PanTree

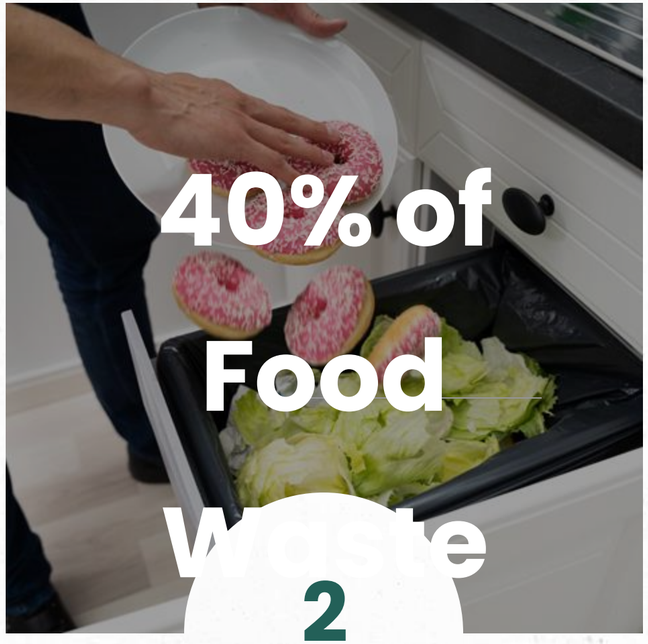
An app to prevent food waste

Neo Zhi Xuan, Kiew Ten Wei, Kauthar Basharahil, Law Wei Lu, Mitra Sachithananthan, Ng Zhuo Quan

The Problem

1

Food Waste is an important issue in SG, with **813,000 tonnes** generated in 2022



2

Commercial and industrial (C&I) premises account for around **40%** of SG's food waste each year

The Problem

1

Food Waste is an important issue in SG, with **813,000 tonnes** generated in 2022

2

Commercial and industrial (C&I) premises account for around **40%** of SG's food waste each year

Following NEA's food waste management hierarchy, **preventing food wastage at its source** is most desirable for the long term.

User Needs – Digital Transformation & Susti

No formal means to optimise inventory and redistribute excess food

F&B Outlets **dispose off** expired ingredients and excess food

F&B Businesses are unsure of how to gauge **demand** and **perform forecasting**

Many restaurants have a **manual** pen & paper **stock taking** and **procurement**

Consumers are hard hit by **rising food costs and GST**

PanTree – Mobile App to manage inventory and reduce food waste

Meeting the System Requirements

Inventory Updates

Inventory List Viewing

Predictive Demand

Inventory Management

Smart Menu Generation

Agile Methodology

Scrums

Weekly scrum meetings

Sprints

Sprint planning, sprints, sprint reviews, healthy pipeline

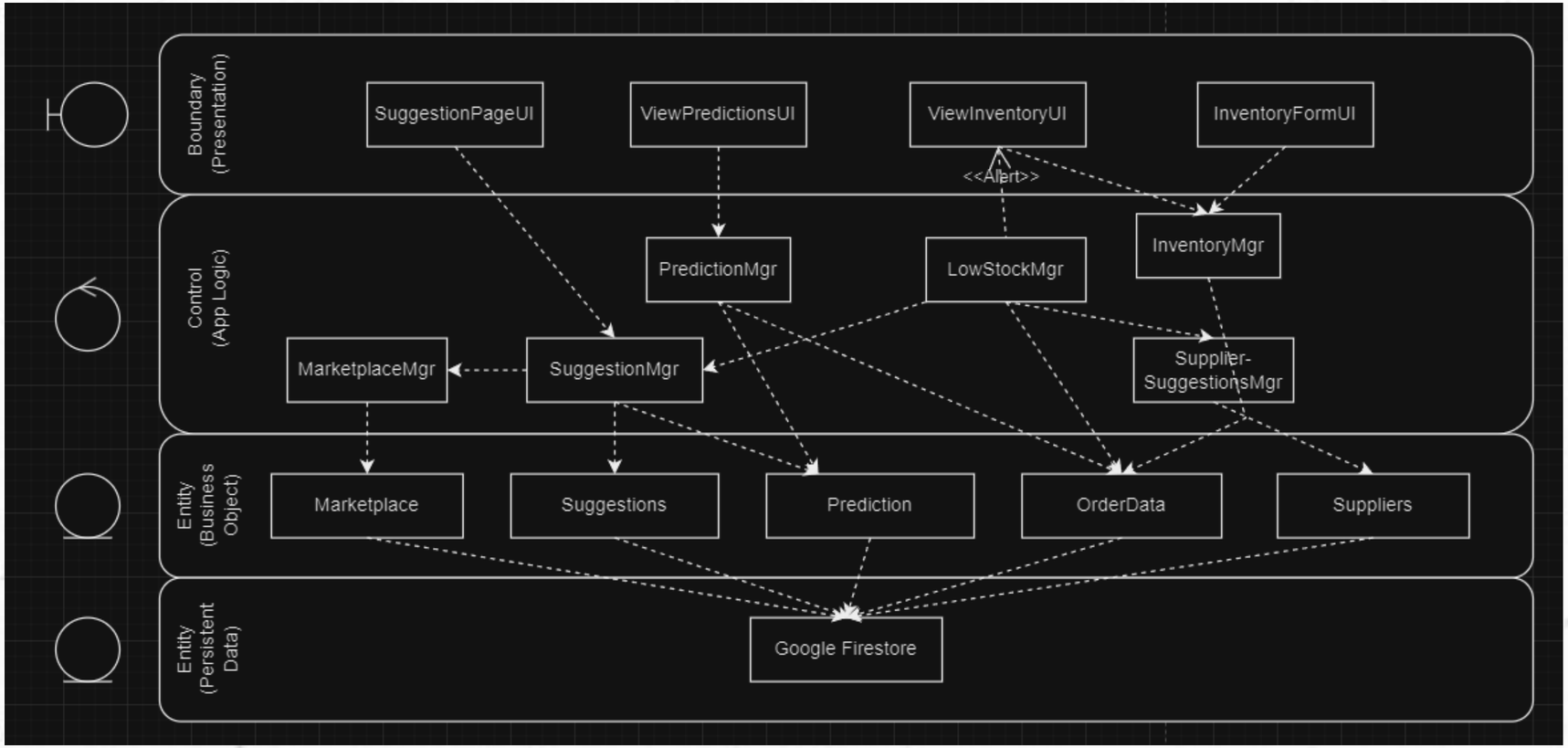
Kanban

Kanban ticketing system to track tasks

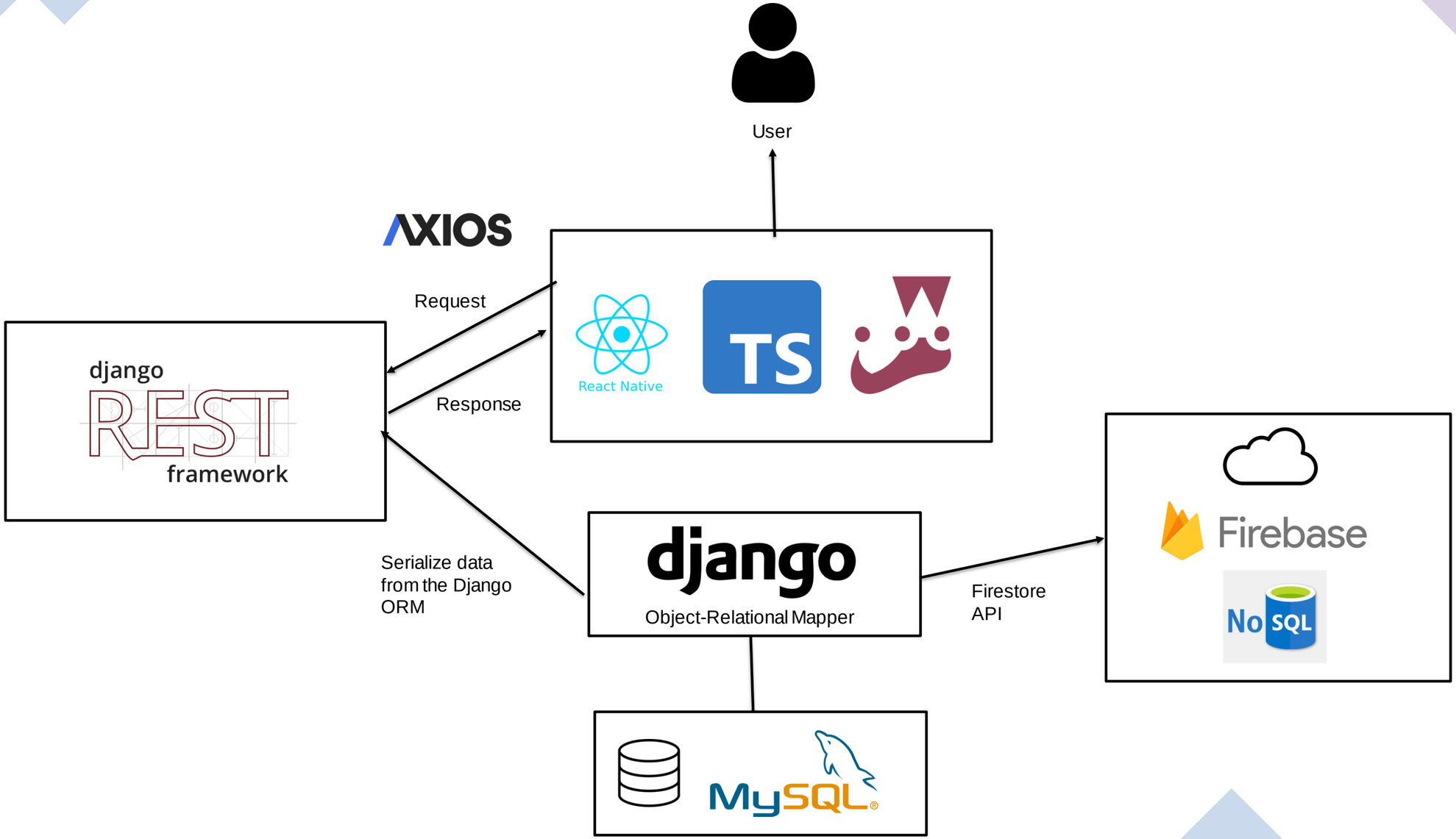
System Architecture

4-Layered Architecture

Dependencies flow downwards



Technical Architecture



PanTree

(FrontEnd) React Native

- JavaScript library for building user interfaces
- Create dynamic and responsive user experiences

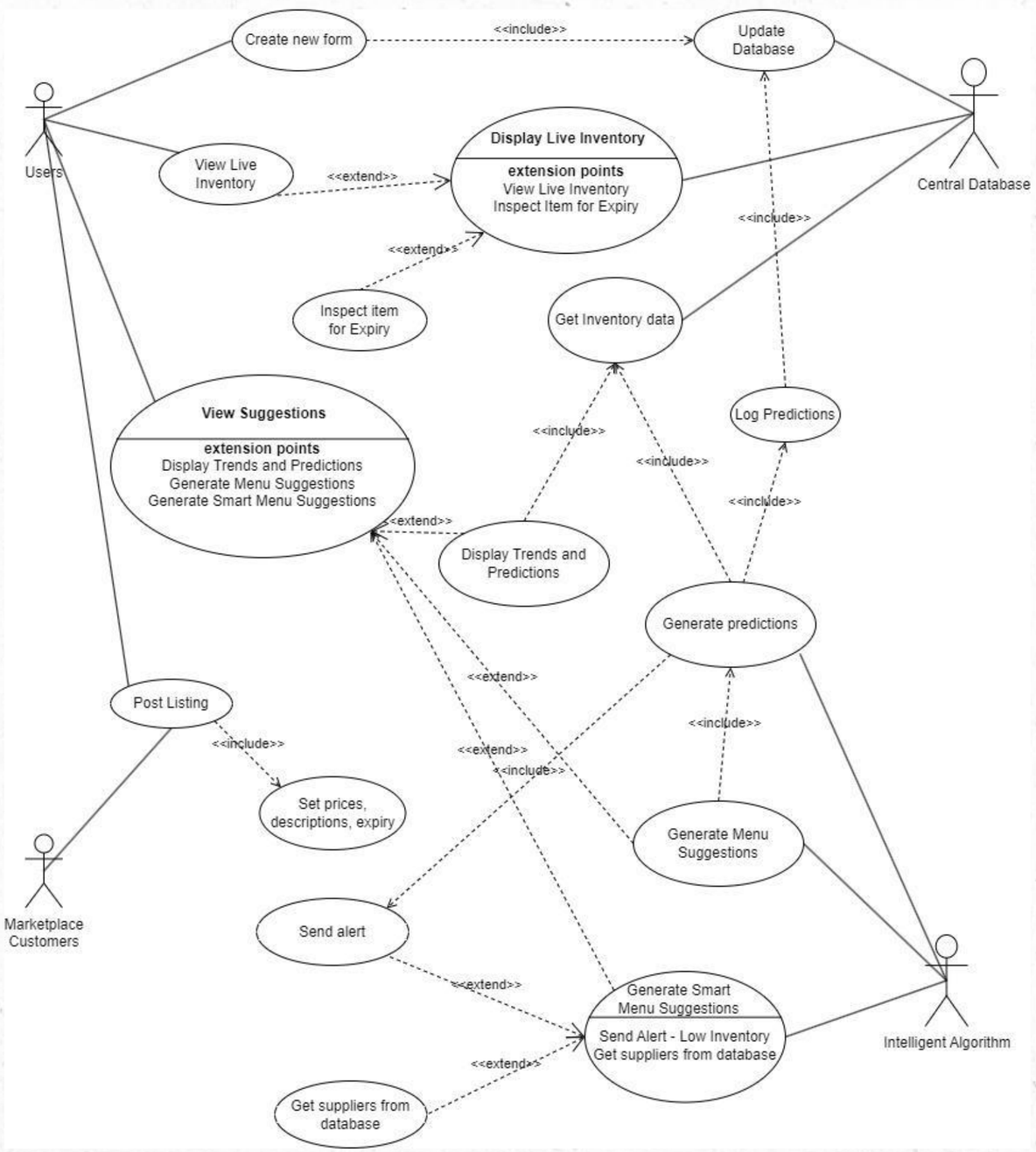
(Backend) Django

- Simplifies development process
- Built-in admin panel, authentication system and object relational mapping

(Database) Firebase

- Real-time NoSQL database
- Seamless Integration

Use Case Diagram

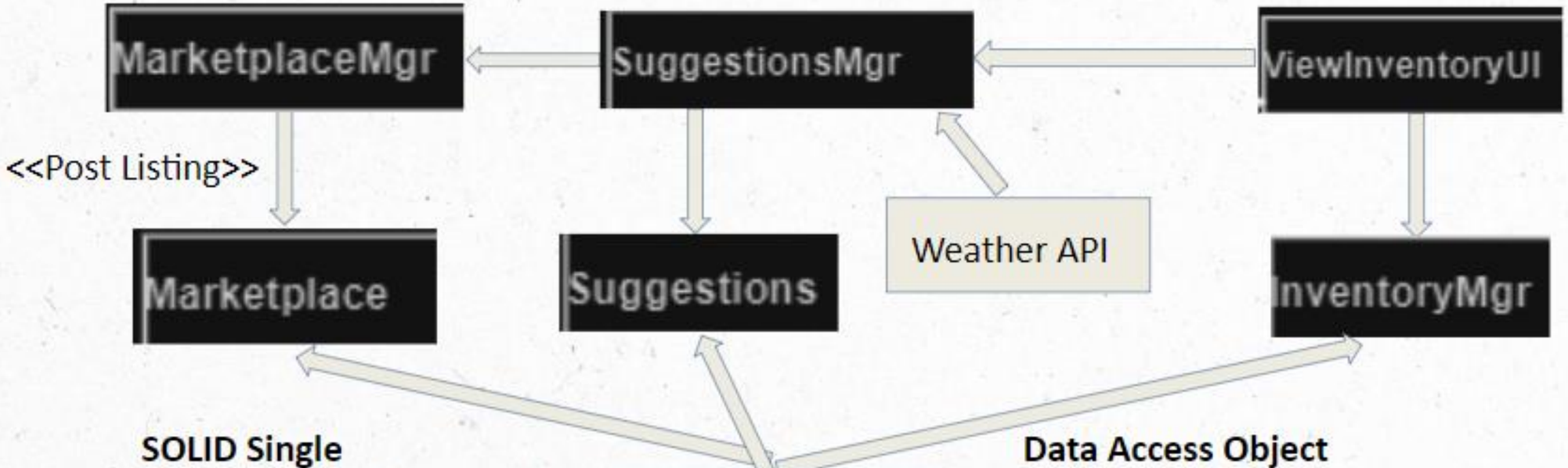


CHANGEX

Use Case 1:

Functionality to post a listing to our marketplace

System Design for Post Listing



SOLID Single Responsibility Framework

- every class, module, or function in a program should have one responsibility/purpose in a program

Data Access Object Pattern

- allows our application to interact with the data storage without exposing the underlying details of the data source

System Design for Post Listing

```python
def get_weather_data():
    # Use the data.gov.sg weather API endpoint or your specific API endpoint
    api_url = "https://api.data.gov.sg/v1/environment/24-hour-weather-forecast"
```

```python
# You can customize this logic based on your menu and the desired weather conditions
if "thundery showers" in forecast:
    # Suggest warm and hearty dishes for rainy weather and high humidity
    for item in menu_items:
        if "soup" in item["item_name"].lower() and check_ingredient_availability(ingredients, item.get("ingredients", {})):
            item["message"] = "Suggested"
            suggested_menu.append(item)
elif "partly cloudy" in forecast:
    # Suggest lighter items for partly cloudy weather and low humidity
    for item in menu_items:
        if "salad" in item["item_name"].lower() and check_ingredient_availability(ingredients, item.get("ingredients", {})):
            item["message"] = "Suggested"
            suggested_menu.append(item)
for item in menu_items:
    if check_ingredient_availability(ingredients, item.get("ingredients", {})) and not (item["id"] == suggested_menu[0]["id"]):
        suggested_menu.append(item)

suggested_menu.append({"forecast": f"{forecast}"})
suggested_menu.append({"humidity_low": f"{humidity_low}"})
suggested_menu.append({"humidity_high": f"{humidity_high}"})
```

Black Box Testing for Suggestions

a. Price

Test ID	Scenario	Expected Result	Actual Result
1	No price is entered	The system prompts the user to enter a price	The system prompts the user to enter a price
2	An invalid price is entered	The system prompts the user to enter a valid price	The system prompts the user to enter a valid price
3	A valid price is entered	The system displays a successful message provided other fields are valid	The system displays a successful message when other fields are valid

b. Description

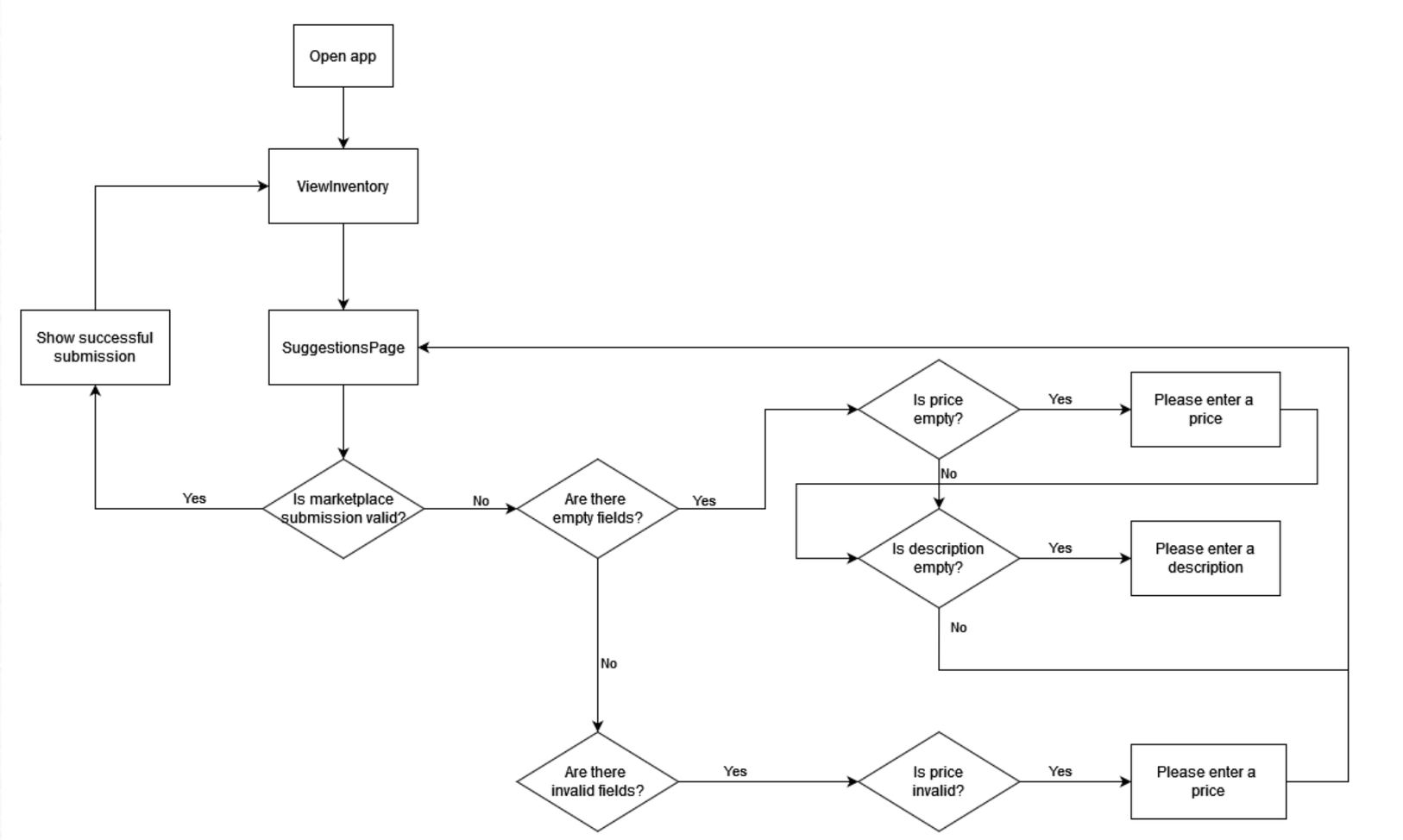
Test ID	Scenario	Expected Result	Actual Result
1	No description is entered	The system prompts the user to enter a description	The system prompts the user to enter a description
2	A valid description is entered	The system displays a successful message provided other fields are valid	The system displays a successful message when other fields are valid

Black Box Testing for Suggestions

c. Specific cases (combination)

Price	Description	Expected Result	Actual Result
50	Delicious	Successful posting on the marketplace	Successful posting on the marketplace
Empty(“”)	Delicious	Please enter a price	Please enter a price
50	Empty(“”)	Please enter a description	Please enter a description
-1	Delicious	Please enter a valid price	Please enter a valid price

White Box Testing for Suggestions

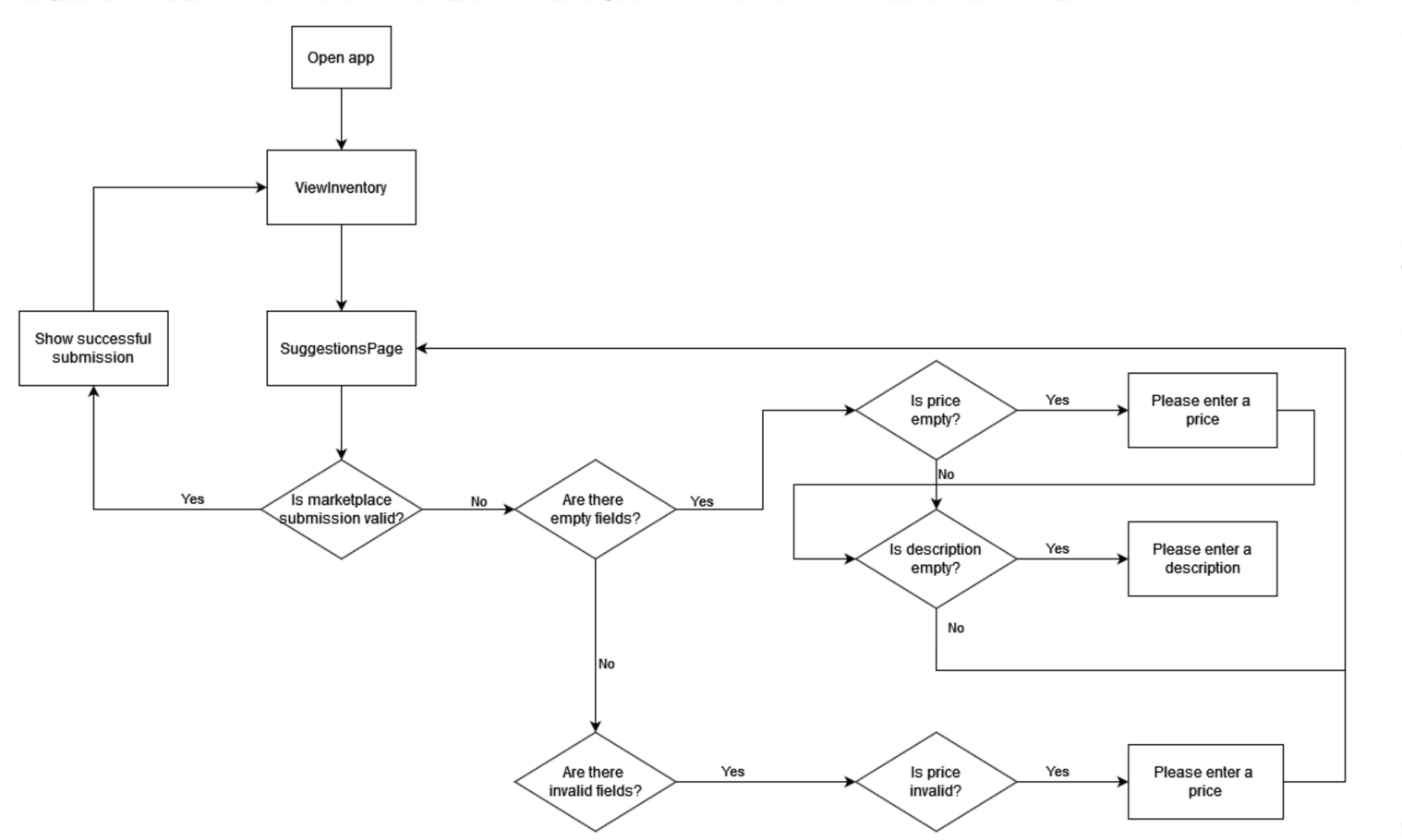


White Box Testing for Suggestions

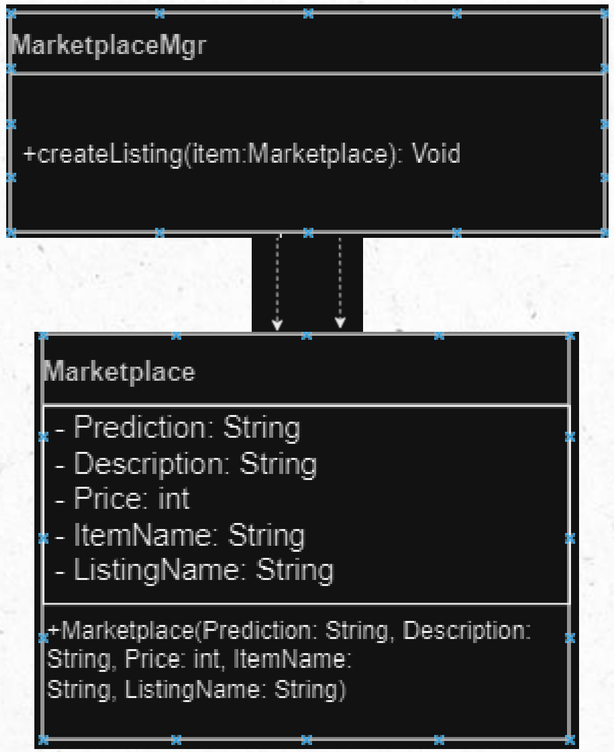
.

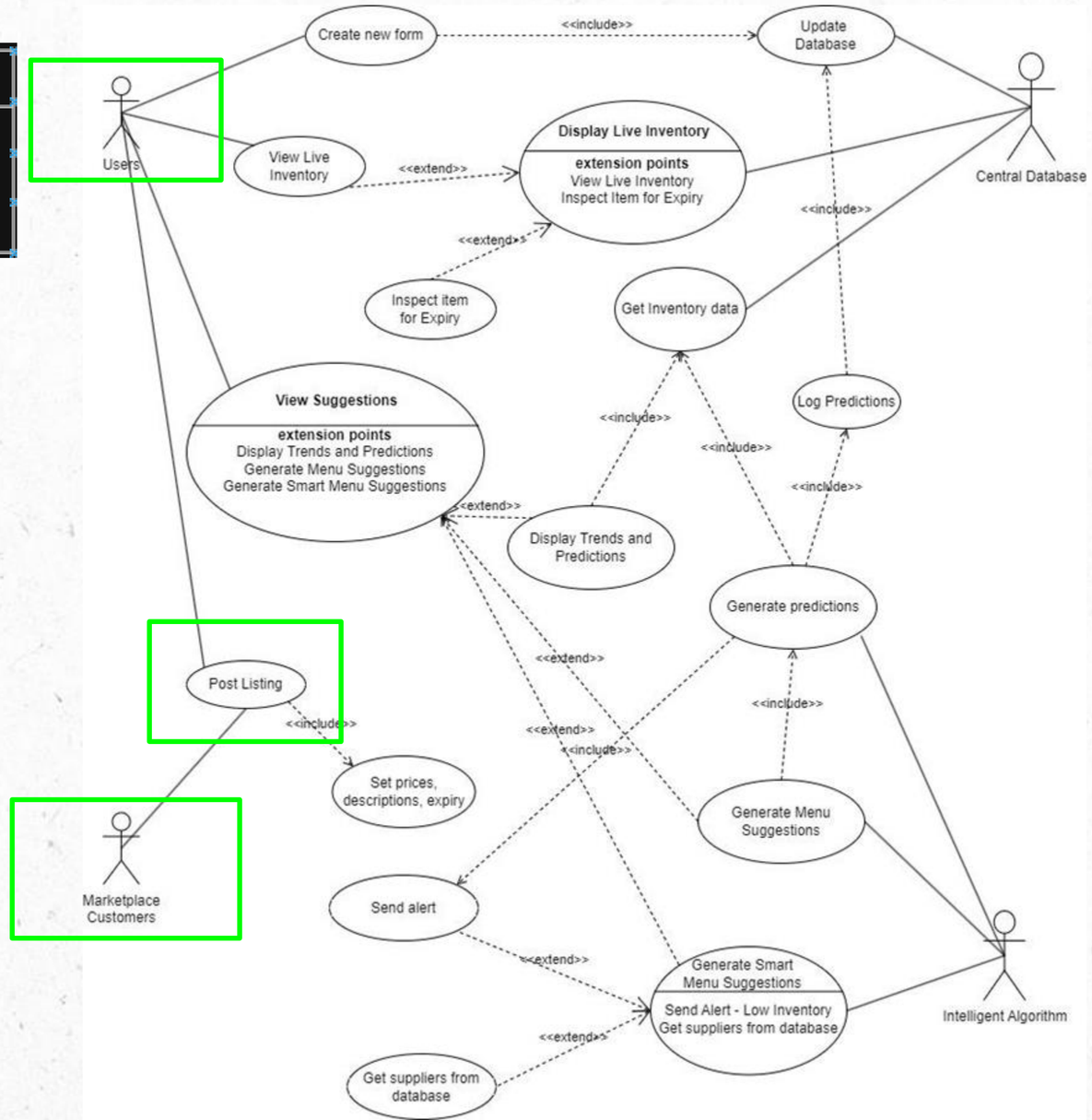
Control Flow Diagram

- User (Frontend) side
- Checks for validity of Marketplace submission



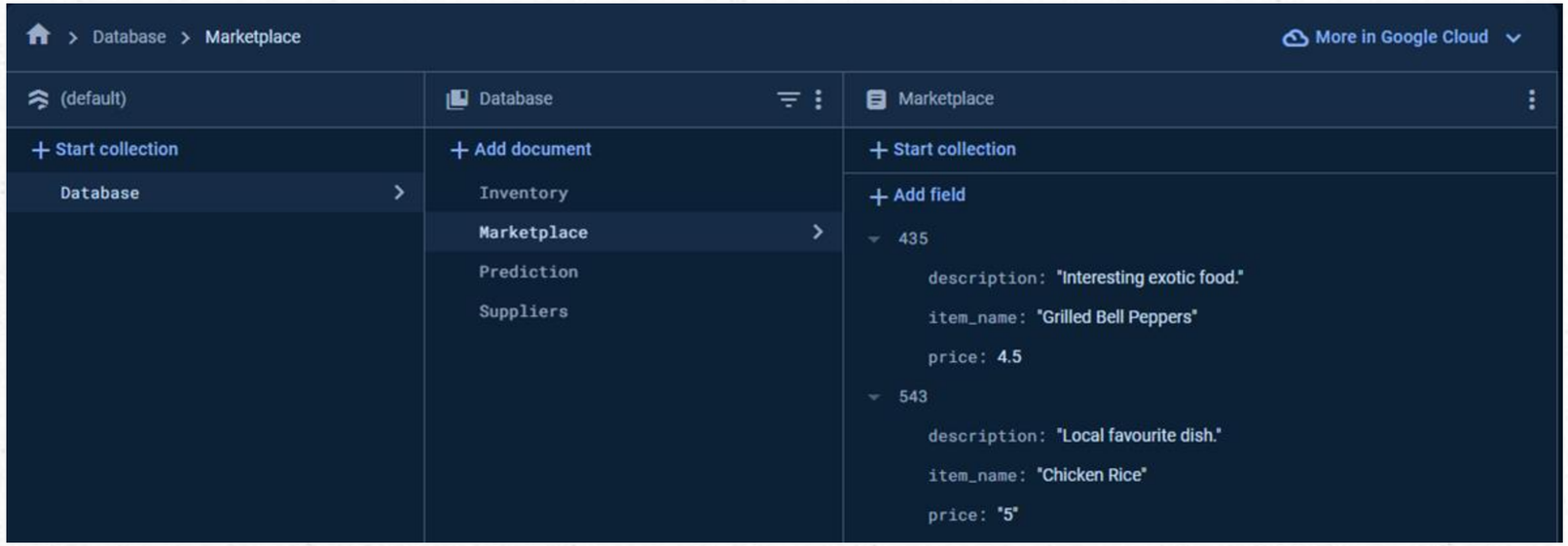
Post Listing





Post Listing

Generates random non existing ID for item



Stores input data from Frontend into Firestore Database

Stores input data from Frontend into SQL models

```python
@api_view(['POST'])
def createMarketplace(request):
    serializer = MarketplaceSerializer(data=request.data)

    if serializer.is_valid():
        # Generate a random item_id
        while True:
            random_item_id = random.randint(
                1, 1000)  # Adjust the range as needed

            # Check if the random item_id already exists in the Django model
            if not Inventory.objects.filter(item_id=random_item_id).exists():
                # Check if the random item_id already exists in Firestore
                db = firestore.Client()
                doc_ref = db.collection('Database').document('Marketplace')
                firestore_data = doc_ref.get().to_dict()
                if str(random_item_id) not in firestore_data:
                    break

        # Save the data to the Django model
        instance = serializer.save(item_id=random_item_id)

        # Convert the serializer data to a dictionary
        data_dict = serializer.data

        # Initialize Firestore client
        db = firestore.Client()

        # Get a sanitized document ID (replace spaces with underscores)
        document_id = instance.item_name.replace(" ", "_")

        # Get the document reference in Firestore
        doc_ref = db.collection('Database').document('Marketplace')

        # Update the data directly under the "Marketplace" document
        doc_ref.update({
            str(random_item_id): {
                "item_name": data_dict["item_name"],
                "description": data_dict["description"],
                "price": data_dict["price"]
            }
        })

        return Response(serializer.data, status=status.HTTP_201_CREATED)

    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

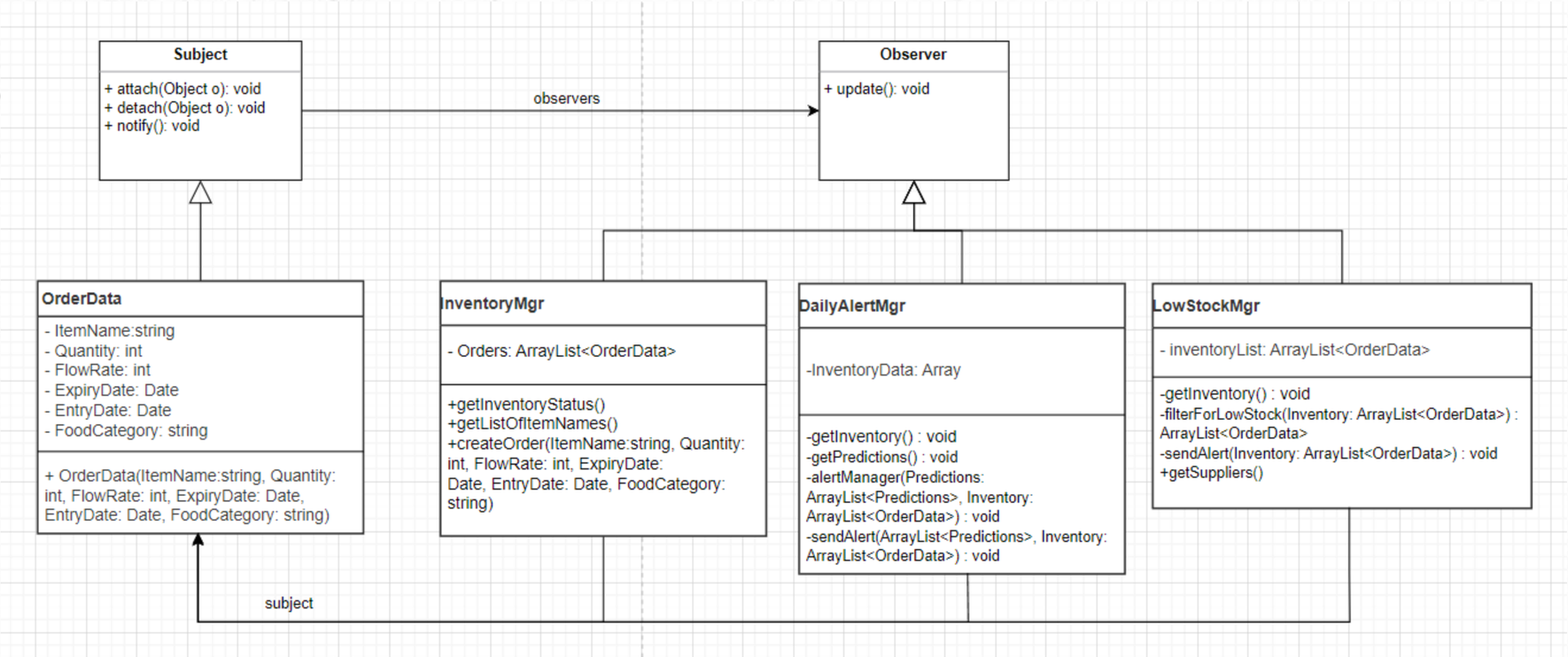
CHANGEX

Use Case 2: Low-stock alert function

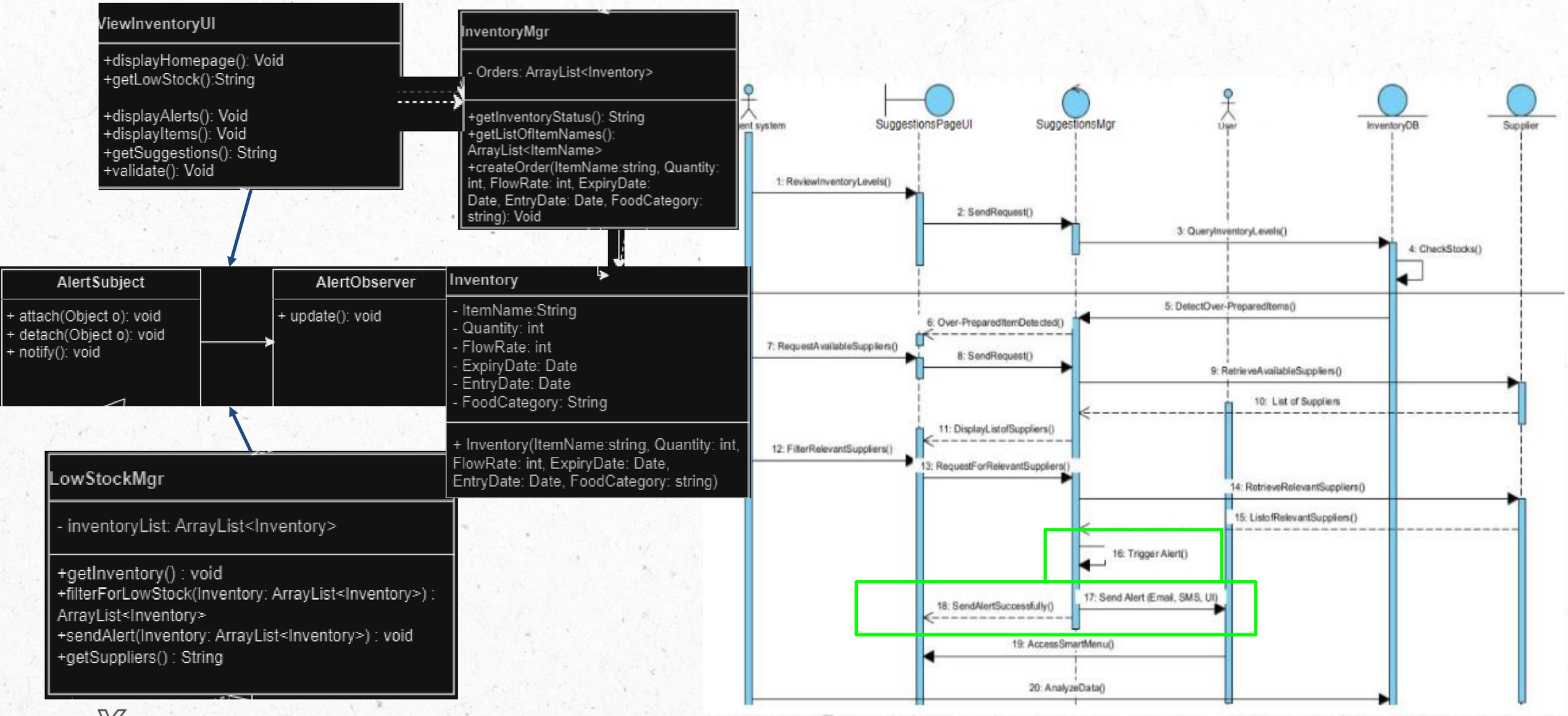
System Design for Low Stock Alert

Observer Pattern

- establish a separation between Subjects and Observers, so that Subjects do not know the Observers' specification
- controller has to constantly listen for an update, which might be costly and slows down performance



Low Stock Alert



CHANGEX

Low Stock Alert

path that activates every few seconds to check the inventory of low stock

```
path('fn/filterForLowStock', views.filterForLowStock,
     name="filterForLowStock"),
```

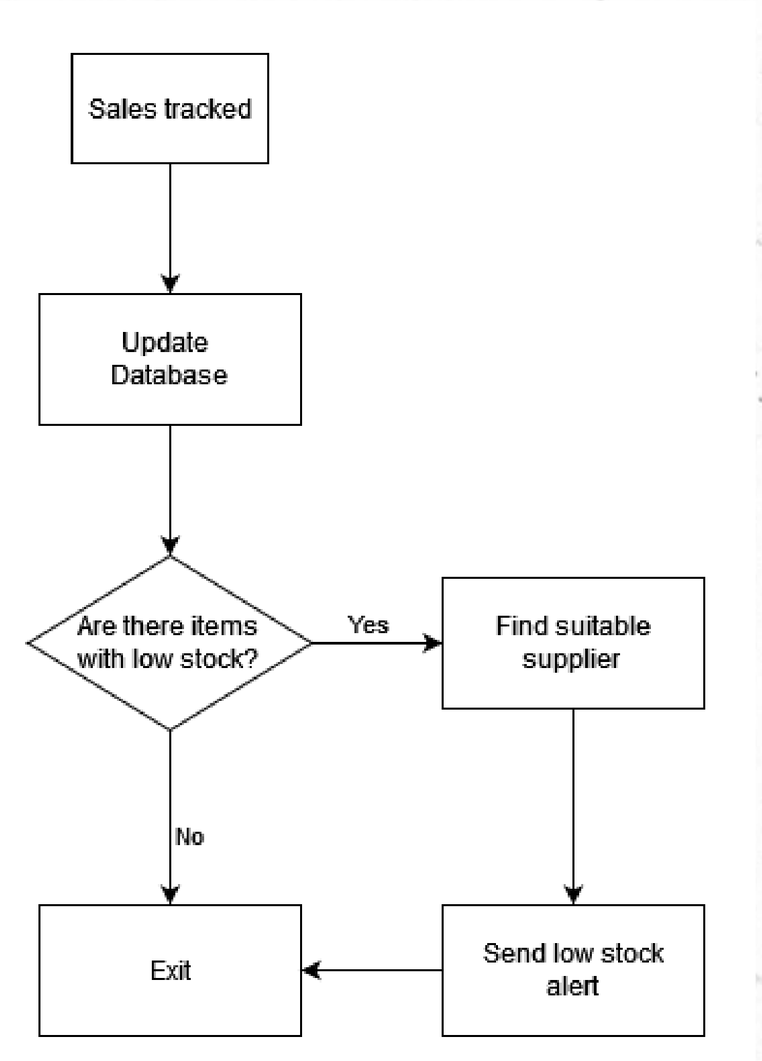
Api get the item name and quantity which has a quantity below 5 and store it in low_quantitty_items

```
@api_view(['GET'])
def filterForLowStock(request):
    # Aggregate the quantities based on item_name
    aggregated_items = Inventory.objects.values(
        'item_name').annotate(total_quantity=Sum('quantity'))

    # Filter out items with total quantity less than 5
    low_quantity_items = [
        item for item in aggregated_items if item['total_quantity'] < 5]
    return Response(low_quantity_items)
```

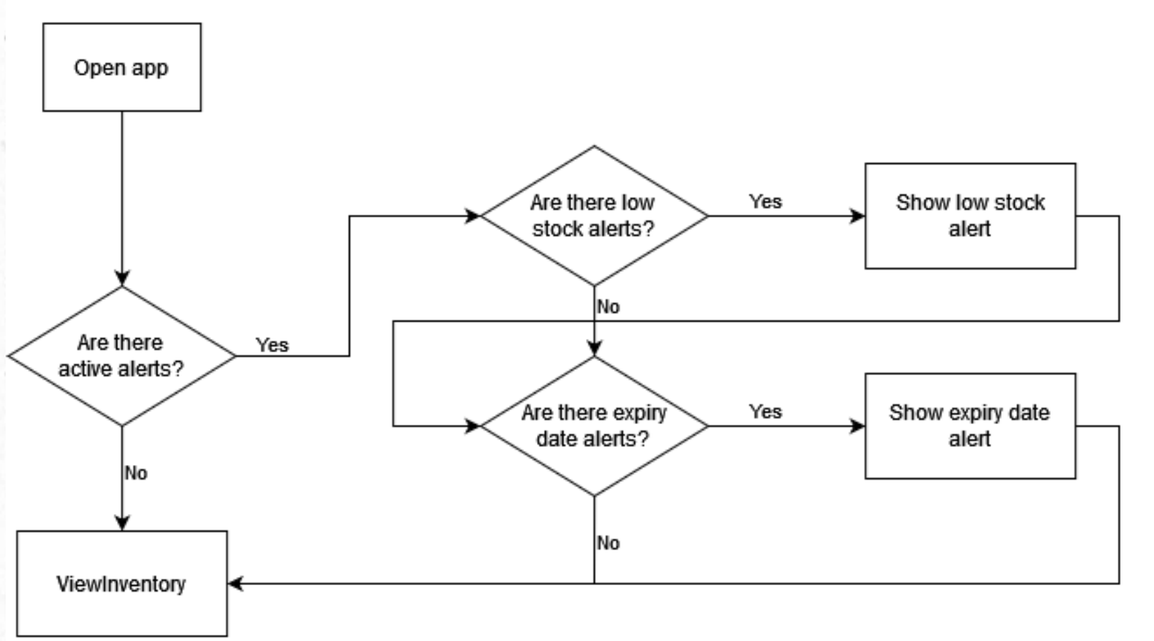
White Box Testing for Alerts

- Control Flow Diagram
  - Backend side
  - Checks for low stock each time an item is sold
  - Finds suitable sellers and their contact information
  - Sends alert to Frontend

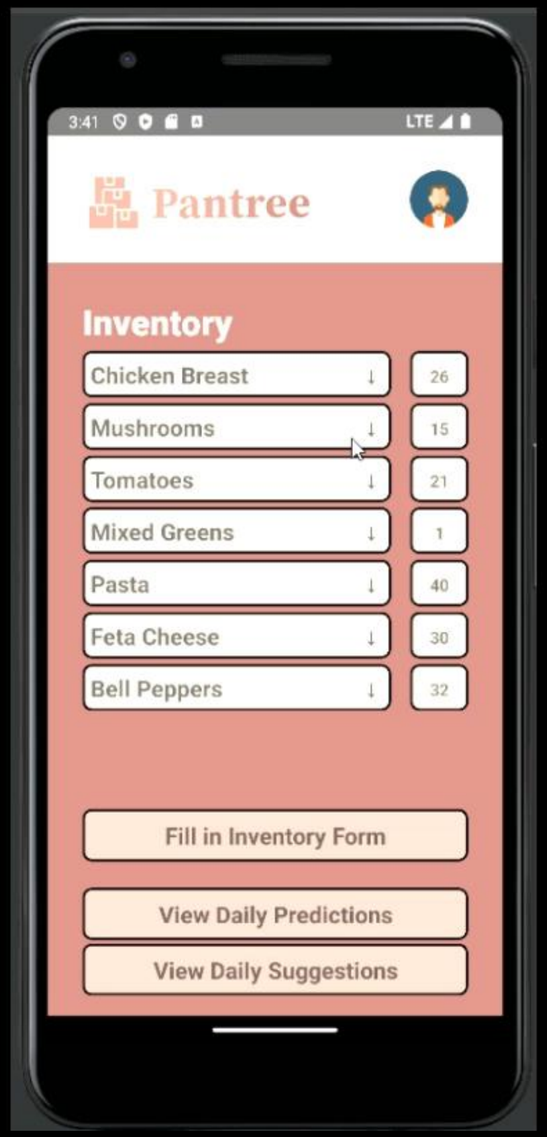


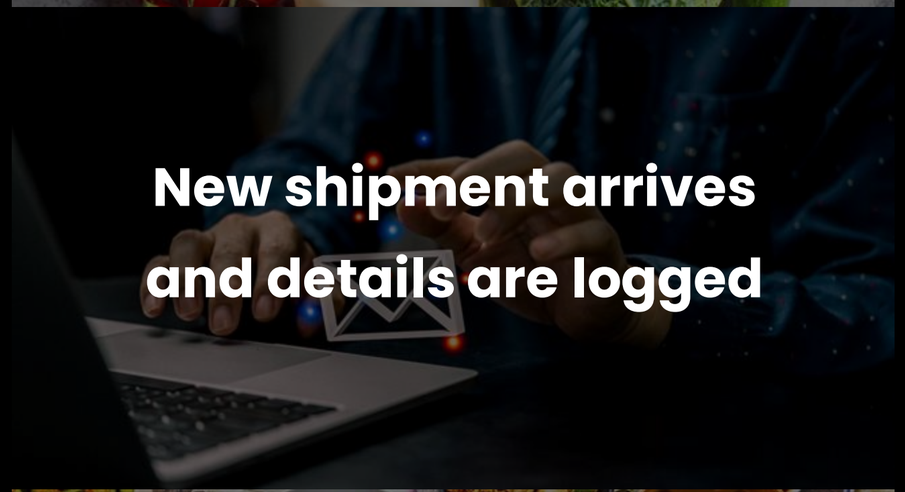
White Box Testing for Alerts

- Control Flow Diagram
  - User (Frontend) side
  - Checks for any alerts
  - Displays any alerts before showing Inventory page

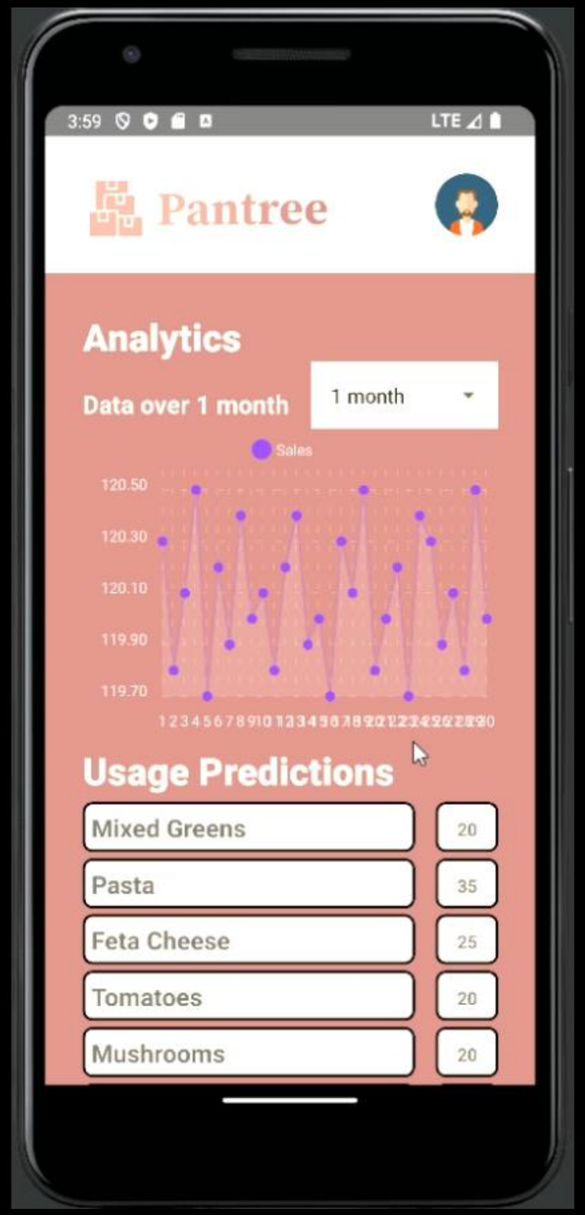


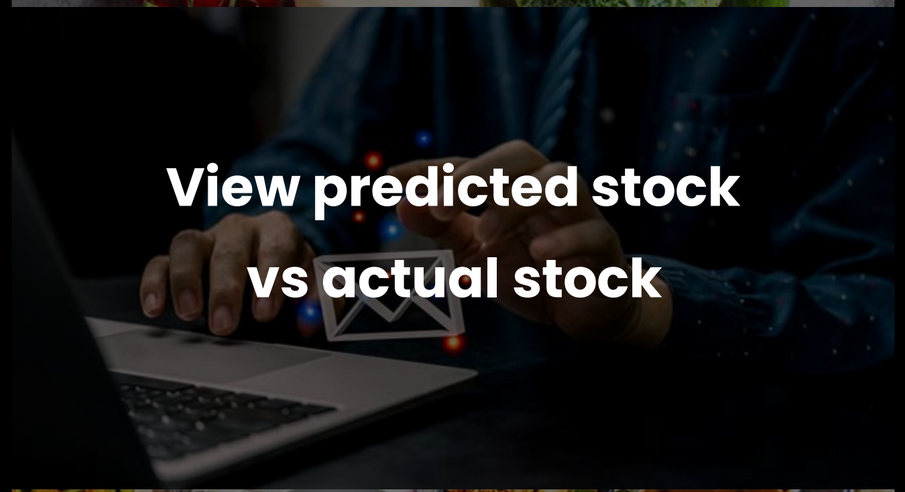
Feature 1 – Add & Display Inventory



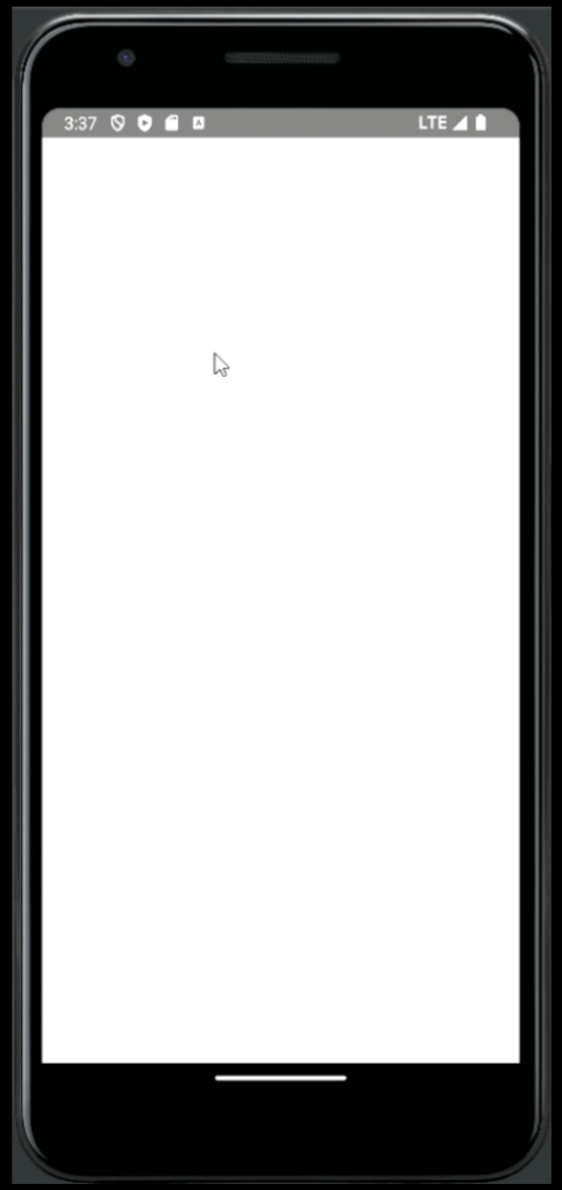


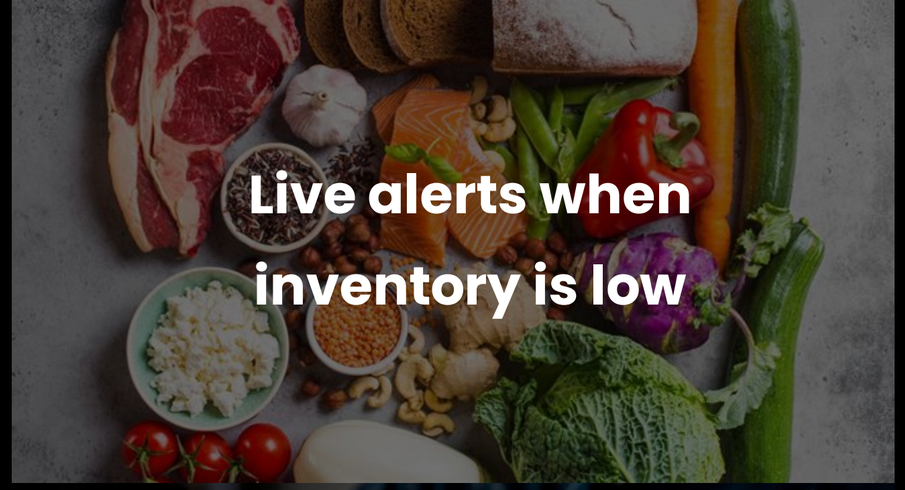
Feature 2 – Inventory Insights

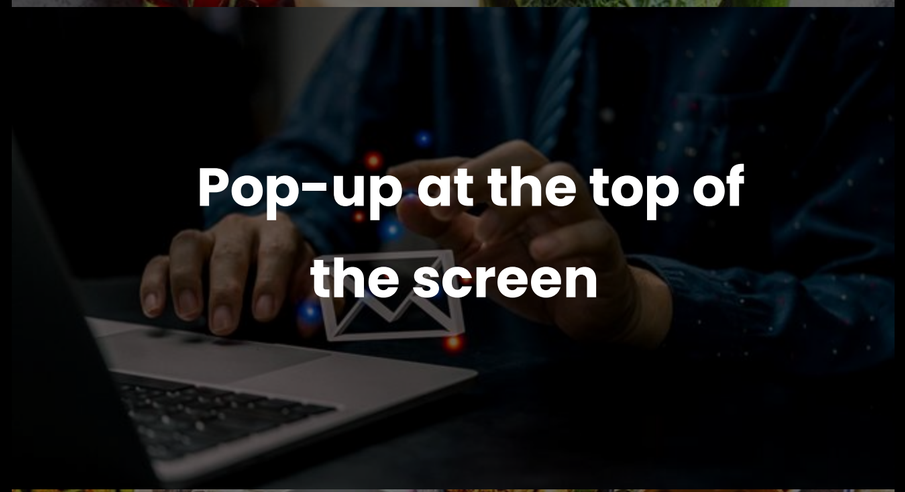




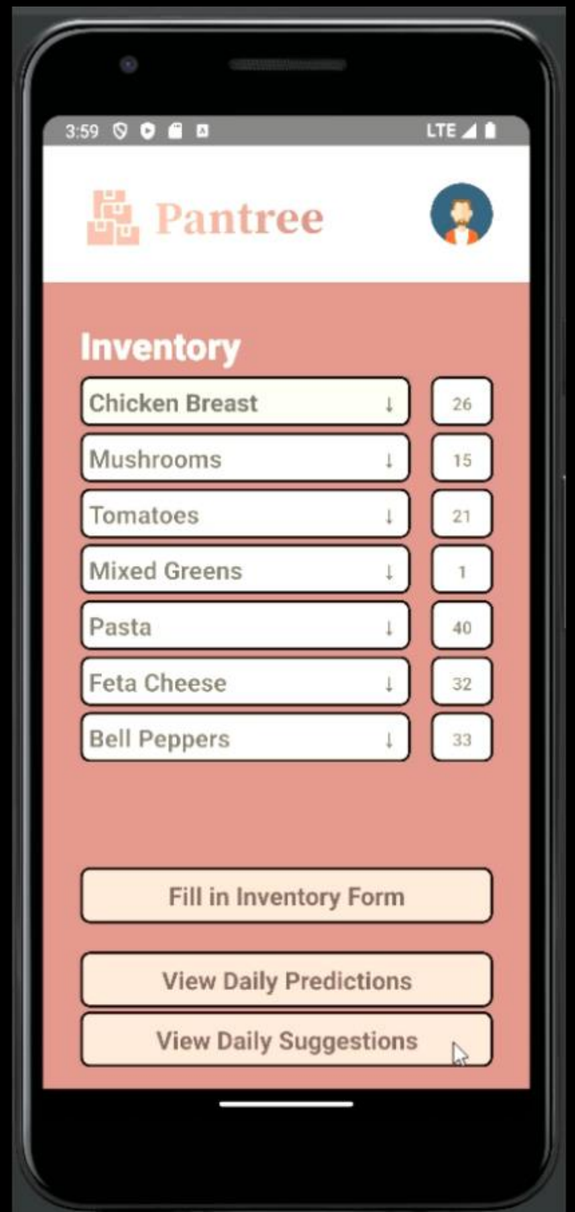
Feature 3 – Stockout Alert

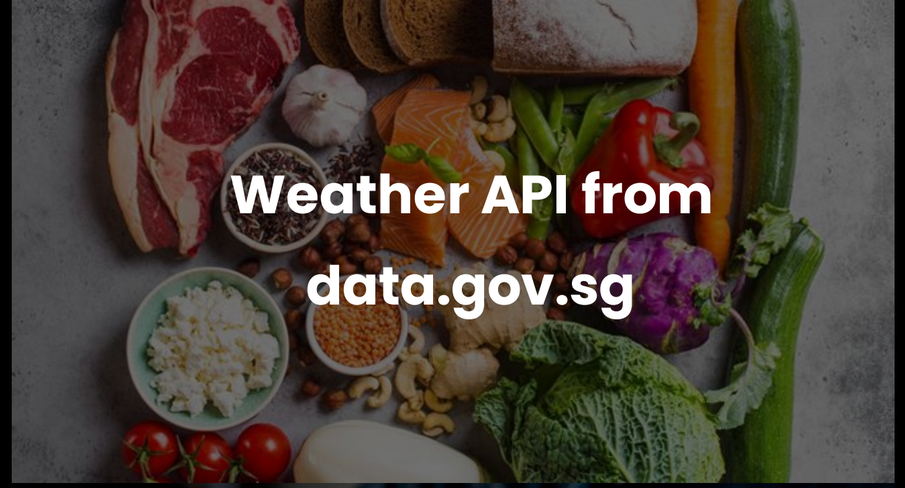


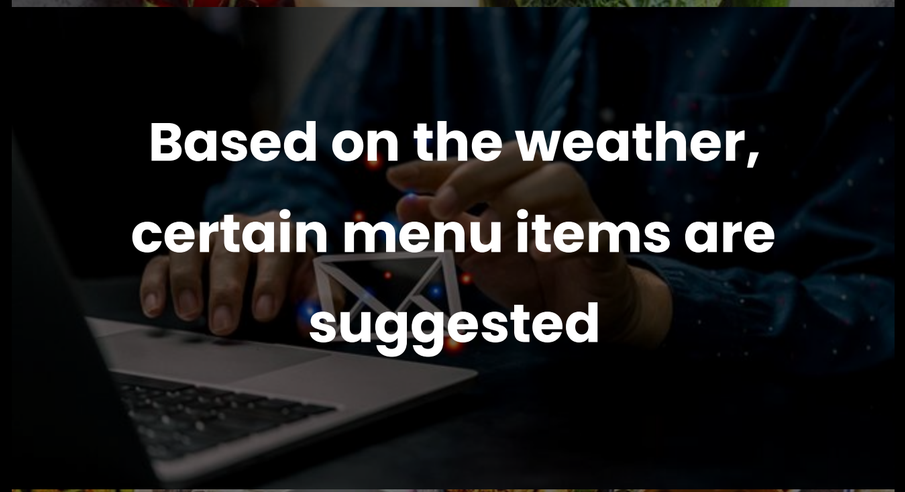




Feature 4 – Suggestions based on weather API







Intended Impact - Sustainable smart nation

Economic

- Reduced revenue loss through accurately matching supply & demand
- Generate extra revenue from unsold excess through Marketplace
- Cost savings from recycling, treatment & disposal (~$77/ tonne)

Social

- Reduction of food waste as a social issue to contribute to ESG efforts
- Cultivate social responsibility that aligns with company values
- Connecting with consumers to participate in food rescue

Environmental

- Reduce of food waste going to landfills
- Reduce wastage of resources & energy used to create ingredients
- Reduce increasing GHG footprint from food loss and waste

Our Value Proposition

Low barrier to entry due to low subscription costs

Helping F&B outlets achieve their ESG goals

Extra source of revenue through sales of mystery boxes

Seamless inventory planning

Cost savings through purchase of discounted items

Pantree Demo

w/ end-users

Warehouse Manager

Restaurant Manager

Marketplace Manager