**BASICS**

**Set up project**

django-admin startproject my\_site

.vscode (folder) ------- settings.json (file) ---- {

    "python.languageServer": "Pylance",

    "emmet.includeLanguages": {

        "django-html": "html",

    }

}

**Create environment**

py -m venv my\_env ------- my\_env/Scripts/activate -----pip install Django

**Create app**

Python manage.py startapp blog

**Setting URL**

Create urls.py (file locally in blog app)------- urlpatterns = [

    path("", views.starting\_page, name="starting-page"),

    path("posts", views.posts, name="post-page"),

    path("posts/<slug:slug>", views.post\_detail, name="post-detail-page") #/posts/my-first-post

]

These functions will be defined in views.py

Connect urls.py with global urls.py ----- urlpatterns = [

    path("admin/", admin.site.urls),

    path("", include("blog.urls")) # https://localhost:8000/

]

**Adding First template**

Globally: templates(folder) ---- base.html -----

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>{% block title %}{% endblock %}</title>

    {% block css\_files %}{% endblock %}

</head>

<body>

    {% block content %}

    {% endblock %}

</body>

</html>

Locally: templates (folder) --- blog(folder) ---- index.html (file) --- {% extends "base.html" %} (to inherit from base.html)

To extend/inherit base.html go to settings.py ---- TEMPLATES ----

    "DIRS": [

            BASE\_DIR / "templates"

To make sure that app specific folders are picked up we need to make Django aware of this blog app.

Go to settings.py ----- INSTALLED\_APPS ----- INSTALLED\_APPS = [……….,'blog']

In views.py Render html.index :

def starting\_page(request):

    return render(request, "blog/index.html")

\*\*End of adding first template\*\*

**Creating django Model with fields**

from django.db import models

# Create your models here.

class Book(models.Model):

    title = models.CharField(max\_length=50)

    rating = models.IntegerField()

**Migrations**

Make dbsqlite3 file

python manage.py makemigrations (done every time we change the class in models.py to add tables or columns)

python manage.py migrate

**Inserting Data**

python manage.py shell

then import the class we created in models.py on shell -----🡪 from book\_outlet.models import Book

Add some data

>>> harry\_potter = Book(title="Harry Potter 1 - The Philosopher's Stone", rating=5)

>>> harry\_potter.save()

>>> lord\_of\_the\_rings = Book(title="Lord of the Rings", rating=4)

>>> lord\_of\_the\_rings.save()

Then get the data to see if it worked

>>> Book.objects.all()

<QuerySet [<Book: Book object (1)>, <Book: Book object (2)>]>

To use “OR” keyword in filtering:

from django.db.models import Q

FOR SLUG

In models class:

Make this attribute

slug = models.SlugField(default="", null=False)

make this method

from django.utils.text import slugify

def save(self, \*args, \*\*kwargs):

        self.slug = slugify(self.title)

        super().save(\*args, \*\*kwargs)

we can just write /admin in the url to access django admin. Then create our superuser by

python manage.py createsuperuser

This is an inbuilt frontend in Django to avoid the use of shell to add and remove data.

We can edit how this look by creating a class in admin.py like following:

class BookAdmin(admin.ModelAdmin):

    prepopulated\_fields = {"slug": ("title",)}

    list\_filter = ("author", "rating",)

    list\_display = ("title", "author",)

Relationships

To connect two models (classes in models.py) like author and book class, we need to add foreign key field. For that we need to replace the author value in the book class with

author = models.ForeignKey()

So instead of directly inserting the author data in the author, we POINT at another database at another table. Django will set up that pointer (ID) to be stored in the database and manage that connection for use behind the scenes just like sql queries. We just need Django to know which model should be the model to which we point here. So we need to mention author in the brackets.

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

on\_delete tells Django what to do in case an entry is deleted from the table. on\_delete=models.CASCADE tells that if an author is deleted from the table then all the books associated with that author will also be deleted i.e. it will affect all related models. On the other hand, on\_delete=models.PROTECT does not affect the related models.

def \_\_str\_\_(self):

        return f"{self.street}, {self.postal\_code}, {self.city}"

This is added in model classes to make admin page look better

Nested “Meta class” can be added in model classes. For meta configuration of the model because any property that we add to the model itself is treated as fields of the model which has implication on what is stored in database and what is rendered in the form in admin template etc. But if you want to configure behavior about your model, you cant do that with properties added in the model. And for that we make a nested class called Meta. Not necessary but can add to render special settings

class Meta:

        verbose\_name\_plural = "Address Entries"

**Circular Relations & Lazy Relations**

Sometimes, you might have two models that depend on each other - i.e. you end up with a **circular relationship**.

Or you have a model that has a relation with itself.

Or you have a model that should have a relation with some built-in model (i.e. built into Django) or a model defined in another application.

Below, you find examples for all three cases that include Django's solution for these kinds of "problems": **Lazy relationships**. You can also check out the [**official docs**](https://docs.djangoproject.com/en/3.2/ref/models/fields/#module-django.db.models.fields.related) in addition.

1) Two models that have a **circular relationship**

1. class Product(models.Model):
2. # ... other fields ...
3. last\_buyer = models.ForeignKey('User')
5. class User(models.Model):
6. # ... other fields ...
7. created\_products = models.ManyToManyField('Product')

In this example, we have multiple relationships between the same two models. Hence we might need to define them in both models. By using the model name as a string instead of a direct reference, Django is able to resolve such dependencies.

2) Relation with the **same model**

1. class User(models.Model):
2. # ... other fields ...
3. friends = models.ManyToManyField('self')

The special self keyword (used as a string value) tells Django that it should form a relationship with (other) instances of the same model.

3) Relationships with **other apps** and their models (built-in or custom apps)

1. class Review(models.Model):
2. # ... other fields ...
3. product = models.ForeignKey('store.Product') # '<appname>.<modelname>'

You can reference models defined in other Django apps (no matter if created by you, via python manage.py startapp <appname> or if it's a built-in or third-party app) by using the app name and then the name of the model inside the app.