**Django Blog Documentation:**

1. **Setting up the project**

This is the command to setup the project ‘ django-admin startproject My\_Blog’ on the terminal of visual studio code or the command prompt at the appropriate location

To change directory in the terminal command is cd tab

After this enter the command python manage.py startapp App\_login to create sub-applications inside the main folder

Created 2 applications: using the command

python manage.py startapp App\_login

python manage.py startapp App\_Blog

App\_login: will handle all the login and autentication related tasks of our blog

App\_Blog: Will handle all our Blog related tasks.

Templates and static folder: will store all the static files, jss and css files

App\_login, App\_Blog needs to be installed in the main My\_Blog project ( that has the manage.py file)

**In the settings.py file of the My\_Blog project folder need to make these changes:**

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

TEMPLATES\_DIR = BASE\_DIR.joinpath('templates')

STATIC\_DIR = BASE\_DIR.joinpath('static')

MEDIA\_DIR = BASE\_DIR.joinpath('media')

In the Templates section add this :

"DIRS": [TEMPLATES\_DIR,],

In the Static file section this needs to be added:

STATICFILES\_DIR = [STATIC\_DIR]

#media

MEDIA\_ROOT = MEDIA\_DIR

MEDIA\_URL = '/media/'

In the Installed app section need to register the app we created:

INSTALLED\_APPS = [

    "django.contrib.admin",

    "django.contrib.auth",

    "django.contrib.contenttypes",

    "django.contrib.sessions",

    "django.contrib.messages",

    "django.contrib.staticfiles",

    "App\_login",

    "App\_Blog",

]

1. **Creating Models:**

Inside the App\_Login folder we have the model.py file

We can use Django’s default authentication system here:

from django.contrib.auth.models import User

The User class has the following fields that we can use-

Username, email, firstname, lastname etc:

<https://docs.djangoproject.com/en/5.0/ref/contrib/auth/>

In the App\_Login section there is a models.py file where I created the following model class-

class UserProfile(models.Model):

    user = models.OneToOneField(User, related\_name='user\_profile', on\_delete=models.CASCADE)

    profile\_pic = models.ImageField(upload\_to='profile\_pics')

After doing this run the following command on the vs terminal

**python manage.py migrate**

This will create a db.sqlite file

After that this command

**python manage.py makemigrations App\_login**

**then this command python manage.py migrate**

Creates a table that has the class name and column names with the variables defined inside the class in the db.sqlite file

Created a class called blog in the models.py file of App\_Blog

**Understanding the database design**

A user should be able to add their comment on a blog,

Therefore class and variables were created accordingly, ( object relation mapping)

**After this we create the super user-**

Gave this on the commandline : python manage.py createsuperuser

Username shreya

Email [scsilvershark@gmail.com](mailto:scsilvershark@gmail.com)

Password shreya

Now we need to register these models on the admin page

On the admin.py file for each folder

from django.contrib import admin

from . models import Blog,Comment,Likes

# Register your models here.

admin.site.register(Blog)

admin.site.register(Comment)

admin.site.register(Likes)

Register the models classes like this

**One Example(detailed explanation):**

class Blog(models.Model):

    author = models.ForeignKey(User, on\_delete=models.CASCADE,related\_name='post\_author')

    blog\_title = models.CharField(max\_length=264,verbose\_name="Give a title")

    # slug feature : the URL will take the blog title name

    slug = models.SlugField(max\_length=264,unique=True)

    blog\_content = models.TextField(verbose\_name="What is on your mind")

    blog\_image = models.ImageField(upload\_to='blog\_images',verbose\_name="Blog\_Image")

    publishDate = models.DateField(auto\_now\_add=True)

    updateDate = models.DateField(auto\_now=True)

    def \_\_str\_\_(self):

        return self.blog\_title

User is a class in Django.auth.contrib with the following fields

firstName

lastname

Username

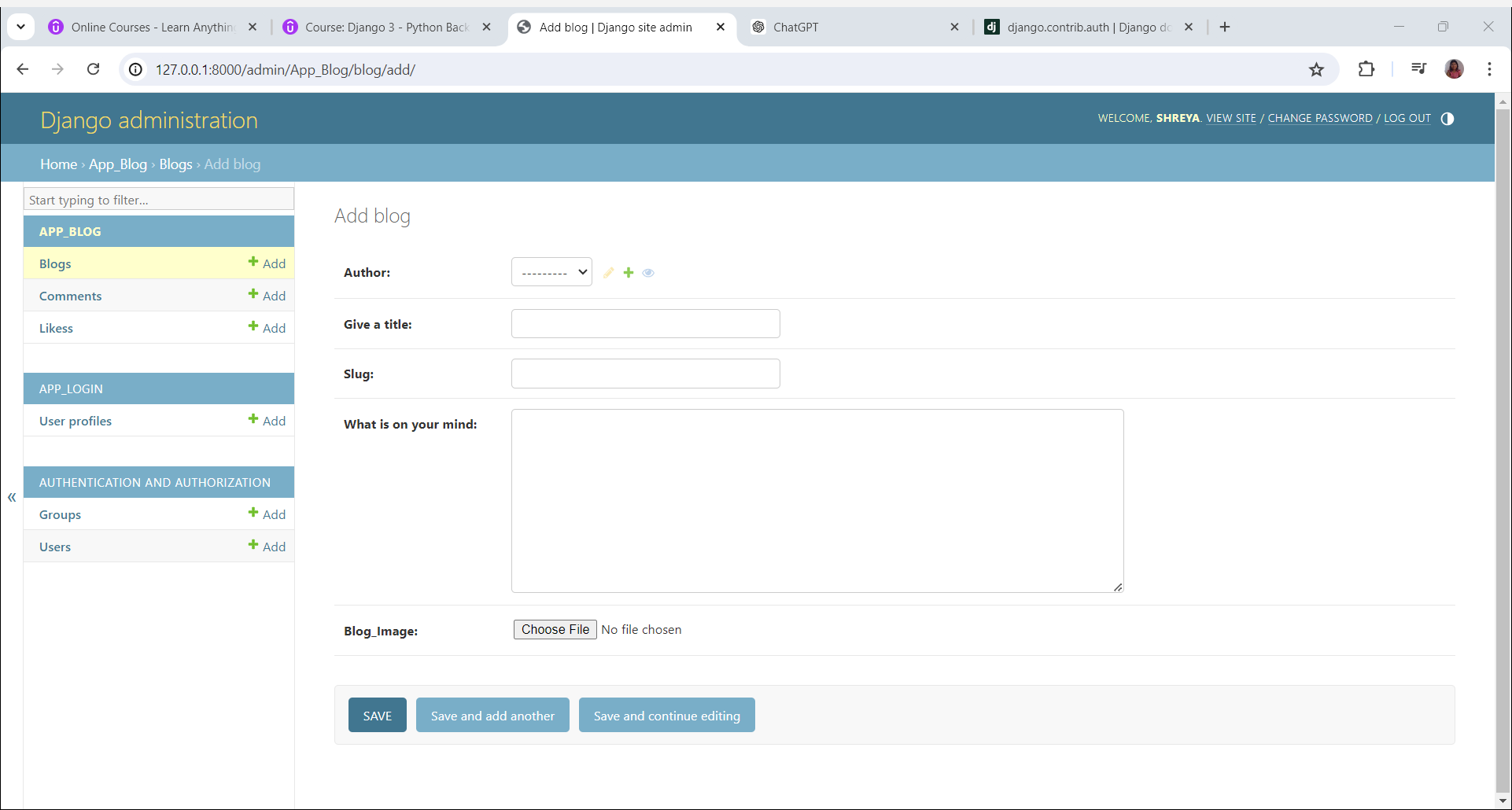
Email

Password

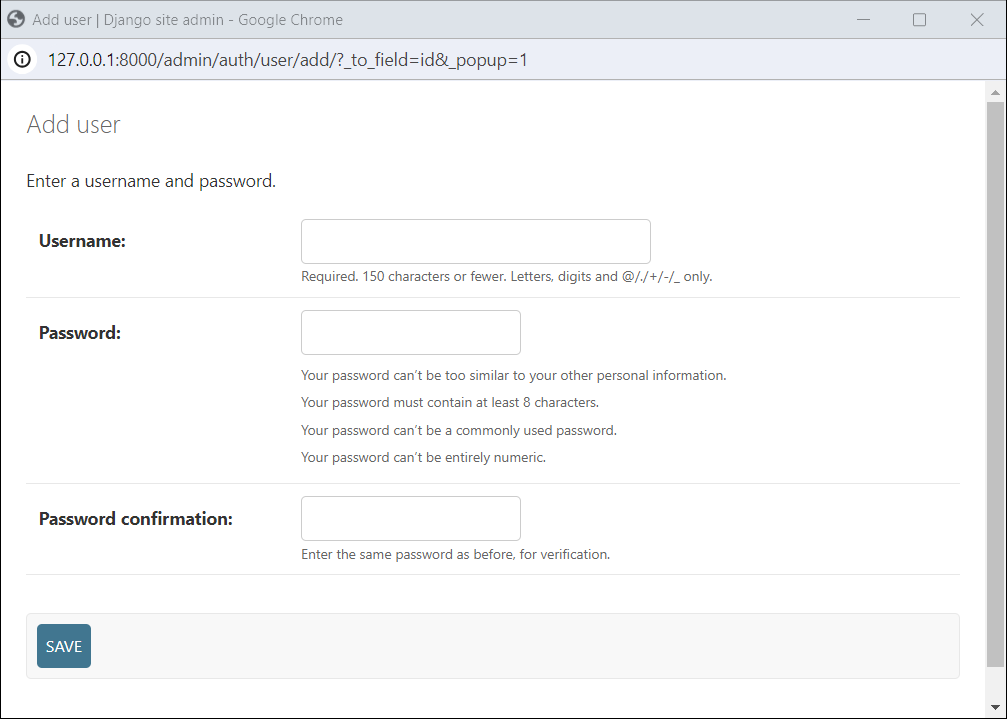
We added to this and created a few custom fields in the Blog Class

(models.Model): This indicates that the Blog class inherits from models.Model, which is a base class provided by Django. By inheriting from models.Model, the Blog class becomes a Django model.

This is the output:



When you click on the add button next to the author field this is what we get:



1. **URL, VIEW and Templates for the Blog**

To view our created tables we need to configure the url, views

Go to the url.py file in the main blog folder:

from django.contrib import admin

from django.urls import path,include

from . import views

urlpatterns = [

    path("admin/", admin.site.urls),

    path("account/",include('App\_login.urls')),

    path("blog/",include('App\_Blog.urls')),

    path("",views.index,name='index')

]

Add this in the url patterns

Create a view.py file and create a function called index

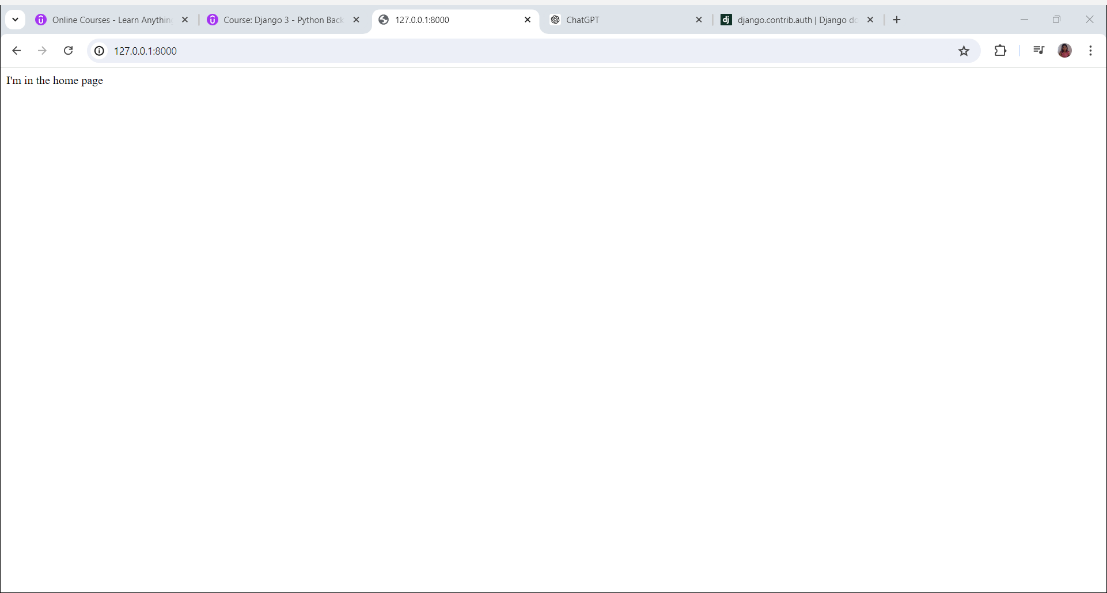
In the views.py file

from django.http import HttpResponse

def index(request):

    return HttpResponse("I'm in the home page")

When we run the server this is our output



Create urls.py for each of the apps created

Explanation:

The Urls.py file in the main blog folder will be redirected to the App\_login and App\_Blog depending on that we enter

Url will be redirected to the view.py file that contains the contents to be displayed to the viewer

Create 2 folders inside the template folder called App\_Login and App\_Blog and a file called base.html

The base.html file will be extended to all the html files in the other folders

{% block body\_block %}

      {% endblock %}

This syntax will give the help us design the other html files after we extend base.html

1. **User Signup**

We have used the UserCreationForm for creating the signup functionality

In the Views.py File of App.login we wrote this function

from django.contrib.auth.forms import UserCreationForm

def sign\_up(request):

    form = UserCreationForm()

    registered = False

    if request.method == 'POST':

        form = UserCreationForm(data=request.POST)

        if form.is\_valid():

            form.save()

            registered=True

    dict = {'form':form,'registered':registered}

    return render(request,'App\_Login/signup.html',context=dict)

Using Crispy Forms help to make changes to the default signup page

**How is this working ? AKA Flow of control:**

1. **User Login**

url.py path:

views

login.html page

Here is a typical flow of what happens when the form is submitted:

1. **Form Submission**: When the user clicks the "Login" button, the form data is sent to the server via a POST request.
2. **URL Handling**: If the action attribute is not specified, the data is sent to the current URL. If the action attribute is specified, the data is sent to the specified URL.
3. **View Processing**: The Django view associated with the URL processes the form data. This view is typically responsible for validating the form data, handling authentication, and determining the next steps.
4. **Redirection or Response**: After processing the form data, the view can redirect the user to another page, render a new template, or return an appropriate HTTP response.

In urls.py, we define a url pattern:

For example-

path('signin',views.login\_page,name="signin")

 **'signin'**:

* This is the URL pattern that users will navigate to in their web browser to access this view. For example, if your website is www.example.com, users would go to www.example.com/signin to access this page.
* This part of the URL pattern can include variables and dynamic segments if needed, but in this case, it is a simple static string.

 **views.login\_page**:

* This specifies the view function that will handle requests to this URL pattern. In this case, login\_page is a function defined in the views module.
* When a user navigates to www.example.com/signin, Django will call the login\_page view function to process the request and return the appropriate response (e.g., rendering a template, processing form data).

 **name="signin"**:

* This is an optional parameter that provides a name for this URL pattern. Naming URL patterns is useful for a few reasons:
  + It allows you to refer to this URL pattern unambiguously from anywhere in your Django project, such as in templates, views, and redirects, without hardcoding the URL.
  + It makes your code more maintainable. If you ever change the URL pattern (e.g., from 'signin' to 'login'), you only need to update the path function, not every place in your code that references this URL.
* You can use the named URL in templates and views. For example, in a template, you can use {% url 'signin' %} to generate the URL for the signin page.

1. **User Logout**

Same process as the login functionality

1. **User Profile**

To create the profile page we have used these two libraries UserCreationForm and UserChangeForm

1. **Navigation Bar**
2. **Write Blog**

For the App Blog views we have used the class based view and for the login part we used the function based views

1. **Blog Homepage**
2. **Blog Details**

The Div class in web development:

In HTML and web development, the div element itself doesn't "do" anything in the sense of performing actions or behaviors like scripting or event handling. Instead, it serves as a structural container or a "division" of the content within a web page. Here’s what it does:

1. **Structural Container:** The div element is used to logically divide sections of content within an HTML document. It's a block-level element, meaning it typically creates a visible block on a web page that can contain other HTML elements like text, images, forms, etc.
2. **Grouping and Styling:** Developers use div elements to group together other HTML elements and apply CSS styles to them collectively. For example, multiple div elements can be used to structure a page layout (header, footer, sidebar, etc.) or to group related content within a section.
3. **Scripting and Accessibility:** While div itself doesn't directly handle scripting or accessibility features, it provides a way to organize content that can then be targeted and manipulated via JavaScript or styled using CSS. This helps in creating interactive and visually appealing web pages.

Let us analyse this block of code

<div class="row">

  <div class="col-sm-6">

    <h2>{{ blog.blog\_title }}</h2>

    <h4>Posted By : {{ blog.author }}</h4>

    <i><h6>Published On:  {{ blog.publish\_date }}</h6></i>

    <i><h6>Updated On:  {{ blog.update\_date }}</h6></i>

    <h6>{{ blog.liked\_blog.count }} Likes | {{ blog.blog\_comment.count }} Comments</h6>

  </div>

The class="col-sm-6" attribute on the <div> element is a reference to a CSS class. Let's break down what it typically does in the context of front-end web development:

1. **Bootstrap Grid System:** The col-sm-6 class is likely part of a grid system, specifically from Bootstrap, a popular front-end framework. In Bootstrap's grid system, it defines the width and behavior of the <div> element based on the screen size (in this case, "sm" stands for small screens).
2. **Column Width:** col-sm-6 suggests that this <div> should take up half of the available width on small screens (sm). Bootstrap's grid system is based on a 12-column layout, so col-sm-6 occupies 6 columns out of 12, which equals 50% width.
3. **Responsive Design:** Bootstrap's grid classes (col-\*-\*) are designed to create responsive layouts that adapt to different screen sizes (extra small, small, medium, large, etc.). The col-sm-6 class ensures that the content within this <div> is displayed in a two-column layout on small screens and larger, depending on how it's nested and what other grid classes are applied.
4. **Styling and Layout:** Along with defining the width, the col-sm-6 class may include additional CSS properties or styles (defined in Bootstrap's CSS) to manage spacing, alignment, and other layout-related aspects of the content inside the <div>.
5. **Blog Comments**
6. **Like/Unlike**
7. **My Blog Page**
8. **Edit Blog**