

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

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



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

Figure 2.5.1 Implementation of product controller

Figure 2.5.1 shows a part of the product controller. Here the two important functions search product in inventory and search product have been shown respectively. For example in the

search product in inventory function the vendor id and the product name that is searched are being passed. It uses the vendor id and finds its corresponding product inventory id, then uses that product inventory id and finds its corresponding product id and based on the product id and the product name that has been searched fetches all the products and sends them as a response.

Implementation of Relations

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

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

Figure 2.6 Importing the models in the index file

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

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

Figure 2.7 One to one relation

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

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

Figure 2.8 One to many relation

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

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

Figure 2.8 Many to many relation

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

Implementation of Routes

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

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

Figure 2.9 Implementation of user routes

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

Figure 2.9.1 Implementation of product routes

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

Implementation of Routes in App

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

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

Figure 3.0 Implementation of main routes

Postman Testing

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

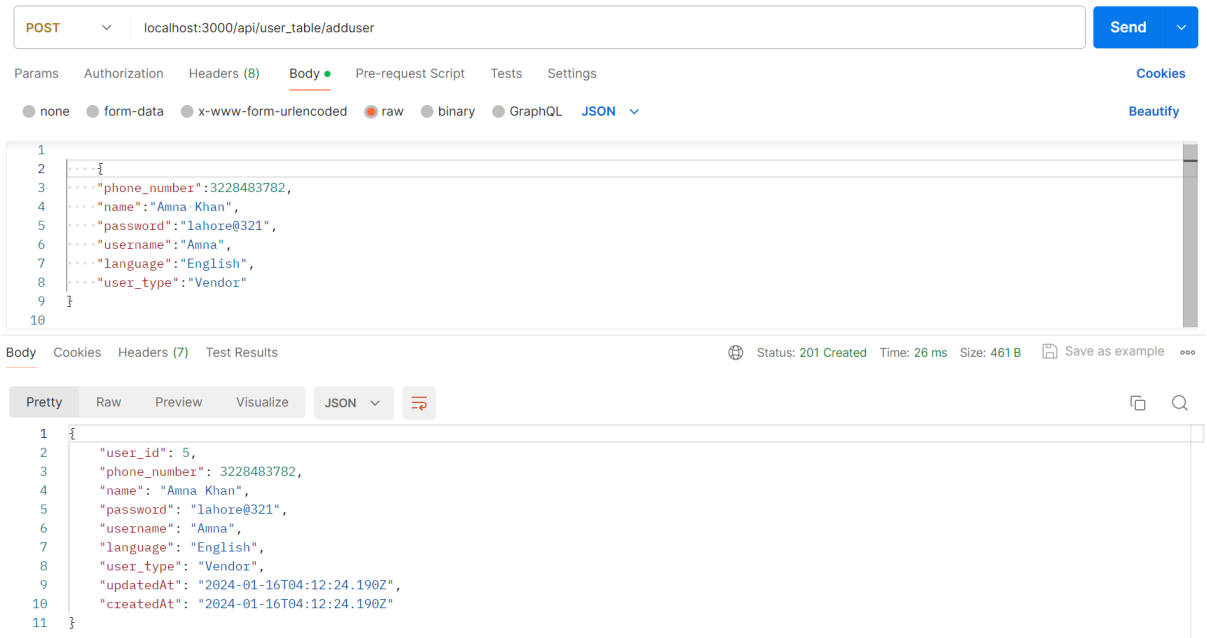


Figure 3.1 Adding a user

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

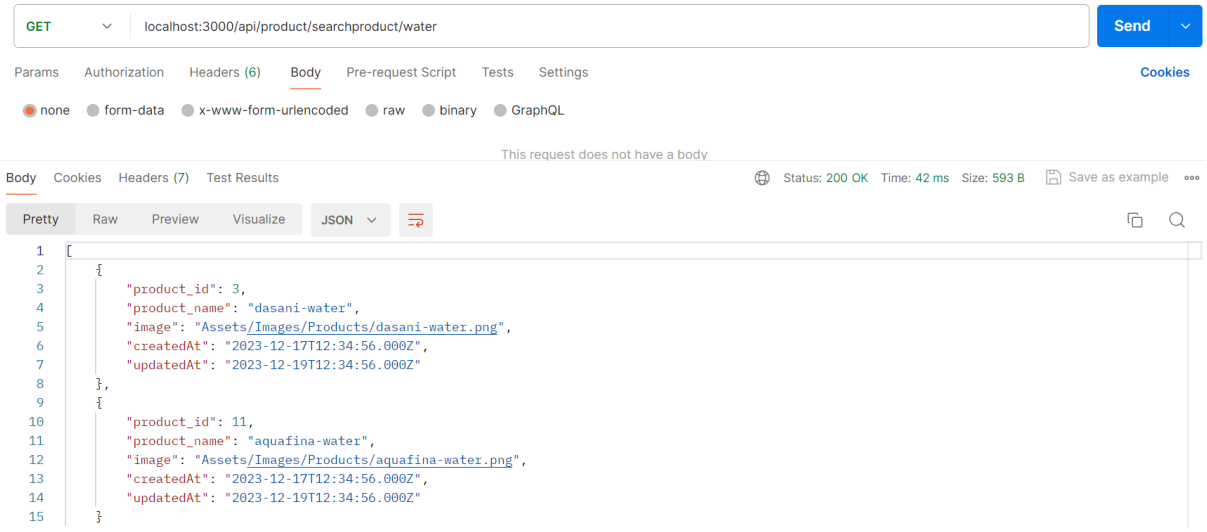


Figure 3.2 Searching a product

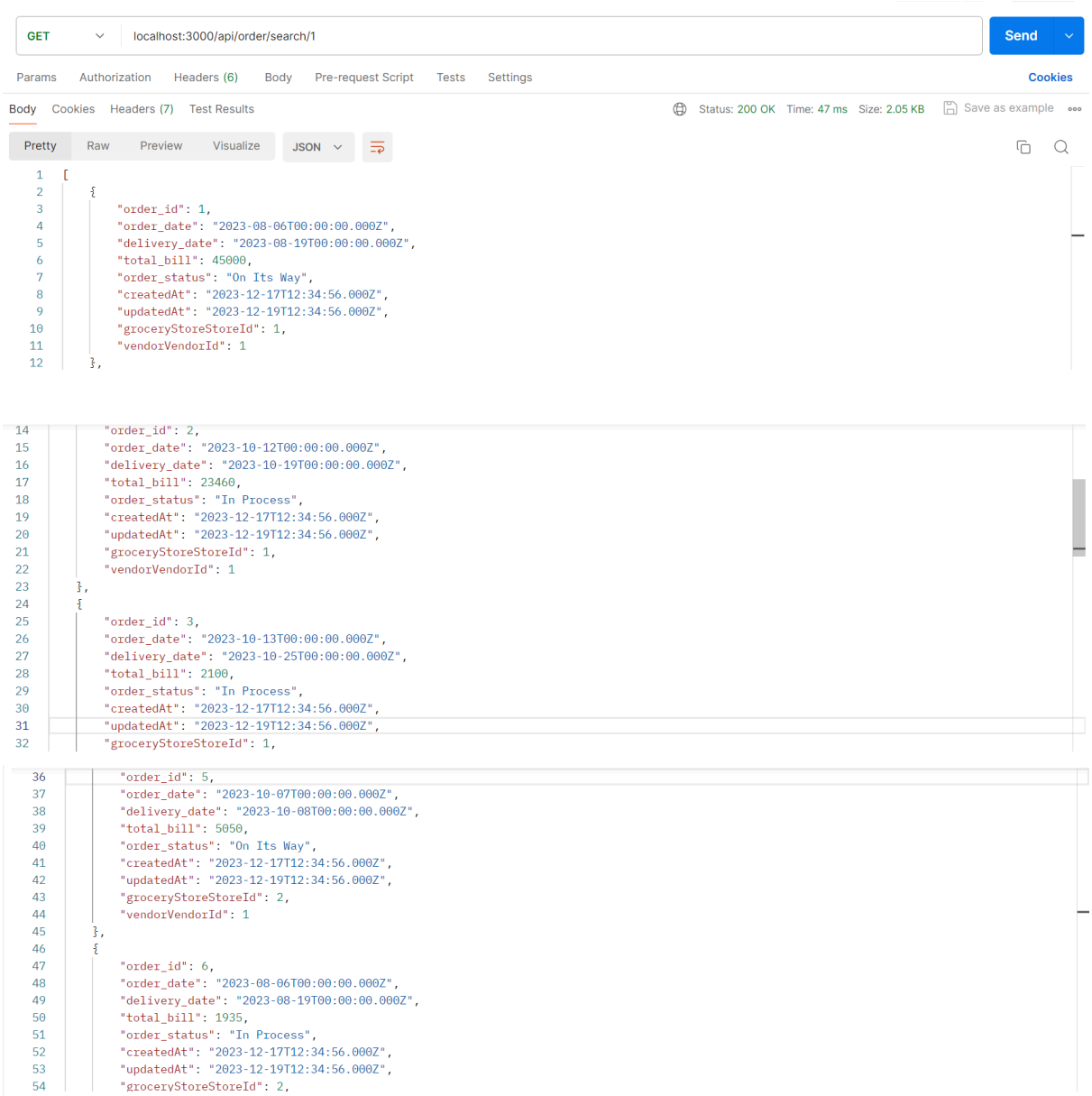


Figure 3.3 Current Orders

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

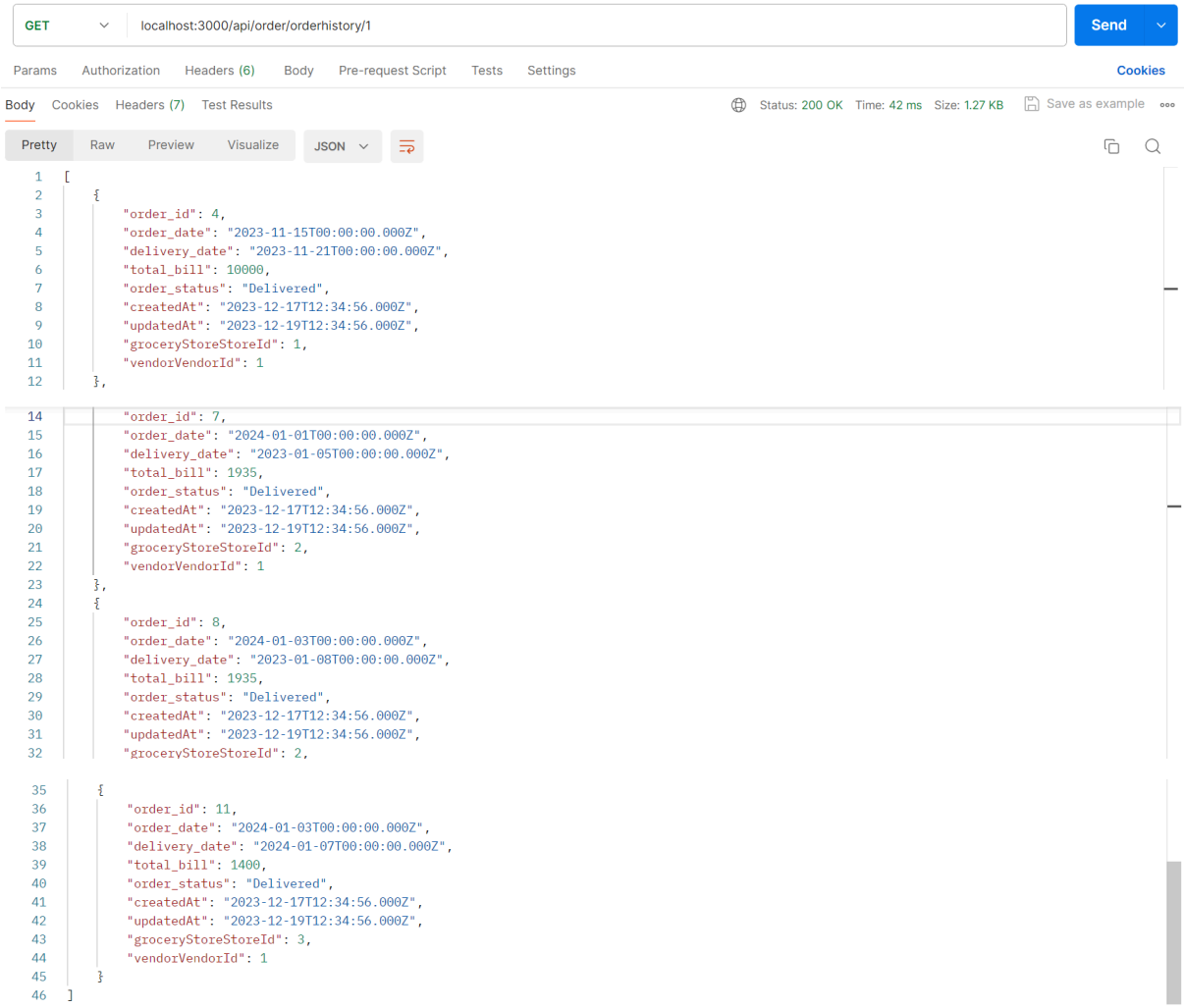


Figure 3.4 Order History

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

Conclusion

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

Test Cases

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

Test Case 1: Language Selection

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

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

Test Case 2: Phone Registration

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

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

Test Case 3: Phone Number Confirmation

Test Scenario: The user wants to confirm their entered phone number.
Preconditions: The user has entered their phone number.
Postconditions: The phone confirmation dialogue box should open with options to edit and/or confirm.

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

Test Case 4: Phone Number Exists

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

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

Test Case 5: OTP Code Generation and Delivery

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

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system can confirm the phone number of the user.	3355600598	The phone number should be confirmed by the system.	The phone number is confirmed by the system.	Pass
2	The system generates an OTP code and		A 4-digit unique OTP code should be generated and sent.	A 4-digit unique OTP code is not generated and sent.	Fail

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

Test Case 6: Login

Test Scenario: Registered users want to log in using their credentials.
Preconditions: The user is on the login page.
Postconditions: The user logged in.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The user	lahore123	The system, over an	The system asks for a	Pass

	enters their password		incorrect password, asks for a re-entry	re-entry.	
		lahore@1	The system, over a correct password, should log the user in.	The user is logged in.	Pass

Test Case 7: Logout

Test Scenario: The system should allow the user to logout.
Preconditions: The user is logged in and on the menu.
Postconditions: The user is logged out.

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

Test Case 8: Reset Password

Test Scenario: The system should allow the user to reset their password.
Preconditions: The user is registered.
Postconditions: The user's password is successfully reset.

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

Test Case 9: View Profile

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

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

Test Case 10: Vendor Registration

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

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

Test Case11: Valid Password

Test Scenario: The user should enter a password.

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

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

Test Case 12: Username Exists

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

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

Test Case 13: Search Product in Inventory

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

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system directs the user to the		The user is on the home page	The user successfully navigates to the home page	Pass

	homepage where the search bar is				
2	The user can search product name from their own inventory	Olpers	The system allows the user to enter the search criteria	The system allows the user to enter the search criteria	Pass
3	The system fetches results for searched product by name	Olpers	The system retrieves product matching to name criteria	The system does not retrieve product matching to name criteria	Pass

Test Case 14: Add Product to Inventory

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

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

	the addition to the user inventory.		be displayed under the SKUs list on the inventory page with its details	displayed under the SKUs list on the inventory page with its details.	

Test Case 15: All Product Search

Test Scenario: The user should be able to search for a product from the product listings.
Preconditions: The user is on the products search page.
Postconditions: The has searched the product.

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

Test Case 16: Remove Product

Test Scenario: The user should remove a product from their inventory.
Preconditions: The user is on the product's page.
Postconditions: The user has removed a product.

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

Test Case 17: Edit Details of Product

Test Scenario: The user wants to edit the details of a product in their inventory.
Preconditions: The user has added the product to their inventory.
Postconditions: The user has edited the product.

#	Test Steps	Test Data	Expected Result	Actual Result	Status
1	The system displays the product inventory page for the user.	Coca Cola Lays lays water	All the products in vendor inventory should be displayed.	All the products in vendor inventory are displayed.	Pass
2	The system gives the user the option to edit the product details: listed amount, available amount, and price.	Selected Product: Coca Cola Old details: Price: 120 Listed: 28 Available: 23	The current details of the product, price, listed amount, and the available amount in the system should be visible when the edit dialog opens.	The current details of the product, price, listed amount, and the available amount in the system are visible when the edit dialog opens.	Pass
3	The user edits the product details over the old details.	Selected Product: Coca Cola New details: Price: 120 Listed: 35 Available: 28	The product details should be editable over the current ones displayed.	The product details are editable over the current ones displayed.	Pass
4	The system saves the edited product details.	Coca Cola Details: Price: 120 Listed: 35 Available: 28	The details should be saved and updated in the inventory.	The details get saved and updated in the inventory.	Pass

Test Case 18: View Inventory

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

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

	their inventory displayed by the system.				

Test Case 19: View Current Orders

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

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

Test Case 20: View Grocery Store List

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

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

Test Case 21: View Grocery Store Profile

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

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

Test Case 22: View Grocery Store Current Order

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

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

Test Case 23: View Order History

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

#	Test Steps	Test Data	Expected Result	Actual Result	Status

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

Test Case 24: Edit Profile

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

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

Test Case 25: Restrict Product Duplication

Test Scenario: The system should restrict duplication of products.

Preconditions: The user attempts to add an existing product to their inventory.
Postconditions: The system prevents the addition of duplicate products and alerts the user.

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

Machine Learning

Introduction

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

Data Collection

Covered in Requirement Analysis Document.

- Software Requirement Specification
    - Machine Learning Requirements
        - Data Collection

Feature Engineering

Covered in Requirement Analysis Document (Theoretical) and Design Document (Practical).

- Requirement Analysis Document
    - Software Requirement Specification
        - Machine Learning Requirements
            - Feature Engineering

- Design Document
    - Machine Learning Design
        - Feature Engineering

Model Shortlisting

Covered in Requirement Analysis Document (Initial Research) and Design Document (Final Selection).

- Requirement Analysis Document
    - Software Requirement Specification
        - Machine Learning Requirements
            - Model Shortlisting

- Design Document
    - Machine Learning Design
        - Model Shortlisting

Training and Testing

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

Local Pharmacy Dataset

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

Pharmacy Dataset Update Glossary

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

Figure 1.0.1 Pharmacy Dataset Update Glossary

Random Forest

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

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

Figure 1.1.1 Importing necessary libraries

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

```
df.head()
```

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

Figure 1.1.2 Loading the dataset and inspecting it

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

```
df.shape
```

```
(213056, 6)
```

Figure 1.1.3 Shape of the dataframe

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

Figure 1.1.4 Separating features and label

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

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

```
X.shape
```

```
(213056, 7477)
```

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

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

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

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

Figure 1.1.6 Splitting data into training set and testing set

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

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

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

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

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

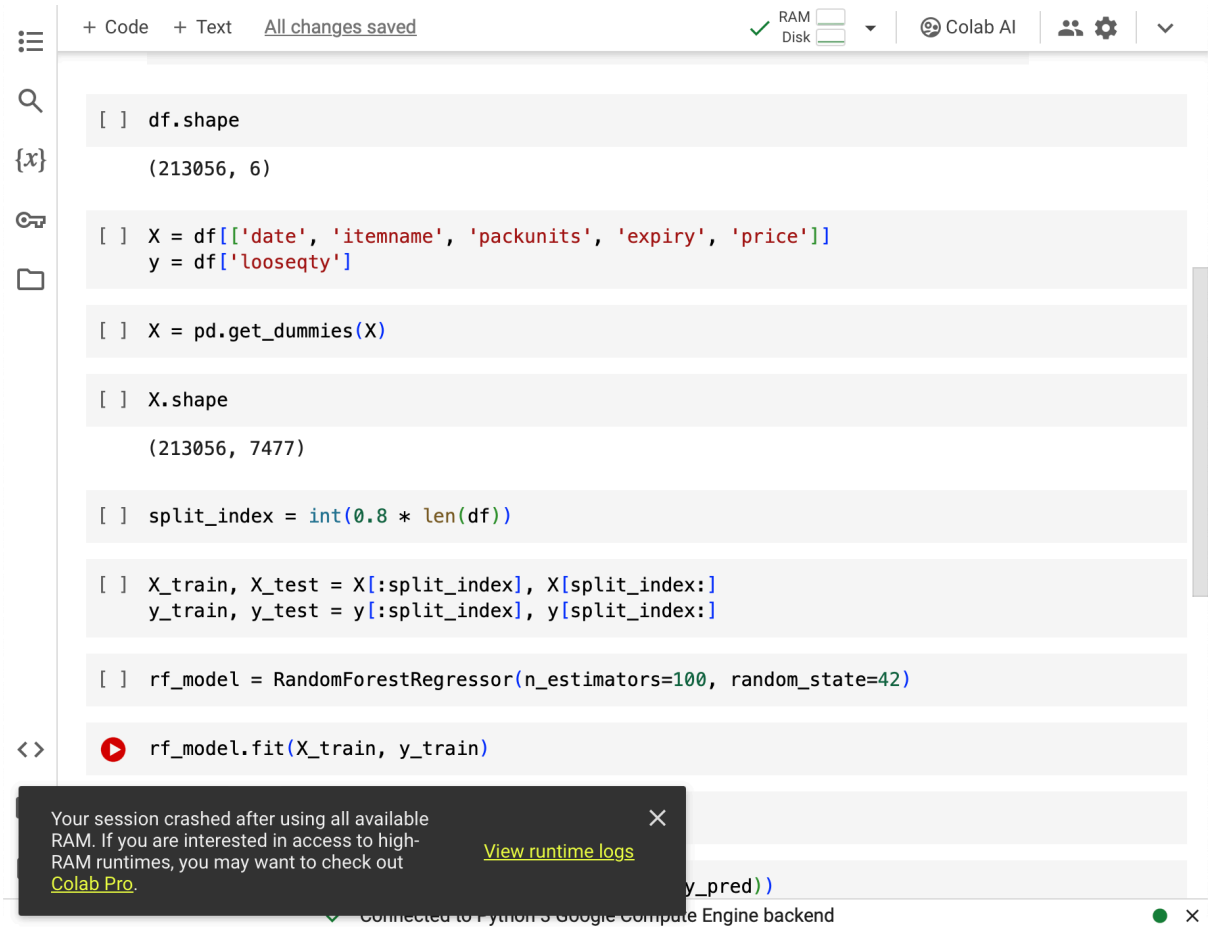


Figure 1.1.8 Google Colab crashing

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

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

```
df.head()
```

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

Figure 1.2.1 D5.csv used instead of D4.csv

```
df.shape
```

```
(337, 6)
```

Figure 1.2.2 Shape of the dataframe

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

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

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

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

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

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

Figure 1.2.4 Printing the RMSE and RMSLE values

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

```
y_test.shape
```

```
(68,)
```

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

Figure 1.2.5 Converting y_test to a normal np array

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

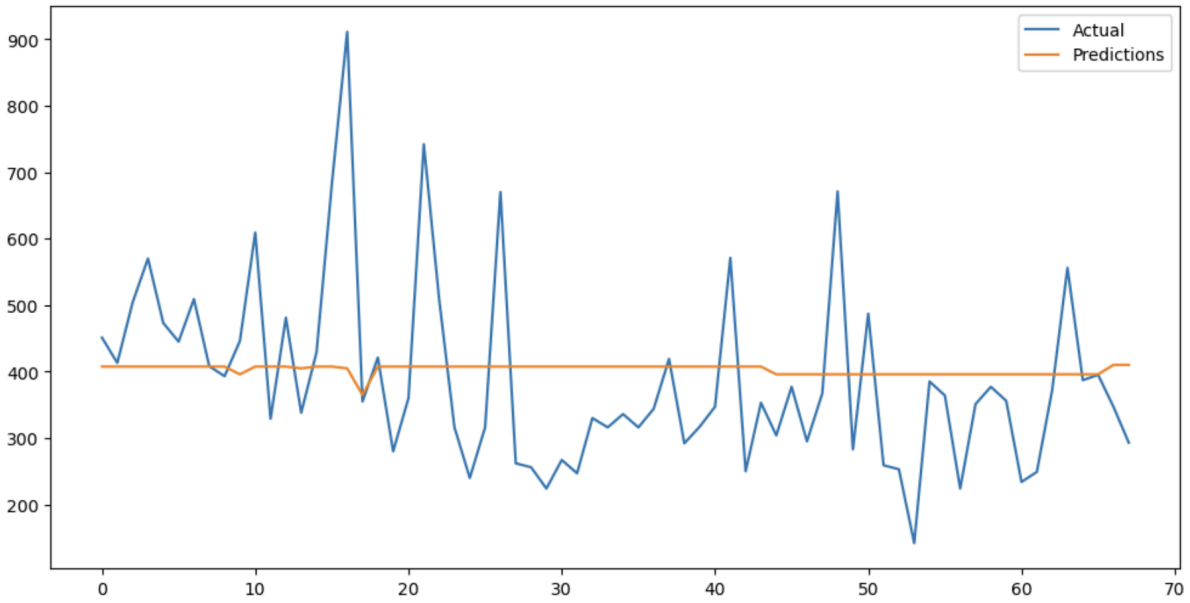


Figure 1.2.6 Actual values vs predicted values

XGBoost

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

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

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

Figure 1.3.1 Training a XGBoost model

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

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

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

Figure 1.3.2 Printing the RMSE and RMSLE values

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

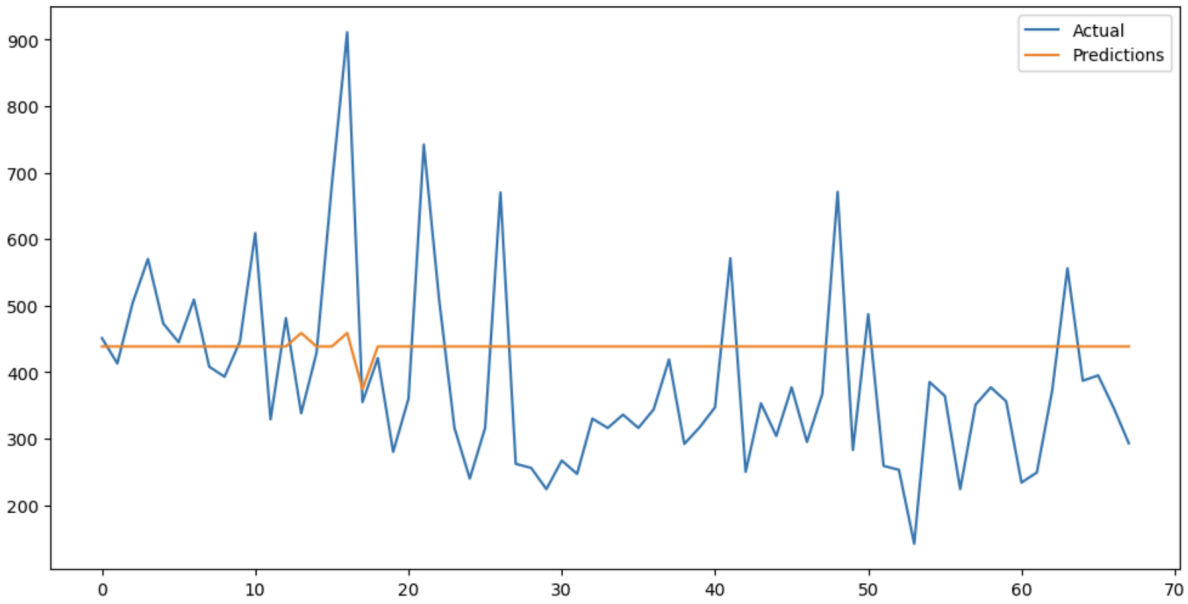


Figure 1.3.3 Actual values vs predicted values

Prophet

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

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

Figure 1.4.1 Importing necessary libraries

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

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

Figure 1.4.2 Renaming date and looseqty columns for Prophet

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

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

Figure 1.4.3 Creating and training the Prophet model

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

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

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

Figure 1.4.4 Defining time period for predictions

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

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

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

Figure 1.4.5 Printing the RMSE and RMSLE values

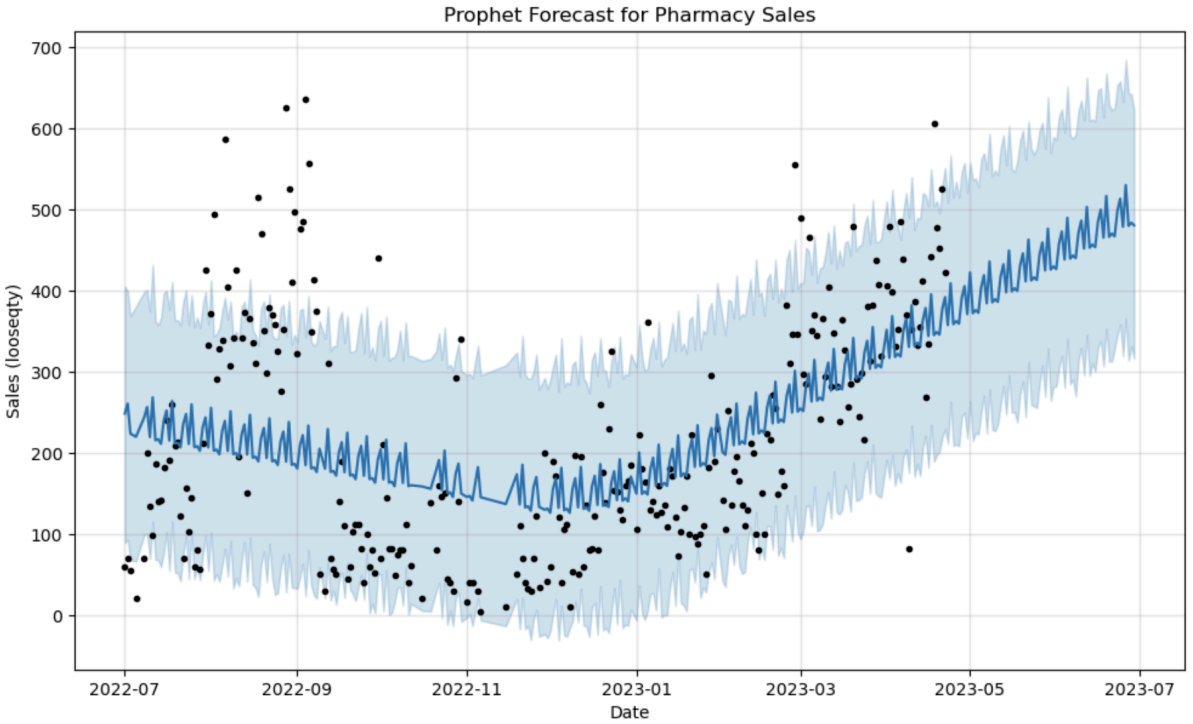


Figure 1.4.6 Prophet model's plot

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

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

```
6053
```

Figure 1.5.1 Unique products in the Pharmacy dataset

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

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

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

```
top_10_products
```

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

Figure 1.5.2 Finding the top 10 products

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

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

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

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

        forecast = model.predict(future)

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

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

        predicted_values = np.maximum(predicted_values, 0)

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

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

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

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

Figure 1.5.3 Training models for the top 10 products

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

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

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

Figure 1.5.4

Recurrent Neural Networks

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

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

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

Figure 1.6.1 Creating a univariate dataframe

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

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

```
sequence_length = 10
```

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

Figure 1.6.2 Generating sequences for training the model

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

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

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

Figure 1.6.3 Creating and training the LSTM model

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

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

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

Figure 1.6.4 Printing the RMSE and RMSLE values

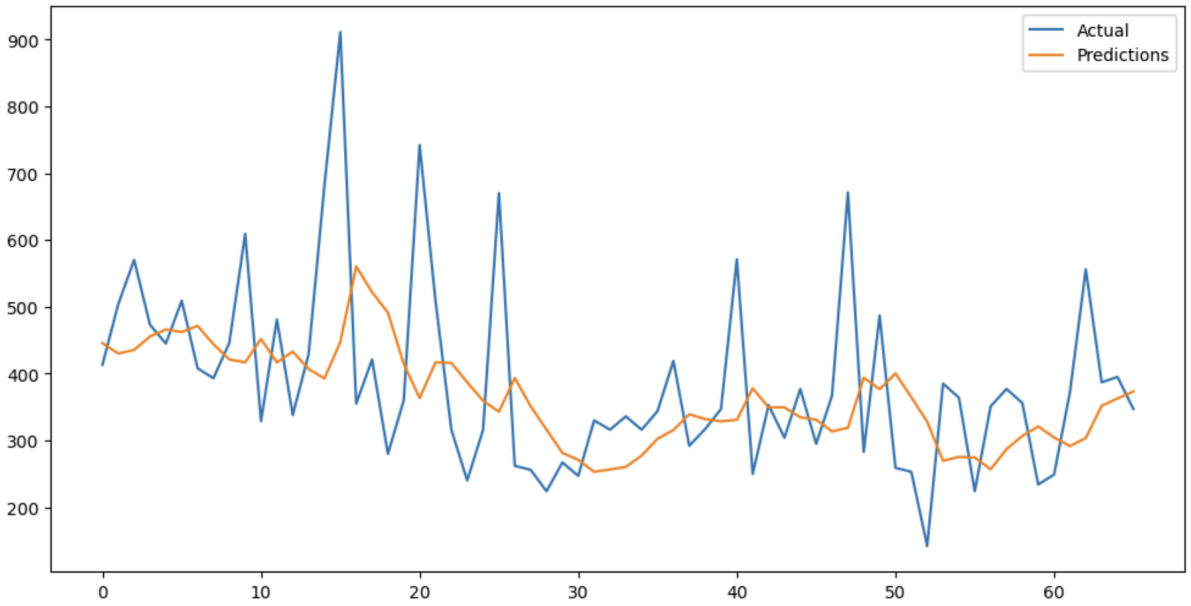


Figure 1.6.5 Actual values vs predicted values

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

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

Figure 1.7.1 Creating and training the GRU model

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

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

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

Figure 1.7.2 Printing the RMSE and RMSLE values

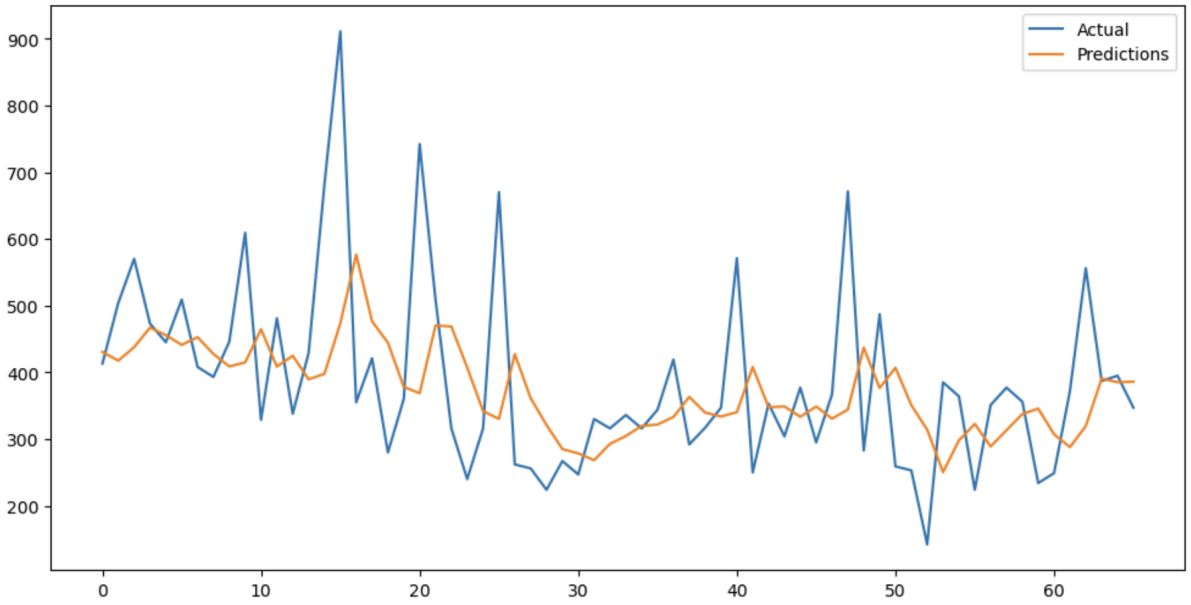


Figure 1.7.3 Actual values vs predicted values

Summary

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

Model	RMSE	RMSLE
Random Forest	137	0.338
XGBoost	145	0.371
Prophet	160	0.401
LSTM	139	0.330
GRU	137	0.325

Figure 1.8.1 Model Summary

Corporación Favorita Grocery Sales Forecasting

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

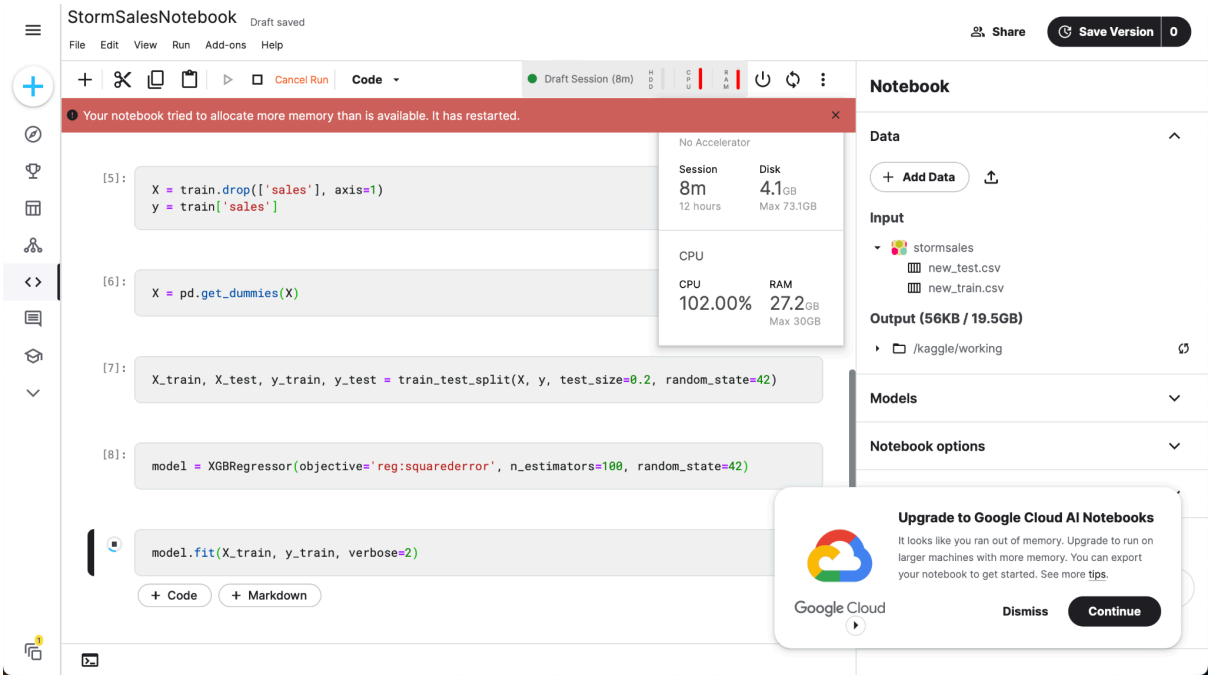


Figure 2.1.1 Kaggle crashing

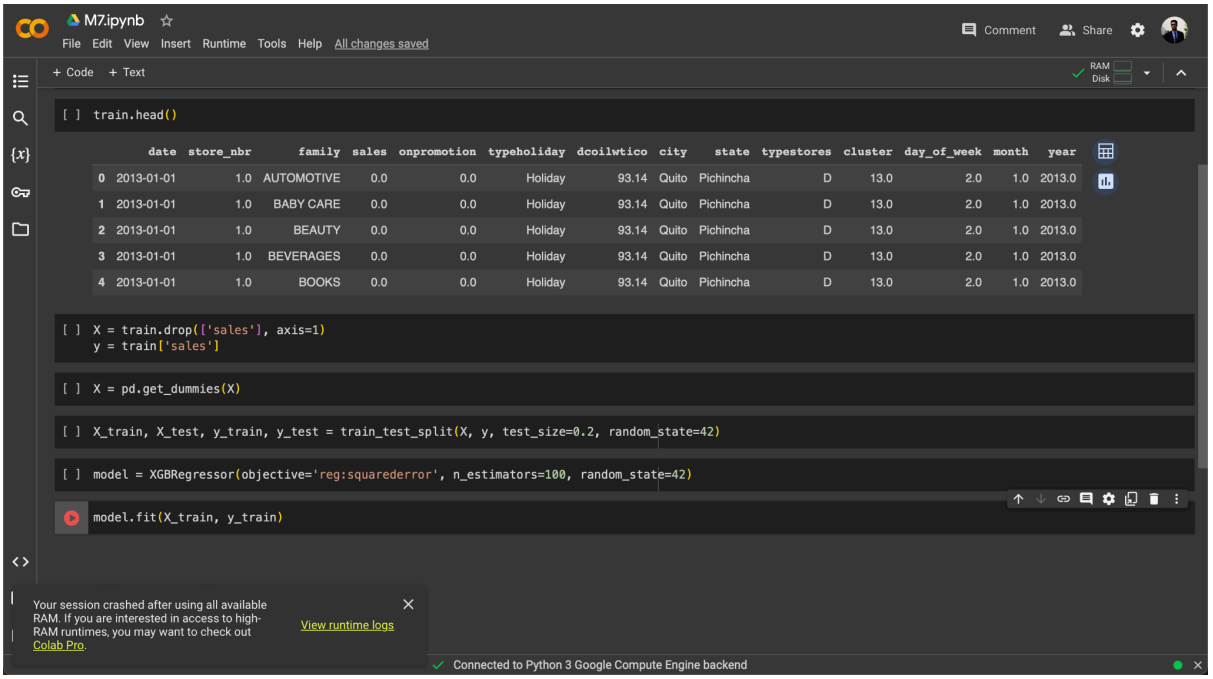


Figure 2.1.2 Google Colab crashing

During the initial exploration of the Corporación Favorita dataset, we kept the date feature as seen in the train.head() in Figure 2.1.2. The preprocessing was done on the available datasets as mentioned in the Design Document. While making predictions on the unseen

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

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

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

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

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

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

Figure 2.1.3 Combining train and test dataset to apply get_dummies together

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

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

Out[8]:

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

5 rows × 1791 columns

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

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

Figure 2.1.4 Information about the updated train dataset

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

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

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

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

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

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

    model.fit(X_batch, y_batch)

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

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

Figure 2.1.4 Batch Training

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

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

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

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

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

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

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

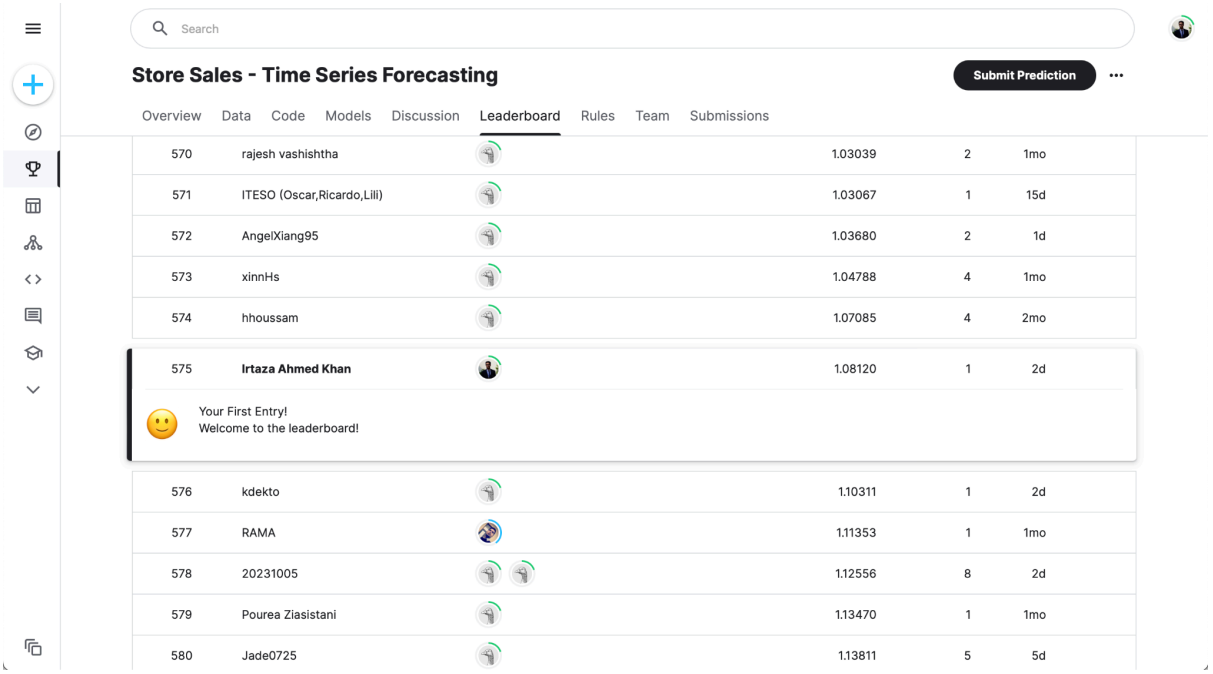


Figure 2.1.6 Store Sales - Time Series Forecasting Leaderboard

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

Random Forest

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

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

```
df.head()
```

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

Figure 2.2.1 Loading the dataset and inspecting it

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

and year) this time, it did not increase the dimensionality to the same extent it did in the pharmacy dataset.

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

```
X.shape
```

```
(3000888, 90)
```

Figure 2.2.2 Converting categorical attributes using get_dummies

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

```
batch_size = 32
```

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

Figure 2.2.3 Setting batch size and calculating number of batches

We created and trained a RF model. Considering the size of the dataset, batch training was used (Figure 2.2.4). The dataset was divided in batches of 32 samples. The total number of batches were calculated to cover the entire training dataset (which in this case were 75023). Lastly, the model is trained one batch at a time.

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

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

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

    model.fit(X_batch, y_batch)

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

```
Batch 1/75023 completed
```

Figure 2.2.4 Using batch training to train the RF model

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

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

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

Figure 2.2.5 Printing the RMSE and RMSLE values

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

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

Figure 2.2.6 Plotting 100 values from actual and predicted values

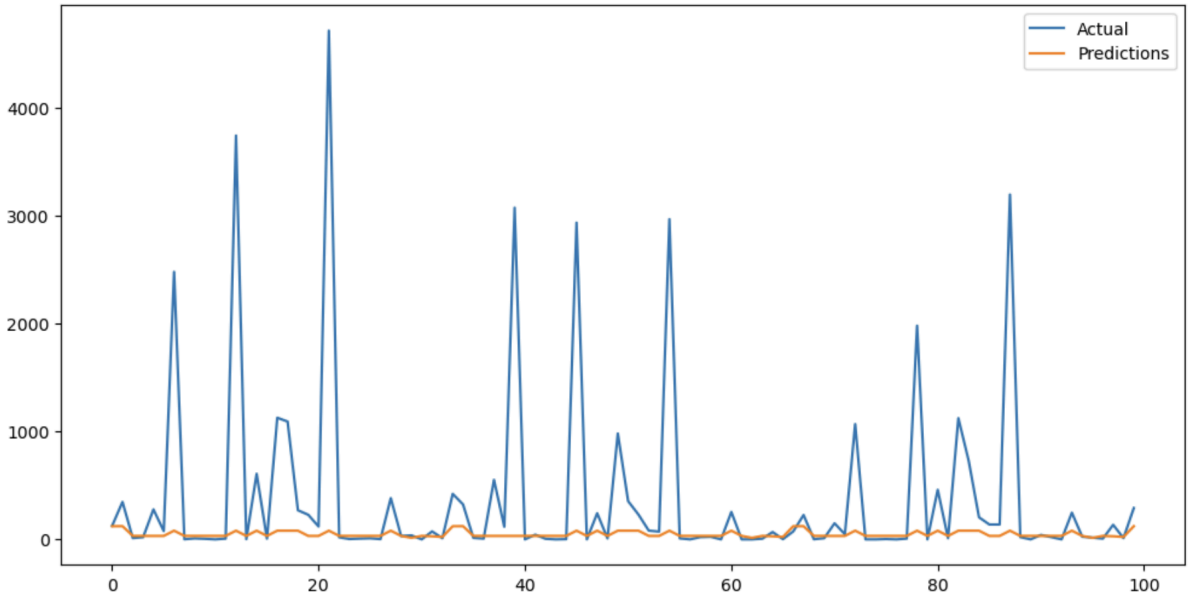


Figure 2.2.7 Actual values vs predicted values

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

```
batch_size = 512
```

Figure 2.3.1 Batch size

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

Figure 2.3.2 Total number of batches

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

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

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

Figure 2.3.3 Printing the RMSE and RMSLE values

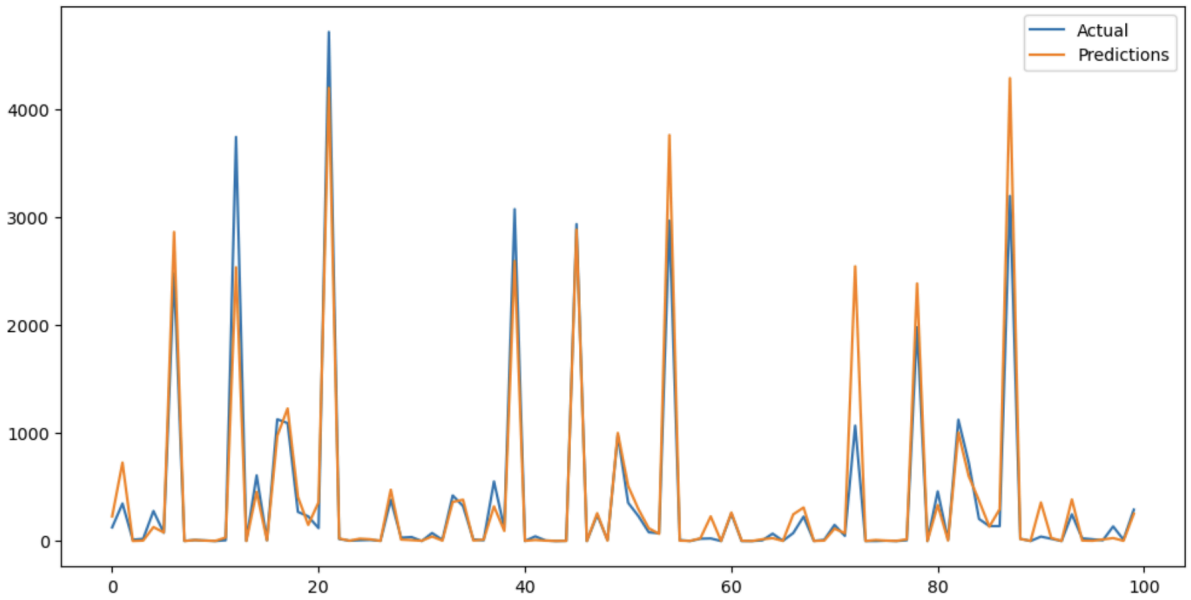


Figure 2.3.4 Actual values vs predicted values

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

```
batch_size = 1024
```

Figure 2.4.1 Batch size

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

Figure 2.4.2 Total number of the batches

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

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

Figure 2.4.3 Printing the RMSE and RMSLE values

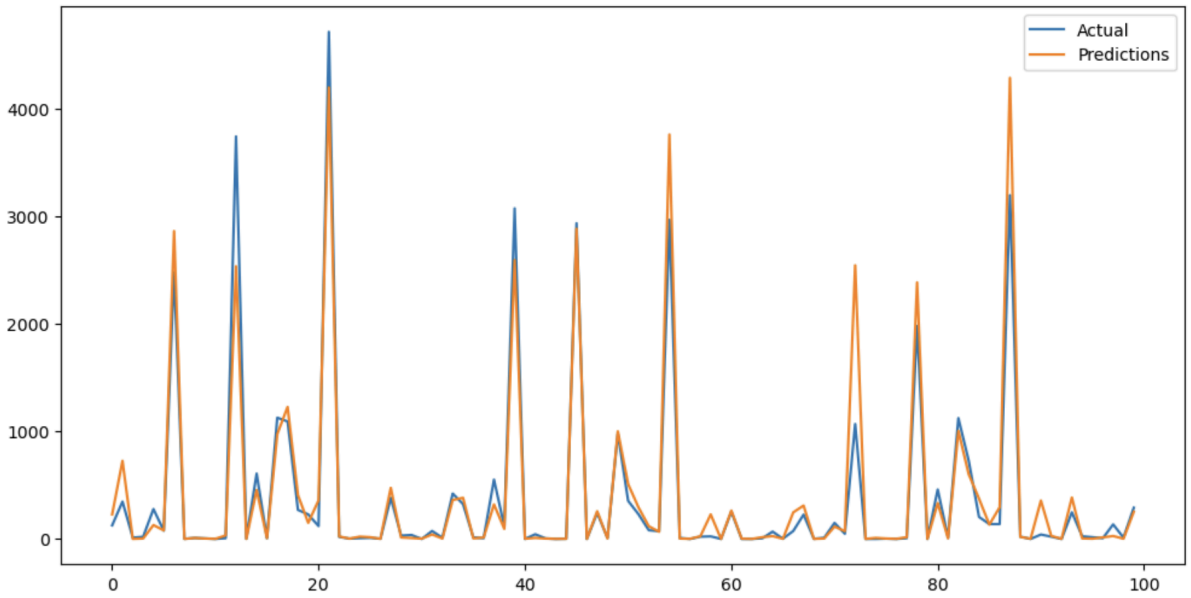


Figure 2.4.4 Actual values vs predicted values

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

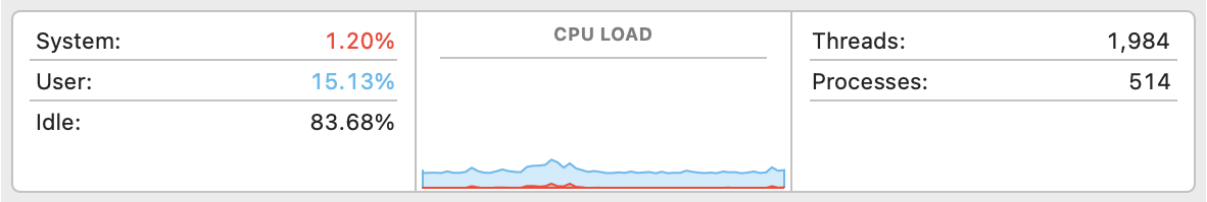


Figure 2.4.5 CPU usage

XGBoost

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

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

Figure 2.5.1 Creating a XGBoost model

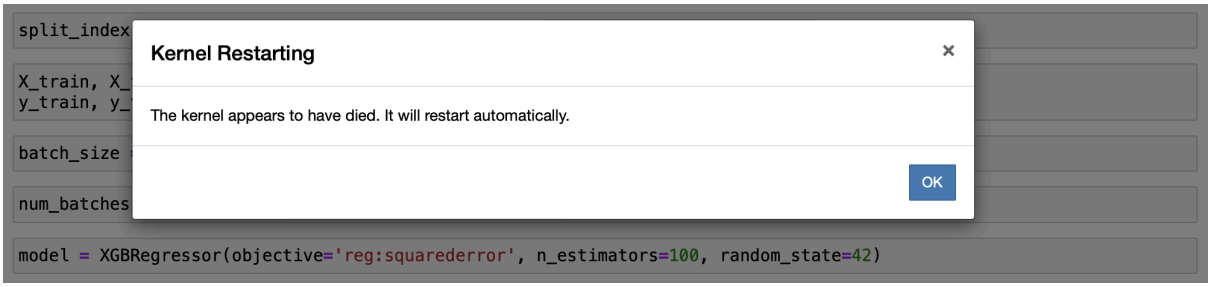


Figure 2.5.2 Kernel died error

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

```
batch_size = 512
```

Figure 2.5.3 Batch size

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

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

Figure 2.5.4 Printing the RMSE and RMSLE values

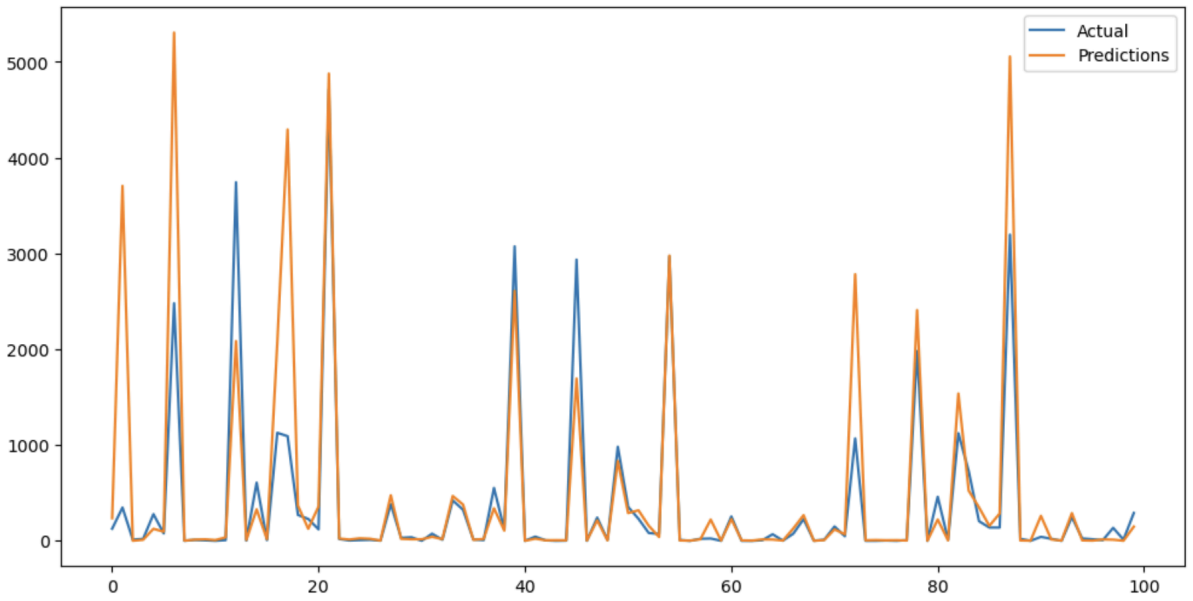


Figure 2.5.5 Actual values vs predicted values

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

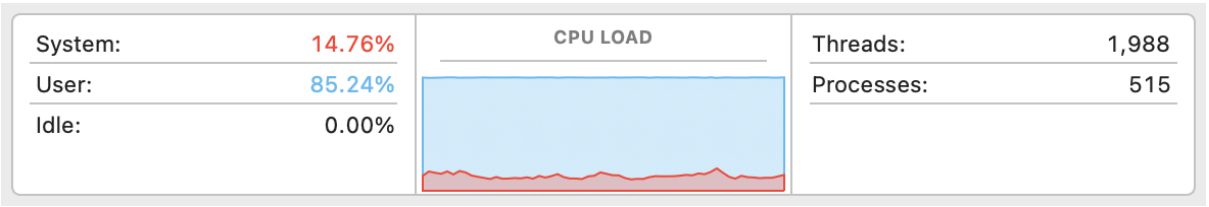


Figure 2.5.6 CPU usage

Prophet

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

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

```
df.head()
```

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

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

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

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

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

Figure 2.6.2 Printing the RMSE and RMSLE values

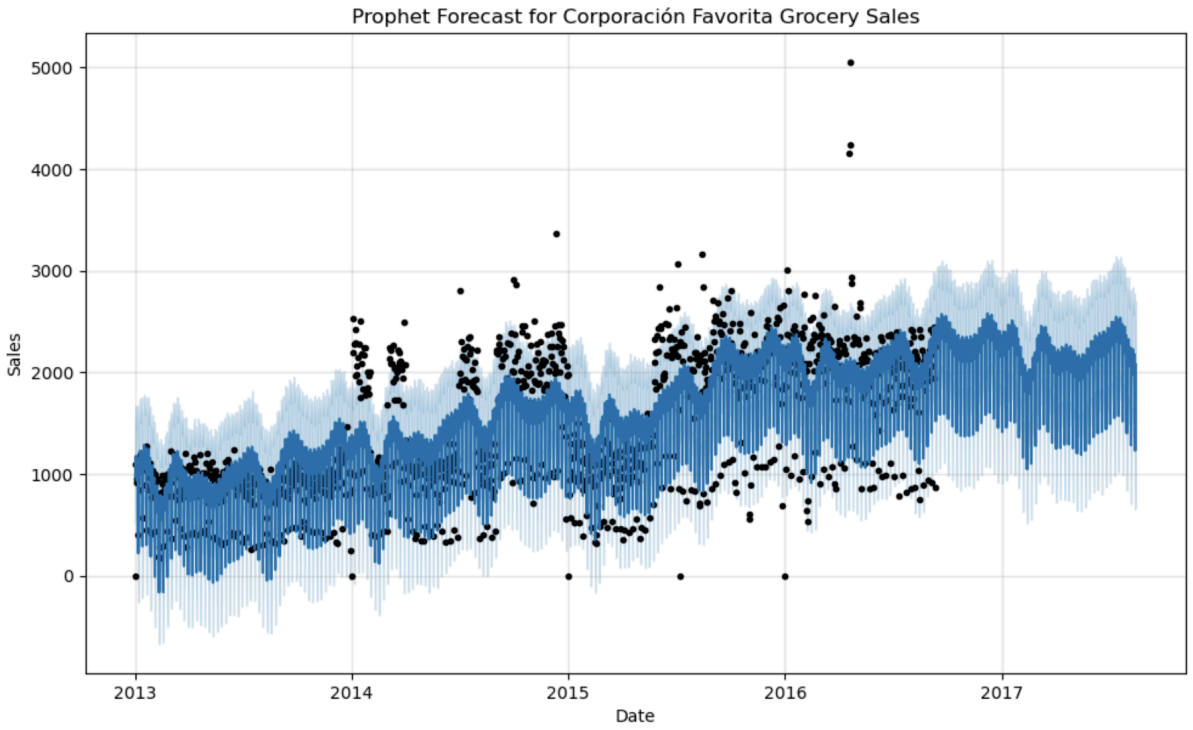


Figure 2.6.3 Prophet model's plot

Recurrent Neural Networks

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

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

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

Figure 2.7.1 Importing libraries and loading the dataset

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

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

```
WINDOW_SIZE = 7
```

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

Figure 2.7.2 Converting the data frame into training sequences

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

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

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

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

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

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

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

Next, we import various components from TensorFlow Keras specific to the training of the model. We create a sequential model (Figure 2.7.4), a model with a linear stack of layers. The first layer of the model is an InputLayer, where we pass our training sequences of length 7 and only 1 feature (sales) per time step as it is a univariate model. The second layer is an LSTM layer with 64 neurons. The third layer is a Dense layer with 8 ReLUs. Lastly, the output layer is a dense layer with a single ReLU neuron as it's a regression task.

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

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

Figure 2.7.4 Importing libraries and initialising our model

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

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

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

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

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

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

Figure 2.7.5 Compiling and fitting the model

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

Figure 2.7.6 Lowest RMSE

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

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

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

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

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

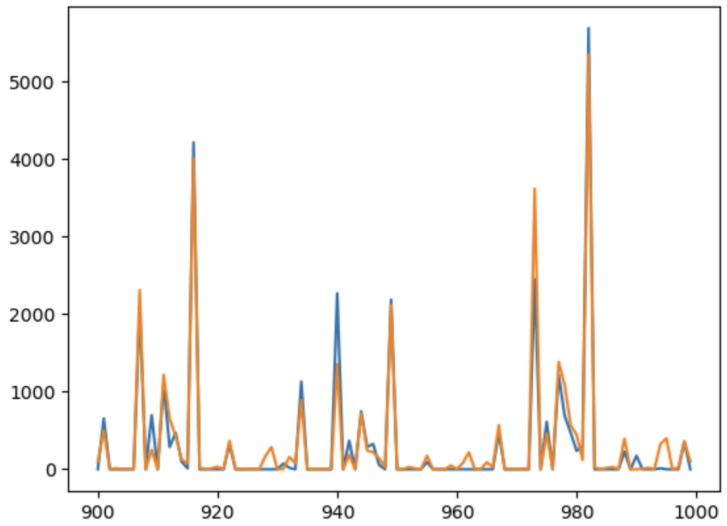


Figure 2.7.7 Test predictions vs actual values plotted

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

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

Figure 2.7.8 GRU (Univariate)

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

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

Figure 2.7.9 LSTM (Multivariate)

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

Figure 2.7.10 Lowest RMSE on LSTM (Multivariate)

Summary

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

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

Figure 2.8.1 Model Summary

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

Deployment

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

requirements.txt U

ModelDeployment > requirements.txt

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

Figure 3.1 pip freeze > requirements.txt

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

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

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

Figure 3.2 Label Encoder

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

In [12]:
```
with open('pickle/family_encoder.pkl', 'wb') as file:
    pickle.dump(family_encoder, file)

with open('pickle/typeholiday_encoder.pkl', 'wb') as file:
    pickle.dump(typeholiday_encoder, file)

with open('pickle/city_encoder.pkl', 'wb') as file:
    pickle.dump(city_encoder, file)

with open('pickle/state_encoder.pkl', 'wb') as file:
    pickle.dump(state_encoder, file)

with open('pickle/typestores_encoder.pkl', 'wb') as file:
    pickle.dump(typestores_encoder, file)
```

Figure 3.3 Exporting all the encoders through Pickle

```
In [28]: dump(model, 'joblib/M10.joblib')

Out[28]: ['joblib/M10.joblib']
```

Figure 3.4 Exporting the model using Joblib

To check whether the model was working after being deployed locally, a set of hard coded values (Figure 3.5) were given to the model. The values were in the form as they would be input in the actual deployment. The only attribute that was not hard coded was the family (category of the product). This deployment made predictions for all the listed products, but only for one specific (hard coded) date. Functionality to dynamically manipulate variables was added later.

```
data = {
    'store_nbr': 1,
    'family_encoded': "",
    'onpromotion': 0,
    'typeholiday_encoded': "Holiday",
    'dcoilwtico': 46.8,
    'city_encoded': "Quito",
    'state_encoded': "Pichincha",
    'typestores_encoded': "D",
    'cluster': 13,
    'day_of_week': 3,
    'day': 16,
    'month': 8,
    'year': 2017
}
```

Figure 3.5 Hard coded values for the model

The encoders that were exported earlier were imported on Flask using Pickle. Each encoder was used to transform the value of their respective attribute (Figure 3.6).

```
if request.method == 'POST':
    df['family_encoded'] = request.form.get('family_encoded')

with open('models/family_encoder.pkl', 'rb') as file:
    family_encoder = pickle.load(file)

with open('models/typeholiday_encoder.pkl', 'rb') as file:
    typeholiday_encoder = pickle.load(file)

with open('models/city_encoder.pkl', 'rb') as file:
    city_encoder = pickle.load(file)

with open('models/state_encoder.pkl', 'rb') as file:
    state_encoder = pickle.load(file)

with open('models/typestores_encoder.pkl', 'rb') as file:
    typestores_encoder = pickle.load(file)

df['family_encoded'] = family_encoder.transform([df['family_encoded'].iloc[0]])[0]
df['typeholiday_encoded'] = typeholiday_encoder.transform([df['typeholiday_encoded'].iloc[0]])[0]
df['city_encoded'] = city_encoder.transform([df['city_encoded'].iloc[0]])[0]
df['state_encoded'] = state_encoder.transform([df['state_encoded'].iloc[0]])[0]
df['typestores_encoded'] = typestores_encoder.transform([df['typestores_encoded'].iloc[0]])[0]
```

Figure 3.6 Importing encoders and encoding categorical attributes

Once all the attributes were in a form that can be fed to the model, the model was loaded using Joblib and given the feature vector X (Figure 3.7). The model produced a prediction which was then sent to the homepage where it was displayed (Figure 3.8).

```
features = [
    'store_nbr',
    'onpromotion',
    'dcoilwtico',
    'cluster',
    'day_of_week',
    'day',
    'month',
    'year',
    'family_encoded',
    'typeholiday_encoded',
    'city_encoded',
    'state_encoded',
    'typestores_encoded',
]

X = df[features]

model = load('models/M10.joblib')
```

Figure 3.7 Creating feature vector X and importing the model using Joblib

```
prediction = model.predict(X)

return render_template(
    'index.html',
    prediction = prediction
)
```

Figure 3.8 Generating predictions using the XGBoost model

Figure 3.9 and 3.10 show a simple interface, which was improved later. The interface shows a dropdown list of all the listed products. The user can select the family (product category) and the model generates predicted sales for that family for the specified (hardcoded) store and the specified (hard coded) date.

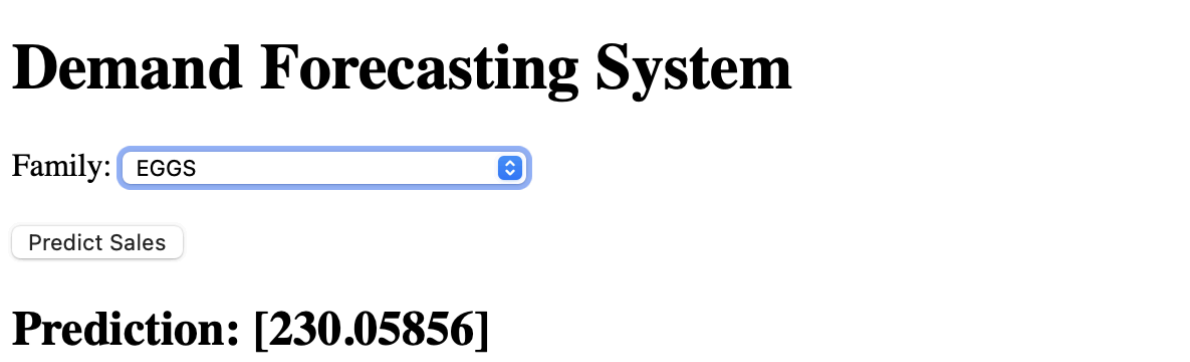


Figure 3.9 Prediction generated by the model for ‘eggs’ on the hardcoded store and date

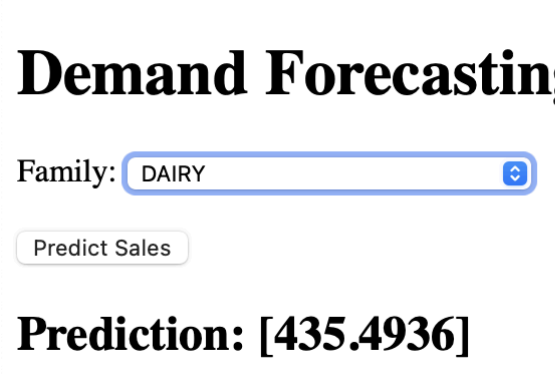


Figure 3.10 Prediction generated by the model for 'dairy' on the hardcoded store and date

For the demo, the users were given control over the selections dynamically, the data for holiday, oil pricing, and promotion was fetched in accordance to the date selected, and a side-by-side comparison was presented of the predicted sales and actual sales.

Bootstrap was used to improve the user experience visually and add functionality (Date Picker) as shown in Figure 3.11. The dates allowed to be selected are the dates from the test set. This was done so that we can show predicted values and actual values.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.9.0/js/bootstrap-datepicker.min.js"></script>
<script>
    $(document).ready(function(){
        $('.datepicker').datepicker({
            format: 'yyyy-mm-dd',
            startDate: '2016-09-14',
            endDate: '2017-08-14',
            autoclose: true
        });
    });
</script>
```

Figure 3.11 Date Picker

As discussed with Ma'am Huda, only 5 families (Beverages, Dairy, Frozen Foods, Meats, and Seafood) were selected for the demo (Figure 3.12).

```
<select class="form-control" id="family" name="family" required>
    <option value="BEVERAGES">BEVERAGES</option>
    <option value="DAIRY">DAIRY</option>
    <option value="FROZEN FOODS">FROZEN FOODS</option>
    <option value="MEATS">MEATS</option>
    <option value="SEAFOOD">SEAFOOD</option>
</select>
```

Figure 3.12 Shortlisted categories in the drop down list

To allow the user to select the date, the conversion from date to the extracted features we trained the model on had to be performed live (Figure 3.13).

```
if request.method == 'POST':
    df['date'] = request.form.get('datepicker')
    df['date'] = pd.to_datetime(df['date'])
    df['day_of_week'] = df['date'].dt.day_of_week
    df['day_of_week'] = df['day_of_week']+1
    df['day'] = df['date'].dt.day
    df['month'] = df['date'].dt.month
    df['year'] = df['date'].dt.year
```

Figure 3.13 Converting date into day_of_week, day, month, and year

In order to make it possible to fetch the holiday and oil price data for the selected date, a filtered dataset was created. The filter dataset only included dates from the test set, last 20% of the training data (Figure 3.14)

```
split_index = int(0.8 * train.shape[0])
```

```
filtered_train = train.iloc[split_index:]
```

```
filtered_train.shape
```

```
(600178, 10)
```

Figure 3.14 Filtering test dates

As discussed, the user was not allowed to select the store. So, for the sake of the demo, we selected store number 1 (Figure 3.15).

```
filtered_train = filtered_train[filtered_train['store_nbr'] == 1]
```

```
filtered_train.shape
```

```
(11088, 10)
```

```
filtered_train = filtered_train.drop(['store_nbr'], axis=1)
```

Figure 3.15 Filtering store number 1

The data was filtered for the shortlisted families (Figure 3.16) and inspected to ensure there were no issues.

```
filtered_train = filtered_train[(filtered_train['family'] == 'BEVERAGES') | (filtered_train['family'] == 'DAIRY')
```

Figure 3.16 Filtering shortlisted families

```
filtered_train.head()
```

	date	family	sales	onpromotion	typeholiday	dcoilwtico	day	month	year
2402139	2016-09-13	BEVERAGES	1942.000	0	NDay	44.91	13	9	2016
2402144	2016-09-13	DAIRY	703.000	0	NDay	44.91	13	9	2016
2402147	2016-09-13	FROZEN FOODS	95.000	0	NDay	44.91	13	9	2016
2402160	2016-09-13	MEATS	288.823	0	NDay	44.91	13	9	2016
2402168	2016-09-13	SEAFOOD	34.034	0	NDay	44.91	13	9	2016

```
filtered_train.tail()
```

	date	family	sales	onpromotion	typeholiday	dcoilwtico	day	month	year
2999109	2017-08-15	BEVERAGES	1942.000	11	Holiday	47.57	15	8	2017
2999114	2017-08-15	DAIRY	602.000	19	Holiday	47.57	15	8	2017
2999117	2017-08-15	FROZEN FOODS	89.000	1	Holiday	47.57	15	8	2017
2999130	2017-08-15	MEATS	274.176	0	Holiday	47.57	15	8	2017
2999138	2017-08-15	SEAFOOD	22.487	0	Holiday	47.57	15	8	2017

Figure 3.17 Inspecting the final dataset

In order to access the holiday, oil pricing, promotion, and actual sales data for the selected date and family, the filtered dataset is searched for the selected day, month, year, and family (Figure 3.18). As it would always return only one row, the data for holiday, oil pricing, and promotion is filled in the feature vector that is going to be used for the prediction. Actual sales are stored in a variable that is passed on to the html template (Figure 3.19).

```
filtered_df = pd.read_csv("data/filtered_train.csv")
filtered_df = filtered_df[(filtered_df['day'] == int(df['day'])) & (filtered_df['month'] == int(df['month'])) & (filtered_df['year'] == int(df['year']))]
filtered_df = filtered_df[(filtered_df['family'] == str(df['family'].iloc[0]))]
```

Figure 3.18 Filtering data based on day, month, year, and family

```
df['typeholiday'] = filtered_df['typeholiday'].iloc[0]
df['dcoilwtico'] = filtered_df['dcoilwtico'].iloc[0]
df['onpromotion'] = filtered_df['onpromotion'].iloc[0]
actual = round(filtered_df['sales'].iloc[0])
```

Figure 3.19 Collecting holiday, oil pricing, promotion, and actual sales data

The feature vector is passed on to the model as explained earlier and the prediction generated by the model along with the actual sales are sent to the home page where they are displayed (Figure 3.20).

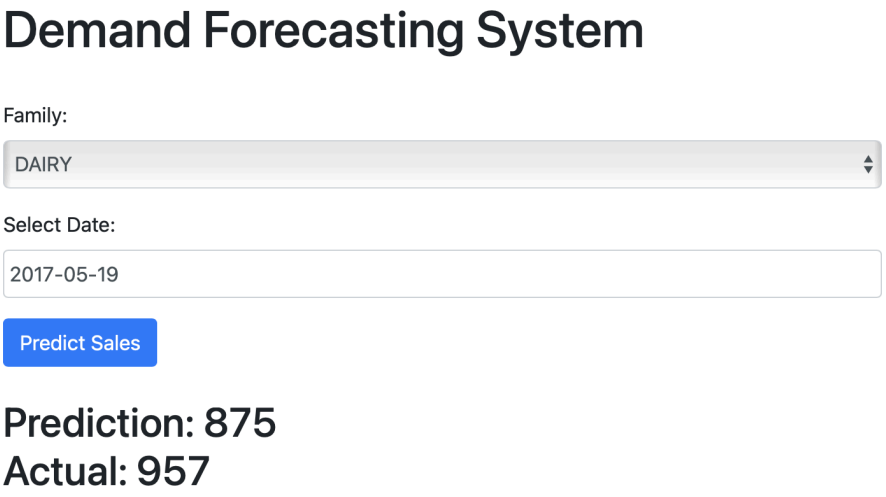


Figure 3.20 Flask Application

This concluded the development of the Flask application. The Flask application was then deployed on Digital Ocean (Figure 3.21). The demo of the ML model was given live on the cloud during the final presentation.

Deployment: https://hammerhead-app-6m6td.ondigitalocean.app/

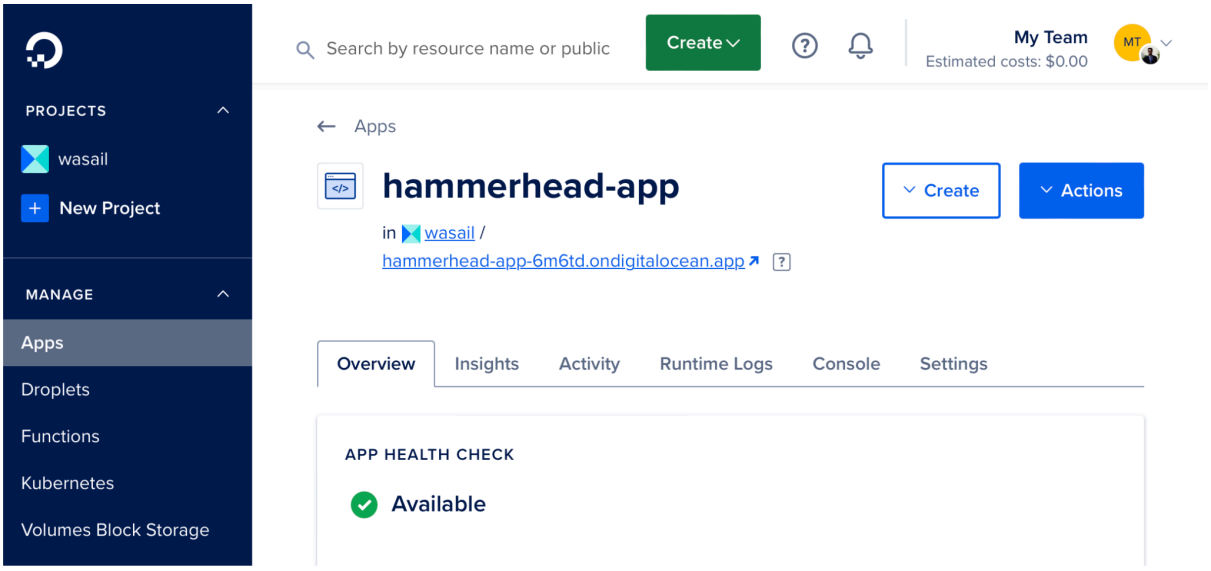


Figure 3.21 Deployment on Digital Ocean

Version Control

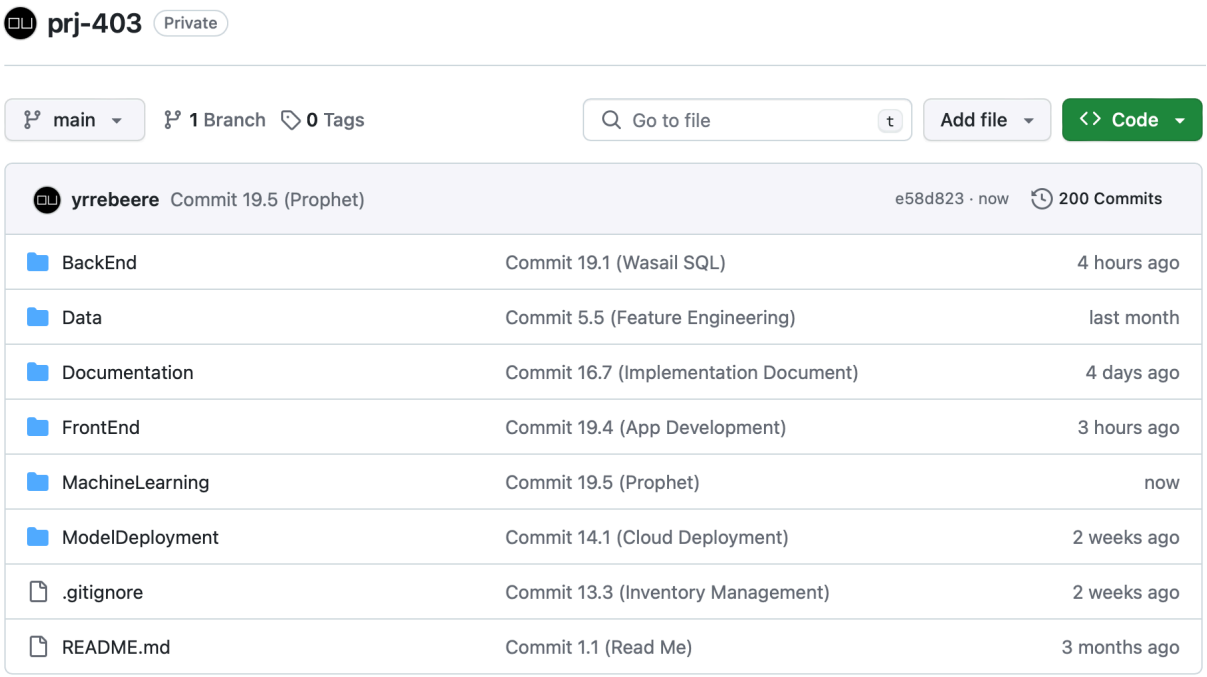


Figure 4.1 200 commits on GitHub