

“E-COMMERCE WEB APP”

A Project Report

Submitted in partial fulfilment of the
Requirement for the award of the degree of

BACHELOR OF SCIENCE

(COMPUTER SCIENCE)

By

Mr. Omkar Rajendra Kawale

Under the esteemed guidance of

Mrs. Pritam Kamble

Assistant Professor



DEPARTMENT OF COMPUTER SCIENCE

B. K. BIRLA COLLEGE, KALYAN

(EMPOWERED AUTONOMOUS STATUS)

KALYAN- 421304

2023-2024

B. K. BIRLA COLLEGE (AUTONOMOUS), KALYAN
(DEPARTMENT OF COMPUTER SCIENCE)



C E R T I F I C A T E

This is to certify that the project entitled E-COMMERCE WEB APPLICATION submitted by Omkar Rajendra Kawale, Seat no 63241030 is record of bonafide work carried out by him, under my guidance, in partial fulfillment of the requirement for the award of the Degree of B.Sc. Computer Science of University of Mumbai.

Date: 30-04-2024

Place: Kalyan

Project Guide

External Examiner

Head of Department
(Computer Science)

ACKNOWLEDGEMENT

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

I am highly indebted to **Mrs. Pritam Kamble** ma'am for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

I would like to express my gratitude towards all the respected teachers and head of the department **Mr. Vinod Rajput** sir of Computer Science department and principal **Dr. Avinash Patil** sir and Vice principal **Ms. Esmita Gupta** mam of **B. K. Birla College of Science Commerce and (Arts Empowered Autonomous Status)** for their kind cooperation and encouragement which help me in completion of this project.

DECLARATION

I hereby declare that the project entitled “E-COMMERCE WEB APPLICATION” named as “ESHOP” done at place where project is done, has not been in any case duplicated to submit to any other university for the award of any degree to the best of my knowledge other than me, no one has submitted to any other university.

The project is done in partial fulfillment of requirements for the award of degree of **BACHLEOR OF SCIENCE (COMPUTER SCIENCE)** to be submitted as sixth semester project as part of our curriculum.

TABLE OF CONTENTS

ABSTRACT.....	2
CHAPTER 1. INTRODUCTION.....	3
1.1 Fundamentals.....	3
1.2 The Objectives.....	5
CHAPTER 2. FEATURES OF ESHOP.....	7
2.1 Features.....	7
2.2 Uniqueness of project.....	9
CHAPTER 3. ABOUT THE DEVELOPMENT.....	10
3.1 Basic Requirements for Development.....	10
3.1.1 System Requirements.....	10
3.1.2 Installation of essentials.....	11
3.2 Creating a Django Web Project in Visual Studio.....	18
3.3 Making changes in template.....	21
3.3.1 Setup virtual environment.....	21
3.3.2 Create app entities.....	22
3.3.3 Create HTML for Web Pages.....	43
3.3.4 Start server on web browser.....	50
3.3.5 Django Administration.....	52
CHAPTER 4. PERFORMANCE AND RESULTS.....	56
4.1 Implementation and results.....	56
4.2 Output.....	59
CHAPTER 5. APPLICATIONS.....	62
CHAPTER 6. CONCLUSION.....	65
REFERENCES.....	67

ABSTRACT

Eshop is a robust e-commerce web application developed using the Django framework, designed to provide an efficient and seamless shopping experience for users. Leveraging Django's powerful features, Eshop offers a scalable, secure, and customizable platform for businesses to showcase and sell their products online.

Key features of Eshop include a user-friendly interface, intuitive product browsing and search functionalities, secure payment gateways integration, and order management tools. The application incorporates responsive design principles, ensuring compatibility across various devices and screen sizes, thus maximizing accessibility for users.

Eshop prioritizes security through implementation of best practices, including encryption protocols for sensitive data transmission and user authentication mechanisms. Furthermore, the platform includes robust administrative capabilities, enabling business owners to manage product listings, inventory, and customer interactions efficiently.

Built with extensibility in mind, Eshop facilitates easy integration of additional features and modules, allowing businesses to adapt and grow with evolving market demands. The use of Django's MVC architecture streamlines development and maintenance processes, ensuring scalability and maintainability of the application over time.

In conclusion, Eshop represents a comprehensive solution for businesses seeking to establish a strong online presence and streamline their e-commerce operations. With its feature-rich functionality, security measures, and flexibility, Eshop empowers businesses to succeed in the competitive online marketplace.

Keywords: E-Commerce, Django

CHAPTER 1

INTRODUCTION

1.1 Fundamentals

Introduction to Eshop

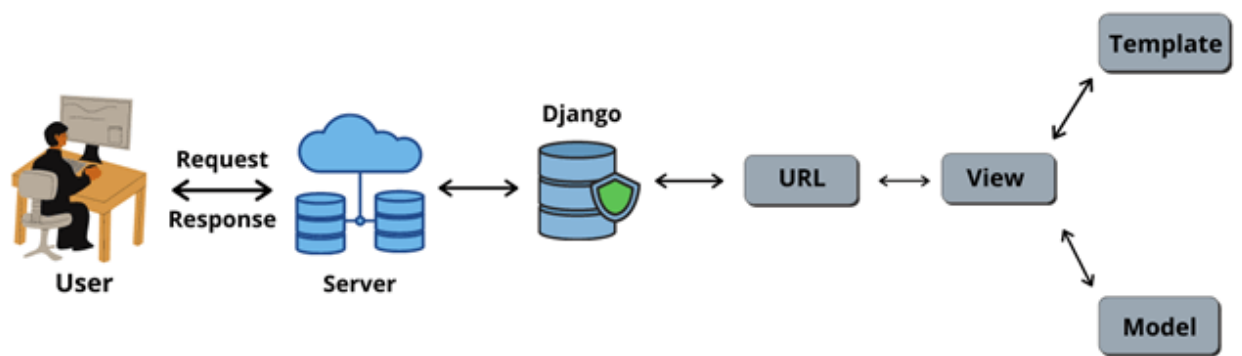
Welcome to Eshop, your comprehensive solution for seamless online buying and selling experiences. Powered by Django, Eshop is a sophisticated web application designed to facilitate effortless transactions between buyers and sellers in the dynamic world of e-commerce.

Eshop is tailored to meet the diverse needs of both consumers and businesses, offering a user-friendly platform where products can be easily showcased, discovered, and purchased with confidence. Whether you're a small boutique seeking to expand your reach or a seasoned retailer aiming to optimize your online presence, Eshop provides the tools and functionalities to elevate your e-commerce game.

For buyers, Eshop presents a captivating shopping experience with intuitive navigation, advanced search capabilities, and secure payment options. Discover a vast array of products spanning various categories, all conveniently accessible from the comfort of your own device. With Eshop's responsive design, shopping on-the-go has never been easier.

For sellers, Eshop offers a streamlined platform to showcase and manage product listings, track inventory, and engage with customers. With robust administrative features, sellers can efficiently monitor sales, analyze performance metrics, and tailor their strategies to maximize profitability.

Eshop isn't just another online marketplace—it's a dynamic ecosystem where businesses thrive and consumers find exactly what they're looking for. Whether you're buying or selling, Eshop invites you to embark on a journey of seamless transactions, unparalleled convenience, and endless possibilities. Welcome to the future of e-commerce. Welcome to Eshop.



DeveloperStacks.com

Fig 1. Basic Structure of Django

1.2 The objectives of E-commerce Web App Eshop

1. **Seamless User Experience:** Ensure a smooth and intuitive interface for users to navigate, search, and purchase products effortlessly.
2. **Secure Payment Processing:** Implement robust payment gateways to safeguard sensitive financial transactions and user data.
3. **Product Catalog Management:** Enable sellers to easily create, update, and manage product listings, including descriptions, images, and pricing.
4. **Advanced Search Functionality:** Provide users with powerful search filters and sorting options to quickly find desired products.
5. **Mobile Responsiveness:** Optimize the application for mobile devices to cater to the increasing number of users shopping on smartphones and tablets.
6. **User Authentication and Authorization:** Implement secure user authentication mechanisms to protect user accounts and ensure authorized access to sensitive functionalities.
7. **Order Management System:** Develop a comprehensive system for users to view order history, track shipments, and manage returns or exchanges.
8. **Inventory Management:** Offer sellers tools to efficiently monitor and update inventory levels, including automatic notifications for low stock items.
9. **Customer Reviews and Ratings:** Allow users to leave feedback and ratings for products, fostering trust and transparency within the community.
10. **Promotions and Discount:** Enable sellers to create promotional campaigns and offer discounts to attract customers and increase sales.
11. **Multi-Language and Currency Support:** Cater to a global audience by providing multilingual and multicurrency support, enhancing accessibility and user experience.

12. **Social Media Integration:** Allow users to share products on social media platforms and leverage social proof to drive traffic and sales.

13. **Analytics and Reporting:** Provide comprehensive analytics tools for sellers to track sales performance, analyze customer behavior, and make data-driven decisions.

14. **Customer Support Features:** Offer channels for users to contact customer support, including live chat, email, and FAQs, to address inquiries and resolve issues promptly.

15. **Scalability and Performance Optimization:** Design the application architecture to handle increased traffic and user loads, ensuring optimal performance during peak periods and future scalability as the platform grows.

These objectives aim to create a robust and user-friendly e-commerce platform that caters to the needs of both buyers and sellers, fostering trust, efficiency, and growth in online transactions.

CHAPTER 2

FEATURES OF ESHOP

2.1 Features

1. **User Registration and Authentication:** Allow users to create accounts and securely log in to access personalized features such as order history, saved addresses, and preferences.
2. **Product Catalog:** Provide a comprehensive catalog of products categorized into relevant sections with detailed descriptions, images, and pricing information.
3. **Search and Filter:** Offer robust search and filter options to help users easily find products based on keywords, categories, brands, price range, and other criteria.
4. **Product Details:** Display detailed product pages with high-quality images, specifications, customer reviews, and related products to assist users in making informed purchasing decisions.
5. **Add to Cart:** Enable users to add products to their shopping carts for later purchase, with the ability to adjust quantities and view subtotal amounts.
6. **Secure Checkout:** Implement a secure checkout process with multiple payment options, including credit/debit cards, digital wallets, and other online payment gateways, ensuring the safety of sensitive financial information.
7. **Order Management:** Provide users with order tracking functionality to monitor the status of their purchases, including order confirmation, shipment tracking, and delivery notifications.
8. **Wishlist:** Allow users to create and manage wishlists of products they intend to purchase in the future, with the option to share them with others or save for later.
9. **User Reviews and Ratings:** Enable users to leave reviews and ratings for products they have purchased, helping other shoppers make informed decisions and providing valuable feedback to sellers.

10. **Seller Dashboard:** Offer sellers a dedicated dashboard to manage their product listings, inventory, orders, and customer interactions, with insights into sales performance and analytics.

11. **Promotions and Discounts:** Allow sellers to create and manage promotional campaigns, discount codes, and special offers to attract customers and boost sales.

12. **Multi-Language and Currency Support:** Cater to a diverse user base by providing support for multiple languages and currencies, enhancing accessibility and usability.

13. **Responsive Design:** Ensure the application is optimized for various devices and screen sizes, offering a seamless shopping experience across desktops, laptops, tablets, and smartphones.

14. **Customer Support:** Provide channels for users to contact customer support for assistance with orders, returns, refunds, and other inquiries, including live chat, email, and FAQs.

15. **Security and Privacy:** Implement robust security measures to protect user data, including SSL encryption, secure payment gateways, and adherence to data protection regulations such as GDPR.

These features collectively contribute to creating a user-friendly and efficient e-commerce platform that caters to the needs of both buyers and sellers, fostering trust, engagement, and growth in online transactions.

2.2 Uniqueness of the Project

Eshop : Eshop stands out in the crowded e-commerce landscape with its unique blend of customizable user experience, advanced analytics, and community engagement features. Unlike conventional platforms, Eshop offers businesses the flexibility to tailor their storefronts to match their branding and design preferences, ensuring a memorable and distinctive shopping experience for customers. With robust analytics tools, Eshop goes beyond basic sales tracking, providing deep insights into customer behavior and market trends to inform strategic decision-making. Moreover, Eshop fosters community engagement through user reviews, ratings, and social media integration, creating a vibrant online marketplace where customers feel connected and valued. Combined with its focus on security, scalability, and performance, Eshop sets itself apart as a versatile and powerful solution for businesses looking to thrive in the digital realm of buying and selling products.

CHAPTER 3

ABOUT THE DEVELOPMENT

3.1 Basic Requirements for development

3.1.1 System Requirements

To build a web project using Python Django, you'll need a system with certain configurations to ensure smooth development and deployment. Here are the basic system requirements:

1. **Operating System:** Django is compatible with various operating systems including Linux, macOS, and Windows. Choose an OS based on your preference and familiarity.
2. **Python:** Django is a Python web framework, so you need Python installed on your system. It's recommended to use Python 3.x (such as Python 3.6 or higher) as Django has dropped support for Python 2.x.
3. **Package Manager:** Install pip, the Python package manager, to install and manage Django and its dependencies.
4. **Virtual Environment:** Set up a virtual environment to isolate your Django project dependencies from other Python projects. This ensures consistency and prevents conflicts between packages. You can use virtualenv or venv, which comes bundled with Python 3.
5. **Database:** Django supports various databases including PostgreSQL, MySQL, SQLite, and Oracle. Install and configure the database of your choice based on your project requirements. PostgreSQL is often recommended for production environments due to its robust features and scalability.

6. **Text Editor or IDE:** Choose a text editor or integrated development environment (IDE) for coding Django projects. Popular choices include Visual Studio Code, PyCharm, Sublime Text, and Atom.

7. **Version Control:** Use version control software like Git for tracking changes to your project codebase and collaborating with other developers.

8. **Web Server:** While Django comes with a built-in development server for testing purposes, you'll need a production-grade web server like Nginx or Apache to deploy your Django application in a production environment.

9. **Additional Tools:** Depending on your project requirements, you may need additional tools such as Redis for caching, Celery for background task processing, and Docker for containerization and deployment.

By ensuring your system meets these configurations, you'll be well-equipped to start building Django web projects efficiently and effectively.

3.1.2 Installation of required essentials.

a. Visual Studio :

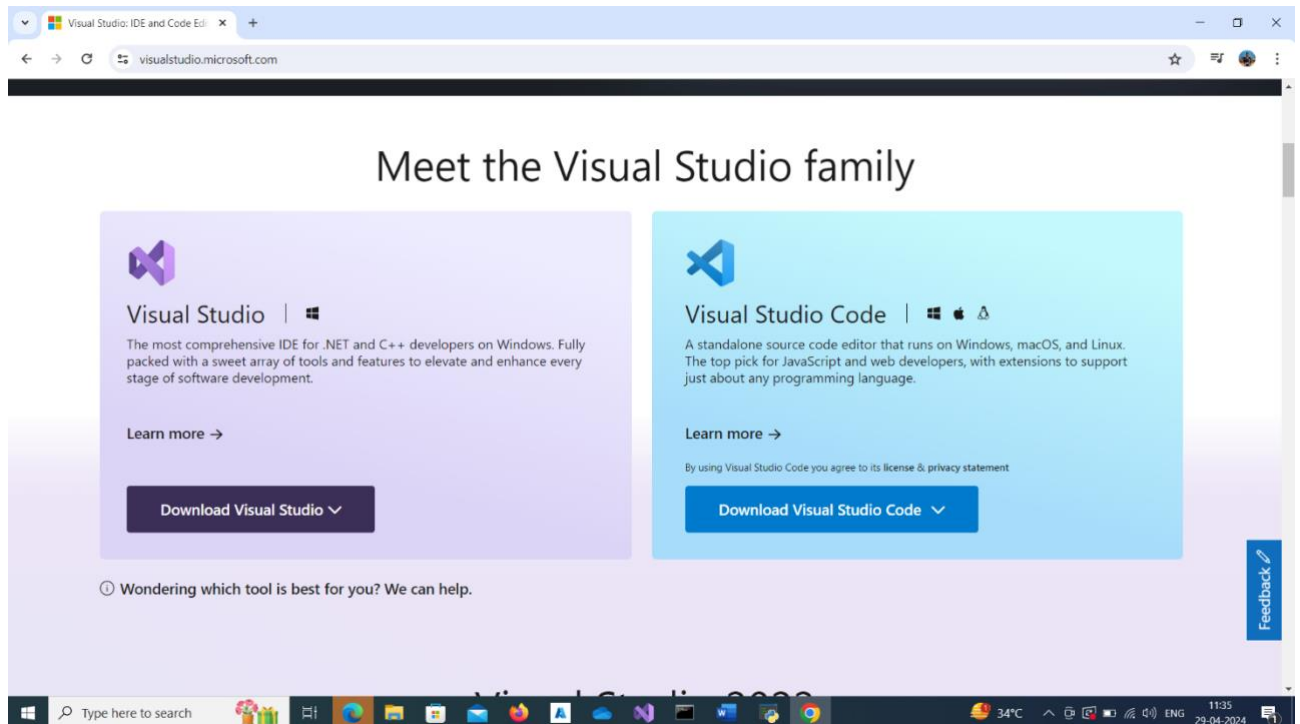
1. ****Download Visual Studio Installer**:** Visit the official Visual Studio website (<https://visualstudio.microsoft.com/>) and navigate to the "Downloads" section. Click on the download button for Visual Studio.

2. ****Run the Installer**:** Once the installer file is downloaded, locate it in your downloads folder or the location where your browser saves downloads. Double-click on the installer file (typically named `vs_community.exe` for the community edition) to launch the Visual Studio Installer.

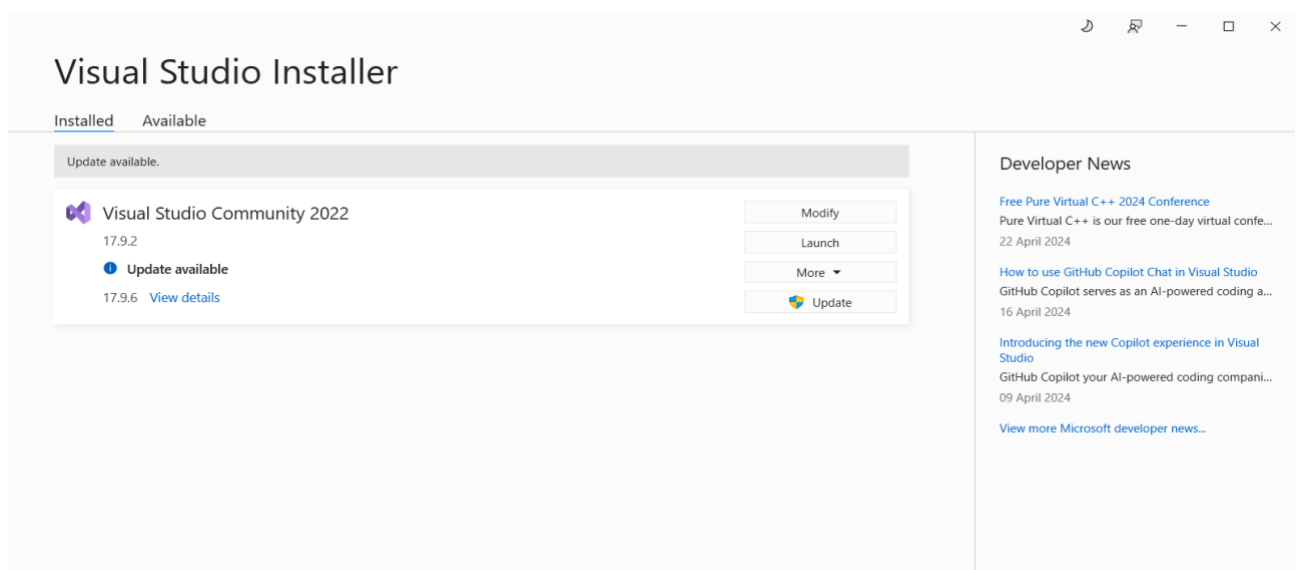
3. ****Choose Workloads****: The Visual Studio Installer will prompt you to choose the workloads you want to install. Workloads are sets of tools and components tailored for specific development scenarios, such as .NET desktop development, web development, or Python development. Select the workloads that best fit your needs. You can also select individual components if you prefer.
4. ****Select Optional Components****: After selecting the workloads, you have the option to select additional components and features to install. These may include SDKs, libraries, and tools for specific programming languages or frameworks.
5. ****Choose Installation Location****: Choose the installation location for Visual Studio. The default location is usually in the Program Files directory on your system drive, but you can customize it if needed.
6. ****Start Installation****: Once you've made your selections, click on the "Install" button to begin the installation process. Visual Studio will download and install the selected components and features.
7. ****Wait for Installation to Complete****: Depending on your internet speed and the selected components, the installation process may take some time. Be patient and wait for the installation to complete.
8. ****Launch Visual Studio****: Once the installation is finished, you can launch Visual Studio from the Start menu or desktop shortcut. The first time you launch Visual Studio, you may need to sign in with a Microsoft account or create a new one if you don't have an account already.
9. ****Configure Visual Studio****: Upon launching Visual Studio for the first time, you may be prompted to configure settings such as themes, development environments, and default settings. Follow the prompts to configure Visual Studio according to your preferences.

10. ****Start Coding****: With Visual Studio installed and configured, you're ready to start coding in your preferred programming languages and frameworks.

Following these steps will guide you through the process of installing Visual Studio on your system.



Download Visual Studio Community Edition



b. Python Installation :**1. **Download Python Installer:****

- Visit the official Python website at <https://www.python.org/downloads/>.
- Choose the version of Python you want to install. Typically, you'll want the latest stable release.
- Select the appropriate installer for your operating system. Python is available for Windows, macOS, and Linux.

2. **Run the Installer:**

- Once the installer is downloaded, run it by double-clicking on the downloaded file.

3. **Customize Installation (Optional):**

- The installer may provide options for customizing the installation. For most users, the default settings are sufficient, but you can choose to customize the installation directory or add Python to your system's PATH.

4. **Install Python:**

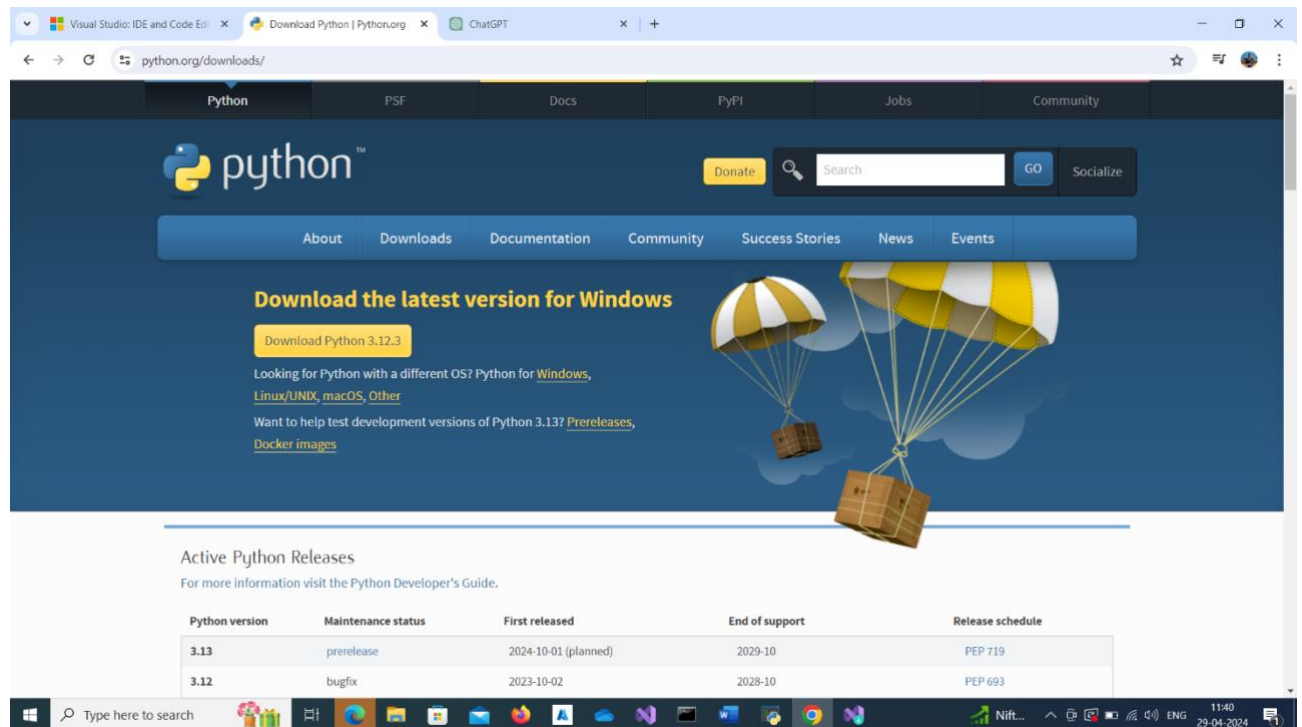
- Follow the prompts in the installer to install Python. The installer will copy the necessary files to your system.

5. **Verify Installation:**

- After the installation is complete, you can verify that Python is installed correctly by opening a command prompt (on Windows) or a terminal (on macOS or Linux) and typing ``python --version`` or ``python3 --version``. This command should display the installed version of Python.

6. **Access Python Interpreter:**

- You can access the Python interpreter by typing ``python`` or ``python3`` (depending on your system) in the command prompt or terminal. This allows you to execute Python code interactively.



c. Django Installation and setup:

1. ****Install Python:****

- Before installing Django, ensure that Python is installed on your system. You can follow the steps mentioned earlier to install Python if you haven't already.

2. ****Create a Virtual Environment (Optional but recommended):****

- It's a good practice to create a virtual environment for each Django project to isolate its dependencies. You can create a virtual environment using the following commands:

```
---
```

```
python3 -m venv myenv
```

```
---
```

Replace `myenv` with the name you want to give to your virtual environment.

3. ****Activate the Virtual Environment:****

- Activate the virtual environment by running the appropriate command based on your operating system:

- On Windows:

```
""  
  
myenv\Scripts\activate  
  
""
```

- On macOS and Linux:

```
""  
  
source myenv/bin/activate  
  
""
```

4. ****Install Django:****

- Once the virtual environment is activated, you can install Django using pip, Python's package manager:

```
""  
  
pip install django  
  
""
```

5. ****Create a Django Project:****

- After Django is installed, you can create a new Django project by running the following command:

```
""  
  
django-admin startproject myproject  
  
""
```

Replace `myproject` with the name you want to give to your Django project.

6. ****Run the Development Server:****

- Navigate to the directory containing your newly created Django project (`myproject` in this case) using the command line.

- Start the development server by running the following command:

```
...
```

```
python manage.py runserver
```

```
...
```

7. ****Access the Django Admin Interface (Optional):****

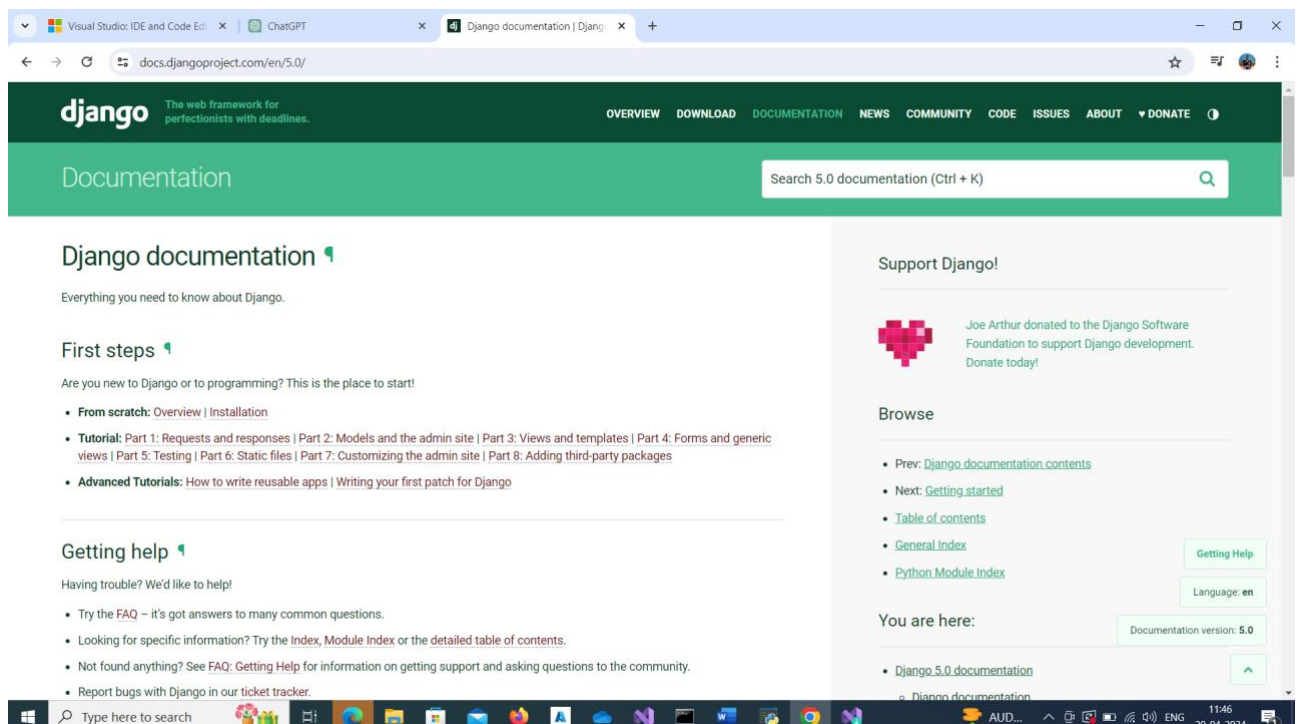
- Django provides a built-in admin interface for managing site content. To access the admin interface, open a web browser and go to `http://127.0.0.1:8000/admin/`. You'll need to create a superuser first by running the following command:

```
...
```

```
python manage.py createsuperuser
```

```
...
```

8. ****Refer Django Documentation:****



3.2 Creating a Django Web Project in Visual Studio

1. ****Install Visual Studio:****

- If you haven't already installed Visual Studio, you can download it from the official website: <https://visualstudio.microsoft.com/downloads/>

2. ****Install Python Development Workload:****

- During the installation of Visual Studio, make sure to select the Python development workload. This will install Python along with necessary tools and extensions for Python development.

3. ****Create a New Django Project:****

- Open Visual Studio.
- Go to `File` > `New` > `Project...` to open the New Project dialog.
- In the search box, type "Django" to filter project templates.
- Select "Django Web Project" from the search results.
- Choose a name and location for your project, then click "Create".

4. ****Configure Django Project Settings:****

- Visual Studio will generate a Django project structure for you. You'll find the project files and folders in the Solution Explorer.
- Open `settings.py` in the `myproject` folder.
- Configure your database settings, static files, templates, etc. as needed. You can refer to the Django documentation for guidance on these settings.

5. ****Set Up Virtual Environment (Optional):****

- It's recommended to use a virtual environment for your Django project to manage dependencies.
- Open the terminal in Visual Studio (View > Terminal).

- Navigate to your project directory and create a virtual environment using the following command:

```
---
```

```
python -m venv venv
```

```
---
```

- Activate the virtual environment:

- On Windows:

```
---
```

```
venv\Scripts\activate
```

```
---
```

- On macOS and Linux:

```
---
```

```
source venv/bin/activate
```

```
---
```

6. ****Install Django and Dependencies:****

- With the virtual environment activated, install Django and any other dependencies your project needs using pip:

```
---
```

```
pip install django
```

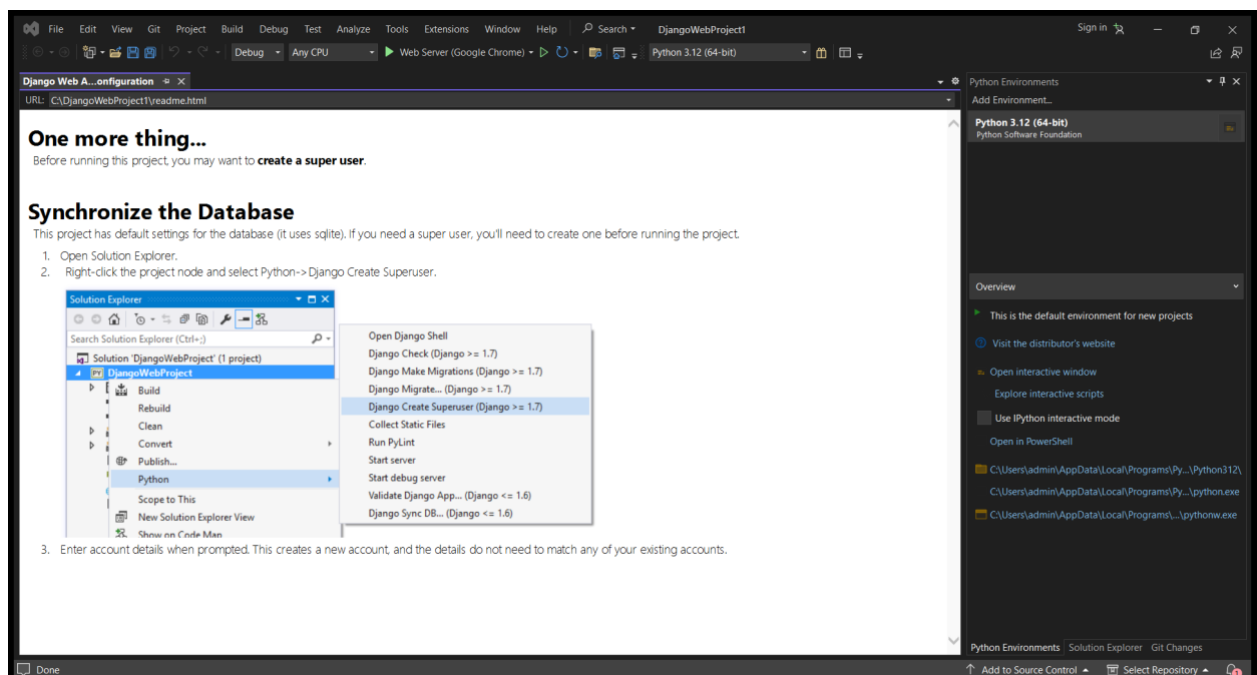
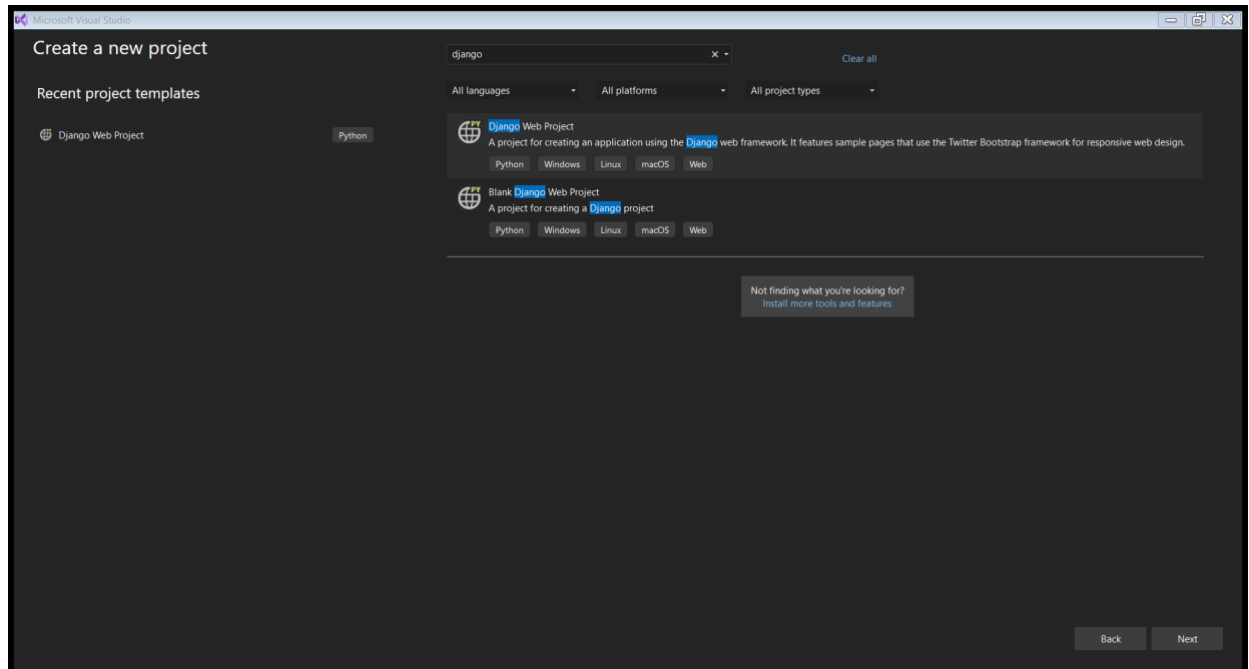
```
---
```

7. ****Run the Django Development Server:****

- You can run the Django development server directly from Visual Studio.

- Go to `Debug` > `Start Without Debugging` (or press `Ctrl + F5`).

- Visual Studio will start the Django development server, and you can access your application by opening a web browser and navigating to the provided URL.



3.3 Start making the changes in the existing template

3.3.1 Setup virtual Environment

Setting up a virtual environment for a Django project in Visual Studio involves a few steps. Here's a guide on how to do it:

1. ****Open Visual Studio:****

- Launch Visual Studio on your computer.

2. ****Open Your Django Project:****

- Open the Django project you want to work on in Visual Studio.

3. ****Open a Terminal:****

- Once your project is open in Visual Studio, open a terminal window within Visual Studio. You can do this by navigating to `View` > `Terminal` in the menu bar.

4. ****Install Virtual Environment:****

- If you haven't already installed virtualenv, you can install it using pip:

```
...
```

```
pip install virtualenv
```

```
...
```

5. ****Create a Virtual Environment:****

- In the terminal, navigate to the directory where you want to create your virtual environment for the Django project.

- Create a new virtual environment using the following command:

```
...
```

```
virtualenv myenv
```

```
...
```

- Replace `myenv` with the name you want to give to your virtual environment.

6. ****Activate the Virtual Environment:****

- Once the virtual environment is created, you need to activate it.

- On Windows:

```
---
```

```
myenv\Scripts\activate
```

```
---
```

- On macOS and Linux:

```
---
```

```
source myenv/bin/activate
```

```
---
```

7. ****Install Django and Other Dependencies:****

- With the virtual environment activated, you can now install Django and any other dependencies your project requires using pip:

```
---
```

```
pip install django
```

```
---
```

8. ****Verify Installation:****

- You can verify that Django is installed correctly by checking its version:

```
---
```

```
python -m django --version
```

```
---
```

3.3.2 Create app entities

a. **admin.py**

In Django, the ``admin.py`` file is used to register models with the Django admin interface. The admin interface is a built-in feature that allows administrators to manage site content without directly manipulating the database or writing custom views.

Here's a breakdown of what you can do with ``admin.py``:

1. ****Registering Models:****

- In `admin.py`, you can import your models and register them with the admin interface using the `admin.site.register()` function. This allows you to manage instances of your models through the Django admin interface.

```
```python
from django.contrib import admin
from .models import MyModel

admin.site.register(MyModel)
```
```

2. **Customizing Model Admin Options:**

- You can customize how your models are displayed and managed in the admin interface by creating a custom `ModelAdmin` class and registering it with the model.

```
```python
from django.contrib import admin
from .models import MyModel

class MyModelAdmin(admin.ModelAdmin):
 list_display = ('field1', 'field2', 'field3')

admin.site.register(MyModel, MyModelAdmin)
```
```

In this example, `list_display` specifies which fields are displayed in the admin list view for `MyModel`.

3. **Inline Models:**

- You can also manage related models inline within the admin interface. This is useful for models with foreign key or many-to-many relationships.

```
```python
from django.contrib import admin
from .models import ParentModel, ChildModel

class ChildModelInline(admin.TabularInline):
 model = ChildModel

class ParentModelAdmin(admin.ModelAdmin):
 inlines = [ChildModelInline]

admin.site.register(ParentModel, ParentModelAdmin)
```
```

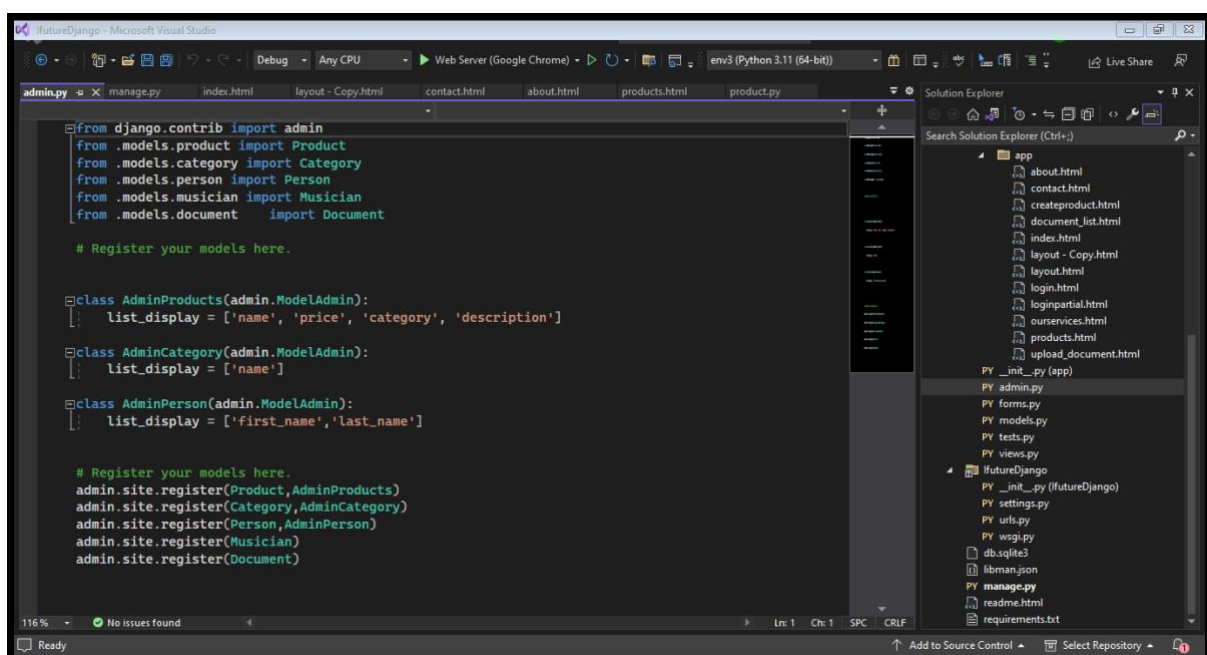
Here, `ChildModelInline` is an inline model admin that allows you to manage `ChildModel` instances within the `ParentModel` admin page.

4. **Customizing Admin Interface:**

- Apart from model-specific configurations, you can also customize the overall behavior and appearance of the admin interface by modifying `AdminSite` or using custom templates.

5. **Permissions and Security:**

- Django's admin interface also handles permissions and security. You can customize who has access to the admin interface and what permissions they have for managing models.



b. forms.py

In Django, the `forms.py` file is used to define forms that are associated with Django models or used independently. Forms are a fundamental part of web applications as they provide a way for users to interact with the application by submitting data.

Here's an overview of what you can do with `forms.py`:

1. **Defining Forms:**

- You define forms in `forms.py` by creating classes that inherit from `django.forms.Form` or `django.forms.ModelForm` depending on whether you want to create a standard form or a form associated with a model.

```
```python
from django import forms
from .models import MyModel

class MyForm(forms.Form):
 name = forms.CharField(max_length=100)
 email = forms.EmailField()

class MyModelForm(forms.ModelForm):
 class Meta:
 model = MyModel
 fields = ['field1', 'field2']
```
```

In this example, `MyForm` is a standard form that contains `name` and `email` fields, while `MyModelForm` is a form associated with the `MyModel` model and includes fields specified in the `Meta` class.

2. ****Customizing Form Fields:****

- You can customize form fields by specifying attributes such as `max_length`, `required`, `widget`, etc. This allows you to control how data is input and validated.

```
```python
class MyForm(forms.Form):
 name = forms.CharField(max_length=100, label='Your Name', required=True)
 email = forms.EmailField(widget=forms.EmailInput(attrs={'placeholder': 'Enter your email'}))
```
```

3. ****Handling Form Submission:****

- Once a form is defined, you can use it in views to handle form submission. In the view, you instantiate the form class and check if the form is valid. If valid, you can process the form data.

```
```python
from django.shortcuts import render
from .forms import MyForm

def my_view(request):
 if request.method == 'POST':
 form = MyForm(request.POST)
 if form.is_valid():
 # Process form data
 name = form.cleaned_data['name']
```
```

```

        email = form.cleaned_data['email']
        # Do something with the data
    else:
        form = MyForm()
    return render(request, 'my_template.html', {'form': form})
...

```

4. ****Rendering Forms in Templates:****

- Forms can be rendered in templates using Django template language. You can access form fields and display them in your HTML template.

```

...html
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Submit</button>
</form>
...

```

This example renders the form fields using the `as_p` method, which displays each form field wrapped in a `

` tag.

5. ****Validation and Error Handling:****

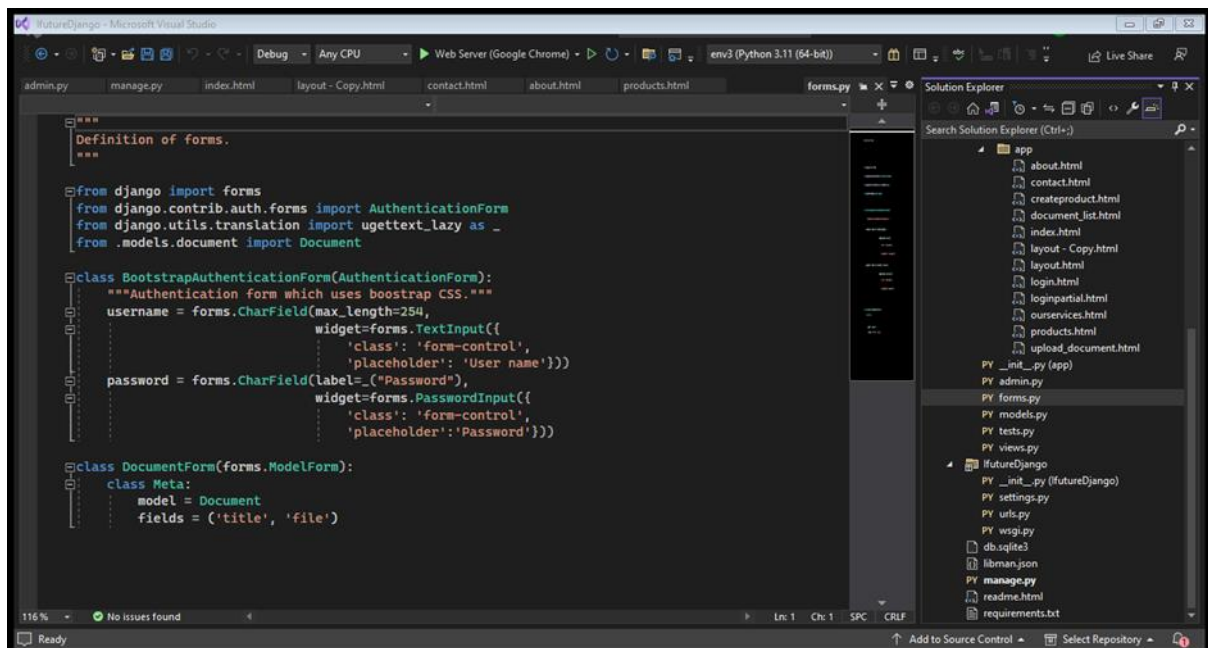
- Django provides built-in form validation. If form data does not pass validation, errors are attached to the form fields, which you can render in your templates to provide feedback to users.

```

...html
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    {% if form.errors %}
        <ul>
            {% for field, error in form.errors.items %}
                <li>{{ field }}: {{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}
    <button type="submit">Submit</button>
</form>
...

```

Overall, `forms.py` allows you to define, customize, and handle forms in your Django application. It helps maintain a separation of concerns by encapsulating form-related logic in a separate module.



c. models.py

In Django, the `models.py` file is where you define your data models, which represent the structure of your application's database tables. Models define the fields and behaviors of the data you're storing, including relationships between different pieces of data.

Here's an overview of what you can do with `models.py`:

1. **Defining Models:**

- Models are defined as Python classes that subclass `django.db.models.Model`. Each attribute of the model represents a database field.

```
```python
from django.db import models

class MyModel(models.Model):
 field1 = models.CharField(max_length=100)
 field2 = models.IntegerField()
 field3 = models.DateTimeField()
```
```

In this example, `MyModel` represents a database table with fields `field1`, `field2`, and `field3`.

2. ****Field Types:****

- Django provides various field types (e.g., `CharField`, `IntegerField`, `DateTimeField`, `ForeignKey`, etc.) that map to corresponding database column types.

3. ****Relationships:****

- You can define relationships between models using `ForeignKey`, `OneToOneField`, and `ManyToManyField`.

```
```python
class Author(models.Model):
 name = models.CharField(max_length=100)

class Book(models.Model):
 title = models.CharField(max_length=100)
 author = models.ForeignKey(Author, on_delete=models.CASCADE)
```
```

In this example, each `Book` has a `ForeignKey` relationship with an `Author`, indicating that each book is written by a single author.

4. ****Meta Options:****

- You can specify model-level metadata using a nested `Meta` class within the model class. This includes options like ordering, database table name, verbose name, etc.

```
```python
class MyModel(models.Model):
 # fields here

 class Meta:
 ordering = ['field1']
 verbose_name_plural = 'My Models'
```
```

5. ****Model Methods and Properties:****

- You can define methods and properties on your model classes to encapsulate logic related to the data.

```
```python
class MyModel(models.Model):
 # fields here

 def get_full_name(self):
 return f"{self.first_name} {self.last_name}"
```
```


...

6. ****Migrations:****

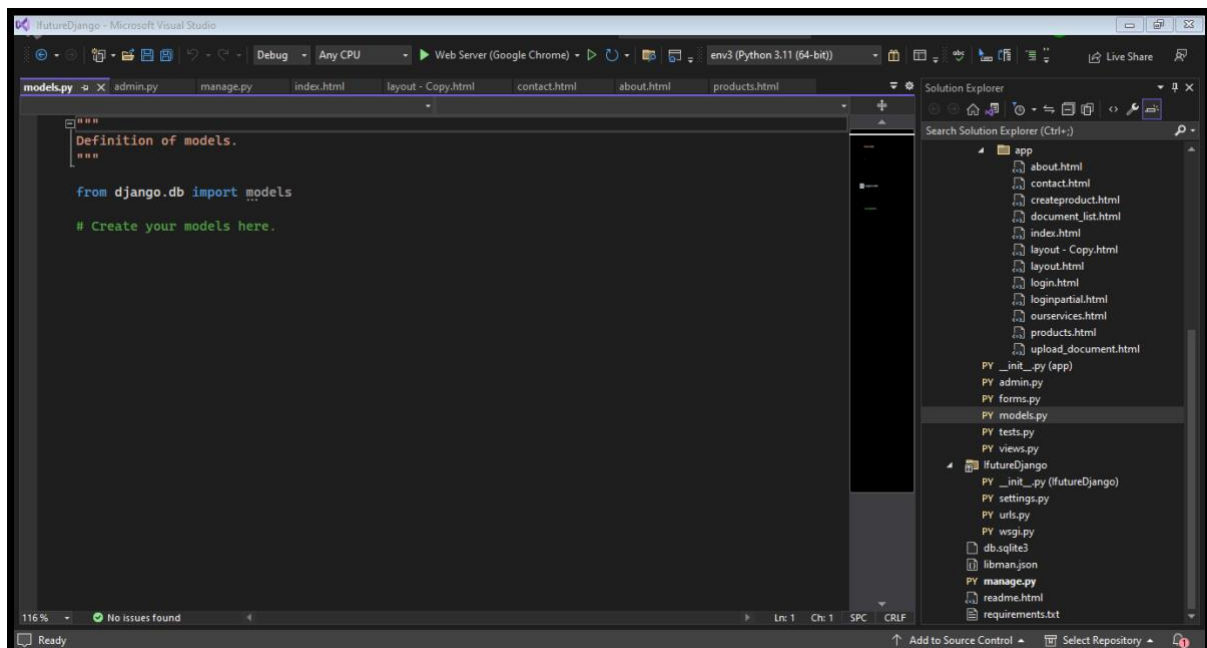
- After defining models, you need to create database tables corresponding to these models. Django's migration system handles this. You generate migrations using the ``makemigrations`` command and apply them using the ``migrate`` command.

```
```bash
python manage.py makemigrations
python manage.py migrate
```
```

7. ****Admin Integration:****

- Django's admin interface automatically introspects your models and provides a simple CRUD interface for managing data. You can further customize the admin interface by registering models in ``admin.py``.

Overall, ``models.py`` is a crucial part of Django development, as it defines the structure of your application's data. It allows you to work with databases in a high-level, Pythonic way, abstracting away the complexities of SQL queries and database management.



d. **tests.py**

In Django, the ``tests.py`` file is where you write tests for your Django applications. Writing tests helps ensure that your code behaves as expected and helps catch bugs early in the development process. Django provides a testing framework that makes it easy to write and run tests for your applications.

Here's an overview of what you can do with `tests.py`:

1. ****Writing Test Cases:****

- Test cases are written as Python classes that subclass `django.test.TestCase`. Each test method within a test case should begin with the word "test" to be recognized as a test method.

```
```python
from django.test import TestCase
from .models import MyModel

class MyModelTestCase(TestCase):
 def setUp(self):
 MyModel.objects.create(field1='test', field2=42)

 def test_model_creation(self):
 obj = MyModel.objects.get(field1='test')
 self.assertEqual(obj.field2, 42)
```
```

In this example, `MyModelTestCase` is a test case that tests the creation of a `MyModel` instance and asserts that the `field2` value is equal to 42.

2. ****Test Fixtures:****

- You can use fixtures to provide initial data for your tests. Fixtures are JSON or XML files containing data that can be loaded into the database before running tests.

```
```python
from django.test import TestCase

class MyTestCase(TestCase):
 fixtures = ['testdata.json']

 def test_something(self):
 # Test code here
```
```

3. ****Running Tests:****

- Django provides management commands to run tests. You can run tests for a specific app, all apps, or specific test cases.

```
```bash
python manage.py test <app_name>
```
```

4. ****Assertions:****

- Django's `TestCase` class provides various assertion methods (`assertEqual`, `assertTrue`, `assertRaises`, etc.) for verifying expected behavior.

5. ****Testing Views:****

- You can test views using Django's `Client` class, which allows you to simulate HTTP requests and test the responses.

```
```python
from django.test import TestCase, Client

class MyViewTestCase(TestCase):
 def test_view(self):
 client = Client()
 response = client.get('/my_url/')
 self.assertEqual(response.status_code, 200)
```
```

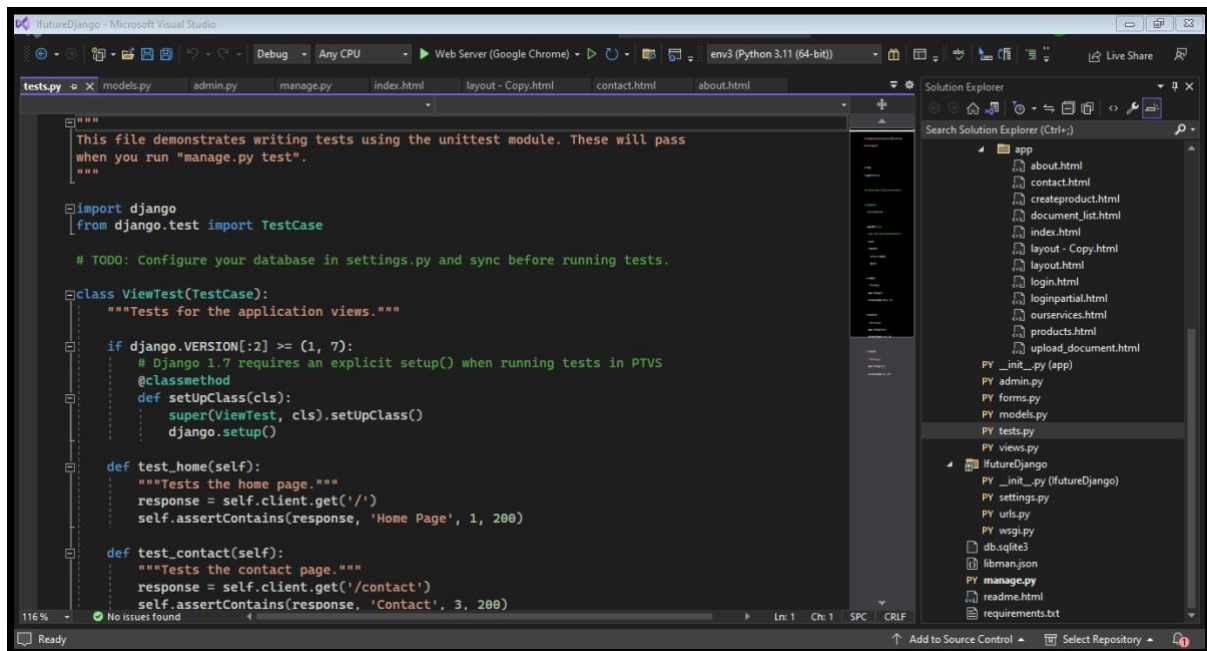
6. ****Testing Forms and Models:****

- You can also write tests to validate forms and model methods.

```
```python
from django.test import TestCase
from .forms import MyForm

class MyFormTestCase(TestCase):
 def test_valid_form(self):
 form_data = {'field1': 'test', 'field2': 42}
 form = MyForm(data=form_data)
 self.assertTrue(form.is_valid())
```
```

Overall, `tests.py` allows you to write automated tests for your Django applications, ensuring that your code behaves correctly and reliably. Writing tests is considered a best practice in software development and helps maintain the quality and integrity of your codebase.



e. views.py

In Django, `views.py` is a Python file within each Django application that contains the views, which are Python functions or classes responsible for processing incoming HTTP requests and returning HTTP responses. Views encapsulate the logic of your application and determine what content is displayed to the user.

Here's a brief overview of `views.py`:

1. ****Function-Based Views (FBVs):****

- Function-based views are simple Python functions that take an HTTP request as input and return an HTTP response. They are defined within `views.py` using the `def` keyword.

```
```python
from django.http import HttpResponse

def my_view(request):
 return HttpResponse("Hello, world!")
```
```

2. ****Class-Based Views (CBVs):****

- Class-based views are Python classes that inherit from Django's `View` class or one of its subclasses. They provide a more structured way to organize view logic and promote reusability.

```
```python
from django.views import View
```

```

from django.http import HttpResponse

class MyView(View):
 def get(self, request):
 return HttpResponse("Hello, world!")
...

```

### 3. **\*\*Request and Response Objects:\*\***

- Views receive an `HttpRequest` object as an argument, which contains information about the incoming request (e.g., GET or POST data, headers, etc.). Views return an `HttpResponse` object, which represents the content to be sent back to the client.

### 4. **\*\*Rendering Templates:\*\***

- Views often render HTML templates to generate dynamic content. Django provides shortcuts like `render()` to simplify the process of rendering templates and passing context data to them.

```

```python
from django.shortcuts import render

def my_view(request):
    context = {'name': 'Alice'}
    return render(request, 'my_template.html', context)
...

```

5. ****View Decorators:****

- Django provides decorators like `@login_required` and `@permission_required` to apply access control to views. Decorators are used to modify the behavior of view functions.

```

```python
from django.contrib.auth.decorators import login_required

@login_required
def restricted_view(request):
 # View logic here
...

```

### 6. **\*\*URL Mapping:\*\***

- Views are mapped to URLs in the `urls.py` file of your Django application. Each URL pattern is associated with a view function or class.

```

```python
from django.urls import path

```

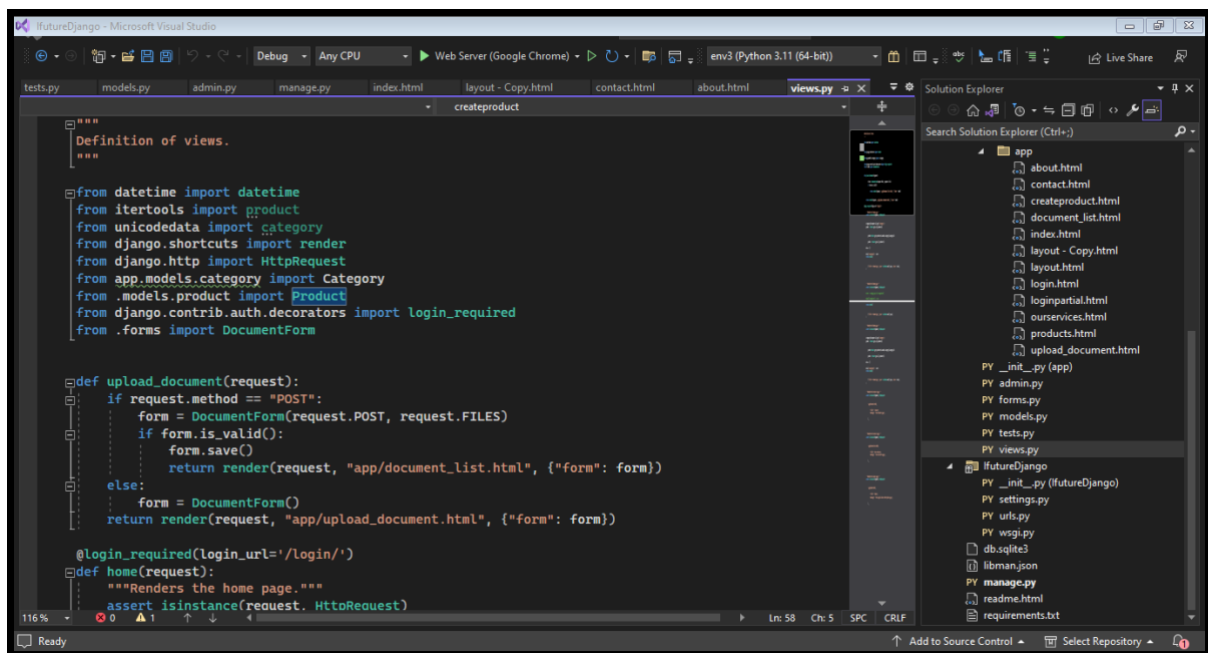
```

from .views import my_view

urlpatterns = [
    path('my-url/', my_view, name='my_view'),
]
...

```

Overall, `views.py` is a crucial component of Django applications, containing the logic that determines how your application responds to user requests. It serves as the bridge between the user's actions and the data displayed by the application.



f. `__init__.py`

In Django, the `__init__.py` file serves as a marker that indicates to Python that the directory containing the file should be considered a Python package. While the `__init__.py` file can be empty, it can also contain initialization code for the package.

Here's a brief overview of `__init__.py` in Django:

1. **Package Initialization:**

- The primary purpose of `__init__.py` is to initialize the package. This file is executed when the package is imported, allowing you to perform any necessary setup actions.

2. **Empty File:**

- In many cases, `__init__.py` may be empty, especially for small Django applications or packages that do not require any initialization code. The presence of the file itself is sufficient to mark the directory as a Python package.

3. **Initialization Code:**

- You can include initialization code in `__init__.py` if your Django application or package requires it. This could include setting up configuration settings, registering signals, importing modules, or performing other setup tasks.

```
```python
Example __init__.py
from .my_module import MyClass

Register signals
from . import signals
```
```

4. ****Relative Imports:****

- `__init__.py` allows you to perform relative imports within your package. This means you can import modules or subpackages relative to the current package.

```
```python
Example relative import in __init__.py
from . import views
```
```

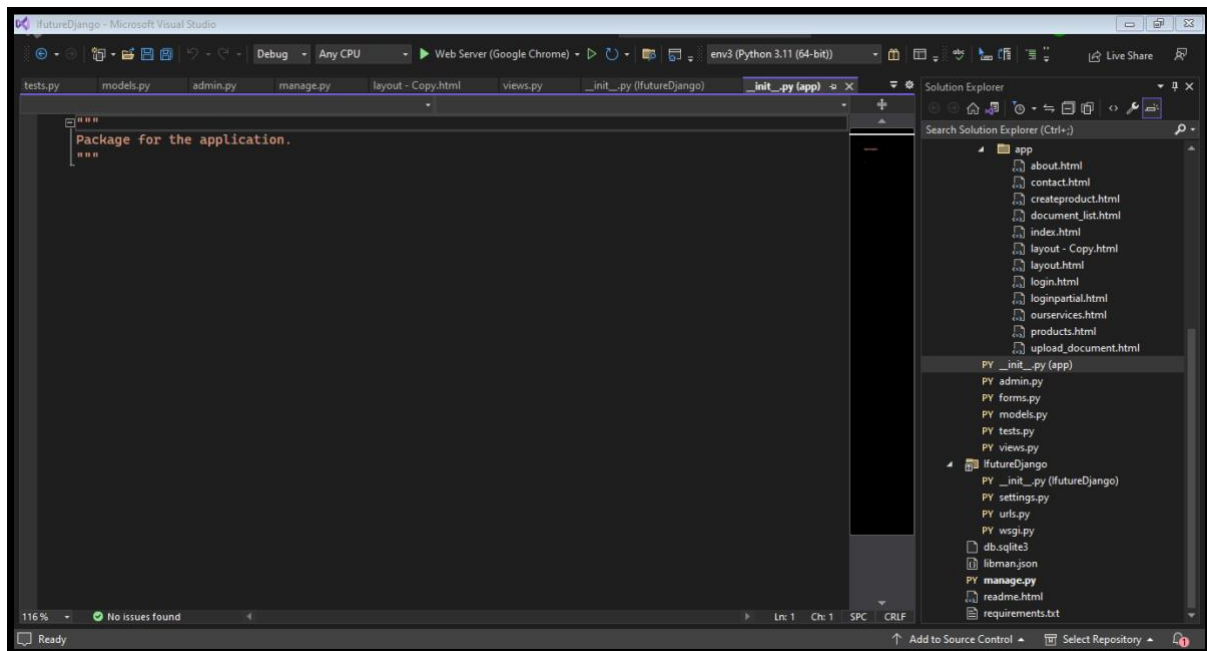
5. ****Namespace Packages:****

- In Python 3.3 and later, `__init__.py` is not required to create a namespace package. A namespace package is a way to split a single Python package across multiple directories.

6. ****Compatibility:****

- While `__init__.py` is still supported in Python 3, it's no longer necessary for defining packages in Python 3.3 and later. However, it's often still included for compatibility with older versions of Python.

Overall, `__init__.py` is a standard file in Python packages, including Django applications and packages. While it may be empty in some cases, it provides a place for initialization code and helps organize the structure of your codebase.



g. settings.py

In Django, `settings.py` is a Python file that contains configuration settings for your Django project. It's a central location where you can customize various aspects of your project, including database configuration, static files, middleware, installed apps, security settings, and much more.

Here's a brief overview of `settings.py`:

1. ****Configuration Settings:****

- `settings.py` contains a dictionary-like object named `settings` that holds all the configuration options for your Django project.

2. ****Database Configuration:****

- You define the database configuration in `settings.py`, including the database engine, connection settings, and other database-specific options.

```
python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

3. ****Installed Apps:****

- Django is composed of reusable apps, and you can specify which apps are installed and enabled for your project in the `INSTALLED_APPS` setting.

```
```python
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'myapp',
]
```
```

4. ****Static Files:****

- You configure static files settings in `settings.py`, including the directories where Django will search for static files (e.g., CSS, JavaScript, images).

```
```python
STATIC_URL = '/static/'
STATICFILES_DIRS = [
 BASE_DIR / "static",
]
```
```

5. ****Middleware:****

- Middleware are hooks that process requests and responses. You can configure middleware classes in `MIDDLEWARE` setting.

```
```python
MIDDLEWARE = [
 'django.middleware.security.SecurityMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.middleware.common.CommonMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware',
 'django.contrib.auth.middleware.AuthenticationMiddleware',
 'django.contrib.messages.middleware.MessageMiddleware',
 'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```
```

6. ****Security Settings:****

- Django provides security settings such as `SECRET_KEY`, `DEBUG`, `ALLOWED_HOSTS`, and more, which you configure in `settings.py`.

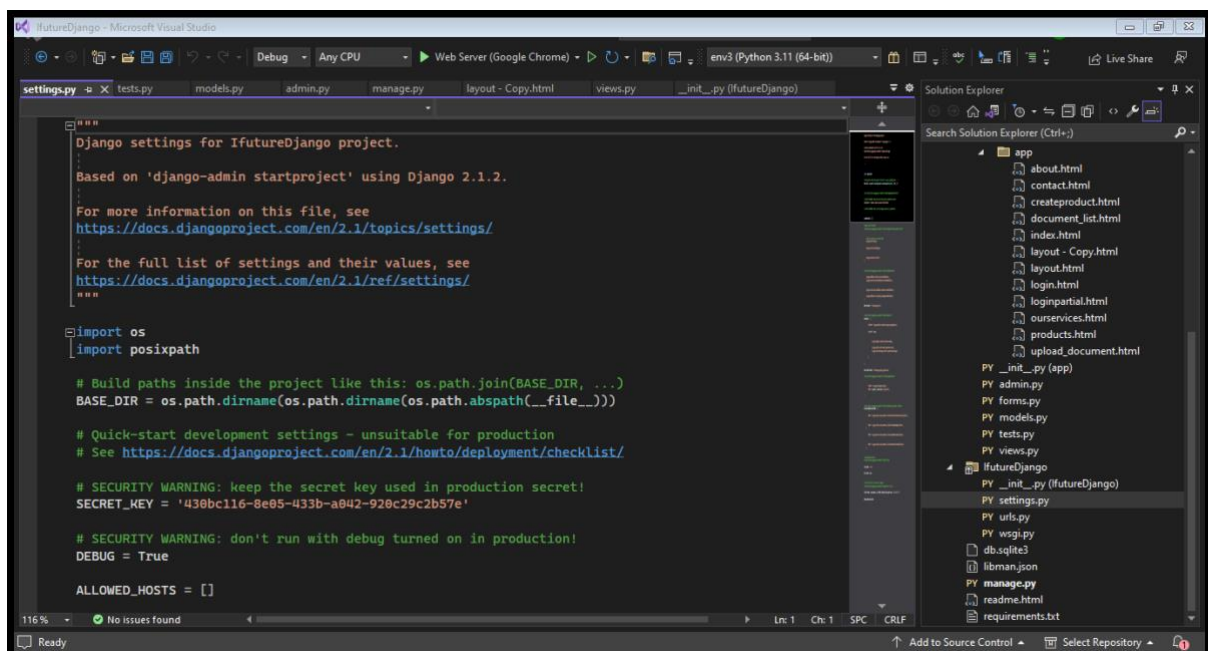
```
```python
SECRET_KEY = 'your_secret_key'
DEBUG = True
ALLOWED_HOSTS = ['localhost', '127.0.0.1']
```
```

7. **Localization and Timezone Settings:**

- `settings.py` allows you to configure localization and timezone settings for your project.

```
```python
LANGUAGE_CODE = 'en-us'
TIME_ZONE = 'UTC'
```
```

Overall, `settings.py` is a crucial file in Django projects, as it allows you to customize and configure various aspects of your application. It's where you define how your project behaves and interacts with other components.



h. urls.py

In Django, `urls.py` is a Python file that contains URL patterns for your Django project. URL patterns map URLs to views, defining the structure of your application's URLs and specifying which views should handle incoming requests.

Here's a brief overview of `urls.py`:

1. **URL Patterns:**

- URL patterns are defined in `urls.py` using the `urlpatterns` list. Each URL pattern is represented as a Python tuple containing a regular expression pattern and the view function or class that should be called when the pattern is matched.

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

urlpatterns = [
 path("", views.index, name='index'),
 path('about/', views.about, name='about'),
 # More URL patterns...
]
```

#### 2. **Path Function:**

- Django provides the `path()` function to define URL patterns using a simpler syntax than regular expressions. The `path()` function takes the URL pattern as the first argument and the view function or class as the second argument.

#### 3. **Regular Expressions:**

- While `path()` is the preferred method for defining URL patterns, you can still use regular expressions with the `re\_path()` function if needed.

```
```python
from django.urls import re_path
from . import views

urlpatterns = [
    re_path(r'^articles/(?P<year>[0-9]{4})/$', views.article_archive),
    # More URL patterns...
]
```

4. **Including Other URL Configurations:**

- You can include URL configurations from other Django apps or modules using the `include()` function.

```
```python
from django.urls import include, path
```

```
urlpatterns = [
 path('myapp/', include('myapp.urls')),
 # More URL patterns...
]
```

#### 5. **\*\*Naming URL Patterns:\*\***

- URL patterns can be named using the `name` argument. Naming URL patterns allows you to refer to them in templates and view functions.

```
python
urlpatterns = [
 path('articles/', views.article_list, name='article_list'),
 path('articles/<int:year>', views.article_detail, name='article_detail'),
 # More URL patterns...
]
```

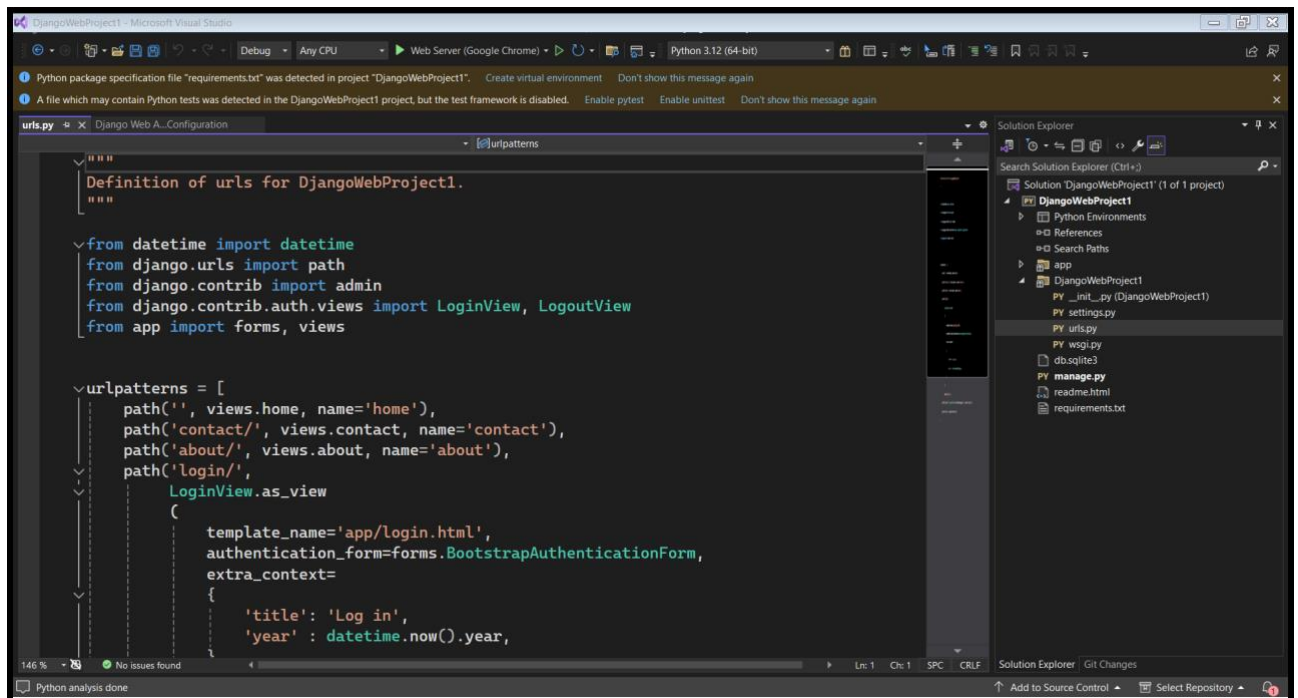
#### 6. **\*\*Reverse URL Resolution:\*\***

- Django provides the `reverse()` function to dynamically generate URLs based on the URL name. This is useful for generating URLs in templates or view functions without hardcoding them.

```
python
from django.urls import reverse

def my_view(request):
 url = reverse('article_detail', args=[2019])
 return redirect(url)
```

Overall, `urls.py` is a fundamental part of Django projects, defining the structure of URLs and routing incoming requests to the appropriate views. It provides a clean and maintainable way to organize your application's URLs and views.



### i. manage.py

In Django, `manage.py` is a command-line utility that provides various tools for managing Django projects. It's automatically generated when you create a new Django project and resides in the project's root directory. `manage.py` allows you to perform common tasks such as running the development server, creating database migrations, managing user authentication, and more.

Here's a brief overview of `manage.py`:

#### 1. **\*\*Running the Development Server:\*\***

- One of the most common tasks performed with `manage.py` is starting the development server. This allows you to run your Django application locally for testing and development purposes.

```
```bash
python manage.py runserver
```
```

This command starts the development server on the default address `127.0.0.1:8000`.

#### 2. **\*\*Creating Django Applications:\*\***

- You can use `manage.py` to create new Django applications within your project. Applications are reusable components that encapsulate related functionality.

```
```bash
python manage.py startapp myapp
```
```

...

This command creates a new Django application named `myapp`.

### 3. **\*\*Database Migrations:\*\***

- Django's migration system allows you to manage changes to your database schema over time. You can use `manage.py` to create new migrations, apply migrations, and inspect the current migration status.

```
```bash
python manage.py makemigrations
python manage.py migrate
```
```

These commands create new migrations based on changes to your models and apply those migrations to the database.

### 4. **\*\*Creating a Superuser:\*\***

- Django's authentication system includes a built-in user model. You can use `manage.py` to create a superuser account, which has access to the Django admin interface and other administrative tasks.

```
```bash
python manage.py createsuperuser
```
```

This command prompts you to enter details for the new superuser account, such as username, email, and password.

### 5. **\*\*Running Tests:\*\***

- Django provides a testing framework for writing and running tests for your applications. You can use `manage.py` to run tests and generate test coverage reports.

```
```bash
python manage.py test
```
```

This command runs all tests defined in your project's test suite.

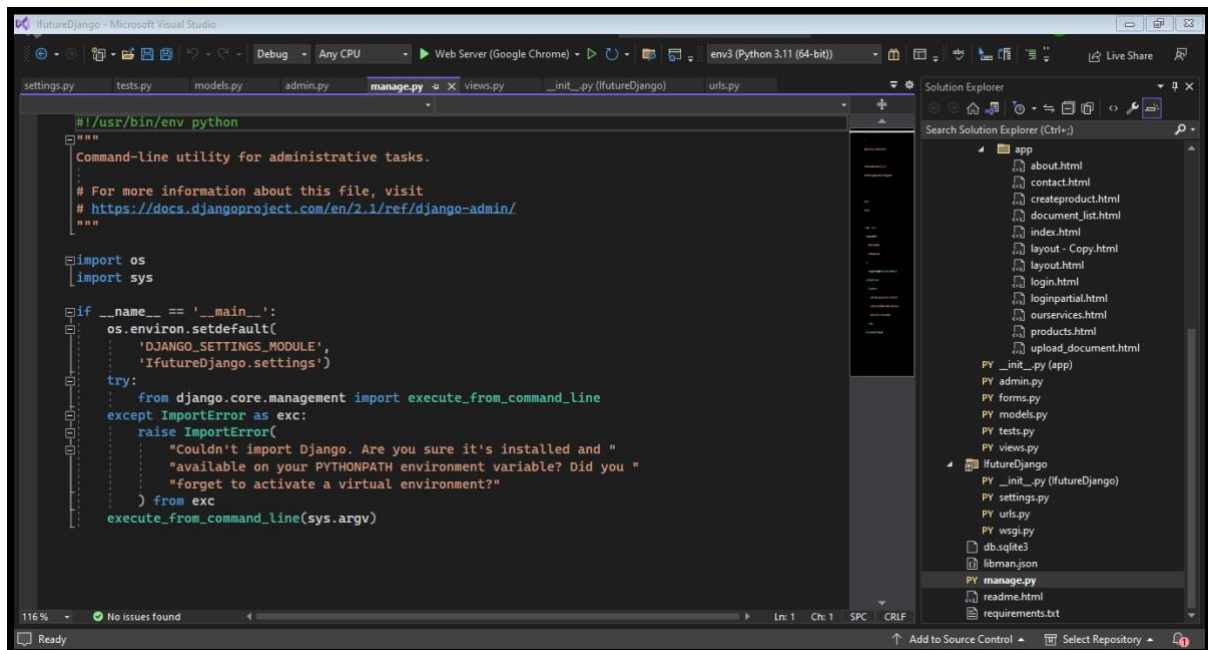
### 6. **\*\*Custom Management Commands:\*\***

- You can define custom management commands in your Django project to perform specific tasks. These commands can be executed using `manage.py`.

### 7. **\*\*Other Tasks:\*\***

- `manage.py` provides many other commands for managing Django projects, such as collecting static files, compressing JavaScript and CSS, running database shell, and more.

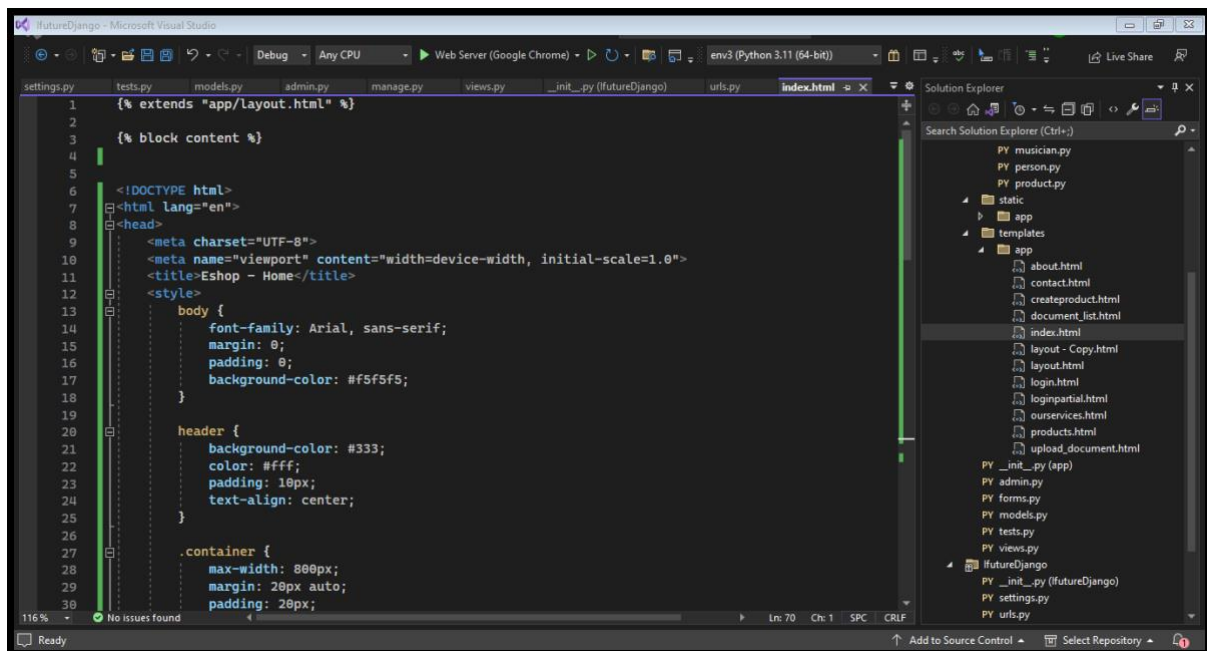
Overall, `manage.py` is a powerful tool for managing Django projects and provides a convenient interface for performing common development and administrative tasks. It's an essential part of the Django framework and is used extensively throughout the development process.



### 3.3.3 Create html documents for the web pages of the app

#### a. index.html

In Django, `index.html` typically refers to a template file used to render the home page or index page of a web application. This HTML file contains the structure and content of the page, including any dynamic data passed from views. It's located within the templates directory of a Django app and is rendered using Django's templating engine to generate dynamic content for the user.



## b. about.html

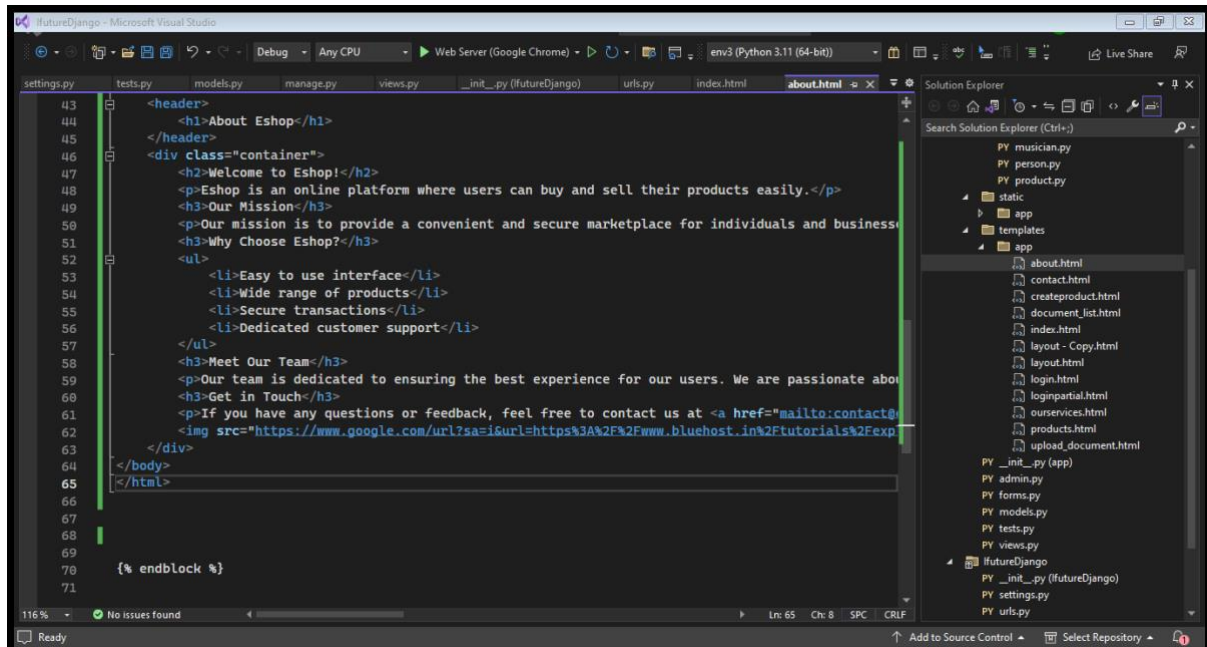
In Django, `about.html` is typically a template file that renders the "About" page of a web application. Template files in Django contain HTML markup along with template tags and filters that allow for dynamic content rendering.

Here's a very short overview of `about.html`:

- **Purpose:** `about.html` is used to display information about the web application, such as its purpose, mission, team, or any other relevant details.
- **Content:** It contains HTML markup to structure the page and may include CSS for styling and JavaScript for interactivity.
- **Dynamic Content:** Django's template language allows you to include dynamic content in `about.html`, such as data from the database, context variables passed from views, or template tags for control flow and logic.
- **Extending Templates:** `about.html` may extend a base template (`base.html`), which defines the common layout and structure shared across multiple pages of the application.



- **URL Mapping:** The URL pattern associated with the "About" page in `urls.py` routes requests to a corresponding view, which renders `about.html` and returns an HTTP response.



### c. contact.html

In Django, `contact.html` is a template file used to render a contact form or page. It typically contains HTML markup along with template tags and filters provided by Django's template engine. The contact form may include fields for the user to input their name, email address, message, etc. The form data is submitted to a Django view for processing, which may involve sending an email or storing the message in a database. Overall, `contact.html` plays a crucial role in allowing users to contact the website owner or administrators.

```

1 {% extends "app/layout.html" %}
2
3 {% block content %}
4
5 <!DOCTYPE html>
6 <html lang="en">
7 <head>
8 <meta charset="UTF-8">
9 <meta name="viewport" content="width=device-width, initial-scale=1.0">
10 <title>Contact Eshop</title>
11 <style>
12 body {
13 font-family: Arial, sans-serif;
14 margin: 0;
15 padding: 0;
16 background-color: #f8f8f8;
17 }
18
19 header {
20 background-color: #333;
21 color: #fff;
22 padding: 20px;
23 text-align: center;
24 }
25
26 .container {
27 max-width: 800px;
28 margin: 20px auto;
29 padding: 20px;
30 background-color: #fff;
31 }
32 </style>
33
34 <div class="container">
35 <div class="header">
36 <h2>Contact Us</h2>
37 </div>
38 <div class="form">
39 <form>
40 <input type="text" value="Name" />
41 <input type="text" value="Email" />
42 <input type="text" value="Subject" />
43 <input type="text" value="Message" />
44 <input type="button" value="Submit" />
45 </form>
46 </div>
47 </div>
48
49 </body>
50 </html>
51
52 {% endblock %}

```

#### d. createproduct.html

In Django, `createproduct.html` is a template file responsible for rendering a form to create a new product. This HTML file typically contains form elements such as input fields, dropdown menus, buttons, etc., allowing users to input information about the new product. Upon submission, the form data is sent to a Django view for processing, where it can be validated and saved to the database.

```

1 {% extends "app/layout.html" %}
2
3 {% block content %}
4
5 <div class="row">
6 <div class="col-md-6 card border-primary mb-3">
7 <div class="card-header">Product operation</div>
8 <div class="card-body">
9 <div class="col-md-8">
10 <hr />
11 <form action="/createproduct/">
12 <div class="mb-3 row">
13 <div>
14 <input type="text" class="form-control" placeholder="Enter product Name" />
15 </div>
16 </div>
17 <div class="mb-3 row">
18 <div>
19 <input type="text" class="form-control" placeholder="Enter product description" />
20 </div>
21 </div>
22 <div class="mb-3 row">
23 <div>
24 <input type="text" class="form-control" placeholder="Enter product price" />
25 </div>
26 </div>
27 <div class="mb-3 row">
28 <div>
29 <input type="text" class="form-control" placeholder="Enter product category" />
30 </div>
31 </div>
32 <input type="button" value="Create Product" />
33 </form>
34 </div>
35 </div>
36 </div>
37
38 {% endblock %}

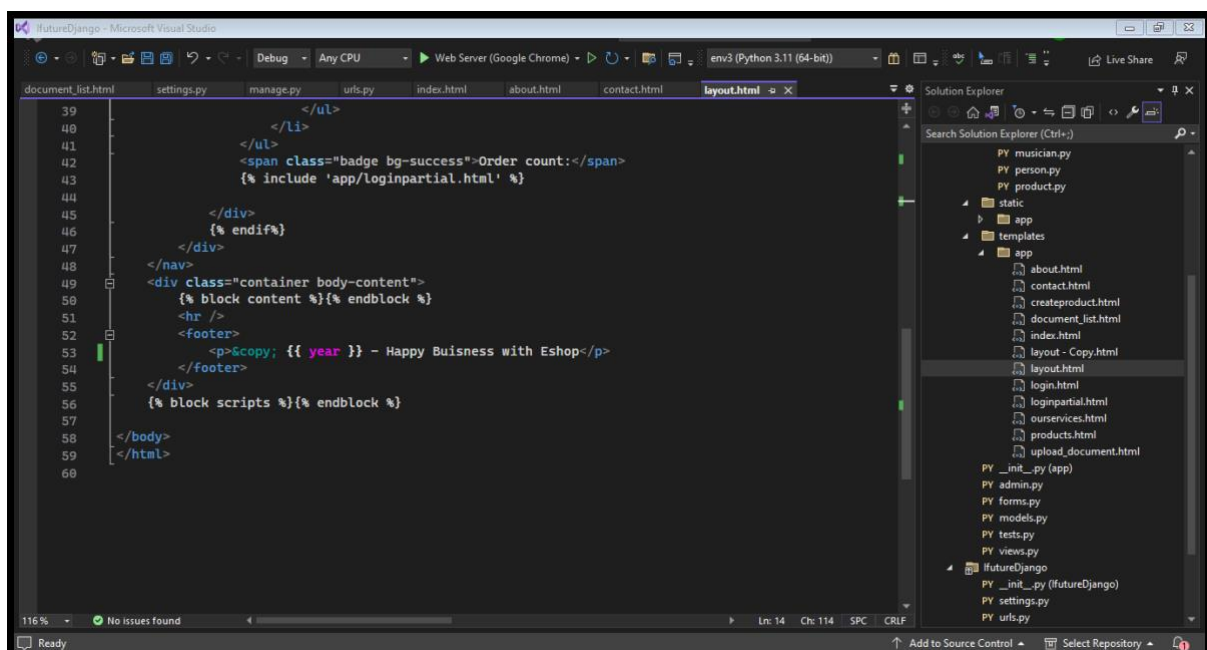
```

### e. layout.html

In Django, `layout.html` is a template file that serves as the base layout for other templates in your project. It typically contains the common HTML structure, including the ``, ``, ``, ``, and other shared elements of your website.

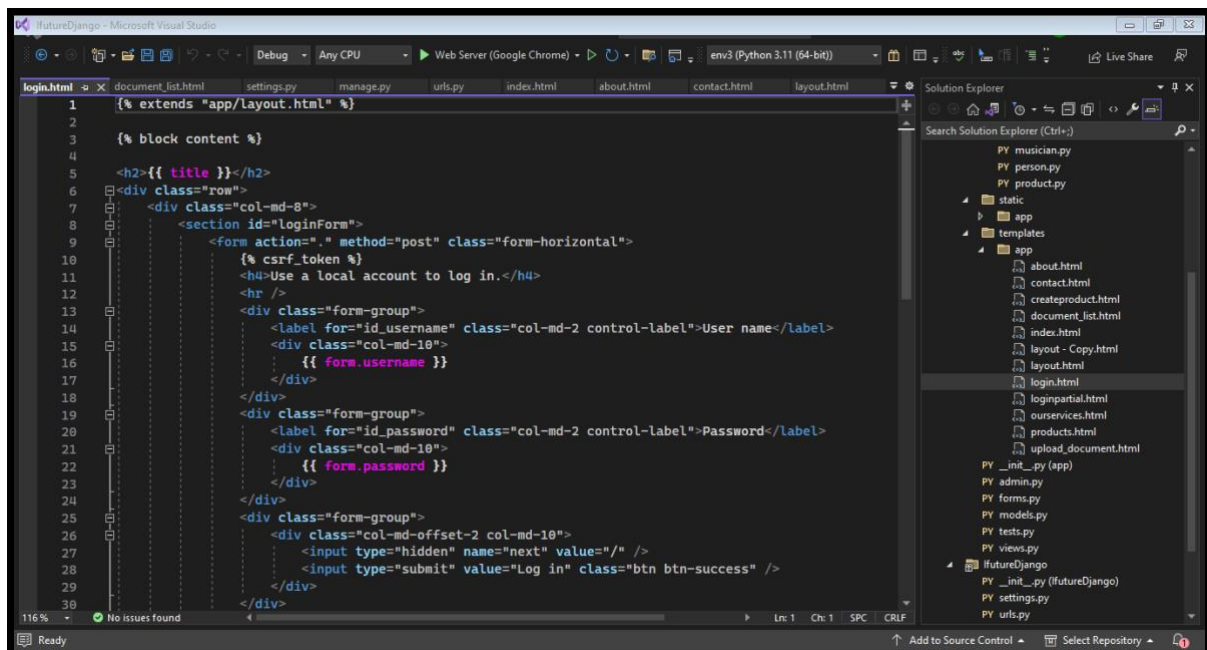
Here's a very short overview:

- **Purpose:** `layout.html` provides a consistent layout and structure for your web pages, reducing duplication and promoting code reuse across multiple templates.
- **Contents:** It usually includes the HTML structure that remains consistent across all pages of your website, such as the header, navigation menu, footer, and any other shared elements.
- **Extending:** Other templates in your project can extend `layout.html` using the Django template language's `{% extends %}` tag, allowing them to inherit its layout and structure while providing content specific to each page.
- **Customization:** `layout.html` can be customized to include dynamic content or to accommodate different layouts for specific pages or sections of your website.



## f. login.html

`login.html` in Django is a template file used to render the login page for user authentication. It typically contains HTML markup along with Django template language tags to handle form rendering, validation errors, and CSRF protection. This file is used in conjunction with Django's authentication views and forms to provide users with a login interface for accessing protected areas of the application.



```

1 {% extends "app/layout.html" %}
2
3 {% block content %}
4
5 <h2>{{ title }}</h2>
6 <div class="row">
7 <div class="col-md-8">
8 <section id="LoginForm">
9 <form action="." method="post" class="form-horizontal">
10 {{ csrf_token }}
11 <h4>Use a local account to log in.</h4>
12 <hr />
13 <div class="form-group">
14 <label for="id_username" class="col-md-2 control-label">User name</label>
15 <div class="col-md-10">
16 {{ form.username }}
17 </div>
18 </div>
19 <div class="form-group">
20 <label for="id_password" class="col-md-2 control-label">Password</label>
21 <div class="col-md-10">
22 {{ form.password }}
23 </div>
24 </div>
25 <div class="form-group">
26 <div class="col-md-offset-2 col-md-10">
27 <input type="hidden" name="next" value="/" />
28 <input type="submit" value="Log in" class="btn btn-success" />
29 </div>
30 </div>
31 </section>
32 </div>
33 </div>
34 {% endblock %}

```

## g. ourservices.html

In Django, `ourservices.html` would typically be an HTML template file used to display information about the services offered by a Django application. It's part of the templates directory within a Django app and is rendered dynamically to present services-related content to users. The template may include HTML markup along with Django template tags and variables to dynamically generate content based on data retrieved from the backend.

```

444 }
445 }
446 </style>
447 </head>
448 <body>
449 <header>
450 <h1>Eshop Services</h1>
451 </header>
452 <div class="container">
453 <h2>Our Services</h2>
454
455
456 <h3>Product Listing</h3>
457 <p>List your products for sale on our platform easily.</p>
458
459
460 <h3>Product Search</h3>
461 <p>Effortlessly find products you want to buy using our advanced search feature.</p>
462
463
464 <h3>Secure Transactions</h3>
465 <p>Ensure safe and secure transactions when buying or selling products.</p>
466
467
468 <h3>Feedback and Reviews</h3>
469 <p>Leave feedback and reviews to help build trust within the community.</p>
470
471
472 <h3>Customer Support</h3>
473 <p>24/7 customer support to assist you with any queries or issues.</p>
474
475
476 </div>
477 </body>
478 </html>

```

## h. products.html

In Django, `products.html` is a HTML template file used to render product-related content in a Django web application. This template typically contains HTML markup along with template tags and filters provided by Django's template engine. It's used to present information about products retrieved from the backend to the end user in a visually appealing and interactive manner.

```

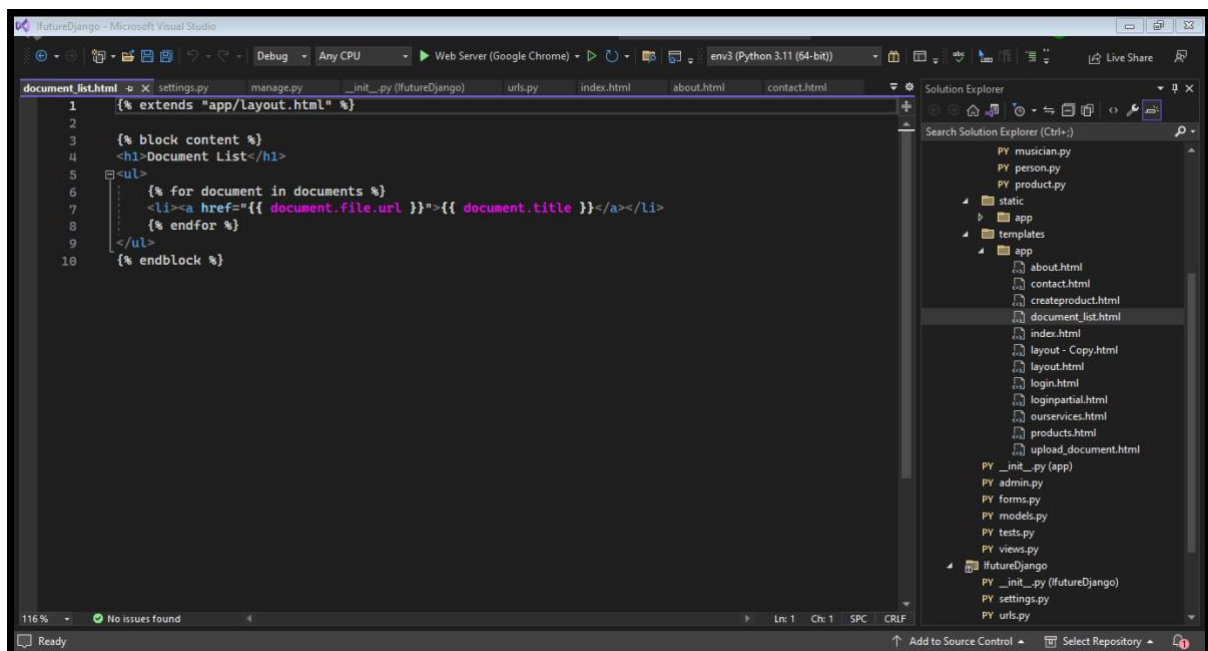
1 {% extends "app/layout.html" %}
2
3 {% block content %}
4
5 <div class="container">
6 <div class="row">
7
8 <div class="col-md-2">
9
10 <ul class="list-group">
11 All
12 {% for cat in data.categories %}
13 <li class="list-group-item">{{cat.name}}
14
15 {% endfor %}
16
17 </div>
18 <div class="col-md-10">
19 <div class="row">
20 <div class="card col-md-3">
21 <div class="card-header">{{prod.name}}</div>
22 <div class="card-body">
23
24 <p class="card-text">{{prod.description}}</p>
25 <p class="card-text">Price: {{prod.price}}</p>
26 </div>
27 </div>
28 </div>
29 </div>
30 </div>
31 </div>
32 </div>
33 </div>
34 </div>
35 </div>
36 </div>
37 </div>
38 </div>
39 </div>
40 </div>
41 </div>
42 </div>
43 </div>
44 </div>
45 </div>
46 </div>
47 </div>
48 </div>
49 </div>
50 </div>
51 </div>
52 </div>
53 </div>
54 </div>
55 </div>
56 </div>
57 </div>
58 </div>
59 </div>
60 </div>
61 </div>
62 </div>
63 </div>
64 </div>
65 </div>
66 </div>
67 </div>
68 </div>
69 </div>
70 </div>
71 </div>
72 </div>
73 </div>
74 </div>
75 </div>
76 </div>
77 </div>
78 </div>
79 </div>
80 </div>
81 </div>
82 </div>
83 </div>
84 </div>
85 </div>
86 </div>
87 </div>
88 </div>
89 </div>
90 </div>
91 </div>
92 </div>
93 </div>
94 </div>
95 </div>
96 </div>
97 </div>
98 </div>
99 </div>
100 </div>

```



### i. upload\_document.html

`uploadDocument.html` in Django is likely an HTML template file used for rendering a web page where users can upload documents. It would contain HTML markup along with template tags to dynamically generate content or handle form submissions. Typically, it's associated with a Django view that processes the document uploads and saves them to the server or performs other actions as needed.



### 3.3.4 Start Server on the Web Browser

To start the Django server in Visual Studio, you can follow these steps:

#### 1. **\*\*Open Visual Studio:\*\***

- Launch Visual Studio on your computer.

#### 2. **\*\*Open Your Django Project:\*\***

- Open the Django project you want to work on in Visual Studio. You can do this by selecting "Open a project or solution" from the Visual Studio start page or by navigating to the project directory and opening the project file.

### 3. **\*\*Open a Terminal:\*\***

- Once your project is open in Visual Studio, open a terminal window within Visual Studio. You can do this by navigating to `View` > `Terminal` in the menu bar.

### 4. **\*\*Navigate to Your Project Directory:\*\***

- Use the terminal to navigate to the directory containing your Django project.

### 5. **\*\*Activate Virtual Environment (Optional):\*\***

- If you're using a virtual environment for your Django project, activate it in the terminal. For example, on Windows:

```

```

```
myenv\Scripts\activate
```

```

```

- On macOS and Linux:

```

```

```
source myenv/bin/activate
```

```

```

### 6. **\*\*Start the Django Server:\*\***

- Once you're in your project directory and the virtual environment is activated (if applicable), use the `manage.py` script to start the Django development server:

```

```

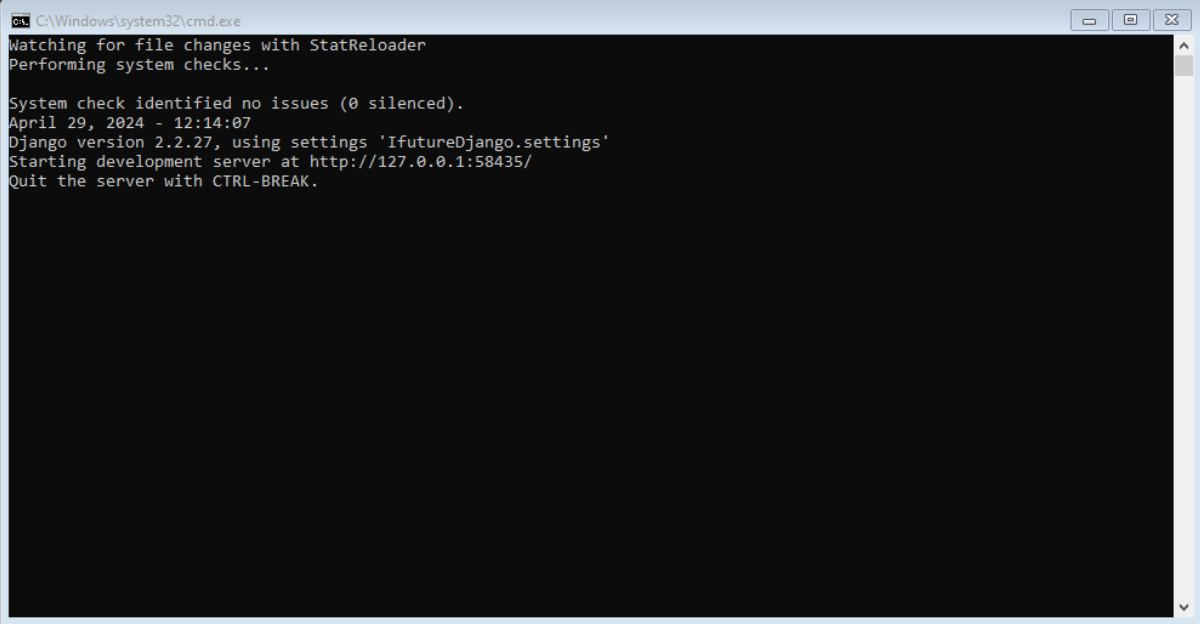
```
python manage.py runserver
```

```

```

### 7. **\*\*Access Your Application:\*\***

- After starting the server, you should see output indicating that the server is running. You can then access your Django application by opening a web browser and navigating to the URL provided in the output (usually `http://127.0.0.1:8000/` by default).



```
C:\Windows\system32\cmd.exe
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 29, 2024 - 12:14:07
Django version 2.2.27, using settings 'IfutureDjango.settings'
Starting development server at http://127.0.0.1:58435/
Quit the server with CTRL-BREAK.
```

### 3.3.5 Django Administration

The Django admin login page provides a secure way for administrators to access the Django admin interface, where they can manage site content, user accounts, and other administrative tasks. By default, Django provides built-in authentication views for logging in and out of the admin interface.

Here's how the Django admin login process works:

#### 1. **\*\*Accessing the Admin Login Page:\*\***

- To access the Django admin login page, navigate to the `/admin`` URL of your Django application in a web browser. For example: `http://yourdomain.com/admin``.
- If you're developing locally, the URL might be `http://127.0.0.1:8000/admin``.

#### 2. **\*\*Entering Credentials:\*\***

- Once you access the admin login page, you'll be presented with a login form.
- Enter the username and password of a user with administrative privileges.
- By default, Django creates a superuser account during the project setup process. You can use these credentials to log in for the first time.



### 3. **\*\*Authentication Process:\*\***

- When you submit the login form, Django validates the provided username and password against the user accounts stored in the database.
- If the credentials are correct and the user account has administrative privileges, Django authenticates the user and grants access to the admin interface.

### 4. **\*\*Session Creation:\*\***

- Upon successful authentication, Django creates a session for the user. This session allows the user to remain authenticated as they navigate through different admin pages without needing to log in again.

### 5. **\*\*Accessing Admin Interface:\*\***

- After logging in, you'll be redirected to the Django admin interface, where you can manage various aspects of your Django application.

### 6. **\*\*Logout Process:\*\***

- To log out of the admin interface, you can click the "Log out" link typically found in the top right corner of the admin interface.
- Django clears the user's session and logs them out, requiring them to log in again to access the admin interface.

Overall, the Django admin login provides a secure authentication mechanism for accessing the admin interface, ensuring that only authorized users can perform administrative tasks on your Django application.

WhatsApp Site administration | Django site

127.0.0.1:58435/admin/?next=/  
Django administration WELCOME, AKHIL VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

APP

|            |       |        |
|------------|-------|--------|
| Categories | + Add | Change |
| Documents  | + Add | Change |
| Musicians  | + Add | Change |
| Persons    | + Add | Change |
| Products   | + Add | Change |

AUTHENTICATION AND AUTHORIZATION

|        |       |        |
|--------|-------|--------|
| Groups | + Add | Change |
| Users  | + Add | Change |

Recent actions

My actions

- Document object (1)  
Document
- Document object (1)  
Document
- Kids  
Category
- Product object (7)  
Product
- Product object (3)  
Product
- Product object (1)  
Product
- Product object (6)  
Product
- Product object (13)  
Product
- Product object (10)  
Product
- Product object (13)  
Product

Activate Windows  
Go to Settings to activate Windows.

WhatsApp Change product | Django site

127.0.0.1:58435/admin/app/product/7/change/  
Django administration WELCOME, AKHIL VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > App > Products > Product object (7)

Change product HISTORY

Name: Black and white top

Price: 799

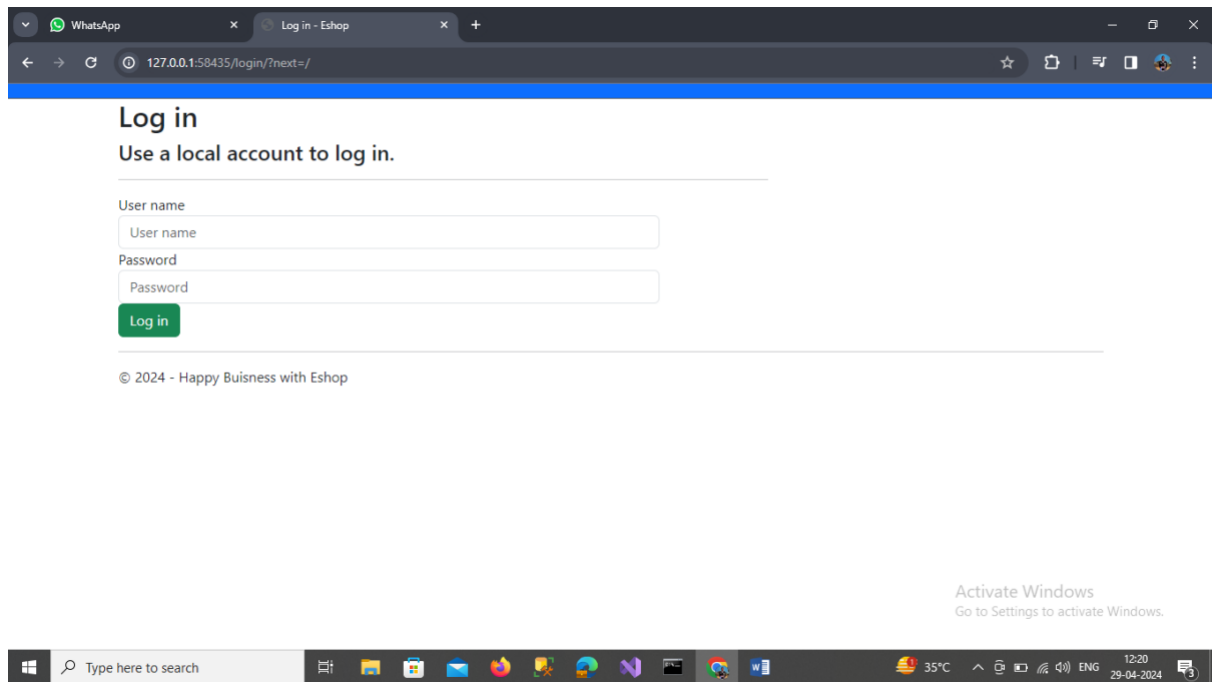
Category: Female

Description: Black and white full sleeve top for women

Image: Currently: products/blackandwhitetop.jpg  
Change: Choose file No file chosen

Delete Save and add another Save and continue editing SAVE

Activate Windows  
Go to Settings to activate Windows.



## CHAPTER 4

### PERFORMANCE AND RESULTS

#### 4.1 Implementation and Results

Implementing an e-commerce application like an online shop (eshop) using Django involves several key components and considerations. Here's an overview of the implementation process and potential results of such a web application:

##### 1. **Database Models:**

- Define Django models to represent products, categories, orders, customers, and other relevant entities.
- Implement relationships between models, such as ForeignKey and ManyToManyField, to represent associations between different entities.

##### 2. **User Authentication and Authorization:**

- Implement user authentication and authorization to allow customers to create accounts, log in, and perform actions like adding items to their cart, placing orders, and viewing order history.
- Use Django's built-in authentication system or integrate third-party authentication solutions if needed.

##### 3. **Product Catalog:**

- Develop views and templates to display the product catalog, including product listings, search functionality, product details, and product filtering by category or attributes.
- Implement features like pagination to manage large product catalogs efficiently.

##### 4. **Shopping Cart:**

- Implement a shopping cart functionality that allows users to add products to their cart, update quantities, remove items, and proceed to checkout.
- Use sessions or databases to store cart information for each user.

#### 5. **Checkout Process:**

- Implement a multi-step checkout process that collects shipping information, payment details, and order confirmation.
- Integrate with payment gateways to handle secure payment processing, such as credit card payments, PayPal, Stripe, etc.

#### 6. **Order Management:**

- Develop views and templates to allow administrators to manage orders, including order processing, order status updates, order fulfillment, and generating order reports.
- Provide order tracking functionality for customers to monitor the status of their orders.

#### 7. **Frontend Design and User Experience:**

- Design responsive and user-friendly frontend interfaces using HTML, CSS, and JavaScript frameworks like Bootstrap or Tailwind CSS.
- Optimize the user experience for easy navigation, intuitive product search, and seamless checkout process.

#### 8. **Security Considerations:**

- Implement security measures to protect sensitive data, such as encrypting passwords, securing communication channels with HTTPS, and implementing CSRF protection.
- Follow security best practices to prevent common vulnerabilities like SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).

#### 9. **Scalability and Performance:**

- Optimize database queries, use caching mechanisms, and employ techniques like lazy loading to improve application performance and scalability.
- Monitor application performance and use profiling tools to identify and address performance bottlenecks.

#### 10. **Testing and Quality Assurance:**

- Write unit tests and integration tests to verify the functionality of different application components.
- Conduct user acceptance testing (UAT) to ensure that the application meets the requirements and expectations of end-users.

#### 11. **Deployment and Maintenance:**

- Deploy the application to a production environment using platforms like AWS, Heroku, or DigitalOcean.
- Monitor application uptime, performance metrics, and security threats.
- Regularly update dependencies, apply security patches, and perform backups to maintain the health and security of the application.

#### Potential Results:

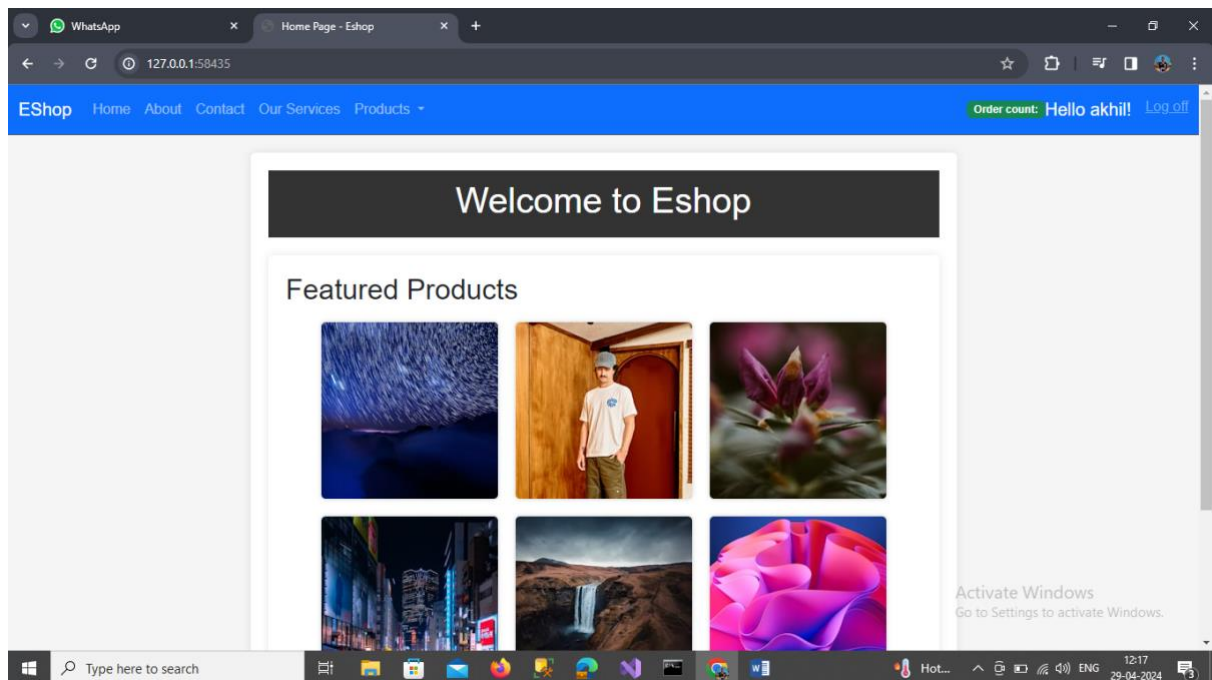
- A fully functional e-commerce website with features like product browsing, shopping cart management, secure checkout process, and order management.
- Improved customer experience and satisfaction through intuitive user interfaces and streamlined shopping workflows.
- Increased sales and revenue through effective product presentation, targeted promotions, and personalized recommendations.
- Enhanced operational efficiency for administrators through automated order processing, inventory management, and reporting tools.

- Expanded market reach and customer base by providing a convenient online shopping experience accessible from desktop and mobile devices.
- Valuable insights into customer behavior, purchasing patterns, and market trends through data analytics and reporting features.

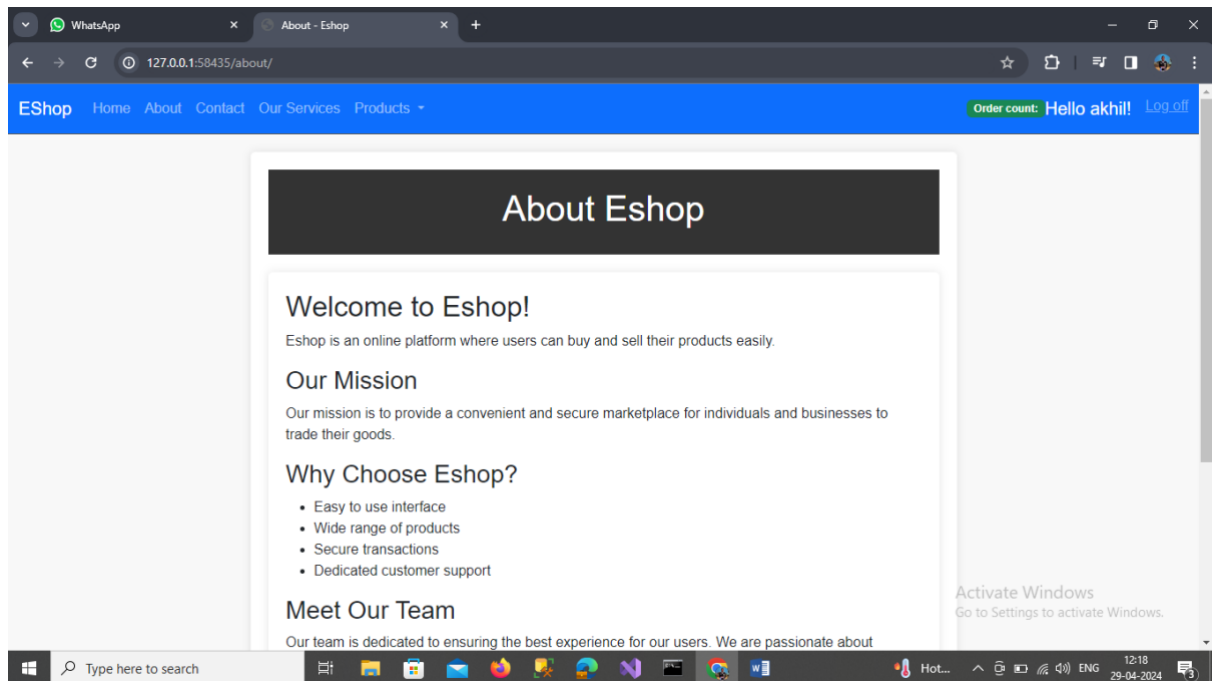
Overall, implementing an e-commerce application using Django can result in a robust and scalable platform for selling products online, providing value to both customers and business owners alike.

## 4.2 Output

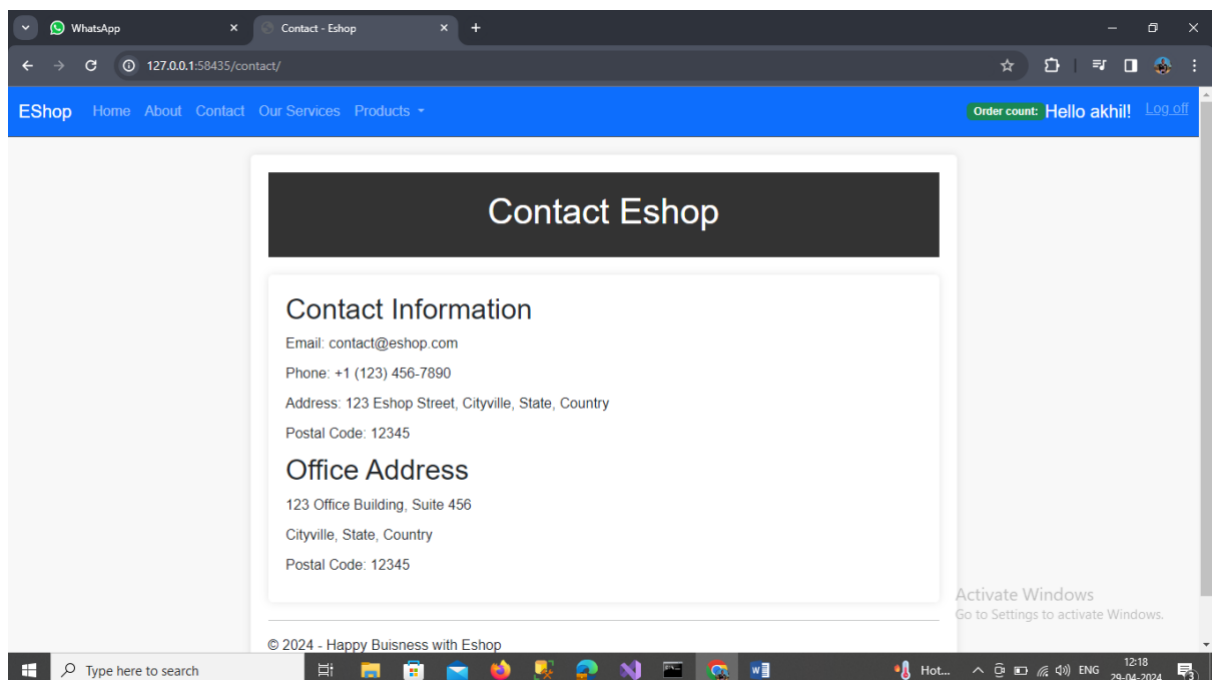
Here are some screen clippings of the Web App Eshop



The image shows the home page

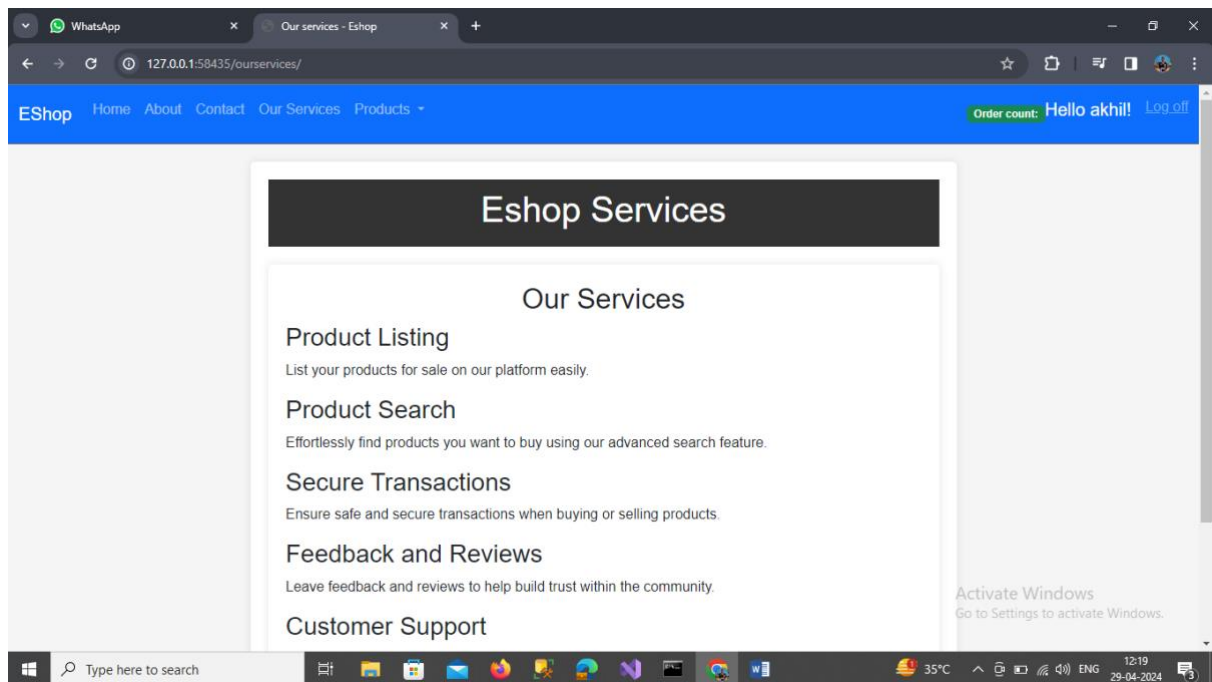


The image shows the About page

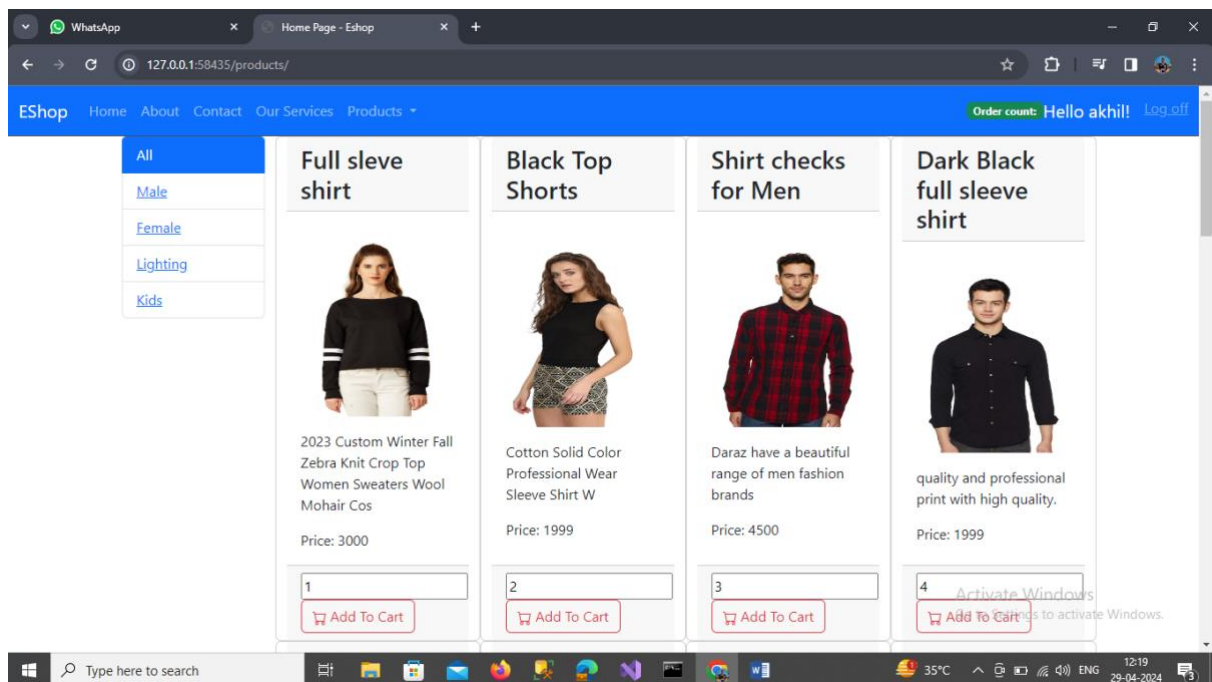


The image shows the Contact page





The image shows the Services provided by Eshop



The image shows the products page

## CHAPTER 5

### APPLICATIONS

#### 1. **\*\*Product Management:\*\***

- This application allows sellers to manage their products. Sellers can add new products, update product details (such as title, description, price, and quantity), and delete products that are no longer available.

#### 2. **\*\*User Authentication and Authorization:\*\***

- Implement authentication functionality to allow users to create accounts, log in, and log out securely. Ensure that only authenticated users have access to certain features, such as adding products to the cart or placing orders.

#### 3. **\*\*Product Catalog:\*\***

- Develop a catalog application that displays a list of available products to users. Users should be able to browse products by category, search for specific products, and view detailed information about each product, including images, descriptions, and prices.

#### 4. **\*\*Shopping Cart:\*\***

- Implement a shopping cart application that allows users to add products to their cart while browsing the catalog. Users should be able to view their cart, update quantities, remove items, and proceed to checkout when ready.

#### 5. **\*\*Checkout Process:\*\***

- Develop a checkout application that guides users through the process of completing their purchase. This includes collecting shipping information, payment details, and order confirmation. Integrate with payment gateways to securely process payments.

**6. \*\*Order Management:\*\***

- Create an order management application that allows users to view their order history and track the status of their orders. Sellers should be able to manage orders, update order statuses, and fulfill orders by shipping products to customers.

**7. \*\*User Profile:\*\***

- Implement a user profile application where users can view and update their personal information, such as shipping addresses, contact details, and payment methods. This application can also display order history and saved preferences.

**8. \*\*Reviews and Ratings:\*\***

- Develop a reviews and ratings application that allows users to leave feedback and ratings for products they have purchased. This helps build trust and credibility for sellers and provides valuable insights for other users.

**9. \*\*Messaging and Notifications:\*\***

- Implement messaging functionality that allows users to communicate with sellers, ask questions about products, and receive notifications about order updates, promotions, and other relevant information.

**10. \*\*Admin Dashboard:\*\***

- Create an admin dashboard application for site administrators to manage users, products, orders, and other site settings. Admins should have access to advanced features for monitoring site activity, analyzing sales data, and resolving disputes.

**11. \*\*Search and Filtering:\*\***

- Develop search and filtering functionality to help users find products more easily. Users should be able to filter products by various criteria, such as price range, brand, size, color, and availability.

**12. \*\*Recommendation Engine:\*\***

- Implement a recommendation engine that suggests products to users based on their browsing and purchasing history. This can help increase sales and provide a personalized shopping experience.

By including these applications/modules in your e-commerce web application, you can create a comprehensive platform that enables users to buy and sell products effectively while providing a seamless and enjoyable shopping experience.

## CHAPTER 6

### CONCLUSION

In conclusion, developing an e-commerce web application (eshop) using Django offers a comprehensive solution for facilitating online buying and selling activities. By harnessing Django's powerful features and flexibility, developers can create a robust platform that enables seamless transactions between users and merchants. Here are some key points to summarize the process and potential outcomes:

1. **\*\*Efficient Development Process:\*\*** Django's high-level framework streamlines the development process by providing built-in tools for handling authentication, database management, and URL routing. This accelerates the creation of essential features such as user registration, product catalog management, and order processing.
2. **\*\*Secure and Scalable Platform:\*\*** Security is paramount in e-commerce applications, and Django's built-in security features, such as protection against common vulnerabilities like SQL injection and cross-site scripting (XSS), help safeguard sensitive user data and financial transactions. Additionally, Django's scalability allows the platform to accommodate growing user bases and increasing transaction volumes.
3. **\*\*User-Friendly Experience:\*\*** With intuitive interfaces and responsive design, the eshop web application ensures a user-friendly experience for both buyers and sellers. Features such as product search, filtering, and recommendations enhance discoverability, while a streamlined checkout process simplifies the purchasing journey. Sellers benefit from easy-to-use tools for managing product listings, inventory, and order fulfillment.
4. **\*\*Enhanced Business Opportunities:\*\*** By providing a platform for merchants to showcase their products to a wide audience, the eshop web app opens up new opportunities for businesses to reach potential customers and increase sales. Sellers can leverage features like promotional

campaigns, discounts, and analytics to optimize their marketing strategies and drive revenue growth.

**5. \*\*Data-Driven Insights:\*\*** The eshop web app generates valuable insights into user behavior, purchasing patterns, and product performance through advanced analytics and reporting capabilities. By analyzing this data, businesses can make informed decisions about product offerings, pricing strategies, and marketing campaigns, leading to improved competitiveness and market positioning.

**6. \*\*Community and Collaboration:\*\*** The eshop web app fosters a vibrant community of buyers and sellers, facilitating interactions, feedback, and collaboration. Features such as customer reviews, ratings, and seller profiles promote trust and transparency, encouraging repeat purchases and building brand loyalty.

In conclusion, developing an eshop web app using Django empowers businesses to create a dynamic online marketplace that connects buyers and sellers, drives sales, and fosters a thriving e-commerce ecosystem. With its robust features, security measures, and scalability, Django provides a solid foundation for building a successful online platform that delivers value to users and businesses alike.

## REFERENCES

1. <https://visualstudio.microsoft.com/>
2. <https://docs.djangoproject.com/en/5.0/>
3. <https://www.python.org/downloads/>
4. <https://chat.openai.com/c/6aebded3-83c1-4225-bdaa-98a09e6f66b5>
5. <https://www.google.com/>