HASHMART

Session: 2024-2028

Submitted by:

Muhammad Taaha 2024-DS-12

Supervised by:

Dr. Faiza Iqbal

Sir Sharjeel

Course:

CSC-103L Object Oriented Programming A

Institute of Data Science

University of Engineering and Technology,

Lahore, Pakistan

Table of Contents

1. Introduction

1.1 Project Overview

The Hashmart project is designed to revolutionize the e-commerce experience by providing a seamless, user-friendly platform for browsing, purchasing, and managing products. With a focus on modern technology, the Hashmart system integrates key features such as payment processing, product management, user authentication, and a smooth user interface. The system aims to enhance the shopping experience, ensuring convenience, security, and efficiency for both customers and administrators.

1.2 Scope and Objectives

The scope of the Hashmart project encompasses the development of a comprehensive e-commerce platform for managing products, processing payments, handling customer data, and facilitating smooth transactions. The project also aims to enhance the user experience through a user-friendly interface, secure payment integrations, and an efficient product catalog management system. The primary objectives of the project include:

- Streamlining the process of browsing, purchasing, and managing products.
- Providing a centralized platform for managing product details, orders, and customer information.
- Facilitating seamless payment processing and transaction management.
- Enhancing the overall shopping experience by implementing modern UI/UX designs and a secure, reliable back-end system.

1.3 Key Features

The Hashmart system incorporates several key features designed to offer a seamless e-commerce experience for both customers and administrators:

- **Product Management:** Efficient management of products, including detailed product information, images, prices, and stock quantities. Admins can easily add, update, and remove products from the system.
- **User Management:** Comprehensive user profiles, with features for account registration, login, and purchase history tracking. Admins can view and manage user details, including order history and payment statuses.
- **Shopping Cart & Checkout:** A robust shopping cart system that allows customers to add, update, and remove products. The checkout process includes secure payment integration via JazzCash, with options for single product and cart-based purchases.
- **Order Management:** Streamlined order processing system to track customer orders,

payment statuses, and delivery details. Admins can update and manage orders efficiently.

- **Product Ratings & Reviews:** Users can leave ratings and reviews for products, with average ratings calculated to help others make informed decisions.
- **Data Integrity and Security:** Ensuring the accuracy and security of product, order, and user data through encryption and best practices in data handling. Payment data is handled securely through the JazzCash payment gateway.
- **User-Friendly Interface:** A sleek, dark-themed, and responsive interface designed for easy navigation. Customers can browse, filter, and view products with ease, while administrators have a simple interface to manage the system's backend operations.

1.4 Technology Stack

The development of the Hashmart e-commerce system leveraged a wide range of tools and technologies to ensure optimal performance, scalability, and user experience. Below are the key tools and technologies utilized:

1.5 Programming Languages

1.5.1 Backend Development:

For the backend development of **Hashmart**, Python served as the primary programming language. Using **simple Django**, we built the core web application, allowing for efficient handling of business logic, user authentication, product management, and order processing. The backend leverages Django's built-in features for routing, templating, and database interaction, ensuring a seamless and scalable solution. It also includes integration with the **JazzCash API** for payment processing, ensuring secure and reliable transactions.

1.5.2 Frontend Development:

For the frontend development of **Hashmart**, we utilized **Bootstrap** as our primary framework to create responsive and visually appealing user interfaces. With **HTML**, **CSS**, and **JavaScript**, we built interactive elements that provide a seamless user experience, while **Bootstrap** facilitated rapid layout design, ensuring the application looks great on any device.

The responsive grid system and pre-built components of **Bootstrap** helped us quickly design and implement the interface, while allowing us to focus on customizing the design to align with **Hashmart**'s branding. The modular approach provided by **Bootstrap** ensured easy maintenance and future scalability of the frontend.

1.5.3 Platform Access:

Currently, **Hashmart** operates entirely as a web-based application, providing both customers and administrators with seamless access through any modern browser.

1.6 Database Management System (DBMS)

SQLite was selected as the relational database management system (RDBMS) for **Hashmart** to store and manage product information, customer accounts, order history, and payment details. Leveraging the lightweight yet efficient features of **SQLite**, we ensured data integrity, consistency, and reliability, supporting fast and reliable data retrieval and manipulation operations throughout the **Hashmart** system. The choice of **SQLite** provides a seamless database solution, ensuring smooth operation and scalability for the e-commerce platform.

1.7 Version Control

For the Hashmart project, Git, along with GitHub, was employed for version control throughout the development process. Although it was a solo project, GitHub provided a structured environment to manage and track changes to the codebase. It allowed for efficient version management, enabling the user to experiment with new features or fix bugs without the risk of losing progress. GitHub also facilitated the backup of the entire project, ensuring that the code was safely stored and accessible across different devices. The commit history allowed for transparent tracking of changes, making it easier to review the project's progression over time and ensuring accountability for each update made during development.

1.8 Development Environment

For the Hashmart project, an Integrated Development Environment (IDE) such as Visual Studio Code provided a comprehensive platform for writing, debugging, and testing the code. This tool offered a streamlined development experience with features like syntax highlighting, IntelliSense for code suggestions, and an integrated terminal for efficient workflow. Visual Studio Code's robust debugging capabilities allowed for quick identification and resolution of issues, enhancing productivity. Additionally, the use of extensions for Django and Python improved code quality and testing efficiency, making the development process smoother and faster, despite it being a solo project.

1.9 Web Pages

Web pages played a crucial role in providing an interactive and engaging user interface for the Hashmart project. Designed with a focus on user-friendliness and intuitive navigation, the web pages allowed users to effortlessly browse products, manage their carts, complete transactions, and interact with features like product ratings and chat support. The clean, dark-themed UI combined with dynamic components such as carousels and responsive layouts ensured a smooth user experience across devices. These design

elements significantly enhanced the overall usability and effectiveness of the platform, making it easier for users to engage with the system and complete their shopping journey with ease.

2. Users of the Project

1. Customers: Customers are the primary users of the platform. They can browse and search for products, view details, add items to their cart, complete purchases through integrated payment options like JazzCash, rate products, and interact with the built-in chatbot for support. The platform is designed to provide a smooth and intuitive shopping experience tailored to their needs.

2. Admin (Web Owner): The Admin, who is also the web owner, has full control over the backend and administrative operations of Hashmart. Responsibilities include adding and managing product listings, updating quantities, overseeing user registrations, processing orders, handling post ratings, and monitoring payment transactions. The admin ensures the platform remains secure, up-to-date, and fully functional.

By addressing the specific needs of both customers and the web owner, Hashmart creates a well-balanced and efficient e-commerce environment.

3. Database Overview

3.1 CRUD Database

The CRUD (Create, Read, Update, Delete) database acts as the operational core of the Hashmart e-commerce platform. It is responsible for managing essential transactional data, including user information, product details, cart data, order records, payment information, and product ratings.

3.1.1 Business Rules and Explanations:

3.1.1.1 Product Data Management:

Business Rule: All product records must contain complete and accurate details, including product name, description, price, quantity, and category.

Explanation: This rule ensures that the CRUD database maintains high-quality product data, which is vital for providing users with clear information, supporting smooth operations, and avoiding confusion during purchases.

3.1.1.2 Order and Transaction Records:

Business Rule: Each order must include user details, product(s) purchased, quantities, total amount, and payment status.

Explanation: By enforcing this rule, the system ensures all transactions are properly recorded, allowing the admin (web owner) to manage and track sales effectively and provide support or refunds when

necessary.

3.1.1.3 Cart and Checkout Functionality:

Business Rule: The cart must accurately reflect the selected products and quantities, and update automatically with any changes.

Explanation: This supports a reliable and real-time shopping experience, ensuring that users see the correct totals and can proceed to checkout without errors.

3.1.1.4 Product Ratings and Reviews:

Business Rule: Each user may rate a product only after purchase, and each product must reflect the average rating based on all user submissions.

Explanation: This promotes fair and trustworthy feedback, helps other users make informed buying decisions, and allows the admin to evaluate product popularity and performance

4. Admin Product Management Mechanism

Here are the steps involved in how the admin manages products within the Hashmart system:

4.1 Add New Product

- Admin accesses the backend dashboard and fills in product details (name, description, category, price, stock quantity, image, etc.).
- On form submission, the product is saved into the product table in the database.

4.2 Update Existing Product

- Admin selects an existing product from the product list.
- Updates one or more fields (e.g., price, stock, description, or image).
- Changes are saved to the database and reflected immediately on the frontend.

4.3 Delete Product

- Admin selects a product and clicks delete.
- The product record is either permanently deleted or soft-deleted (marked inactive) based on the system's policy.

4.4 Manage Product

- Admin can create, edit, or delete product categories.
- These categories help users filter and search products easily.

4.5 Monitor Stock & Inventory

- Admin panel shows current stock status (e.g., low stock alerts).

- Admin can restock or mark items as "out of stock," preventing further purchases.

5. Entity-Relationship Diagram (ERD)

Below is the Entity-Relationship Diagram (ERD) illustrating the database structure of the CRUD database.

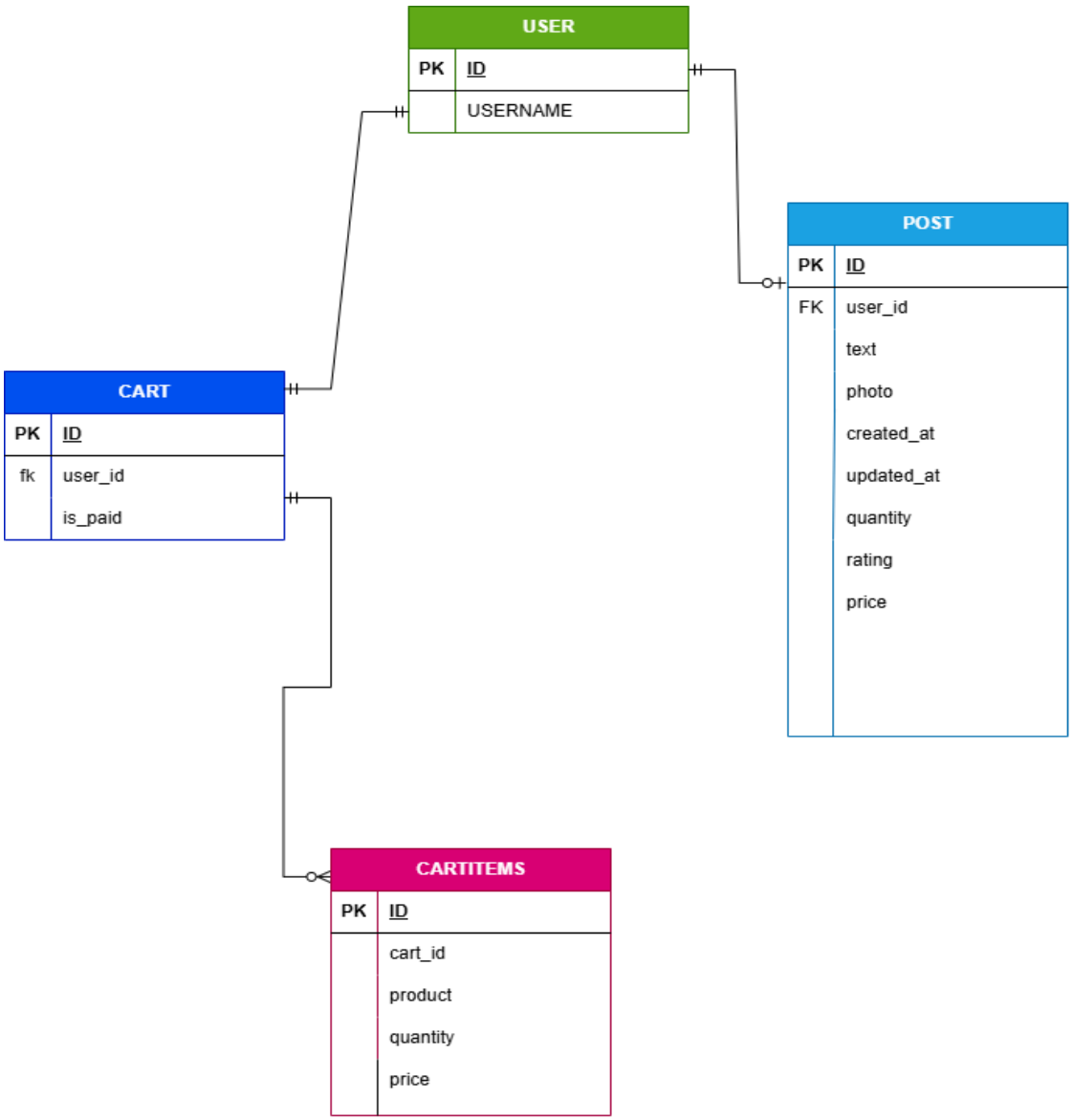

Figure 1: ER Diagram of CRUD database

Explanation

User: Represents any registered account in Hashmart—customers who browse and purchase products (and administrators who manage the store). Inherits from Django's built-in User model, storing authentication and profile information.
Product: Stores all catalog items available for sale. Tracks who listed it (user), descriptive text, optional image, inventory (quantity), base price, and aggregate rating.
Cart: Represents a shopping session for a User. Flags (is_paid) whether the cart has been checked out (converted into an order) versus still active or abandoned.
Cartitems: Line-items within a Cart. Each record links one Product to its containing Cart, storing the chosen quantity and the price at time of addition (to support price changes later.

5.1 Relationship Participation

5.1.1 User-post:

Participation: Mandatory for User, Optional for Post.

Every product post in Hashmart must belong to a registered user, but a user account doesn't have to have any posts.

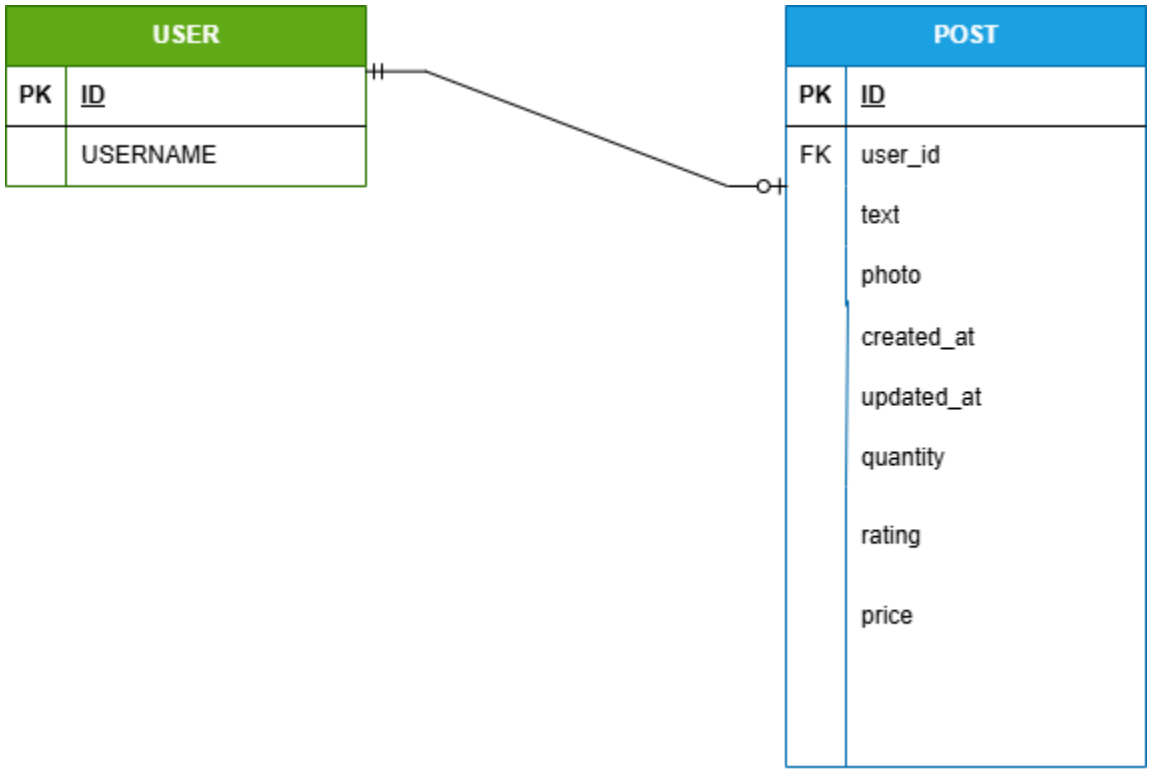

Figure 2: Relationship between User and post

5.1.2 User-cart:

Participation: Mandatory for User, Optional for Cart.

Every Cart must be associated with exactly one User (i.e. you can't have a stray cart in the system without an owner). This mirrors common e-commerce setups: you always know who "owns" a cart, but not every user account will necessarily have an active or past car

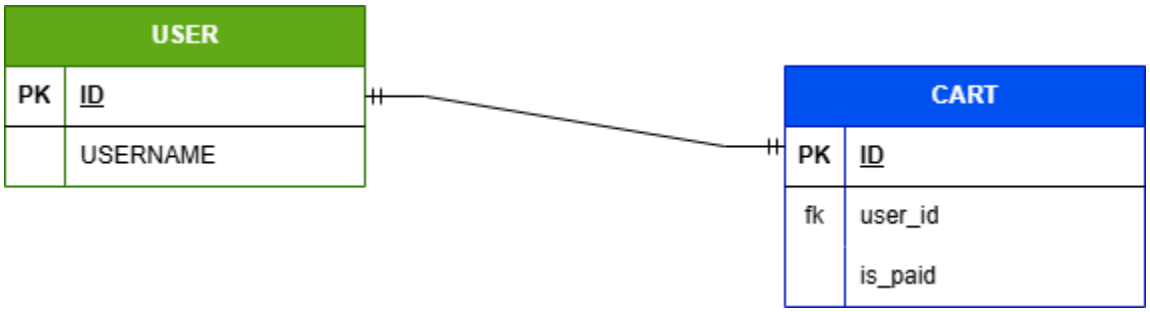

Figure 3: Relationship between User and Cart

5.1.3 Cart-Cartitems:

Participation: Mandatory for Cart and Cartitems.

In other words, you can't have an "empty" cart in the system—each shopping cart must contain at least one line-item—and every line-item record must belong to a valid cart.

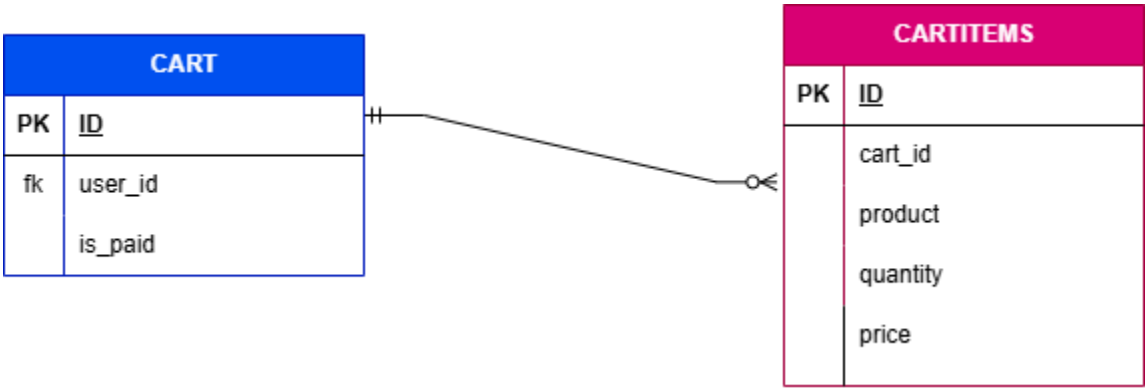

Figure 4: Relationship between Cart and Cartitems

6. Data Flow Diagrams

6.1 Level zero

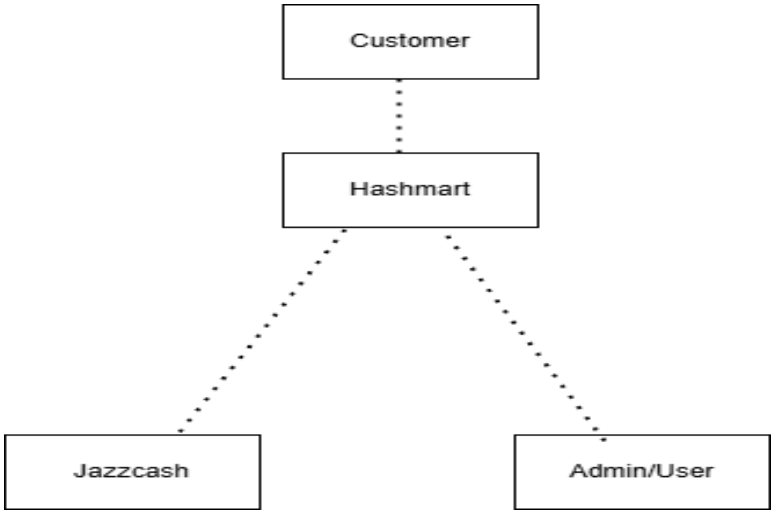

Figure 5: level zero DFD

6.2 Level One

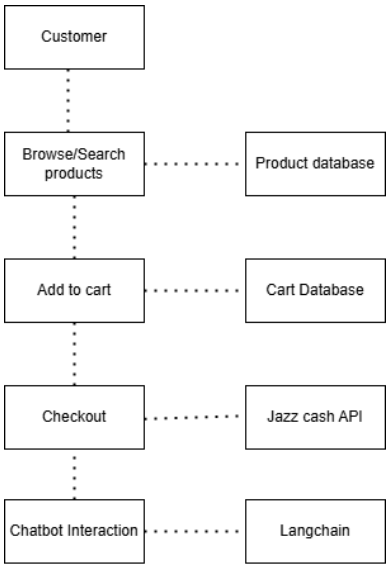

Figure 6: level 1 DFD

7. View Product Performance

- Admin can view stats like number of purchases, average rating, and revenue generated for each product.
- This helps in understanding customer preferences and improving inventory strategy.

8. Web Pages

The Hashmart system will feature a modern, responsive, and dark-themed user interface accessible via standard web browsers. These web pages will offer an intuitive experience for customers and administrators alike, enabling them to interact with various functionalities, such as browsing products, managing carts, completing checkouts, tracking orders, and submitting product reviews.

8.1 Landing Page

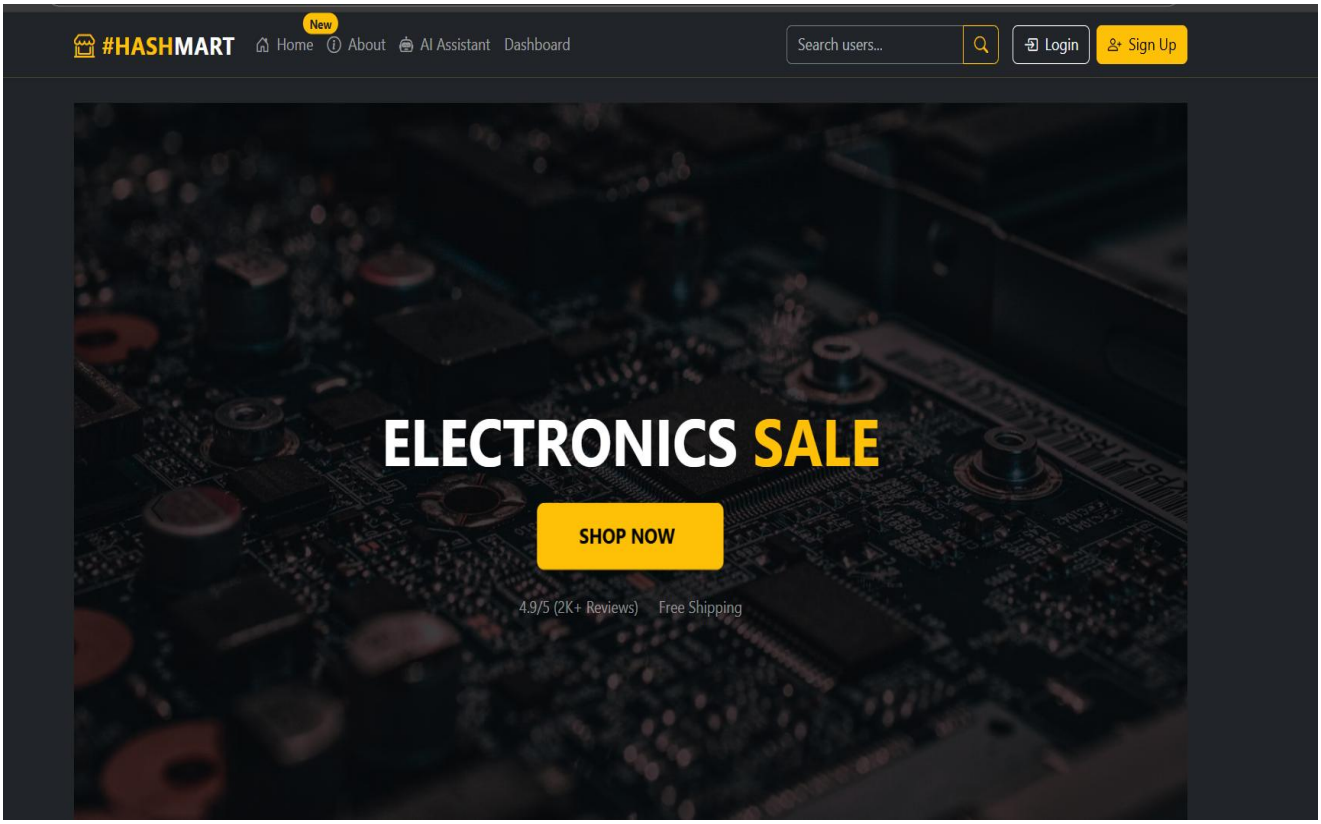

Figure 7: landing page

8.2 About Page

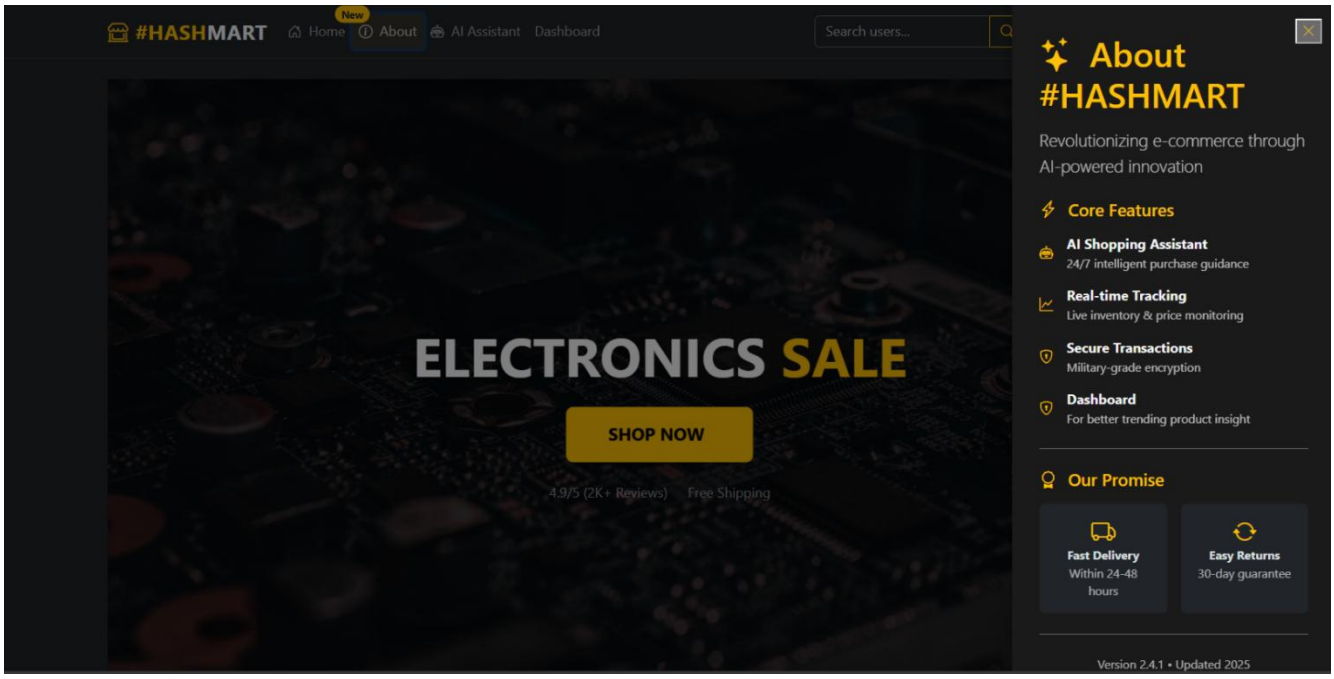

Figure 8: About page

8.3 AI chat-bot

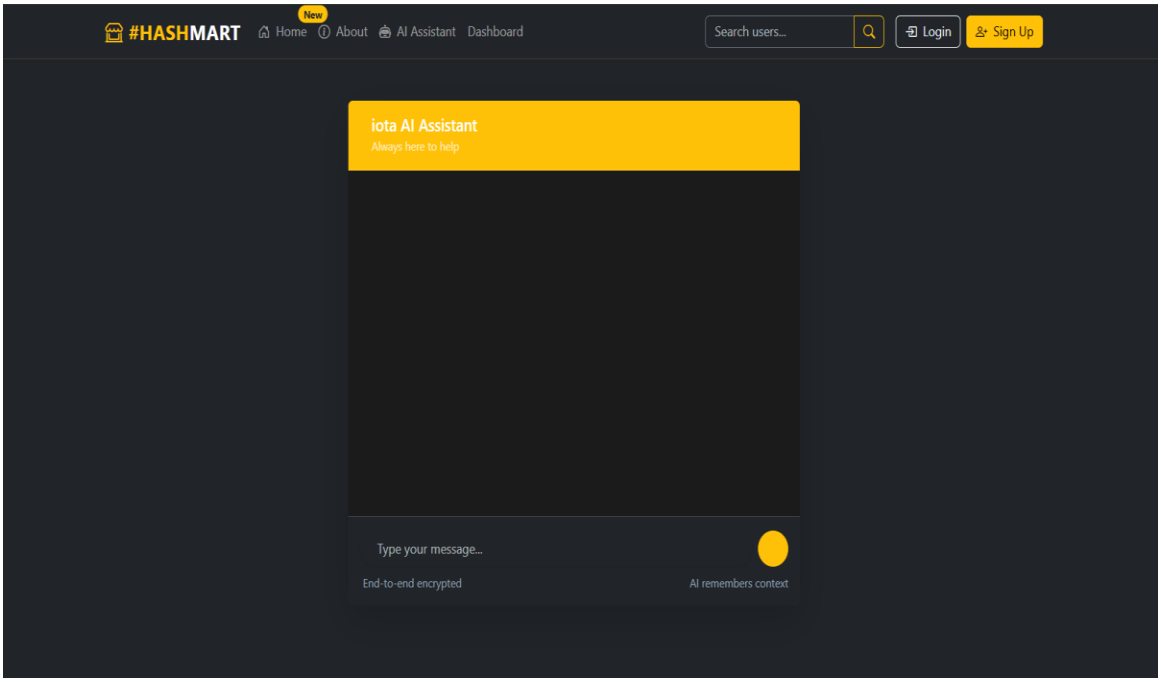

Figure 9: AI chat bot

8.4 Trend Capture Dashboard

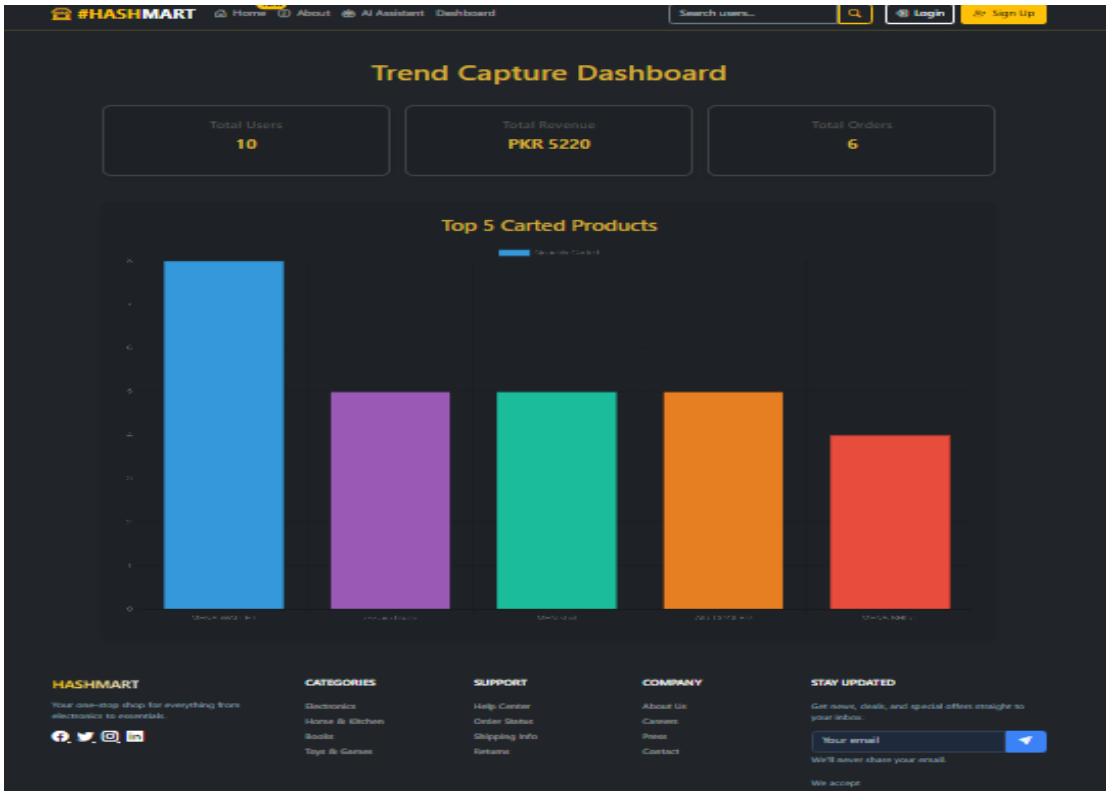

Figure 10: Dashboard

8.4 Home page

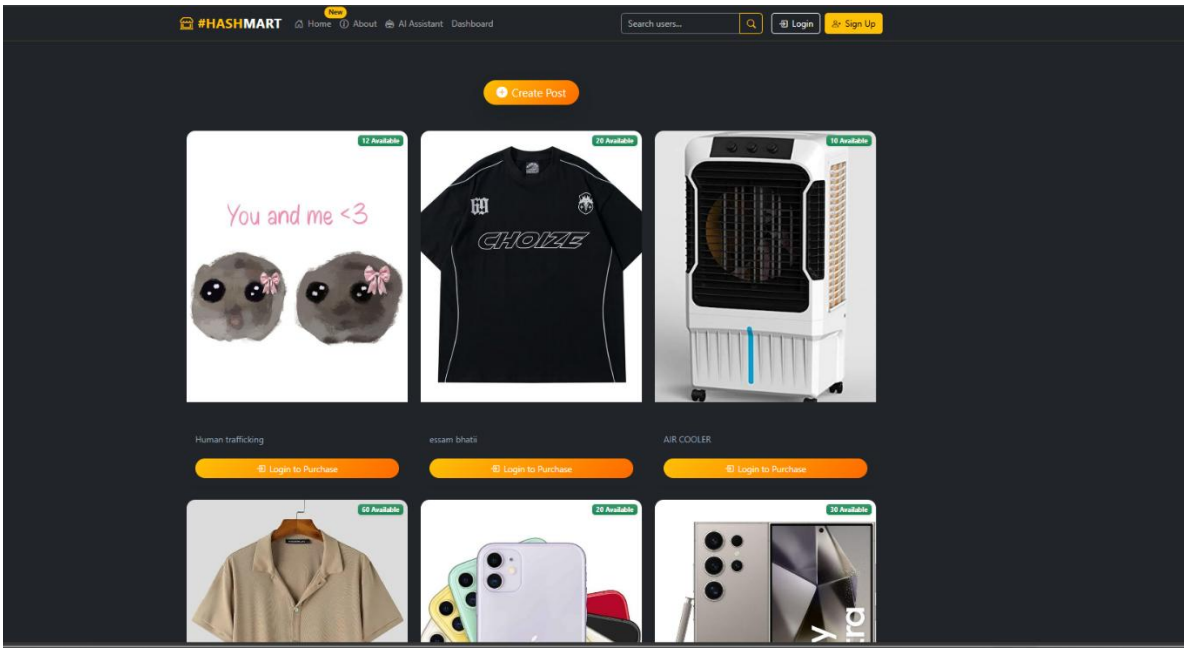

Figure 11: Homepage

8.5 Admin pannel

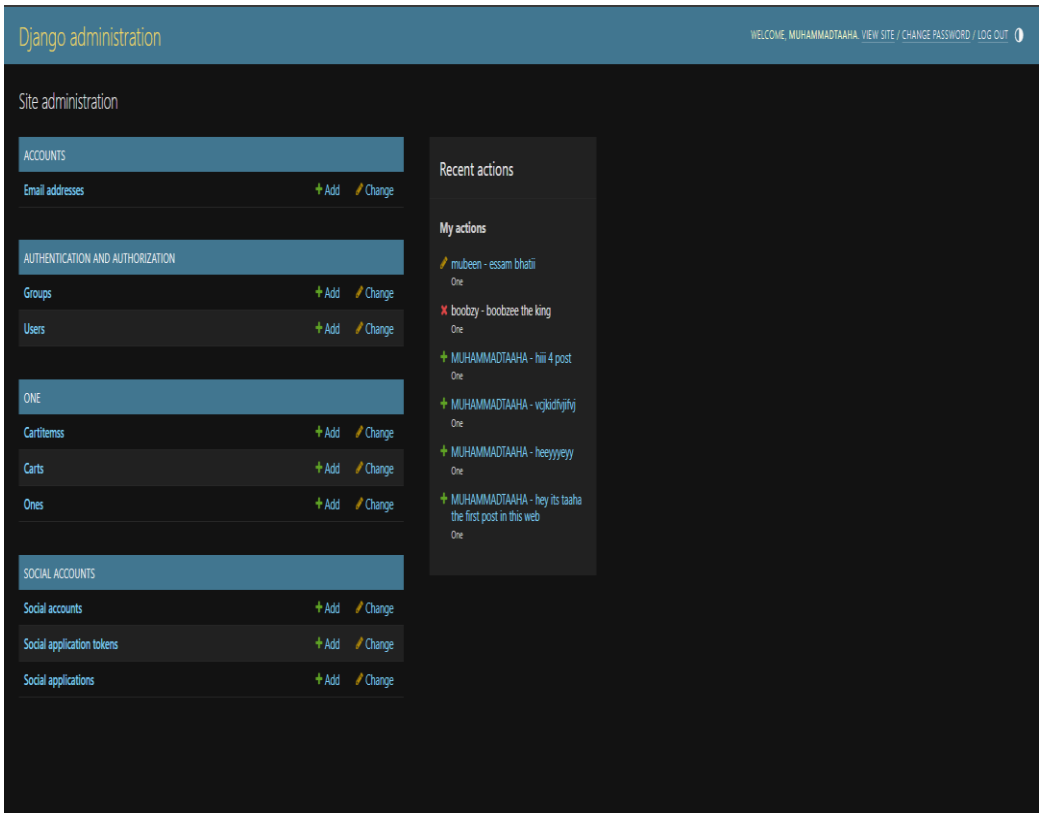

Figure 12: Admin pannel

9. Dependency

9.1 Functional Dependency

In Hashmart's database, functional dependencies occur within each table where one attribute determines another. For example, in the **Product** table (model One), the primary key id functionally determines all other attributes (user_id, text, photo, created_at, updated_at, quantity, rating, price). Similarly, in **CartItem**, the id key determines its cart_id, product_id, quantity, and price.

9.2 Full Functional Dependency

A full functional dependency exists when a non-key attribute depends on the entire composite key rather than part of it. In **CartItem**, the combination of (cart_id, product_id) fully determines the quantity and price. You cannot derive quantity or price from just cart_id or just product_id—you need both.

9.3 Existence Dependency

Certain records in Hashmart cannot exist without their parent records:

- **Product** (One) records depend on a **User**: you cannot have a product without an owner.
- **Rating** records depend on both **User** and **Product**: every rating must reference an existing user and product.
- **Cart** records depend on a **User**: each cart must belong to a registered account.
- **CartItem** records depend on a **Cart** (and implicitly a **Product**): you cannot have a line-item without a cart, nor can that line-item point to a non-existent product.

10 Relationship Strength

10.1 Strong Relationship

A strong (identifying) relationship means a child's primary key does *not* include the parent's key, and the child can be uniquely identified on its own.

10.1.1 User ↔ Product

Each product has its own primary key and simply holds a foreign key to its owner. The product record is independent and uniquely identifiable by its id. This relationship is strong because the existence and identification of an employee record do not depend on the presence of any other entity.

10.1.2 User ↔ Order

- Each Order has a unique order_id (PK) and contains a user_id (FK).
- The order is identifiable without combining user information in the key

10.1.3 Review ↔ Product

- Each Review has a unique review_id and stores product_id and user_id as foreign keys.
- Review exists independently once created.

10.2 Weak Relationship

These relationships **depend on other entities' keys** for uniqueness. Often represented by **composite keys** in relational logic, even if Django uses a surrogate primary key.

10.2.1 CartItem ↔ Cart, Product

- The CartItem table links a cart and a product.
- A cart_id + product_id combo uniquely identifies each cart item.
- Cart items cannot exist without both.

10.3 Weak Entities

- **Rating:**
  – Represents the rating given by a user to a specific product (One).
  – Lacks a standalone primary key; instead, it uses a composite key (user, post) defined by unique_together to ensure uniqueness.
  – Depends on both the User and the One (product) entity for identification.
  – Cannot exist independently without being linked to a specific user and product.
- **Cartitems:**
  – Represents individual product items added to a specific cart.
  – Does not have a unique primary key that identifies each item record on its own.
  – Relies on a combination of foreign keys (cart and product) for identification.
  – Dependent on the Cart and One (product) entities to exist.

10.4 Strong Entities

- **User (from django.contrib.auth.models):**
  – Represents registered users of Hashmart.
  – Possesses a primary key attribute (id) that uniquely identifies each user.
  – Stands independently and is referenced by multiple other models (e.g., One, Rating, Cart).
- **One (Product/Post):**
  – Represents a product or post listed by a user on Hashmart.
  – Has a primary key (id) automatically created by Django.
  – Contains attributes such as text, photo, created_at, quantity, rating, and price.
  – Stands independently and is referenced in Rating and Cartitems
- **Cart:**
  – Represents a shopping cart associated with a user.
  – Has a primary key (id) and is linked to the user placing the order.
  – The cart is used to hold items before payment confirmation.

11 OOP In Hashmart

11.1 OOP Concepts Used

- Encapsulation
- Inheritance
- Polymorphism

11.2 Why to use OOP

In Hashmart, object-oriented programming principles such as encapsulation, inheritance, and polymorphism play a vital role in building a modular, secure, and scalable e-commerce system. Encapsulation is used to safeguard sensitive attributes like product quantity, price, and ratings by restricting direct access and placing control logic within methods and properties (e.g., update_quantity() and average_rating). This ensures data consistency and prevents invalid operations, like overselling stock. Inheritance is heavily utilized through Django's architecture, where models inherit from django.db.models.Model and views from generic base classes like ListView or CreateView, allowing the reuse of pre-built features such as ORM behavior, form handling, and template rendering — reducing redundancy and speeding up development. Polymorphism is applied when different classes define the same method name but behave differently — for example, the __str__() method in both One and Rating models, or when serializers and payment gateways override shared functions like to_representation() or pay(). This allows the system to interact with various components using a common interface, enabling flexible logic extensions and reducing tight coupling between parts of the application. Together, these principles make the Hashmart codebase more maintainable, reusable, and adaptable to new features or business logic.

11.3 Sample Code Snippet (OOP Style)

Inheritance Example

This is the sample for the inheritance this this the basic template named layout.html all the other html files are generated by the basic layout provided by the layout.html this is inheritance

```
You, 3 weeks ago | 1 author (You)

{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}ONE{% endblock %}</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <!-- Bootstrap Icons -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css" rel="stylesheet">

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet"
          integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">
</head>

<body data-bs-theme="dark">



    <nav class="navbar navbar-expand-lg navbar-dark bg-dark shadow-sm">
        <div class="container">
            <!-- Brand with logo/icon -->
            <a class="navbar-brand d-flex align-items-center" href="{% url 'one:all_list' %}">      You, 3 weeks ago • Add all project files
                <i class="bi bi-shop-window text-warning me-2"></i>
                <span class="fw-bold"><span class="text-warning">#HASH</span>MART</span>
            </a>



            <!-- Mobile toggle button -->
            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarContent"
                    aria-controls="navbarContent" aria-expanded="false">
                <span class="navbar-toggler-icon"></span>
            </button>

            <!-- Navbar content -->
            <div class="collapse navbar-collapse" id="navbarContent">
                <!-- Main navigation -->
                <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                    <li class="nav-item">
                        <a class="nav-link position-relative" href="{% url 'one:all_list' %}">
                            <i class="bi bi-house-door me-1"></i> Home
                            <span class="position-absolute top-0 start-100 translate-middle badge rounded-pill bg-warning text-dark">
                                New
                            </span>
                        </a>
                    </li>
                    <li>
                      <a class="nav-link" href="#about">
                        <i class="bi bi-info-circle me-1"></i> About
                    </a>

                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url 'one:chat' %}">
                            <i class="bi bi-robot me-1"></i> AI Assistant
                        </a>
                    </li>

                    <li class="nav-item">
                        <a class="nav-link" href="{% url 'one:dashboard_view' %}">
                         Dashboard
                        </a>
                    </li>

                    <!-- User dropdown -->
                    {% if user.is_authenticated %}
                    <li class="nav-item dropdown">
                        <a class="nav-link dropdown-toggle d-flex align-items-center" href="#" role="button"
                           data-bs-toggle="dropdown" aria-expanded="false">
                            <i class="bi bi-person-circle me-1"></i> {{ user.username }}
                        </a>
                        <ul class="dropdown-menu dropdown-menu-dark border-0 shadow">
                            <li><a class="dropdown-item" href="{% url 'one:all_list' %}">
                                <i class="bi bi-house me-2"></i> Home
                            </a></li>
                            <li><a class="dropdown-item" href="{% url 'one:user_posts' user.username %}">
                                <i class="bi bi-collection me-2"></i> Your Posts
                            </a></li>
                            <li><hr class="dropdown-divider"></li>
                            <li><a class="dropdown-item" href="#">
                                <i class="bi bi-gear me-2"></i> Settings
```

Figure 13: Layout.html

Abstraction

This is the example of abstraction in this I will show you two files one file is views.py and the other file is urls.py there are the function which are defined in the views.py and I will then use them in the urls.py via abstraction

This is views.py

```
views.py M ×
hashmart > one > views.py > ...
You, 3 weeks ago | 1 author (You)
from django.shortcuts import render
from django.db.models import Avg
from .models import One, Rating
from .forms import U_form,User_registration
from django.shortcuts import get_object_or_404 ,redirect
from django.shortcuts import render, get_object_or_404, redirect
from .models import One
from .models import Cart,Cartitems
from .forms import U_form
from .forms import RatingForm
from django.contrib.auth.decorators import login_required
from django.contrib.auth import login
from huggingface_hub import InferenceClient
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import json
from django.http import HttpResponse
import requests
import traceback
from django.shortcuts import render
from django.contrib import messages
from django.shortcuts import render
import datetime        You, 3 weeks ago • Add all project files
import hmac
import hashlib
from django.views.decorators.csrf import csrf_exempt


# Home Page View
def index(request):
    return render(request, 'index.html')

def all_list(request):
    posts = One.objects.all().order_by("-created_at")
    print("DEBUG - Posts:", list(posts.values('id', 'text')))  # Print actual data
    context = {
        'posts': posts,
        'debug_test': "THIS SHOULD APPEAR IN TEMPLATE"
    }
    return render(request, "all_list.html", context)
# Create New Item View
@login_required
def create(request):
    if request.method == "POST":
        form = U_form(request.POST, request.FILES)
        if form.is_valid():
            one = form.save(commit=False)
            one.user = request.user
            one.save()
            return redirect("one:all_list")
    else:
        form = U_form()
    return render(request, "Oneform.html", {'form': form})

def edit(request, post_id):
```

Figure 14: views.py

This is the urls.py files giving the paths to the views.py

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

from . import views
from .views import chat_view, empty_chat_view, addtocart,checkout,checkin,drop_cart,jazzcash_payment,search_view,about,dashboard_view
from django.contrib.auth import views as auth_views

app_name = 'one'

urlpatterns = [

    path('', views.index, name="index"),
    path('all/', views.all_list, name="all_list"),
    path('create/', views.create, name="create"),
    path('<int:post_id>/delete/', views.delete, name="delete"),
    path('<int:post_id>/edit/', views.edit, name="edit"),
    path('register/', views.register, name="register"),
    path('accounts/logout/', auth_views.LogoutView.as_view(), name='logout'),
    path('user/<str:username>/', views.user_posts, name='user_posts'),
    path('chat/', views.chat_view, name='chat'),
    path('empty-chat/', views.empty_chat_view, name='empty_chat_view'),
    path('addtocart/<int:post_id>/<str:username>/', views.addtocart, name='addtocart'),
    path('checkout/', views.checkout, name='checkout'),
    path('checkin/', checkin, name="checkin"),
    path('<int:post_id>/drop_cart/<str:username>/', views.drop_cart, name='drop_cart'),
    path('cart/<str:username>/', views.view_cart, name='view_cart'),
    path('jazzcash/', views.jazzcash_payment,name="jazzcash_payment"),
     path("search/", search_view, name="search_view"),
     path("about/", about, name="about"),
     path("dashboard_view", dashboard_view, name="dashboard_view"),


]
```

Figure 15: Urls.py

Encapsulation

Update_quantity(purchased_quantity)

Location: One model (Product)

Purpose:

Encapsulates the logic for reducing product stock after a purchase. Prevents direct manipulation of the quantity field and adds validation to avoid negative inventory.

Average_rating

Location: One model (Product) — Defined as a @property

Purpose:

Calculates the average rating for a product based on related Rating entries.

Total_price

Location: Cartitems model — Defined as a @property

Purpose:

Returns the total price of a cart item (product price × quantity).

Total_amount

Location: Cart model — Defined as a @property

Purpose:

Calculates the full cost of the shopping cart by summing up each item's total price.

This the models.py file for the example for encapsulation

views.py M urls.py models.py

hashmart > one > models.py > One

You, 3 weeks ago | 1 author (You)

```python

from django.db import models
from django.contrib.auth.models import User

You, 3 weeks ago | 1 author (You)
class One(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField(max_length=240)
    photo = models.ImageField(upload_to="photos/", blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    update_at = models.DateTimeField(auto_now=True)
    quantity = models.PositiveIntegerField(default=0)
    rating = models.FloatField(default=0)        You, 3 weeks ago • Add all project files
    price=models.PositiveIntegerField(default=0)
    def __str__(self):
        return f'{self.user.username} - {self.text}'

    @property
    def average_rating(self):
        ratings = self.ratings.all()
        if ratings.exists():
            total = sum(r.value for r in ratings)
            return round(total / ratings.count(), 2)
        return 0
    def update_quantity(self, purchased_quantity):
        """Update the quantity of the product when a purchase is made."""
        if purchased_quantity <= self.quantity:
            self.quantity -= purchased_quantity
            self.save()
        else:
            raise ValueError("Not enough stock available.")

You, 3 weeks ago | 1 author (You)
class Rating(models.Model):
    post = models.ForeignKey(One, related_name='ratings', on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    value = models.FloatField()

    You, 3 weeks ago | 1 author (You)
    class Meta:
        unique_together = ('user', 'post')
    def __str__(self):
        return f'{self.user.username} rated {self.value} for {self.post}'
You, 3 weeks ago | 1 author (You)
class Cart(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    is_paid=models.BooleanField(default=False)
You, 3 weeks ago | 1 author (You)
class Cartitems(models.Model):
    cart=models.ForeignKey(Cart, related_name='cartitems',on_delete=models.CASCADE)
    product=models.ForeignKey(One, on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(default=1)
```

Figure 16: Models.py

12 Limitations

12.1 Growing Pains

As Hashmart gains more users and listings, the current setup might struggle to keep up with increased demand. Without optimization and scalable infrastructure, performance issues such as slower load times or delayed transactions may emerge.

12.2 Bridging the Gap

Integrating Hashmart with other platforms—like third-party payment gateways, CRMs, or inventory systems—may require extra effort. Differences in data structures, APIs, or technologies can complicate seamless communication between systems.

12.3 Digital Vulnerabilities

The platform is potentially exposed to cybersecurity threats, such as unauthorized access, data leaks, and malicious attacks. Strengthening authentication, data protection, and monitoring measures is crucial to protect both the business and users

12.4 Getting Everyone Onboard

Not all users will adapt quickly. Resistance to new tech, unfamiliar UI design, or lack of training resources might discourage adoption and regular use, especially for non-tech-savvy customers.

12.5 Maintenance Overhead

Maintaining and updating Hashmart takes time, effort, and manpower. Frequent bug fixes, new features, and database upkeep can strain resources if not planned well.

12.6 Talking to Other Systems

Compatibility problems may surface when trying to connect Hashmart to other services, especially if those systems use different protocols or data formats. Achieving smooth integration might need additional middleware or translation layers.

13 Future Vision for Hashmart

13.1 Scaling for Success

Adopt a scalable architecture (e.g., cloud hosting, database optimization, caching strategies) to support growing product listings, user registrations, and concurrent transactions without sacrificing performance.

13.2 Smooth System Interactions

Enhance API support and data structure compatibility to easily integrate Hashmart with other systems like payment gateways, inventory trackers, analytics tools, and marketing platforms for streamlined operations.

13.3 Fortified Security Framework

Implement stronger security protocols such as multi-factor authentication, end-to-end encryption, and real-time intrusion detection to protect sensitive customer, order, and payment data from cyber threats

13.4 User Experience Improvements

Continuously enhancing the user interface design, navigation, and functionality to improve user experience and increase user satisfaction and productivity.

13.5 Smart Data-Driven Insights

Leverage machine learning and predictive analytics to understand shopping behaviors, optimize inventory, recommend products, and forecast sales trends—empowering smarter business decisions.

13.6 Anytime, Anywhere Access

Developing mobile applications or responsive web interfaces to provide users with anytime, anywhere access to the EPE system, facilitating remote performance evaluations and decision- making processes.

14 Conclusion

Hashmart presents a solid foundation for a dynamic, user-friendly e-commerce experience. The current system supports key functionalities like product listings, user ratings, and cart management, with room for growth. With strategic improvements in scalability, integration, security, and user engagement,

Hashmart can evolve into a robust platform capable of handling high volumes of traffic and transactions. By embracing innovation and staying user-focused, Hashmart is well-positioned to grow sustainably while delivering value to both customers and sellers.

15 Github Project Link

The project code and related materials can be found on GitHub at the following link:

https://github.com/Muhammad-Taaha/hashmart.git