University of Regina

Software Engineering (CS-872)

Heritage Hive

Submitted to: Dr. Samira Sadaoui

Submitted on: 27th March, 2024

Nasik Sami Khan	**200496882**	**nku618@uregina.ca**
Md Rezaul Hoque	**200484979**	**mrh165@uregina.ca**
Sumaiya Nawsheen	**200496599**	**snq745@uregina.ca**

Table of Contents

1. Problem Definition

The primary challenge is that artists and producers of traditional goods are not able to reach international markets. These people, who are frequently from isolated or rural locations, have few opportunities to exhibit and market their culturally significant goods, such as traditional attire, tools, crafts, and decorations. This ultimately results in an overall decline in the practice of various crafts, which affects both the diversity of culture and the livelihoods of these artists and possible extinction of the cultural heritages.

We propose to develop an e-commerce platform named "Heritage Hive". It is an innovative e-commerce platform designed to connect local artisans from around the world with a global customer base. Its mission is to facilitate the sale & exhibition of heritage items, thereby aiding in the preservation of global cultural heritage. The platform supports local economies by empowering small businesses and rural families by getting remittance, and it also serves an educational purpose by providing detailed cultural information about each item. With a focus on user-friendly design and sustainable, ethical trade practices, "Heritage Hive" aspires to be more than just a marketplace. The platform will serve as a global marketplace, bringing together artisans and buyers in a unique online space where the rich heritage of different nations is showcased and exchanged.

2. Application Benefits

The platform "Heritage Hive" is an innovative e-commerce solution that offers major benefits to regional artists, their communities, and the conservation of world cultural heritage. Fundamentally, this platform bridges the gap between local artisans in remote or isolated places and a worldwide consumer base, thereby addressing the critical issue of economic sustainability for producers of traditional goods. This relationship is more than just commercial, it serves as a means of reviving and preserving traditional crafts that face extinction.

Two related e-commerce website for comparing with our platform:

1. https://www.arngren.net/

Design and Usability:

Heritage Hive has a modern, clean, and intuitive interface considering its focus on a user-friendly design. This approach is crucial for engaging a range of users that may not have extensive experience shopping online and prefers straightforward, accessible tools.

Arngren.net, on the other hand, is recognized for its cluttered layout and outdated design, making it challenging for users to find what they're looking for. This disconnect underlines how important design and user experience are to drawing in and keeping users.

Features and Services:

Heritage Hive sets itself apart by focusing on a specific target customer base and offers product from several categories. Gives the feature of search, filter, recommendation, semantic search, and various payment gateways.

The website arngren.net suffers from a lack of clear organization and an outdated design. It doesn't have any filter option to choose from. It has very few payment integration options. It doesn't have a clear, understandable product description.

Overall, the usability and design of the website arngren.net present a serious problem. It is packed with a huge variety of goods, from cars to electronics, yet the arrangement is incredibly disorganized. The website has an antiquated layout that might be confusing for users due to its unclear structure, making it challenging to browse or find certain content. Images and product details are packed together with little space between them, making the experience visually overwhelming. Furthermore, the website's design does not adhere to contemporary web standards, which may have an impact on how easily navigable it is across a range of screens and devices.

2. https://www.homebase.co.uk

Homebase website has issues such as keyword stuffing, repetitive product images, and limited product descriptions may not be immediately visible without analyzing a specific product page, especially for bathroom furniture.

Content Placement: Important information is placed at the bottom, reducing visibility. Whereas it is far better in our Heritage Hive platform, which enhances the user experience.

Product Representation: The limited range of "bestselling" and clearance items may not inspire purchase confidence for the users. Also some of the product reviews says that the product shown in the website and the actual product are not similar. Which is not the case for our Heritage Hive platform.

Service Descriptions: Misleading in-store service descriptions could confuse customers on the website. They can reorganize content to prioritize valuable information. C Improve photo quality and ensure consistent design aesthetics. In the Heritage Hive platform, we have focused on these details.

Design Quality: Problems with photo quality and text alignment detract from the overall user experience. They could clarify and enhance product and service descriptions. This focus on accessibility and personalized support makes Heritage Hive a great resource for the buyes to shop from.

In conclusion, Heritage Hive offers a multifaceted solution to the challenges faced by local artisans and their communities. By empowering artisans economically, preserving cultural heritage, and strengthening local economies, the platform sets a new standard for how traditional goods are marketed and sold globally.

3. Requirements Elicitation and Specification

3.1 Functional Requirements

2 user roles : Seller, Buyer

- **Main Functional Requirements for Buyers:**

a. Account Creation: Users will be able to create and login an account on the platform.

b. Product Search: Users will be able to search their desired products using natural language (Semantic Search - AI Feature)

c. Product Selection: Adds selected products to the cart.

d. Select Variations: Selects the product quantity and variations.

e. Cancel: Cancels or removes the products if the user changes their mind.

f. Checkout: Proceeds to product checkout and fills in the customer information.

g. Review: Users will be able to review the product.

h. Rating: Users will be able to rate the product on a scale based on their satisfaction.

i. Payment: Completes the payment of the product.

j. Social Media Share: Users will be able to share the product across multiple social media platforms.

● **Main Functional Requirements for Sellers:**

a. Product add: Add products to the store.

b. Product update: Update product information and stock items in the marketplace.

c. Product remove: Remove products that are not available in the inventory.

d. Order Management: Seller will be able to view and manage incoming orders from buyers and contact them if required.

3.2 Use case diagram for each user role

- Use case diagram for Buyer

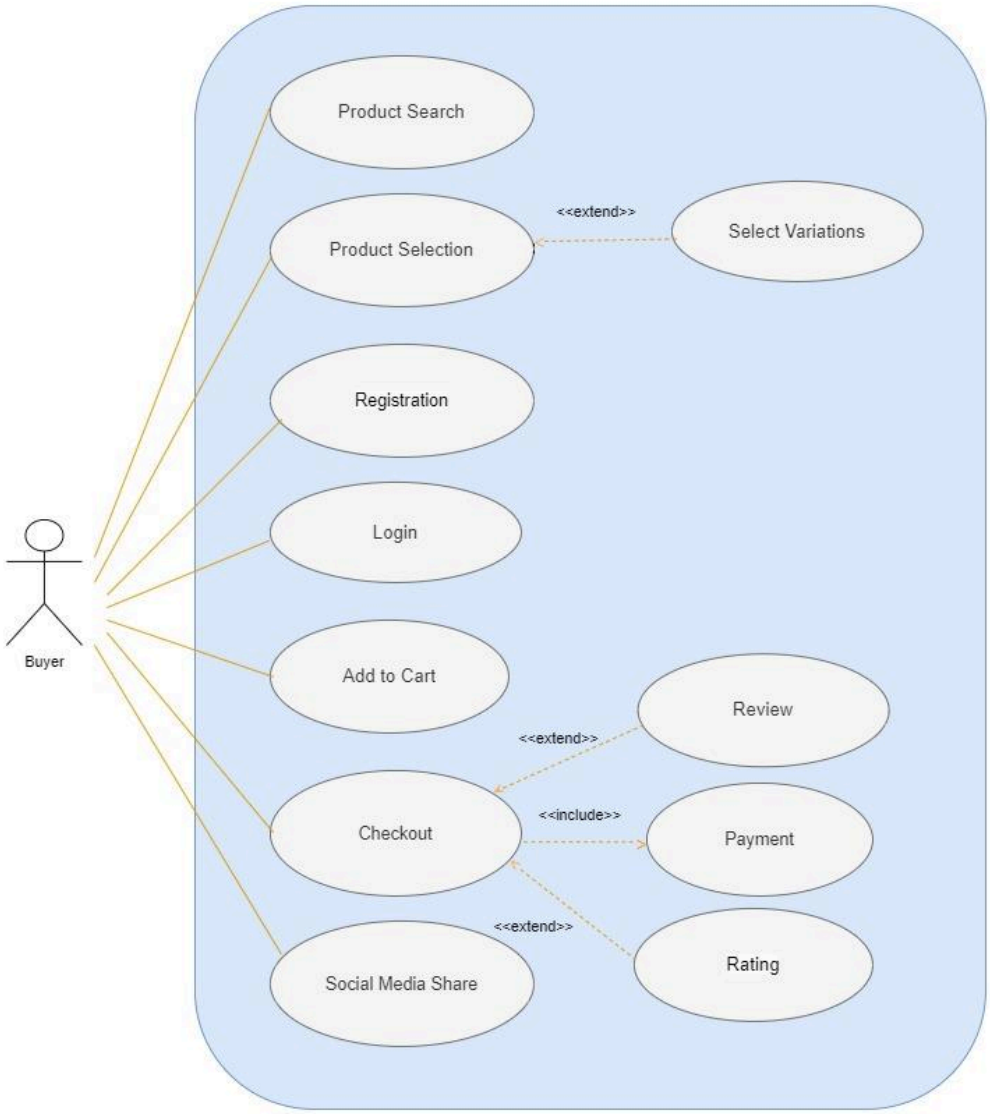

Fig.1: Use case diagram for Buyer

This Buyer Use case diagram demonstrates the functional requirements for Buyers interacting with the system. The main actor, The Buyer, can initiate various actions. These include searching for products, viewing product details, selecting products with variations (like size or color),

potentially adding wanted items and removing unwanted items from their cart, creating an account, login to the account and finally checking out. The checkout process includes payment and extended rating and review. Additionally, buyers can share their experience on social media platforms.

- Use case diagram for Seller

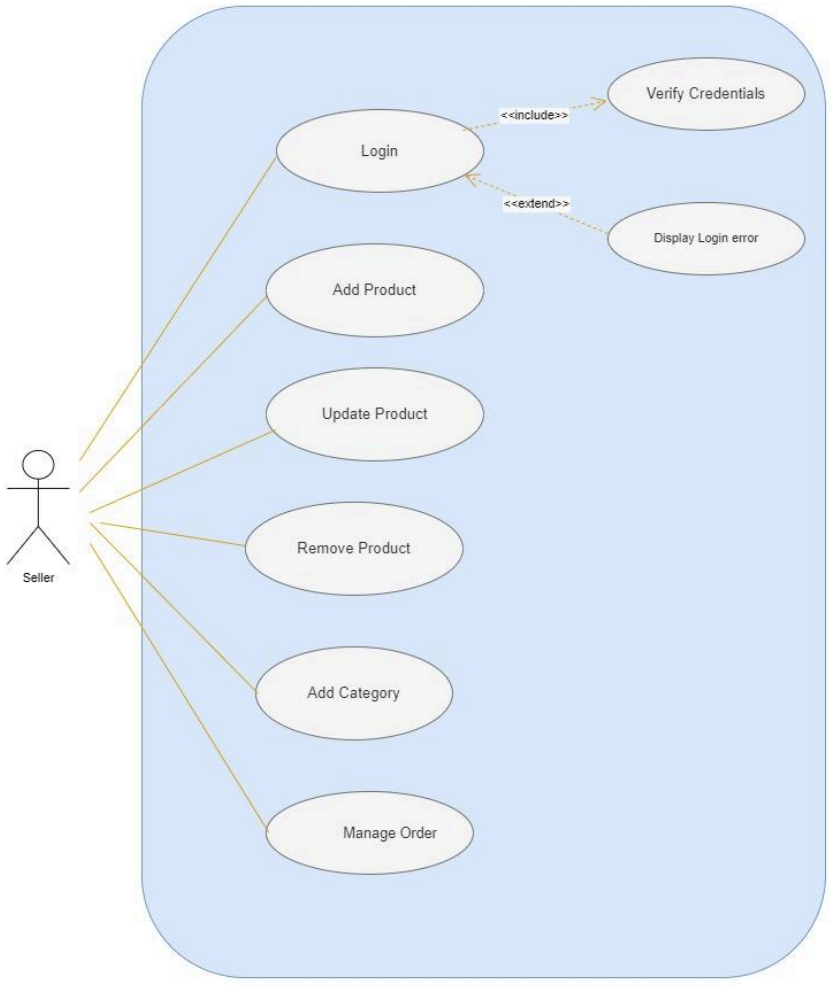

Fig. 2: Use case diagram for Seller

The use case diagram for Seller outlines functionalities for managing products and user interactions. Sellers can log in with credential verification, potentially encountering login errors. Once logged in, they can add new products with details, update existing product information, or remove products from the system entirely. Additionally, sellers can create product categories for better organization. The diagram also highlights functionalities for managing orders, potentially including tracking order status, processing orders, and interacting with buyers. Depending on the system's design, managing user information or access levels might also be a seller function. This use case diagram showcases the core functionalities for seller interaction and product management.

3.3 Activity Diagrams for Use Cases

The activity diagram for the buyer's portal is depicted below:

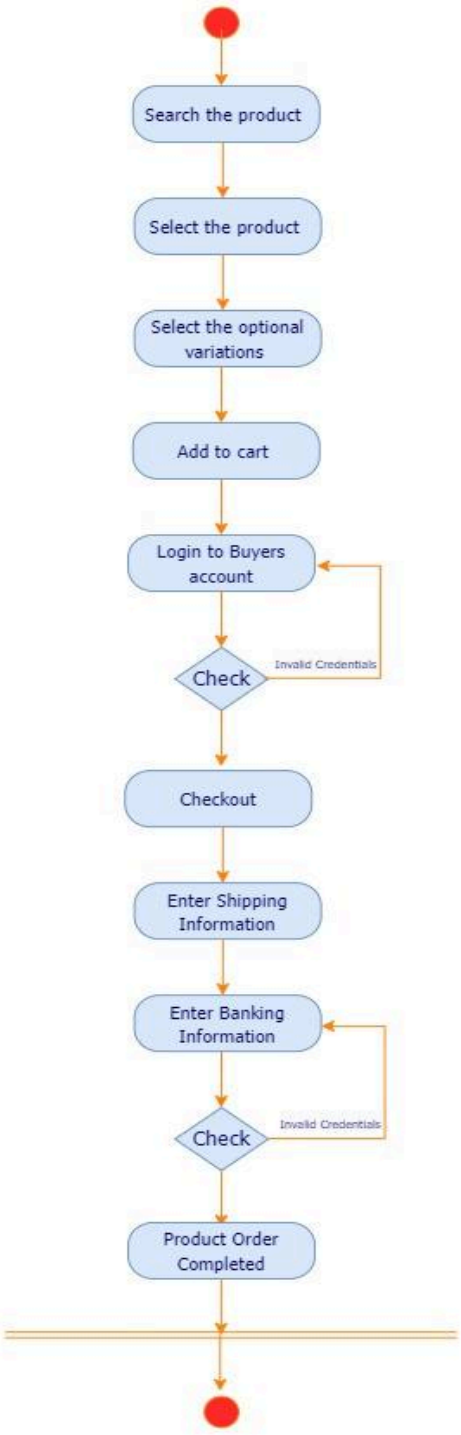

Fig 3: Activity Diagram from the buyer's scenario

The activity diagram for the buyer's scenario provides a representation of the user interaction flow within our e-commerce platform, starting from the initiation of an order to its conclusion in the life cycle. The process begins when a user initiates an order ('Start Order'). The user is then presented with two primary options to locate a product: by using the 'Search Product' function or by browsing through various 'Categories'. In our Heritage Hive project, we offer a semantic search feature for the user as well, where the user will be able to search for their desired product in their natural language. The system then shows all the relevant products in the interface for the user. If the desired product is found via search, the user views the product's details ('View Product'). For products with options, users can 'Select Product Variation' such as size color, or other variations. Once the selection is made, the item is added to the shopping cart ('Add to cart'). If the user has not previously registered on the platform, they are prompted to 'Register User'. For registered users, the system checks if they are 'Authenticated?'. If not, they must 'Login Properly' using the correct user ID and password. After logging in, the user can use 'View Cart', where they can review the items selected. Here, the user can 'Update Cart', modifying items or quantities as needed. When satisfied with the cart's contents, the user proceeds to 'Checkout and enter details', filling in necessary shipping and billing information. Then, they must 'Checkout: Choose Payment Type', selecting a preferred method of payment. The outcome of the payment process leads to two possibilities: if the payment is successful, the user is directed to a 'Payment Success Page', marking the completion of the transaction. Otherwise, if the payment fails, the user is taken to a 'Payment Failure Page', where they can decide to 'Try Again?' and if it fails, then the user is again redirected to the view cart page.

Fig 4: Activity Diagram from the Seller's scenario

This activity diagram illustrates the process flow for a seller within an e-commerce system The workflow begins with the 'Login' process, where a seller must authenticate their credentials. The 'Authentication' step evaluates the input credentials and proceeds to 'Check' if they are 'Valid' or 'Invalid'. If valid, the user logs in to the platform.

Upon successful login, the user has various task options, such as 'Add Category', where new product categories can be created, or 'Add Item', which allows for new products to be added to the platform. 'Manage Order' provides the user with the capability to oversee customer orders, including viewing, processing, or updating the order status. 'Manage User' is a crucial administrative function that includes activities like activating or deactivating user accounts and editing user roles and permissions.

Furthermore, the seller can perform detailed edits within these tasks. Under 'Add Category', they have the option to 'Modify Details', adjusting the specifics of product categories. Within 'Add Item', they can 'Change Price/Unit' to update pricing information or 'Update Item' to modify product details. 'Manage Order' includes the ability to 'Cancel Order', giving the seller over order management. After completing all the actions, the seller can logout from the system.

3.4 Software Qualities

Correctness: Upon entering the correct credentials, the user should be able to access the platform. Buyer gets the correct product upon completing the transaction. Sellers can add products and manage orders.

Time-efficiency: Time to complete the transaction is less than 10 seconds. Product updates should take less than 5 seconds.

Robustness: The platform should be able to handle the errors or unwanted actions taken from the users and provide friendly information to the user. This applies for platform authentication and authorization as well. Also applicable for the billing and shipping information in product purchase.

4. Top-level and Low-level Software Design

4.1 MVT Architecture and its Benefits

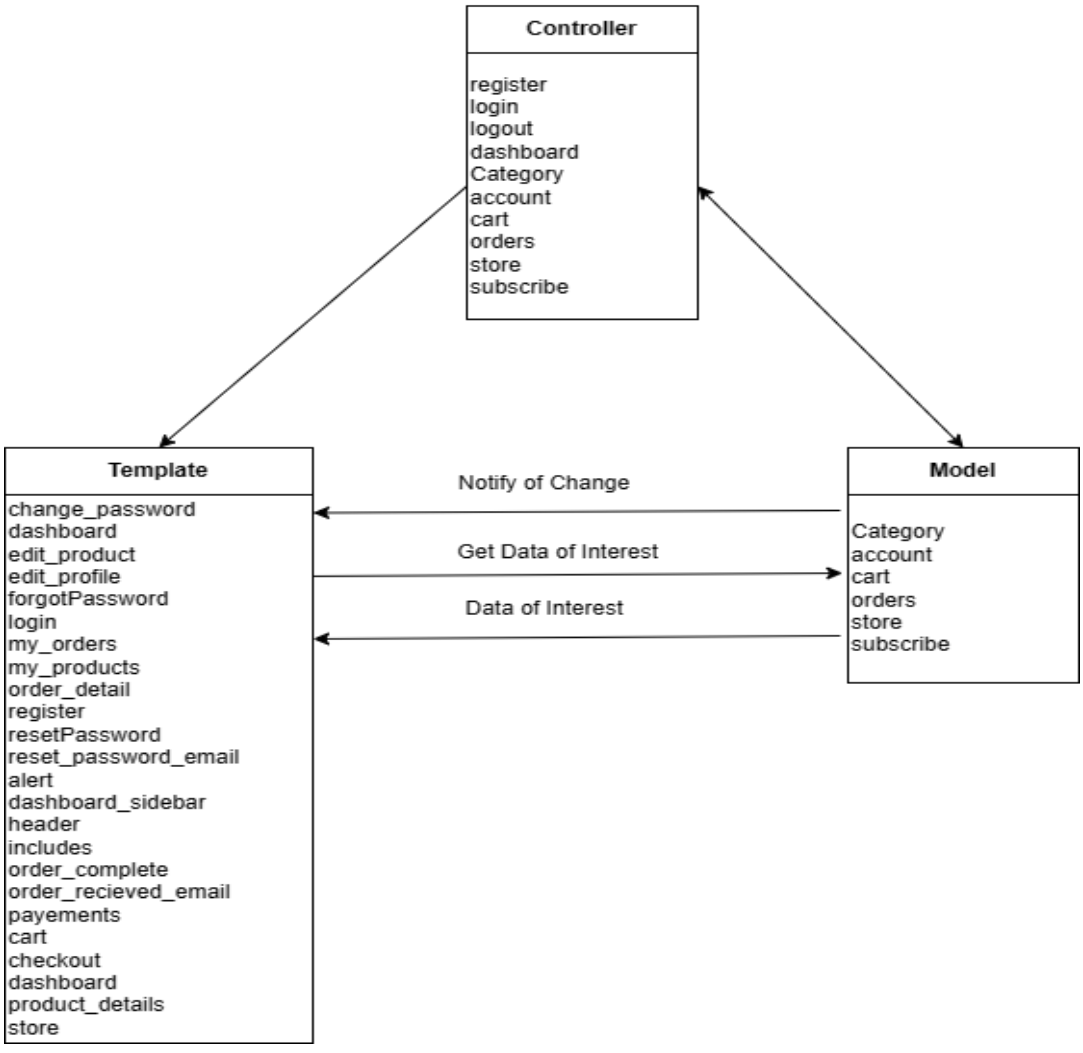

Fig 5: MVT architecture

MVT facilitates a speedier development process by allowing simultaneous development. As it's a group project, MVT provides the functionality where it allows multiple developments at the same time without interfering with each other. For example, one developer can work on the Template while the other works on the Views or Model. Furthermore, MVT allows for the creation of many views, that we need for our website, thus this was a critical argument for utilizing MVT. Also, code duplication is reduced with this. Last but not the least, because the Model component is independent of the Views section, adding a new type of view is simple with the MVT paradigm. Hence, any modifications in the part of the Model will have no change on the complete designs.

Model→ Handles data(data access layer)

Template → Presentation Layer(handles UI)

View → Communicates with the model

In our project, when a user wants to see all products, its details and choose variations, then can request products from ‘Store’ app, then views ask for views.py and access it from the database and with the help of template it views the products page. In between, model.py inside the Store app will help to handle all the logical activities according to the user's search.

4.2 Design Patterns

4.2.1 Observer Design Pattern

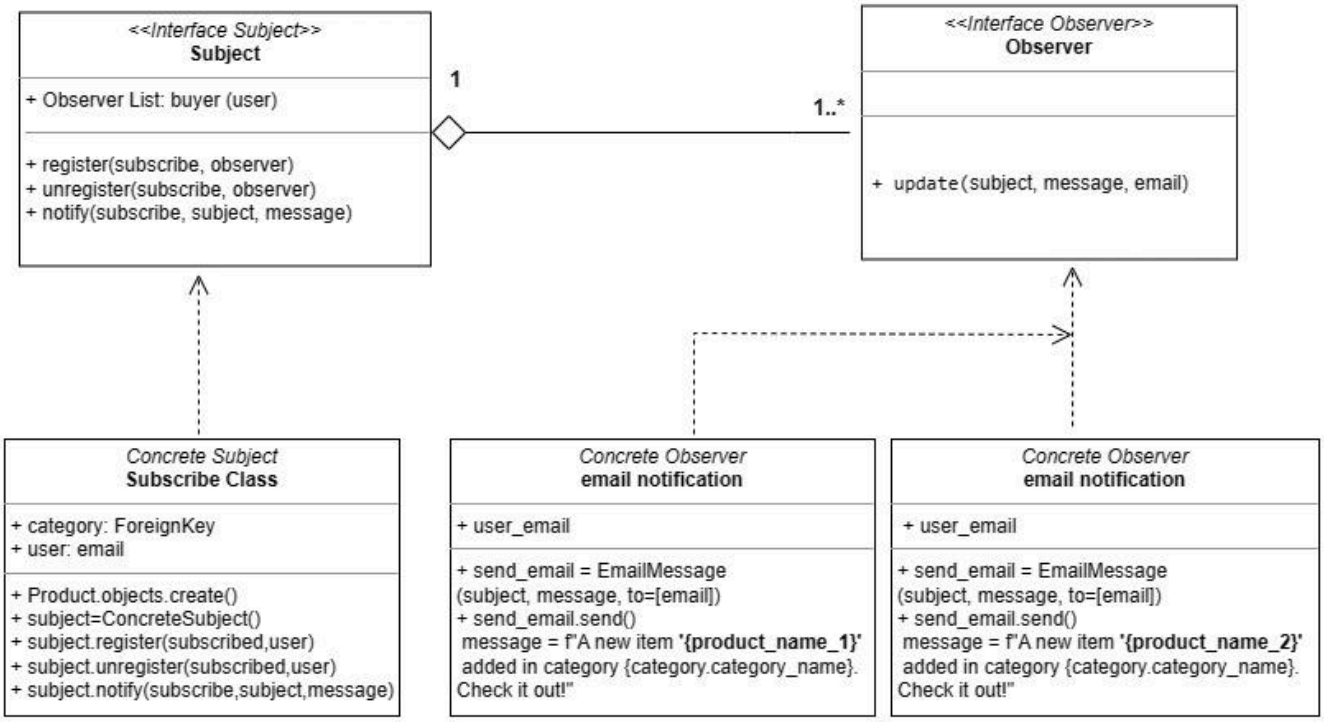

Fig 6: Observer Pattern Class Diagram

Class Subject:

This is our interface subject class

```
def register(self, subscribe,observer):
        pass
  def unregister(self,subscribe, observer):
        pass
  def notify(self,subscribe,subject,message):
        Pass
```

ConcreteSubject class:

This class implements the Subject interface.

The register method takes a SubscribeModel instance and an observer (user) as arguments.

```
def register(self,subscribe, observer):
    print("attached")
    subscribe.subscribers.add(observer)
```

It adds the observer to the subscriber list of the SubscribeModel instance.

The unregister method takes a SubscribeModel instance and an observer (user) as arguments.

```
def unregister(self,subscribe, observer):
    subscribe.subscribers.remove(observer)
```

It removes the observer from the subscriber list of the SubscribeModel instance.

The notify method takes a SubscribeModel instance, subject object, and message as arguments.

```
  def notify(self,subscribe,subject,message):
        for subscriber in subscribe.subscribers.all():
        observer=ConcreteObserver()
        observer.update(subject,message,subscriber.email)
```

It loops through all subscribers of the SubscribeModel instance.
For each subscriber, it creates a ConcreteObserver instance and calls its update method with subject, message, and subscriber's email.

class Observer:

This is our observer Interface Class

```
def update(self,subject,message,email)
pass
```

ConcreteObserver class:

This class implements the Observer interface.

The update method takes subject, message, and email as arguments.

```
def update(self,subject,message,email):
    send_email = EmailMessage(subject, message, to=[email])
```

```
send_email.send()
```

It sends an email notification using the provided email address.

Calling Functions

subscribe function:

```
def subscribe(request,category_id):
  try:
    user=request.user
        if request.user.is_authenticated==False:
        return JsonResponse({'error':"Sign in first"},status=400)
    category=Category.objects.get(id=category_id)
    print(category)
    subscribed, created = SubscribeModel.objects.get_or_create(category=category)
    subject=ConcreteSubject()
    subject.register(subscribed,user)
        return JsonResponse({'message':"Category Subscribed"},status=200)
```

Get the current user from the request.
Check if the user is authenticated.
If not authenticated, return JSON error response to sign in first.
Get the category object based on the category ID.
Use get_or_create method to get or create a SubscribeModel instance for the category.
Create a ConcreteSubject object.
Register the user as an observer for the category using the ConcreteSubject object.

add_product function:

Get the category object based on the category ID.

```
category=Category.objects.get(id=category_id)
```

Create a new product object with user, slugified name, description, image, stock, price, and category information.

Product.objects.create(created_by=user,slug=slug,product_name=product_name,description=product_description,images=product_image,stock=stock_count,price=item_price,category=category)

```
            context= {
               'category':categories,
               'message':"Item successfully added"
               }
```

Check if subscribers exist for the category.
subscribeModel =SubscribeModel.objects.filter(category=category)

If subscribers exist:
Create a ConcreteSubject object.

subject=ConcreteSubject()
subject.notify(subscribe,subject,message)

Notify function will Loop through subscribers and send them an email notification about the new product.

Unsubscribe function:

```
def unsubscribe(request,category_id):
    user=request.user
        if request.user.is_authenticated==False:
        return JsonResponse({'error':"Sign in first"},status=400)

    category=Category.objects.get(id=category_id)
    subscribed = SubscribeModel.objects.get(category=category)
    subject=ConcreteSubject()
    subject.unregister(subscribed,user)
        return JsonResponse({'message':"Category Unsubscribed"},status=200)
```

Get the current user from the request.
Check if the user is authenticated.
If not authenticated, return JSON error response to sign in first.
Get the category object based on the category ID.
Get the SubscribeModel instance for the category.
Create a ConcreteSubject object.
Unregister the user as an observer for the category using the ConcreteSubject object.
Return JSON response with success message.

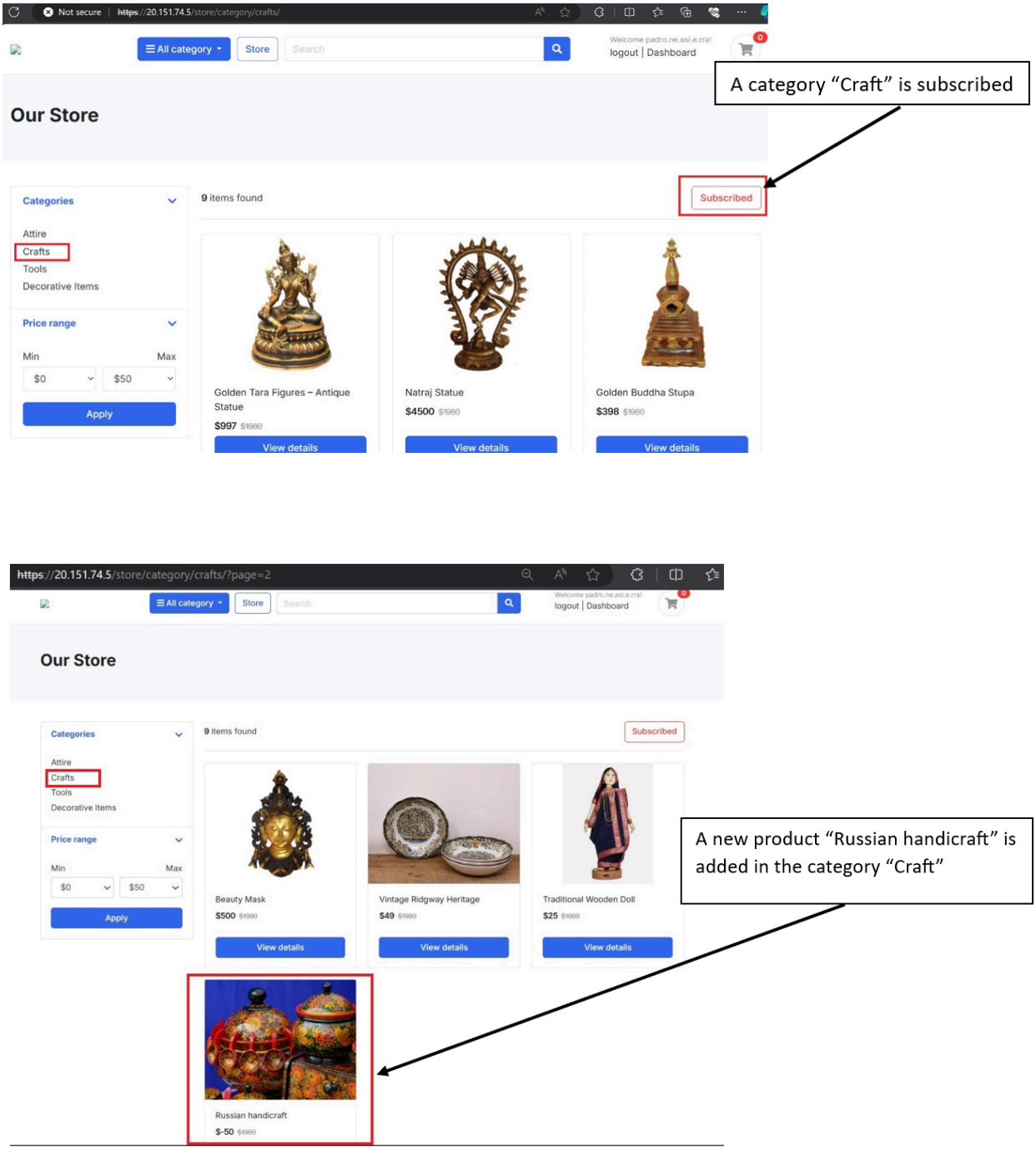

Fig 7: Observer Pattern UI

Users received the below mail after the addition of the new product in the subscribed category.

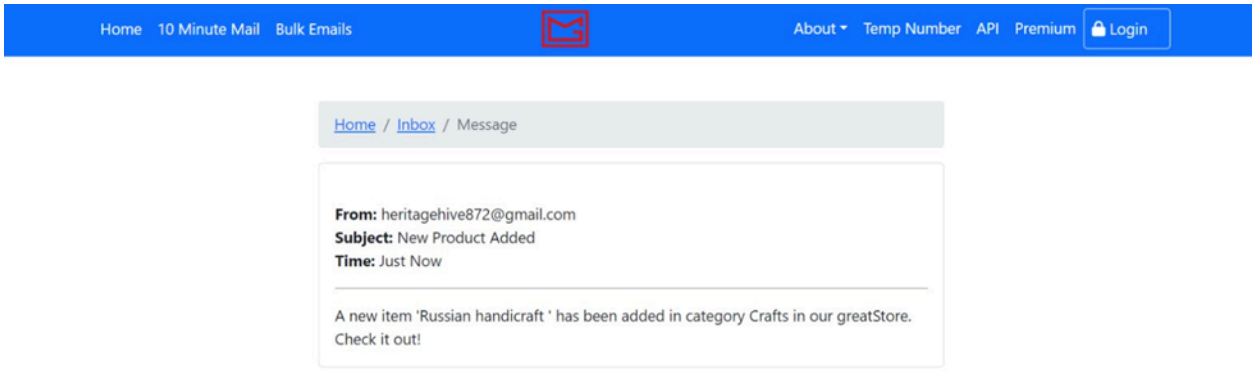

Fig 8: Observer Pattern UI

4.2.2 Simple Factory Design Pattern

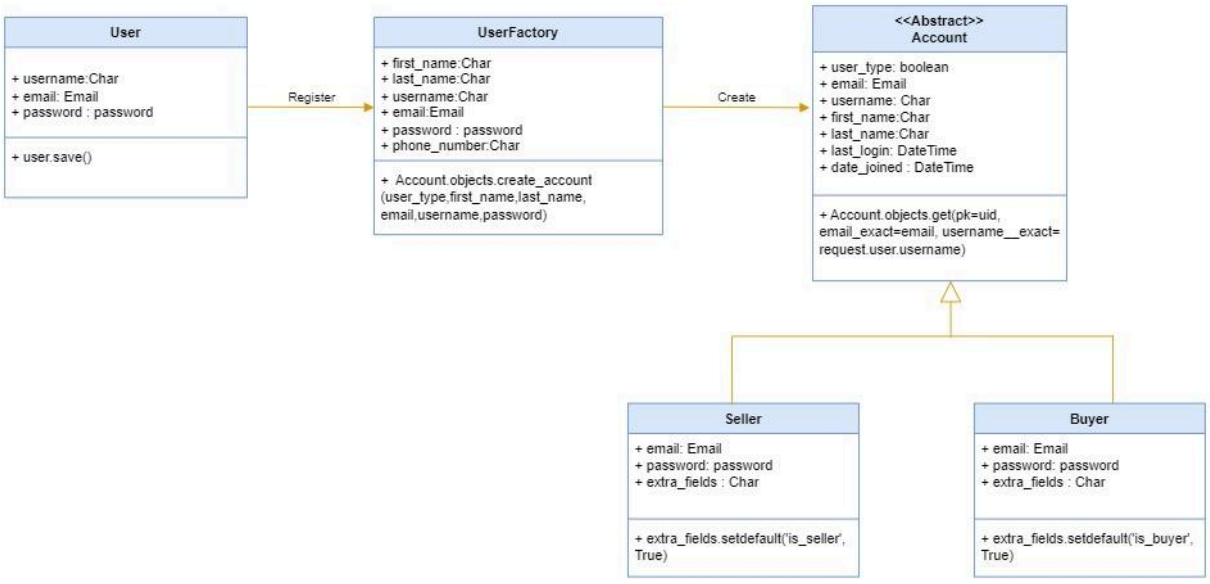

Fig 9: Simple Factory Design Pattern.

We are using the simple factory design pattern in our website ‘HeritageHive’ for the account creation functionality. We have 2 user roles, i) Seller and ii) Buyer. When a user tries to register on our website they have to create an account according to their role. Based on the different roles, they can do different activities. By using the simple factory design pattern in our website, we defined a centralized abstract class 'Account' to create different role-based accounts of the same e-commerce site. Then we encapsulated the accounts’ construction code in a separate class named 'SellerFactory' and 'BuyerFactory'. Further if we need to add more user roles, we can encapsulate the code again and easily build new user role-based functionality. This pattern will ease the maintainability of user accounts by adding new products and removing old products in one place.

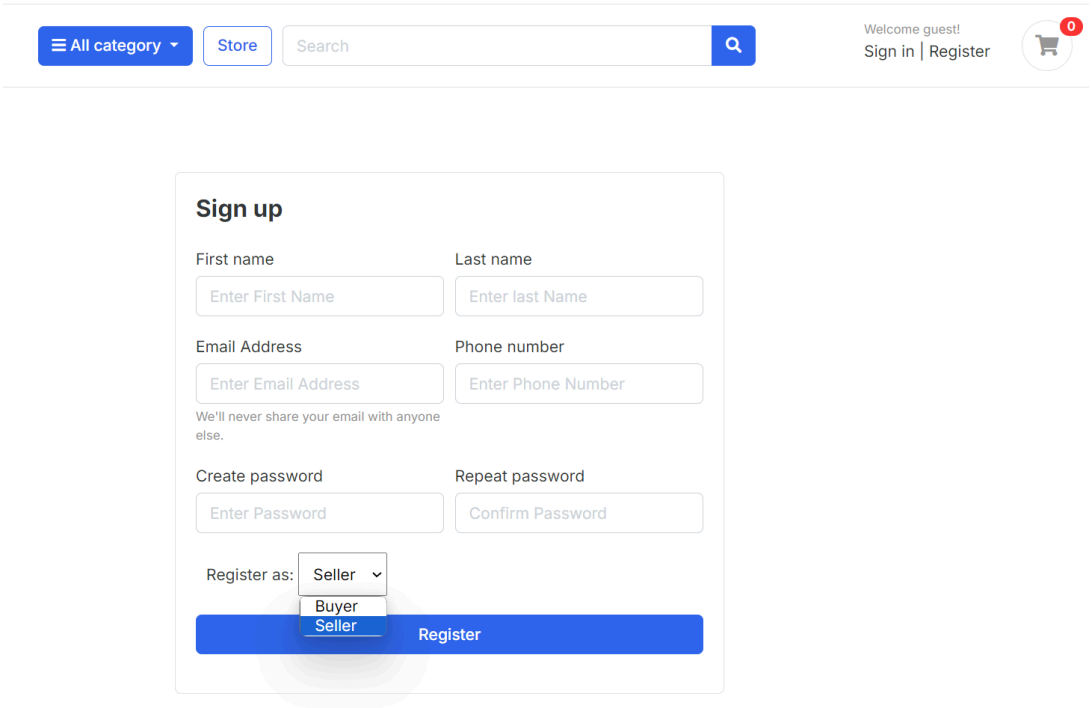

Fig 10: Simple Factory Design Pattern.

After filling up all the required details in this ‘Sign up’ page, users have to select the role in the drop-down box of ‘Register as’. This date will be added to the database as a boolean data. By

pressing the Register button the user can complete this task and can login to our website after authentication.

These are the data types of the attributes:

User:

- username:Char
- email: Email
- password : password

Account:

- user_type: boolean
- email: Email
- username: Char
- first_name:Char
- last_name:Char
- last_login: DateTime
- date_joined : DateTime

The prototypes of the methods:

Account.objects.create_account()

Account.objects.get()

extra_fields.setdefault()

user.save()

This is the algorithm that we used:

```
# create the Abstract class and static method for extending the different roles as seller and buyer
class UserFactory(BaseUserManager):
    @staticmethod
    def create_user(email, password, **extra_fields):
        if not email:
            raise ValueError('The Email field must be set')
        email = email
        user = Account(email=email, **extra_fields)
        user.set_password(password)
        user.save()
        return user

    @classmethod
    def create_account(self, user_type='buyer', **kwargs):
        if user_type == 'buyer':
            buyer = BuyerFactory.create_buyer(**kwargs)
            return buyer
        if user_type=="seller":
            seller = SellerFactory.create_seller(**kwargs)
            return seller
        else:
```

```
            raise ValueError("Invalid user type")


class BuyerFactory(UserFactory):
    @staticmethod
    def create_buyer(email, password, **extra_fields):
            return UserFactory.create_user(email, password, **extra_fields)


class SellerFactory(UserFactory):
    @staticmethod
    def create_seller(email, password, **extra_fields):
        extra_fields.setdefault('is_seller', True)
        return UserFactory.create_user(email, password, **extra_fields)


# get the details for authentication  from the account objects
Account.objects.get(pk=uid, email_exact=email, username__exact=
request.user.username)


# create account object and save it
Account.objects.create_account(user_type=user_type,first_name=first_name,
last_name=last_name, email=email, username=username, password=password)
            user.save()
```

4.3 AI Component

Semantic Search Feature:

In this section, we talk about the AI feature as a functional requirement from the buyer's perspective. The integration of semantic search functionality within the Heritage Hive e-commerce project leverages the capabilities of vector databases. This approach seeks to enhance product search functions for buyers by adding an understanding of the inherent semantic meaning embedded within their search queries and recommending relevant products even if they don't use exact keyword matches. By going beyond exact keyword matches, the feature recommends relevant products, ensuring a more intuitive and user-centric shopping experience.

Semantic Search Implementation Methodology:

Library Installation: We used several effective libraries in our project to expedite our data processing and search functions. Pandas was necessary for effective data processing, making it simple for us to input and prepare our product data. We used the qdrant_client package to connect with Qdrant, our vector database option. This allowed us to create collections, upload data, and do vector similarity searches. In order to do advanced searches based on textual similarity, the sentence_transformers library played a crucial role in providing pre-trained models for transforming product text into vector embeddings. Finally, pickle was used in serializing and deserializing Python objects. Particularly, it helped us save our created product vectors for future use, which made our data pipeline efficient and reliable.

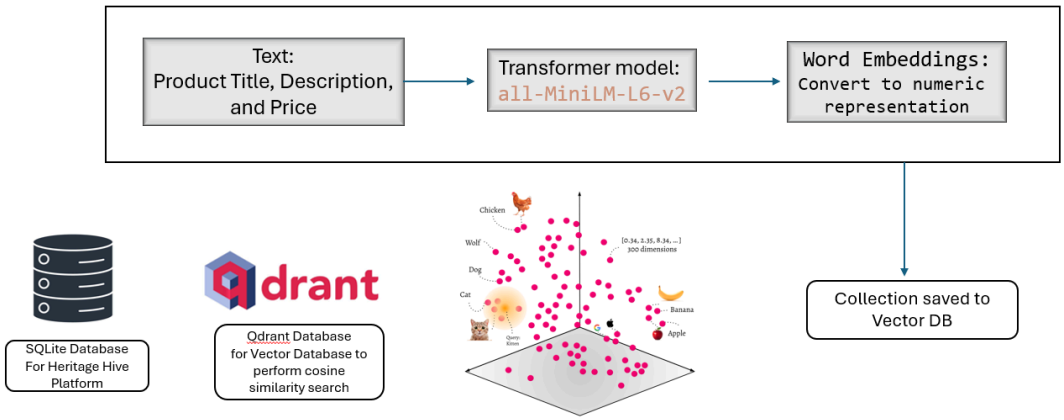

Fig 11:Semantic Search Feature Methodology

Data Acquisition and Preparation: Product data was obtained from our project's database, including attributes like titles, descriptions, and specifications. This data was cleaned and preprocessed to ensure optimal performance within the vector search model.

Text Vectorization: Product text data (such as titles) was transformed into vector embeddings using a pre-trained sentence transformer model (all-MiniLM-L6-v2). The embeddings provided effective similarity comparisons by numerically expressing the semantic meaning of the text data.

Vector Database Integration: To store the generated product embeddings, a collection was created inside the Qdrant vector database. Subsequent semantic search procedures are built upon this collection.

Implementing Semantic Search: Using the same sentence transformer model (all-MiniLM-L6-v2), the buyer's query was converted into a vector embedding during a search. Next, a cosine similarity search was run through the vector database's product embeddings.

Semantically relevant products were found by this search, with embeddings that were closest to the user's query embedding.

The products identified through the semantic search were displayed to the buyers as an output to their query. This approach provides a more comprehensive product search experience compared to traditional keyword-based search.

views.py | services.py | utils.py

store > services.py > load_vectors

```python
import pandas as pd
from qdrant_client import QdrantClient
from qdrant_client.models import models
from sentence_transformers import SentenceTransformer
import pickle

# loading and preparing

def load_data(data):
    return pd.read_csv(data)

def prepare_data(df):
    docx = df['product_name'].tolist()
    payload = df[['id','product_name', 'images','stock','price','is_available','slug', 'description']].to_dict('recor
    return docx, payload

def save_vectors(vectors):
    with open('vectorized_courses.pickle', 'wb') as f:
        pickle.dump(vectors, f)

def load_vectors(vector_file):
    with open(vector_file, 'rb') as f:
        myobject = pickle.load(f)
    return myobject

client = QdrantClient(path='vector_database.db')
client.recreate_collection(collection_name='product_collection',
                           vectors_config=models.VectorParams(
                               size=384, distance=models.Distance.COSINE
                           ))

```

Figure 12: Data Loading and Preprocessing Workflow

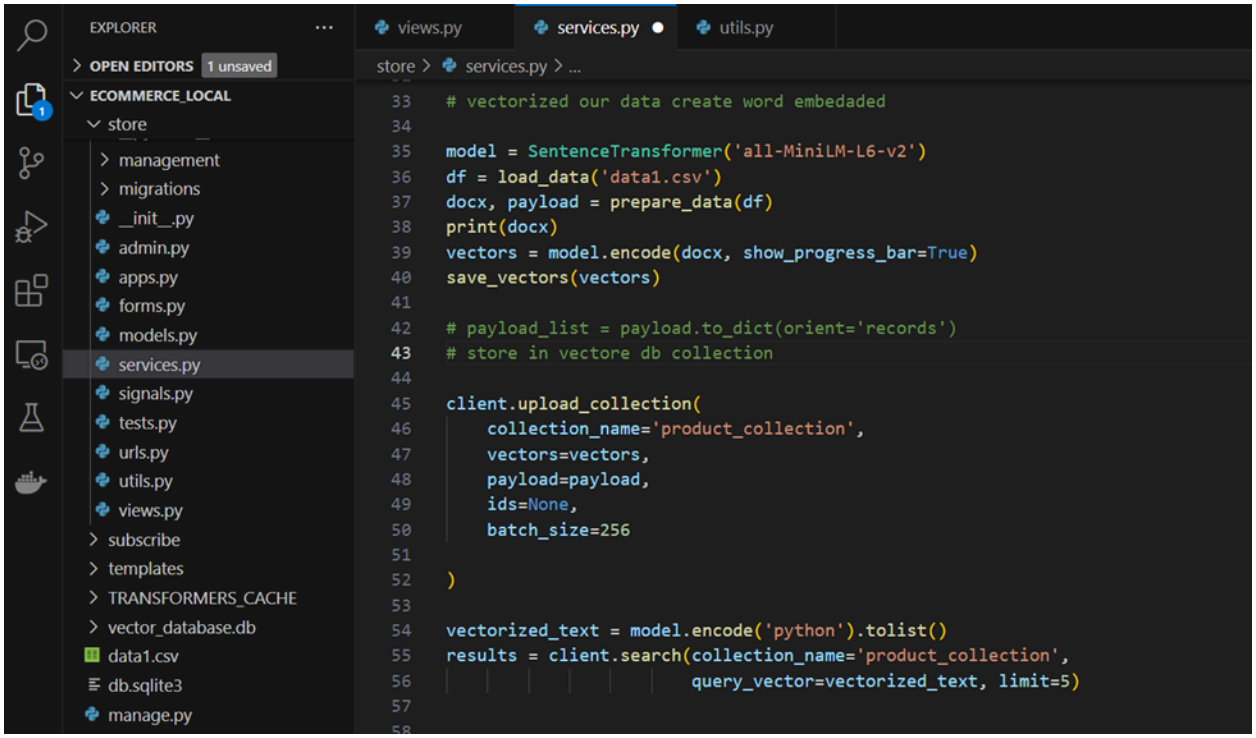

Figure 13: Product Data Preparation for Vectorization

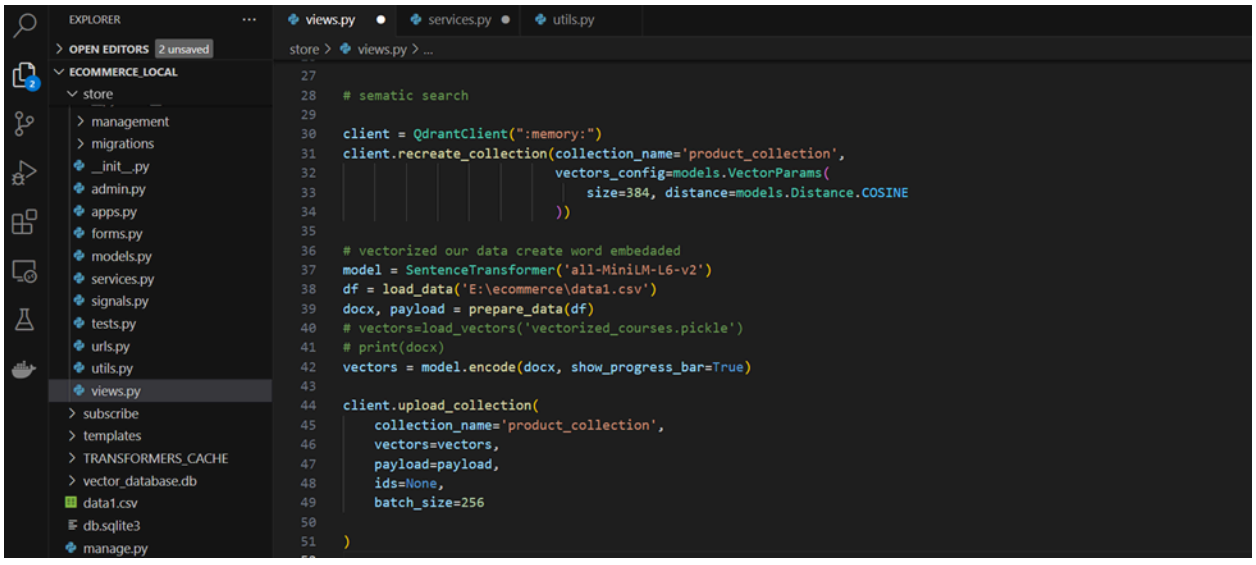

Figure 14: Building model for the product data in the view file

Figure 15: Product Search Results Filtering and Scoring Threshold Configuration Setup

By adding the semantic search feature within Heritage Hive, the project delivers a great product retrieval experience for its buyers. This approach leverages the strength of vector databases to understand user search intent beyond literal keywords, leading to more relevant product recommendations and enhanced customer satisfaction.

5. Software Construction

5.1 Screenshot of entire code structure

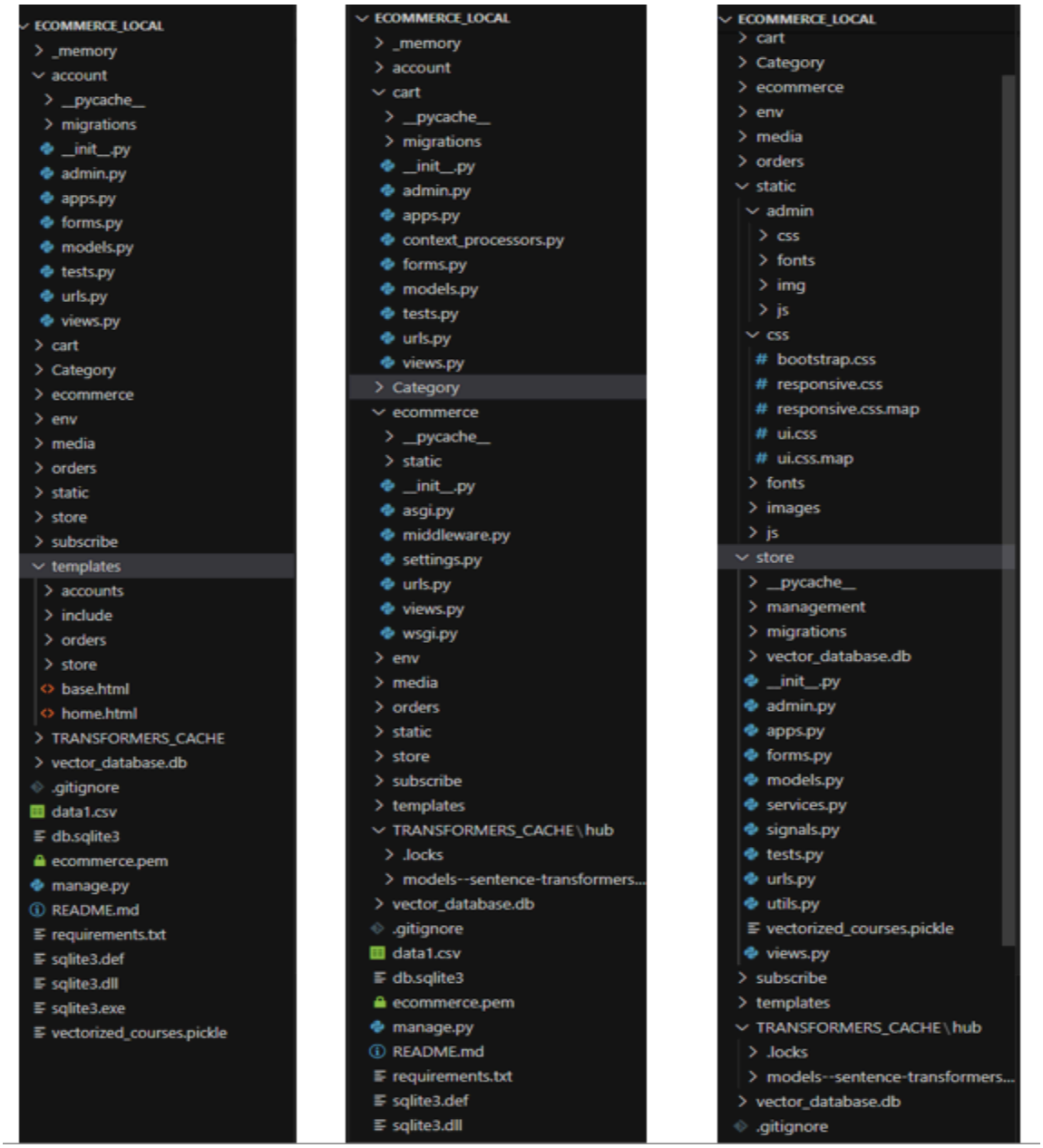

Figure 16: Directory Structure

5.2 Deployment diagram

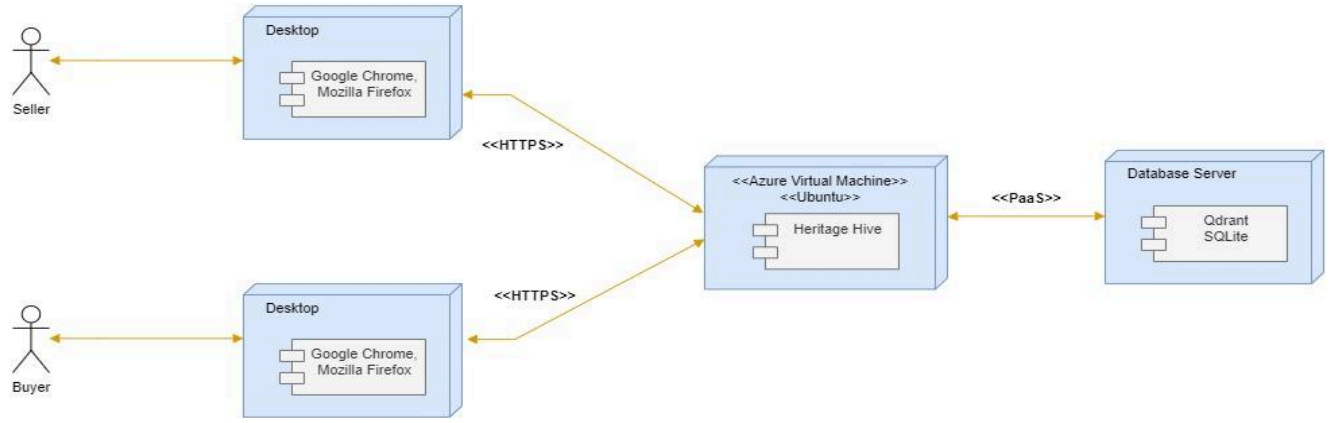

Fig 17: Deployment Diagram.

In our project, initially both buyer and seller can access the website using any desktop and likely running web browsers like Google Chrome or Mozilla Firefox. These desktops symbolize the user interface where users interact with the website.

After that, users connect to the web server using HTTPS, likely over the internet. This secure communication protocol encrypts data transmission between clients and the server. The web server is running on the Ubuntu software system in the virtual machine hosted on Microsoft Azure's cloud platform. By the use of PaaS - Platform as a Service, this web server delivers web pages and other content to client web browsers. That's how our project 'Heritage Hive' is deployed in the cloud. Regarding the database server, we used Qdrant for semantic search data and SQLite for the rest of the features. This server stores product information, user data, and potentially other e-commerce related details.

5.3 Screenshots of all table contents of the system data

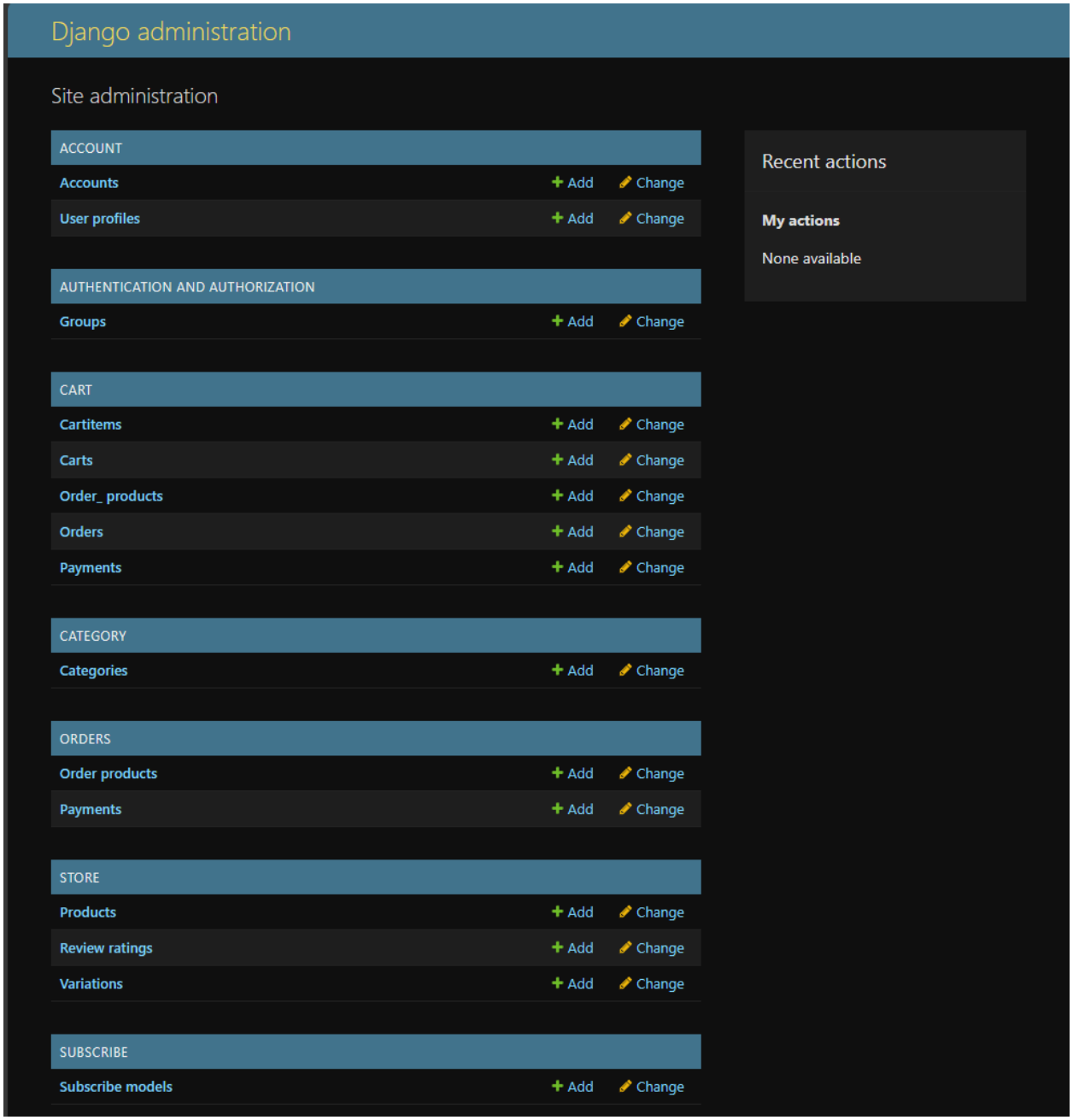

Figure 18: Data Tables

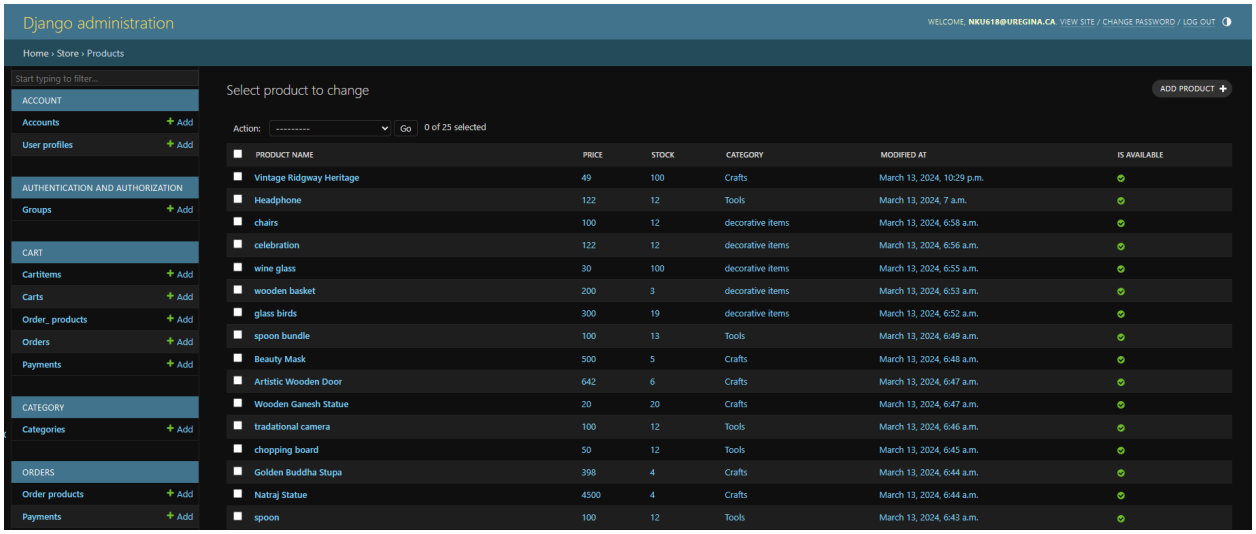

Figure 19: Products Table

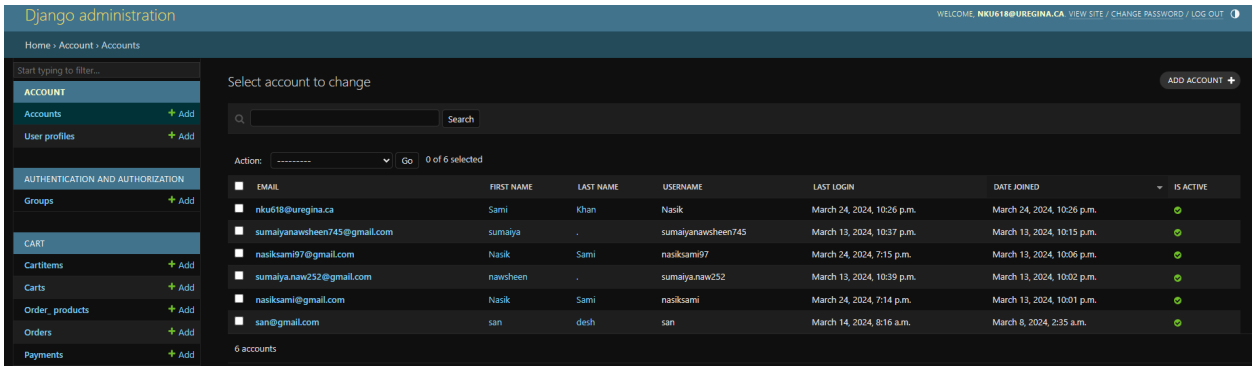

Figure 20: User Table

5.4 GitHub link to entire program

https://github.com/nasiksami/Heritage_Hive

5.5 Link of Web-based application

https://20.151.74.5/

6. Technical Documentation

6.1 List of programming languages

- Django Framework
- Python
- HTML
- CSS
- JavaScript
- AJAX
- Qdrant - Vector Database

6.2 List of reused algorithms and programs

Referred to the following for front-end design:

• Bootstrap: https://getbootstrap.com/docs/5.3/getting-started/introduction/

• MDBootstrap: https://mdbootstrap.com/docs/standard/extended/login/

Referred to the following git repositories:

- https://github.com/dancergraham/HeadFirstDesignPatterns_python/tree/main/chapter02_observer
- https://github.com/rajan2275/Python-Design-Patterns/blob/master/Behavioral/observer_example.py

6.3 List of software tools and environments

• Visual Studio Code: This was a very familiar IDE to work with. Not only did we use it in our prior courses, it is also user-friendly and it's features for code editing and

debugging proved to be beneficial for our project. It is really useful to work with MVT pattern in Django.

• GitHub: Making a repository where everyone can be up to date with their work

ensured good collaboration between the front end and the back end. We have created a total of 4 branches, so that we can collaborate with each other and resolve the code conflicts in a very systematic way.

7. Acceptance Testing

7.1 Correctness testing

Test Case 1: Seller trying to login.

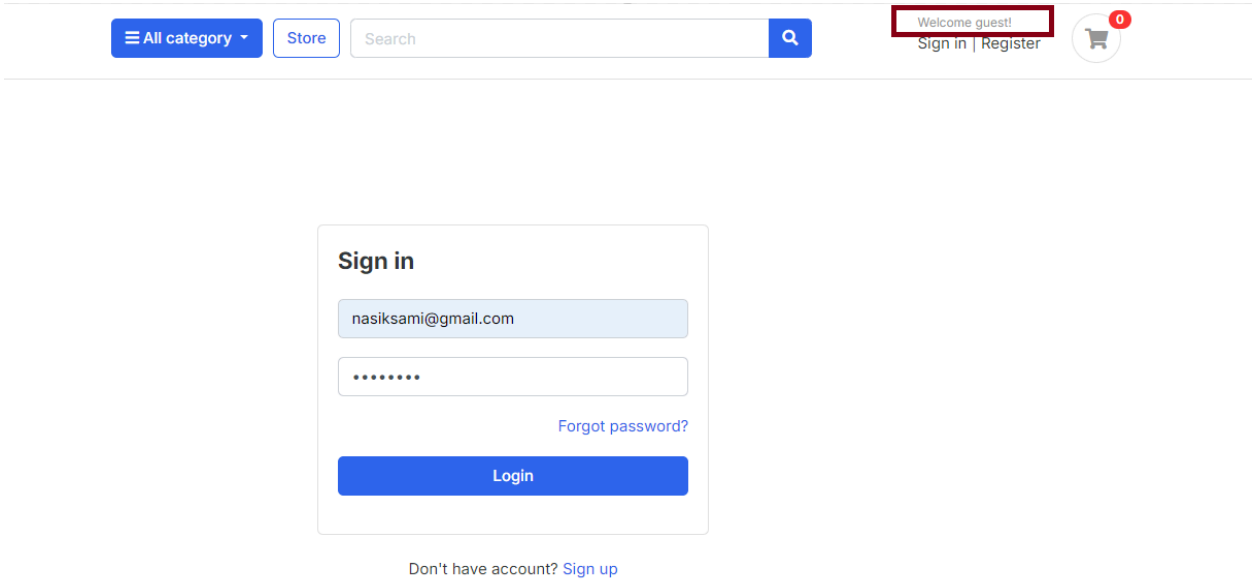

Figure 21: Test Case 1 – Input

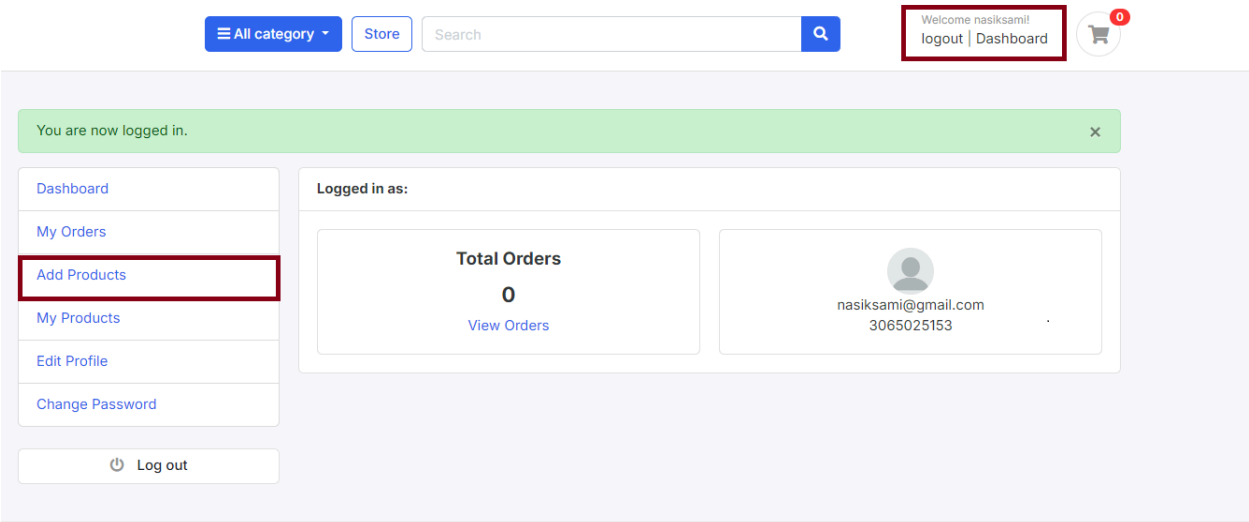

Figure 22: Test Case 1 – Output

Test Case 2: Seller trying to add product.

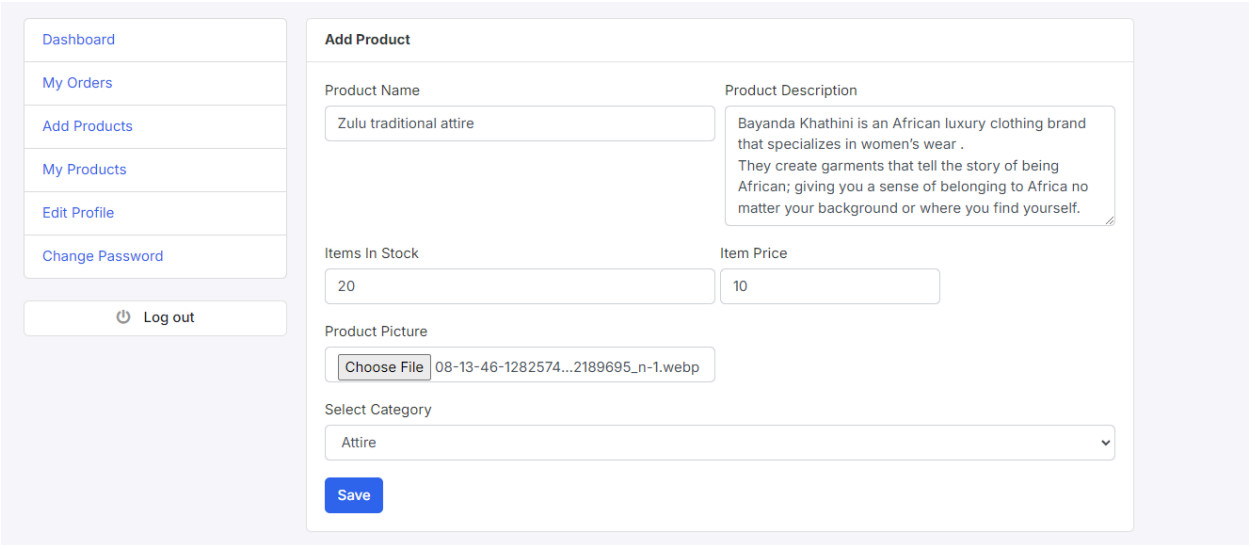

Figure 23: Test Case 2 – Input

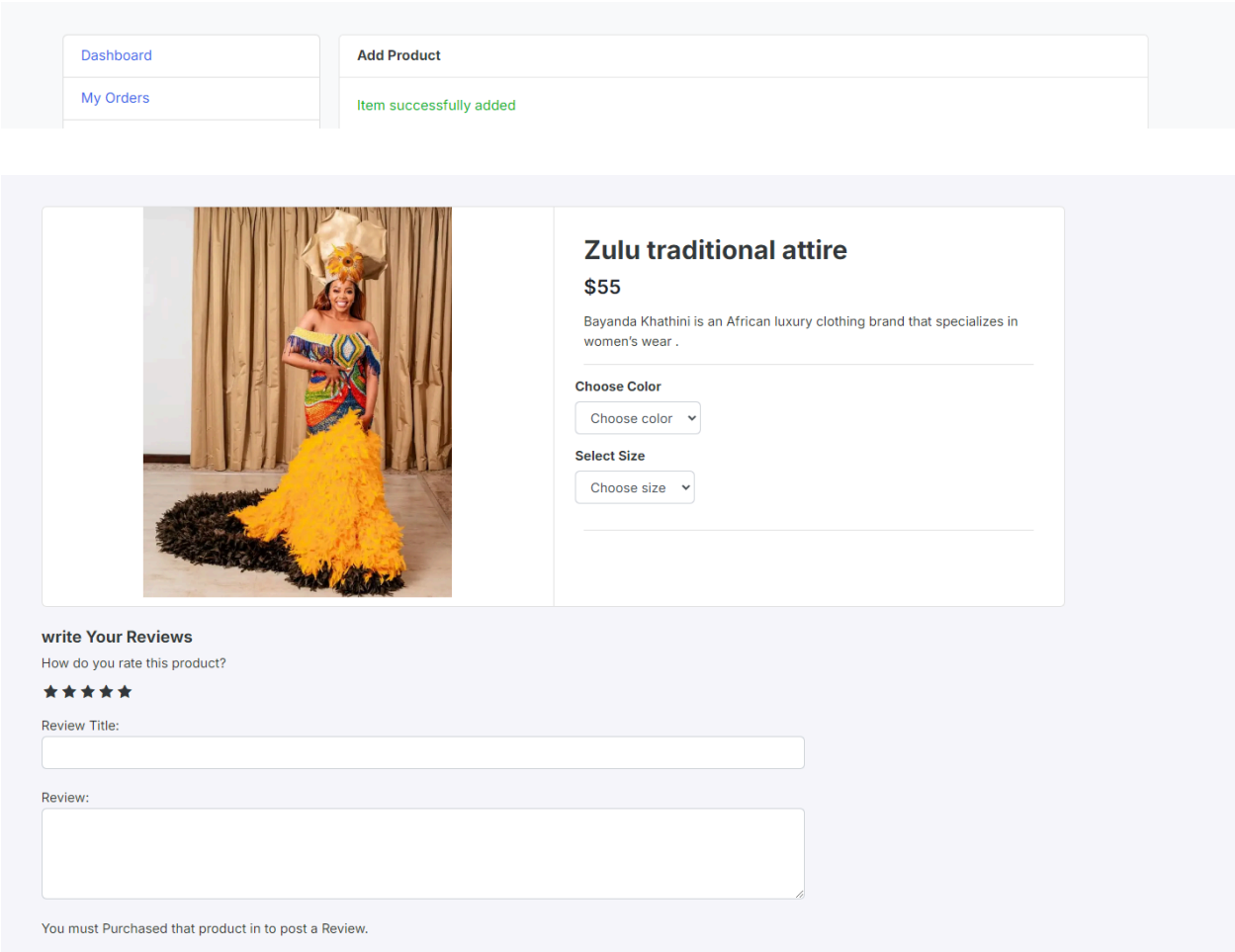

Figure 24: Test Case 2 – Output

Test Case 3: Buyer trying to product

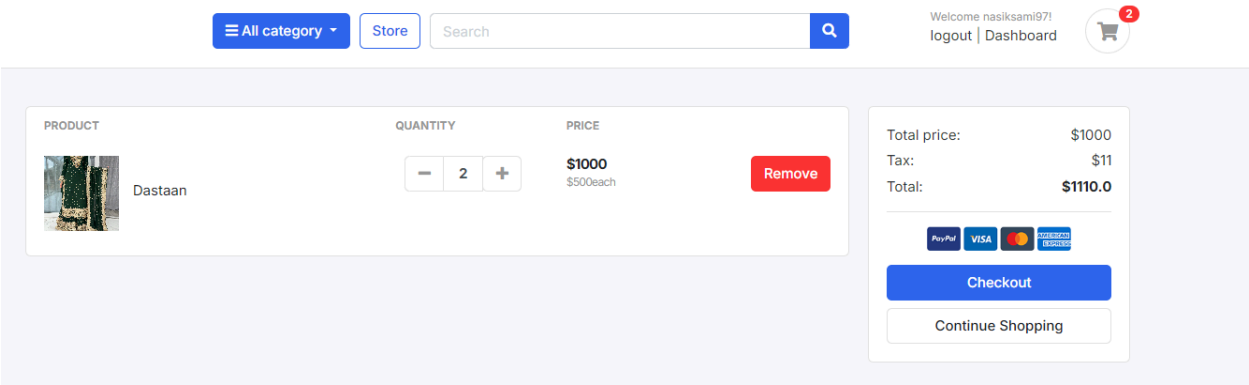

Figure 25: Test Case 3 – Input

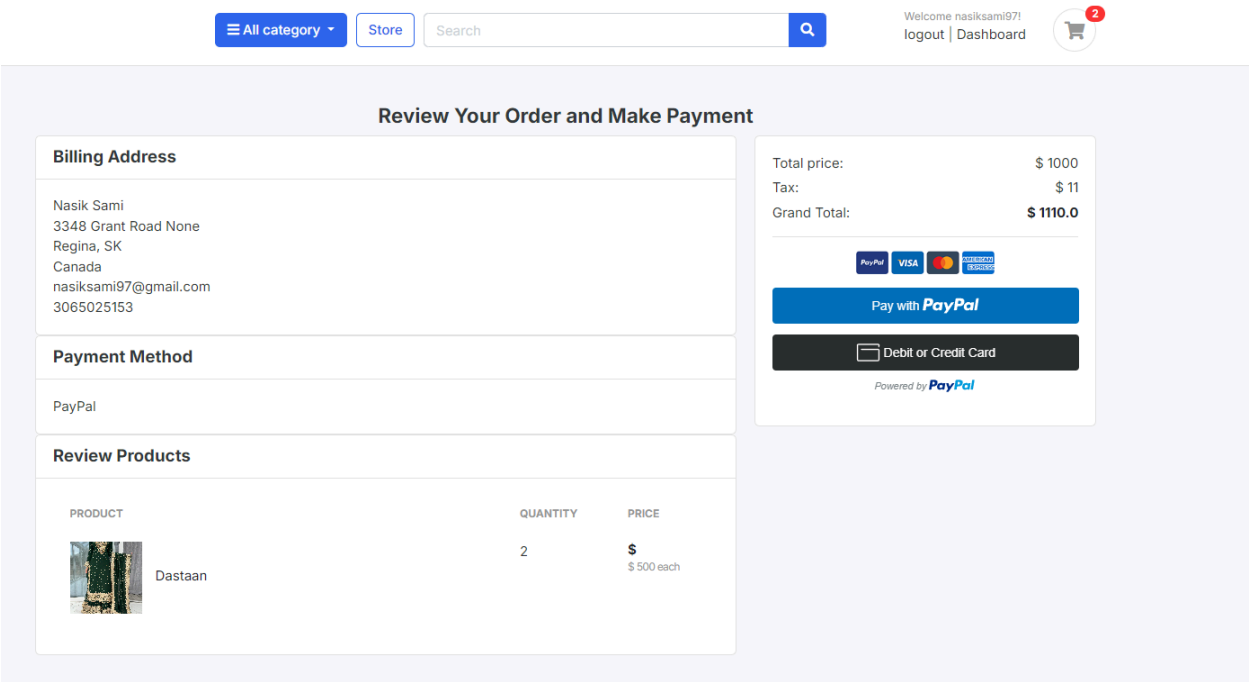

Figure 26: Test Case 3 – Input

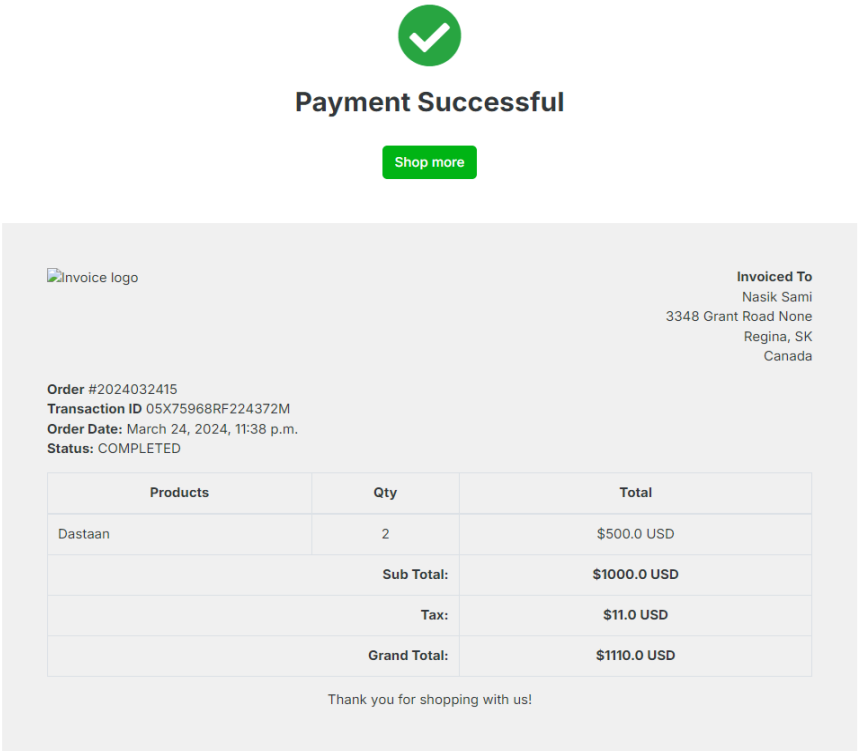

Figure 27: Test Case 3 – Output

Test Case 4: Buyer trying to filter out product

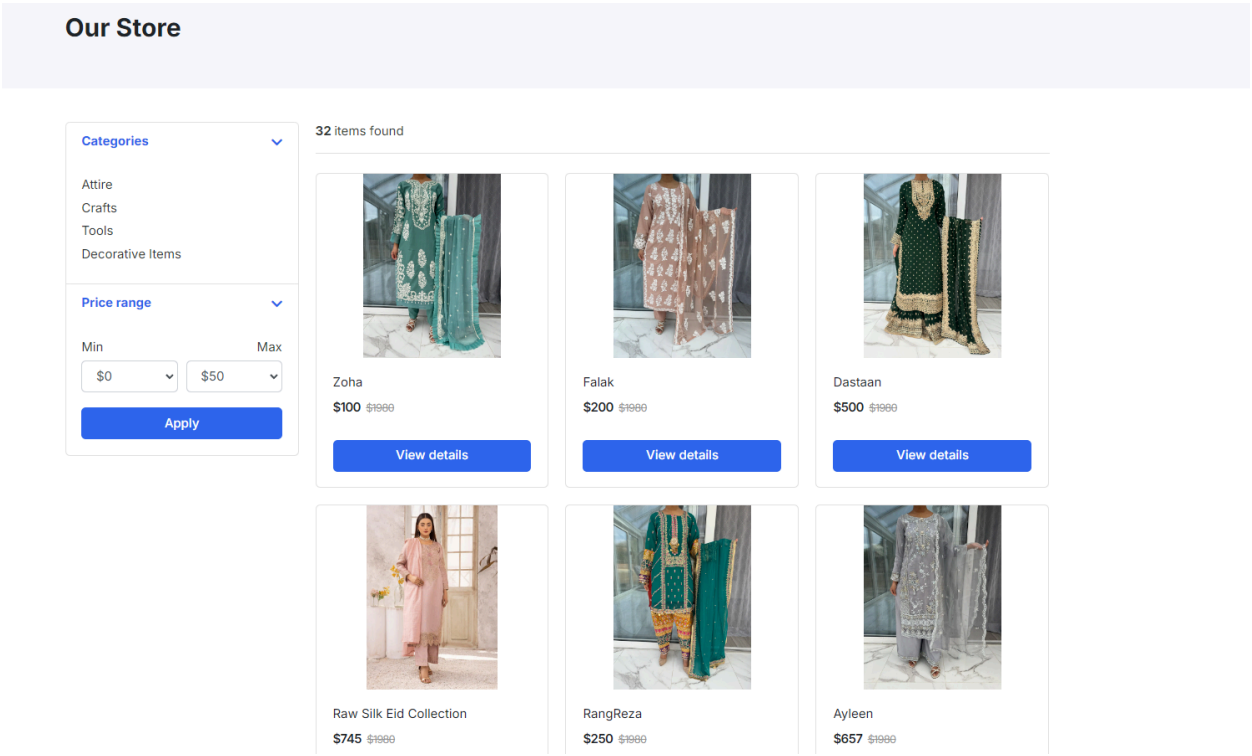

Figure 28: Test Case 4 – Input

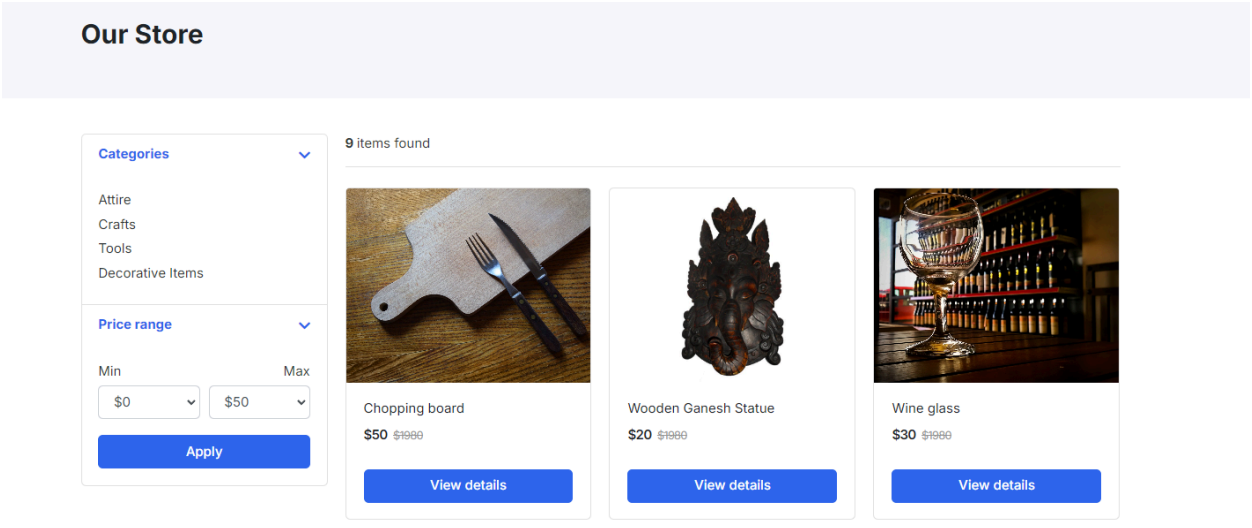

Figure 29: Test Case 4 – Output

7.2 Robustness testing

Test Case 1: Buyer trying to add negative number in the product cart

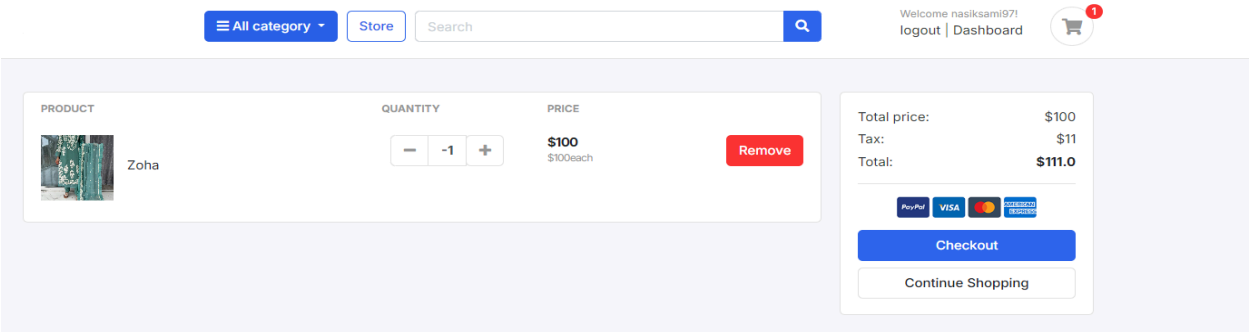

Figure 30: Test Case 1 – Input

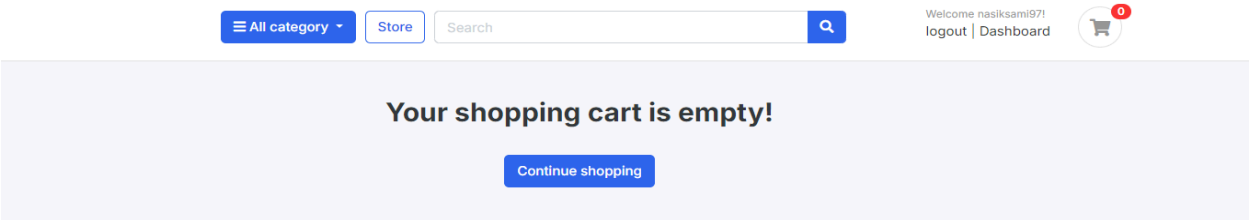

Figure 31: Test Case 1– Output

Test Case 2: Seller trying to enter wrong password

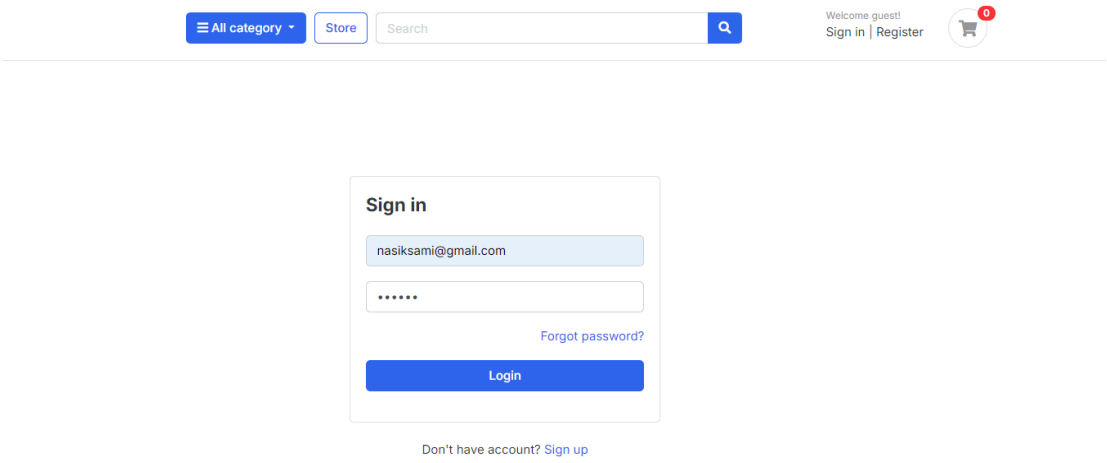

Figure 32: Test Case 2 – Input

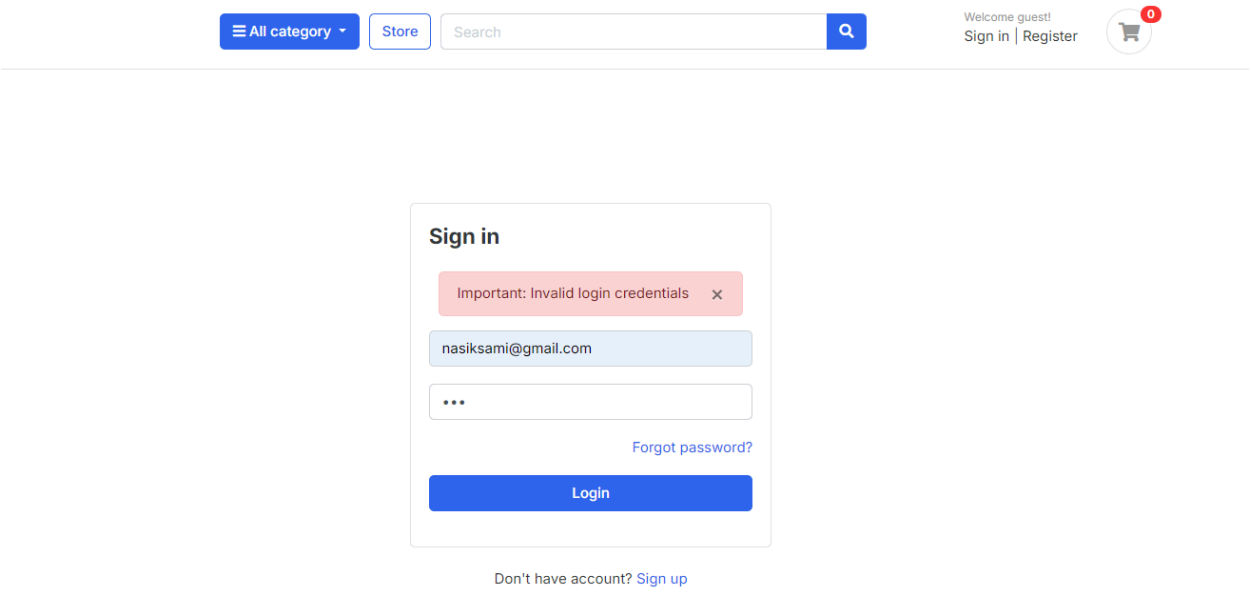

Figure 33: Test Case 2 – Output

Test Case 3: Buyer trying to add product with incomplete information

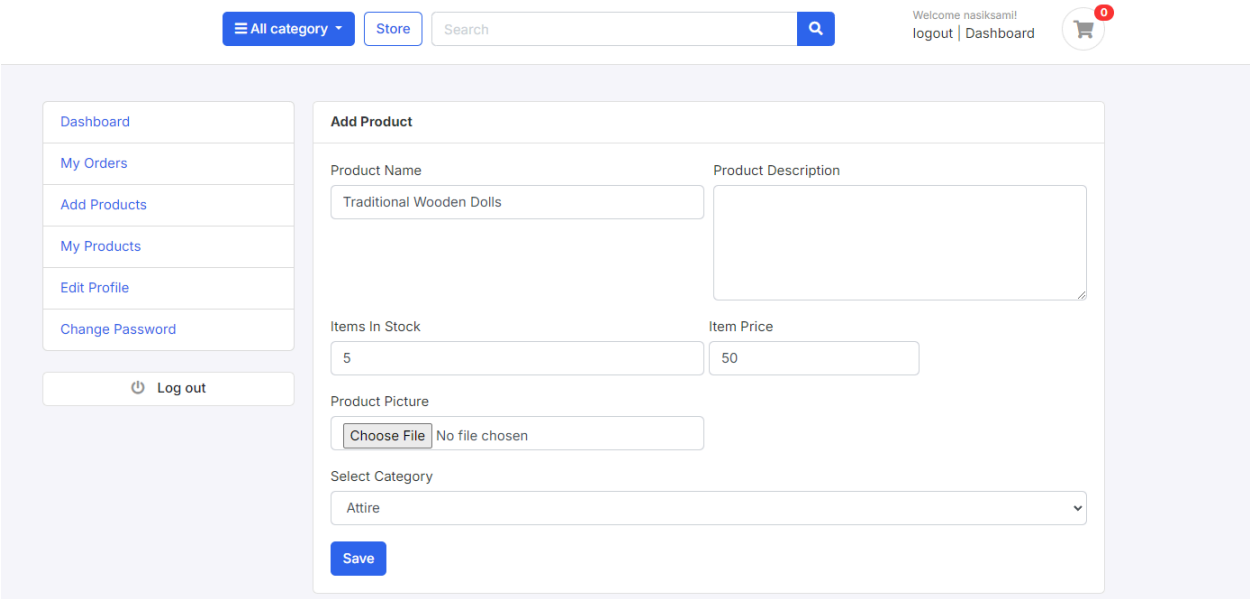

Figure 34: Test Case 3 – Input

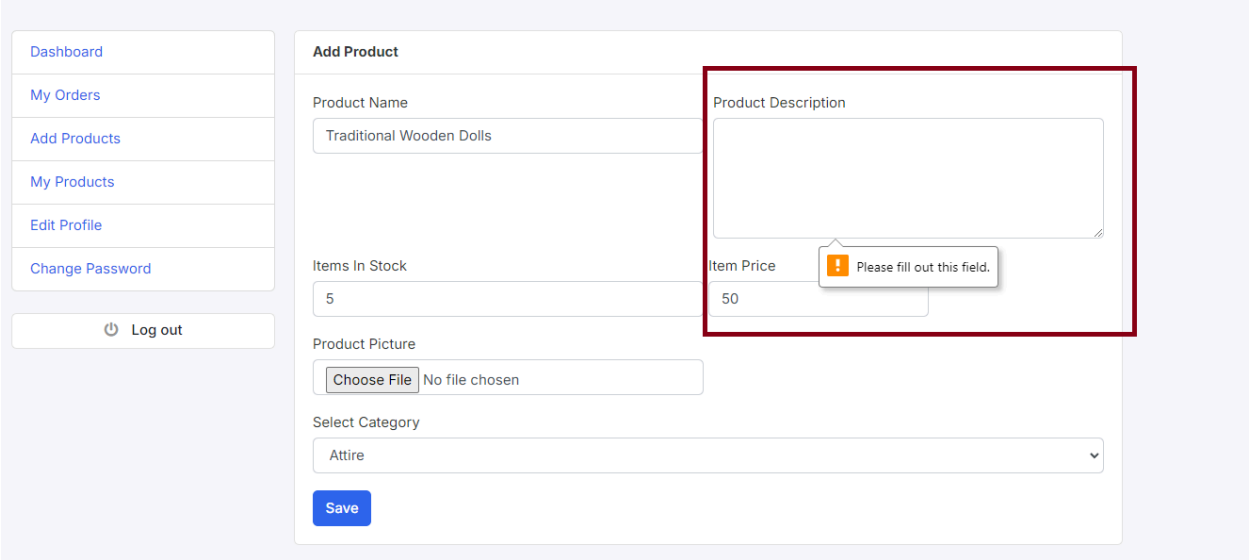

Figure 35: Test Case 3 – Output

Test Case 4: Buyer trying to perform semantic search with relevant keyword

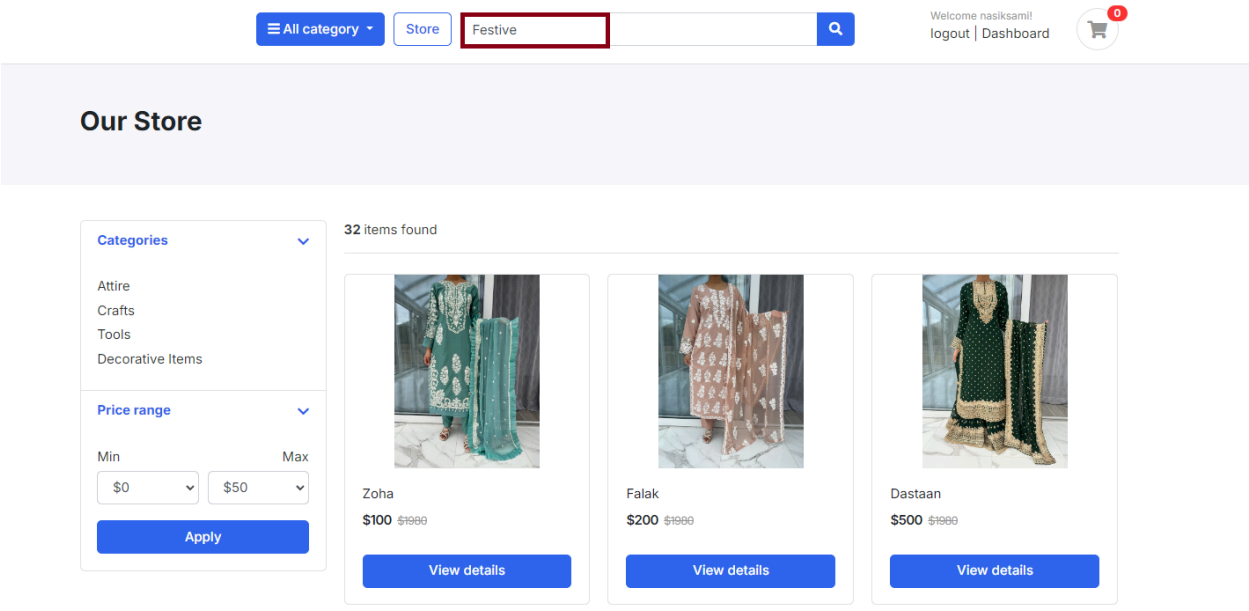

Figure 36: Test Case 4 – Input

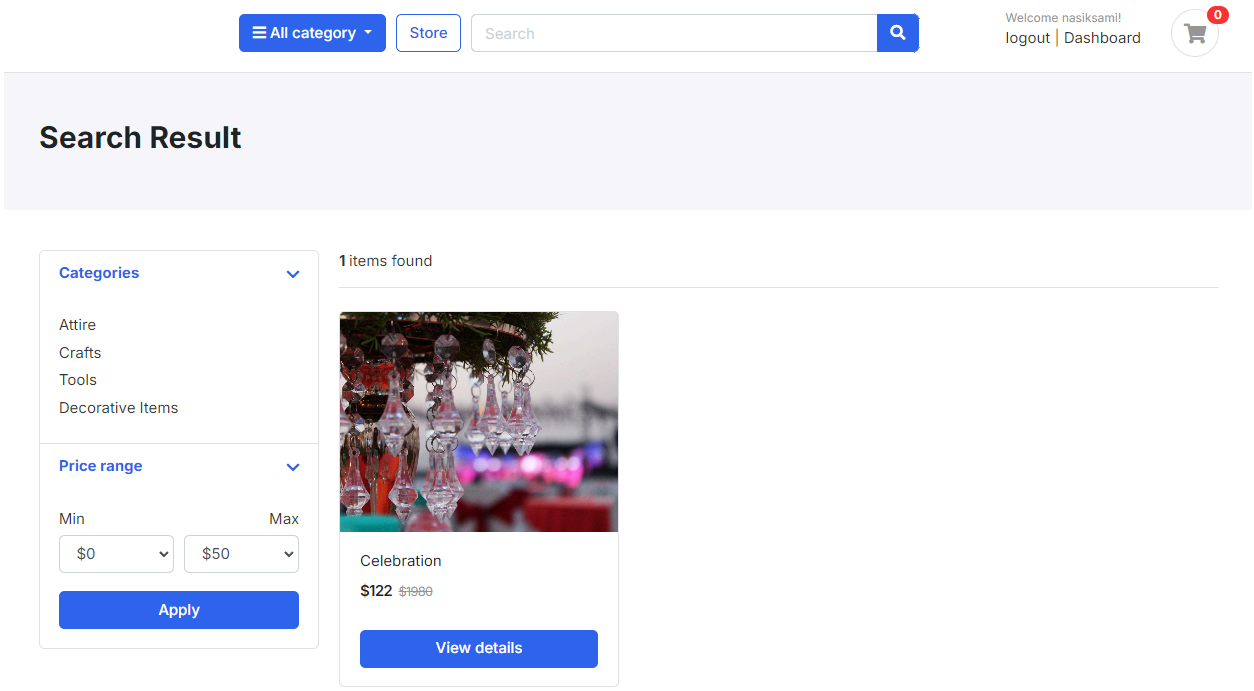

Figure 37: Test Case 4 – Output

7.3 Time-efficiency testing

Test Case 1: Time taken for seller to add a product to the store.

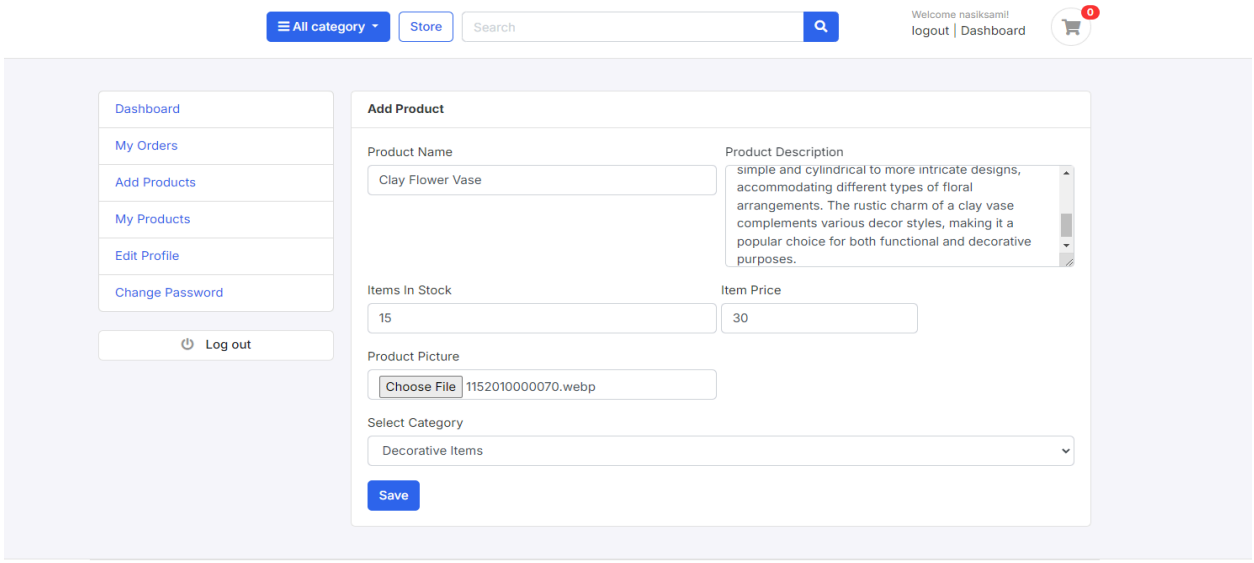

Figure 38: Test Case 1 – Input

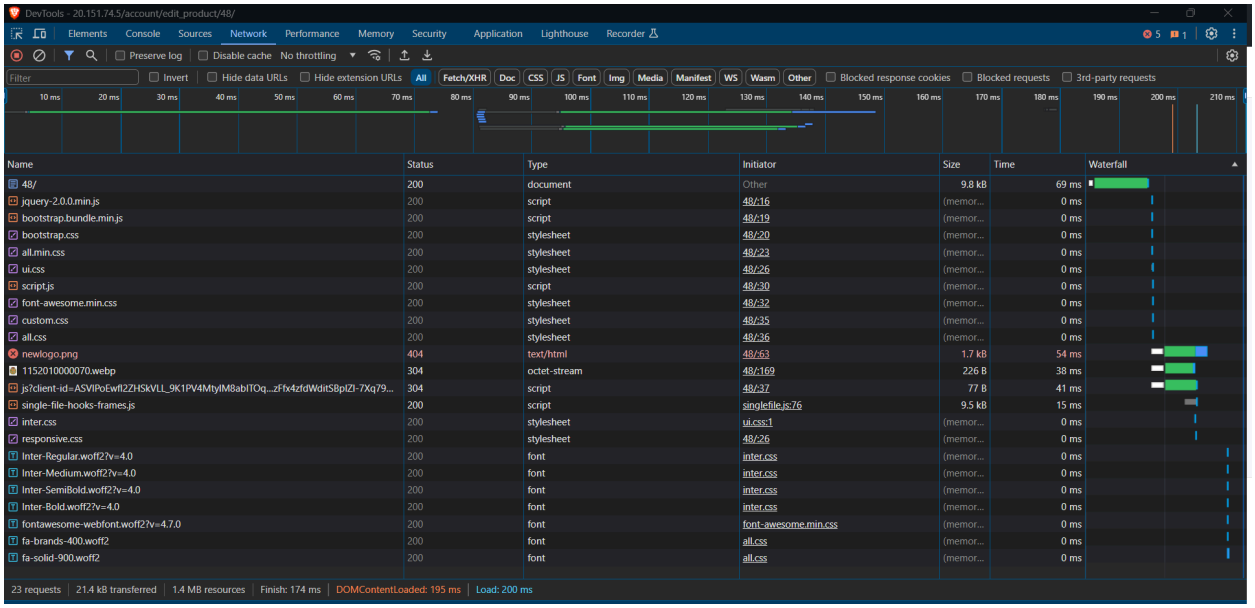

Figure 39: Test Case 1 – Output

Test Case 2: Time taken for the system to respond when a student does not fill up the form properly.

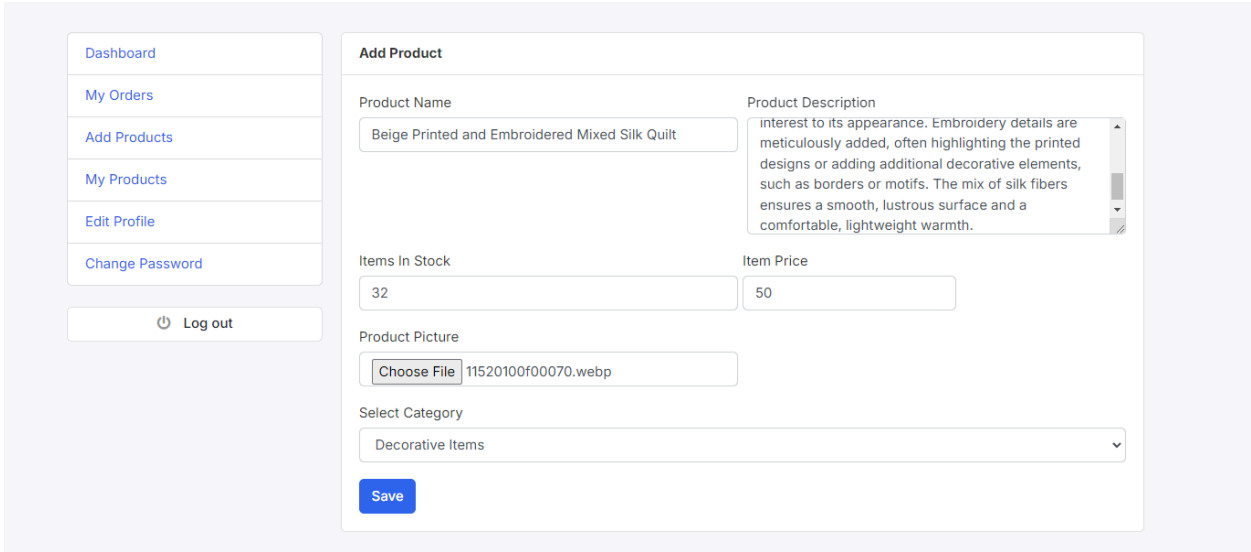

Figure 40: Test Case 2 – Input

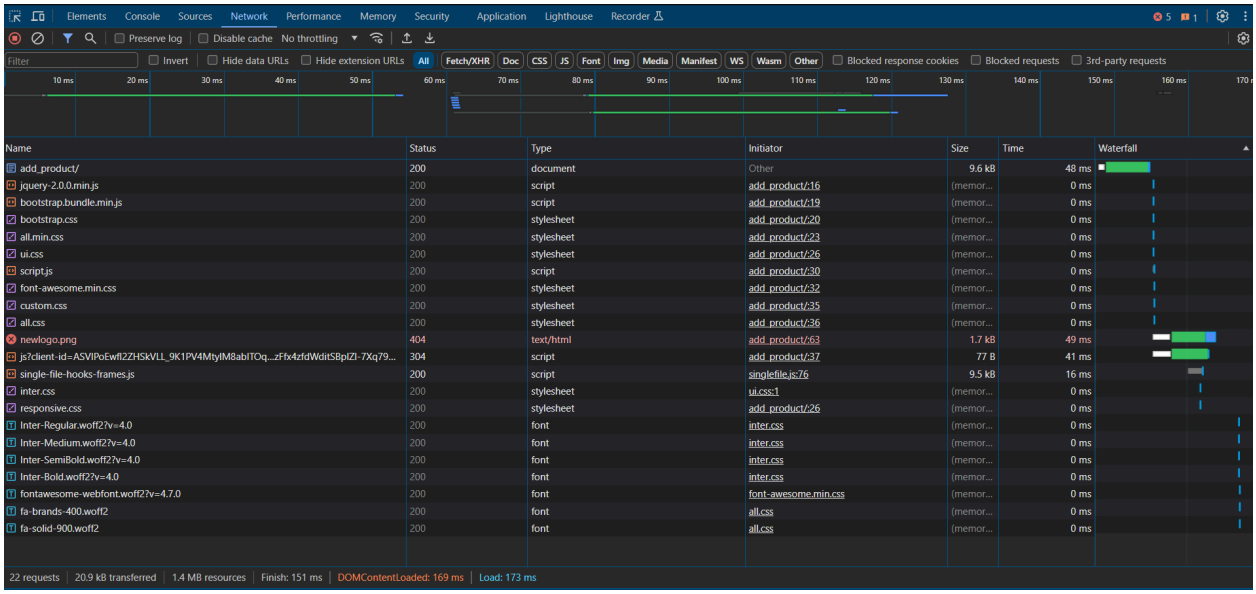

Figure 41: Test Case 2 – Output

8. Summary

The Heritage Hive project is a modern online marketplace designed to transform the way regional craft persons from remote or isolated locations interact with a worldwide consumer base. Its main goal is to make it easier for historical and cultural artifacts to be exhibited and sold. By doing so, it not only helps to preserve the world's cultural legacy but also provides small companies and rural households with economic stability. The platform gives local artisans direct access to global marketplaces so they may sell their culturally relevant objects for more money, improving their standard of living and helping their communities.

Additionally, Heritage Hive is essential to the preservation of cultural heritage since it provides traditional crafts with a worldwide platform. This exposure guarantees that these distinctive artistic forms are acknowledged and valued globally. Heritage Hive essentially tackles the crucial problem of bridging the gap between traditional goods producers and artists and global marketplaces.

9. References:

1. https://www.noupe.com/design/4-examples-of-bad-ecommerce-website-design.html
2. https://www.founderjar.com/inspiration/bad-website-design-examples/
3. https://www.arngren.net/elsykkel-3hjul-1.html
4. https://www.homebase.co.uk/
5. https://www.djangoproject.com/