Prerequisite

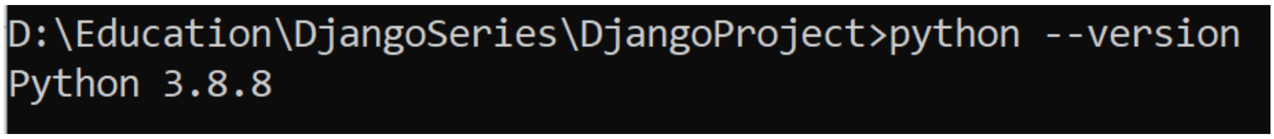
Python

- We are going to work on python 3.x for this tutorial series, so if you have not installed python yet. Go to https://www.python.org/downloads/ and install the required version.

Check python version:

```
python --version
```

If this does not work, use **python3 --version**



```
D:\Education\DjangoSeries\DjangoProject>python --version
Python 3.8.8
```

Installation

Optional settings for windows users

If you are using command line terminal with admin privileges, you might need to run the below command:

```
Get-ExecutionPolicy -List

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Isolated Python environment

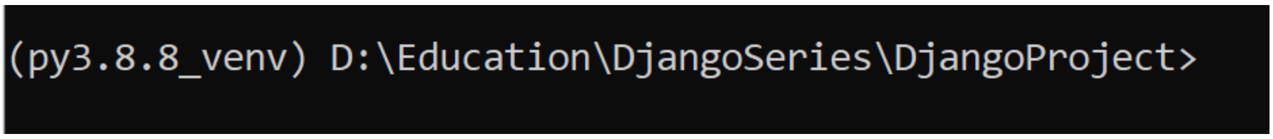
We can use the Python **venv** module to create isolated Python environments that allows you to use different package versions for different projects, which is far more practical than installing Python packages system-wide.

Create an isolated environment with the following command:

```
python -m venv my_env
```

To activate your virtual environment, run the following command:

For Ubuntu users	source my_env/bin/activate
For Windows users	my_env\Scripts\activate.bat



```
(py3.8.8_venv) D:\Education\DjangoSeries\DjangoProject>
```

Install required packages:

You can install required packages using the below command:

```
pip install package_name
e.g  pip install django
```

Check if django is installed correctly:

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

Creating your first project

Start django project by using the below command:

```
django-admin startproject my_project
```

Let's take a look at the project structure generated:

```
my_website.
|    manage.py
|
└──── my_website
      asgi.py
      settings.py
      urls.py
      wsgi.py
      __init__.py
```

These files are as follows:

- **manage.py**: Django's command-line utility for administrative tasks.
- **mysite**/: This is your project directory, which consists of the following files:
- __init__.py: An empty file that tells Python to treat the mysite directory as a Python module.
- **asgi.py**: This is the configuration to run your project as ASGI, the emerging Python standard for asynchronous web servers and applications.
- **settings.py**: This indicates settings and configuration for your project and contains initial default settings.
- **urls.py**: This is the place where your URL patterns live. Each URL defined here is mapped to a view.
- **wsgi.py**: This is the configuration to run your project as a Web Server Gateway Interface (WSGI) application.

Running development server:

```
python manage.py runserver
```

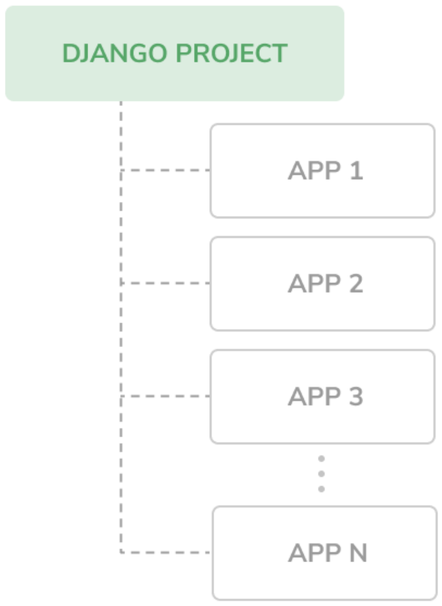
Now open **http://127.0.0.1:8000/** in your browser.You should see a page stating that the project is successfully running.

Projects and applications

In Django, a project is considered a Django installation with some settings. On the other hand, An application is a group of models, views, templates, and URLs. Applications interact with the framework to provide some specific functionalities and may be reused in various projects. **You can think of a project as your website, which contains several applications, such as a blog, wiki, or forum, that can also be used by other projects.**

Projects vs. apps

What's the difference between a project and an app? An app is a Web application that does something – e.g., a Weblog system, a database of public records or a small poll app. A project is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.



Creating a blog application

```
python manage.py startapp blog
```

This will create the basic structure of the application, which looks like the one below:

```
├────blog
│    │ admin.py
│    │ apps.py
│    │ models.py
│    │ tests.py
│    │ views.py
│    │ __init__.py
│    │
│    └────migrations
│          __init__.py
```

These files are as follows:

- **admin.py** : This is where you register models to include them in the Django administration site—using this site is optional.
- **apps.py** : This includes the main configuration of the blog application.
- **migrations** : This directory will contain database migrations of your
- **models.py** : This includes the data models of your application; all Django applications need to have a models.py file, but this file can be left empty.
- **tests.py** : This is where you can add tests for your application.
- **views.py** : The logic of your application goes here; each view receives an HTTP request, processes it, and returns a response.

Creating Views and adding routes

create home and about view inside blog **view.py**

```
from django.shortcuts import render
from django.http import HttpResponse

def home(request):
   return HttpResponse('<h1>Blog Home Page </h1>')

def about(request):
   return HttpResponse('<h1>Blog About Page</h1>')
```

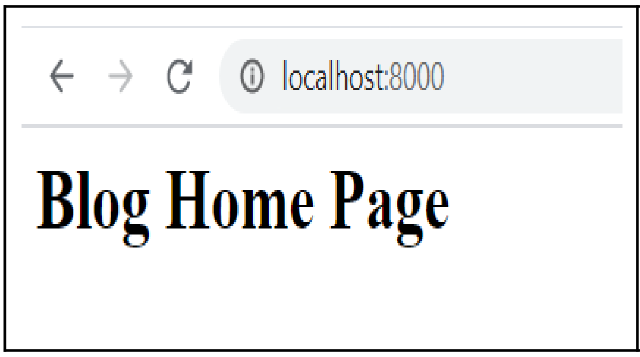
Add **urls.py** file inside blog app and add the following code:

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

urlpatterns = [
   path('', views.home, name='blog-home'),
   path('about/', views.about, name='blog-about'),
]
```

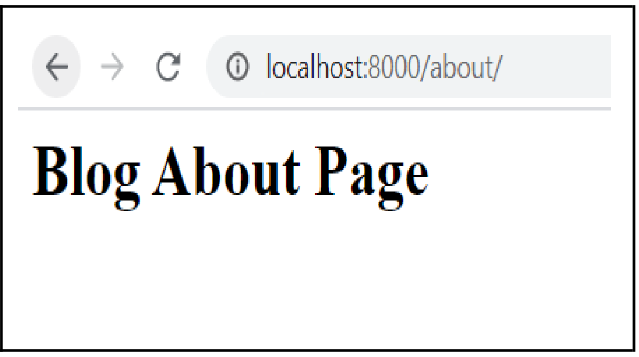
Add below code inside your project's **urls.py**

```
from django.urls import path, include

urlpatterns = [
   ….
   path('', include('blog.urls')),
]
```

Now run server and go to '**localhost:8000**' and check home page and /about page


localhost:8000

Blog Home Page


localhost:8000/about/

Blog About Page

Since we have created a blog app, we need to register it into INSTALLED_APPS in settings.py

```
INSTALLED_APPS = [
…….
‘blog.apps.BlogConfig’,
]
```

We can create a template that can be rendered with context data from views.
Create the following structure inside your blog app for templates:

```
templates
└────blog
          about.html
          base.html
          home.html
```

base.html serves as our **base file** which home.html and about.html extends. You can use bootstrap starter template to create base structure and then use jinja templating to use variables and update html.

Since we have created html files, we can now render these files from **blog/views.py**

```
def home(request):
    return render(request, 'blog/home.html')

def about(request):
    return render(request, 'blog/about.html')
```

We can also pass context data to the html that we can use in the html like the one shown below:

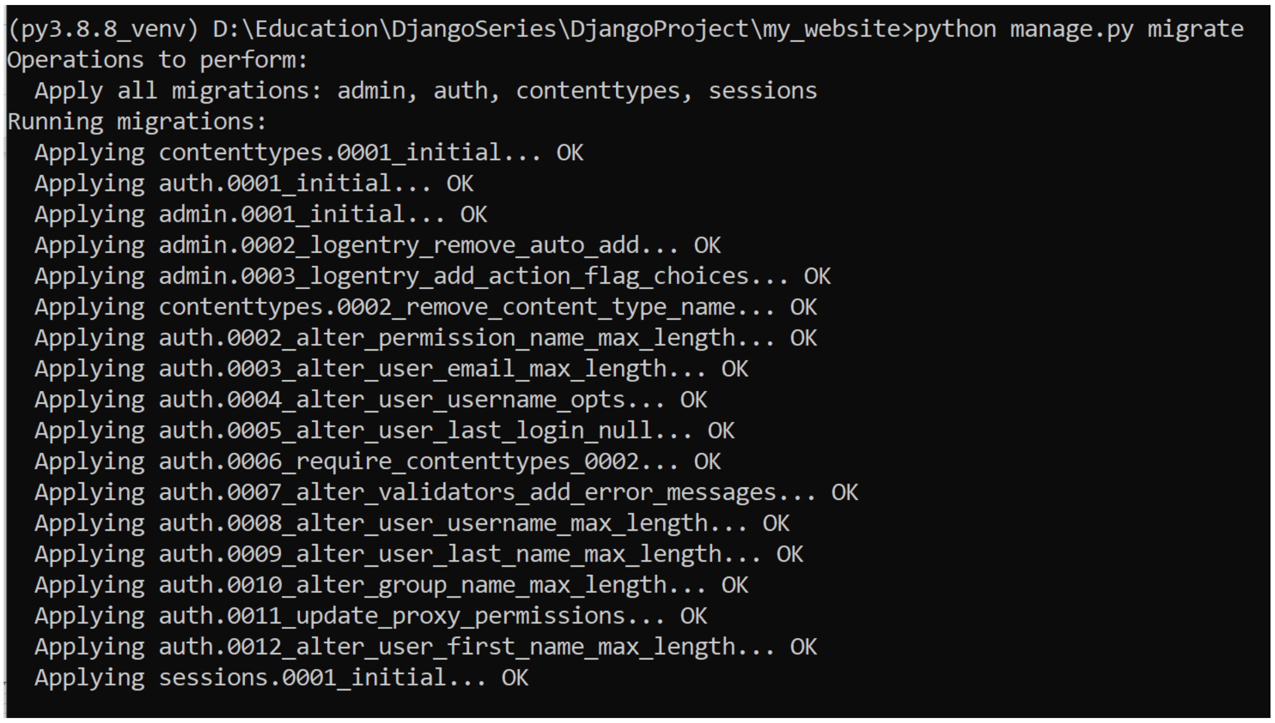
```
posts = [
    {'author':'Sumit Chawla',
    'date_posted':'16 March 2021',
    'title':'First Post',
    'content':'This is my first post'
    },
    {'author':'Sumit S Chawla',
    'date_posted':'16 March 2021',
    'title':'Second Post',
    'content':'This is my second post'
    }
]

def home(request):
    context = {
        'posts':posts,
    }
    return render(request, 'blog/home.html', context)
```

Running migrations

Django’s default user model is default and easy to use. But sometimes, it is more convenient to extend or create different user model to store extra information for the user. E.g description, location etc. For now, we will use the default User model, later on we will use the Profile model that maps the User model.

```
python manage.py migrate
```

You should see some migrations running like the one in the image shown below.



```
(py3.8.8_venv) D:\Education\DjangoSeries\DjangoProject\my_website>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Create Superuser and login to admin

Django comes with a built-in administration interface that is very useful for editing content. The Django site is built dynamically by reading your model metadata and providing a production-ready interface for editing content. You can use it out of the box, configuring how you want your models to be displayed in it.

The **django.contrib.admin** application is already included in the INSTALLED_APPS setting, so you don't need to add it.

Create superuser by running the following command:

```
python manage.py createsuperuser
```

Enter username, email and password to create superuser. Once, superuser is created, go to **/admin** (http://localhost:8000/admin) , enter username and password to access the Admin panel.

Designing schema for Posts model

A model is a Python class that subclasses django.db.models.Model in which each attribute represents a database field. Django will create a table for each model defined in the models.py file.

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User

class Post(models.Model):

        STATUS_CHOICES = ( ('draft', 'Draft'), ('published', 'Published'), )

        title = models.CharField(max_length=250)
        slug = models.SlugField(max_length=250, unique_for_date='published_at')
        author = models.ForeignKey(User, on_delete=models.CASCADE,
        related_name='blog_posts')
        content= models.TextField()
        published_at = models.DateTimeField(default=timezone.now)
        created_at = models.DateTimeField(auto_now_add=True)
        updated_at = models.DateTimeField(auto_now=True)
        status = models.CharField(max_length=10, choices=STATUS_CHOICES,
        default='draft')

        class Meta:
                ordering = ('-published_at',)

        def __str__(self):
                return self.title
```

* What is the difference between auto_now and auto_now_add?

Creating and applying migrations

First, we will need to create an initial migration for your Post model. In the root directory of your project, run the following command:

```
python manage.py makemigrations blog
```

The **sqlmigrate** command takes the migration names and returns their SQL without executing it.

```
python manage.py sqlmigrate blog 0001
```

Let's sync your database with the new model. Run the following command to apply existing migrations:

```
python manage.py migrate
```

Registering Post model in admin

Let's add your post model to the administration site. Edit the admin.py file of the blog application and make it look like this:

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

You can also customize the django admin site for a model using below options:

```
from django.contrib import admin
from .models import Post

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'published_at', 'status')
```

Let's customize the admin model with some more options, using the following code:

```
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'published_at', 'status')
    list_filter = ('status', 'created_at', 'published_at', 'author')
    search_fields = ('title', 'content')
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ('author',)
    date_hierarchy = 'published_at'
    ordering = ('status', 'published_at')
```

Creating model managers

objects is the default manager of every model that retrieves all objects in the database. However, we can also define custom managers for your models.

Edit the **models.py** file of your blog application to add the custom manager:

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super(PublishedManager,
        self).get_queryset()\
        .filter(status='published')

class Post(models.Model):
    # ...
    objects = models.Manager() # The default manager.
    published = PublishedManager() # Our custom manager.
```

- The first manager declared in a model becomes the default manager.
- You can use the **Meta** attribute **default_manager_name** to specify a different default manager.
- If you declare any managers for your model but you want to keep the objects manager as well, you have to add it explicitly to your model.

Passing posts to views

Now, since we have created a model for posts and have also created a custom model manager, we can now pass all the posts in **views.py** and remove the dict we defined earlier.

```
from .models import Post

def home(request):
    posts = Post.published.all()
    context = {
        'posts':posts,
    }
    ...
```

Working with Querysets:

Django comes with a powerful database abstraction API that lets you create, retrieve, update, and delete objects (CRUD Operation) easily. The Django object-relational mapper (ORM) is compatible with MySQL, PostgreSQL, SQLite, Oracle, and MariaDB. Once you have created your data models, Django gives you a free API to interact with them.

The Django ORM is based on QuerySets. A QuerySet is a collection of database queries to retrieve objects from your database. You can apply filters to QuerySets to narrow down the query results based on given parameters.

Open the terminal and run the following command to open the Python shell:

```
python manage.py shell
```

Creating an object:

```
>>> from django.contrib.auth.models import User
>>> from blog.models import Post

>>> user = User.objects.get(username='admin')
>>> post = Post(title='Another post',
                slug='another-post',
                content='Post Content',
                author=user)

>>> post.save()
```

Update object value:

```
>>> post.body = ‘Updated post body’’
>>> post.save()
```

Retrieving objects: We have already seen how to retrieve a single object using **get().** To retrieve all objects from a table, you just use the **all()** method on the default objects manager.

```
>>> all_posts = Post.objects.all()
```

Note that this QuerySet (Post.objects.all()) has not been executed yet. Django QuerySets are lazy, which means they are only evaluated when they are forced to be. This behavior makes QuerySets very efficient. If you don't set the QuerySet to a variable, but instead write it directly on the Python shell, the SQL statement of the QuerySet is executed because you force it to output results.

Using the filter() method: To filter a QuerySet, you can use the **filter()** method of the manager. For example, you can retrieve all posts from admin user using the following QuerySet:

```
>>> Post.objects.filter(author__username='admin')
```

*Queries with field lookup methods are built using two underscores, for example, publish__year, but the same notation is also used for accessing fields of related models, such as author__username.

Using exclude():

```
>>> Post.objects.filter(publish__year=2021)
                              .exclude(title__startswith='My')
```

Using order_by():

```
>>> Post.objects.order_by('title')
```

By default, Ascending order is implied. You can indicate descending order with a negative sign prefix: **'-title'**

Deleting objects: If you want to delete an object, you can do it from the object instance using the delete() method.

```
>>> post = Post.objects.get(id=1)
>>> post.delete()
```

QuerySets are only evaluated in the following cases:

- The first time you iterate over them
- When you slice them, for instance, Post.objects.all()[:3]
- When you pickle or cache them
- When you call repr() or len() on them
- When you explicitly call list() on them
- When you test them in a statement, such as bool() , or , and , or if

Learn more about querysets at: https://docs.djangoproject.com/en/3.1/topics/db/queries/

User Registration

Now, since we have created the Post model, a user should be able to register itself and login from Front End and not just the admin panel to be able to create posts and post comments.

Django comes with a built-in authentication framework that can handle user authentication, sessions, permissions, and user groups. The authentication system includes views for common user actions such as log in, log out, password change, and password reset.

Create a user app called account:

```
python manage.py startapp account
```

Register app in settings.py

```
INSTALLED_APPS = [
…….
‘account.apps.AccountConfig’,
]
```

Create a simple view for user registration:

```
from django.shortcuts import render
from django.contrib.auth.forms import UserCreationForm

def register(request):
   form = UserCreationForm()
   return render(request, ‘account/register.html’, {‘form’: form})
```

Similar to how we created templates folder for blog, we will create templates folder and then account folder inside it where we will define `**register.html**`

```
account
├────templates
│   │   └────account
│   │          register.html
```

Now, the next step is to add our view to url patterns: add register view inside my_website/urls.py
Add register view in URLS:

```
from account import views as user_views

urlpatterns= [
...
path('register/', user_views.register, name='register'),
]
```

If we go to register and do the signup, it will not add a user in our database as we have defined it as a POST request, but have not added the same in views. So, we will need to update register views to handle POST requests and create a user on signup.

```
from django.shortcuts import render, redirect
from django.contrib import messages

def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data.get('username','User')
            messages.success(request, f'Account created for {username}!')
            return redirect('blog-home')
    else:
        form = UserRegisterForm()
        return render(request, 'account/register.html', {'form': form})
```

In order to add a new field in forms, we need to create a new Form which inherits UserCreationForm. So, for that we will create a new file called **forms.py** inside account app.

```
from django import forms
from django.contrib.auth.models import User
from django.contrib.auth.forms import UserCreationForm


class UserRegisterForm(UserCreationForm):
    email = forms.EmailField()

    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']
```

Replace UserCreationForm with UserRegistrationForm in views and now, you will see an email field added in the form.

```
from .forms import UserRegisterForm

..
 if request.method == 'POST':
        form = UserRegisterForm(request.POST)
...
else:
        form = UserRegisterForm()
```

So, the complete register view looks like the one below:

```
from django.shortcuts import render, redirect
from django.contrib import messages
from .forms import UserRegisterForm

def register(request):
    if request.method == 'POST':
        form = UserRegisterForm(request.POST)
        if form.is_valid():
            form.save()
            username = form.cleaned_data.get('username')
            messages.success(request, f'Account created for {username}!')
            return redirect('blog-home')
    else:
        form = UserRegisterForm()
    return render(request, 'users/register.html', {'form': form})
```

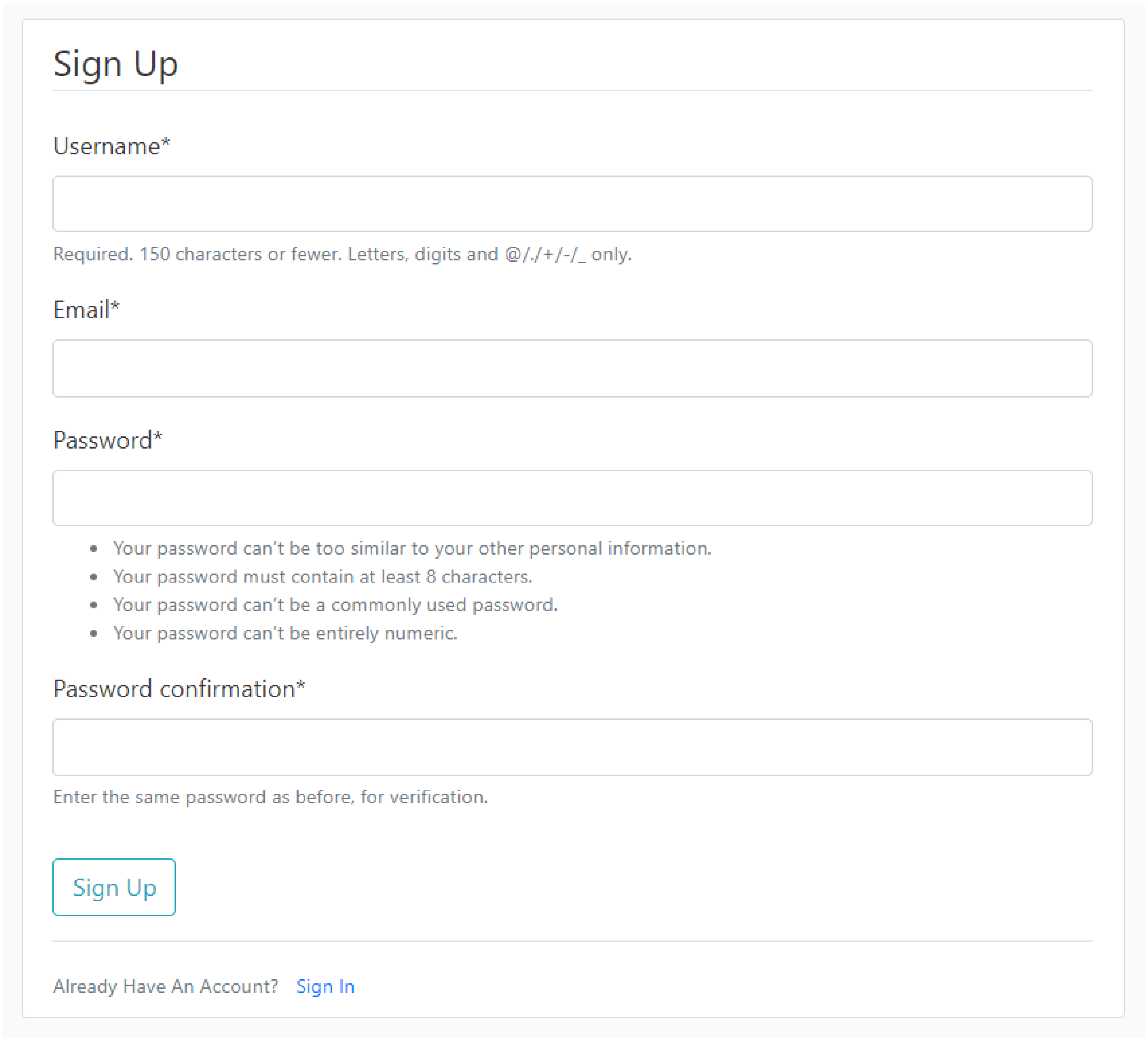
To make our form look nice, we can add django-crispy-forms which provides nice and clean styling. Install django crispy forms for adding some css to the form

```
pip install django-crispy-forms
```

Add crispy_forms in installed apps:

```
INSTALLED_APPS = [
    ...
    'crispy_forms',
]

CRISPY_TEMPLATE_PACK = ‘bootstrap4’
```

SignUp form looks like this now:



Sign Up

Username*

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Email*

Password*

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation*

Enter the same password as before, for verification.

Login and Logout System

Django comes in with a default auth view where we have a login and logout system. So, we are going to use that in our project. For that, we need to add auth_views in our project urls. Open **my_website/urls.py** and add the following code.

```
from django.contib.auth import views as auth_views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
    ....
]
```

Login and Logout view does not contain the templates, but contains the template_name in the view. So, we need to create custom templates for both login and logout. The default path is **registration/login.html** and **registration/logout.html**, but we can pass template_name in as_view(). Since we have an account app, it makes more sense to keep our login and logout templates in the same app.Update login and logout url to pass **template_name**.

```
path('login/', auth_views.LoginView.as_view(template_name='account/login.html'),
name='login'),
path('logout/', auth_views.LogoutView.as_view(template_name='account/logout.html'),
name='logout'),
```

Once we login,django tries to redirect us to a user profile, which we haven't created yet. But what we want is to redirect the user to the home page on login. We can define this inside settings.py

```
LOGIN_REDIRECT_URL = 'blog-home'
```

Now that we have created a login view, we can now redirect our users to the login page on successful registration. So we update our register view and redirect them to login.

```
def register(request):
    ...
    messages.success(request, f'Your account is created! you can now login')
    return redirect('login')
    ....
```

User Profile

It is important for a user to update it's details. So we will now create an endpoint where users can update their details. Create a new view inside account named profile

```
def profile(request):
    return render(request, 'account/profile.html')
```

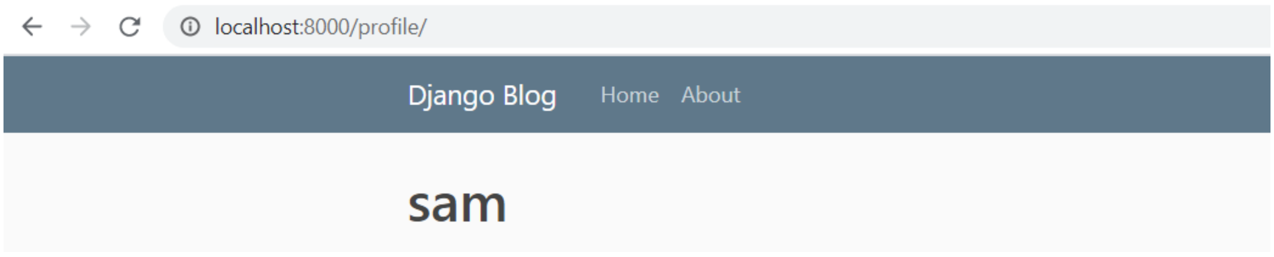
Now, create a **profile.html** inside the account app template.

```
{% extends 'blog/base.html' %}
{% block content %}
    <h1>{{user.username}}</h1>
{% endblock %}
```

And update our urls.py to add an endpoint for profile.

```
urlpatterns = [
...
path('profile/', user_views.profile, name='profile'),
...
]
```

Now, if you go to localhost:8000/profile, you will see your username like this:



sam

Since we only want to allow logged in users to go to the profile page, we need to add a built in decorator called **login_required**. So, let's add this decorator to the profile view.

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

@login_required
def profile(request):
    return render(request, 'account/profile.html')
```

Also, update LOGIN_URL in settings to 'login'

```
LOGIN_URL='login'
```

Create a profile model which extends the User model so that we can add extra fields in the user model. We'll create one in account app.

```
from django.db import models
from django.contrib.auth.models import User

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    image = models.ImageField(default='default.jpg', upload_to='profile_pics')

    def __str__(self):
        return f'{self.user.username} Profile'
```

We need to install a package called **pillow** that is used for processing images. So , install that package first. Next step is to do makemigration and then run migration.

```
>> pip install pillow
>> python manage.py makemigrations
>> python manage.py migrate
```

We also need to create a folder call **profile_pics** and add a default image for the profile.

Register your model in account/admin.py

```
from django.contrib import admin
from .models import Profile
```

```
admin.site.register(Profile)
```

Now,the profile_pic folder is fine, but it is in the base directory and that is not a good option if we are creating a multi app project. So, the ideal way is to provide a media root folder where all media is stored.

Settings.py: add MEDIA_ROOT and MEDIA_URL below STATIC_URL

```
import os

MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

If we create a profile, or update an image in one, we will see that the image is stored inside the media folder. To use the image url created, we need to serve static files and to do that during development, we can add that in urlpatterns.

```
from django.conf import settings
from django.conf.urls.static import static

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Everytime, a user is created we want to link that user to a profile automatically and for that we will use signals. So, create a file called signals inside the account app.

```
from django.db.models.signals import post_save
from django.contrib.auth.models import User
from django.dispatch import receiver
from .models import Profile


@receiver(post_save, sender=User)
def create_profile(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)


@receiver(post_save, sender=User)
def save_profile(sender, instance, **kwargs):
    instance.profile.save()
```

This will create a user profile or update one whenever we do some changes in the user model. To make this work, we need to add signals inside our AppConfig.

```
def ready(self):
        import account.signals
```

Update User Profile

The next thing we want to do is to be able to update the user profile. Now to do that, we will create Forms.update this inside account/forms.py

```
from .models import Profile

class UserUpdateForm(forms.ModelForm):
    email = forms.EmailField()
    class Meta:
        model = User
        fields = ['username', 'email']

class ProfileUpdateForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['image']
```

Update the newly created forms inside views

```
from .forms import UserRegisterForm, UserUpdateForm, ProfileUpdateForm

def profile(request):
        if request.method == 'POST':
        u_form = UserUpdateForm(request.POST, instance=request.user)
        p_form = ProfileUpdateForm(request.POST,
                                request.FILES,
                                instance=request.user.profile)
        if u_form.is_valid() and p_form.is_valid():
            u_form.save()
            p_form.save()
            messages.success(request, f'Your account has been updated!')
            return redirect('profile')

    else:
        u_form = UserUpdateForm(instance=request.user)
        p_form = ProfileUpdateForm(instance=request.user.profile)

    context = {
        'u_form': u_form,
        'p_form': p_form
    }

    return render(request, 'users/profile.html', context)
```

The image that is stored needs to be stored in a way that it does not take much space. So we can update our profile model to update the saved image like this:

```
from PIL import Image

Class Profile(models.Model):
    ...
   def save(self):
        super().save()

        img = Image.open(self.image.path)

        if img.height > 300 or img.width > 300:
            output_size = (300, 300)
            img.thumbnail(output_size)
            img.save(self.image.path)
```

Create, Update, Delete Posts

For this, we will use class based views as they provide more inbuilt functionalities to be used. There are different class based views and depending on the requirements, we can use ListView, DetailView, UpdateView etc.

Our home page contains the list of posts, so it is a good candidate for ListView. So, go ahead and create PostListView for our home page.

```
from django.views.generic import ListView

class PostListView(ListView):
    model = Post
    template_name = 'blog/home.html'
    context_object_name = 'posts'
    ordering = ['-date_posted']
```

Update urls.py to replace home view with PostListView

```
from .views import PostListView

urlpatterns = [
    path('', PostListView.as_view(), name='blog-home'),
    ...
]
```

Next, what we want is DetailView to check any specific post

```
from django.views.generic import DetailView

class PostDetailView(DetailView):
    model = Post
```

```
from .views import PostDetailView

urlpatterns = [
    path('post/<int:pk>/', PostDetailView.as_view(), name='post-detail'),
    ...
]
```

Now, we want a template for detail view. So we will copy the home template and just skip the loop. Next, update url links for posts in home.html

```
<h2><a class="article-title" href="{% url 'post-detail' post.id %}">{{ post.title }}</a></h2>
```

Then, we will add CreateView to add posts

```
from django.views.generic import CreateView

class PostCreateView(CreateView):
    model = Post
    fields = [‘title’, ‘content’]
```

```
from .views import PostCreateView

urlpatterns = [
    path('post/new/', PostCreateView.as_view(), name='post-create'),
    ...
]
```

For CreateView, we will be using a shared template that we can use for UpdateView. We can name it post_form.html

Now, if we create a new post using the front end, we will get an integrity error. To resolve that, add these below lines in the code:

```
 def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)
```

Now, it will create a post successfully, but for redirection, we will need to provide an absolute url, so, we will update the post model and define the `**get_absolute_url()**` method.

```
from django.urls import reverse

def get_absolute_url(self):
        return reverse('post-detail', kwargs={'pk': self.pk})
```

We are now able to create posts, but a user should be logged in at the time of creating the post as well as at the time of updating the post. So we will add a LoginMixinRequired class that handles this.

```
from django.contrib.auth.mixins import LoginRequiredMixin

class PostCreateView(LoginRequiredMixin, CreateView):
    ...
```

Also, we can create an UpdateView:

```
from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin

class PostUpdateView(LoginRequiredMixin, UserPassesTestMixin, UpdateView):
    model = Post
    fields = ['title', 'content']

    def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)

    def test_func(self):
        post = self.get_object()
        if self.request.user == post.author:
            return True
        return False
```

Update it in urls.py

```
from .views import PostUpdateView

path('post/<int:pk>/update/', PostUpdateView.as_view(), name='post-update'),
```

Now, the only one left is DeleteView. So go ahead and create one:

```
from django.views.generic import  DeleteView

class PostDeleteView(LoginRequiredMixin, UserPassesTestMixin, DeleteView):
    model = Post
    success_url = '/'

    def test_func(self):
        post = self.get_object()
        if self.request.user == post.author:
            return True
        return False
```

```
from .views import PostDeleteView

path('post/<int:pk>/delete/', PostDeleteView.as_view(), name='post-delete'),
```

Once, we are done with the views, update templates

Pagination

Ref: https://docs.djangoproject.com/en/3.1/topics/pagination/

We can do the pagination by using the default functionality given by Django.

We just need to add **paginate_by** in out PostListView:

```
class PostListView(ListView):
    model = Post
    template_name = 'blog/home.html'
    context_object_name = 'posts'
    ordering = ['-date_posted']
    paginate_by = 5
```

HitCount

Ref:

1. https://django-hitcount.readthedocs.io/en/2.0/installation.html
2. https://stackoverflow.com/questions/58005521/how-to-use-django-hitcount-package-to-build-a-view-count

Install hit-count package

```
pip install django-hitcount
```

Add django-hitcount to your INSTALLED_APPS:

```
# settings.py
INSTALLED_APPS = (
    ...
    'hitcount'
)
```

Run migrations : **python manage.py migrate**

Password Reset

We want to provide a functionality where a user can rest his/her password using email id.For this, we will be using django's inbuilt **PasswordResetView**. So, first we will add the url in urls.py

```
path('password-reset/',
        auth_views.PasswordResetView.as_view(
            template_name='account/password_reset.html'
        ),
        name='password_reset'),
```

We also need to add **PasswordResetDoneView** which is used after password reset is done.

```
path('password-reset/done/',
        auth_views.PasswordResetDoneView.as_view(
            template_name='account/password_reset_done.html'
        ),
        name='password_reset_done'),
```

Now, the only ones left are **PasswordResetConfirmView** and **PasswordResetCompleteView**. So lets add them too.

```
path('password-reset-confirm/<uidb64>/<token>/',
        auth_views.PasswordResetConfirmView.as_view(
            template_name='account/password_reset_confirm.html'
        ),
        name='password_reset_confirm'),
    path('password-reset-complete/',
        auth_views.PasswordResetCompleteView.as_view(
            template_name='account/password_reset_complete.html'
        ),
        name='password_reset_complete'),
```

Setting up secured password for email:
https://support.google.com/accounts/answer/185833?hl=en

Once you are done setting up app password :

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
EMAIL_HOST_USER = os.environ.get('EMAIL_USER')
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASS')
```

Using AWS S3 for File Storage

While saving user model, we forgot to add **args** and **kwargs**, so let's update that:

```
def save(self, *args, **kwargs):
    super().save(*args, **kwargs)
```

We can create an s3 bucket using the following link: https://s3.console.aws.amazon.com/
If you do not have an account, create one and follow the instructions given and you should be good to go.

For our deployment on heroku server, we can follow these instructions:
https://devcenter.heroku.com/articles/s3

Updating CORS:
https://docs.aws.amazon.com/AmazonS3/latest/userguide/ManageCorsUsing.html

Once we're done with this, we want to create an IAM user for accessing S3 bucket. So, search for IAM in the search bar and create a new user.
E.g username: **django-blog-user** , access-type: **Programmatic access**

On Next: Use Attach existing policies directly , select S3 Full Access

We also need to install boto3 and django-storages to be able to use S3 storage. So go ahead and install them using **pip install boto3** and **pip install django-storages**

Django-storages ref: https://django-storages.readthedocs.io/en/latest/index.html

Update project settings to add storage related variables:

```
INSTALLED_APPS = [
    ...
    'storages'
]

AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCESS_KEY')
AWS_STORAGE_BUCKET_NAME = os.environ.get('AWS_STORAGE_BUCKET_NAME')

AWS_S3_FILE_OVERWRITE = False
AWS_DEFAULT_ACL = None

DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
```

Deploying your application to Heroku

SignUp	https://signup.heroku.com/

Install GIT	https://git-scm.com/downloads
Install Heroku CLI	https://devcenter.heroku.com/articles/heroku-cli#download-and-install

Once installed, type **heroku** in your terminal and it should print out the list of commands available.
Use this command to login to Heroku using CLI: **heroku login**
Install gunicorn : **pip install gunicorn**

Now, for deployment, we also need **requirements.txt** file. So, go ahead and create one:

```
pip freeze > requirements.txt
```

Github Essentials:

Initialize repository	git init
Add gitignore file	.gitignore > Python Gitignore File
Add all the files in repository	git add -A
Commit all files	git commit -m “initial commit”

Create Heroku app

```
heroku create app_name
```

This will create an app on heroku, now you can either go to your app using the URL shown in terminal or type **heroku open** and it will open up the application.
Next, type “**git push heroku master**” and it will detect the requirements and install them.

Probably it will run into an error as we have not defined STATIC_ROOT in our settings. So we will define one now:

```
STATIC_ROOT = os.path.join(‘BASE_DIR’, ‘staticfiles’)
```

Repeat the above steps : add, commit and push files.
Now to check the error, if any go to **heroku logs --tail**

Once this is done, we will need to create a file called **Procfile:**
Ref: https://devcenter.heroku.com/articles/django-app-configuration

Repeat the steps again : add, commit and push files.

Keys need to be set in environment variables using: **heroku config:set key=value**

```
SECRET_KEY = os.environ.get('SECRET_KEY')
DEBUG = (os.environ.get('DEBUG_VALUE') == 'True')
EMAIL_HOST_USER = os.environ.get('EMAIL_USER')
EMAIL_HOST_PASSWORD = os.environ.get('EMAIL_PASS')
AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCESS_KEY')
AWS_STORAGE_BUCKET_NAME = os.environ.get('AWS_STORAGE_BUCKET_NAME')
```

Setup Postgres on heroku : https://devcenter.heroku.com/articles/heroku-postgresql#local-setup

Check **heroku addons** to see if the db is already created. If not created. Create one using this command:

```
heroku addons:create heroku-postgres:hobby-dev
```

Type **heroku pg** for more info on postgres.
Next install django-heroku to use related functions: **pip install django-heroku**

Update settings:

```
Import django_heroku

django_heroku.settings(locals())
```

Push changes to heroku master
heroku run python manage.py migrate
heroku run bash
python manage.py createsuperuser
exit
heroku releases

Django Checklist: https://docs.djangoproject.com/en/3.1/howto/deployment/checklist/

Django Cheat Sheet

* Use python or python3 depending on your installation

Command	Description
python -m venv my_env	To create a virtual environment named my_env using the venv package. replace **my_env** with your environment name.
source my_env/bin/activate	Activate the virtual environment (Ubuntu) make sure you are in the same folder where you created the virtual environment.
my_env\Scripts\activate.bat	Activate the virtual environment (Windows) make sure you are in the same folder where you created the virtual environment.
pip install django	To install the django package. Replace **django** with the package that you want to install
python -m django --version	To check django version
django-admin startproject my_project	Create a django project. Replace **my_project** with your project name
python manage.py runserver	Run your server on localhost with default port 8000
python manage.py startapp blog	Create a blog app.replace blog with whatever app you want to create Make sure you are inside your my_project folder to run this command successfully.
python manage.py makemigrations blog	Creates new migrations based on the changes you have made in your model. In this case, it checks changes for the blog app.
python manage.py sqlmigrate blog 0001	Pass on the app and the migration file number to check the sql command it creates which runs on doing migrate
python manage.py migrate	Run commands created during makemigrations to update DB
python manage.py showmigrations	lists a project's migrations and their status.
python manage.py createsuperuser	Create super user for your project
pip freeze	Check installed packages in your environment.

Templates link: https://gist.github.com/SamChawla/eb9bbbee764a6121e87a4c78b152ee80

- ORM [Done]
- User Registration [Done]
- Login Logout System [Done]
- Profile [Done]
- CBVs (Class Based Views) [Done]
- No. of Views on a Post [Done]
- Detail Page View [Done]
- Pagination [Done]
- Password Reset [Done]
- Setting up environment variables [Done]
- AWS S3 File Storage [Done]
- Deployment (**Heroku** or AWS or pythonanywhere) [Done]
- Settings Config [Done]
- Writing Test Cases (Optional)

Reference Links:

1. Django Architecture and Structure
2. Password Reset
3. Django Rest framework - Complete guide
4. DRF Official Docs