Table of Contents

Prerequisites

Before we begin, ensure you have the following installed on your machine:

- **Python** (preferably Python 3.8 or higher)
- **pip** (Python package installer)
- **Git** (version control system, optional but recommended)
- **Virtual Environment Tools** (venv or `virtualenv`)
- **Basic Knowledge** of Python, HTML, CSS, and JavaScript
- **Understanding of Django's MVC Architecture** (Models, Views, Templates)

Project Setup

1. Install Django and Create a Virtual Environment

It's best practice to create a virtual environment to manage project dependencies.

```
# Navigate to your desired directory
cd ~/Projects/

# Create a virtual environment named 'env'
python -m venv env

# Activate the virtual environment
# On macOS/Linux:
source env/bin/activate
```

```
# On Windows:
env\Scripts\activate

# Upgrade pip
pip install --upgrade pip

# Install Django and other essential packages
pip install django Pillow django-crispy-forms django-filter stripe
```

Explanation:

- **Virtual Environment (venv)**: Isolates project dependencies, preventing version conflicts.
- **Pillow**: Handles image uploads.
- **django-crispy-forms**: Enhances form rendering with Bootstrap styling.
- **django-filter**: Simplifies filtering querysets based on user input.
- **stripe**: Integrates Stripe for payment processing.

2. Create a New Django Project

```
# Create the Django project named 'm2collection'
django-admin startproject m2collection

# Navigate into the project directory
cd m2collection
```

This creates the following structure:

```
m2collection/
    manage.py
    m2collection/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Creating Django Apps

We'll create three apps to modularize our project:

1. **Store**: Handles product listings and details.
2. **Cart**: Manages shopping cart functionality.
3. **Orders**: Processes orders and payments.

```
# Create the 'store' app
python manage.py startapp store
```

```
# Create the 'cart' app
python manage.py startapp cart

# Create the 'orders' app
python manage.py startapp orders
```

Project Structure Now:

```
m2collection/
    manage.py
    m2collection/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    store/
        __init__.py
        admin.py
        apps.py
        migrations/
            __init__.py
        models.py
        tests.py
        views.py
    cart/
        __init__.py
        admin.py
        apps.py
        migrations/
            __init__.py
        models.py
        tests.py
        views.py
    orders/
        __init__.py
        admin.py
        apps.py
        migrations/
            __init__.py
        models.py
        tests.py
        views.py
```

3. Add Apps to `INSTALLED_APPS`

Open `m2collection/settings.py` and add the newly created apps to the `INSTALLED_APPS` list along with third-party apps.

```
# m2collection/settings.py

INSTALLED_APPS = [
    # Default Django apps...
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Third-party apps
    'crispy_forms',  # For better form rendering
    'django_filters',  # For filtering
    'store',
    'cart',
    'orders',
]
```

Explanation:

- **crispy_forms**: Enhances the appearance and usability of forms.
- **django_filters**: Facilitates easy filtering of products.

Defining Models

We'll define models for our applications to represent the data structure of our e-commerce platform.

1. Store App Models

a. `Category` Model

Represents product categories.

```
# store/models.py

from django.db import models

class Category(models.Model):
    """
    Represents a product category (e.g., Shirts, Pants).
    """
    name = models.CharField(max_length=200, unique=True)
```

```
    slug = models.SlugField(max_length=200, unique=True)

    class Meta:
        ordering = ['name']
        verbose_name_plural = 'Categories'  # Corrects the plural
name in admin

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

Explanation:

- **name**: Name of the category.
- **slug**: URL-friendly representation of the category name.
- **unique=True**: Ensures no duplicate categories.

b. Product Model

Represents individual products.

```
# store/models.py

from django.db import models
from django.urls import reverse

class Product(models.Model):
    """
    Represents a product in the store.
    """
    category = models.ForeignKey(Category, related_name='products',
on_delete=models.CASCADE)
    name = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    image = models.ImageField(upload_to='products/%Y/%m/%d',
blank=True)
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)  #
Automatically set on creation
    updated = models.DateTimeField(auto_now=True)      #
Automatically updated on save

    class Meta:
        ordering = ['-created']
        index_together = (('id', 'slug'),)  # Speeds up queries
```

```
involving both id and slug

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        """
        Returns the URL to access a particular product instance.
        """
        return reverse('store:product_detail', args=[self.id,
self.slug])
```

Explanation:

- **category**: Associates the product with a category.
- **name**: Product name.
- **slug**: URL-friendly name.
- **price**: Product price.
- **image**: Optional product image.
- **available**: Indicates if the product is in stock.
- **created & updated**: Timestamps for product creation and updates.
- **get_absolute_url**: Provides a way to retrieve the URL for a product detail view.

c. Apply Migrations

```
# Make migrations for 'store' app
python manage.py makemigrations store

# Apply migrations
python manage.py migrate
```

2. Cart App Models

Our cart will be session-based, so we don't need to create persistent models. However, we can create a `CartItem` model if we want to track items over sessions. For simplicity, we'll manage the cart using session data.

3. Orders App Models

a. `Order` Model

Represents customer orders.

```
# orders/models.py

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

```python
class Order(models.Model):
    """
    Represents a customer's order.
    """
    user = models.ForeignKey(User, related_name='orders',
on_delete=models.CASCADE)
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    address = models.CharField(max_length=250)
    postal_code = models.CharField(max_length=20)
    city = models.CharField(max_length=100)
    created = models.DateTimeField(default=timezone.now)
    updated = models.DateTimeField(auto_now=True)
    paid = models.BooleanField(default=False)  # Indicates if the
order has been paid
    stripe_charge_id = models.CharField(max_length=255, blank=True)
# Stores Stripe charge ID

    class Meta:
        ordering = ['-created']

    def __str__(self):
        return f'Order {self.id}'

    def get_total_cost(self):
        """
        Calculates the total cost of the order by summing the cost
of each item.
        """
        return sum(item.get_cost() for item in self.items.all())

class OrderItem(models.Model):
    """
    Represents an individual item within an order.
    """
    order = models.ForeignKey(Order, related_name='items',
on_delete=models.CASCADE)
    product = models.ForeignKey(Product, related_name='order_items',
on_delete=models.CASCADE)
    price = models.DecimalField(max_digits=10, decimal_places=2)  #
```

```
Price at the time of order
    quantity = models.PositiveIntegerField(default=1)

    def __str__(self):
        return f'{self.id}'

    def get_cost(self):
        """
        Calculates the cost of the item based on price and quantity.
        """
        return self.price * self.quantity
```

Explanation:

- **Order**:
  - **user**: Associates the order with the user who placed it.
  - **paid**: Indicates whether the order has been paid.
  - **stripe_charge_id**: Stores the Stripe charge identifier for payment tracking.
  - **get_total_cost**: Computes the total cost of the order.
- **OrderItem**:
  - **order**: Associates the item with an order.
  - **product**: References the purchased product.
  - **price**: Captures the product's price at the time of the order.
  - **quantity**: Number of units purchased.
  - **get_cost**: Computes the total cost for this item.

b. Apply Migrations

```
# Make migrations for 'orders' app
python manage.py makemigrations orders

# Apply migrations
python manage.py migrate
```

Setting Up the Admin Interface

Django's admin interface allows for easy management of models.

1. Register Models in Admin

a. Store App Admin

```
# store/admin.py

from django.contrib import admin
from .models import Category, Product

@admin.register(Category)
```

```python
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}  # Automatically
populate slug from name
    search_fields = ['name']

@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price', 'available', 'created',
'updated']
    list_filter = ['available', 'created', 'updated', 'category']
    list_editable = ['price', 'available']
    prepopulated_fields = {'slug': ('name',)}
    search_fields = ['name', 'description']
```

Explanation:

- **prepopulated_fields**: Automatically fills slug based on the name.
- **list_display**: Columns shown in the admin list view.
- **list_filter**: Adds filter options in the sidebar.
- **list_editable**: Makes specified fields editable directly from the list view.

b. Orders App Admin

```python
# orders/admin.py

from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    """
    Inline display of order items within the order admin.
    """
    model = OrderItem
    raw_id_fields = ['product']

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'user', 'first_name', 'last_name',
'email', 'paid', 'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]
    search_fields = ['id', 'first_name', 'last_name', 'email']
```

Explanation:

- **OrderItemInline**: Allows for editing order items within the order admin page.
- **list_display**: Shows key order details in the list view.
- **list_filter & search_fields**: Enhances data retrieval in the admin.

2. Create a Superuser

To access the admin interface, create a superuser account.

```
python manage.py createsuperuser
```

Follow the prompts to set up the superuser credentials.

3. Access the Admin Site

Start the development server and navigate to http://127.0.0.1:8000/admin/ in your browser. Log in using the superuser credentials to manage categories, products, and orders.

```
python manage.py runserver
```

User Authentication

Implementing user authentication ensures that only registered users can perform specific actions, such as adding items to the cart or placing orders.

1. Utilize Django's Built-in Authentication System

Django provides robust authentication features out of the box.

a. Include Authentication URLs

In `m2collection/urls.py`, include Django's authentication URLs.

```
# m2collection/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('store/', 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')),  # Adds
login, logout, password management URLs
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
```

```
document_root=settings.MEDIA_ROOT)
```

Explanation:

- **include('django.contrib.auth.urls')**: Adds URLs for login, logout, password reset, etc.
- **path Conventions**:
  - `/store/`: Access store-related views.
  - `/cart/`: Access cart-related views.
  - `/orders/`: Access order-related views.
  - `/accounts/`: Access authentication-related views.

b. Create Signup View and Template

Django doesn't provide a signup view by default, so we'll create one.

i. Create a `SignupForm`

```
# store/forms.py

from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

class SignupForm(UserCreationForm):
    """
    Extends UserCreationForm to include email.
    """
    email = forms.EmailField(max_length=254, required=True,
help_text='Required. Enter a valid email address.')

    class Meta:
        model = User
        fields = ('username', 'email', 'password1', 'password2', )
```

Explanation:

- **email**: Adds an email field to the default signup form.
- **Meta.fields**: Specifies the order and inclusion of form fields.

ii. Create a Signup View

```
# store/views.py

from django.shortcuts import render, redirect
from django.contrib.auth import login, authenticate
from .forms import SignupForm
from django.contrib import messages

def signup(request):
```

```
    """
    Handles user registration.
    """
    if request.method == 'POST':
        form = SignupForm(request.POST)
        if form.is_valid():
            user = form.save()
            # Authenticate and log in the user
            username = form.cleaned_data.get('username')
            raw_password = form.cleaned_data.get('password1')
            user = authenticate(username=username,
password=raw_password)
            login(request, user)
            messages.success(request, 'Registration successful.')
            return redirect('store:product_list')
        else:
            messages.error(request, 'Please correct the errors
below.')
    else:
        form = SignupForm()
    return render(request, 'store/signup.html', {'form': form})
```

Explanation:

- **POST Request**: Processes form submission, validates data, creates a new user, authenticates, logs them in, and redirects to the product list.
- **GET Request**: Displays the empty signup form.
- **Messages**: Provides feedback to the user about the success or failure of the registration.

iii. Create Signup URL

```
# store/urls.py

from django.urls import path
from . import views

app_name = 'store'

urlpatterns = [
    path('', views.product_list, name='product_list'),
    path('signup/', views.signup, name='signup'),  # Signup URL
    path('product/<int:id>/<slug:slug>/', views.product_detail,
name='product_detail'),
```

```
]
```

Explanation:

- **app_name**: Namespaces the store app URLs, preventing clashes with other apps.
- **'signup/'**: URL for the signup view.

iv. Create Signup Template

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

{% extends 'store/base.html' %}

{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <h2>Sign Up</h2>
    <form method="post">
        {% csrf_token %}
        {{ form|crispy }}
        <button type="submit" class="btn btn-
primary">Register</button>
    </form>
</div>
{% endblock %}
```

Explanation:

- **{% load crispy_forms_tags %}**: Loads crispy forms template tags.
- **{{ form|crispy }}**: Renders the form using crispy forms for better styling.

c. Create Login Template

Django's authentication views expect specific templates to be in place. We'll create a custom login template.

```
<!-- store/templates/registration/login.html -->

{% extends 'store/base.html' %}

{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <h2>Login</h2>
    <form method="post">
        {% csrf_token %}
```

```
        {{ form|crispy }}
        <button type="submit" class="btn btn-primary">Login</button>
    </form>
    <p class="mt-3">
        Don't have an account? <a href="{% url
'store:signup' %}">Sign up here</a>.
    </p>
</div>
{% endblock %}
```

Explanation:

- **Custom Path**: Ensure the template is located at `templates/registration/login.html` as Django expects this path for the login view.
- **Link to Signup**: Provides navigation to the signup page for new users.

Product Listing with Search and Filter

Implementing product search and filtering enhances user experience by allowing users to find products easily.

1. Define Views for Product Listing and Detail

a. `product_list` View

```
# store/views.py

from .models import Product, Category
from django.core.paginator import Paginator, EmptyPage,
PageNotAnInteger
from django.shortcuts import render
from django.db.models import Q
from .forms import SignupForm
from django.contrib import messages

def product_list(request):
    """
    Displays a list of available products with search and category
filter functionality.
    """
    products = Product.objects.filter(available=True)
    categories = Category.objects.all()

    # Search functionality
    query = request.GET.get('q')
```

```
    if query:
        products = products.filter(
            Q(name__icontains=query) |
Q(description__icontains=query)
        )

    # Category filter
    category_slug = request.GET.get('category')
    if category_slug:
        products = products.filter(category__slug=category_slug)

    # Pagination
    paginator = Paginator(products, 6)  # Show 6 products per page
    page = request.GET.get('page')
    try:
        products_paginated = paginator.page(page)
    except PageNotAnInteger:
        products_paginated = paginator.page(1)
    except EmptyPage:
        products_paginated = paginator.page(paginator.num_pages)

    context = {
        'products': products_paginated,
        'categories': categories,
        'query': query,
        'selected_category': category_slug,
    }
    return render(request, 'store/product_list.html', context)
```

Explanation:

- **Search (q)**: Filters products based on the query matching the name or description.
- **Category (category)**: Filters products belonging to a selected category.
- **Pagination**: Splits the product list into pages, displaying 6 products per page.

b. `product_detail` View

```
# store/views.py

from django.shortcuts import render, get_object_or_404
from .models import Product

def product_detail(request, id, slug):
    """
    Displays detailed information about a specific product.
```

```
    """
    product = get_object_or_404(Product, id=id, slug=slug,
available=True)
    return render(request, 'store/product_detail.html', {'product':
product})
```

Explanation:

- **get_object_or_404**: Retrieves the product or returns a 404 error if not found or unavailable.

2. Create `product_list` and `product_detail` Templates

a. `product_list.html`

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

{% extends 'store/base.html' %}
{% load static %}
{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <h2>Products</h2>

    <!-- Search Form -->
    <form method="get" class="d-flex mb-4">
        <input type="text" name="q" class="form-control me-2"
placeholder="Search products..." value="{{ query }}">
        <button type="submit" class="btn btn-outline-
success">Search</button>
    </form>

    <!-- Category Filters -->
    <div class="mb-4">
        <h5>Filter by Category</h5>
        <ul class="list-inline">
            <li class="list-inline-item">
                <a href="{% url 'store:product_list' %}" class="btn
btn-sm {% if not selected_category %}btn-primary{% else %}btn-
outline-primary{% endif %}">All</a>
            </li>
            {% for category in categories %}
                <li class="list-inline-item">
                    <a href="?category={{ category.slug }}"
class="btn btn-sm {% if selected_category == category.slug %}btn-
```

```
primary{% else %}btn-outline-primary{%
endif %}">{{ category.name }}</a>
                </li>
            {% endfor %}
        </ul>
    </div>

    <!-- Product Grid -->
    <div class="row">
        {% for product in products %}
            <div class="col-md-4 mb-4">
                <div class="card h-100">
                    {% if product.image %}
                        <img src="{{ product.image.url }}"
class="card-img-top" alt="{{ product.name }}">
                    {% else %}
                        <img src="{% static
'store/images/no_image_available.png' %}" class="card-img-top"
alt="No image">
                    {% endif %}
                    <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="mt-auto btn btn-primary">View Details</a>
                    </div>
                </div>
            </div>
        {% empty %}
            <p>No products found.</p>
        {% endfor %}
    </div>

    <!-- Pagination Controls -->
    <nav aria-label="Product pagination">
        <ul class="pagination justify-content-center">
            {% if products.has_previous %}
                <li class="page-item">
                    <a class="page-link"
href="?page={{ products.previous_page_number }}{% if
```

```
query %}&q={{ query }}{% endif %}{% if
selected_category %}&category={{ selected_category }}{% endif %}"
aria-label="Previous">
                        <span aria-hidden="true">&laquo;</span>
                    </a>
                </li>
            {% else %}
                <li class="page-item disabled">
                    <span class="page-link" aria-label="Previous">
                        <span aria-hidden="true">&laquo;</span>
                    </span>
                </li>
            {% endif %}

            {% for num in products.paginator.page_range %}
                {% if products.number == num %}
                    <li class="page-item active"><span class="page-
link">{{ num }}</span></li>
                {% elif num > products.number|add:'-3' and num <
products.number|add:'3' %}
                    <li class="page-item"><a class="page-link"
href="?page={{ num }}{% if query %}&q={{ query }}{% endif %}{% if
selected_category %}&category={{ selected_category }}{%
endif %}">{{ num }}</a></li>
                {% endif %}
            {% endfor %}

            {% if products.has_next %}
                <li class="page-item">
                    <a class="page-link"
href="?page={{ products.next_page_number }}{% if
query %}&q={{ query }}{% endif %}{% if
selected_category %}&category={{ selected_category }}{% endif %}"
aria-label="Next">
                        <span aria-hidden="true">&raquo;</span>
                    </a>
                </li>
            {% else %}
                <li class="page-item disabled">
                    <span class="page-link" aria-label="Next">
                        <span aria-hidden="true">&raquo;</span>
                    </span>
```

```
                </li>
            {% endif %}
        </ul>
    </nav>
</div>
{% endblock %}
```

Explanation:

- **Search Form**: Users can search for products by name or description.
- **Category Filters**: Allows users to filter products by category.
- **Product Grid**: Displays products in a responsive grid layout using Bootstrap cards.
- **Pagination Controls**: Navigates through multiple pages of products.

b. product_detail.html

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

{% extends 'store/base.html' %}
{% load static %}
{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <div class="row">
        <!-- Product Image -->
        <div class="col-md-6">
            {% if product.image %}
                <img src="{{ product.image.url }}" class="img-fluid"
alt="{{ product.name }}">
            {% else %}
                <img src="{% static
'store/images/no_image_available.png' %}" class="img-fluid" alt="No
image">
            {% endif %}
        </div>

        <!-- Product Details -->
        <div class="col-md-6">
            <h2>{{ product.name }}</h2>
            <p class="text-muted">${{ product.price }}</p>
            <p>{{ product.description }}</p>

            <!-- Add to Cart Form -->
            <form method="post" action="{% url 'cart:cart_add'
```

```
product.id %}">
                {% csrf_token %}
                <div class="mb-3">
                    <label for="quantity" class="form-
label">Quantity:</label>
                    <input type="number" name="quantity" value="1"
min="1" class="form-control" id="quantity">
                </div>
                <button type="submit" class="btn btn-primary">Add to
Cart</button>
            </form>
        </div>
    </div>

    <!-- Back to Products -->
    <a href="{% url 'store:product_list' %}" class="btn btn-
secondary mt-4">Back to Products</a>
</div>
{% endblock %}
```

Explanation:

- **Product Image**: Displays the product image or a placeholder if unavailable.
- **Product Details**: Includes name, price, and description.
- **Add to Cart Form**: Allows users to add the product to their cart with a specified quantity.

3. Create Store App URLs

```
# store/urls.py

from django.urls import path
from . import views

app_name = 'store'

urlpatterns = [
    path('', views.product_list, name='product_list'),  # /store/
    path('signup/', views.signup, name='signup'),        #
/store/signup/
    path('product/<int:id>/<slug:slug>/', views.product_detail,
name='product_detail'),  # /store/product/1/product-name/
]
```

Explanation:

- **app_name**: Namespaces URLs to avoid clashes with other apps.
- **URL Patterns**:
    - `/store/`: Product listing page.
    - `/store/signup/`: User registration page.
    - `/store/product/<id>/<slug>/`: Product detail page.

Cart Functionality

Implementing a shopping cart allows users to add, remove, and modify products before proceeding to checkout.

1. Define Cart Operations

We'll manage the cart using session data to store cart items.

a. Create Cart Utility Functions

```
# cart/cart.py

from decimal import Decimal
from django.conf import settings
from store.models import Product

class Cart:
    """
    A class to manage the shopping cart stored in the user's
session.
    """

    def __init__(self, request):
        """
        Initialize the cart.
        """
        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # Save an empty cart in the session
            cart = self.session[settings.CART_SESSION_ID] = {}
        self.cart = cart

    def add(self, product, quantity=1, update_quantity=False):
        """
        Add a product to the cart or update its quantity.
        """
        product_id = str(product.id)
        if product_id not in self.cart:
```

```
            self.cart[product_id] = {'quantity': 0, 'price':
str(product.price)}
        if update_quantity:
            self.cart[product_id]['quantity'] = quantity
        else:
            self.cart[product_id]['quantity'] += quantity
        self.save()

    def save(self):
        """
        Mark the session as "modified" to ensure it is saved.
        """
        self.session.modified = True

    def remove(self, product):
        """
        Remove a product from the cart.
        """
        product_id = str(product.id)
        if product_id in self.cart:
            del self.cart[product_id]
            self.save()

    def __iter__(self):
        """
        Iterate over the items in the cart and get the products from
the database.
        """
        product_ids = self.cart.keys()
        # Get the product objects and add them to the cart
        products = Product.objects.filter(id__in=product_ids)
        for product in products:
            self.cart[str(product.id)]['product'] = product

        for item in self.cart.values():
            item['price'] = Decimal(item['price'])
            item['total_price'] = item['price'] * item['quantity']
            yield item

    def __len__(self):
        """
        Count all items in the cart.
```

```
        """
        return sum(item['quantity'] for item in self.cart.values())

    def get_total_price(self):
        """
        Calculate the total price of the cart.
        """
        return sum(Decimal(item['price']) * item['quantity'] for
item in self.cart.values())

    def clear(self):
        """
        Remove cart from session.
        """
        del self.session[settings.CART_SESSION_ID]
        self.save()
```

Explanation:

- **Initialization**: Retrieves or initializes the cart in the session.
- **add**: Adds a product to the cart or updates its quantity.
- **remove**: Removes a product from the cart.
- **iter**: Iterates over cart items, attaching product data and calculating total prices.
- **len**: Returns the total number of items in the cart.
- **get_total_price**: Calculates the total cost of the cart.
- **clear**: Empties the cart.

b. Update Settings for Cart Session ID

Add a cart session identifier in `m2collection/settings.py`.

```
# m2collection/settings.py

# ... (existing settings)

# Cart session ID
CART_SESSION_ID = 'cart'
```

c. Create Cart Views

```
# cart/views.py

from django.shortcuts import render, redirect, get_object_or_404
from store.models import Product
from .cart import Cart
from django.urls import reverse
```

```
from .forms import CartAddProductForm

def cart_detail(request):
    """
    Displays the cart details and allows modifications.
    """
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] =
CartAddProductForm(initial={'quantity': item['quantity'], 'update':
True})
    return render(request, 'cart/detail.html', {'cart': cart})

def cart_add(request, product_id):
    """
    Adds a product to the cart or updates its quantity.
    """
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id,
available=True)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(
            product=product,
            quantity=cd['quantity'],
            update_quantity=cd['update']
        )
    return redirect('cart:cart_detail')

def cart_remove(request, product_id):
    """
    Removes a product from the cart.
    """
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id,
available=True)
    cart.remove(product)
    return redirect('cart:cart_detail')
```

Explanation:

- **cart_detail**: Shows the current cart with options to update quantities.

- **cart_add**: Handles adding or updating products in the cart based on form data.
- **cart_remove**: Removes a specified product from the cart.

d. Create Cart Forms

```
# cart/forms.py

from django import forms

class CartAddProductForm(forms.Form):
    """
    Form to add a product to the cart or update its quantity.
    """
    quantity = forms.IntegerField(min_value=1, max_value=100,
initial=1)
    update = forms.BooleanField(required=False, initial=False,
widget=forms.HiddenInput)
```

Explanation:

- **quantity**: Number of units to add or set.
- **update**: Determines if the action should update the quantity or add to it.

e. Create Cart URLs

```
# cart/urls.py

from django.urls import path
from . import views

app_name = 'cart'

urlpatterns = [
    path('', views.cart_detail, name='cart_detail'),  # /cart/
    path('add/<int:product_id>/', views.cart_add, name='cart_add'),
# /cart/add/1/
    path('remove/<int:product_id>/', views.cart_remove,
name='cart_remove'),  # /cart/remove/1/
]
```

Explanation:

- **shop:cart_add**: Adds a product to the cart.
- **shop:cart_remove**: Removes a product from the cart.

f. Create Cart Templates

i. cart_detail.html

```
<!-- cart/templates/cart/detail.html -->

{% extends 'store/base.html' %}
{% load static %}
{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <h2>Your Shopping Cart</h2>
    {% if cart %}
        <table class="table table-bordered">
            <thead>
                <tr>
                    <th>Product</th>
                    <th>Price</th>
                    <th>Quantity</th>
                    <th>Total</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody>
                {% for item in cart %}
                    <tr>
                        <td><a
href="{{ item.product.get_absolute_url }}">{{ item.product.name }}</
a></td>
                        <td>${{ item.price }}</td>
                        <td>
                            <form method="post" action="{% url
'cart:cart_add' item.product.id %}">
                                {% csrf_token %}

{{ item.update_quantity_form.quantity }}

{{ item.update_quantity_form.update }}
                                <button type="submit" class="btn
btn-sm btn-primary">Update</button>
                            </form>
                        </td>
                        <td>${{ item.total_price }}</td>
```

```
                        <td>
                            <a href="{% url 'cart:cart_remove'
item.product.id %}" class="btn btn-sm btn-danger">Remove</a>
                        </td>
                    </tr>
                {% endfor %}
                <tr>
                    <td colspan="3" class="text-
end"><strong>Total:</strong></td>
                    <td
colspan="2"><strong>${{ cart.get_total_price }}</strong></td>
                </tr>
            </tbody>
        </table>
        <a href="{% url 'orders:order_create' %}" class="btn btn-
success">Proceed to Checkout</a>
    {% else %}
        <p>Your cart is empty.</p>
        <a href="{% url 'store:product_list' %}" class="btn btn-
primary">Continue Shopping</a>
    {% endif %}
</div>
{% endblock %}
```

Explanation:

- **Cart Items**: Displays each item with options to update quantity or remove from the cart.
- **Total Price**: Shows the total cost of all items in the cart.
- **Checkout Button**: Navigates to the order creation page.
- **Empty Cart Message**: Informs the user if the cart is empty.

2. Update Product Detail Template to Include Add to Cart

Update `product_detail.html` to use the `cart_add` view.

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

{% extends 'store/base.html' %}
{% load static %}
{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <div class="row">
        <!-- Product Image -->
```

```
        <div class="col-md-6">
            {% if product.image %}
                <img src="{{ product.image.url }}" class="img-fluid"
alt="{{ product.name }}">
            {% else %}
                <img src="{% static
'store/images/no_image_available.png' %}" class="img-fluid" alt="No
image">
            {% endif %}
        </div>

        <!-- Product Details -->
        <div class="col-md-6">
            <h2>{{ product.name }}</h2>
            <p class="text-muted">${{ product.price }}</p>
            <p>{{ product.description }}</p>

            <!-- Add to Cart Form -->
            <form method="post" action="{% url 'cart:cart_add'
product.id %}">
                {% csrf_token %}
                <div class="mb-3">
                    <label for="quantity" class="form-
label">Quantity:</label>
                    <input type="number" name="quantity" value="1"
min="1" class="form-control" id="quantity">
                </div>
                <button type="submit" class="btn btn-primary">Add to
Cart</button>
            </form>
        </div>
    </div>

    <!-- Back to Products -->
    <a href="{% url 'store:product_list' %}" class="btn btn-
secondary mt-4">Back to Products</a>
</div>
{% endblock %}
```

Explanation:

- **Add to Cart Form**: Posts to the `cart_add` view to add the selected product to the cart.

Payment Integration

Integrating a payment gateway like **Stripe** allows users to securely make payments. We'll use Stripe's API for handling payments.

1. Configure Stripe Settings

a. Obtain Stripe API Keys

- Sign up for a Stripe account.
- Obtain your **Publishable key** and **Secret key** from the Stripe Dashboard under Developers > API Keys.

b. Add Stripe Keys to Settings

Add Stripe keys to your project's settings. It's best practice to use environment variables for sensitive information, but for simplicity, we'll add them directly. **Note**: In production, always use environment variables or secure storage for API keys.

```
# m2collection/settings.py

# ... (existing settings)

# Stripe Integration
STRIPE_PUBLIC_KEY = 'your-publishable-key-here'
STRIPE_SECRET_KEY = 'your-secret-key-here'
```

Explanation:

- **STRIPE_PUBLIC_KEY**: Used on the client side to handle payments.
- **STRIPE_SECRET_KEY**: Used on the server side to process payments securely.

2. Create Order Creation View

```
# orders/views.py

from django.shortcuts import render, redirect
from cart.cart import Cart
from .models import Order, OrderItem
from django.contrib.auth.decorators import login_required
from .forms import OrderCreateForm
from django.conf import settings
import stripe
from django.urls import reverse
from django.contrib import messages

stripe.api_key = settings.STRIPE_SECRET_KEY  # Initialize Stripe
with secret key
```

```python
@login_required
def order_create(request):
    """
    Handles the creation of an order and processes payment via
Stripe.
    """
    cart = Cart(request)
    if len(cart) == 0:
        messages.error(request, "Your cart is empty.")
        return redirect('store:product_list')

    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
        if form.is_valid():
            # Create Order object but don't save to database yet
            order = form.save(commit=False)
            order.user = request.user
            order.save()
            # Create OrderItem objects for each item in the cart
            for item in cart:
                OrderItem.objects.create(
                    order=order,
                    product=item['product'],
                    price=item['price'],
                    quantity=item['quantity']
                )
            # Process payment with Stripe
            try:
                charge = stripe.Charge.create(
                    amount=int(cart.get_total_price() * 100),  #
Amount in cents
                    currency='usd',
                    description=f'Order {order.id}',
                    source=request.POST['stripeToken']
                )
                # Save the charge ID in the order
                order.paid = True
                order.stripe_charge_id = charge.id
                order.save()
                # Clear the cart
                cart.clear()
                messages.success(request, "Your order has been
```

```
placed successfully.")
                return redirect('orders:order_detail',
order_id=order.id)
            except stripe.error.CardError as e:
                messages.error(request, "Your card was declined.")
    else:
        form = OrderCreateForm()

    context = {
        'cart': cart,
        'form': form,
        'stripe_public_key': settings.STRIPE_PUBLIC_KEY,
    }
    return render(request, 'orders/order/create.html', context)
```

Explanation:

- **Authentication**: Only logged-in users can place orders.
- **Order Creation**:
  - **OrderCreateForm**: Captures billing and shipping information.
  - **OrderItem**: Saves each item in the cart as an order item.
- **Stripe Payment**:
  - **Charge Creation**: Processes the payment using the Stripe API.
  - **Error Handling**: Catches card errors and notifies the user.
- **Post-Payment Actions**:
  - **Mark Order as Paid**: Updates the order status.
  - **Clear Cart**: Empties the user's cart.
  - **Redirect**: Sends the user to the order detail page upon success.

3. Create Order Creation Form

```
# orders/forms.py

from django import forms
from .models import Order

class OrderCreateForm(forms.ModelForm):
    """
    Form for customers to enter billing and shipping information.
    """
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
'postal_code', 'city']
```

Explanation:

- **ModelForm**: Automatically generates form fields based on the `Order` model.

4. Create Order Detail View

```
# orders/views.py

from django.shortcuts import render, get_object_or_404
from .models import Order
from django.contrib.auth.decorators import login_required

@login_required
def order_detail(request, order_id):
    """
    Displays the details of a specific order.
    """
    order = get_object_or_404(Order, id=order_id, user=request.user)
    return render(request, 'orders/order/detail.html', {'order':
order})
```

Explanation:

- **Authorization**: Ensures that users can only view their own orders.

5. Create Forms and Templates for Orders

a. Create Order Templates

i. `create.html`

```
<!-- orders/templates/orders/order/create.html -->

{% extends 'store/base.html' %}
{% load static %}
{% load crispy_forms_tags %}

{% block content %}
<div class="container mt-5">
    <h2>Checkout</h2>
    <form method="post">
        {% csrf_token %}
        {{ form|crispy }}

        <!-- Stripe Payment Button -->
        <script
            src="https://checkout.stripe.com/checkout.js"
class="stripe-button"
            data-key="{{ stripe_public_key }}"
            data-
```

```
amount="{{ cart.get_total_price|floatformat:2|floatformat:0|add:'0'|
multiply:'100' }}"
            data-name="M2 Collection"
            data-description="Order {{ order.id }}"
            data-email="{{ user.email }}"
            data-currency="usd"
            data-label="Pay with Stripe">
        </script>
    </form>
</div>
{% endblock %}
```

Explanation:

- **Stripe Checkout**: Integrates Stripe's checkout form for secure payments.
- **Data Attributes**:
    - **data-key**: Stripe publishable key.
    - **data-amount**: Total amount in cents.
    - **data-name**: Merchant name.
    - **data-description**: Order description.
    - **data-email**: Customer email.
    - **data-currency**: Currency code.

ii. `detail.html`

```
<!-- orders/templates/orders/order/detail.html -->

{% extends 'store/base.html' %}
{% load static %}

{% block content %}
<div class="container mt-5">
    <h2>Order Details</h2>
    <p><strong>Order ID:</strong> {{ order.id }}</p>
    <p><strong>Placed By:</strong> {{ order.first_name }}
{{ order.last_name }}</p>
    <p><strong>Email:</strong> {{ order.email }}</p>
    <p><strong>Address:</strong> {{ order.address }},
{{ order.city }}, {{ order.postal_code }}</p>
    <p><strong>Created:</strong> {{ order.created }}</p>
    <p><strong>Updated:</strong> {{ order.updated }}</p>
    <p><strong>Status:</strong> {% if order.paid %}Paid{% else %}Not
Paid{% endif %}</p>

    <h4 class="mt-4">Items</h4>
```

```
    <table class="table table-bordered">
        <thead>
            <tr>
                <th>Product</th>
                <th>Price</th>
                <th>Quantity</th>
                <th>Total Price</th>
            </tr>
        </thead>
        <tbody>
            {% for item in order.items.all %}
                <tr>
                    <td>{{ item.product.name }}</td>
                    <td>${{ item.price }}</td>
                    <td>{{ item.quantity }}</td>
                    <td>${{ item.get_cost }}</td>
                </tr>
            {% endfor %}
            <tr>
                <td colspan="3" class="text-
end"><strong>Total:</strong></td>

<td><strong>${{ order.get_total_cost }}</strong></td>
            </tr>
        </tbody>
    </table>

    <a href="{% url 'store:product_list' %}" class="btn btn-
primary">Continue Shopping</a>
</div>
{% endblock %}
```

Explanation:

- **Order Details**: Displays comprehensive information about the order.
- **Items Table**: Lists all products in the order with their prices and quantities.
- **Total Cost**: Shows the total amount paid.

6. Create Orders App URLs

```
# orders/urls.py

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

```
app_name = 'orders'

urlpatterns = [
    path('create/', views.order_create, name='order_create'),  # /orders/create/
    path('<int:order_id>/', views.order_detail, name='order_detail'),  # /orders/1/
]
```

Explanation:

- **order_create**: Initiates the checkout process.
- **order_detail**: Displays specific order details.

Creating Templates

Creating consistent and responsive templates is crucial for user experience.

1. Create Base Template

We'll create a base template to be inherited by other templates, ensuring consistency across pages.

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

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>M2 Collection - {% block title %}{% endblock %}</title>
    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
    <!-- Custom CSS -->
    <link href="{% static 'store/css/styles.css' %}" rel="stylesheet">
    {% load static %}
    {% load crispy_forms_tags %}
</head>
<body>
    <!-- Navigation Bar -->
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <div class="container">
            <a class="navbar-brand" href="{% url 'store:product_list' %}">M2 Collection</a>
```

```
            <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>
            <div class="collapse navbar-collapse" id="navbarNav">
                <ul class="navbar-nav ms-auto">
                    <li class="nav-item">
                        <a class="nav-link" href="{% url
'store:product_list' %}">Products</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{% url
'cart:cart_detail' %}">Cart ({{ request.session.cart|length }})</a>
                    </li>
                    {% if user.is_authenticated %}
                        <li class="nav-item">
                            <a class="nav-link" href="#">Hello,
{{ user.username }}!</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="{% url
'logout' %}">Logout</a>
                        </li>
                    {% else %}
                        <li class="nav-item">
                            <a class="nav-link" href="{% url
'store:signup' %}">Sign Up</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link" href="{% url
'login' %}">Login</a>
                        </li>
                    {% endif %}
                </ul>
            </div>
        </div>
    </nav>

    <!-- Flash Messages -->
    <div class="container mt-3">
```

```
        {% if messages %}
            {% for message in messages %}
                <div class="alert alert-{{ message.tags }} alert-
dismissible fade show" role="alert">
                    {{ message }}
                    <button type="button" class="btn-close" data-bs-
dismiss="alert" aria-label="Close"></button>
                </div>
            {% endfor %}
        {% endif %}
    </div>

    <!-- Main Content -->
    {% block content %}
    {% endblock %}

    <!-- Footer -->
    <footer class="bg-dark text-white py-4 mt-5">
        <div class="container text-center">
            &copy; {{ now|date:"Y" }} M2 Collection. All rights
reserved.
        </div>
    </footer>

    <!-- Bootstrap JS Bundle (includes Popper) -->
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.
bundle.min.js"></script>
</body>
</html>
```

Explanation:

- **Navigation Bar**: Contains links to products, cart, signup, login, and logout. Shows the number of items in the cart.
- **Flash Messages**: Displays one-time notifications to users (e.g., success or error messages).
- **Footer**: Provides consistent footer across pages.

2. Custom CSS (Optional)

Create a CSS file to add custom styles.

```
/* store/static/store/css/styles.css */

/* Add any custom styles here */
```

```css
body {
    padding-top: 70px; /* Adjust based on navbar height */
}
```

Explanation:

- **Padding**: Ensures the main content isn't hidden behind the fixed navbar.

Static and Media Files Management

Managing static and media files correctly is essential for the proper functioning and appearance of your website.

1. Update Settings for Static and Media Files

```python
# m2collection/settings.py

import os

# ... (existing settings)

# Static files (CSS, JavaScript, Images)
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static'),
]
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')  # For production

# Media files (Uploaded images)
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

Explanation:

- **STATIC_URL & STATICFILES_DIRS**: Define where Django looks for static files during development.
- **STATIC_ROOT**: Destination for `collectstatic` in production.
- **MEDIA_URL & MEDIA_ROOT**: Define the base URL and filesystem path for user-uploaded files.

2. Collect Static Files (For Deployment)

When deploying, collect all static files into `STATIC_ROOT`.

```
python manage.py collectstatic
```

URLs Configuration

Ensure that all apps have their URLs correctly configured and namespaced.

1. Store App URLs

Already defined in `[store]/urls.py`.

2. Cart App URLs

```
# cart/urls.py

from django.urls import path
from . import views

app_name = 'cart'

urlpatterns = [
    path('', views.cart_detail, name='cart_detail'),  # /cart/
    path('add/<int:product_id>/', views.cart_add, name='cart_add'),
# /cart/add/1/
    path('remove/<int:product_id>/', views.cart_remove,
name='cart_remove'),  # /cart/remove/1/
]
```

3. Orders App URLs

Already defined in `[orders]/urls.py`.

4. Update Project URLs

Ensure all app URLs are included in the project's `urls.py`.

```
# m2collection/urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('store/', 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')),  # Adds
login, logout, password management URLs
]

if settings.DEBUG:
```

```
    urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

Enhancing User Experience

A good user experience is vital for an e-commerce site. Here are some enhancements to consider:

1. Responsive Design

Using Bootstrap ensures that the website is mobile-friendly and adapts to different screen sizes.

2. Flash Messages

Already implemented in the base template to provide immediate feedback to users.

3. Quantity Updates in Cart

Allow users to update the quantity of items directly from the cart page.

4. Notifications

Inform users about successful actions (e.g., adding to cart, order placement) using Django messages.

5. Improve Search and Filter

Integrate **django-filter** for more advanced filtering options in product listings.

Deployment

Deploying your Django application allows it to be accessible to users worldwide. We'll outline the deployment process using **Heroku** as an example. Heroku simplifies deployment and offers a free tier suitable for small projects.

1. Prepare the Project for Deployment

a. Install `gunicorn` and `dj-database-url`

```
pip install gunicorn dj-database-url whitenoise django-heroku
```

Explanation:

- **gunicorn**: A Python WSGI HTTP server for UNIX.
- **dj-database-url**: Simplifies database configurations.
- **whitenoise**: Serves static files efficiently in production.
- **django-heroku**: Automatically configures Django settings for Heroku.

b. Update `settings.py`

```
# m2collection/settings.py

import os
from pathlib import Path
import django_heroku
```

```
import dj_database_url
from decouple import config  # Use python-decouple for environment
variables

# ... (existing settings)

# SECRET_KEY and DEBUG should be set via environment variables for
security
SECRET_KEY = config('SECRET_KEY', default='your-default-secret-key')
DEBUG = config('DEBUG', default=True, cast=bool)

ALLOWED_HOSTS = ['*']  # Adjust for production

# WhiteNoise for static files
STATICFILES_STORAGE =
'whitenoise.storage.CompressedManifestStaticFilesStorage'

# Database Configuration
DATABASES = {
    'default': dj_database_url.config(
        default='sqlite:///db.sqlite3',
        conn_max_age=600
    )
}

# Activate Django-Heroku
django_heroku.settings(locals())
```

Explanation:

- **Environment Variables**: Use environment variables to manage sensitive data.
- **WhiteNoise**: Handles serving static files in production.
- **dj_database_url**: Configures the database dynamically based on the `DATABASE_URL` environment variable.
- **django-heroku**: Automatically configures logging, databases, and static assets for Heroku.

c. Create `Procfile`

Defines the commands to run your app on Heroku.

```
# Create a file named 'Procfile' in the project root
(m2collection/Procfile)
echo "web: gunicorn m2collection.wsgi" > Procfile
```

d. Create `requirements.txt`

List all project dependencies.

```
pip freeze > requirements.txt
```

e. Configure Static Files with WhiteNoise

Ensure that `WhiteNoise` is properly set up to serve static files.

```
# m2collection/settings.py

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',  # WhiteNoise
middleware
    # ... (other middleware)
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    # ...
]
```

2. Deploy to Heroku

a. Initialize Git Repository

If not already initialized, set up Git.

```
git init
git add .
git commit -m "Initial commit"
```

b. Create a Heroku App

```
# Install the Heroku CLI if not already installed
# https://devcenter.heroku.com/articles/heroku-cli

heroku login

# Create a new Heroku app
heroku create m2collection-app
```

Explanation:

- **heroku create**: Initializes a new Heroku application with a unique URL.

c. Add Heroku Remote (If Not Automatically Added)

```
heroku git:remote -a m2collection-app
```

d. Set Environment Variables on Heroku

```
heroku config:set SECRET_KEY='your-production-secret-key'
heroku config:set DEBUG=False
```

Explanation:

- **SECRET_KEY**: Must be a unique, unpredictable value.
- **DEBUG=False**: Disables debug mode in production for security.

e. Push Code to Heroku

```
git push heroku master
```

Explanation:

- **Deployment**: Heroku detects the Django project, installs dependencies, runs migrations, collects static files, and starts the application.

f. Run Migrations on Heroku

```
heroku run python manage.py migrate
```

g. Create a Superuser on Heroku

```
heroku run python manage.py createsuperuser
```

Follow the prompts to set up the superuser account.

h. Open the App

```
heroku open
```

Your deployed e-commerce site should now be live at https://m2collection-app.herokuapp.com/.

Conclusion

Congratulations! You've successfully built "M2 Collection," a full-fledged e-commerce website for men's clothing using Django. This project encompasses essential features like product management, user authentication, shopping cart functionality, search and filtering, and payment processing with Stripe. Additionally, you've learned how to deploy your Django application to Heroku, making it accessible to users worldwide.

Key Takeaways:

- **Django's Robust Framework**: Leveraged Django's powerful features to build scalable web applications.
- **Modular Design with Apps**: Organized the project into distinct apps for better maintainability.
- **User Experience Enhancements**: Implemented search, filter, and responsive design to improve usability.
- **Secure Payment Processing**: Integrated Stripe for handling secure payments.

- **Deployment Practices**: Gained insights into deploying Django applications using Heroku.

Next Steps:

1. **Enhance Security**:
    a. Implement HTTPS using SSL certificates.
    b. Use environment variables for all sensitive settings.
    c. Regularly update dependencies to patch vulnerabilities.
2. **Add Advanced Features**:
    a. **User Profiles**: Allow users to manage their profiles and view order history.
    b. **Product Reviews**: Enable customers to leave reviews and ratings.
    c. **Email Notifications**: Send order confirmations and updates via email.
    d. **Inventory Management**: Track stock levels and automate notifications for low stock.
3. **Optimize Performance**:
    a. Implement caching mechanisms.
    b. Optimize database queries.
    c. Use a Content Delivery Network (CDN) for static and media files.
4. **Scale the Application**:
    a. Move to more scalable hosting solutions as traffic grows.
    b. Implement load balancing and database replication.
5. **Continuous Integration/Deployment (CI/CD)**:
    a. Set up automated testing.
    b. Implement CI/CD pipelines for streamlined deployments.