

Interim Project Report

Askar Matayev, Ruslan Nagimov, Daniyar Kshibekov, Dana Koshkinbayeva
Department of Computer Science, Nazarbayev University
Astana, Kazakhstan

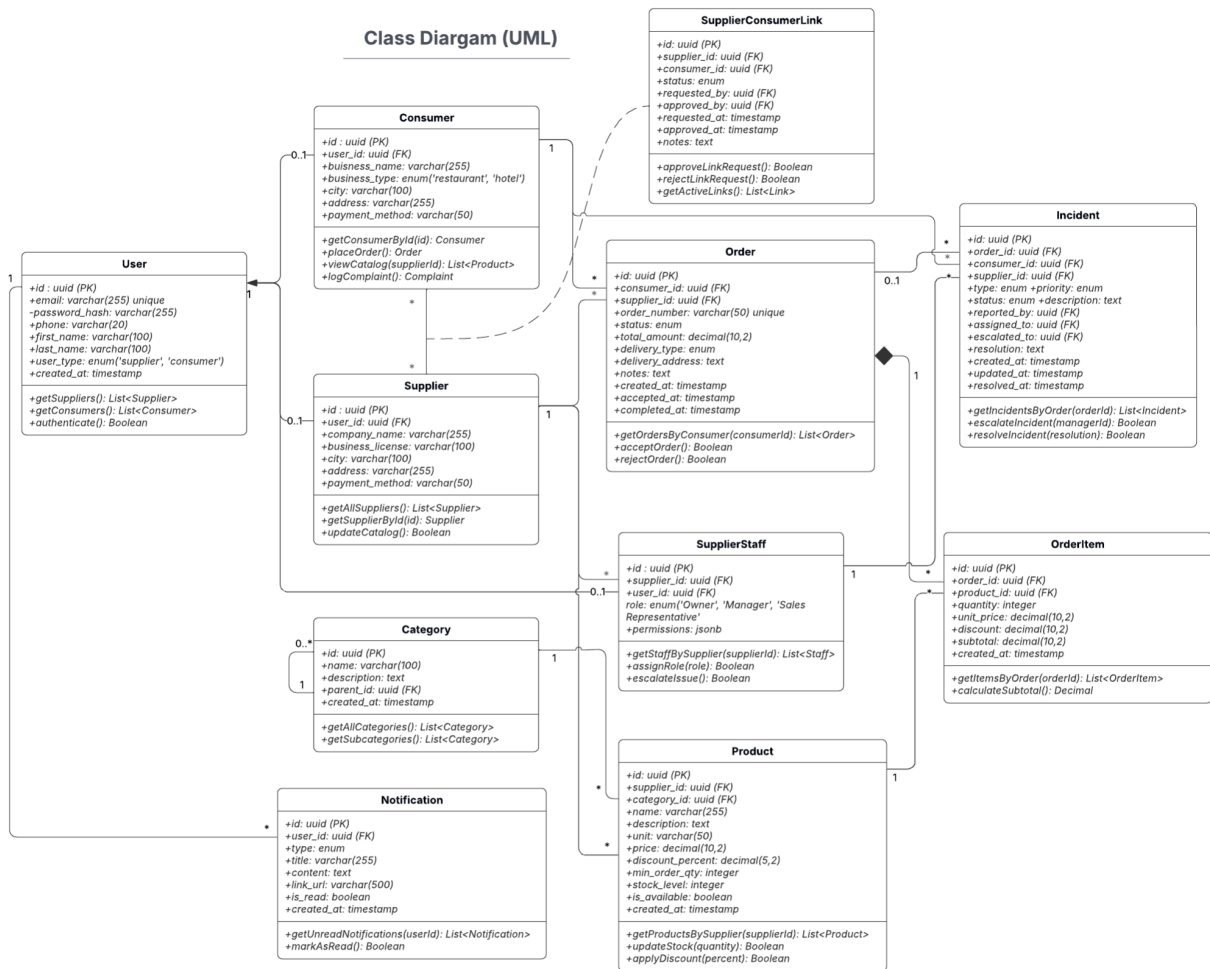
Table of Contents

- 1. Individual Contributions**
 1. Contributions
- 2. UML Diagrams**
 - 2.1. Class Diagram
 - 2.2. Use-Case Diagram
 - 2.3. Activity Diagram
 - 2.4. Component Diagram
 - 2.5. Sequential Diagrams
- 3. Additional Documentation and Evidence**
 - 3.1. Screenshots of Admin Panel
 - 3.2. API Endpoints Demonstration
 - 3.3. Git and Docker Configuration Snapshots
 - 3.4. Closed Tasks and Progress Indicators
- 4. Conclusion**
 - 4.1. Summary of Current Progress
 - 4.2. Future Work and Next Steps

UML Diagrams

Askar's contribution - UML design 3 diagrams

1. Class Diagram

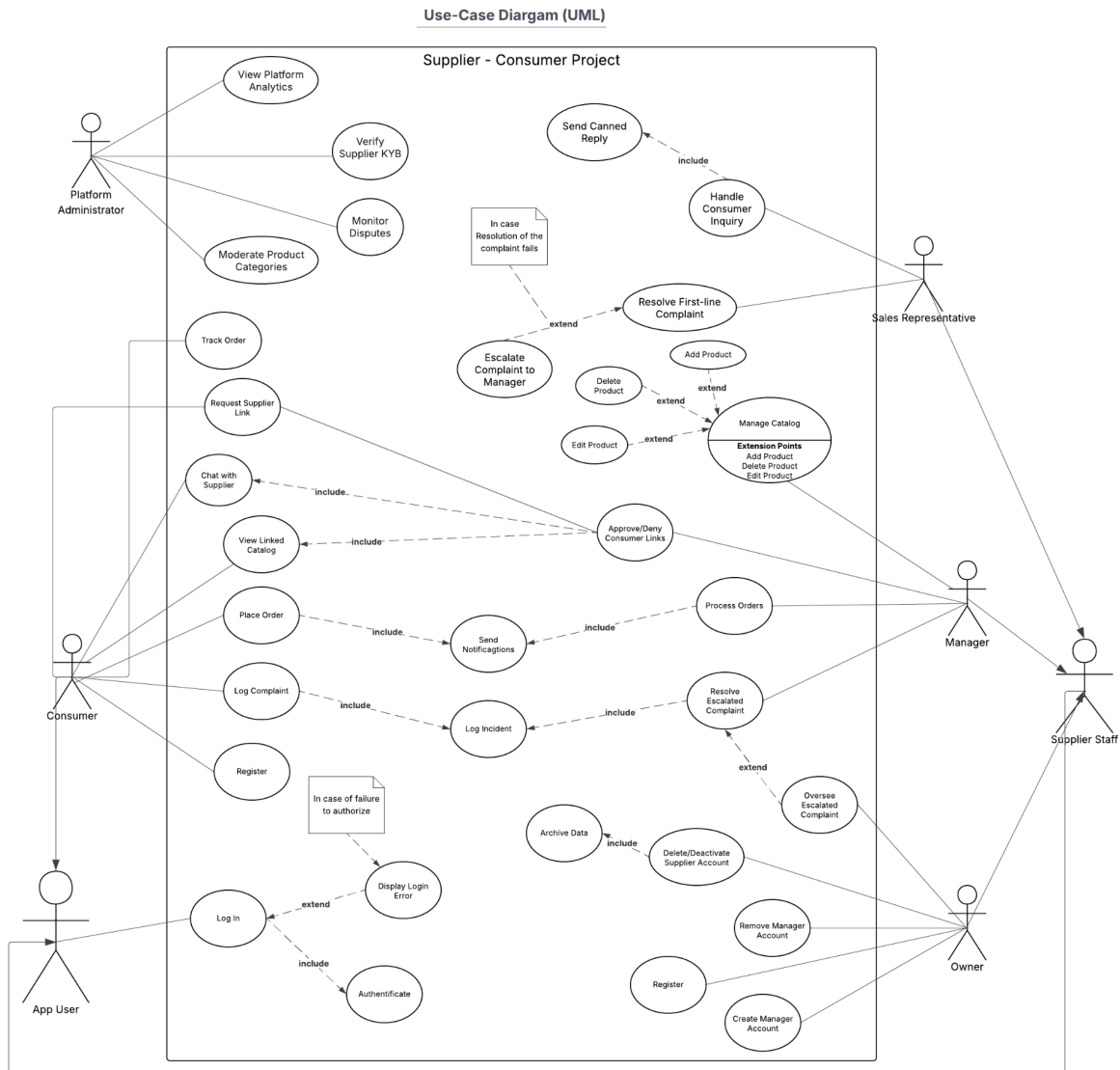


[Link to the class diagram](#)

In the UML Class diagram the Supplier Consumer Platform (SCP)'s core organization and plan were represented as a structured collection of entities, their attributes and methods as well as the relationships between those entities. Relying on the Software Requirements Specification document I created core entities (stakeholders) participating in the process and interacting with each other: Consumer and Supplier, which inherit the User entity to have access to user operation and to be able to have an account on the platform to authenticate. SupplierStaff was added to specify permission of different roles coming from the

supplier company: Owner, Manager, and Sales Representative which also inherit the User entity. SupplierConsumerLink was added as an Association Class resolving between many-to-many, having no direct access, and, most importantly, including the catalog visibility and order placement which will be requested by the consumer and denied/approved by the manager. The product entity is connected to the Supplier (manager) who updates it and divided into categories for the best use. Categories could have subcategories, so they have a recursive association with itself. The Order class has a composition relationship with the OrderItem class. While OrderItem records the transaction's specifics at the time the order is placed. Finally, Notification and Incident entities were added to satisfy the needed functionality.

2. Use-Case Diagram



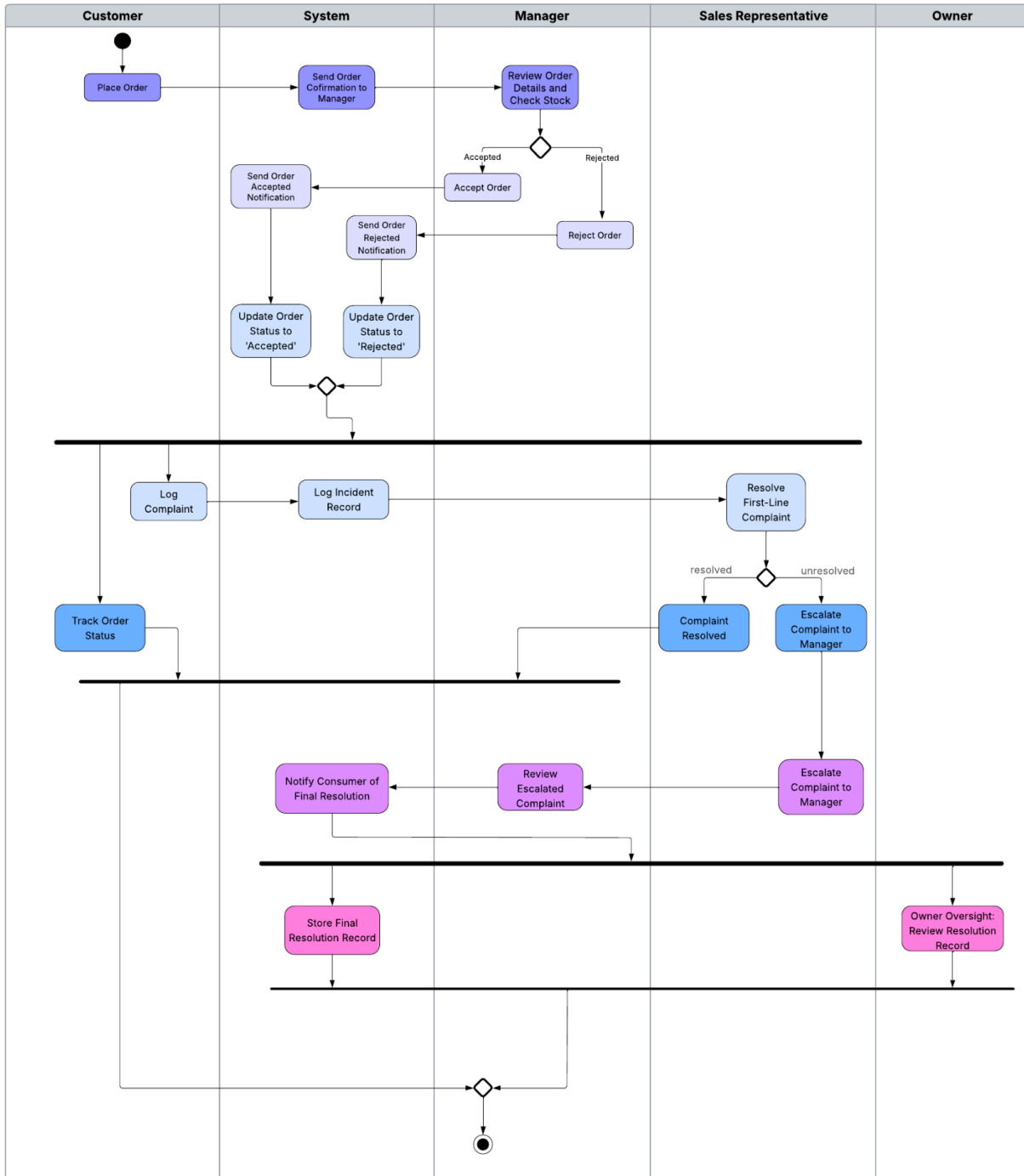
[Link to the use-case diagram](#)

Use-Case diagram (arguably the most difficult to design) demonstrates the functionality of the software for different actors (primary and secondary) which interact with each other via the use cases and relationships between them. All the actors inherit all the properties of an App User that can log in and authenticate. The main difference between B2B and B2C platforms that we've noticed is the link-based access requirement. In the diagram above I attempted to show how the consumer requests the supplier link or tries to place an order and waits for the

manager to approve or reject it. As was shown above, consumers can leave a complaint which is turned into a logged systematic incident. If the Sales Representative fails to satisfy the first-line complaint even after automatic canned inquiry, the complaint is escalated to the Manager who's responsible for resolving it. In addition, managers can also process the orders and update the products in the catalog. The owner has the most power among all supplier actors. He or she is overseeing managers' work, can create and delete/deactivate the manager's account. Finally, platform administrator was added for monitoring disputes, platform analytics and verifying KYB.

3. Activity Diagram

Activity Diagram (UML)



[Link to the activity diagram](#)

The Activity diagram shows all the stages and activities and the process of functionality.

Askar's Contribution continued:

Besides designing and learning how to design these three diagrams, I've learned everything about PostgreSQL and how to operate on it in PGAdmin (now I need to transfer things from diagrams), done courses on Flutter and built a demo following this Course -

<https://www.youtube.com/watch?v=IfUjHNODRoM&list=PL6yRaaP0WPkVtoeNIGqILtRAgd3h2CNpT>

Ruslan's contribution - UML design

4. Component diagram

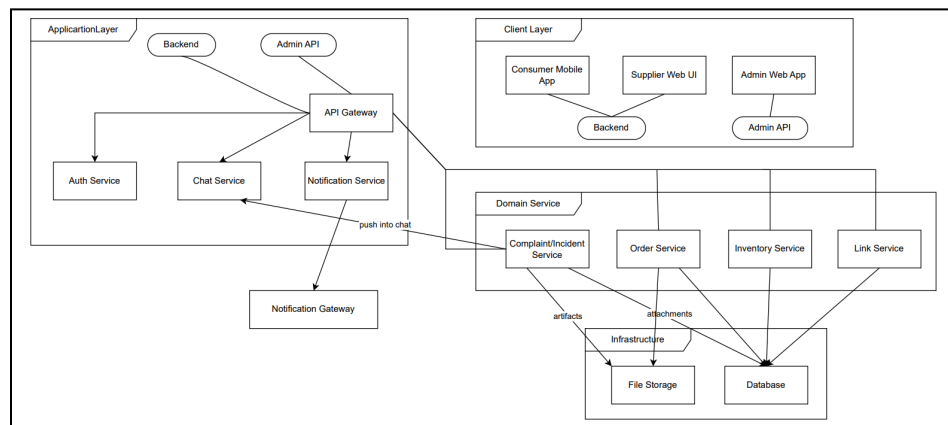


Figure 4. Component Diagram ([link](#))

This diagram proposes high-level structural organization of our Supplier-Consumer platform. Its goal is to illustrate how the system is modularized into components, how these modules interact, and which interfaces or dependencies they expose. There you can see four major components: Application Layer, Client Layer, Domain Service and Infrastructure. Client and Application Layers are frontiers between our application and user that will handle authentication, chat and notifications. Also API Gateway will route to requested services. Domain Service and Infrastructure are inner modules that will handle all data manipulations and movements.

5. Sequential Diagram

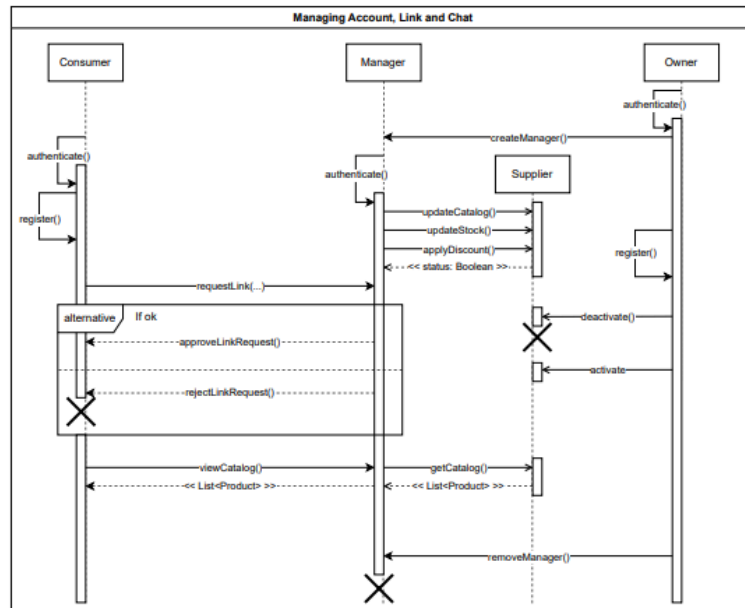


Figure 5. Sequential Diagram 1 ([link](#))

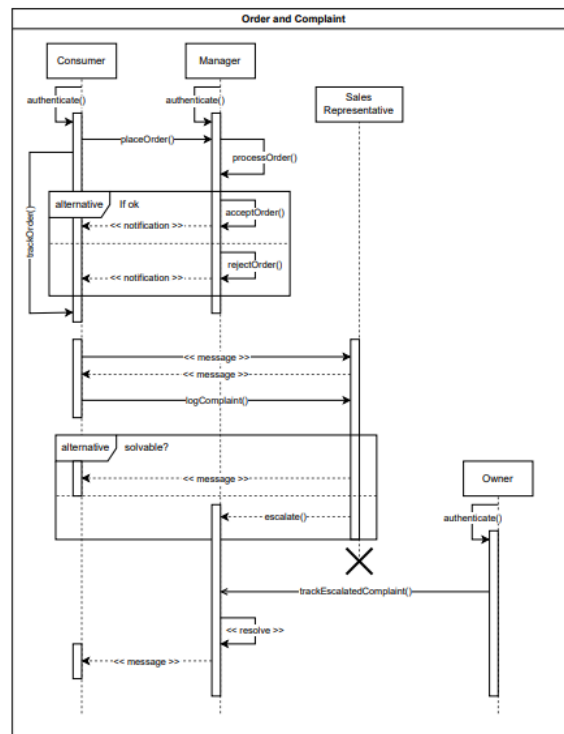


Figure 6. Sequential Diagram 2 ([link](#))

Sequential diagram describes step-by-step process of . Its goal is to illustrate how different core entities will interact during their lifespan within a system. We divided this diagram into two parts to isolate different processes and let actions of

one pipeline be depicted together. As it can be seen, all user entities can authenticate and some of them can register themselves as special users: Consumer, Owner. They can interact with each other using endpoints' calls or messages. Sales Representatives, Manager and Owner are three subentities of Supplier user. In our system it is reasonable to think about them as different users as these roles will probably be delivered to various employees to drive business.

Dana's contribution - FrontEnd

Low and high fidelity mockups in Figma. The first link contains static design mockups. The second link contains a clickable prototype with clickable navigation between screens.

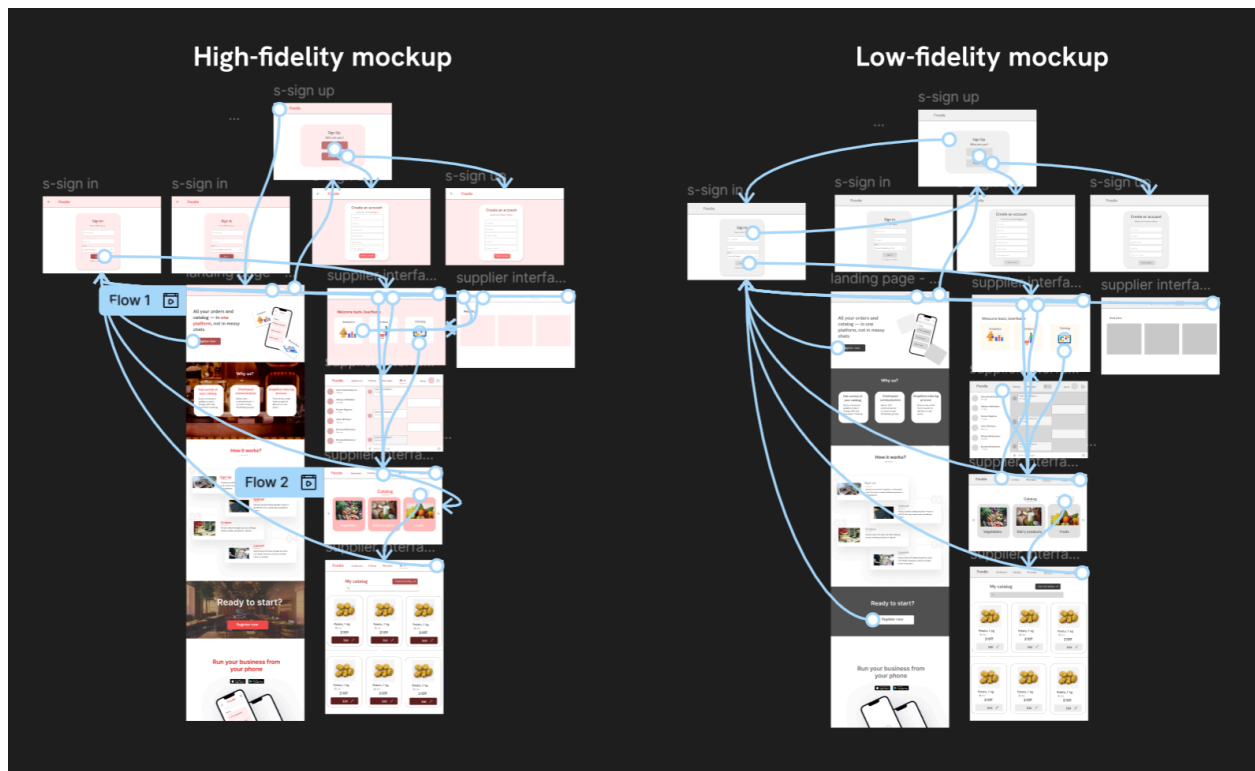


Figure 5. Figma design

Links to the design and prototype:

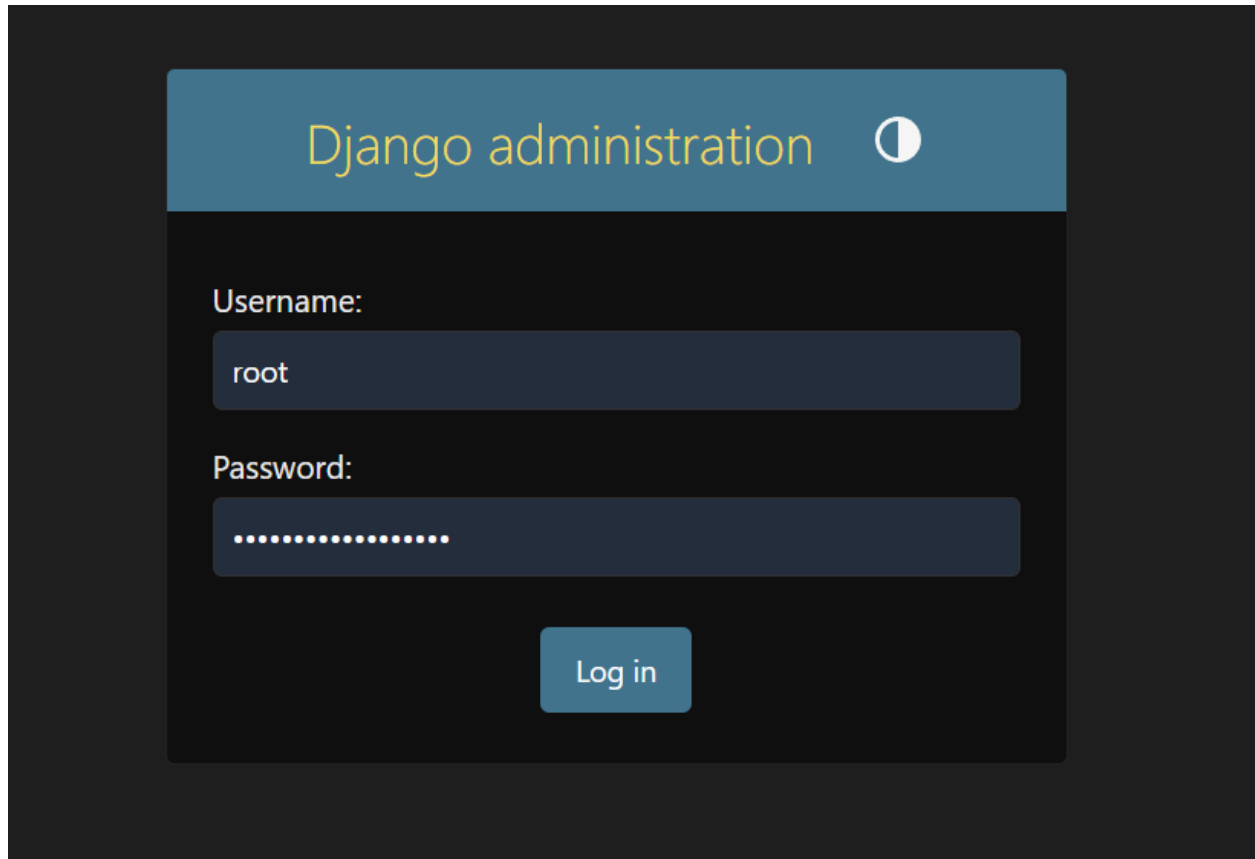
[Figma design](#)

[Figma prototype](#)

Daniyar's contribution - BackEnd

1. Authentication and Authorization

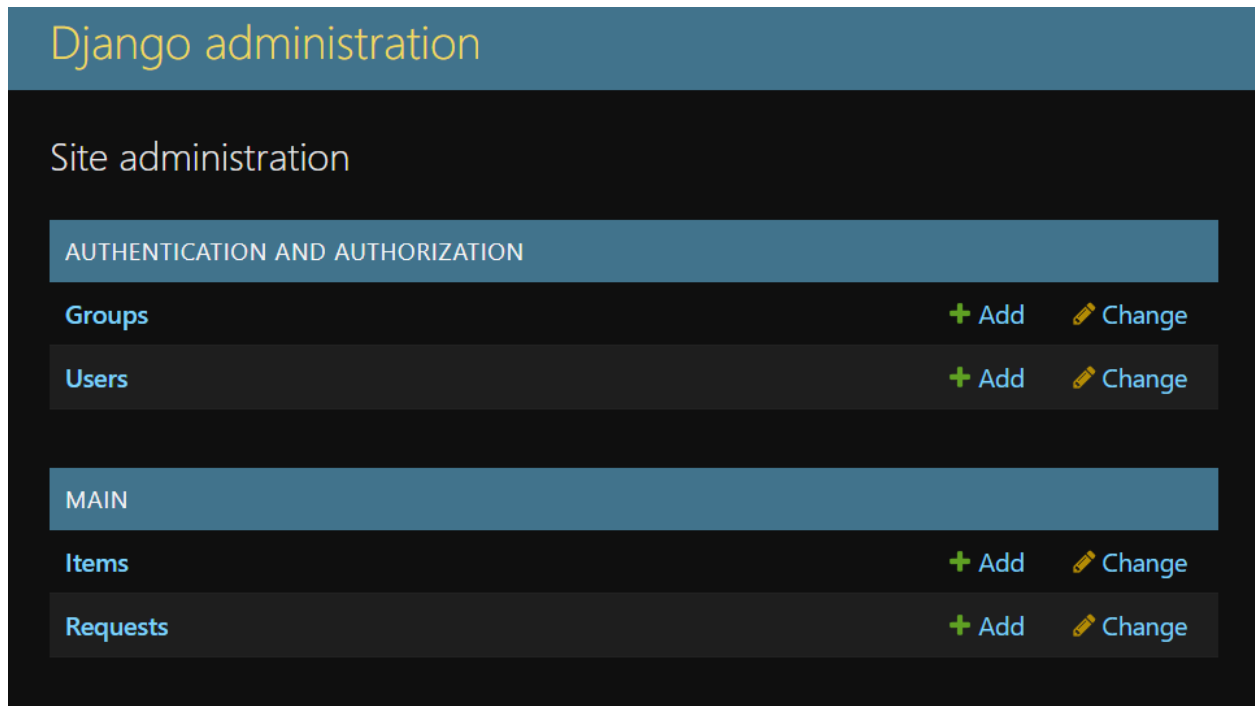
Admin's panel - <http://localhost:8000/admin/>

A screenshot of the Django administration login interface. The page has a dark background. At the top, there is a teal header bar with the text "Django administration" in a light yellow font, followed by a circular icon with a white crescent. Below the header, the login form is centered. It includes a "Username:" label, a text input field containing the text "root", a "Password:" label, and a password input field filled with dots. At the bottom of the form is a teal "Log in" button.

Access to data modification requires entering the administrator username and password.

This rule was added to protect the system and keep the data secure.

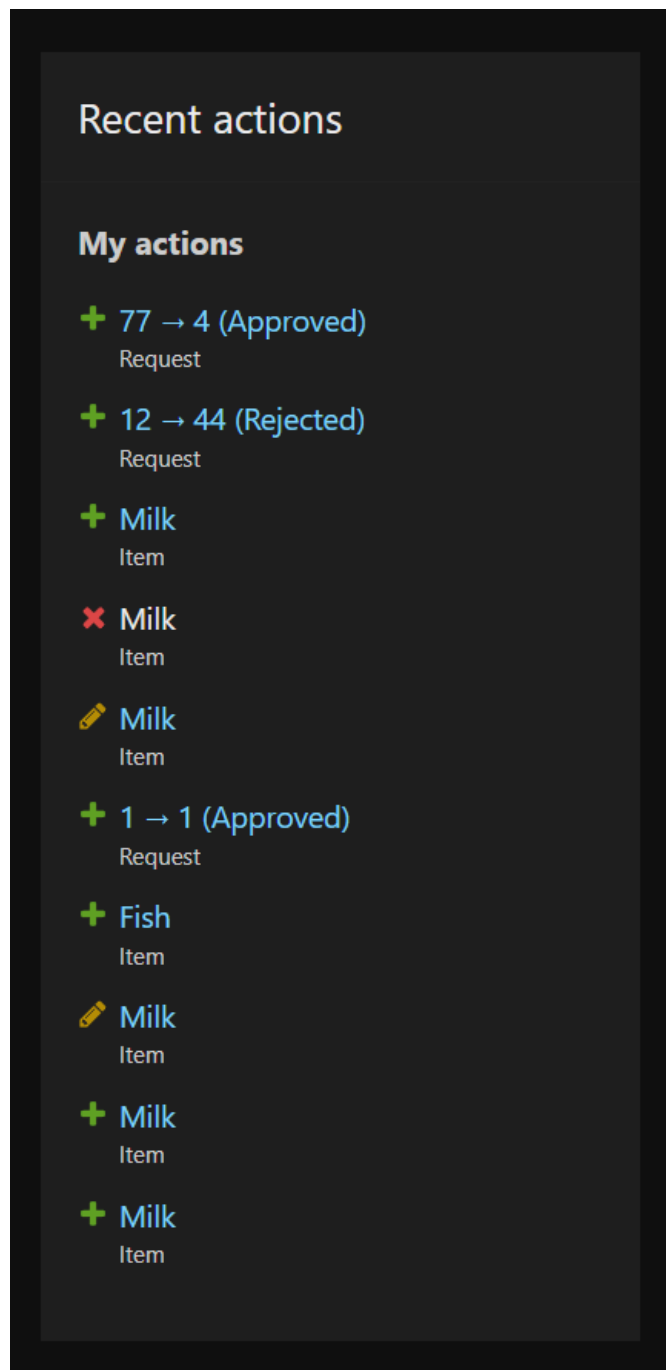
2. Site Administration Overview



After gaining access to the system, the administrator can add or change information in the Items and Requests sections.

These actions allow the user to update the database with new records or modify existing ones.

3. Managing Data Entities



I have already tested the system by adding and changing data. Now I will demonstrate how it works.

Add item

Name:	<input type="text"/>
Description:	<input type="text"/>
Price:	<input type="text"/>
Weight:	<input type="text"/>
Quantity:	<input type="text"/>
SupplierID:	<input type="text"/>

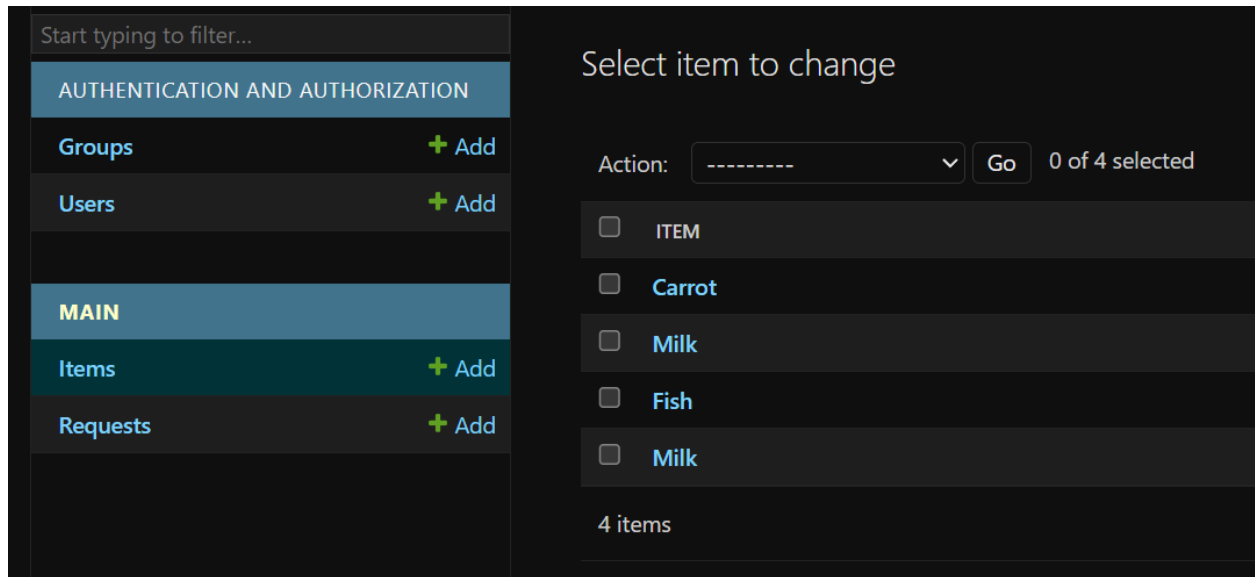
SAVESave and add anotherSave and continue editing

In this section we can add new items to the table called Items in the database.

We need to fill in the fields Name, Description, Price, Weight, Quantity, and SupplierID.

After that we can save and close the form, save and add another item, or save and stay on the same page to make changes to the current entry.

4. Editing and Deleting Data



In this part we can edit existing data.

It is possible to delete records one by one or select several records and delete them together as shown in the picture below:

Select item to change

Action: Delete selected items ▾ Go 2 of 4 selected

<input type="checkbox"/>	ITEM
<input checked="" type="checkbox"/>	Carrot
<input checked="" type="checkbox"/>	Milk
<input type="checkbox"/>	Fish
<input type="checkbox"/>	Milk

4 items

We can also click on any record to open it and change its data.

Change item

Milk HISTORY

Name:

Description:

Price:

Weight:

Quantity:

SupplierID:

SAVE Save and add another Save and continue editing Delete

Here we can change the information, save it using different options as described before, or delete the record if needed.

Change history: Milk

DATE/TIME	USER	ACTION
Oct. 17, 2025, 1:44 p.m.	root	Added.
1 entry		

There is also a history section that shows all changes made to the data.

Select request to change

Action: 0 of 3 selected

<input type="checkbox"/>	REQUEST
<input type="checkbox"/>	77 → 4 (Approved)
<input type="checkbox"/>	12 → 44 (Rejected)
<input type="checkbox"/>	1 → 1 (Approved)

3 requests

We can perform the same actions with the Requests section.

Change request

77 → 4 (Approved)

ConsumerID:

77

SupplierID:

4

Status:

Approved

ConsumerComment:

I am your biggest fan! Could you sell me 1000 kg of fish?

SAVE

Save and add another

Save and continue editing

This table represents the structure of the Requests entity.

It includes the fields ConsumerID, SupplierID, Status, and ConsumerComment.

Each record can be saved, updated, or deleted as needed.

Add request

ConsumerID:	<input type="text"/>
SupplierID:	<input type="text"/>
Status:	<input type="text"/>
ConsumerComment:	<div></div>

SAVE

Save and add another

Save and continue editing

It also uses the same adding system as the Items entity.

Now the first part is completed, and we can continue to the second part.

5. Searching and Ordering by

Api Root

The default basic root view for DefaultRouter

```
GET /api/
```

```
HTTP 200 OK
```

```
Allow: GET, HEAD, OPTIONS
```

```
Content-Type: application/json
```

```
Vary: Accept
```

```
{  
  "items": "http://localhost:8000/api/items/",  
  "requests": "http://localhost:8000/api/requests/"  
}
```


In this section we have two links:

Items: <http://localhost:8000/api/items/>

Requests: <http://localhost:8000/api/requests/>

Api Root / Item List

Item List

 Filters

OPTIONS

GET ▾

GET /api/items/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[
  {
    "id": 1,
    "name": "Milk",
    "description": "Super tasty",
    "price": 999.0,
    "weight": 1.0,
    "quantity": 178,
    "supplierID": 12
  },
  {
    "id": 3,
    "name": "Fish",
    "description": "fish 100% fresh",
    "price": 5000.0,
    "weight": 3.0,
    "quantity": 12,
    "supplierID": 55
  },
  {
    "id": 4,
    "name": "Milk",
    "description": "white milk",
    "price": 800.0,
    "weight": 2.0,
    "quantity": 1231,
    "supplierID": 1
  },
  {
    "id": 5,
    "name": "Carrot",
    "description": "vegetable",
    "price": 300.0,
    "weight": 0.5,
    "quantity": 33
  }
]
```

If we open the Items link, we can see the structure and data of the entries that were created earlier.

There is also a Filters button that allows us to sort or search for specific records.

Filters



Ordering

price - ascending

price - descending

weight - ascending

weight - descending

Search

If we click the Filters button, we can search for information by name or sort the data by price or weight in ascending or descending order.

Filters



Ordering

price - ascending	✓
price - descending	
weight - ascending	
weight - descending	

Search

MILK

🔍 Search

Item List

🔧 Filters

OPTIONS

GET ▾

GET /api/items/?ordering=price&search=MILK

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
[
  {
    "id": 4,
    "name": "Milk",
    "description": "white milk",
    "price": 800.0,
    "weight": 2.0,
    "quantity": 1231,
    "supplierID": 1
  },
  {
    "id": 1,
    "name": "Milk",
    "description": "Super tasty",
    "price": 999.0,
    "weight": 1.0,
    "quantity": 178,
    "supplierID": 12
  }
]
```

In this example I selected Order as price ascending and entered MILK in the search field.

The system displayed two suppliers who sell milk, arranged in ascending order by price.

There is also an option to add new data, just like in the administrator panel:

[Raw data](#) [HTML form](#)

Name

Description

Price

Weight

Quantity

SupplierID

POST

We can also view the data stored in the Requests table:

Api Root / Request List

Request List


OPTIONS GET

GET /api/requests/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[
  {
    "id": 1,
    "consumerID": 1,
    "supplierID": 1,
    "status": "Approved",
    "requestDate": "2025-10-17",
    "consumerComment": "I wanna buy your milk"
  },
  {
    "id": 2,
    "consumerID": 12,
    "supplierID": 44,
    "status": "Rejected",
    "requestDate": "2025-10-17",
    "consumerComment": "Please I need to buy some food"
  },
  {
    "id": 3,
    "consumerID": 77,
    "supplierID": 4,
    "status": "Approved",
    "requestDate": "2025-10-17",
    "consumerComment": "I am your biggest fan! could you sell me 1000 kg of fish?"
  }
]
```

We can also add new data to the Requests table when needed:



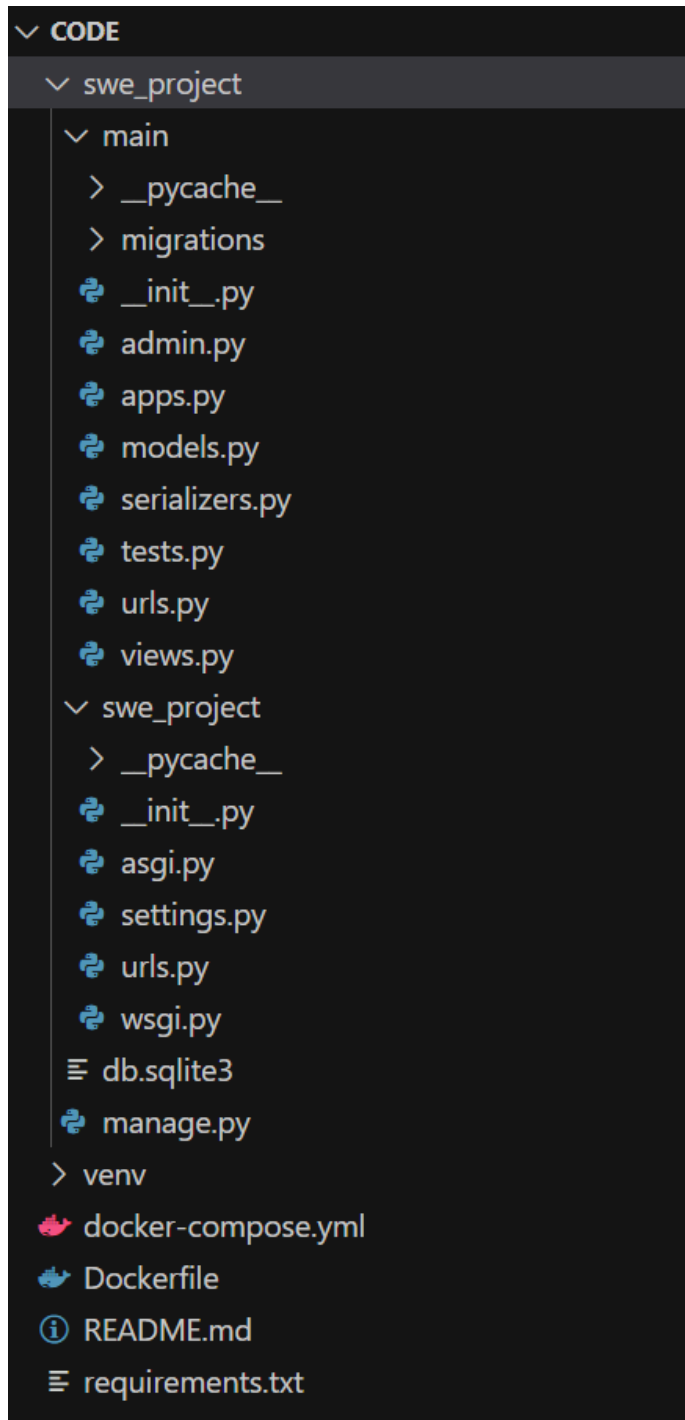
The screenshot shows a web interface for adding new data to the Requests table. At the top right, there are two tabs: "Raw data" (highlighted in red) and "HTML form". Below the tabs is a form with four input fields: "ConsumerID", "SupplierID", "Status", and "ConsumerComment". Each field is a white rectangular box with a light gray border. The "ConsumerComment" field is a larger text area. At the bottom right of the form is a blue button labeled "POST".

6. Learning Path

Before starting this project, I had very little knowledge of the technologies used in it.

To complete the work, I studied Git and Django through online tutorials and learned how to use Docker and PostgreSQL from the beginning.

7. Structure of the Project



The structure of the project is shown above.