SECURECART E-COMMERCE PLATFORM DEVELOPMENT

Stanley Shaw

October 2024

Contents

1. Introduction

E-commerce platforms are indispensable to the global economy, yet their handling of sensitive user data makes them prime targets for cyberattacks [1]. To address these challenges, *SecureCart* follows the V-Model software development lifecycle with a focus on stringent requirements, system design, and subsequent secure coding [2]. Developed using Django, Bootstrap, PostgreSQL, and Stripe, the platform incorporates robust security measures such as secure authentication, encrypted data transmission, and strict data handling [3, 4, 5, 6]. This report focuses not only on the foundational stages of the system but also highlights its implementation through secure code snippets, ensuring that potential vulnerabilities are identified and mitigated both before and during development.

2. Requirements Phase

2.1 Functional Requirements

During the requirements-gathering phase, I identified the essential functionalities needed by two primary user groups: the customers and the administrators. Validating these requirements ensures that no critical security or functional requirements are overlooked [7].

Customer Requirements

Functional Area	Requirements
Account Management	• Secure registration and login processes [8]. • Password recovery mechanisms [9]. • Optional multi-factor authentication (2FA) [10].
Product Browsing	• Display of product listings with images and descriptions. • Visibility of inventory levels. • Search and filtering capabilities to locate products efficiently.
Shopping Cart and Checkout	• Ability to add products to the cart. • Adjusting product quantities within the cart. • Integration of secure payment methods. • Order confirmation upon successful checkout.
Order History and Profile Management	• Viewing past orders and their statuses. • Seamless logout functionality. • Manage personal information.

Table 2.1: Customer Functional Requirements

Administrator Requirements

Functional Area	Requirements
Product Management	• Adding new products to the catalogue. • Updating existing product information and prices. • Removing discontinued or out-of-stock products. • Managing inventory levels to prevent stockouts or overstocking.
Order Management	• Viewing all customer orders and their details. • Marking orders as completed or shipped. • Managing refunds and processing returns efficiently.
User Management	• Viewing and managing user accounts. • Adjusting user roles and permissions. • Enforcing account-related security measures, such as password resets or account lockouts.
Reporting	• Generating sales summaries to track revenue. • Accessing inventory data to monitor stock levels. • Analyzing user activity to inform business decisions.

Table 2.2: Administrator Functional Requirements

Verification of these requirements involves ensuring that they are clear, feasible, and security-centric. I will review each of them in the context of cybersecurity best practices and compliance regulations, ensuring that the functionality is both realistic and necessary.

2.2 Non-Functional Requirements

Non-functional requirements define the overall standards the system must uphold, independent of specific functionalities. These requirements ensure that the *SecureCart* platform is secure, efficient, reliable, and compliant with relevant regulations.

Category	Requirements
Security	• Secure password hashing and salted storage • Mandatory HTTPS for all network communications • Option for two-factor authentication (2FA) • Input validation, parameterised queries, and output encoding to prevent common attacks
Performance and Scalability	• Handling peak traffic and large user loads without excessive latency [12] • Techniques like caching and load balancing to ensure responsiveness
Availability and Reliability	• High uptime, redundancy, and disaster recovery procedures • Reliable backup strategies and failover mechanisms
Compliance and Privacy	• Conformance to data protection and privacy regulations (e.g., GDPR) [11] • Proper storage and encryption of personally identifiable information (PII)

Table 2.3: Non-Functional Requirements for the SecureCart Platform

3. System Analysis and High-Level Design

3.1 Introduction

This section details how the *SecureCart* platform's requirements shape the overall structure and high-level design. The goal is to create an architecture that meets the needs of both customers and administrators, emphasising security, scalability, and maintainability.

3.2 Review and Consolidate Requirements

I began by validating the gathered requirements, ensuring completeness and consistency:

- **Functional Needs:**
  - *For Customers:* Registration, login, product browsing, cart/checkout, order history, profile management.
  - *For Administrators:* Product CRUD (Create, Read, Update, and Delete), order oversight, user role management, and reporting.
- **Non-Functional Aspects:**
  - **Security:** Encryption (in-transit and at-rest), secure authentication, authorisation, and privacy compliance.
  - **Performance:** Ability to handle peak traffic with minimal latency.
  - **Scalability:** Support for horizontal or vertical scaling as user load increases.
  - **Compliance:** GDPR or local data protection regulations.

These validated requirements form the basis for the system's high-level architecture.

3.3 High-Level Architectural Style

SecureCart adopts a **three-tier** architecture [13]:

- **UI Layer:** Implements the user interface (UI) for customers and administrators, ensuring a responsive and user-friendly design (e.g., Django templates, Bootstrap).
- **Business Logic Layer:** Manages core business logic, data processing, authentication, and security enforcement (e.g., Django services and middleware).
- **Data Layer:** Stores persistent data on the cloud (PostgreSQL) with strict access policies and encryption at rest.

This layered approach ensures clear separation of concerns, easier maintenance, and the ability to apply role-based security at distinct points in the data flow.

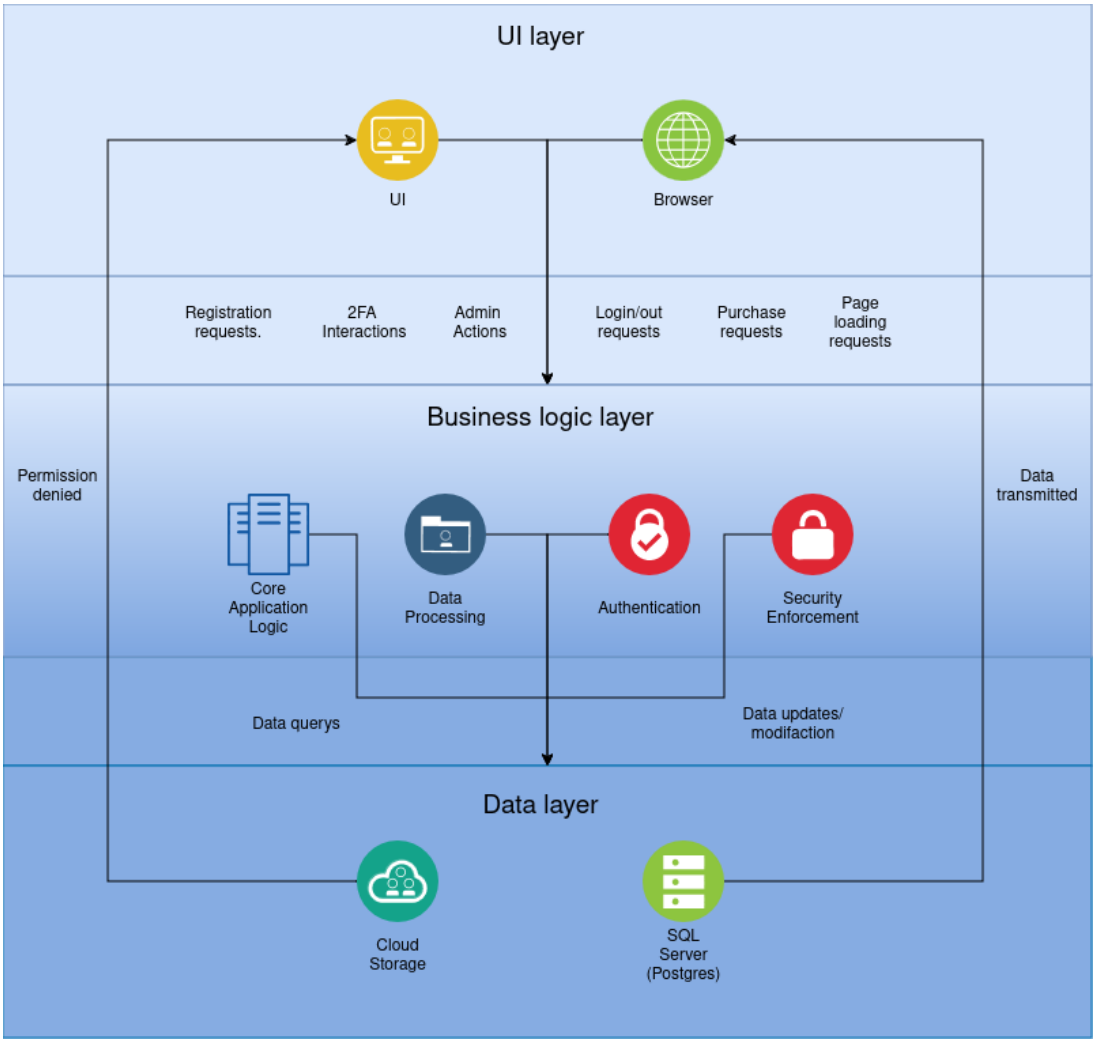

Figure 3.1: High level Component diagram

3.4 Key Components and Interactions

While the architecture is layered, there are critical components that handle specific functionalities. Below is a table highlighting the most important ones:

Component	Responsibilities and Notes
Authentication	• Enforces secure login (hashed passwords, optional 2FA). • Role-based access control differentiating admins vs. customers.
Product Management	• Allows admins to add or update product info and prices. • Inventory checks and validations to avoid negative stock.
Cart & Order Processing	• Manages cart sessions, checkout flows, and payment integration. • Updates order records and triggers inventory adjustments.
Reporting & Analytics	• Generates sales reports, user activity analytics. • Summarizes product performance to inform business decisions.
Security Middleware	• Handles HTTPS redirection, session management, and intrusion detection. • Parameterised queries and input validation to prevent SQL injection, XSS.

Table 3.1: High-Level Components and Their Roles

Inter-Layer Communication:

- **Frontend to Backend:** Requests pass via encrypted protocols (HTTPS), with CSRF tokens and session IDs.
- **Backend to Database:** Django's ORM or parameterised SQL queries uphold data integrity.
- **Admin Functions:** Advanced privileges allow direct product/user management, protected by role-based authentication checks.

3.5 UI Design

The user interface (UI) of the *SecureCart* platform is designed to provide a seamless and intuitive experience for both customers and administrators. In order to guide the early design process I created a navigational flow diagram and a homepage mock-up.

3.5.1 Navigational Flow

As shown in Figure 3.2, users can quickly progress from the login screen to browsing products, managing their cart, checking out, and viewing their profiles. Administrators can additionally navigate to the admin page where they can view and update both users and orders.

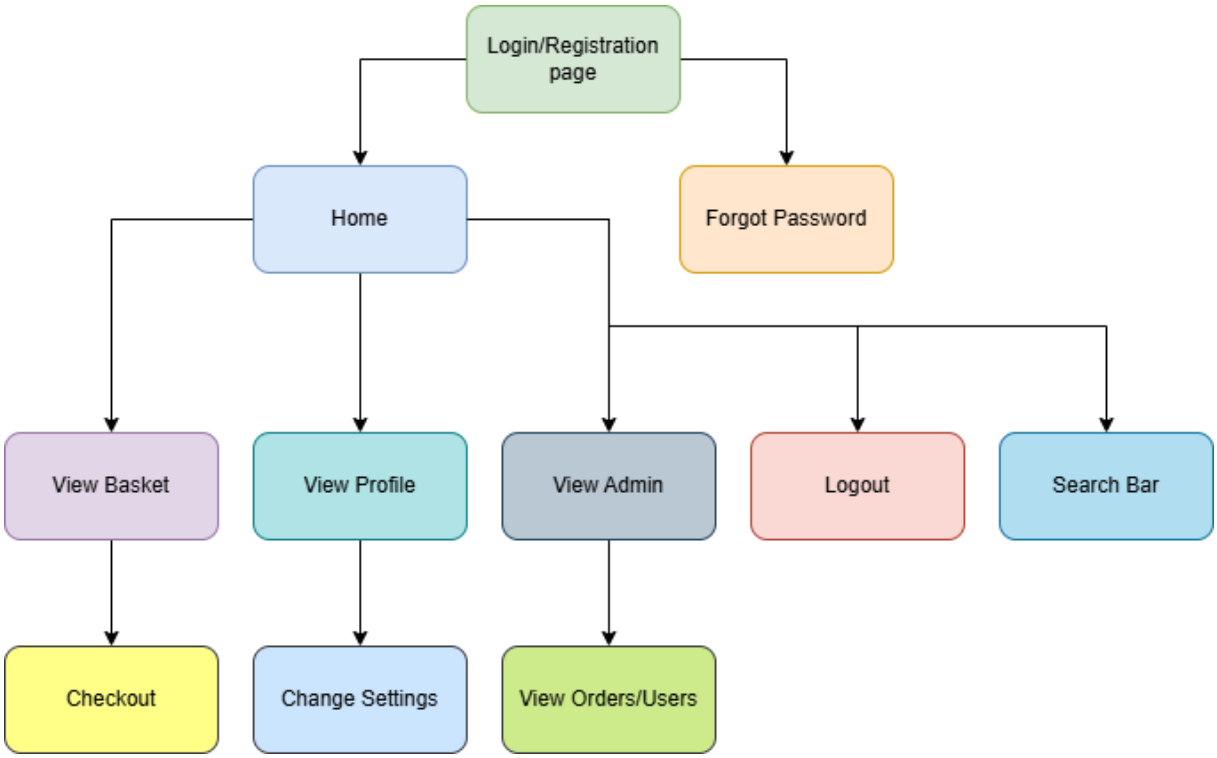

Figure 3.2: High-level navigational flow diagram.

3.5.2 Homepage Mock-up

Figure 3.3 shows a simple mock-up of the homepage. Key elements include a site-wide navigation bar with quick links to the home, cart, and profile pages, as well as a search feature and product listings.

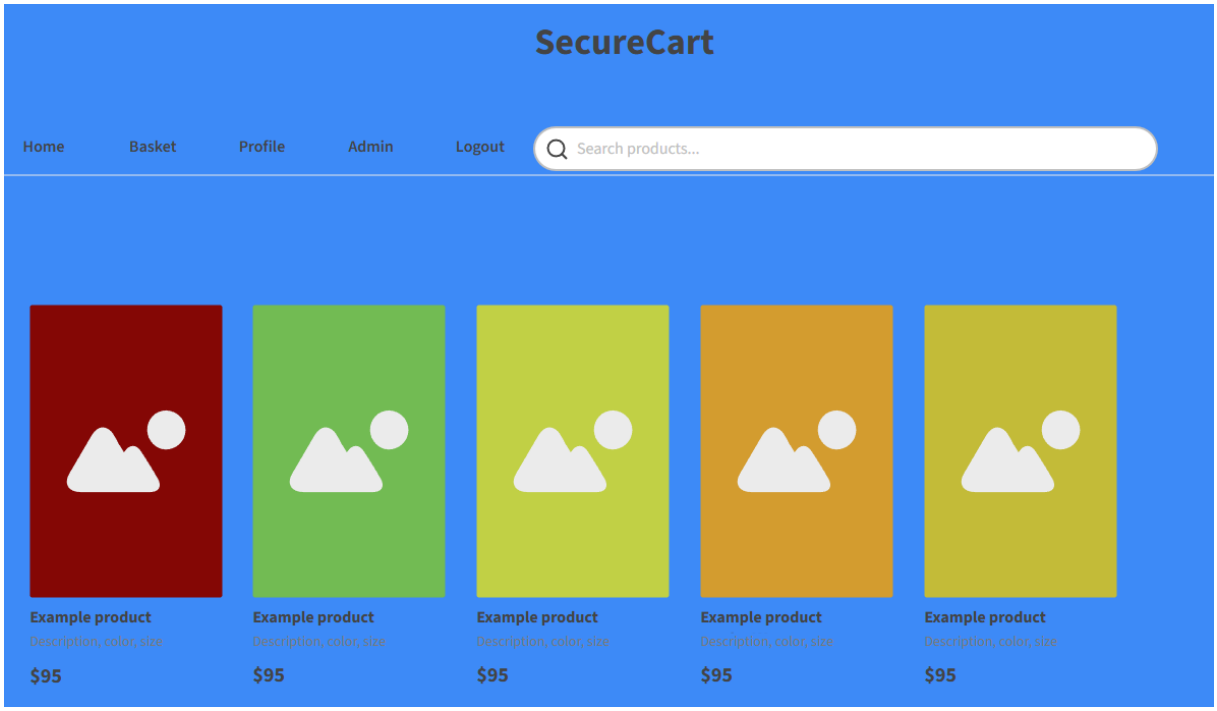

Figure 3.3: Mock-up of the SecureCart homepage.

3.5.3 UI Design Principles

- **Consistency:** An appealing colour scheme, typography, and layout.
- **Clarity:** Key actions (e.g., “Profile” or “Basket”) are clearly visible.
- **Responsiveness:** The layout adapts smoothly to various screen sizes.
- **Accessibility:** Semantic HTML, suitable labels, and high colour contrast ensure inclusivity.

3.6 Database Design

A robust and secure database is fundamental to the *SecureCart* platform, ensuring reliable storage of user data, product information, and transaction records. PostgreSQL is chosen as the primary data store due to its strong security features and scalability. Furthermore, the database will be hosted on AWS to allow for increased flexibility and scalability.

3.6.1 Database Query Design

In order to break down the database design into a simpler format, I created a flow diagram highlighting both users and admins' queries.

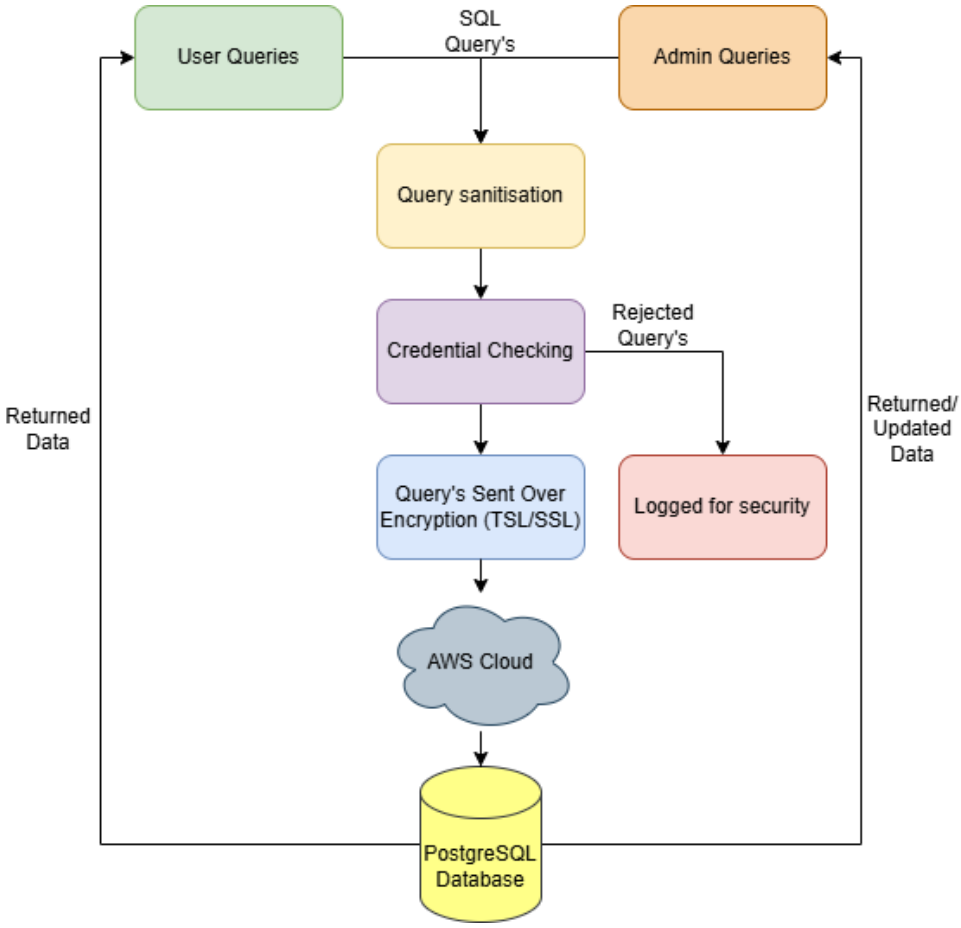

Figure 3.4: Database

3.7 Security Considerations

Security is integrated into each layer:

- **Encryption:** All data in transit via TLS/HTTPS; sensitive data at rest (e.g., passwords) hashed and salted using strong algorithms (PBKDF2) [14].
- **Authentication:** Django's auth system plus optional two-factor authentication for heightened security.
- **Authorisation:** Strict RBAC, ensuring customers cannot access admin endpoints or data.
- **Input Validation:** parameterised queries, server-side checks on all user inputs (product searches, checkout data, etc.) [15].
- **Monitoring & Logging:** Audit logs for key actions (logins, product changes, purchases) to detect anomalies.

3.8 Performance and Scalability

To accommodate growth and periodic traffic spikes, the design includes:

- **Efficient Database Queries:** optimising data retrieval with indexing and query optimisation.
- **Caching:** Reducing repeated computations or database queries by storing frequently accessed data temporarily.
- **Session Management:** Handling user sessions efficiently to maintain state across requests.
- **Static File Management:** Serving and optimising static files like CSS, JavaScript, and images for faster delivery.
- **Template Rendering:** Dynamically generating HTML content based on user data or application state.

4. Detailed Design and Development Phase

This section provides specific details on the implementation of each system component, emphasising secure design principles. Data flow diagrams, entity relationship diagrams, class diagrams and sequence diagrams are included to ensure clarity.

4.1 System Components

Layer	Description	Security Features
UI	User interface using Django templates and Bootstrap.	Content Security Policy (CSP), input validation.
Business Logic	Business logic, authentication, and data processing.	RBAC, CSRF protection, input validation.
Data	Persistent storage using PostgreSQL.	Encrypted fields, strict access controls.

4.2 Data Flow Diagrams

Level 1 Data Flow Diagram: High-level interaction between layers.

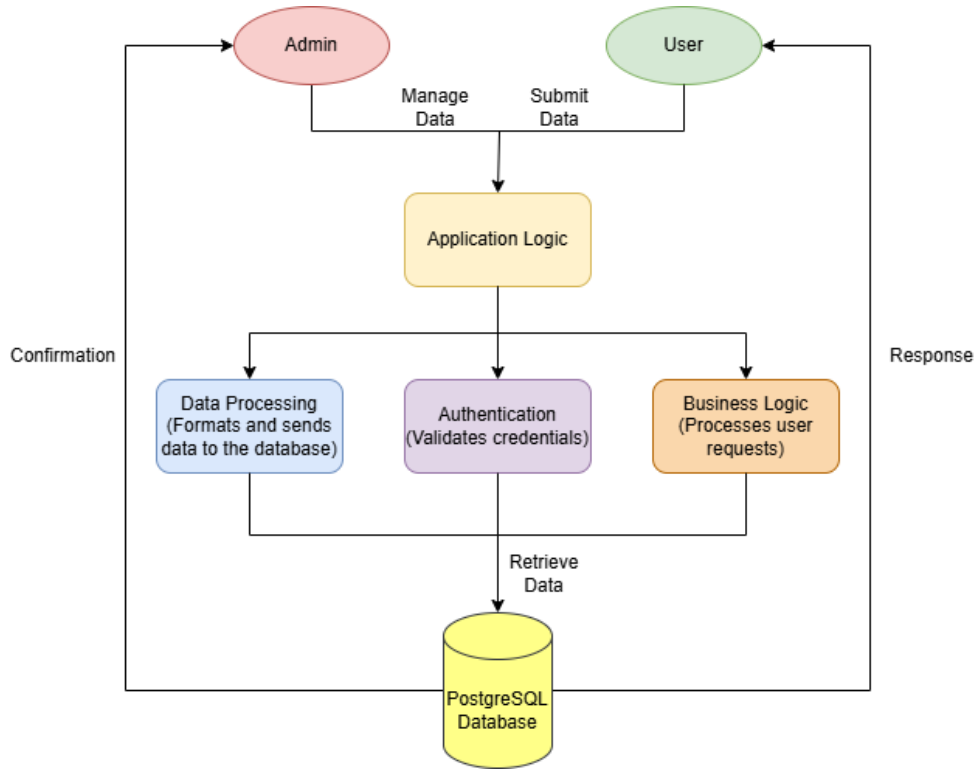

Figure 4.1: High-Level Data Flow Diagram.

Level 2 Data Flow Diagram: Low-level interaction between Users and the database.

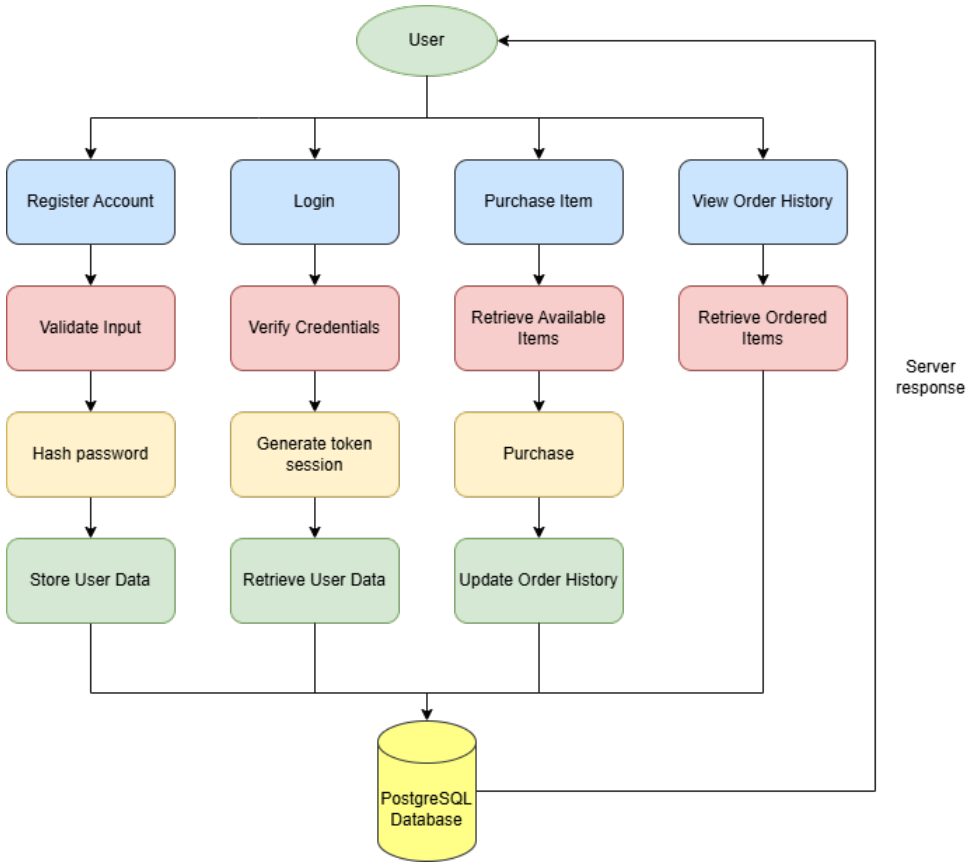

Figure 4.2: Low-Level User Data Flow Diagram.

Level 2 Data Flow Diagram: Low-level interaction between Admins and the database.

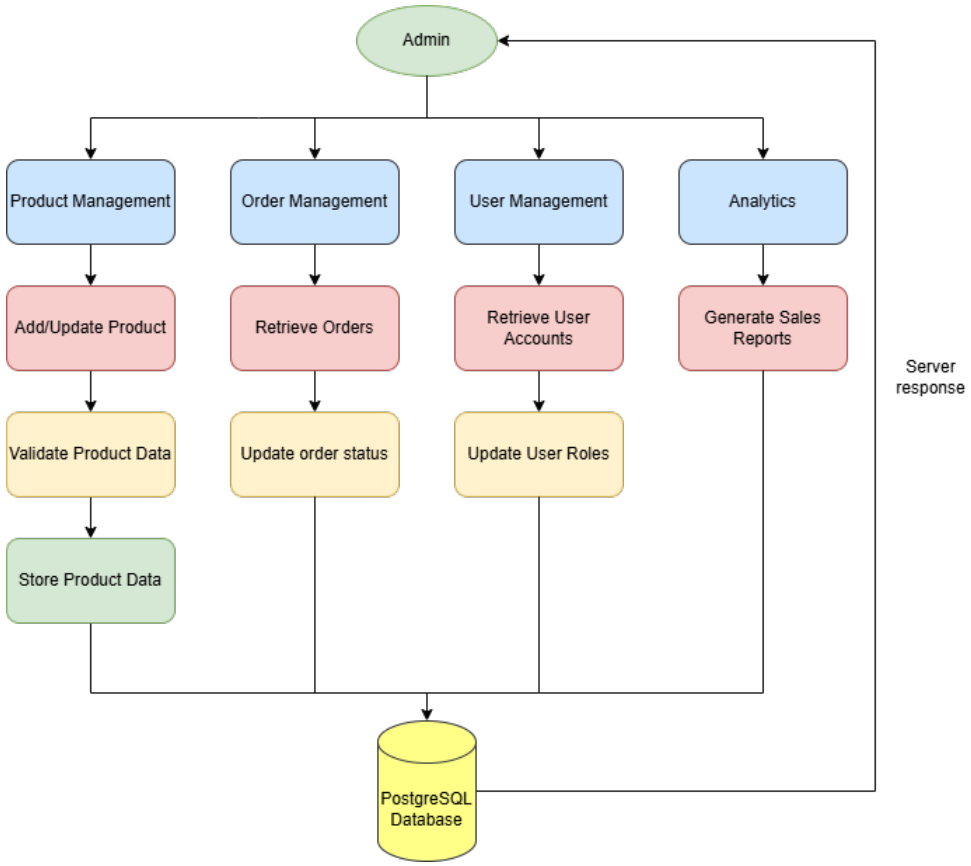

Figure 4.3: Low-Level Admin Data Flow Diagram.

4.3 Data Models

Database Schema:

Table 4.1: Data Models Overview

Model	Fields	Security Features
User	username, password, email, is_admin, is_2FA	Hashed-passwords (PBKDF2). Encryption at rest.
Product	name, description, price, inventory	Validation on updates.
Order	user, status, total_cost	Role-based access.
Cart	amount, name, total_cost, inventory	Validation Checks

Entity Relationship Diagram:

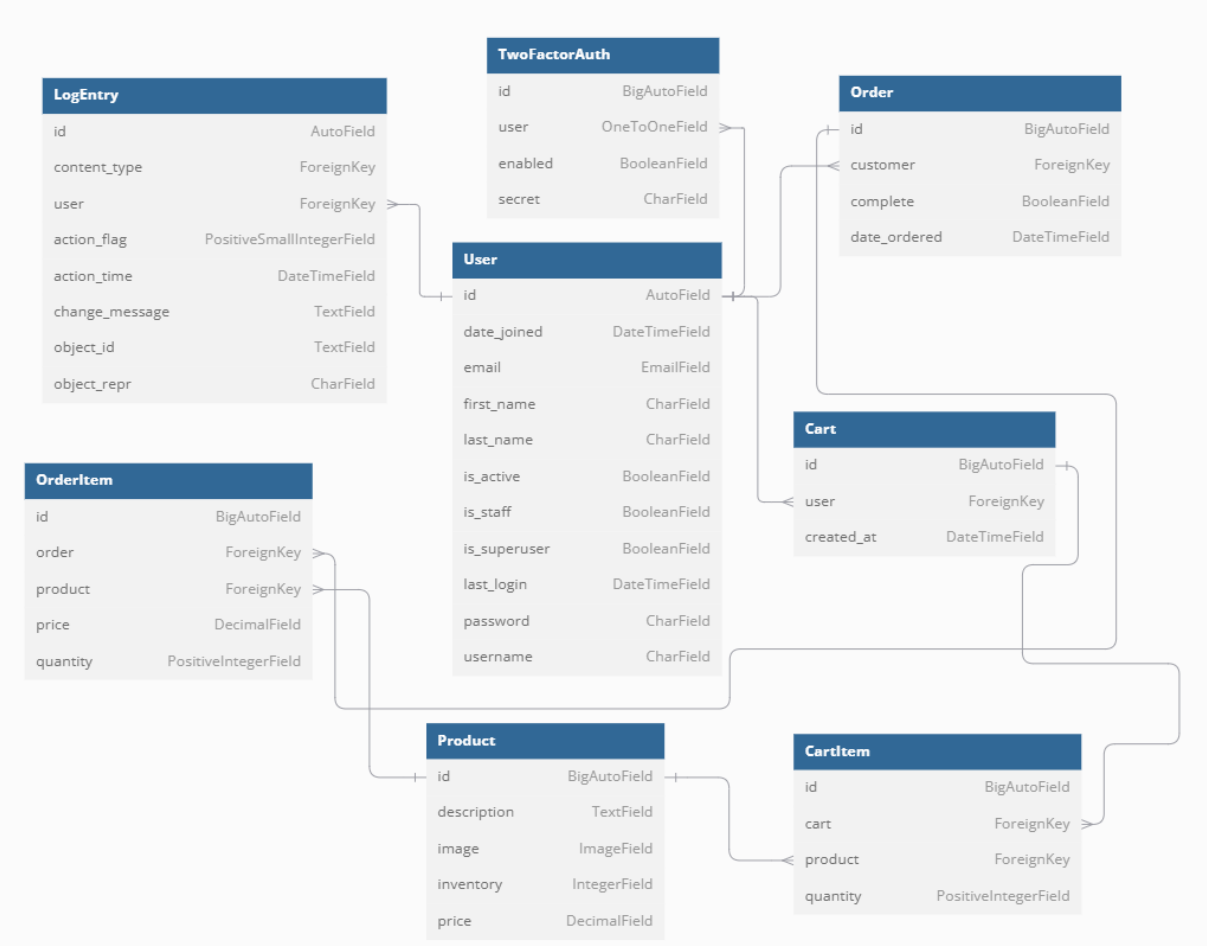

Figure 4.4: Entity Relationship Diagram.

4.4 Sequence Diagrams

In order to break down each part of my system into more digestible sections, I created a number of sequence diagrams that will help guide me through the process of implementing the required functionality [16].

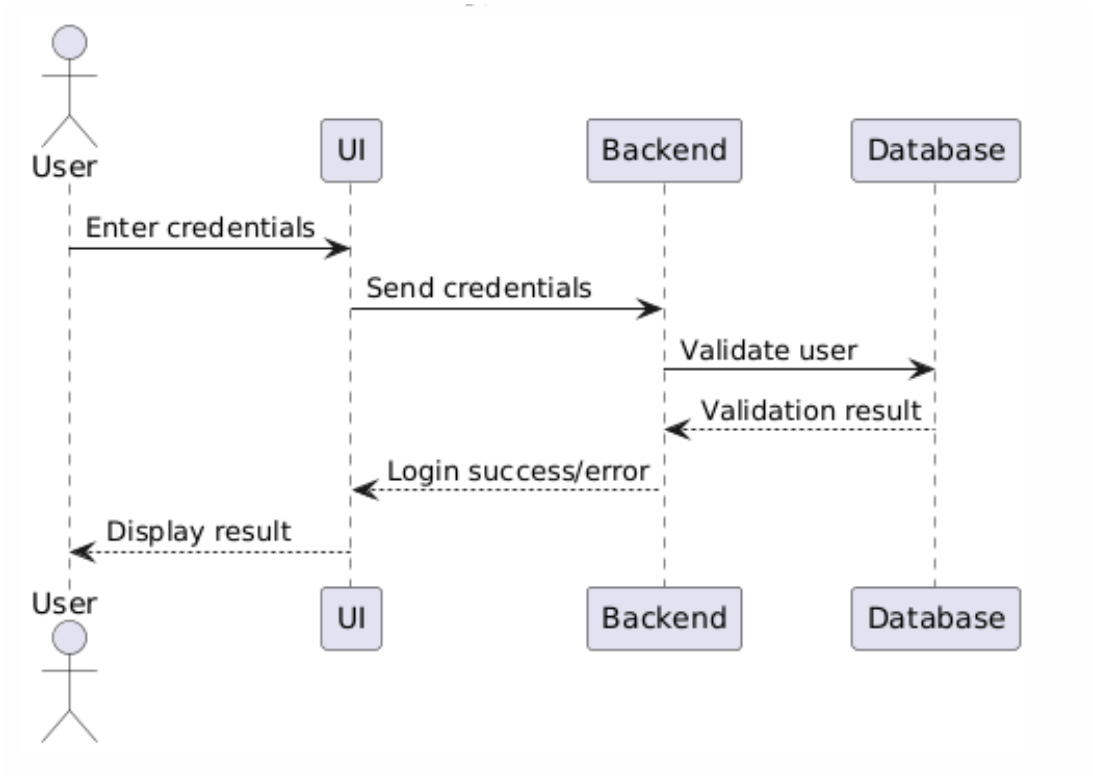

Figure 4.5: User login sequence diagram

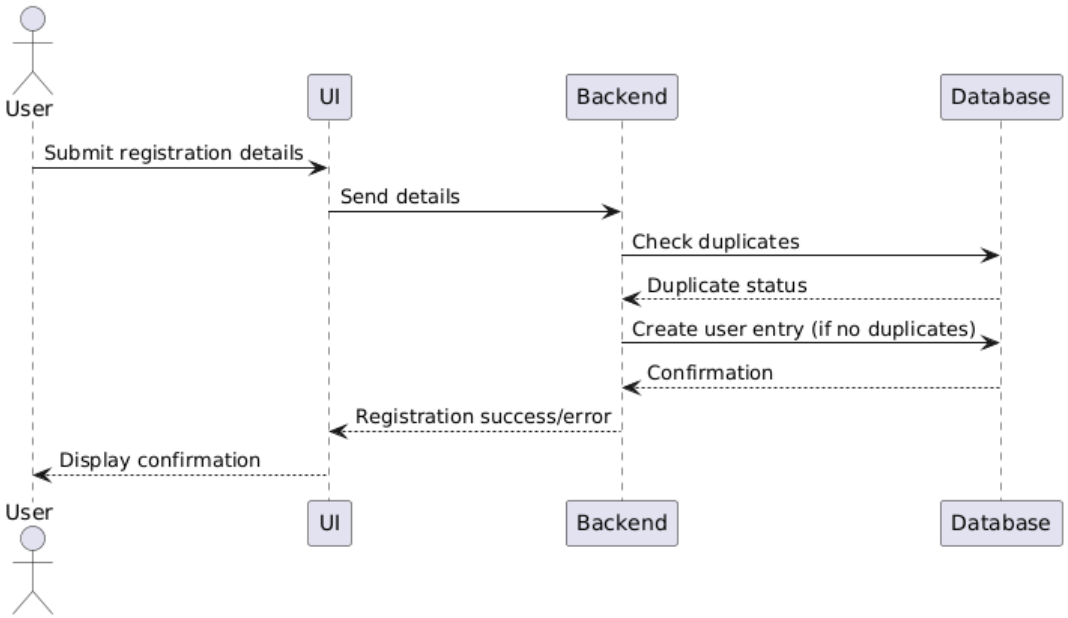

Figure 4.6: User registration sequence diagram

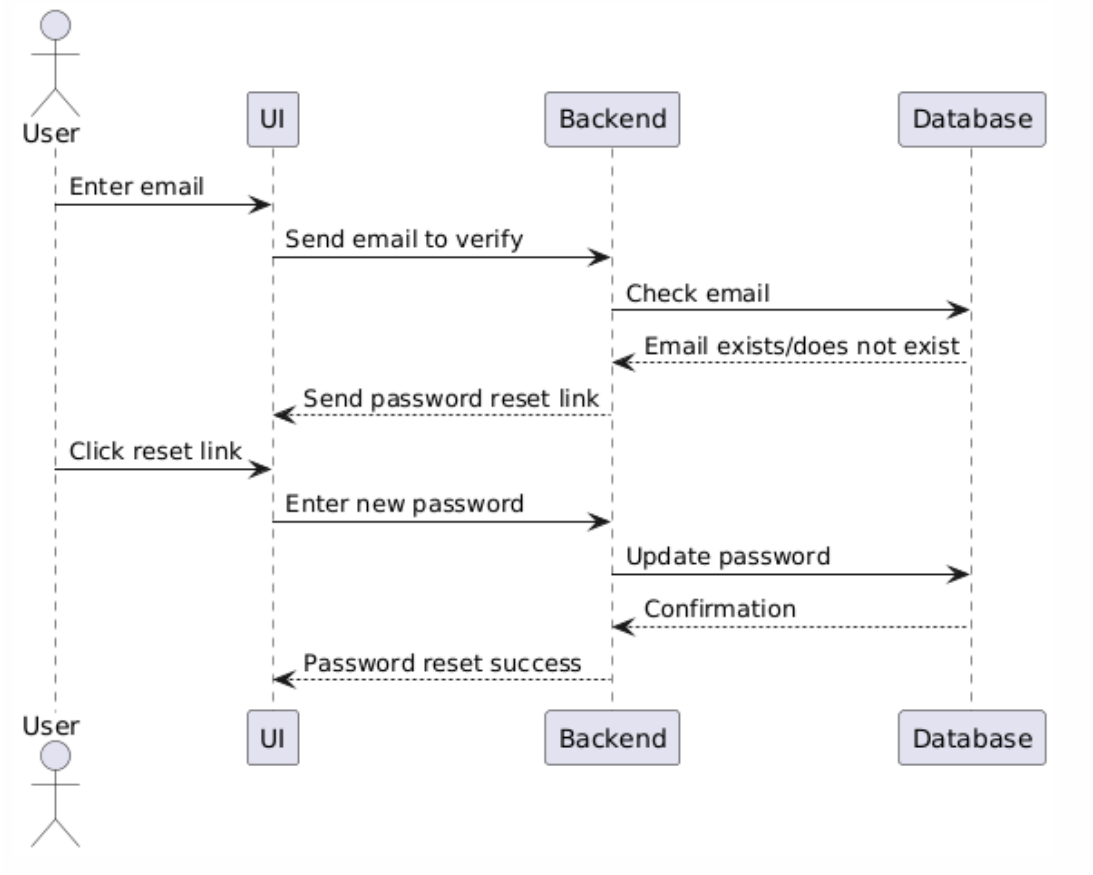

Figure 4.7: Forgot password sequence diagram

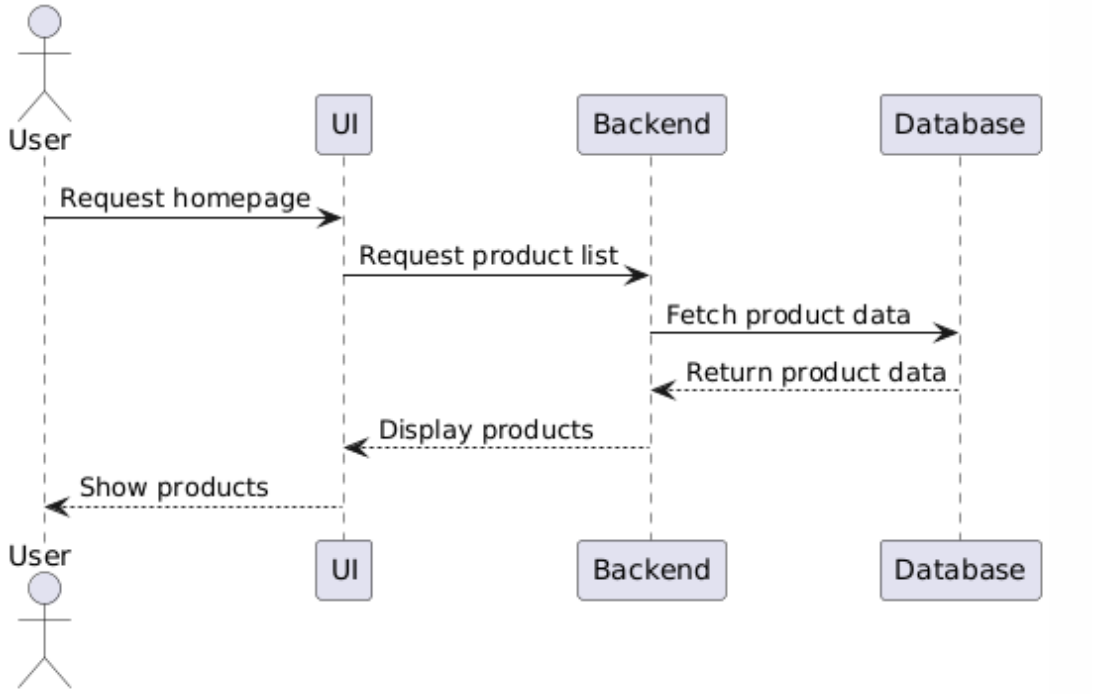

Figure 4.8: Product browsing sequence diagram

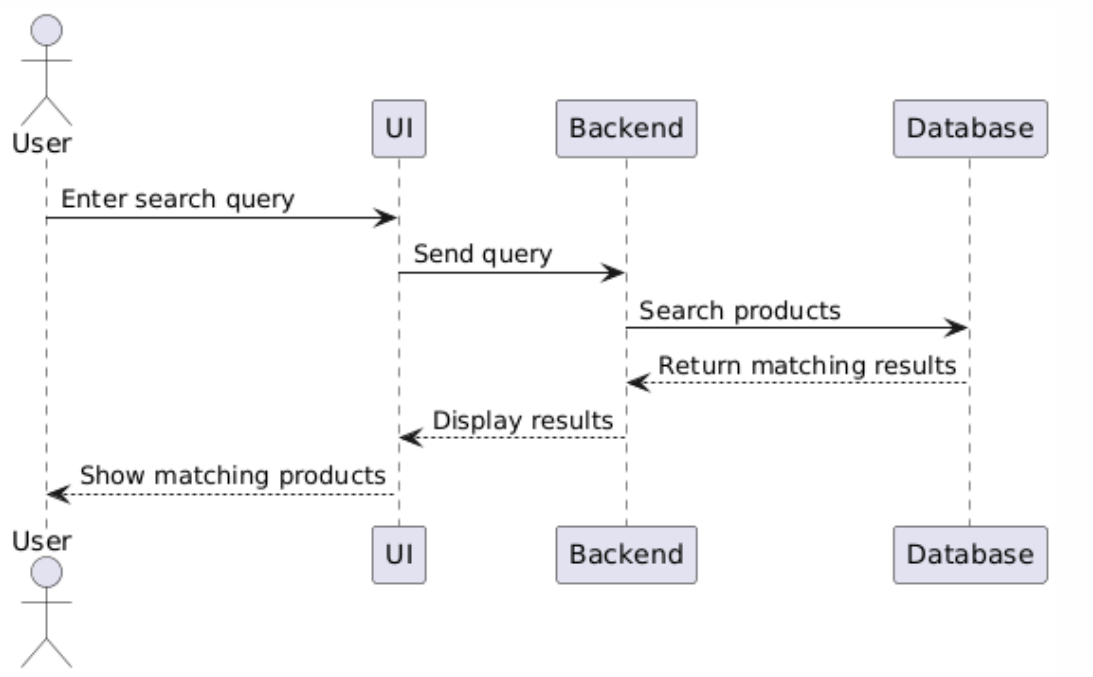

Figure 4.9: Searching products sequence diagram

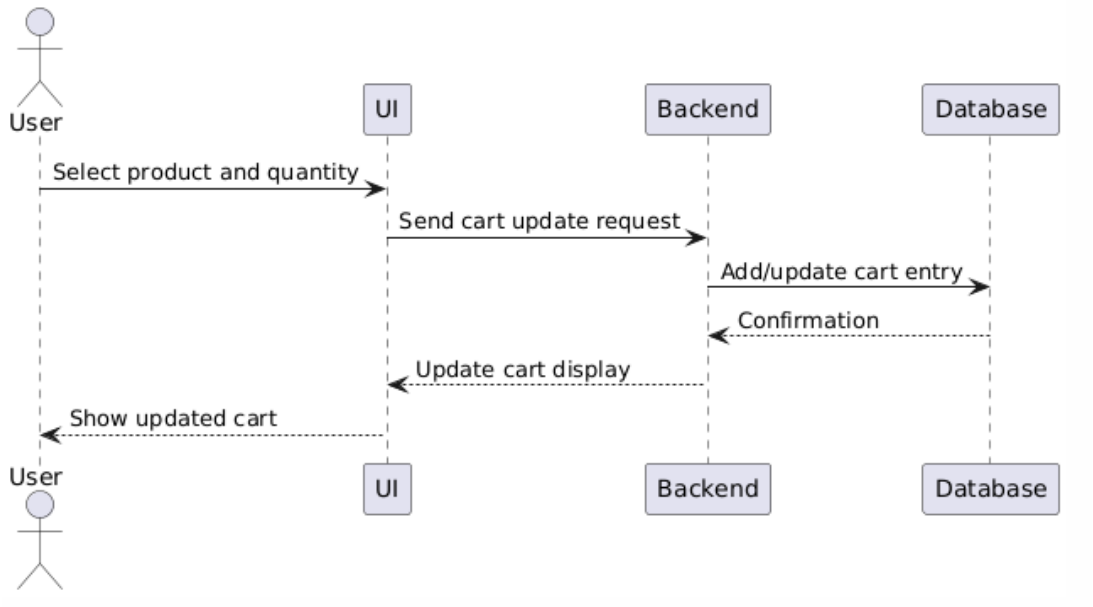

Figure 4.10: Adding items to cart sequence diagram

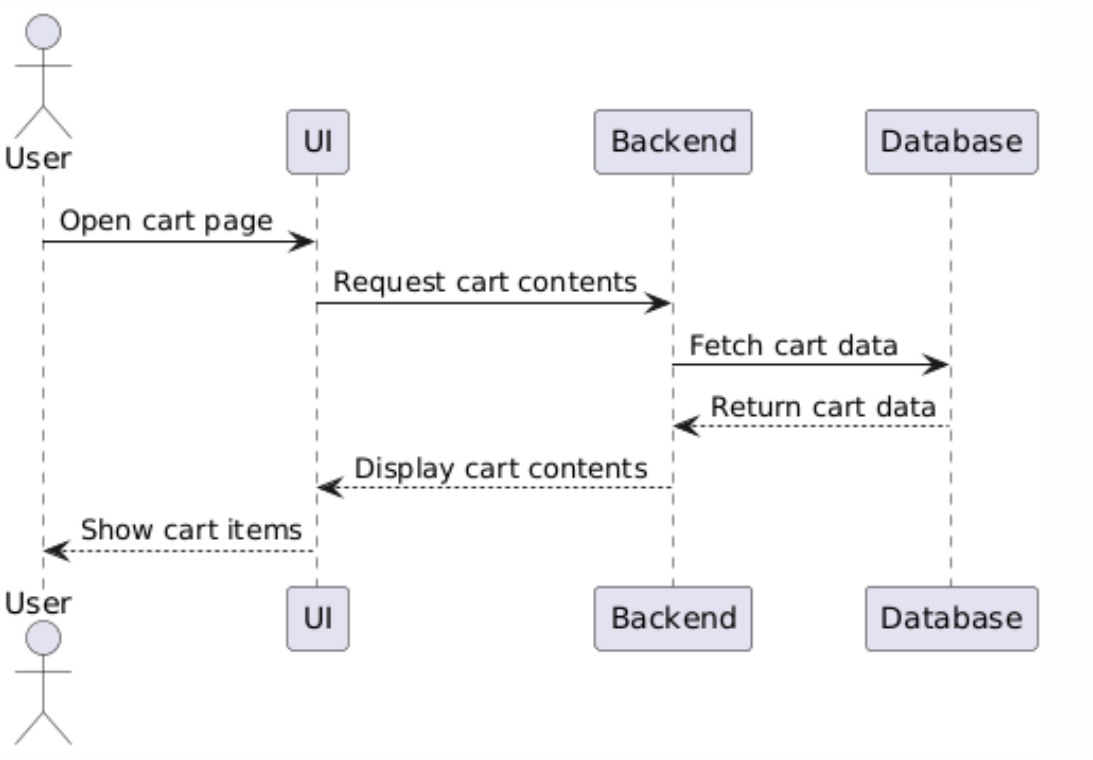

Figure 4.11: Viewing the shopping cart sequence diagram

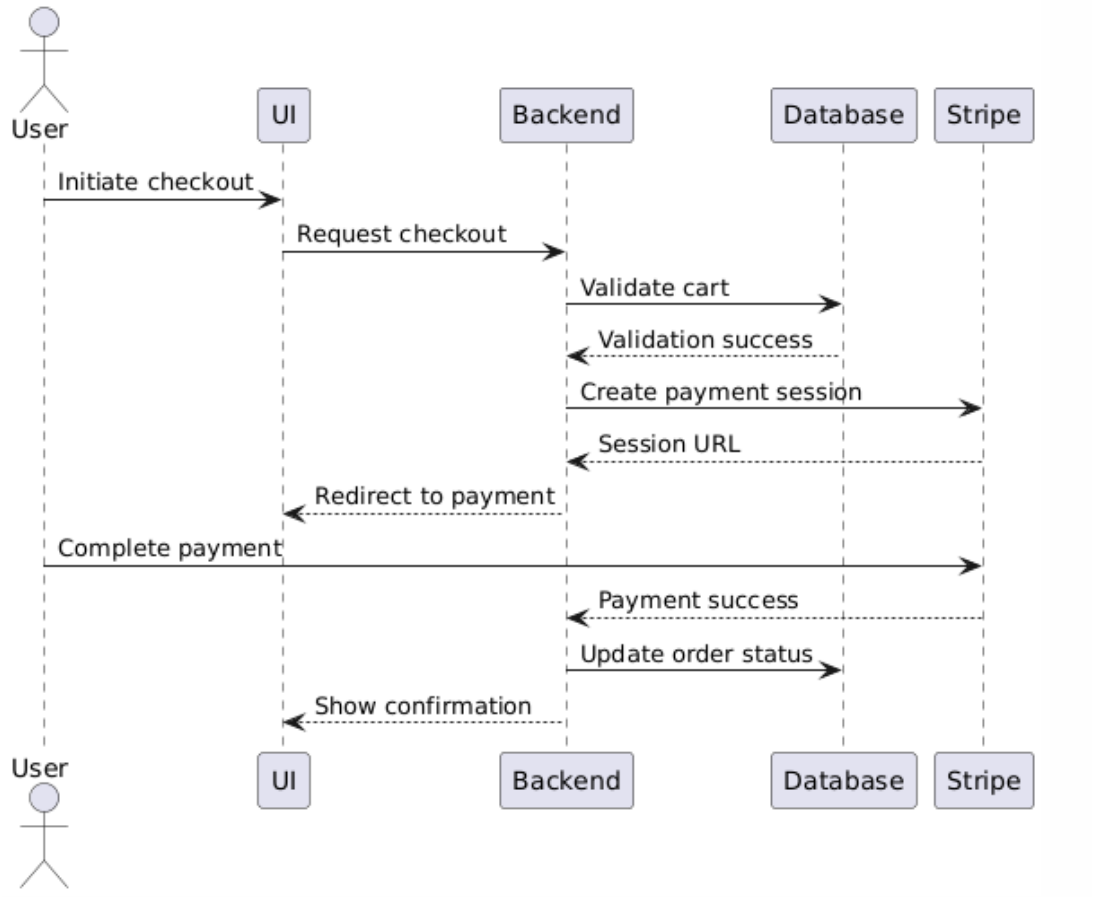

Figure 4.12: Checkout sequence diagram

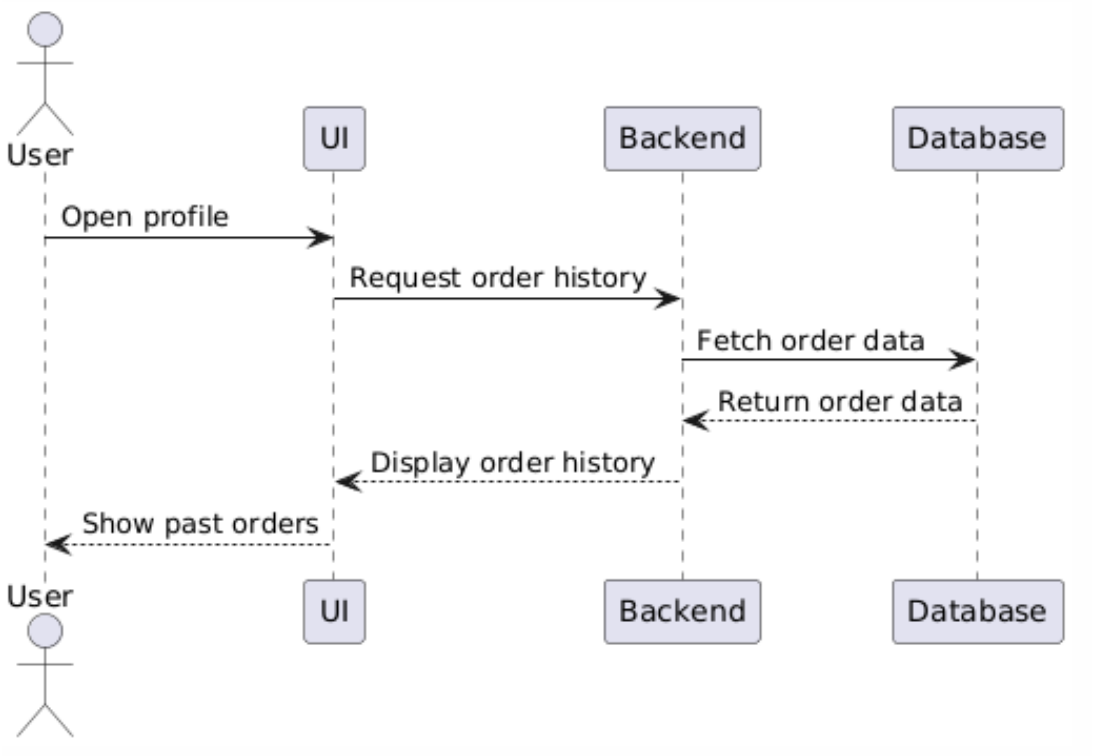

Figure 4.13: Viewing order history sequence diagram

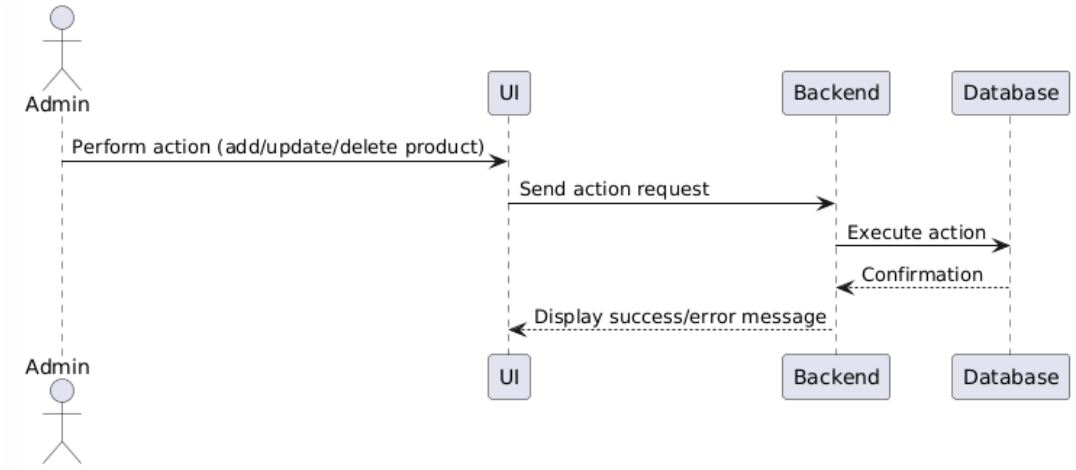

Figure 4.14: Admin managing product sequence diagram

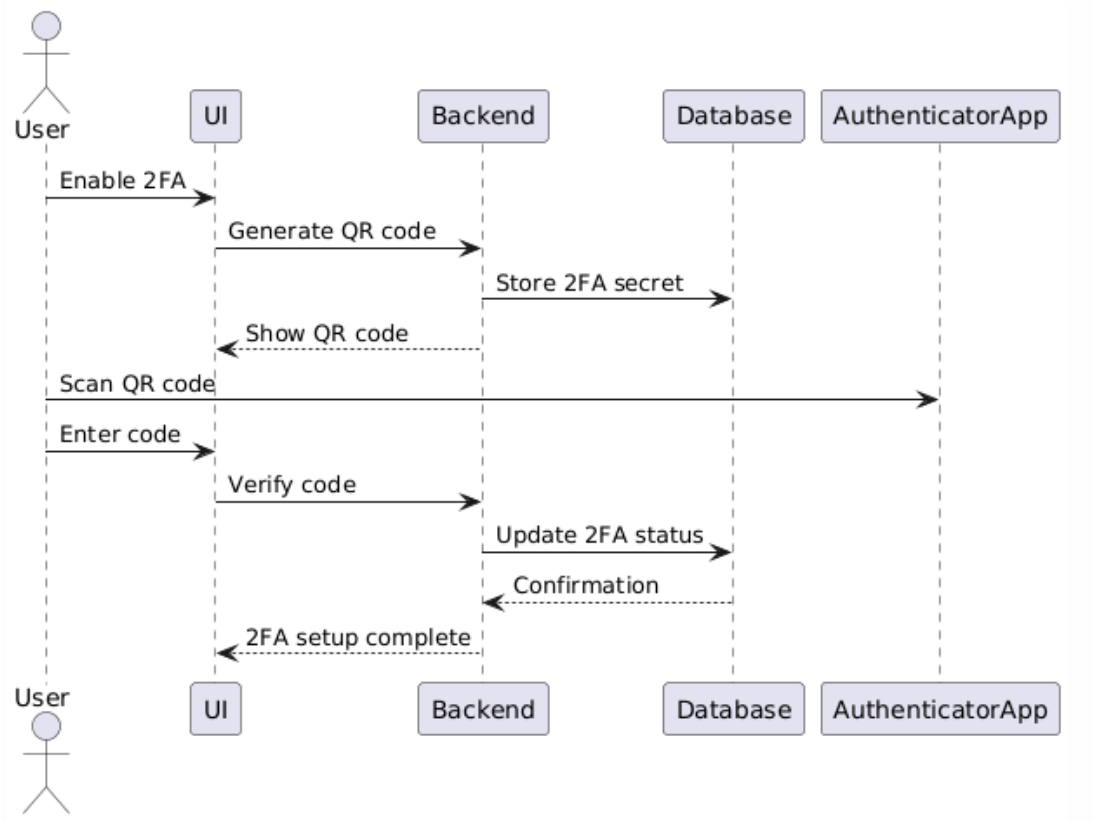

Figure 4.15: Enabling 2FA sequence diagram

4.5 Class diagram

After creating all the sequence diagrams mapping out the core functionality of my program I then moved onto creating a class diagram that highlights the available functions for each class as well as their respective interactions with other classes [17].

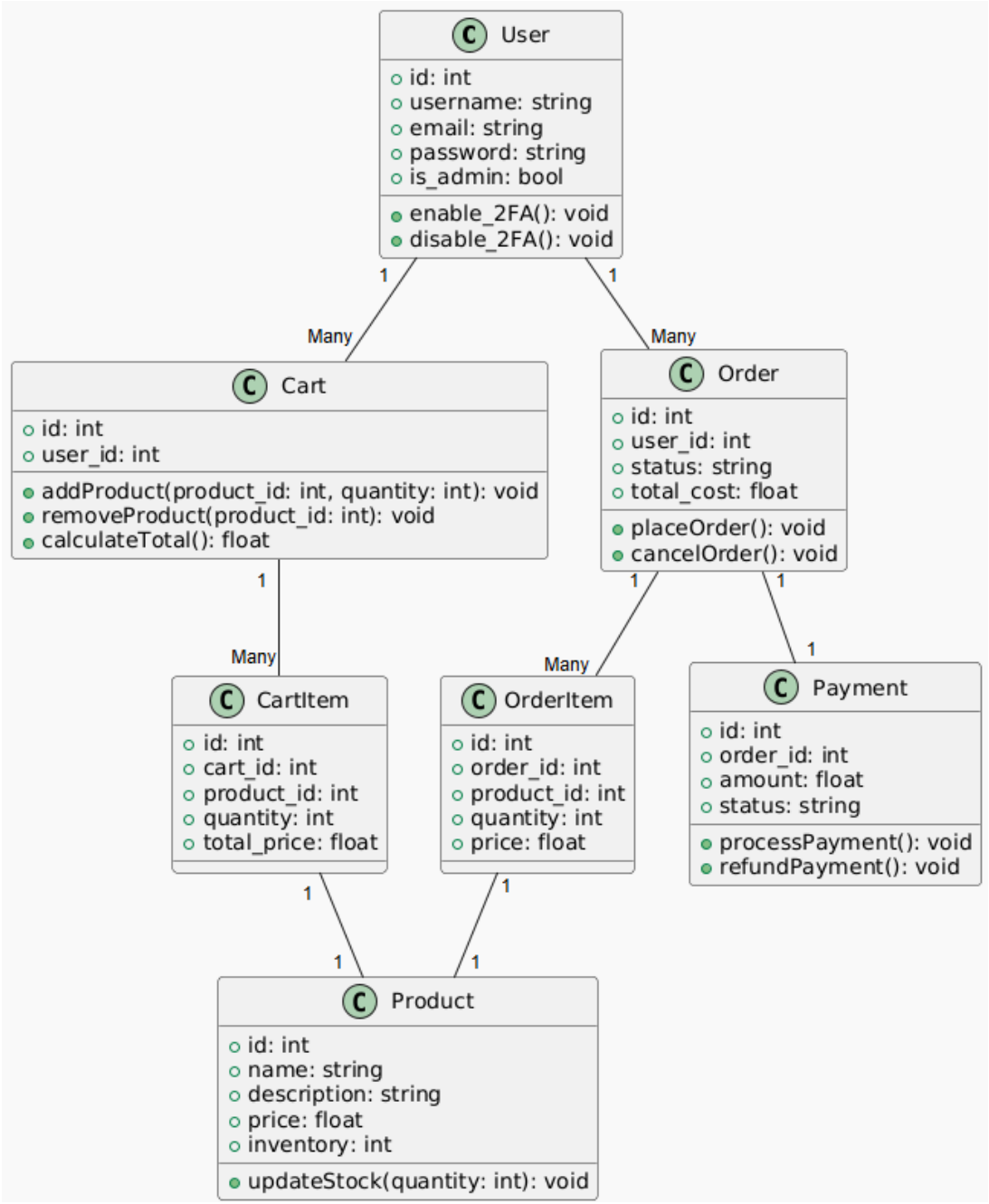

Figure 4.16: A class diagram highlighting all functions

4.6 Secure Design Principles

SecureCart leverages Django and PostgreSQL on AWS to implement robust security measures, ensuring the protection of user data and authentication processes. These technologies, all adhere to the secure design principles that lay out the framework for creating a secure application [18, 19].

Principle	**Implementation Using Django and PostgreSQL on AWS**
Password Hashing	Django hashes passwords using the PBKDF2 algorithm, ensuring secure, irreversible storage. Options like Argon2 or bcrypt can provide even stronger security [20].
Encryption at Rest	PostgreSQL on AWS encrypts all stored data using AES-256, protecting sensitive user and transactional data from unauthorized access [21].
Secure Communication	All communications between the client, server, and database are encrypted with HTTPS and TLS/SSL, ensuring data in transit is secure and protected from interception [22].
Input Validation	Django's ORM and form validation sanitize and validate user inputs to prevent common vulnerabilities, such as SQL injection and XSS [23].
Logging and Monitoring	Django provides detailed logging to track key events like login attempts, while AWS CloudWatch monitors database activity and detects anomalies [24, 25].

Table 4.2: Secure Design Principles and Their Implementation Within SecureCart

Django hashes passwords with PBKDF2, ensuring they are stored securely in the database. PostgreSQL on AWS encrypts data at rest with AES-256, while AWS Key Management Service (KMS) securely handles encryption keys [26]. HTTPS and TLS/SSL ensure secure data transmission, protecting against man-in-the-middle attacks. Django's ORM and form validation prevent common attacks like SQL injection and XSS. Finally, Django's logging and AWS CloudWatch enable monitoring and anomaly detection, ensuring prompt responses to potential threats. These measures collectively enhance platform security and data integrity.

5. Implementation and Coding

The implementation phase involved setting up the development environment, constructing core functionalities, and integrating essential security measures to ensure the robustness and reliability of the *SecureCart* platform. This chapter details each step of the development process, highlighting key decisions and configurations.

5.1 Environment Setup

To begin, a Python virtual environment was established to manage project dependencies effectively. This isolated environment ensures that project-specific packages do not interfere with system-wide installations.

```
C:\Users\stanl\Downloads\newproject>python -m venv venv
```

Figure 5.1: Creating a virtual environment

Within this environment, Django and psycopg2 were installed. Django serves as the primary web framework, while psycopg2 facilitates seamless interaction with PostgreSQL databases.

```
(venv) C:\Users\stanl\Downloads\newproject>pip install django
```

Figure 5.2: Installing Django

```
(venv) C:\Users\stanl\Downloads\newproject>pip install psycopg2-binary
```

Figure 5.3: Installing psycopg2 for PostgreSQL

5.2 Project Initialization

With the environment prepared, the *SecureCart* project was initiated using Django's built-in commands. This initial setup created the foundational structure necessary for further development.

```
(venv) C:\Users\stanl\Downloads\newproject>django-admin startproject securecart
```

Figure 5.4: Creating the SecureCart project

Subsequently, the default Django app, named "store", was created to house the e-commerce functionalities.

```
(venv) C:\Users\stanl\Downloads\newproject\securecart>python manage.py startapp store
```

Figure 5.5: Creating the store app

5.3 Authentication Implementation

Developing secure authentication mechanisms was a priority. Login and registration pages were crafted to allow users to create accounts and access their profiles securely. The corresponding Python files(`settings.py`, `urls.py`, and `views.py`) were updated to handle these authentication processes effectively.

```
{% extends "base.html" %}

{% block content %}

<div class="container mt-5">
    <h2 class="text-center">Login</h2>

    {% if form.errors %}
        <div class="alert alert-danger">
            {% for field in form %}
                {% for error in field.errors %}
                    <p>{{ error }}</p>
                {% endfor %}
            {% endfor %}
            {% for error in form.non_field_errors %}
                <p>{{ error }}</p>
            {% endfor %}
        </div>
    {% endif %}

    <div class="card mx-auto" style="max-width: 400px;">
        <div class="card-body">
            <form method="post" action="{% url 'login' %}">
                {% csrf_token %}
                <div class="form-group">
                    <label for="username">Username</label>
                    <input type="text" name="username" id="username" class="form-control" placeholder="Enter username" required>
                </div>
                <div class="form-group mt-3">
                    <label for="password">Password</label>
                    <input type="password" name="password" id="password" class="form-control" placeholder="Enter password" required>
                </div>
                <button type="submit" class="btn btn-primary w-100 mt-4">Login</button>
            </form>

        </div>
    </div>
</div>
{% endblock %}
```

Figure 5.6: Login.html Template

```
<!-- store/templates/store/register.html -->
{% extends 'base.html' %}

{% block title %}Register - SecureCart{% endblock %}

{% block content %}
<div class="row justify-content-center">
    <div class="col-md-6">
        <div class="card shadow-sm">
            <div class="card-body">
                <h2 class="card-title text-center mb-4">Register</h2>
                <form method="post">
                    {% csrf_token %}
                    {{ form.as_p }}
                    <div class="d-grid gap-2">
                        <button type="submit" class="btn btn-primary">Register</button>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Figure 5.7: Register.html Template

The `settings.py` file was configured to redirect users appropriately upon accessing the webpage.

```
LOGIN_REDIRECT_URL = '/'  # Redirect here after login
LOGOUT_REDIRECT_URL = 'login'  # Redirect to login after logout
LOGIN_URL = 'login'  # Where unauthenticated users are redirected
```

Figure 5.8: Updated settings.py for redirections

Similarly, `urls.py` was modified to define the URL patterns for the login and registration pages.

```
from django.contrib import admin
from django.urls import path, include
from django.shortcuts import redirect

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),  # Includes login/logout/password views
    path('', lambda request: redirect('/accounts/login/')),  # redirect to built-in login URL
    path('register/', include('store.urls')),
]
```

Figure 5.9: Updated urls.py for page redirections

In `views.py`, views were established to render the registration page and handle user input.

```
from django.shortcuts import render, redirect
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import login
from django.contrib import messages

def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)  # Log the user in after registration
            messages.success(request, "Registration successful!")
            return redirect('login')  # Redirect to login or another page
    else:
        form = UserCreationForm()
    return render(request, 'store/register.html', {'form': form})
```

Figure 5.10: Updated views.py for register view

Following these updates, the login and registration pages were operational, as illustrated below:

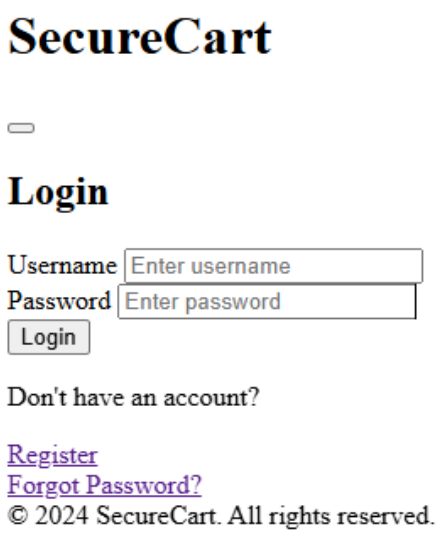

Figure 5.11: Login page

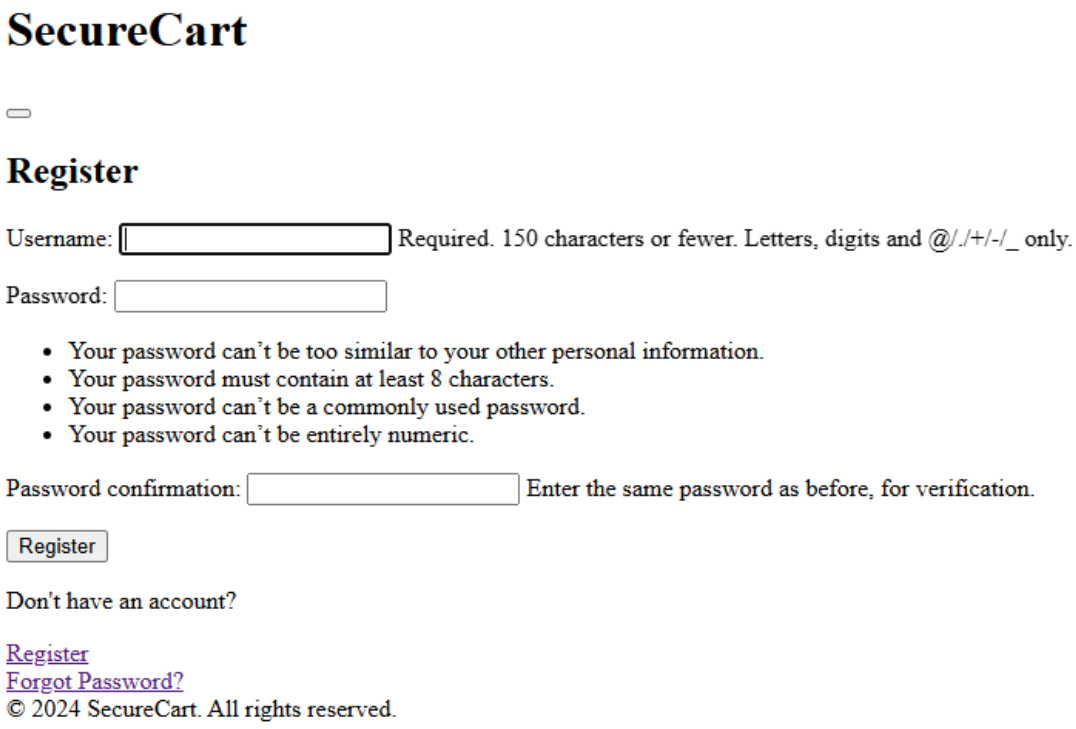

Figure 5.12: Registration page

5.4 Database Integration

To ensure that user authentication was securely linked to the backend, the login functionality was connected to a PostgreSQL database hosted on AWS. This integration was achieved by updating the `settings.py` file with the appropriate database configurations.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config('DB_NAME'),  #  DB name
        'USER': config('DB_USER'),  #  DB user
        'PASSWORD': config('DB_PASSWORD'),  #  password
        'HOST': config('DB_HOST'),  #  host
        'PORT': config('DB_PORT'),  #  port
        'OPTIONS': {
            'sslmode': 'require',
            'sslrootcert': os.path.join(BASE_DIR, 'global-bundle.pem'), # encyption key
        },
    }
}
```

Figure 5.13: Connecting to PostgreSQL with encryption

Sensitive credentials, such as database usernames and passwords, were stored as environment variables to enhance security and prevent exposure in the codebase.

```
DB_NAME='defaultdb'
DB_USER='avnadmin'
DB_PASSWORD='AVNS_7FCMyIg0-3bB94yaTPh'
DB_HOST='pg-38c9f943-shawstan96-44a1.c.aivencloud.com'
DB_PORT=18833
```

Figure 5.14: Storing sensitive data in environment variables

5.5 Password Recovery

Implementing a password recovery feature was essential for user convenience and security. Django's built-in functionality was utilised, requiring configuration of SMTP settings to enable email transmissions.

```
EMAIL_PASSWORD='rynl ftkc urwb onxg'
EMAIL='securecart.pw@gmail.com'
```

Figure 5.15: Storing email credentials in environment variables

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_HOST_USER = config('EMAIL')
EMAIL_HOST_PASSWORD = config('EMAIL_PASSWORD')
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
```

Figure 5.16: SMTP configuration in settings.py

This setup allowed users to reset their passwords seamlessly, as demonstrated below:

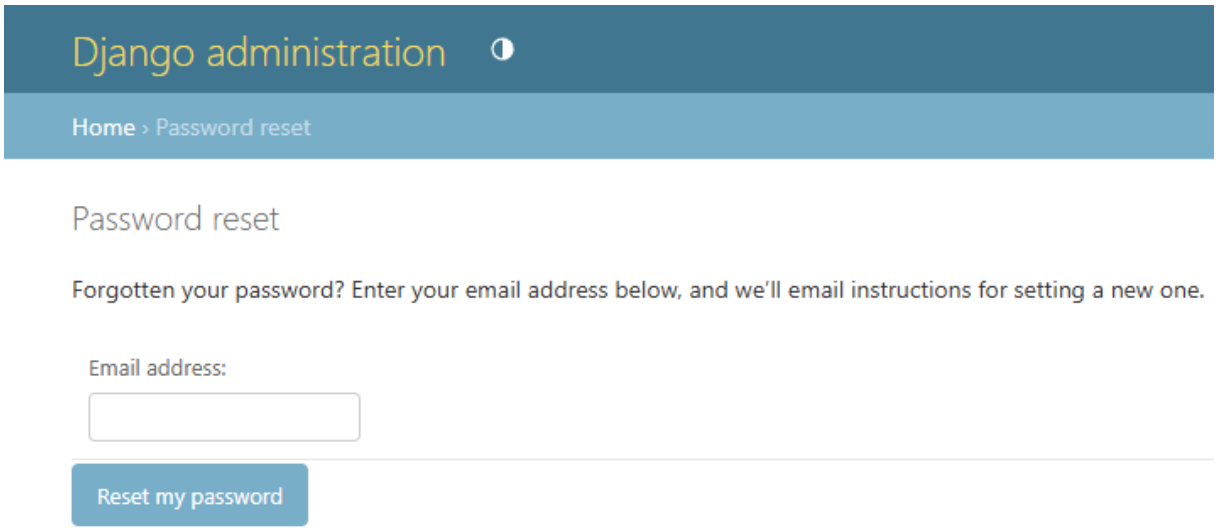

Figure 5.17: Forgot password page

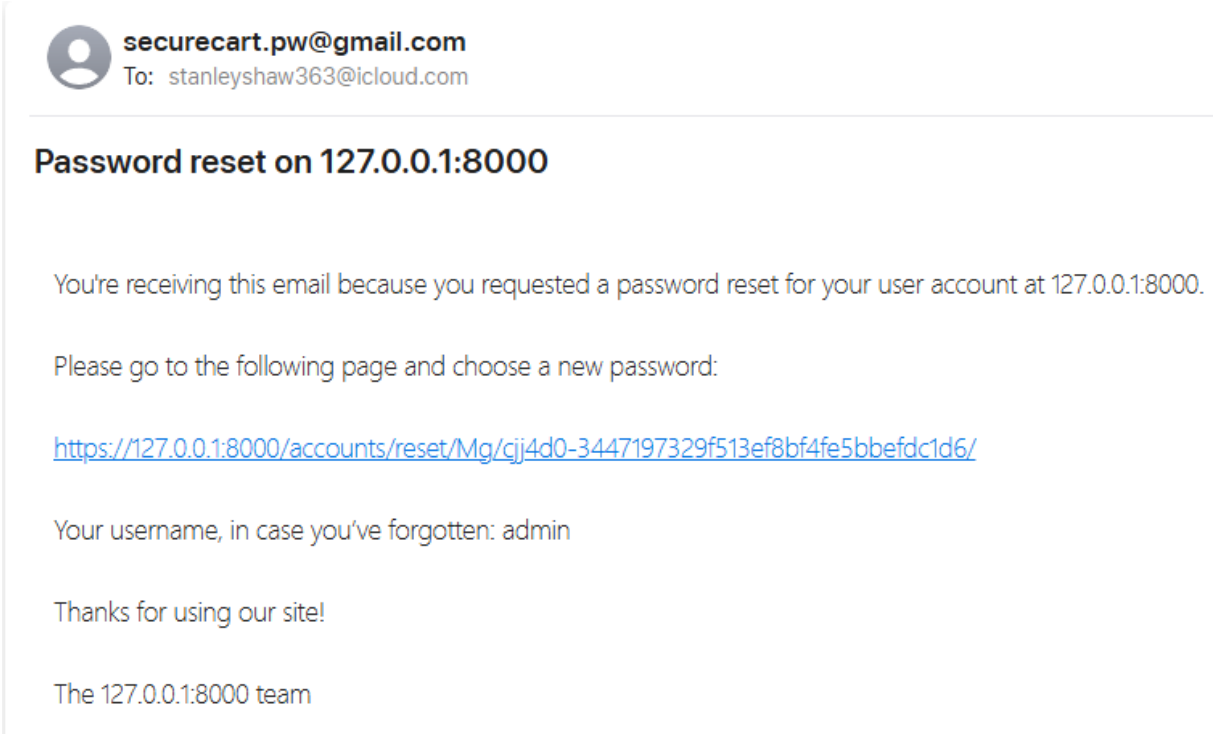

Figure 5.18: Password reset email

5.6 Homepage Development

The homepage serves as the central hub for users to browse products. A dedicated template was first created, and product models were developed to store relevant information such as name, description, price, and inventory levels.

```
<!-- store/templates/store/home.html -->

{% extends 'base.html' %}

{% block title %}SecureCart - Home{% endblock %}

{% block content %}

{% if user.is_authenticated %}
    <!-- Products Grid -->
    <div class="row mt-4">
        {% if products %}
        {% for product in products %}
            <div class="col-lg-3 col-md-4 col-sm-6 mb-4">
                <div class="card h-100">
                    <div class="card-img-top" style="height: 200px;"> <!-- Set a fixed height for uniform image size -->
                        {% if product.image %}
                            <img src="{{ product.image.url }}" class="product-image" alt="{{ product.name }}">
                        {% else %}

                        {% endif %}
                    </div>
                    <div class="card-body d-flex flex-column">
                        <h5 class="card-title">{{ product.name }}</h5>
                        <p class="card-text">{{ product.description|truncatewords:15 }}</p>
                        <p class="card-text"><strong>Price:</strong> ${{ product.price }}</p>
                        <a href="{% url 'product_detail' product.id %}" class="btn btn-primary mt-auto">View Details</a>
                    </div>
                </div>
            </div>


        {% endfor %}
        {% else %}
        <p>No products available at the moment.</p>
        {% endif %}
    </div>
{% else %}

{% endif %}
{% endblock %}
```

Figure 5.19: Home.html template

```
class Product(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    inventory = models.IntegerField()
    image = models.ImageField(upload_to='product_images/', null=True, blank=True)

    def __str__(self):
        return self.name
```

Figure 5.20: Product model

To ensure the homepage loaded correctly post-login, `urls.py` and `views.py` were updated accordingly.

```python
# securecart/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('store.urls')),  # Include your app's URLs
    path('register/', views.register, name='register'),
    path('', views.home, name='home'), # Home page URL
    path('login/', views.custom_login, name='login'),  # Use custom login view

]
```

Figure 5.21: Updated urls.py for homepage

```python
@login_required(login_url='/accounts/login/')
def home(request):
    products = Product.objects.filter(inventory__gt=0)  # Only display products in stock
    context = {'products': products}
    return render(request, 'store/home.html', context)
```

Figure 5.22: Updated views.py for homepage

The updated homepage, featuring example products, is shown below:

Figure 5.23: Homepage with product example

5.7 Navigation Bar

A navigation bar was developed to facilitate easy movement between key sections of the website, including the cart, profile, admin dashboard, and logout functionalities.

```
<!-- Navigation Bar -->
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <div class="container">

        <!-- Toggler/collapsible Button for Mobile -->
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav"
            aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <!-- Navbar Links -->
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav ms-auto">
                {% if user.is_authenticated %}
                    <li class="nav-item">
                        <a class="nav-link" href="/">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url 'cart_detail' %}">Cart</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url 'profile' %}">Profile</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url 'admin:index' %}">Admin</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url 'logout' %}">Logout</a>
                    </li>


                    <form class="d-flex ms-3" method="get" action="{% url 'product_search' %}">
                        <input class="form-control me-2" type="search" placeholder="Search" name="q" aria-label="Search">
                        <button class="btn btn-outline-success" type="submit">Search</button>
                    </form>
                {% else %}

                    {% endif %}
            </ul>

        </div>

    </div>
</nav>
```

Figure 5.24: Navigation bar HTML

Additionally the Logout functionality was implemented using Django's built-in features, ensuring users could securely exit their accounts.

```
path('logout/', views.custom_logout, name='logout'),
```

Figure 5.25: Updated urls.py for logout

5.8 Search Functionality

To enhance user experience, a search bar was implemented, allowing users to filter products based on specific criteria.

```
{% extends 'base.html' %}
{% csrf_token %}
{% block content %}
<h1>Search Results for "{{ query }}"</h1>
{% if products %}
<div class="row mt-4">
    {% for product in products %}
    <div class="col-lg-3 col-md-4 col-sm-6 mb-4">
        <div class="card h-100">
            <div class="card-img-top" style="height: 200px;"> <!-- Set a fixed height for uniform image size -->
                {% if product.image %}
                    <img src="{{ product.image.url }}" class="product-image" alt="{{ product.name }}">
                {% else %}

                {% endif %}
            </div>
            <div class="card-body d-flex flex-column">
                <h5 class="card-title">{{ product.name }}</h5>
                <p class="card-text">{{ product.description|truncatewords:15 }}</p>
                <p class="card-text"><strong>Price:</strong> ${{ product.price }}</p>
                <a href="{% url 'product_detail' product.id %}" class="btn btn-primary mt-auto">View Details</a>
            </div>
        </div>
    </div>
    {% endfor %}
</div>
{% else %}
<p>No products found.</p>
{% endif %}
{% endblock %}
```

Figure 5.26: Product_search.html template

```
def product_search(request):
    query = request.GET.get('q')
    products = Product.objects.filter(name__icontains=query)
    return render(request, 'store/product_search.html', {'products': products, 'query': query})
```

Figure 5.27: Product search view in views.py

The functioning search capability is depicted below:

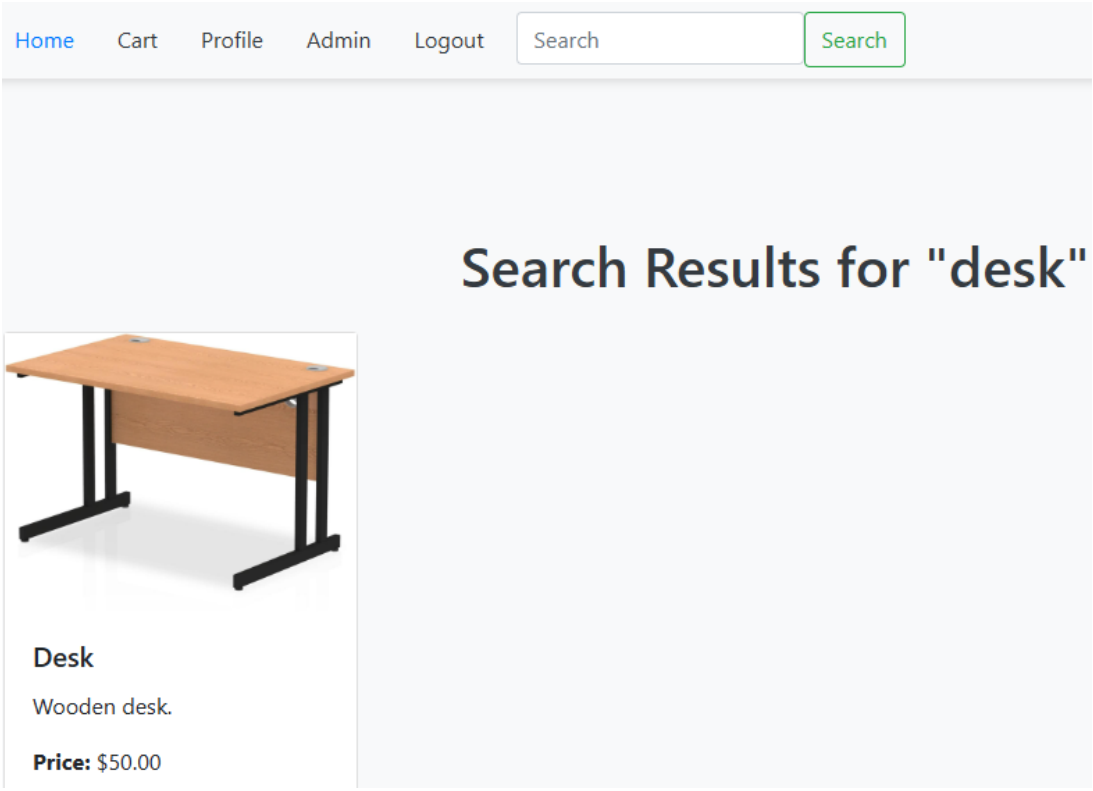

Figure 5.28: Working search functionality

5.9 Shopping Cart and Checkout

The shopping cart is a pivotal feature, enabling users to select and manage products before finalising purchases. The cart template and associated models were developed to handle these functionalities.

```
{% extends 'base.html' %}
{% load store_extras %}
{% block content %}
{% csrf_token %}
<h1>Your Shopping Cart</h1>
{% if cart.items.all %}
<table class="table">
    <thead>
        <tr>
            <th>Product</th>
            <th>Quantity</th>
            <th>Price</th>
            <th>Total</th>
        </tr>
    </thead>
    <tbody>
        {% for item in cart.items.all %}
        <tr>
            <td>{{ item.product.name }}</td>
            <td>{{ item.quantity }}</td>
            <td>${{ item.product.price }}</td>
            <td>${{ item.product.price|mul:item.quantity }}</td>
        </tr>
        {% endfor %}
    </tbody>
</table>
<p><strong>Grand Total: ${{ cart.get_total_price }}</strong></p>
<div class="d-inline-flex">
    <a href="{% url 'checkout' %}" class="btn btn-success" style="margin-right: 20px;">Proceed to Checkout</a>
    <a href="{% url 'home' %}" class="btn btn-primary">Continue Shopping</a>
</div>


{% else %}
<p>Your cart is empty.</p>
{% endif %}
{% endblock %}
```

Figure 5.29: Cart_detail.html template

```
path('cart/', views.cart_detail, name='cart_detail'),
```

Figure 5.30: Cart URL configuration

```
@login_required
def cart_detail(request):
    cart, created = Cart.objects.get_or_create(user=request.user)
    ip = get_client_ip(request)
    logger.info(f"Cart viewed - User: {request.user.username}, IP: {ip}")
    return render(request, 'store/cart_detail.html', {'cart': cart})
```

Figure 5.31: Cart detail view in views.py

Cart and CartItem models were established to track user selections effectively.

```python
class Cart(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)

    def get_total_price(self):
        return sum(item.get_total_price() for item in self.items.all())

class CartItem(models.Model):
    cart = models.ForeignKey(Cart, on_delete=models.CASCADE, related_name='items')
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(default=1)

    def get_total_price(self):
        return self.product.price * self.quantity
```

Figure 5.32: Cart and CartItem models

The functionality to add items to the cart was implemented as follows:

```python
@login_required
def add_to_cart(request, pk):
    product = get_object_or_404(Product, pk=pk)
    cart, created = Cart.objects.get_or_create(user=request.user)
    quantity = int(request.POST.get('quantity', 1))  # Get quantity from the form, default to 1 if not specified

    # Check if requested quantity exceeds available inventory
    if quantity > product.inventory:
        messages.error(request, f"Only {product.inventory} units of {product.name} are available.")
        return redirect('product_detail', pk=pk)

    # Get or create a cart item for the specified product
    cart_item, created = CartItem.objects.get_or_create(cart=cart, product=product)

    # If the cart item already exists, increase the quantity, but cap it to the available inventory
    if not created:
        cart_item.quantity += quantity
        if cart_item.quantity > product.inventory:
            cart_item.quantity = product.inventory
            messages.warning(request, f"Quantity adjusted to available stock for {product.name}.")
    else:
        cart_item.quantity = quantity

    cart_item.save()

    messages.success(request, f"{quantity} units of {product.name} added to cart.")
    return redirect('cart_detail')
```

Figure 5.33: Add to cart functionality

Subsequent development focused on the checkout process and order confirmation. Order and OrderItem models were created to manage transactions, and corresponding templates were developed to facilitate user confirmations.

```python
class OrderItem(models.Model):
    order = models.ForeignKey(Order, on_delete=models.CASCADE, related_name='items')
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def get_total_price(self):
        return self.price * self.quantity

    def __str__(self):
        return f"{self.quantity} x {self.product.name} in Order {self.order.id}"

class Order(models.Model):
    customer = models.ForeignKey(User, on_delete=models.CASCADE)
    date_ordered = models.DateTimeField(auto_now_add=True)
    complete = models.BooleanField(default=False)


    def __str__(self):
        return f"Order {self.id} by {self.customer.username}"

    def get_total(self):
        return sum(item.get_total_price() for item in self.items.all())
```

Figure 5.34: Order and OrderItem models

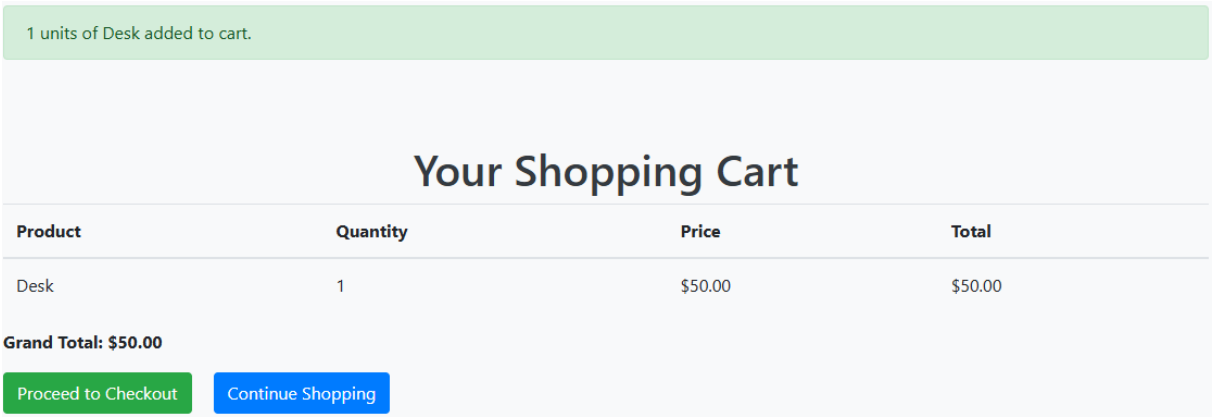

Figure 5.35: Shopping cart page

```
{% extends 'base.html' %}
{% csrf_token %}
{% block content %}
<h1>Checkout</h1>
<form method="post">
    {% csrf_token %}
    <p>Total Amount: ${{ cart.get_total_price }}</p>
    <button type="submit" class="btn btn-primary">Confirm Order</button>
</form>
{% endblock %}
```

Figure 5.36: Checkout.html template

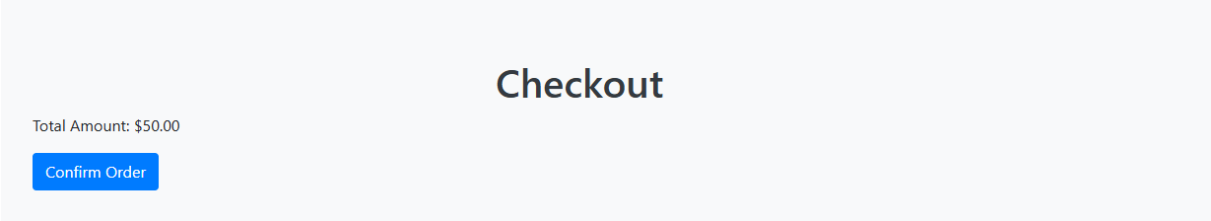

Figure 5.37: Working checkout page

```
{% extends 'base.html' %}
{% csrf_token %}
{% block content %}
<h1>Order Confirmation</h1>
<p>Thank you for your purchase, {{ user.username }}!</p>
<p>Your order number is {{ order.pk }}.</p>
{% endblock %}
```

Figure 5.38: Order_confirmation.html template

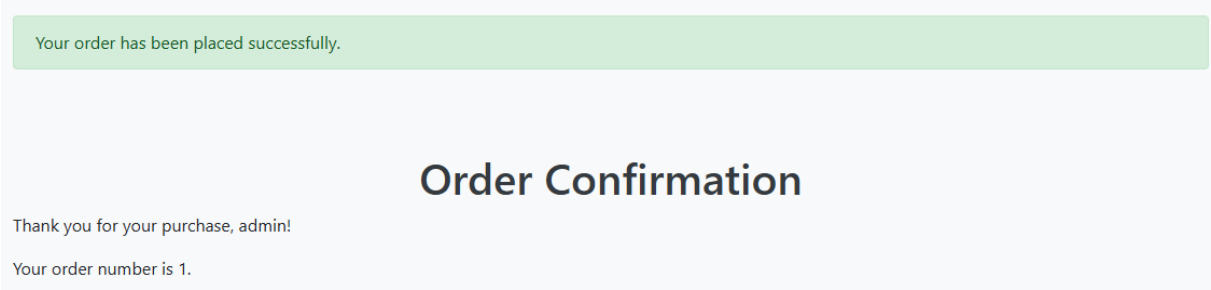

Figure 5.39: Successful order page

5.10 Profile Page

The profile page was developed to allow users to view and manage their personal information and order history. This involved updating URL configurations and creating dedicated templates.

```
path('accounts/profile/', views.profile, name='profile'),  # Profile page
```

Figure 5.40: Updated urls.py for profile page

```
<!-- store/templates/store/profile.html -->

{% extends 'base.html' %}
{% load static %}
{% block content %}
<div class="container mt-5">
 <!-- User Information Card -->
    <div class="card mb-1">
        <div class="card-header bg-primary text-white">
            <h2>Account Information</h2>
        </div>
        <div class="card-body">
            <p><strong>Username:</strong> {{ user.username }}</p>
            <p><strong>Email:</strong> {{ user.email }}</p>
        </div>
    </div>

    <!-- Orders Section -->
    <div class="card">
        <div class="card-header bg-info text-white">
            <h2>Your Orders</h2>
        </div>
        <div class="card-body">
            {% if orders %}
                <div class="table-responsive">
                    <table class="table table-striped table-hover align-middle">
                        <thead class="table-dark">
                            <tr>
                                <th scope="col">Order ID</th>
                                <th scope="col">Date Placed</th>
                                <th scope="col">Status</th>
                                <th scope="col">Total Amount</th>
                                <th scope="col">Details</th>
                            </tr>
                        </thead>
                        <tbody>
                            {% for order in orders %}
                                <tr>
                                    <td>{{ order.id }}</td>
                                    <td>{{ order.date_ordered|date:"F j, Y, g:i a" }}</td>
                                    <td>
                                        {% if order.complete %}
                                            <span class="badge bg-success">Completed</span>
                                        {% else %}
                                            <span class="badge bg-warning text-dark">Pending</span>
                                        {% endif %}
                                    </td>
                                    <td>${{ order.get_total }}</td>
                                    <td>
                                        <a href="{% url 'order_confirmation' order.id %}" class="btn btn-info btn-sm">View</a>
                                    </td>
                                </tr>
                            {% endfor %}
                        </tbody>
                    </table>
                </div>
        </div>
    </div>
</div>
{% endblock %}
```

Figure 5.41: Profile.html template

A corresponding view was created to render the profile page correctly.

```python
@login_required
def profile(request):
    user = request.user

    # Fetch orders related to the user, ordered by most recent first
    orders = Order.objects.filter(customer=user).order_by('-date_ordered')
    # Pass orders to the template context
    context = {
        'orders': orders,
    }

    return render(request, 'store/profile.html', context)
```

Figure 5.42: Profile view in views.py

The functional profile page is displayed below:

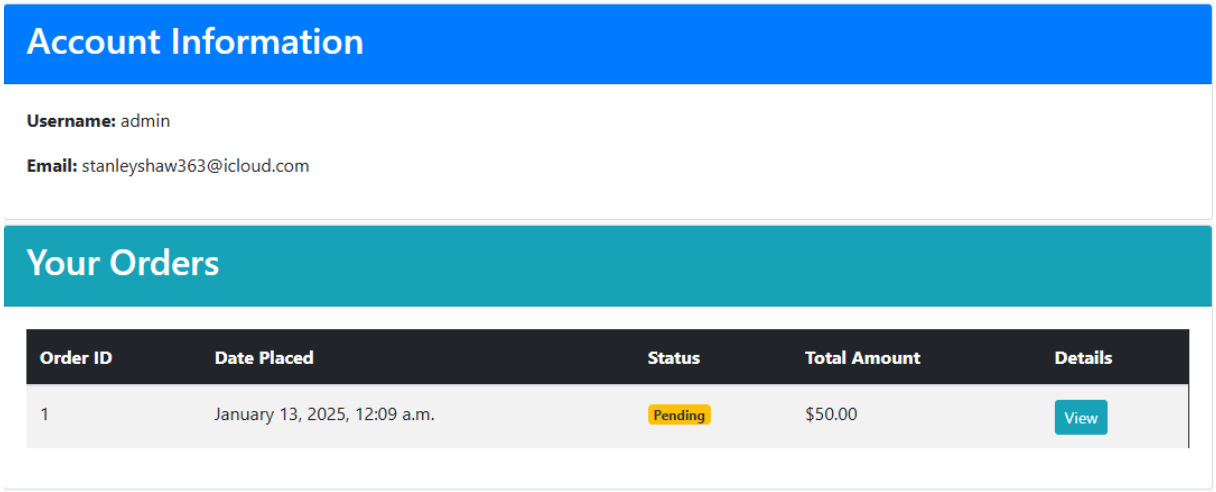

Figure 5.43: Working profile page

5.11 Admin Functionality

Enhancing Django's built-in admin dashboard was crucial for managing products, orders, and user roles effectively. Custom functionalities were added by updating `admin.py`.

```
path('admin/', admin.site.urls),
```

Figure 5.44: Updated urls.py for admin page

```
# store/admin.py

from django.contrib import admin, messages
from .models import Product, Order, OrderItem, TwoFactorAuth
from django.utils.translation import ngettext

class ProductAdmin(admin.ModelAdmin):
    list_display = ('name', 'price', 'inventory')
    search_fields = ('name',)
    list_editable = ('inventory',)
    list_filter = ('inventory',)

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    readonly_fields = ('product', 'quantity', 'price')
    can_delete = False
    extra = 0

class OrderAdmin(admin.ModelAdmin):
    list_display = ('id', 'customer', 'date_ordered', 'complete')
    inlines = [OrderItemInline]
    readonly_fields = ('customer', 'date_ordered')
    actions = ['mark_as_complete']  # Register the custom action

    def mark_as_complete(self, request, queryset):
        updated = queryset.update(complete=True)  # Bulk update the 'complete' field to True
        self.message_user(request, ngettext(
            '%d order was successfully marked as complete.',
            '%d orders were successfully marked as complete.',
            updated,
        ) % updated, messages.SUCCESS)
    mark_as_complete.short_description = "Mark selected orders as COMPLETE"
```

Figure 5.45: Admin.py with custom functions

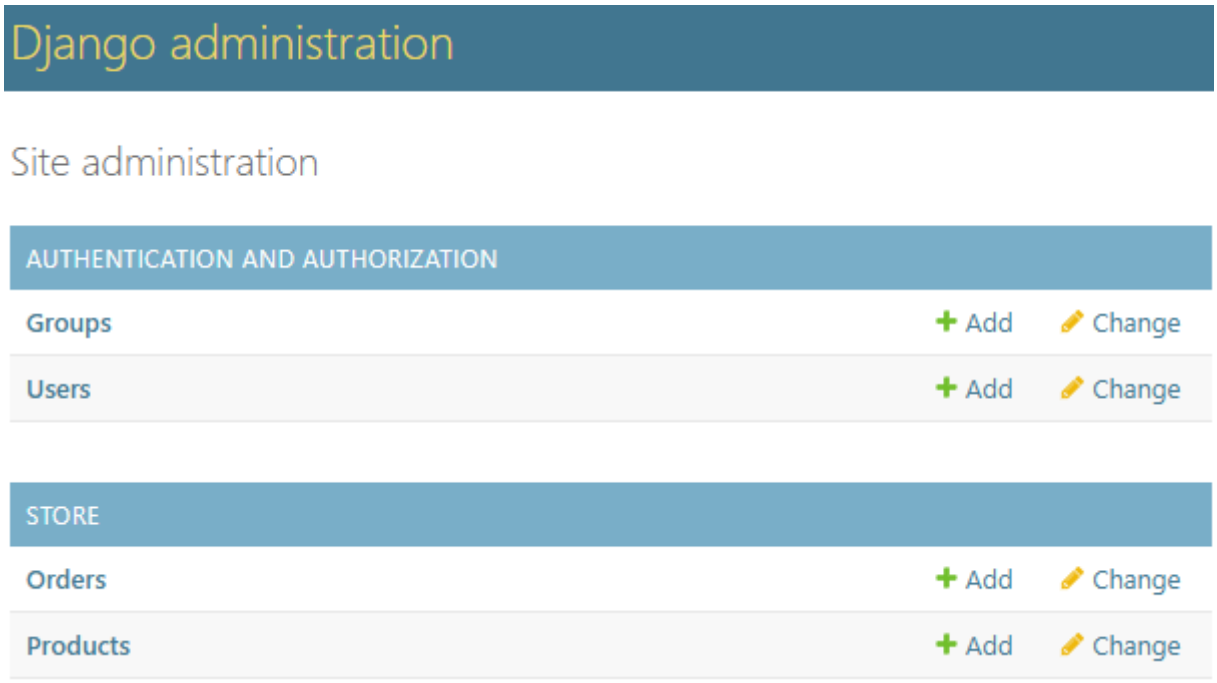

Figure 5.46: Working admin dashboard

5.12 Two-Factor Authentication (2FA)

To bolster security, Two-Factor Authentication (2FA) was integrated, allowing users to enable or disable this feature via their profile.

```
<!-- Two-Factor Authentication Card -->
<div class="card mb-4">
    <div class="card-header bg-warning text-dark d-flex justify-content-between align-items-center">
        <h2>Two-Factor Authentication (2FA)</h2>
        {% if two_fa.enabled %}
            <a href="{% url 'two_factor_setup' %}" class="btn btn-secondary btn-sm">Manage 2FA</a>
        {% endif %}
    </div>
    <div class="card-body">
        <form method="post">
            {% csrf_token %}
            {% if two_fa.enabled %}
                <p>2FA is currently <span class="badge bg-success">Enabled</span>.</p>
                <button type="submit" name="disable_2fa" class="btn btn-danger">Disable 2FA</button>
            {% else %}
                <p>2FA is currently <span class="badge bg-secondary">Disabled</span>.</p>
                <button type="submit" name="enable_2fa" class="btn btn-primary">Enable 2FA</button>
            {% endif %}
        </form>
    </div>
</div>
```

Figure 5.47: 2FA option in profile page

Templates for 2FA setup and verification were created to facilitate user interactions.

```
<!-- store/templates/store/two_factor_setup.html -->

{% extends "base.html" %}

{% block content %}
<div class="container mt-5">
    <h2 class="text-center">Set Up Two-Factor Authentication</h2>
    <div class="card mx-auto" style="max-width: 500px;">
        <div class="card-body text-center">
            <p>Scan the QR code below with your authenticator app (e.g., Google Authenticator, Authy).</p>
            <img src="{{ qr_code }}" alt="QR Code" class="img-fluid">
            <form method="post" class="mt-4">
                {% csrf_token %}
                <div class="form-group">
                    <label for="token">Enter the token generated by your app</label>
                    <input type="text" name="token" id="token" class="form-control" placeholder="123456" required>
                </div>
                <button type="submit" class="btn btn-success w-100 mt-3">Verify 2FA</button>
            </form>
            {% if messages %}
                <ul class="mt-3">
                    {% for message in messages %}
                        <li class="text-danger">{{ message }}</li>
                    {% endfor %}
                </ul>
            {% endif %}
        </div>
    </div>
</div>
{% endblock %}
```

Figure 5.48: 2FA setup template

```
{% extends "base.html" %}

{% block content %}
<div class="container mt-5">
    <h2 class="text-center">Two-Factor Authentication</h2>
    <div class="card mx-auto" style="max-width: 400px;">
        <div class="card-body">
            <p>Please enter the 2FA token from your authenticator app.</p>
            <form method="post">
                {% csrf_token %}
                <div class="form-group">
                    <label for="token">2FA Token</label>
                    <input type="text" name="token" id="token" class="form-control" placeholder="Enter 2FA token" required>
                </div>
                <button type="submit" class="btn btn-primary w-100 mt-3">Verify</button>
            </form>
            {% if messages %}
                <ul class="mt-3">
                    {% for message in messages %}
                        <li class="text-danger">{{ message }}</li>
                    {% endfor %}
                </ul>
            {% endif %}
        </div>
    </div>
</div>
{% endblock %}
```

Figure 5.49: 2FA verify template

A dedicated model was then developed to store 2FA secrets securely.

```
class TwoFactorAuth(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    enabled = models.BooleanField(default=False)
    secret = models.CharField(max_length=32, blank=True, null=True)  # Increased max_length to 32

    def __str__(self):
        return f"{self.user.username} - 2FA {'Enabled' if self.enabled else 'Disabled'}"
```

Figure 5.50: 2FA model

Views and URL configurations were updated to manage the 2FA setup and verification processes.

```python
@login_required
def two_factor_setup(request):
    user = request.user
    two_fa = get_object_or_404(TwoFactorAuth, user=user, enabled=True)

    if not two_fa.secret:
        two_fa.secret = generate_totp_secret()
        two_fa.save()
        logger.debug(f"Generated TOTP secret for user {user.username}: {two_fa.secret} (Length: {len(two_fa.secret)})")

    if request.method == 'POST':
        token = request.POST.get('token')
        if verify_totp(token, two_fa.secret):
            messages.success(request, '2FA setup is complete.')
            logger.info(f"User {user.username} completed 2FA setup.")
            return redirect('profile')  # Replace with your profile view
        else:
            messages.error(request, 'Invalid token. Please try again.')
            logger.warning(f"User {user.username} entered invalid 2FA token during setup.")

    totp_uri = get_totp_uri(user, two_fa.secret)
    qr_code_base64 = generate_qr_code(totp_uri)

    return render(request, 'store/two_factor_setup.html', {
        'qr_code': qr_code_base64,
    })
```

Figure 5.51: 2FA setup view

```python
def two_factor_verify(request):
    user_id = request.session.get('pre_2fa_user_id')
    if not user_id:
        messages.error(request, "Session expired. Please log in again.")
        return redirect('login')

    try:
        user = User.objects.get(id=user_id)
        two_fa = TwoFactorAuth.objects.get(user=user, enabled=True)
    except (User.DoesNotExist, TwoFactorAuth.DoesNotExist):
        messages.error(request, "Invalid session. Please log in again.")
        return redirect('login')

    if request.method == 'POST':
        token = request.POST.get('token')
        if verify_totp(token, two_fa.secret):
            # 2FA successful; log the user in
            login(request, user)
            # Remove the temporary session variable
            del request.session['pre_2fa_user_id']
            # Set the 2FA authenticated session variable
            request.session['two_fa_authenticated'] = True
            logger.info(f"User {user.username} successfully completed 2FA.")
            messages.success(request, "Successfully logged in with 2FA.")
            return redirect('home')
        else:
            messages.error(request, "Invalid 2FA token. Please try again.")

    return render(request, 'store/two_factor_verify.html')
```

Figure 5.52: 2FA verify view

The default Django login was customised to incorporate 2FA, redirecting users to the 2FA verification page post-login.

```python
def custom_login(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password')
            user = authenticate(username=username, password=password)
            if user is not None:
                try:
                    two_fa = TwoFactorAuth.objects.get(user=user, enabled=True)
                    # Store user's ID in session for 2FA verification
                    request.session['pre_2fa_user_id'] = user.id
                    logger.info(f"User {user.username} authenticated, requires 2FA.
                    return redirect('two_factor_verify')
                except TwoFactorAuth.DoesNotExist:
                    # 2FA is not enabled; log the user in
                    login(request, user)
                    logger.info(f"User {user.username} logged in without 2FA.")
                    return redirect('home')
            else:
                messages.error(request, "Invalid username or password.")
        else:
            messages.error(request, "Invalid username or password.")
    else:
        form = AuthenticationForm()
    return render(request, 'store/login.html', {'form': form})
```

Figure 5.53: Custom login redirecting to 2FA

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),  # Keep for password management
    path('', include('store.urls')),
    path('checkout/', views.checkout, name='checkout'),
    path('order_confirmation/<int:pk>/', views.order_confirmation, name='order_confirmation'),
    path('register/', views.register, name='register'),
    path('', views.home, name='home'),
    path('login/', views.custom_login, name='login'),  # Use custom login view
    path('2fa/verify/', views.two_factor_verify, name='two_factor_verify'),  # 2FA verification
    path('2fa/setup/', views.two_factor_setup, name='two_factor_setup'),    # 2FA setup
    path('logout/', views.custom_logout, name='logout'),                    # Use custom logout view

]
```

Figure 5.54: 2FA redirection in urls.py

QR codes for 2FA were generated using the `pyotp` and `qrcode` Python modules.

```python
# store/utils.py

import pyotp
import qrcode
import io
import base64

def generate_totp_secret():
    return pyotp.random_base32(length=32)  # Generates a 32-character base32 secret (160 bits)

def get_totp_uri(user, secret):
    return pyotp.totp.TOTP(secret).provisioning_uri(
        name=user.email, issuer_name="SecureCart"
    )

def generate_qr_code(uri):
    qr = qrcode.QRCode(
        version=1,
        box_size=10,
        border=5
    )
    qr.add_data(uri)
    qr.make(fit=True)
    img = qr.make_image(fill='black', back_color='white')
    buf = io.BytesIO()
    img.save(buf, format='PNG')
    image_stream = buf.getvalue()
    # Encode the image to base64 to embed in HTML
    return base64.b64encode(image_stream).decode('utf-8')

def verify_totp(token, secret):
    totp = pyotp.TOTP(secret)
    return totp.verify(token)
```

Figure 5.55: QR code generation for 2FA

The 2FA functionality was thoroughly tested, ensuring users could enable or disable it as required.

Figure 5.56: 2FA QR code in action

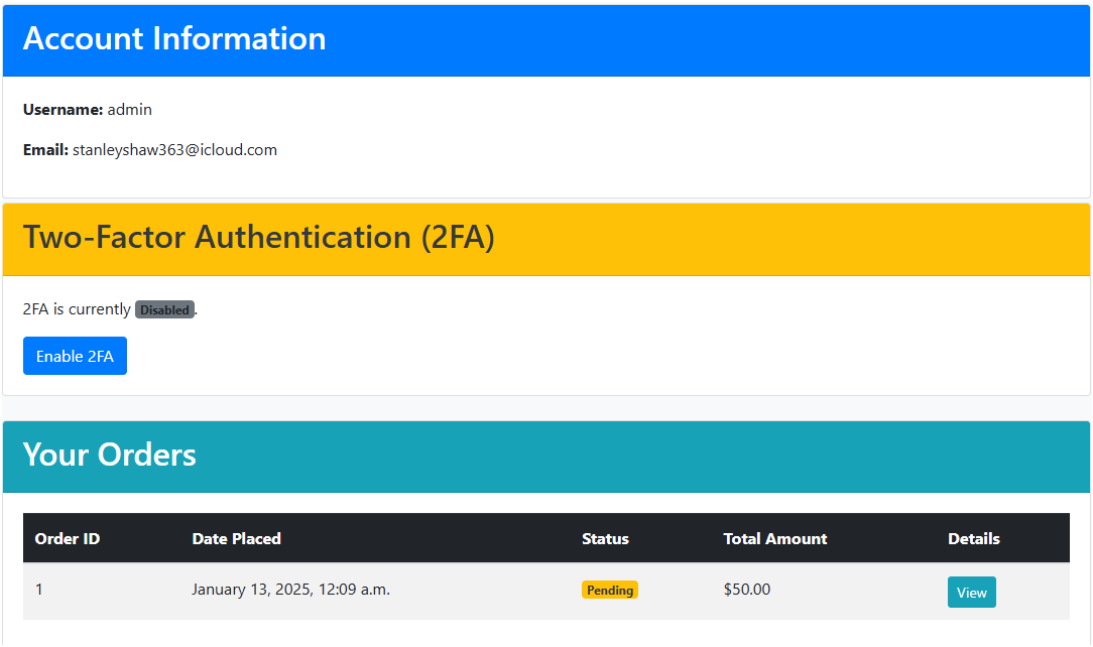

Figure 5.57: Profile page with 2FA option

5.13 Secure Payment System

Integrating a secure payment system was crucial for the platform's e-commerce functionality. Stripe was selected for its robust API and PCI compliance, allowing for secure and efficient payment processing.

Initially, the Stripe library was imported, and the API keys were configured via environment variables to maintain security.

```
STRIPE_PUBLISHABLE_KEY=pk_test_51Qgw3AF8csQZ5iopj96DokQnn70TWdCm383paWZBVCBOL
STRIPE_SECRET_KEY=sk_test_51Qgw3AF8csQZ5iopUOungNZyP580IWs4DOfhol5w98YYlF771T
```

Figure 5.58: Stripe API key configuration

```
STRIPE_PUBLISHABLE_KEY = config('STRIPE_PUBLISHABLE_KEY', default='')
STRIPE_SECRET_KEY = config('STRIPE_SECRET_KEY', default='')
```

Figure 5.59: Retrieving Stripe API keys

Templates for handling successful and cancelled purchases were created to provide clear feedback to users.

```
{% extends 'base.html' %}
{% load static %}

{% block content %}
<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <div class="alert alert-warning text-center" role="alert">
        <h1 class="display-4">Payment Canceled</h1>
        <p class="lead mb-4">
          Your payment was canceled. You have not been charged.
        </p>
        <a href="{% url 'cart_detail' %}" class="btn btn-secondary btn-lg">Return to Cart</a>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

Figure 5.60: Cancelled order template

```
{% extends 'base.html' %}
{% load static %}

{% block content %}
<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <div class="alert alert-success text-center" role="alert">
        <h1 class="display-4">Payment Successful!</h1>
        <p class="lead mb-4">
          Your payment has been confirmed. Order #{{ order.id }} has been placed.
        </p>
        <a href="{% url 'home' %}" class="btn btn-primary btn-lg">Return Home</a>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

Figure 5.61: Successful order template

The `views.py` file was updated to redirect users to the Stripe purchase page during checkout and to handle the responses for successful or cancelled payments.

```
def payment_cancel(request):
    """
    Called if the user cancels on Stripe's checkout page.
    """
    messages.warning(request, "Payment canceled. You have not been charged.")
    return render(request, 'store/payment_cancel.html')
```

Figure 5.62: Payment cancel view

```python
@login_required
def payment_success(request):

    session_id = request.GET.get('session_id', None)
    if not session_id:
        messages.error(request, "No session ID provided. Unable to confirm payment.")
        return redirect('cart_detail')

    # Retrieve session from Stripe
    try:
        checkout_session = stripe.checkout.Session.retrieve(session_id)
    except Exception as e:
        logger.error(f"Stripe session retrieve error: {e}")
        messages.error(request, "Could not retrieve your payment session. Please contact support.")
        return redirect('cart_detail')

    # Check if the payment was successful
    if checkout_session.payment_status != 'paid':
        # Payment not completed or was canceled
        messages.warning(request, "Payment not completed yet or canceled.")
        return redirect('cart_detail')

    # Now that we know payment_status == 'paid':
    # Create the Order from the user's cart
    cart_id = checkout_session.client_reference_id
    cart = get_object_or_404(Cart, id=cart_id, user=request.user)

    order = Order.objects.create(customer=request.user)
    for item in cart.items.all():
        OrderItem.objects.create(
            order=order,
            product=item.product,
            quantity=item.quantity,
            price=item.product.price
        )
        # Reduce product inventory
        item.product.inventory -= item.quantity
        item.product.save()

    # Clear cart
    cart.delete()

    logger.info(f"Order {order.pk} created for user {request.user.username} after Stripe payment.")
    messages.success(request, "Payment successful! Your order has been placed.")
    return render(request, 'store/payment_success.html', {'order': order})
```

Figure 5.63: Payment success view

```python
@login_required
def checkout(request):
    cart, _ = Cart.objects.get_or_create(user=request.user)
    cart_total = cart.get_total_price()  # e.g., 49.99
    if cart_total <= 0:
        messages.error(request, "Your cart is empty or invalid.")
        return redirect('cart_detail')
    # Convert total dollars to cents
    amount_in_cents = int(cart_total * 100)

    # Build absolute URLs for success/cancel
    YOUR_DOMAIN = request.build_absolute_uri('/')[:-1]  -
    success_url = YOUR_DOMAIN + reverse('payment_success') + "?session_id={CHECKOUT_SESSION_ID}"
    cancel_url = YOUR_DOMAIN + reverse('payment_cancel')

    try:
        checkout_session = stripe.checkout.Session.create(
            payment_method_types=['card'],
            line_items=[
                {
                    'price_data': {
                        'currency': 'usd',
                        'product_data': {
                            'name': 'SecureCart Purchase'
                        },
                        'unit_amount': amount_in_cents,
                    },
                    'quantity': 1,
                },
            ],
            mode='payment',
            success_url=success_url,
            cancel_url=cancel_url,
            client_reference_id=str(cart.id),  # Store cart.id or user.id, etc.
        )
        return redirect(checkout_session.url, code=303)
    except Exception as e:
        logger.error(f"Error creating Stripe Checkout Session: {e}")
        messages.error(request, "There was a problem starting your payment. Please try again.")
        return redirect('cart_detail')
```

Figure 5.64: Updated checkout view

Comprehensive testing confirmed that Stripe integration functioned as intended, as evidenced by the following:

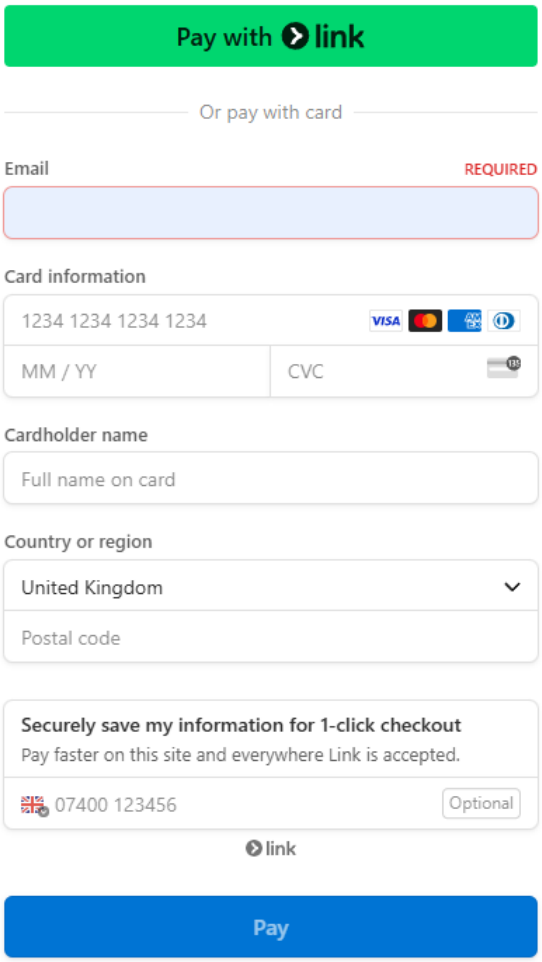

Figure 5.65: Working card payments

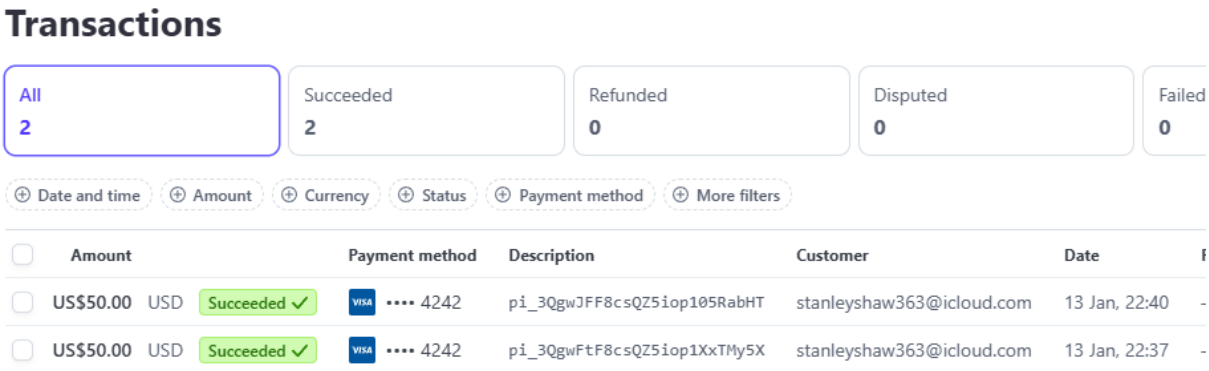

Figure 5.66: Stripe dashboard

5.14 Security Enhancements

Ensuring the security of the *SecureCart* platform was paramount. Several measures were implemented to protect data integrity and user privacy.

Detailed logging was set up using Django's built-in logging features to monitor and detect potential threats.

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {message}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    'handlers': {
        'file': {
            'level': 'INFO',
            'class': 'logging.FileHandler',
            'filename': os.path.join(BASE_DIR, 'django_debug.log'),
            'formatter': 'verbose',

        },
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
        },
    },
    'loggers': {
        # This logger handles all Django server requests.
        'django.server': {
            'handlers': ['console'],  # Only log HTTP requests to console, not to file
            'level': 'WARNING',  # Set to WARNING to avoid logging INFO-level HTTP requests
            'propagate': False,
        },
        'django': {
            'handlers': ['file', 'console'],
            'level': 'INFO',
            'propagate': True,
        },
        'securecart': {
            'handlers': ['file', 'console'],
            'level': 'INFO',
            'propagate': False,
        },
    },

}
```

Figure 5.67: Logging configuration in settings.py

```
INFO User admin logged in without 2FA.
INFO Home page accessed by user: admin from IP: 127.0.0.1
INFO Product detail viewed - Product ID: 1, Product Name: Desk, Viewed by: admin from IP: 127.0.0.1
INFO Product added to cart - Product ID: 1, Product Name: Desk, User: admin, IP: 127.0.0.1, Quantity: 1
INFO Cart viewed - User: admin, IP: 127.0.0.1
```

Figure 5.68: Detailed logging implementation

Encryption was enforced across all network traffic by mandating HTTPS, ensuring that data in transit was secure. Additionally, all database transactions utilised TLS/SSL protocols.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': config('DB_NAME'),  # Default DB name
        'USER': config('DB_USER'),  # Default DB user
        'PASSWORD': config('DB_PASSWORD'),  # Default password
        'HOST': config('DB_HOST'),  # Default host
        'PORT': config('DB_PORT'),  # Default port
        'OPTIONS': {
            'sslmode': 'require', # Ensures encryption
            'sslrootcert': os.path.join(BASE_DIR, 'global-bundle.pem'),
        },
    }
}
```

Figure 5.69: Encrypted database transactions

```
SECURE_HSTS_SECONDS = 31536000
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True
SECURE_SSL_REDIRECT = True
```

Figure 5.70: Enforcing HTTPS connections

Furthermore, CSRF tokens and cookies were configured to be transmitted exclusively over HTTPS, preventing interception and ensuring session security.

```
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
```

Figure 5.71: Securing CSRF tokens and cookies over HTTPS

5.15 UI Enhancements

The aesthetic appeal of the website was enhanced using custom CSS and the Bootstrap framework, resulting in a more engaging and user-friendly interface.

```css
body {
    background-color: #f8f9fa; /* Light grey background */
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif; /* Modern font */
}

.navbar {
    margin-bottom: 30px; /* Space below navbar */
    box-shadow: 0 4px 6px -1px rgba(0,0,0,0.1); /* Subtle shadow */
}

h1 {
    text-align: center;
    margin-top: 40px;
    color: #343a40; /* Dark grey color */
}

.navbar-brand {
    font-weight: bold;
    color: #007bff !important; /* Bootstrap primary color */
}

.nav-link {
    color: #343a40 !important; /* Dark grey color */
    margin-right: 15px;
}

.nav-link:hover {
    color: #007bff !important;
}

.form-inline {
    display: flex;
    align-items: center;
}
.form-inline input {
    margin-right: 10px;
}

.container {
    padding-top: 20px;
}
.product-image {
    width: 100%;
    height: 100%;
    object-fit: fill;
}
.alert .btn-close {
    display: none;
}
footer {
    text-align: center;
    padding: 20px 0;
    margin-top: 50px;
    border-top: 1px solid #eaeaea;
    color: #6c757d;
}
```

Figure 5.72: Updated styles.css for website aesthetics

These enhancements culminated in a visually appealing store page, featuring a cohesive colour scheme and improved layout.

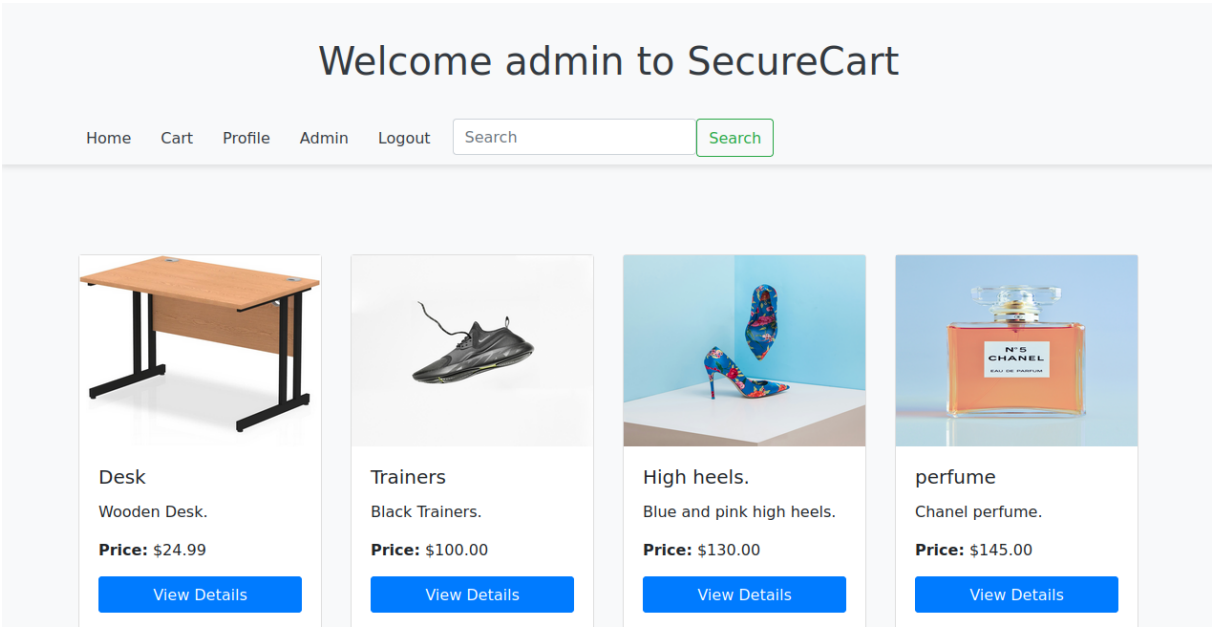

Figure 5.73: Store page with new CSS

6. Conclusion

This report has demonstrated how *SecureCart* was developed using the V-Model SDLC to ensure both robust functionality and security. It has covered gathering and verifying requirements, designing a three-tier architecture, and implementing essential features including secure login, two-factor authentication, and a Stripe-integrated checkout. Security measures, such as encrypted data storage, HTTPS enforcement, and environment variables for sensitive information, were systematically employed. Through careful testing, detailed logging, and a focus on user-friendly design (including a clean Bootstrap interface), the final result is an e-commerce platform that offers reliability, scalability, and enhanced protection for all user data.

7. Appendix

Project files:
https://github.com/stanly363/Secure-ecomerse-website

Bibliography

[1] Darktrace.com. (2024). Darktrace. [online] Available at: https://darktrace.com/cyber-ai-glossary/cybersecurity-for-retail-ecommerce.

[2] Nakkasem, T. (2020). V-Model. [online] Medium. Available at: https://medium.com/software-engineering-kmitl/v-model-3a71622b3d82.

[3] Django (2024). Django documentation — Django documentation. [online] Django Project. Available at: https://docs.djangoproject.com/en/5.1/.

[4] Otto, M. (2019). Introduction. [online] Getbootstrap.com. Available at: https://getbootstrap.com/docs/4.1/getting-started/introduction/.

[5] PostgreSQL.org. (n.d.). PostgreSQL: Documentation. [online] Available at: https://www.postgresql.org/docs/.

[6] Stripe.com. (2024). Documentation. [online] Available at: https://docs.stripe.com/?locale=en-GB.

[7] Fanmuy, G., Fraga, A., and Llorens, J. (2012). Requirements verification in the industry. In *Complex Systems Design & Management: Proceedings of the Second International Conference on Complex Systems Design & Management CSDM 2011* (pp. 145-160). Springer Berlin Heidelberg.

[8] OWASP (2017). Authentication · OWASP Cheat Sheet Series. [online] Available at: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html.

[9] Kelseyk (2021). NIST Password Reset Guidelines. [online] Specops Software. Available at: https://specopssoft.com/blog/nist-password-reset-guidelines/.

[10] NIST (2022). Multi-Factor Authentication. [online] NIST. Available at: https://www.nist.gov/itl/smallbusinesscyber/guidance-topic/multi-factor-authentication.

[11] ICO (2023). Overview – Data Protection and the EU. [online] ICO. Available at: https://ico.org.uk/for-organisations/data-protection-and-the-eu/overview-data-protection-and-the-eu/.

[12] wa.aws.amazon.com. (n.d.). Performance Efficiency - AWS Well-Architected Framework. [online] Available at: https://wa.aws.amazon.com/wellarchitected/2020-07-02T19-33-23/wat.pillar.performance.en.html.

[13] AWS (2024). Three-Tier Architecture Overview - AWS Serverless Multi-Tier Architectures with Amazon API Gateway and AWS Lambda. [online] Available at: https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/three-tier-architecture-overview.html.

[14] OWASP (2025). WSTG - v4.1 — OWASP Foundation. [online] Available at: https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/04-Authentication_Testing/07-Testing_for_Weak_Password_Policy.

[15] OWASP (2019). Input Validation OWASP Cheat Sheet Series. [online] Available at: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html.

[16] PlantUML.com. (n.d.). Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams. [online] Available at: https://plantuml.com/.

[17] Drawing UML with PlantUML PlantUML Language Reference Guide. (n.d.). Available at: https://pdf.plantuml.net/1.2019.3/PlantUML_Language_Reference_Guide_en.pdf [Accessed 20 Jan. 2025].

[18] OWASP. (n.d.). Secure Product Design - OWASP Cheat Sheet Series. [online] Available at: https://cheatsheetseries.owasp.org/cheatsheets/Secure_Product_Design_Cheat_Sheet.html.

[19] National Cyber Security Centre (2019). Secure design principles. [online] Available at: https://www.ncsc.gov.uk/collection/cyber-security-design-principles.

[20] Django Project. (2024). Password management in Django — Django documentation. [online] Available at: https://docs.djangoproject.com/en/5.1/topics/auth/passwords/.

[21] Amazon.com. (2025). Encrypt an existing Amazon RDS for PostgreSQL DB instance - AWS Prescriptive Guidance. [online] Available at: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/encrypt-an-existing-amazon-rds-for-postgresql-db-instance.html [Accessed 21 Jan. 2025].

[22] CLOUDFLARE (2024). What is Transport Layer Security? — TLS protocol — Cloudflare UK. Cloudflare. [online] Available at: https://www.cloudflare.com/en-gb/learning/ssl/transport-layer-security-tls/.

[23] Duisebekova, K., Khabirov, R. and Zholzhan, A., 2021. Django as Secure Web-Framework in Practice.(1), pp.275-281.

[24] AWS (2024). What is Amazon CloudWatch? - Amazon CloudWatch. [online] Amazon.com. Available at: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html.

[25] Django Project. (2025). Logging — Django documentation. [online] Available at: https://docs.djangoproject.com/en/5.1/topics/logging/ [Accessed21Jan.2025].

[26] AWS (2024). What Is AWS Key Management Service? - AWS Key Management Service. [online] docs.aws.amazon.com. Available at: https://docs.aws.amazon.com/kms/latest/developerguide/overview.html.