Django

MVT:

MVT stands for Model-View-Template, which is the architecture used in Django. It's similar to the MVC pattern but with some key differences, particularly in terminology and how the components interact. Let's break down MVT:

1. Model:

The Model component in MVT represents the application's data and business logic, similar to the Model in MVC. It defines the structure of the data, enforces business rules, and interacts with the database.

- In Django, models are Python classes that subclass **django.db.models.Model**. Each model class corresponds to a database table, and its attributes represent the table's fields.
- Models encapsulate data-related logic and provide methods for querying, creating, updating, and deleting records in the database.

2. View:

The View component in MVT corresponds to the Controller in MVC but with some differences in terminology and approach. Views in Django handle the request-response cycle, process user input, and generate responses to be sent back to the client.

- Views in Django are Python functions or classes that receive HTTP requests and return HTTP responses. They interact with the Model to retrieve and manipulate data and render templates to generate dynamic HTML content.
- Unlike traditional controllers in MVC, Django's views are not responsible for deciding which template to render. Instead, they focus on processing requests and generating context data to be used by templates.

3. Template:

The Template component in MVT corresponds to the View in MVC. Templates in Django are responsible for presentation logic and generating the user interface.

- Templates are HTML files that contain placeholders for dynamic data (using Django template language syntax) and static content.
- Templates are rendered by views to generate dynamic HTML content that is sent back to the client as part of the HTTP response.
- Templates allow developers to separate the presentation layer from the application logic, promoting code reusability and maintainability.

ORM:

Django's ORM (Object-Relational Mapping) is a feature that lets you interact with your database using Python objects and methods, without needing to write SQL queries directly. It maps database tables to Python classes (models) and their rows to instances of those classes. With the ORM, you can perform database operations like querying, creating, updating, and deleting records using Python code, and Django handles the translation of these operations into SQL queries behind the scenes. This abstraction simplifies database access and maintenance, making it easier to work with databases in Django projects.

1.Virtual Environment:

A virtual environment is a self-contained directory that contains a Python installation for a particular version of Python, plus a number of additional packages. It allows you to work on a Python project in isolation from your system's Python installation. This is useful because different projects may require different versions of libraries, and using virtual environments helps avoid conflicts between them.

2.Creating a Virtual Environment:

```
# Install virtualenv if you haven't already
pip install virtualenv

# Create a virtual environment
python -m venv env1
```

3. Activating the Virtual Environment:

```
env1\Scripts\activate
```

4.Install Django:

Before creating a Django project, ensure Django is installed within your virtual environment. You can install it via pip:

```
pip install django
```

5. Create Django Project:

Once Django is installed, you can create a new Django project by using the django-admin command:

```
django-admin startproject project_name
```

Replace **project_name** with the desired name of your Django project.

6.Folder Structure:

After running the command, you'll have a folder named **project_name** containing your Django project. This folder structure is the same as you would get outside of a virtual environment.

- **project_name/**

- **manage.py**: Django's command-line utility.
- **project_name/**: Your Django project package.
  - **__init__.py**
  - **settings.py**: Settings/configuration for this Django project.
  - **urls.py**: URL declarations for the project.
  - **wsgi.py**: WSGI config for deploying your project.

7.Starting the Development Server:

Navigate into the project directory:

```
cd project_name
Then, you can start the development server to see your Django project in action:
python manage.py runserver
```

This command will start the development server, and you can access your Django project by opening a web browser and navigating to the address provided in the output, usually **http://127.0.0.1:8000/**.

Creating an APP

1.Navigate to Your Django Project Directory:

First, make sure you're in the root directory of your Django project. If you're not already there, use the cd command to navigate to it.

```
cd /path/to/your/project_name
```

2.Create the App:

Use the **manage.py** script to create a new Django app within your project. Replace **app_name** with the desired name of your app.

```
python manage.py startapp app_name
```

This command will create a new directory named app_name within your Django project directory.

3.Folder Structure of the App:

After running the above command, you'll have a folder named **app_name** containing your new Django app. Here's the basic structure:

- **app_name/**
  - **__init__.py**: This file indicates to Python that this directory should be treated as a Python package.
  - **admin.py**: Register models to appear in the Django admin.
  - **apps.py**: Application configuration.

- **models.py**: Define database models for your app.
- **views.py**: Define views (functions) that handle HTTP requests and return responses.
- **urls.py**: URL configurations specific to this app.
- **tests.py**: Write tests for your app here.
- **migrations/**: Directory containing database migration files generated by Django.

Rendering an html file in the browser

1.Create a Template:

- **Location:** Inside the main project directory.
- **Folder:** Create a folder named templates inside your main project directory.
- **File:** Within the templates folder, create an HTML file for your template.

For example, in your main project directory, create a directory structure like this:

```
project_name/
├── templates/
│   └── index.html
```

2.Define a View:

- **Location**: Inside the app's directory.
- **File:** Open views.py within your app directory.
- **Define a View Function:** Create a function that will render your template.

For example, in myapp/views.py:

```
from django.shortcuts import render

def index(request):
    return render(request, 'index.html')
```

3. Map URL to View:

- **Location:** Inside the app's directory.
- **File**: Create a urls.py file if it doesn't exist.
- **Define** URL Patterns: Map URLs to the views you created.

For example, in myapp/urls.py:

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

urlpatterns = [
    path('', views.index, name='index'),
]
```

4. Include App URLs in Project URLs:

- **Location**: Inside the main project directory.
- **File:** Open urls.py within your main project directory.
- **Include App URLs:** Include your app's URLs in the project's URL configuration.

For example, in project_name/urls.py:

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('myapp.urls')),
]
```

5.Settings Configuration:

- **Location:** Inside the main project directory.
- **File:** Open settings.py within your main project directory.
- **Configure Templates:** Make sure Django knows where to find your templates.

In **project_name/settings.py**, ensure you have:

```
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]
```

6.Start Development Server:

- **Run Server:** Make sure your development server is running.
- **Command:** Navigate to your project directory in the terminal and run python manage.py runserver.

7. **View in Browser:**

Open Browser: Open your web browser and navigate to the URL where your app is served (usually **http://127.0.0.1:8000/**). You should see your template rendered in the browser.

Inheriting Templates

Inheriting templates in Django allows you to create a base template with common elements (like headers, footers, navigation bars) and then extend or override specific sections of that base template in child templates. This promotes code reuse and makes it easier to maintain consistent design across your web application.

1. **Create a Base Template:**
   First, create a base template that contains the common elements you want to reuse across multiple pages of your website.
   Location: Inside the **templates** directory of your main project directory.
   For example, create a file named **base.html** inside the **templates** directory:

```
<!-- project_name/templates/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>{% block title %}My Website{% endblock %}</title>
</head>
<body>
  <header>
    <h1>My Website</h1>
    <!-- Common header content -->
  </header>

  <nav>
    <!-- Navigation bar content -->
  </nav>

  <main>
    {% block content %}
    {% endblock %}
  </main>

  <footer>
    <!-- Footer content -->
  </footer>
</body>
</html>
```

2.Create Child Templates:

Next, create child templates that inherit from the base template and override or extend specific blocks.

Location: Inside the templates directory of your main project directory or app directories.

For example, create a file named child.html inside an app's templates directory:

```
<!-- project_name/myapp/templates/child.html -->
{% extends 'base.html' %}

{% block title %}
   My Child Page - {{ block.super }}
{% endblock %}

{% block content %}
   <h2>Child Page Content</h2>
   <!-- Child page specific content -->
{% endblock %}
```

3. Inheriting and Overriding Blocks:

- The **{% extends 'base.html' %}** tag at the beginning of the child template tells Django to inherit from the **base.html** template.
- The **{% block %}** and **{% endblock %}** tags define blocks that can be overridden in child templates.
- In the child template, use **{% block block_name %}** to override the content of a block defined in the base template. Use **{{ block.super }}** to include the content of the parent block.

4. Use Child Templates:

In your views, render the child templates as usual. Django will automatically render the base template first, then insert the content from the child template into the appropriate block.

Static Folder

The static folder in Django is where you store static files such as CSS, JavaScript, images, or any other assets that are not dynamically generated by Django itself. Here's how you can set up and use the static folder in your Django project:

1. Create a Static Folder:

First, create a folder named **static** in your main project directory. This folder will contain all your static files.

Location: Inside the main project directory.

For example, create a directory structure like this:

```
project_name/
 ├── static/
```

2. Organize Your Static Files:

Within the **static** folder, you can organize your static files into subdirectories based on their type or purpose. For example:

```
static/
├── css/
│   └── style.css
├── js/
│   └── script.js
└── img/
    └── logo.png
```

3. Configure Static Files in Settings:

In your settings.py file, make sure you have configured the STATIC_URL and STATICFILES_DIRS settings to tell Django where to find your static files.

```
STATIC_URL = 'static/'
STATICFILES_DIRS=[
   os.path.join(BASE_DIR,'static')
]
STATIC_ROOT=os.path.join(BASE_DIR,'assets')
```

4. Collecting Static Files for Deployment:

For deployment, you'll need to run the collectstatic management command to collect all your static files into a single directory for serving by your web server.

```
python manage.py collectstatic
```

This will copy all your static files from various locations (including apps' **static** directories) into the directory specified by the **STATIC_ROOT** setting.

5. Link Static Files in HTML Templates:

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>My Website</title>
   <link rel="stylesheet" href="{% static 'css/style.css' %}">
</head>
<body>
   <h1>Welcome to My Website</h1>
   <img src="{% static 'assets/img/logo.png' %}" alt="Logo">
   <!-- Your HTML content here -->
</body>
</html>
```

- The **{% load static %}** line loads the static tag library, allowing us to use the {% static %} template tag.
- The CSS file style.css is loaded using the **{% static %}** tag. This assumes the CSS file is located in the static/css directory.
- The image logo.png is loaded using the {% static %} tag. This assumes the image is located in the static/assets/img directory.
- You can replace "Welcome to My Website" with your desired content.

Addition of Two Number Using Template Language

1. Create a Django Project:

Run the following command to create a new Django project named sum_project:

```
->django-admin startproject sum_project
```

2. Create a Django App:

Navigate into the project directory:

```
-> cd sum_project
```

Create a new Django app named `sum_app`:

```
-> python manage.py startapp sum_app
```

3. Create HTML Templates:

Create two HTML templates:

- **sum_form.html** to display the form.
- **result.html** to display the sum.

Here's an example of **sum_form.html**:

```
<!-- sum_app/templates/sum_form.html -->
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Sum Form</title>
</head>
<body>
   <h2>Enter Two Numbers</h2>
```

```
    <form method="post" action="{% url 'sum_form' %}">
        {% csrf_token %}
        <label for="num1">Number 1:</label>
        <input type="text" id="num1" name="num1"><br><br>
        <label for="num2">Number 2:</label>
        <input type="text" id="num2" name="num2"><br><br>
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

And here's an example of `result.html`:

```
<!-- sum_app/templates/result.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Result</title>
</head>
<body>
    <h2>Sum Result</h2>
    <p>The sum of the two numbers is: {{ result }}</p>
</body>
</html>
```

4. Create a View:

Open the views.py file inside your sum_app directory and define a view function to handle the form submission and display the sum:

```
# sum_app/views.py
from django.shortcuts import render

def sum_view(request):
    if request.method == 'POST':
        num1 = int(request.POST.get('num1'))
        num2 = int(request.POST.get('num2'))
        result = num1 + num2
        return render(request, 'result.html', {'result': result})
    return render(request, 'sum_form.html')
```

5. Configure URLs:

Configure URL patterns to map the view function to a URL. Open the urls.py file inside your sum_app directory and define a URL pattern:

```
# sum_app/urls.py
from django.urls import path
from . import views

urlpatterns = [
   path('', views.sum_view, name='sum_form'),
]
```

Then, include this app's URLs in the main project's URLs. Open the `urls.py` file inside your `sum_project` directory and include the app's URLs:

```
# sum_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
   path('admin/', admin.site.urls),
   path('', include('sum_app.urls')),
]
```

With these steps, you've created a Django project that takes two numbers from the user using an HTML form and displays the sum of the two numbers.

Migrations

1. Create Models:

Models in Django are Python classes that represent database tables. You define your models in the models.py file of your app.

For example, let's create a simple model representing a book:

```
# sum_app/models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    publication_year = models.IntegerField()
    def __str__(self):
        return self.title
```

2. Generate Migrations:

After defining your models, you need to create migrations. Django includes a command-line tool for managing migrations.

Run the following command to generate a migration for your app:

=> **python manage.py makemigrations sum_app**

This command examines the current state of your models and creates a migration file that describes the changes you've made.

3. Apply Migrations:

To apply the migrations and update your database schema, run:

=> **python manage.py migrate**

This command executes all pending migrations and synchronizes the database schema with the current state of your models.

4. Advantages of Migrations:

- **Database Agnosticism:** Migrations allow you to work with different database backends without having to manually manage the database schema for each one.
- **Version Control:** Migrations are tracked in version control systems (like Git), allowing you to revert changes if needed and enabling collaboration among developers.
- **Consistency:** Migrations help maintain consistency across different environments (development, staging, production) by ensuring that the database schema is the same everywhere.
- **Ease of Deployment:** Migrations make it easy to deploy changes to your database schema in a consistent and reproducible manner, which is crucial in a production environment.

5. Modify Models:

If you need to make changes to your models after they've been migrated, simply update your **models.py** file, generate new migrations using **makemigrations**, and apply them with **migrate**.

MYSQL CONFIGURATION

1. Database Configuration:

Django uses SQLite as the default database, but you'll want to switch to MySQL for a production environment. Update the DATABASES setting in your settings.py file to specify MySQL as the database backend and provide connection details:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'your_database_name',
        'USER': 'your_database_user',
        'PASSWORD': 'your_database_password',
        'HOST': 'localhost',  # Or the IP address of your MySQL server
        'PORT': '3306',       # MySQL default port
    }
}
```

Replace **'your_database_name'**, **'your_database_user'**, and **'your_database_password'** with your MySQL database name, username, and password respectively.

2. Install MySQL Connector:

You also need to install the MySQL database connector for Python. You can do this using pip:

=>**pip install mysqlclient**