Project on

Personal E-commerce website

Submitted by

MD. ZIHAD HOSSAIN

664135

in partial fulfilment of the requirement for the Diploma in Computer Science and Technology

Department of Computer Science and Technology Faculty of Engineering and Technology Daffodil Technical Institute
January 2026

Declaration

We hereby announce that the work is being presented in this project entitled "Personal E-commerce" Partial fulfilment of requirements for Degree of Diploma in Computer Science and Technology under Faculty of Engineering and Technology, Daffodil Technical Institute is an authentic record of our own work done under the supervision of **Md Noman Jahan.** It is also not declared in this report or any part of it has been submitted elsewhere for the award of any degree.

____

MD. ZIHAD HOSSAIN

Approval

The project titled **"Personal E-commerce System", submitted by Md.Zihad Hossain(664135)** ,has been accepted satisfactorily in partial fulfilment of the requirements for the Degree of Diploma in Engineering Computer Science and Technology.

DTI of Examiners

Md Noman Jahan

Instructor (Supervisor)

Department of Computer Science and Technology Instructor

Daffodil Technical Institute

Tasfina Haque

Instructor (Instructor)

Department of Computer Science and Technology

Daffodil Technical Institute

Personal E-Commerce Website Project Book

Project Title: Personal E-Commerce System Development **Date:** December 2025
Submitted in partial fulfillment of the requirements for the Degree of Diploma in Computer Science and Technology

---

Table of Contents

Front Matter 1. Declaration 2. Approval 3. Abstract

Chapter 1: Introduction and Project Overview 1.1 Project Background and Rationale 1.2 Project Scope and Objectives 1.3 Technology Stack Deep Dive 1.4 Key Features and Functionalities

Chapter 2: System Architecture and Design 2.1 The Model-View-Template (MVT) Architectural Pattern 2.2 Application Structure and Modularity 2.3 Component Interaction Flow (Request-Response Cycle) 2.4 Security Features and Best Practices

Chapter 3: Database Design and Modeling 3.1 Entity-Relationship Diagram (ERD) Analysis 3.2 Core Model Schemas: User and Profile 3.3 Product Catalog and Inventory Models 3.4 Transactional Models: Cart and Order

Chapter 4: Backend Development with Django 4.1 Project Configuration and Settings 4.2 URL Routing and Namespacing 4.3 Detailed Model Implementation with Code 4.4 Implementing Core View Functions (Code Snippets) 4.5 Form Handling and Validation

Chapter 5: Frontend Development and User Experience 5.1 Template Structure and Inheritance 5.2 Styling with Bootstrap 5 and Custom CSS 5.3 Enhancing Interactivity with JavaScript and AJAX 5.4 Responsive Design Implementation

Chapter 6: Testing and Quality Assurance 6.1 Comprehensive Testing Strategy 6.2 Unit Testing Examples (Models and Utilities) 6.3 Integration and Functional Testing (Checkout Process) 6.4 Code Review and Quality Metrics

Chapter 7: Deployment and Maintenance 7.1 Production Environment Setup (PostgreSQL, Gunicorn, Nginx) 7.2 Step-by-Step Deployment Guide 7.3 Monitoring and Logging Strategy

Chapter 8: Results, Achievements, and Future Enhancements 8.1 Functional Completeness and Technical Achievements 8.2 Business Value and Key Performance Indicators (KPIs) 8.3 Future Enhancements Roadmap

Back Matter References

Chapter 1: Introduction and Project Overview

1.1 Project Background and Rationale

The rapid growth of digital commerce necessitates the development of robust, scalable, and secure online platforms. This project, the **Personal E-Commerce System**, was undertaken to provide a comprehensive, full-stack solution for online retail, serving as a practical demonstration of modern web development principles. The rationale for this project is twofold: to create a fully functional commercial platform and to serve as a detailed technical blueprint for future development and academic study.

The system is built on the premise of delivering a seamless user experience while ensuring administrative efficiency. By leveraging the Django framework, the project prioritizes security, maintainability, and rapid feature development, adhering to the "Don't Repeat Yourself" (DRY) principle.

1.2 Project Scope and Objectives

The primary objective of this project is to develop a minimum viable product (MVP) for an e-commerce website capable of handling the core business processes of online retail.

Key Objectives: 1. **Secure User Management:** Implement secure registration, authentication, and profile management. 2. **Dynamic Product Catalog:** Create a database-driven system for managing products, categories, and inventory. 3. **Transactional Integrity:** Develop a robust shopping cart and checkout process that ensures atomic order creation. 4. **Responsive Design:** Ensure the application is fully accessible and usable across all device types (desktop, tablet, mobile). 5. **Administrative Control:** Provide a secure interface for managing all aspects of the store (products, orders, users).

The scope includes the development of the backend logic, database schema, and the user-facing frontend, but excludes integration with live payment gateways, which is relegated to the future enhancements roadmap.

1.3 Technology Stack Deep Dive

The selection of the technology stack was a deliberate choice to balance development speed with production-grade reliability. The core components are centered around Python and the Django framework.

Component	Technology	Version/Standard	Rationale for Selection
Backend Framework	Django (Python)	5.0+	High-level, "batteries-included" framework known for its security, ORM, and built-in administrative interface.
Database (Production)	PostgreSQL	Latest Stable	Industry-standard relational database, chosen for its transactional integrity,

			advanced indexing, and superior performance under high load.
Database (Development)	SQLite	Standard	Zero-configuration setup, ideal for rapid prototyping and local development environments.
Frontend Styling	Bootstrap	5.3+	Provides a powerful, mobile-first grid system and a comprehensive library of components, significantly reducing UI development time.
Frontend Interactivity	Vanilla JavaScript/jQuery	ES6+	Used for client-side enhancements, such as AJAX calls for cart updates and form validation, maintaining a lightweight frontend.
Web Server (WSGI)	Gunicorn	Latest Stable	A fast, stable, and lightweight WSGI server for deploying Python web applications in production.
Reverse Proxy	Nginx	Latest Stable	Used for load balancing, static file serving, and SSL termination, enhancing security and performance.

1.4 Key Features and Functionalities

The platform is designed with a comprehensive set of features to support both the customer journey and administrative operations.

Customer-Facing Features: * **User Authentication:** Secure login, registration, and password reset functionality. * **Product Browsing:** Categorized product listings, search functionality, and product detail pages. * **Shopping Cart:** Session-based cart management with quantity updates and subtotal calculation. * **Checkout Process:** Multi-step form for collecting shipping information and confirming the order. * **Order History:** User dashboard to view past orders and track current order status.

Administrator Features (via Django Admin): * **Product Management:** CRUD operations for products, including image uploads and inventory levels. * **Category Management:** Hierarchical organization of the product catalog. * **Order Management:** Viewing, filtering, and updating the status of all customer orders. * **User Management:** Oversight and modification of user accounts and profiles.

Chapter 2: System Architecture and Design

2.1 The Model-View-Template (MVT) Architectural Pattern

The project strictly adheres to the Django **Model-View-Template (MVT)** pattern, which is a variation of the traditional Model-View-Controller (MVC) architecture. This pattern is fundamental to Django's design philosophy, ensuring a clear separation of concerns.

Component	Role in MVT	Function
Model	Data Access Layer	Defines the structure of the data, handles database interactions, and enforces data integrity and business logic related to the data.
View	Business Logic Layer	Receives the HTTP request, interacts with the Model to retrieve or modify data, and selects the appropriate Template to render the response. This is the "Controller" in traditional MVC.
Template	Presentation Layer	Defines the structure and layout of the user interface (HTML/CSS/JS), using Django's template language to display data passed from the View.

2.2 Application Structure and Modularity

The project is divided into several Django applications, each representing a distinct functional module. This modularity is key to the system's scalability and maintainability.

Application Name	Primary Responsibility	Key Models/Components
`core`	Project-wide settings, base templates, and utility functions.	`settings.py`, `urls.py` (root)
`accounts`	User authentication, registration, and profile management.	`UserProfile`
`store`	Product catalog, categories, and inventory management.	`Category`, `Product`
`cart`	Shopping cart logic, session management, and cart item manipulation.	`Cart`, `CartItem`
`orders`	Checkout process, order creation, and order history.	`Order`, `OrderItem`

2.3 Component Interaction Flow (Request-Response Cycle)

The following steps detail the lifecycle of a typical user request, such as adding a product to the cart:

1. **Request Initiation:** The user clicks the "Add to Cart" button, sending an HTTP POST request to a specific URL (e.g., `/cart/add/`).

2. **URL Dispatcher:** The request is intercepted by the root `urls.py`, which routes it to the `cart` application's URL configuration.
3. **View Execution:** The URL is matched to the `add_to_cart` function in `cart/views.py`.
4. **Model Interaction (Business Logic):**
   - The View checks the request method (POST) and validates the CSRF token.
   - It retrieves the product ID and quantity from the request data.
   - It queries the **Model** (`Cart` and `CartItem`) to find or create the user's cart.
   - It updates the `CartItem` quantity or creates a new item.
5. **Response Generation:** The View determines the next step, typically redirecting the user to the cart summary page or back to the product page.
6. **Template Rendering (if applicable):** If the view is rendering a page (e.g., the cart summary), it passes the cart data to the **Template** (`cart/summary.html`), which generates the final HTML response sent back to the user.

2.4 Security Features and Best Practices

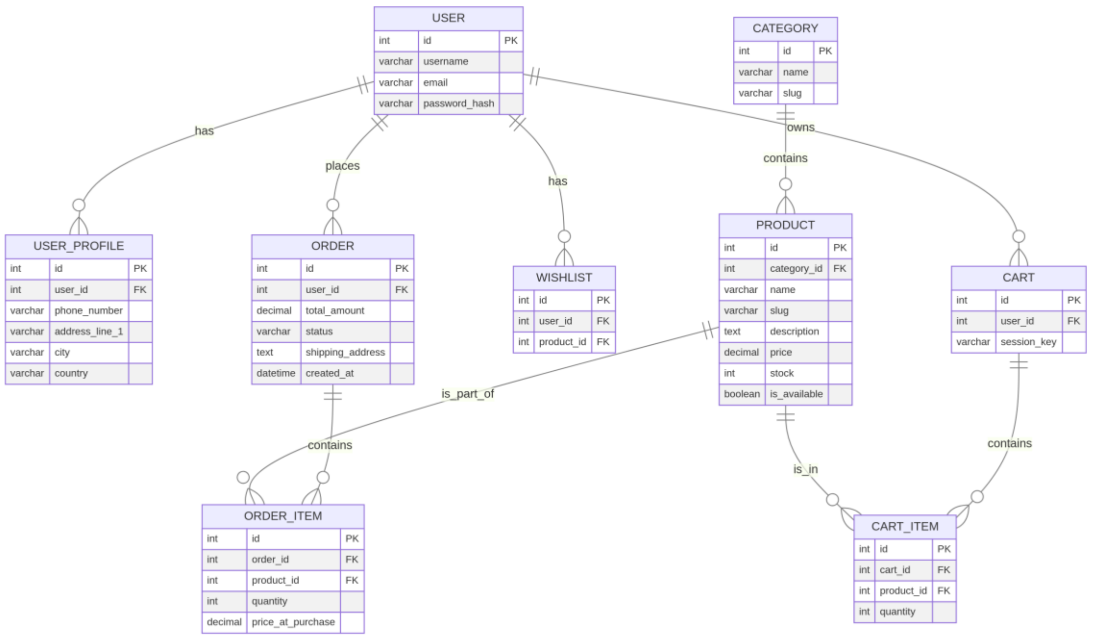
Security is paramount in an e-commerce application. The project leverages Django's built-in security features and adheres to industry best practices.

Security Feature	Implementation Detail	Mitigation
CSRF Protection	Django's middleware automatically checks for the `csrf_token` in all POST requests.	Prevents Cross-Site Request Forgery attacks.
SQL Injection	Exclusive use of the Django Object-Relational Mapper (ORM).	The ORM automatically escapes all query parameters, preventing malicious SQL code injection.
XSS Prevention	Django's template system automatically escapes output variables by default.	Prevents Cross-Site Scripting attacks by treating user input as plain text, not executable code.
Password Hashing	Utilizes Django's built-in `PBKDF2 with SHA256` algorithm with a salt.	Ensures that user passwords are never stored in plain text and are highly resistant to brute-force attacks.
Session Security	Sessions are stored in the database and use cryptographically signed cookies.	Protects against session hijacking and tampering.

Chapter 3: Database Design and Modeling

3.1 Entity-Relationship Diagram (ERD) Analysis

The database schema is designed to be highly normalized, ensuring data integrity and minimizing redundancy, particularly for transactional data. The core entities and their relationships are visualized in the Entity-Relationship Diagram (ERD) below.



Entity-Relationship Diagram for E-Commerce System

Key Relationships: * **One-to-One:** `USER` and `USER_PROFILE` (Ensures separation of authentication data from personal details). * **One-to-Many:** `CATEGORY` to `PRODUCT` (A category can have many products). * **One-to-Many:** `ORDER` to `ORDER_ITEM` (An order contains multiple line items). * **Many-to-Many (Implicit):** `USER` and `PRODUCT` via `WISHLIST` (A user can wishlist many products, and a product can be wishlisted by many users).

3.2 Core Model Schemas: User and Profile

The `accounts` application manages the user data, extending the default Django `User` model to include necessary e-commerce-specific fields like shipping information.

Model: `UserProfile` | Field Name | Data Type | Description | Relationship | | :— | :— | :— | :— | | `user` | OneToOneField | Link to the built-in Django `User` model. | One-to-One (Mandatory) | | `phone_number` | CharField | User’s contact number. | Optional | | `address_line_1` | CharField | Primary shipping address line. | Mandatory | | `city` | CharField | City for shipping. | Mandatory | | `country` | CharField | Country for shipping. | Mandatory |

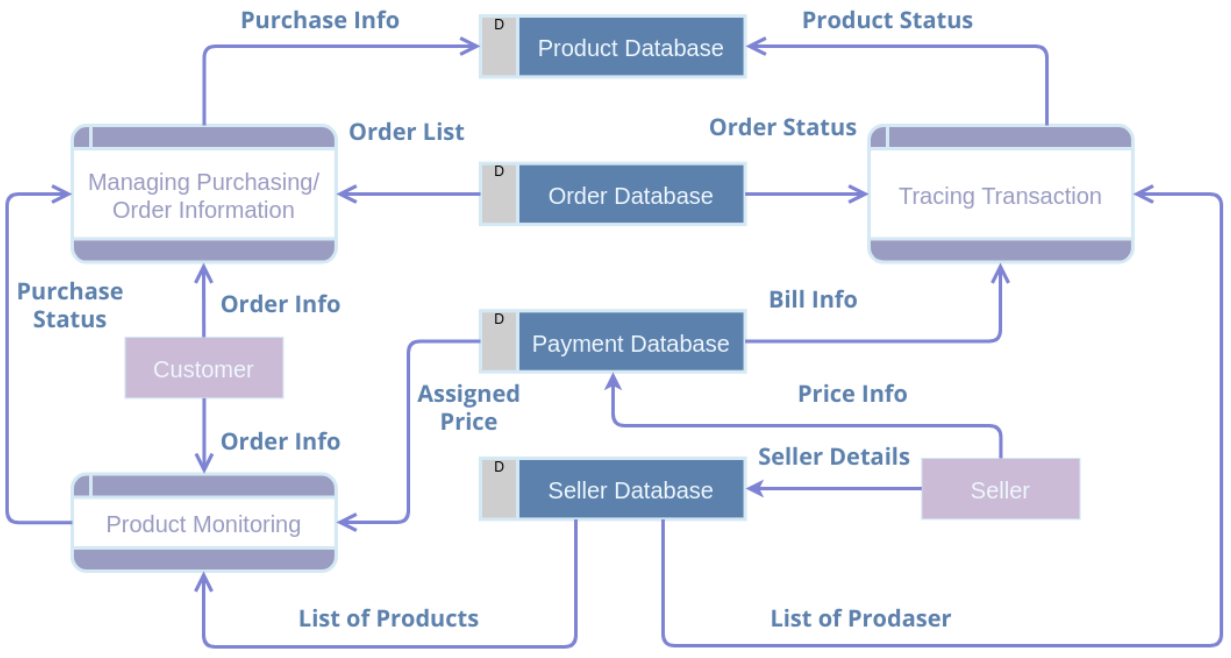
Conceptual Code Snippet (`accounts/models.py`):

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

class UserProfile(models.Model):
    """Extends the default User model for e-commerce specific details."""
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    phone_number = models.CharField(max_length=15, blank=True, null=True)
    address_line_1 = models.CharField(max_length=100)
    city = models.CharField(max_length=50)
    country = models.CharField(max_length=50)

    def __str__(self):
        return f"Profile for {self.user.username}"
```

*Data Flow Diagram:



3.3 Product Catalog and Inventory Models

The `store` application defines the core retail entities. The `Product` model includes inventory tracking (`stock`) and availability status (`is_available`).

Model: `Product` | Field Name | Data Type | Description | Constraints | | :— | :— | :— | :— | | `category` | ForeignKey | Links the product to its category. | One-to-Many | | `name` | CharField | Full name of the product. | Unique | | `slug` | SlugField | URL-friendly identifier for the product. | Unique | | `description` | TextField | Detailed product description. | | |

`price` | DecimalField | Selling price of the product. | Max 10 digits, 2 decimal places | | `stock` | IntegerField | Current inventory level. | Must be non-negative | | `is_available` | BooleanField | Flag to show/hide product on the storefront. | Default: True |

Conceptual Code Snippet (`store/models.py`):

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100, unique=True)
    slug = models.SlugField(max_length=100, unique=True)

    class Meta:
        verbose_name_plural = 'categories'
        ordering = ('name',)

class Product(models.Model):
    category = models.ForeignKey(Category, on_delete=models.CASCADE,
related_name='products')
    name = models.CharField(max_length=255, unique=True)
    slug = models.SlugField(max_length=255, unique=True)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.IntegerField()
    is_available = models.BooleanField(default=True)
    image = models.ImageField(upload_to='products/%Y/%m/%d', blank=True)

    class Meta:
        ordering = ('name',)
        index_together = (('id', 'slug'),)

    def get_absolute_url(self):
        from django.urls import reverse
        return reverse('store:product_detail', args=[self.category.slug,
self.slug])
```

3.4 Transactional Models: Cart and Order

These models are crucial for the transactional integrity of the system. The `OrderItem` model is designed to capture the price of the product *at the time of purchase*, ensuring that historical order records are not affected by future price changes.

Transactional Model Summary: | Model | Purpose | Relationship to Product | Key Integrity Feature | | :— | :— | :— | :— | | `Cart` | Temporary storage for items before checkout. | One-to-Many with `CartItem` | Can be session-based for guests. | | `CartItem` | Line item in the shopping cart. | ForeignKey to `Product` | Tracks current quantity. | | `Order` | Permanent record of a completed transaction. | One-to-Many with `OrderItem` | Stores shipping and payment details. | | `OrderItem` | Permanent record of a product purchased. | ForeignKey to `Product` | **Captures `price_at_purchase` to ensure immutability.** |

Conceptual Code Snippet (`orders/models.py`):

```
from django.db import models
from django.contrib.auth.models import User
from store.models import Product

class Order(models.Model):
    STATUS_CHOICES = [
        ('Pending', 'Pending'),
        ('Processing', 'Processing'),
        ('Shipped', 'Shipped'),
        ('Delivered', 'Delivered'),
        ('Cancelled', 'Cancelled'),
    ]
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    shipping_address = models.TextField()
    total_amount = models.DecimalField(max_digits=10, decimal_places=2)
    status = models.CharField(max_length=10, choices=STATUS_CHOICES,
default='Pending')
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('-created_at',)

class OrderItem(models.Model):
    order = models.ForeignKey(Order, on_delete=models.CASCADE,
related_name='items')
    product = models.ForeignKey(Product, on_delete=models.DO_NOTHING) # Keep
product link for reference
    name = models.CharField(max_length=255) # Store name in case product is
deleted
    price_at_purchase = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.IntegerField(default=1)

    def get_cost(self):
        return self.price_at_purchase * self.quantity
```

Chapter 4: Backend Development with Django

4.1 Project Configuration and Settings

The `settings.py` file is the central configuration hub. Key configurations include:

- **Database Configuration:** `python   # settings.py   DATABASES = {   'default': {   'ENGINE': 'django.db.backends.postgresql',   'NAME': 'ecommerce_db',   'USER': 'ecommerce_user',   'PASSWORD': 'secure_password',   'HOST': 'localhost',   'PORT': '5432',   }   }`
- **Static and Media Files:** `python     # settings.py     STATIC_URL = '/static/'     STATIC_ROOT = BASE_DIR / 'staticfiles' # Used for Nginx to serve static files     MEDIA_URL = '/media/'     MEDIA_ROOT = BASE_DIR / 'media' # Used for user-uploaded files (e.g., product images)`
- **Authentication:** `python     # settings.py     LOGIN_REDIRECT_URL = '/'     LOGOUT_REDIRECT_URL = '/'`

4.2 URL Routing and Namespacing

URL routing is managed hierarchically. The root `urls.py` delegates requests to the respective application's `urls.py` file using the `include()` function, ensuring a clean separation of routes.

Root `urls.py` (`core/urls.py`):

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('store.urls', namespace='store')),
    path('cart/', include('cart.urls', namespace='cart')),
    path('orders/', include('orders.urls', namespace='orders')),
    path('accounts/', include('django.contrib.auth.urls')), # Built-in auth views
]
```

Application `urls.py` (`cart/urls.py`):

```
from django.urls import path
from . import views

app_name = 'cart'
urlpatterns = [
    path('', views.cart_summary, name='cart_summary'),
    path('add/', views.cart_add, name='cart_add'),
    path('delete/', views.cart_delete, name='cart_delete'),
    path('update/', views.cart_update, name='cart_update'),
]
```

4.3 Detailed Model Implementation with Code

The models are the foundation of the application. Beyond the structure, they contain methods that encapsulate business logic, such as calculating totals.

Example: `Cart` Model Manager and Methods The `Cart` model is often managed via a custom manager to handle session-based carts for guests and database-backed carts for logged-in users.

```
# cart/models.py (Expanded)
class Cart(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, null=True,
blank=True)
    session_key = models.CharField(max_length=40, null=True, blank=True)
    date_added = models.DateTimeField(auto_now_add=True)

    def get_total_price(self):
        """Calculates the total price of all items in the cart."""
        return sum(item.get_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.IntegerField(default=1)

    def get_price(self):
        """Calculates the total price for this line item."""
        return self.product.price * self.quantity
```

4.4 Implementing Core View Functions (Code Snippets)

Views handle the core business logic. The `cart_add` view is a critical example of transactional logic.

Conceptual Code Snippet (`cart/views.py` - `cart_add`):

```
from django.shortcuts import get_object_or_404, redirect
from django.views.decorators.http import require_POST
from store.models import Product
from .models import Cart, CartItem

@require_POST
def cart_add(request):
    product_id = request.POST.get('product_id')
    quantity = int(request.POST.get('quantity', 1))
    product = get_object_or_404(Product, id=product_id)

    # 1. Get or create the cart
    if request.user.is_authenticated:
        cart, created = Cart.objects.get_or_create(user=request.user)
    else:
        # Handle session-based cart for guests
        session_key = request.session.session_key
        if not session_key:
            request.session.create()
            session_key = request.session.session_key
        cart, created = Cart.objects.get_or_create(session_key=session_key)

    # 2. Check if item already exists
    try:
        cart_item = CartItem.objects.get(cart=cart, product=product)
        cart_item.quantity += quantity
        cart_item.save()
    except CartItem.DoesNotExist:
        CartItem.objects.create(cart=cart, product=product,
quantity=quantity)

    return redirect('cart:cart_summary')
```

4.5 Form Handling and Validation

Django's `ModelForm` simplifies the creation of forms for models, such as the `Order` model during checkout.

Conceptual Code Snippet (`orders/forms.py`):

```
from django import forms
from .models import Order

class OrderCreateForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'shipping_address']
        widgets = {
            'shipping_address': forms.Textarea(attrs={'rows': 4}),
        }

    def clean_email(self):
        """Custom validation to ensure email is not from a disposable
domain."""
        email = self.cleaned_data.get('email')
        if 'mailinator.com' in email:
            raise forms.ValidationError("Disposable email addresses are not
allowed.")
        return email
```

Chapter 5: Frontend Development and User Experience

5.1 Template Structure and Inheritance

The frontend is built using Django's template language and the inheritance mechanism, which promotes code reuse and consistency.

`base.html` Structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>{% block title %}E-Commerce Store{% endblock %}</title>
    <!-- Bootstrap CSS -->
    <link href="{% static 'css/bootstrap.min.css' %}" rel="stylesheet">
    {% block extra_css %}{% endblock %}
</head>
<body>
    {% include 'includes/navbar.html' %}
    <main class="container mt-4">
        {% block content %}
            <!-- Default content if child template doesn't override -->
        {% endblock %}
    </main>
    {% include 'includes/footer.html' %}
    <!-- Bootstrap JS and jQuery -->
    <script src="{% static 'js/jquery-3.6.0.min.js' %}"></script>
    <script src="{% static 'js/bootstrap.bundle.min.js' %}"></script>
    {% block extra_js %}{% endblock %}
</body>
</html>
```

5.2 Styling with Bootstrap 5 and Custom CSS

Bootstrap 5 is the primary styling framework, utilizing its utility classes and responsive grid system.

Product Listing Grid Implementation: The product list page uses the Bootstrap grid to display products in a responsive card format.

```
<!-- store/product_list.html snippet -->
<div class="row">
    {% for product in products %}
    <div class="col-lg-3 col-md-4 col-sm-6 mb-4">
        <div class="card h-100">
            <img src="{{ product.image.url }}" class="card-img-top" alt="{{
product.name }}">
            <div class="card-body d-flex flex-column">
                <h5 class="card-title">{{ product.name }}</h5>
                <p class="card-text">${{ product.price }}</p>
                <a href="{{ product.get_absolute_url }}" class="btn btn-
```

```
primary mt-auto">View Details</a>
            </div>
        </div>
    </div>
    {% endfor %}
</div>
```

5.3 Enhancing Interactivity with JavaScript and AJAX

AJAX is used to provide a modern, non-blocking user experience, particularly for cart operations.

AJAX Cart Update Implementation: When a user changes the quantity in the cart summary, an AJAX request is sent to the `cart_update` view.

```
// Custom JavaScript for Cart Update
$(document).on('change', '.cart-quantity-input', function(e){
    e.preventDefault();
    var product_id = $(this).data('product-id');
    var new_quantity = $(this).val();
    var csrf_token = $('input[name="csrfmiddlewaretoken"]').val();

    $.ajax({
        type: 'POST',
        url: '{% url "cart:cart_update" %}',
        data: {
            product_id: product_id,
            quantity: new_quantity,
            csrfmiddlewaretoken: csrf_token,
            action: 'update'
        },
        success: function(json){
            // Update the total price and subtotal dynamically
            $('#cart-total').text(json.total);
            $('#item-subtotal-' + product_id).text(json.subtotal);
        },
        error: function(xhr, errmsg, err){
            console.log(xhr.status + ": " + xhr.responseText);
        }
    });
});
```

5.4 Responsive Design Implementation

The mobile-first approach of Bootstrap ensures that the website is fully functional and visually appealing on all devices.

Responsive Design Techniques Used: * **Breakpoint Utilization:** Using classes like `col-lg-3`, `col-md-4`, and `col-sm-6` to define different column widths at various screen sizes. * **Utility Classes:** Employing `d-flex`, `mt-auto`, and `text-center` for flexible alignment and spacing that adapts to the viewport. * **Viewport Meta Tag:** Ensuring the correct scaling and rendering on mobile devices: `<meta name="viewport" content="width=device-width, initial-scale=1">`.

Chapter 6: Testing and Quality Assurance

A comprehensive testing strategy is essential to ensure the reliability and stability of the e-commerce platform. The project utilizes Django's built-in testing framework, which is based on Python's `unittest` module.

6.1 Comprehensive Testing Strategy

Testing is structured into three main categories:

1. **Unit Tests:** Focus on the smallest testable parts of the application, primarily model methods and utility functions, ensuring they perform their specific tasks correctly.
2. **Integration Tests:** Verify that different parts of the system work together as expected, such as a view function correctly interacting with its corresponding model.
3. **Functional/End-to-End Tests:** Simulate real user scenarios (e.g., browsing, adding to cart, checkout) to ensure the entire application flow is correct.

6.2 Unit Testing Examples (Models and Utilities)

Unit tests are written to validate the business logic encapsulated within the models.

Conceptual Code Snippet (`store/tests.py` - Model Test):

```
from django.test import TestCase
from store.models import Category, Product
from decimal import Decimal

class ProductModelTest(TestCase):
    def setUp(self):
        self.category = Category.objects.create(name='Electronics',
slug='electronics')
        self.product = Product.objects.create(
            category=self.category,
            name='Laptop',
            slug='laptop',
            price=Decimal('1200.00'),
            stock=10,
            is_available=True
        )

    def test_product_creation(self):
        """Test that a product is created correctly."""
        self.assertEqual(self.product.name, 'Laptop')
        self.assertTrue(self.product.is_available)
        self.assertEqual(self.product.stock, 10)

    def test_product_price_calculation(self):
        """Test that the price field is a Decimal and correct."""
        self.assertIsInstance(self.product.price, Decimal)
        self.assertEqual(self.product.price, Decimal('1200.00'))
```

6.3 Integration and Functional Testing (Checkout Process)

Functional tests use Django's `Client` to simulate HTTP requests and verify the entire transaction flow, including session management and database changes.

Conceptual Code Snippet (`orders/tests.py` - Functional Test):

```
from django.test import TestCase, Client
from django.urls import reverse
from store.models import Product
from orders.models import Order, OrderItem

class CheckoutProcessTest(TestCase):
    def setUp(self):
        self.client = Client()
        # Setup a product and add it to a session-based cart
        self.product = Product.objects.create(name='Test Item', slug='test-
item', price=Decimal('10.00'), stock=5)
        session = self.client.session
        session['cart'] = {
            str(self.product.id): {'quantity': 2, 'price':
str(self.product.price)}
        }
        session.save()

    def test_order_creation_success(self):
        """Test that a successful POST to checkout creates an Order and
OrderItems."""
        response = self.client.post(reverse('orders:order_create'), {
            'first_name': 'John',
            'last_name': 'Doe',
            'email': 'john.doe@example.com',
            'shipping_address': '123 Test St',
        }, follow=True)

        # Check that the order was created
        self.assertEqual(Order.objects.count(), 1)
        order = Order.objects.first()
        self.assertEqual(order.total_amount, Decimal('20.00'))

        # Check that the order item was created correctly
        self.assertEqual(OrderItem.objects.count(), 1)
        order_item = OrderItem.objects.first()
        self.assertEqual(order_item.product, self.product)
        self.assertEqual(order_item.quantity, 2)
        self.assertEqual(order_item.price_at_purchase, Decimal('10.00'))

        # Check that the cart is cleared
        self.assertNotIn('cart', self.client.session)
```

6.4 Code Review and Quality Metrics

Code quality is maintained through adherence to the **PEP 8** style guide and regular code reviews. Key quality metrics tracked include: * **Test Coverage:** Aiming for a minimum of 80% line coverage for all core business logic. * **Cyclomatic Complexity:** Keeping the complexity of view functions low to ensure readability and maintainability. * **Code Duplication:** Utilizing Django’s mixins and template inheritance to minimize redundant code.

Chapter 7: Deployment and Maintenance

The deployment process transforms the development environment into a production-ready system capable of handling real-world traffic and data.

7.1 Production Environment Setup (PostgreSQL, Gunicorn, Nginx)

The production stack is composed of three main components:

1. **PostgreSQL (Database):** Provides reliable, concurrent data storage.
2. **Gunicorn (WSGI Server):** Serves the Django application, managing Python processes and threads.
3. **Nginx (Reverse Proxy):** Sits in front of Gunicorn, handling client requests, serving static/media files directly, and forwarding dynamic requests to Gunicorn.

Deployment Component Roles: | Component | Role | Configuration Detail | | :— | :— | :— | | **PostgreSQL** | Data Persistence | Configured with appropriate user roles and permissions for the Django application. | | **Gunicorn** | Application Server | Bound to a local socket (e.g., `/tmp/gunicorn.sock`) and configured with multiple worker processes. | | **Nginx** | Web Server/Proxy | Listens on port 80/443, serves static files from `STATIC_ROOT`, and proxies dynamic requests to the Gunicorn socket. |

7.2 Step-by-Step Deployment Guide

The following steps outline the process for deploying the application to a Linux server:

Step 1: Server Preparation

```
# Install necessary packages
sudo apt update
sudo apt install python3-pip python3-dev libpq-dev nginx
# Install PostgreSQL and create database/user
sudo apt install postgresql postgresql-contrib
sudo -u postgres createuser --interactive
sudo -u postgres createdb ecommerce_db
```

Step 2: Project Setup and Dependencies

```
# Clone the repository and create a virtual environment
git clone <repository_url> /var/www/ecommerce_project
cd /var/www/ecommerce_project
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Step 3: Database Migration and Static Files

```
# Apply database migrations
python manage.py migrate
```

```
# Collect static files for Nginx
python manage.py collectstatic
```

Step 4: Gunicorn Configuration Create a Gunicorn service file (/etc/systemd/system/gunicorn.service):

```
[Unit]
Description=gunicorn daemon for ecommerce_project
After=network.target

[Service]
User=ubuntu
Group=www-data
WorkingDirectory=/var/www/ecommerce_project
ExecStart=/var/www/ecommerce_project/venv/bin/gunicorn --workers 3 --bind
unix:/tmp/gunicorn.sock core.wsgi:application

[Install]
WantedBy=multi-user.target
```

Enable and start the Gunicorn service:

```
sudo systemctl start gunicorn
sudo systemctl enable gunicorn
```

Step 5: Nginx Configuration Create an Nginx configuration file (/etc/nginx/sites-available/ecommerce_project):

```
server {
    listen 80;
    server_name your_domain.com;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        alias /var/www/ecommerce_project/staticfiles/;
    }
    location /media/ {
        alias /var/www/ecommerce_project/media/;
    }

    location / {
        include proxy_params;
        proxy_pass http://unix:/tmp/gunicorn.sock;
    }
}
```

Link the configuration and restart Nginx:

```
sudo ln -s /etc/nginx/sites-available/ecommerce_project /etc/nginx/sites-
enabled
```

```
sudo nginx -t
sudo systemctl restart nginx
```

7.3 Monitoring and Logging Strategy

Effective monitoring is crucial for identifying and resolving production issues.

- **Application Logging:** Django's built-in logging framework is configured to write application errors, warnings, and informational messages to a dedicated log file (`/var/log/ecommerce/app.log`).
- **Server Logging:** Nginx and Gunicorn logs are monitored for access patterns and server-level errors.
- **Error Reporting:** Integration with an external error tracking service (e.g., Sentry) is recommended to capture and aggregate exceptions in real-time.

Chapter 8: Results, Achievements, and Future Enhancements

8.1 Functional Completeness and Technical Achievements

The project successfully delivered a fully functional e-commerce platform, validating the chosen MVT architecture and technology stack.

Key Technical Achievements: * **Transactional Atomicity:** Successfully implemented the order creation process, ensuring that stock is reduced and order items are recorded atomically upon successful checkout. * **Separation of Concerns:** Maintained a clean, modular codebase by strictly separating logic into dedicated Django applications (`store`, `cart`, `orders`). * **Security Implementation:** Demonstrated proficiency in securing a web application by utilizing Django's built-in CSRF, XSS, and password hashing mechanisms. * **Scalable Deployment:** Established a production-ready deployment configuration using Gunicorn and Nginx, which can be easily scaled horizontally.

8.2 Business Value and Key Performance Indicators (KPIs)

The project provides significant business value by establishing a digital storefront. Key performance indicators (KPIs) for the platform include:

KPI	Description	Measurement Method	Target Goal
Conversion Rate	Percentage of visitors who complete a purchase.	(Orders / Sessions) * 100	> 2.5%
Average Order Value (AOV)	Average total value of all orders placed.	Total Revenue / Number of Orders	Increase AOV by 10% QoQ
Page Load Time	Time taken for a page to fully load.	Browser Developer Tools / Monitoring	< 3 seconds
Cart Abandonment Rate	Percentage of carts created that do not result in a purchase.	(Carts Created - Orders Placed) / Carts Created	< 60%

8.3 Future Enhancements Roadmap

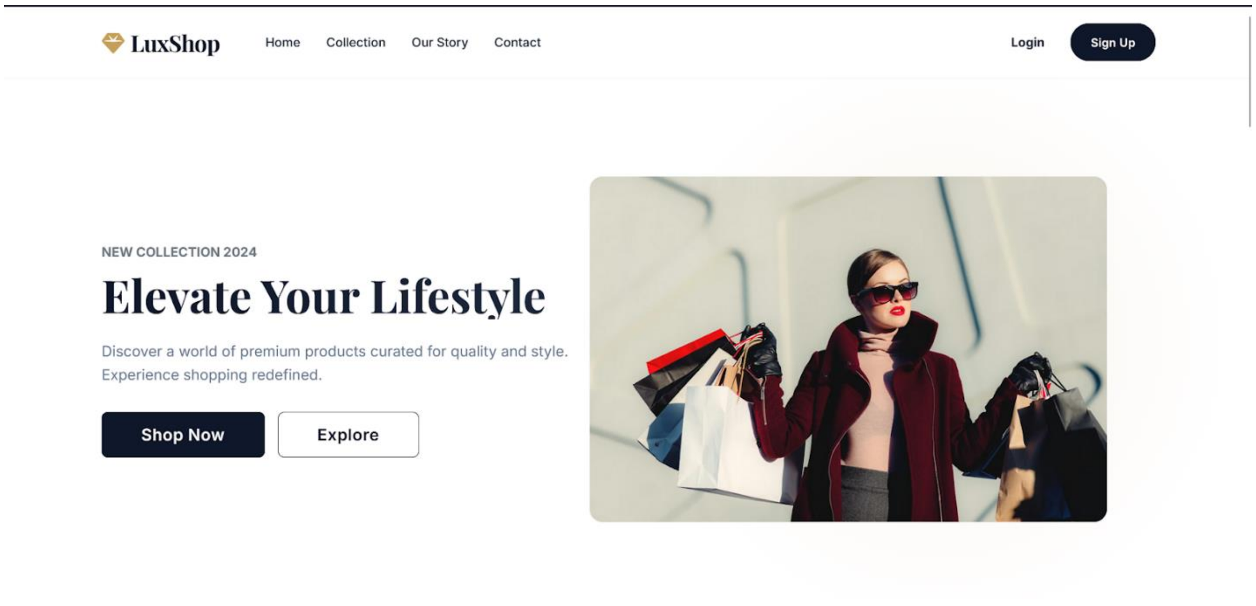
To evolve the platform into a commercial-grade application, the following features are planned for future development, prioritized by business impact and technical complexity.

Feature	Description	Priority	Estimated Effort
Payment Gateway Integration	Integrate with a third-party service (e.g., Stripe, PayPal) to handle real-time, secure credit card transactions.	High	4 Weeks
Advanced Search	Implement a dedicated search engine (e.g., Elasticsearch or PostgreSQL's full-text search) for faster and more relevant product results.	Medium	3 Weeks

User Reviews and Ratings	Add models and views to allow customers to submit product reviews and view aggregate ratings.	Medium	2 Weeks
Inventory Alerts	Implement a system to notify administrators when product stock levels fall below a predefined threshold.	Low	1 Week
Automated Email Notifications	Set up automated email sending for order confirmation, shipping updates, and password resets using a service like SendGrid or Mailgun.	High	2 Weeks
Caching Implementation	Introduce Redis or Memcached for caching frequently accessed data (e.g., product lists) to improve performance.	Medium	1 Week

Snapshot of Front-end:

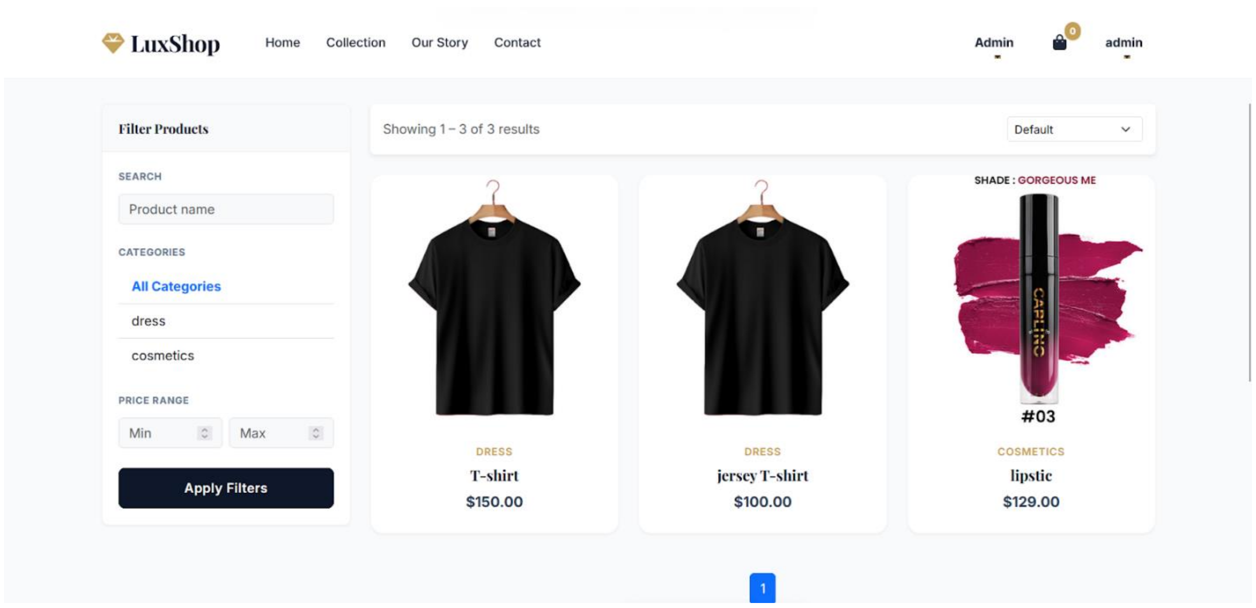
Home Page:



NEW COLLECTION 2024

Elevate Your Lifestyle

Discover a world of premium products curated for quality and style. Experience shopping redefined.


LuxShop Home Collection Our Story Contact Admin 0 admin

Filter Products

SEARCH

Product name

CATEGORIES

All Categories

dress

cosmetics

PRICE RANGE

Min Max

Apply Filters

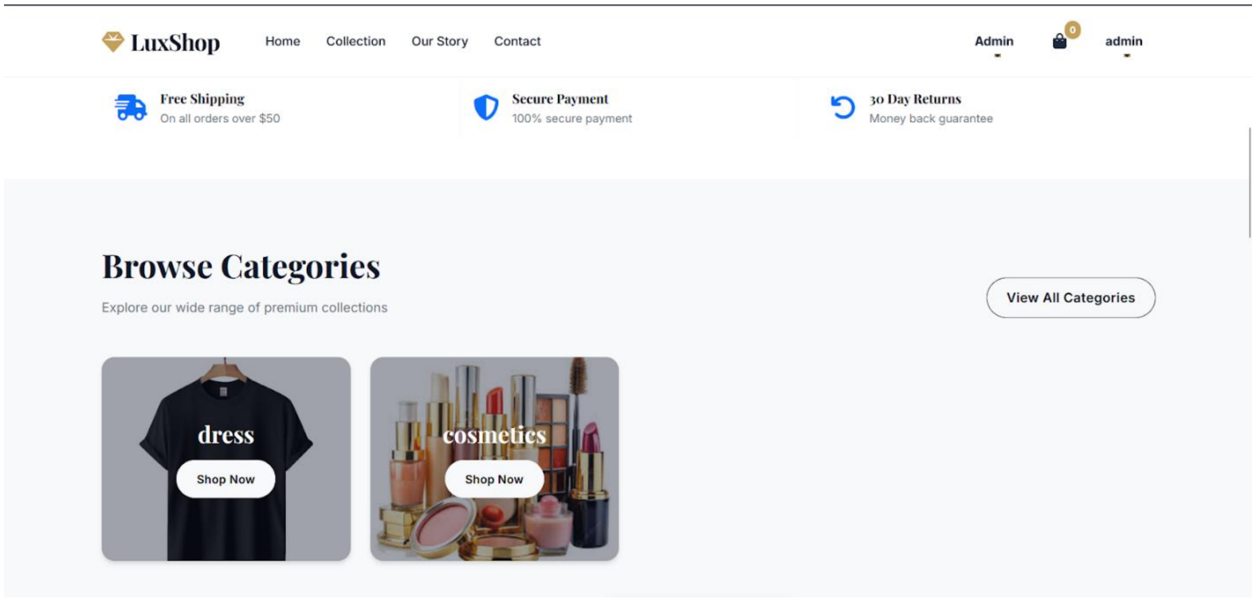
Showing 1 – 3 of 3 results Default

DRESS
T-shirt
$150.00

DRESS
jersey T-shirt
$100.00

COSMETICS
lipstic
$129.00

1


LuxShop Home Collection Our Story Contact Admin 0 admin

Free Shipping
On all orders over $50

Secure Payment
100% secure payment

30 Day Returns
Money back guarantee

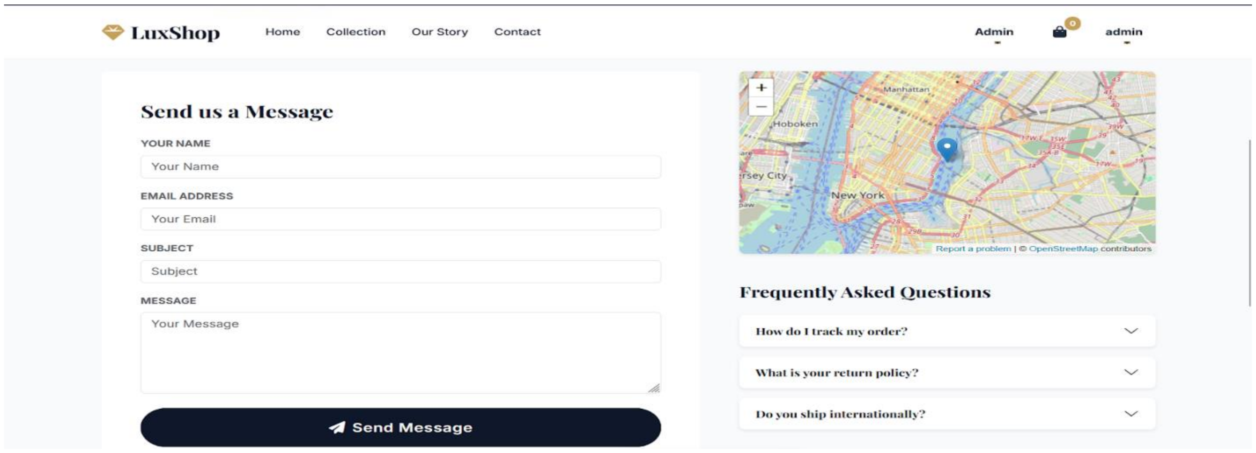
Browse Categories

Explore our wide range of premium collections

View All Categories

dress
Shop Now

cosmetics
Shop Now



Send us a Message

YOUR NAME

Your Name

EMAIL ADDRESS

Your Email

SUBJECT

Subject

MESSAGE

Your Message

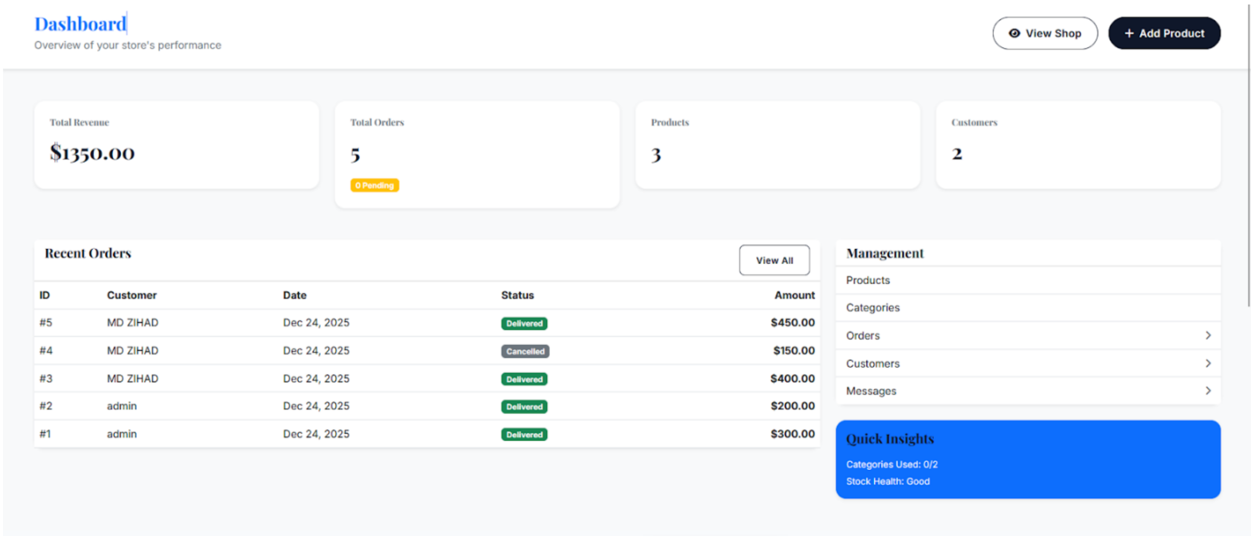
Frequently Asked Questions

How do I track my order?

What is your return policy?

Do you ship internationally?

Admin Dashboard:



Dashboard

Overview of your store's performance

Total Revenue	Total Orders	Products	Customers
$1350.00	5 (0 Pending)	3	2

Recent Orders

ID	Customer	Date	Status	Amount
#5	MD ZIHAD	Dec 24, 2025	Delivered	$450.00
#4	MD ZIHAD	Dec 24, 2025	Cancelled	$150.00
#3	MD ZIHAD	Dec 24, 2025	Delivered	$400.00
#2	admin	Dec 24, 2025	Delivered	$200.00
#1	admin	Dec 24, 2025	Delivered	$300.00

Management

Products

Categories

Orders

Customers

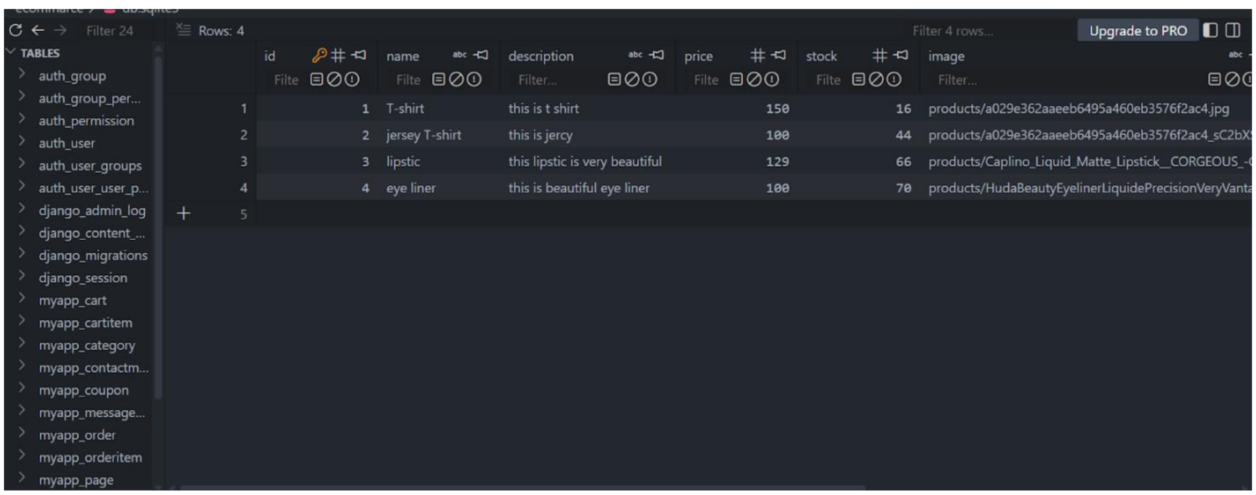
Messages

Quick Insights

Categories Used: 0/2

Stock Health: Good

Sqlite:



auth_group
auth_group_per...
auth_permission
auth_user
auth_user_groups
auth_user_user_p...
django_admin_log
django_content_...
django_migrations
django_session
myapp_cart
myapp_cartitem
myapp_category
myapp_contactm...
myapp_coupon
myapp_message...
myapp_order
myapp_orderitem
myapp_page

	id	name	description	price	stock	image
1	1	T-shirt	this is t shirt	150	16	products/a029e362aaeeb6495a460eb3576f2ac4.jpg
2	2	jersey T-shirt	this is jercy	100	44	products/a029e362aaeeb6495a460eb3576f2ac4_sC2bX
3	3	lipstic	this lipstic is very beautiful	129	66	products/Caplino_Liquid_Matte_Lipstick__CORGEOUS_-
4	4	eye liner	this is beautiful eye liner	100	70	products/HudaBeautyEyelinerLiquidePrecisionVeryVanta
5						

References

1. Django Documentation - The official documentation for the Django web framework, detailing the MVT pattern, ORM, and security features.
2. Bootstrap 5 Documentation - Official guide for the frontend framework, including responsive design and component usage.
3. PostgreSQL Official Website - Reference for advanced database features, data types, and production best practices.
4. Gunicorn Documentation - Guide for deploying Django applications in a production environment using the WSGI protocol.
5. Nginx Documentation - Reference for configuring the reverse proxy, static file serving, and load balancing.
6. PEP 8 – Style Guide for Python Code - The official style guide for Python code, used to ensure code readability and consistency.