**Before You Install Django...Watch This!**

1. We are using functional based views here. (ToDo: Learn the difference between Class-Based views & Functional Based Views)
2. We will be creating an “Social Club App”. In this web-app, we will be able to add events, venues, calendars, food-menu’s, event agenda etc.  
   Users will be able to sign-up, login, add events, venues…

Commands:

1. pwd – to see the present working directory.
2. mkdir – to make a particular folder.
3. cd – to change working directory from one folder address to another.
4. python -m venv virt – for setting up the virtual environment.
5. source virt/Scripts/activate – for activating the virtual environment.
6. deactivate - for deactivating the virtual environment and returning back to the normal git bash environment.
7. pip freeze – to see what all python-related things have been installed in the virtual environment.
8. pip install django==3.1.5 – command to install django 3.1.5 version.
9. django-admin.py startproject myclub-website - to setup a project named myclub\_website.
10. rm -rf myclub-virt – to remove the existing virtual environment.
11. python manage.py runserver – cmd to run development server.
12. python manage.py startapp app\_name – cmd to create a new app inside an existing project.
13. Cd myclub\_website – cmd to get into the project myclub\_website.

**𝘈𝘯𝘴𝘸𝘦𝘳𝘴 𝘱𝘰𝘸𝘦𝘳𝘦𝘥 𝘣𝘺 𝘊𝘰𝘱𝘪𝘭𝘰𝘵**

1Q. A black screen with green and purple text

AI-generated content may be incorrect.

I have created a Django project using command "django-admin.py startproject myclub\_website", when I list the items in that project folder, I see 2 items there, once is manage.py file, what is the other "myclub\_website" I see there?

A. Great question! When you run django-admin.py startproject myclub\_website, Django sets up a default project structure with two components:

1. **manage.py**
   * This script is used for managing your Django project.
   * You use it to run the development server, apply migrations, and execute various commands like:
   * python manage.py runserver
2. **myclub\_website/ (Project Package)**
   * This is a **directory** that contains the actual Django project settings and configurations.
   * Inside this folder, you'll find files like:
     + **\_\_init\_\_.py** (marks it as a Python package)
     + **settings.py** (project configuration)
     + **urls.py** (URL routing)
     + **wsgi.py** (for deploying on a WSGI server)

This structure helps keep your Django project organized. Since you're exploring Django and GitHub workflows, setting up your project efficiently is key!

2Q. balusani.virinchi@m2b-l-4315ldx MINGW64 /c/myclub/myclub\_website $ python manage.py runserver Watching for file changes with StatReloader Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions. Run 'python manage.py migrate' to apply them. June 14, 2025 - 06:19:27 Django version 3.1.5, using settings 'myclub\_website.settings' Starting development server at [http://127.0.0.1:8000/](https://127.0.0.1:8000/) Quit the server with CTRL-BREAK.

What is it saying?

A. This output shows that your **Django development server is running successfully**, but there are **18 unapplied migrations** that need to be executed for certain built-in Django apps.

**Breakdown of the Output:**

1. **System Check Passed** ✅
   * Django scanned your project and found no issues.
2. **Pending Migrations** ⚠️
   * Some built-in Django apps (admin, auth, contenttypes, sessions) require database migrations.
   * You need to run:
   * python manage.py migrate
   * This applies database changes for those apps.
3. **Development Server Started** 🚀
   * Your Django project is running at:
   * http://127.0.0.1:8000/
   * You can access it by opening this link in your browser.
4. **Exit the Server**
   * To stop the server, press **CTRL + BREAK**.

3Q. python mange.py migrate - migrates the databases of few default Django apps. then what does python manage.py makemigrations do?

A. You're absolutely right—python manage.py migrate applies the pending database migrations. But before migrations can be applied, Django needs to **create** them, and that's where python manage.py makemigrations comes in.

**Difference Between makemigrations and migrate**

1. **python manage.py makemigrations** 🏗️
   * **Creates migration files** based on changes you've made to models.
   * Example: If you add a new field to a model, running makemigrations will generate a migration file for that change.
   * It **does NOT modify** the database yet—just prepares the migration files.
2. **python manage.py migrate** 🚀
   * **Applies those migration files** to the database, making actual changes (creating tables, updating fields, etc.).
   * Without running migrate, the changes won’t reflect in the database.

**Example Workflow**

python manage.py makemigrations # Generates migration files

python manage.py migrate # Applies changes to the database

Since you're working on Django, understanding how migrations interact with models is key!

4Q. Detects Changes in Models

* If you've modified a model (like adding a new field or changing a data type), Django identifies these changes

What do you mean, when you say "model"?

A. In Django, a **model** is a blueprint for storing data in a database. It's a Python class that defines **what data should be stored** and how it relates to other data.

**Key Features of a Django Model**

* **Represents a database table**: Each model corresponds to a table in the database.
* **Defines fields**: Specifies what kind of data (text, numbers, dates, etc.) will be stored.
* **Handles relationships**: Supports connections between tables (e.g., one-to-many relationships).

**Example of a Model in Django**

Suppose you're building a **Club Management** system. You might define a model like this:

from django.db import models

class Member(models.Model):

name = models.CharField(max\_length=100) # Stores member names

email = models.EmailField(unique=True) # Stores unique email addresses

joined\_date = models.DateField() # Stores when the member joined

* This creates a **Member** table with three fields: name, email, and joined\_date.
* When you run python manage.py makemigrations, Django detects any changes in this model and prepares migration files.
* Running python manage.py migrate applies those changes to the database.

Since you're diving into Django, understanding how models work is **crucial** for handling data effectively.

5Q. so, don't we have this in flask?

A. Good question! **Flask**, unlike Django, doesn't come with built-in support for models and migrations. Instead, Flask gives you **full flexibility** to choose your database management approach.

**Flask vs. Django (Models & Migrations)**

* **Django**: Uses an ORM (**Object-Relational Mapper**) called **Django ORM**, which automatically handles database migrations with makemigrations and migrate.
* **Flask**: Doesn't have a built-in ORM, but developers often use **SQLAlchemy**, which provides similar functionality.

**How Do We Handle Migrations in Flask?**

If you're working with Flask and databases, you typically:

1. **Use SQLAlchemy** for models:

from flask\_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(100))

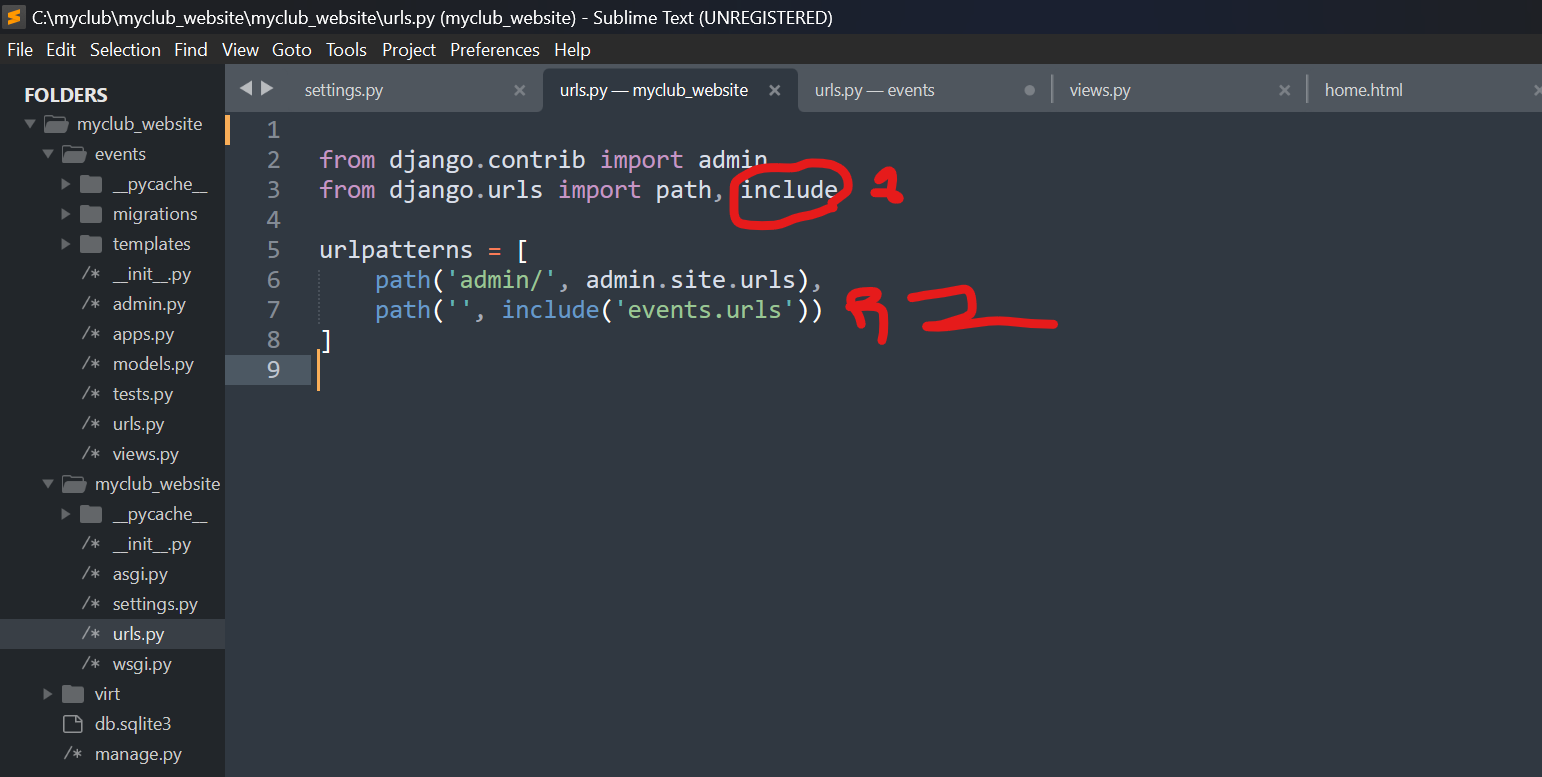
1. **Use Flask-Migrate** to handle migrations:

flask db init # Initialize migrations

flask db migrate -m "Added users table"

flask db upgrade # Apply migrations

Since you're diving into Django, understanding different approaches could help broaden your development skills!

6Q. 

the include which we have imported as you can see in the above image, is it a function?

A. Yes, the include you imported in the image is indeed a **function**. It comes from django.urls and is used to include other URL configurations within your project. This helps keep your Django URLs modular and organized.

In your urls.py file, the include('events.urls') statement ensures that Django will look inside the events app for its own urls.py file, allowing you to define routes specific to that app separately. This is especially useful for large projects where different apps manage their own URLs.

3. We are creating a project named myclub\_website using cmd ‘django-admin startproject project\_name’.

4. Inside the project, we are creating an app for events. So, this new app is one of the functionality which will be added to the project later.

A Django project can consist of one or more sub applications within a project that can consist of different functionalities within a project.

5. New app named ‘events’ is created using ‘python manage.py startapp appName’.

A screenshot of a computer

AI-generated content may be incorrect.

6. Whenever an app is created, in this case it is events, we add that app name to the ‘INSTALLED\_APPS’ list variable present in the settings.py file of myclub\_website project package.

A screenshot of a computer

AI-generated content may be incorrect.

7. After the work in settings.py file, we import include function from django.urls. Also in urlpatterns lists we point the main home url to urls.py file in event(i.e. events.urls)

8. A screen shot of a computer

AI-generated content may be incorrect.

So, the execution comes to urls.py file of events, in here, the main home url is pointing to the home function present in views.py file of events app.

9.

A screenshot of a computer

AI-generated content may be incorrect.

Once the execution comes to views.py file, we see that home.html file is being rendered. Then the execution goes to home.html file.

10. A screenshot of a computer

AI-generated content may be incorrect.

This html code is rendered once execution comes here.

