**VIDEO 1**

Install Python 3, Django

Set up Sublime text

Cd to projects directory (CMD)

Django-admin startproject nameofproject

Cd into nameofproject

set up venv and activate

pip install django

Open project folder in sublime text

(Wsgi is how Python app and web server communicate)

Urls.py has admin page routing

**Basic website should load now: python manage.py runserver (localhost:8000)**

**Video 2**

users and posts will need their own in Django.

Users = password and profile update pages/templates

Main/posts app = pages that hold the post creation and showing

Python manage.py startapp nameofapp

Go to settings.py, installed apps > 'users.apps.UsersConfig'

Blog/views.py – httpresponse import

From django.http import HttpResponse

put in function that handles web request for home page of blog app. Views.py tells app how the page will look

def home(request):

return HttpResponse('<h1>Blog Home</h1>')

Create new file in blog folder – urls.py

blog/urls.py – This is where you’ll tell the app how where to route web traffic for the blog app. Import views.py

from . import views

from django.urls import path

urlpatterns = [

path('', views.home, name='blog-home'),

]

Also have to tell the project (overarching) urls.py where to go for blog related addresses.

project/urls.py

from django.urls import path, **include**

urlpatterns = [

path('admin/', admin.site.urls),

path('blog/', include('blog.urls')),

]

Check to see page loads (8000:blog/)

REPEAT

blog/views – add about function for about page. Tells app how about page will look.

blog/urls – add path for about page use home as template. Refer to views.about

if we want blog home page to be main home page, leave reference to blog.urls blank in project/urls

**Video 3**

Create templates directory within blog(app folder). Then create another folder in that directory called appname. Makes no fucking sense but that’s best practice.

Create html files in blog>templates>blog folder. Typing html + tab equals minimalist html

Blog folder > apps.py

Copy BlogConfig from this page(inherits from appconfig). This is to add blog app to list of installed apps so Django knows to look here for templates. Recommened to add app config to project settings py.

Installed apps = add ‘appname.apps.appnameConfig’. Do this for every app.

blog/views.py REPLACE WITH BETTER

from django.shortcuts import render (already in)

We’re modifying httpresponse to instead render using our html templates.

def home(request):

return render(request, ‘appname/home.html’)

I guess render knows to look for templates folder in this directory? Remove HttpResponse

Do the same for about.

Dummy data: put near top of blog views.py

posts = [ {‘author’: ‘test’, ‘Second thing’: ‘data for second thing’, ‘etc’: ‘data for etc’,}. second dictionary, etc etc ] – don't forget comma after brackets

within def home, set new dictionary called context. Put in this dictionary as 3rd argument in retun render

context = {

'posts': posts

} return render(request,'blog/home.html', context)

Go back to home html template to loop through posts. Post this logic in body tag of template:

{% for post in posts %}

<h1>{{ post.title }}</h1>

<p>By {{ post.author }} on {{ post.date\_posted }}</p>

<p>{{ post.content }}</p>

{% endfor %}

Title logic for head section of template:

{% if title %}

<title>Daniel Blog - {{ title }}</title>

{% else %}

<title>Django Blog</title>

{% endif %}

Back to views page: Check title logic > return render(request, 'blog/about.html', {'title': 'About page goes here'})

Making base html page:

New html called base.html in templates blog. Combine home and about html pages. Make content block in body tags.

{% block content %}{% endblock %}

Remove duplicate html code from non-base templates by adding this at the top:

{% extends "blog/base.html" %}

{% block content %}

Unqique code goes here (indent it )

{% endblock content %}

Add meta tags from bootstrap to top of head in base template. Also add jscript code from bootstrap to bottom of base template body tags. Now we’re using bootstrap classes.

Wrap content block with bootstrap <div class=”container”> contentblock </div>

add corey navigation html (or make your own). Top of body of base.html

<https://github.com/CoreyMSchafer/code_snippets/blob/master/Django_Blog/snippets/navigation.html>

This will replace the div container that had the block content

<https://github.com/CoreyMSchafer/code_snippets/blob/master/Django_Blog/snippets/main.html>

Need to create a static directory to hold css files. Within blog. Then create a folder within static called blog. Because that makes sense. Create main.css within static/blog/

<https://github.com/CoreyMSchafer/code_snippets/blob/master/Django_Blog/snippets/main.css>

We have to link that main.css in our base html template. So html knows what to reference.

{% load static %} at top of base.html. The following line goes in the head where all the other stylesheets are linked.

<link rel="stylesheet" type="text/css" href="{% static 'blog/main.css' %}">

This snippet replaces the html inside the for loop in the home.html template. Makes it fancy.

<https://github.com/CoreyMSchafer/code_snippets/blob/master/Django_Blog/snippets/article.html>

Change href links to django url tags. Instead of / (home), use”{% url ‘blog-home’ %}. These are the names given to the urls in the urls.py file **WE HERE NOW – find html for home.html that matches insta**

**Video 4**

Access admin page: First run migrations to create update database.

python manage.py makemigrations. No changes detected. This just detects changes.

To apply changes, python manage.py migrate

Next create superuser. python manage.py createsuperuser.

Sign in and create regular user for testing. No actual changes to programming files

**Video 5**

**ORM – Object relational mapper**

Can use different databases without changing code. PostGre production, sql development, just change settings.

blog/models.py – create models to save in db. user Posts in this case. Django has builtin user model already (createsuperuser remember). models already imported. Create a class for each model/type of thing you need in the database. Check out doc to see what fields you can use.

from django.utils import timezone (for timezone utility down below)

from django.contrib.auth.models import User – importing the User model table. Start a one-to-many relationship

class Post(models.Model):

title = models.CharField(max\_length=100)

content = models.TextField()

date\_posted = models.DateTimeField(default=timezone.now) auto\_now=True = current time of update, auto\_now\_add = only on first creation. I think this is a good one for date posted.

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

ForeignKey refers to other model table. on\_delete says what happens if user gets deleted. Cascade deletes posts when user deletes. But not other way around.

makemigrations, migrate to update database. Migrations folder has new file.

See SQL code: python manage.py sqlmigrate blog(appname) 0001

Query database using Python shell: python manage.py shell 12:30

under same class in models.py

def \_\_str\_\_(self): - this function tells django how to display things when you query in shell

return self.title

**blog/views.py**

Gotta import posts model to pass them into the views

from .models import Post

change context from ‘posts’: Posts to Post.object.all(), as long as all fields are the same as dummy data no problem. Otherwise you’d have to change the template.

fields should be the same from models.py > views.py > template(home.html)

change timezone formatting:

templates > blog > home.html > post.date\_posted|date:”F d, Y” check docs for this shit

Can’t see posts in admin page without registering post model

**Blog/admin.py**

Register models here: from .models import Post

admin.site.register(Post)

**Video 6**

Create new app for user registration. User app. python manage.py startapp users. Add app to installed apps list. Use name of class in apps.py file of new ap. Project settings.py > installed apps > ‘users.apps.UsersConfig’

newapp/views.py

We’re gonna set up the register page view. Django does this for us. Import this form.

from django.contrib.auth.forms import UserCreationForm

def register(request):

form = UserCreationForm() - This creates a new instance of the form imported.

return render(request, ‘users/register.html’, {‘form’: form})

This is telling django to use the template html and sending UCF as variable to template.

Now make that register template. Inside users > templates > users > register.html (no fucking sense). Use about page as starting base. Inside block content. put it in a div class=”content-section” (this is purely for style purposes. Inside that div, put in a

<form method=”POST”>

{% csrf\_token %} – this is needed. Hidden tag. cross site request forgery token. Protects against attacks. Required for forms to work.

<fieldset class=”form-group”>

<legend class=”border-bottom mb-4”>Join Today</legend> This is for the text that shows on form

{{ form,as\_p }}

</fieldset> fieldsets are used to group related elements in a form

</form>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Sign up</button> - Submit button code

</div>

</form> code below is for the “already have an account?” link

<div class="border-top pt-3">

<small class="text-muted">

Already have account? <a class="ml-2" href="#">Sign In</a>

</small>

</div></div> close out those divs

Next step, create a url pattern to use this html. To direct people here. Urls > views > templates

Temporary solution if you dont wanna create a new urls page for this app, go to main urls and

from users(or appname) import views as app\_views at the top

path('register/', user\_views.register, name='register')

Runserver and make sure it works. Looks ugly. Return to template. form.as\_p

Still not registering students yet. In register view, we’re just creating blank form and rendering.

POST requests tell django where to send form data. Set conditional in users.views > register class

import redirect from django.shortcuts at the top

from django.contrib import messages

def register(request):

if request.method == 'POST': - if http request is post, make use of form data being sent

form = UserCreationForm(request.POST) – put form in variable

if form.is\_valid():

username = form.cleaned\_data.get('username') – form.cleaned\_data is a dictionary

messages.success(request, f'Account created for {username}!') – flash message code. Imported

return redirect('blog-home') – redirect after success. Imported redirect function

else:

form = UserCreationForm()

return render(request, 'users/register.html', {'form': form})

Now to add the messages code in the base html template so that any page can display messages

Put this code right above the content block in base.html

{% if messages %}

{% for message in messages %}

<div class="alert alert-{{ message.tags }}">

{{ message }}

</div>

{% endfor %}

{% endif %}

Add form.save() to users/views.py to save User created by form. Now adding email to form:

Create forms.py in users folder. users/forms.py

from django import forms – this import the forms module

from django.contrib.auth.models import User – import User model. We’re creating users

from django.contrib.auth.forms import UserCreationForm – import Django default user form

class UserRegisterForm(UserCreationForm): - We’re using default form to create custom form

email = forms.EmailField() – adding email field to UCF. Default = required. otherwise required=false

class Meta: - meta class

model = User –we’re telling them what model we are creating/registering

fields = ['username', 'email', 'password1', 'password2'] – what order to display fields of form

Go back to users/views.py and import this new form

from .forms import UserRegisterForm then replace UCF with URF

Then remove UCF from import lines, its’ not used. Check functionality.

Style changes for forms should be in template. Using Crispy forms.

pip install django-crispy-forms

Then tell django it’s installed

Project settings > installed apps > ‘crispy\_forms’

Add setting to bottom to make crispy form using bootstrap 4

CRISPY\_TEMPLATE\_PACK = 'bootstrap4'

Back to register.html

{% load crispy\_forms\_tags %} underneath extends

form|crispy – remove as\_p. That’s it?!

**video 7**

Creating login page for users. First need to import login and logout views in project urls.py

from django.contrib.auth import views as auth\_views – the below are class-based views. add em. Django built in.

path('login/', auth\_views.LoginView.as\_view(template\_name=’users/login.html’), name='login'),

path('logout/', auth\_views.LogoutView.as\_view(template\_name=’users/logout.html’), name='logout'),

Now create these templates in users/templates/users login/out.html. Use register as starting point.

Replace legend, submit button text and muted text with relevant prompts instead. So like instead of Join, it will say log in. Instead of already got account? It will say need an account? With link to register url. {% url ‘register’ %}. Do same in register.html

To set a redirect url for when users login, in project settings.py: LOGIN\_REDIRECT\_URL = 'blog-home'

users/views.py – change the redirect for register go to login page now instead of home page

messages.success(request, f'Your account {username} has been created! You are now able to log in.')

return redirect('login')

users/logout.html –

<h2>You have been logged out.</h2>

<div class="border-top pt-3">

<small class="text-muted">

<a href="{% url 'login' %}">Log in again</a>

</small>

</div>

Go to base template and insert code to check for login/log out status.

<!-- Navbar Right Side -->

<div class="navbar-nav">

{% if user.is\_authenticated %}

<a class = blah blah href={% url ‘profile’ %}”> profile</a>

<a class="nav-item nav-link" href="{% url 'logout' %}">Logout</a>

{% else %}

<a class="nav-item nav-link" href="{% url 'login' %}">Login</a>

<a class="nav-item nav-link" href="{% url 'register' %}">Register</a>

{% endif %}

</div>

users/views.py – create profile view for signed in users

def profile(request):

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

Create users/profile.html – use login as template

Delete everything and just use <h1>{{ user.username }}</h1>

project/urls.py – add profile view to project urls

path('profile/', user\_views.profile, name='profile'),

Put in login required decorator to restrict access to profile page. Back to user/views.py

from django.contrib.auth.decorators import login\_required

@login\_required (above function profile)

Tell Django where to find login route so if people try to access restricted page, it prompts to login.

LOGIN\_URL = ‘login’ – put in settings.py

**Video 8**

Add profile picture functionality. New models in

users/models.py

We are extending the existing User model. So we gotta import django contrib blah blah User

Then create Profile Class that will extend off User. OneToOne field because one profile per User. Create image field called image. default specifies what image to use at first. Upload to specifies folder to upload to. duh. Add an \_\_str\_\_ method to tell Django how to display this class. Without this, Django just goes “profile object 12323erdere”. With this, it actually prints the username.

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'

pip install Pillow if you don’t have it already. It’s needed to use the imagefield. Then python manage.py makemigrations. Then python manage.py migrate. To view user profiles on admin page, we need to register model in the admin file of the app (Users in this case).

users/admin.py

from .models import Profile - Then admin.site.register(Profile)

By default, upload\_to puts images in a root folder. To change this, go to settings.py.

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media') – tell django where to save images.

MEDIA\_URL = '/media/' – How to access image folder through browser

users/templates/users/profile.html

Adding profile pic code to this html to display profile picture. Inside block content:

<div class="content-section">

<div class="media">

<img class="rounded-circle account-img" src="{{ user.profile.image.url }}">

<div class="media-body">

<h2 class="account-heading">{{ user.username }}</h2>

<p class="text-secondary">{{ user.email }}</p>

</div>

</div>

</div>

project/urls.py – adding images to browser. Letting django know where to find images (development)

from django.conf.urls.static import static

from django.conf import settings

The following is put outside the urlpatterns (below it):

if settings.DEBUG:

urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

To force users to make profile with every new users, we need a signal file:

users/signals.py

from django.db.models.signals import post\_save – signal that fires when object is saved

from django.contrib.auth.models import User – the object that will be the sender of signal

from django.dispatch import receiver – function that gets signal and does something. It gets triggered

from .models import Profile – The model that we will be created when user signal is triggered

@receiver(post\_save, sender=User) – This says perform function below when user object created

def create\_profile(sender, instance, created, \*\*kwargs): - create profile function

if created:

Profile.objects.create(user=instance) – if user is created, create a profile for user using the current instance.

@receiver(post\_save, sender=User)

def save\_profile(sender, instance, \*\*kwargs):

instance.profile.save() – Save the profile whenever user makes changed.

Now we gotta import signals inside ready function in

users/apps.py

Inside class UsersConfig –

def ready(self):

import users.signals

**VIDEO 9**

Users/forms.py – Adding a User and Profile UpdateForm to complement the register form.

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

field = ['image']

users/views.py – Add forms just created to the import field at the top. Add this code to profile code above return.

if request.method == 'POST':

u\_form = UserUpdateForm(request.POST, instance=request.user) – adds current user to form

p\_form = ProfileUpdateForm(request.POST,

request.FILES, - necessary to add files or images via user update

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, - putting form into context

'p\_form': p\_form – putting form into context, gotta be a dictionary

}

return render(request, ‘users/profile.html’, context)

users/templates/profile.html

copy form code from register.html. Insert into lowest section past the divs

<form method="POST" enctype="multipart/form-data"> - this enctype important to update info

{% csrf\_token %}

<fieldset class="form-group">

<legend class="border-bottom mb-4">Profile Info</legend>

{{ u\_form|crispy }} – user-update form

{{ p\_form|crispy }} – profile update form

</fieldset>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Update</button>

</div>

</form>

users/models.py – overwrite save method to resize images.

From PIL import Image

At the bottom of class profile:

Overwriting the save method. Creating our own save method.

def save(self):

super().save() – super means “parent class”. This saves the user image.

img = Image.open(self.image.path) – open the image that user saved

if img.height > 300 or img.width > 300: - if it’s bigger than 300x300, resize and overwrite.

output\_size = (300, 300)

img.thumbnail(output\_size)

img.save(self.image.path)

blog/templates/home.html – add user’s profile image next to their name on their posts

Put under article class:

<img class="rounded-circle article-img" src="{{ post.author.profile.image.url }}">

**Video 10**

We’re creating class-based views that will handle a lot of the logic behind the scenes. There’s list class views and details class views. Also update and delete views.

Class views look for this template by default unless you add template\_name variable:

<app name>/<model>\_<viewtype(list, detail etc)>.html

blog/views.py

from django.views.generic import ListView, DetailView, CreateView, UpdateView, DeleteView– Django has generic list views we can use.

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

Class PostListView(ListView): - Creating a class using the the Django generic as a base

model = Post - Using the Post model that was already imported.

template\_name = ‘blog/home.html’

context\_object\_name = ‘posts’ - set this variable to match the context name from above

ordering = [‘-date\_posted’] – the minus sign makes the ordering go from newest to oldest. Otherwise it’s vice versa.

class PostDetailView (DetailView):

model = Post

class PostCreateView(LoginRequiredMixin, CreateView):

model = Post

fields = ['title', 'content']

def form\_valid(self, form): - this sets the author

form.instance.author = self.request.user – this is setting the author as current logged in user.

return super().form\_valid(form) – then form is validated.

Then go to blog/models.py to tell Django how to redirect after creation of post.

Create view template should be called: <app>/<model>\_form.html

OR create an attribute in this view called “success\_url”? and set it to home or whatever.

Now we need to force users to be logged in to view the post creation screen.

from django.contrib.auth.mixins import LoginRequiredMixin

Import that mixin to the left of createview in the PostCreateView class.

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)

class PostDeleteView(LoginRequiredMixin, UserPassesTestMixin, DeleteView):

model = Post

def test\_func(self):

post = self.get\_object()

if self.request.user == post.author:

return True

return False

blog/urls.py

from .views import PostListView – import (PostListView, PostDetailView,

PostCreateView, PostUpdateView, PostDeleteView) we just created

PostListView.as\_view() in place of views.home

Add new route below:

path('post/<int:pk>/', PostDetailView.as\_view(), name='post-detail'),

path('post/new/', PostCreateView.as\_view(), name='post-create'),

path('post/<int:pk>/update/', PostUpdateView.as\_view(), name='post-update'), - will use same form template as the create view. Easy.

path('post/<int:pk>/delete/', PostDeleteView.as\_view(), name='post-delete'), - post\_confirm\_delete.html

The int tell Django that it should only expect to see integers there. The pk = primary key of the post. DetailView expects pk.

Create html template that follows template name scheme Django expects

blog/post\_detail.html

copy all of home.html, remove for loop, remove h2 title link, replace “post” with “object” because that is the default name that Django expects (can be changed with context\_name variable above).

<article class="media content-section">

<img class="rounded-circle article-img" src="{{ object.author.profile.image.url }}">

<div class="media-body">

<div class="article-metadata">

<a class="mr-2" href="#">{{ object.author }}</a>

<small class="text-muted">{{ object.date\_posted|date:"F d, Y" }}</small>

{% if object.author == user %}

<a class="btn btn-secondary btn-sm mt-1 mb-1" href="{% url 'post-update' object.id %}">Update</a>

<a class="btn btn-danger btn-sm mt-1 mb-1" href="{% url 'post-delete' object.id %}">Delete</a>

{% endif %}

</div>

<h2 class="article-title">{{ object.title }}</h2>

<p class="article-content">{{ object.content }}</p>

</div>

</article>

blog/post\_form.html

Copy users/register.html template. Remove the legend text and the submit button text and put something contextual instead.

blog/models.py

redirect will redirect to a specific route. Reverse will return the full URL to that route as a string

from django.urls import reverse

at bottom of class Post:

def get\_absolute\_url(self):

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

blog/post\_confirm\_delete.html

copy from post\_form.html

<div class="content-section">

<form method="POST">

{% csrf\_token %}

<fieldset class="form-group">

<legend class="border-bottom mb-4">Delete Post</legend>

<h2>Are you sure you want to delete the post "{{ object.title }}"?</h2>

</fieldset>

<div class="form-group">

<button class="btn btn-outline-danger" type="submit">Yes, Delete</button>

<a class="btn btn-outline-secondary" href="{% url 'post-detail' object.id %}">Cancel</a>

</div>

</form>

</div>

blog/base.html

Where the profile and logout buttons are:

<a class="nav-item nav-link" href="{% url 'post-create' %}">New Post</a>

FIX FOR TYPE ERROR SAVE()

users/models.py down by the overwritten save method:

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

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

**VIDEO 11**

Pagination. Don’t pull down too much information at once. Paginator object is what we need. Example in shell:

from django.core.paginator import Paginator

posts = [‘1’,’2’.......] This is a list

p = Paginator(posts, 2) will create a paginator object using your list. 2 in a page.

p.num\_pages == 3

for page in p.page\_range: This will loop through each page

print(page)

p1 = p.page(1) – to access a specific page

p1.number = number of page

p1.object\_list will return the objects part of the page

p1.has\_previous() will return true/false

p1.next\_page\_number() will equal 2

**home/views.py**

No need to import the paginator object apparently

Under postListView:

paginate\_by = 2

blog/home.html – add pagination links

Below of current endfor

{% if is\_paginated %}

{% if page\_obj.has\_previous %}

<a class="btn btn-outline-info mb-4" href="?page=1">First</a>

<a class="btn btn-outline-info mb-4" href="?page={{ page\_obj.previous\_page\_number }}">Previous</a>

{% endif %}

{% for num in page\_obj.paginator.page\_range %} – loops through all pages in paginator

{% if page\_obj.number == num %} – if page is current page

<a class="btn btn-info mb-4" href="?page={{ num }}">{{ num }}</a> - just highlight current page

{% elif num > page\_obj.number|add:'-3' and num < page\_obj.number|add:'3' %} – if page is close to current page, then display them

<a class="btn btn-outline-info mb-4" href="?page={{ num }}">{{ num }}</a>

{% endif %}

{% endfor %}

{% if page\_obj.has\_next %}

<a class="btn btn-outline-info mb-4" href="?page={{ page\_obj.next\_page\_number }}">Next</a>

<a class="btn btn-outline-info mb-4" href="?page={{ page\_obj.paginator.num\_pages }}">Last</a>

{% endif %}

{% endif %}

**blog/views.py**

from django.shortcuts import render, get\_object\_or\_404 – to grab user posts using username (or give 404 page)

from django.contrib.auth.models import User – we need to import the User model

Starting from the PostListView

class UserPostListView(ListView):

model = Post

template\_name = 'blog/user\_posts.html' – we’ll create this later

context\_object\_name = 'posts'

paginate\_by = 5

def get\_queryset(self): - this will tell django which user’s posts to use

user = get\_object\_or\_404(User, username=self.kwargs.get('username'))

return Post.objects.filter(author=user).order\_by('-date\_posted') – the date post order has to go here now instead of up top because we’re modifying the get\_query\_set.

{% url 'user-posts' post.author.username %} put this in the href for every link to the author in html.

blog/user\_posts.html

Copy from home.html because it has pagination

{% extends "blog/base.html" %}

{% block content %}

<h1 class="mb-3">Posts by {{ view.kwargs.username }} ({{ page\_obj.paginator.count }})</h1>

{% for post in posts %}

<article class="media content-section">

<img class="rounded-circle article-img" src="{{ post.author.profile.image.url }}">

<div class="media-body">

<div class="article-metadata">

<a class="mr-2" href="{% url 'user-posts' post.author.username %}">{{ post.author }}</a>

<small class="text-muted">{{ post.date\_posted|date:"F d, Y" }}</small>

</div>

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

<p class="article-content">{{ post.content }}</p>

</div>

</article>

{% endfor %}

{% if is\_paginated %}

{% if page\_obj.has\_previous %}

<a class="btn btn-outline-info mb-4" href="?page=1">First</a>

<a class="btn btn-outline-info mb-4" href="?page={{ page\_obj.previous\_page\_number }}">Previous</a>

{% endif %}

{% for num in page\_obj.paginator.page\_range %}

{% if page\_obj.number == num %}

<a class="btn btn-info mb-4" href="?page={{ num }}">{{ num }}</a>

{% elif num > page\_obj.number|add:'-3' and num < page\_obj.number|add:'3' %}

<a class="btn btn-outline-info mb-4" href="?page={{ num }}">{{ num }}</a>

{% endif %}

{% endfor %}

{% if page\_obj.has\_next %}

<a class="btn btn-outline-info mb-4" href="?page={{ page\_obj.next\_page\_number }}">Next</a>

<a class="btn btn-outline-info mb-4" href="?page={{ page\_obj.paginator.num\_pages }}">Last</a>

{% endif %}

{% endif %}

{% endblock content %}

**VIDEO 12**

Email and password reset

project name\_urls.py – We’re gonna add the password-reset view and password email sent successfully view. Also need a password\_reset\_confirm route

path('password-reset/', auth\_views.PasswordResetView.as\_view(template\_name='users/password\_reset.html'),

name='password\_reset')

path('password-reset/done', auth\_views.PasswordResetDoneView.as\_view(template\_name='users/password\_reset\_done.html'),

name='password\_reset\_done')

**The following path has a token and uid needed to reset password to the user who created the password reset request. This link shows up in the email for the user.**

path('password-reset-confirm/<uidb64>/<token>/',

auth\_views.PasswordResetConfirmView.as\_view(template\_name='users/password\_reset\_confirm.html'),

name='password\_reset\_confirm'),

path('password-reset-complete', auth\_views.PasswordResetCompleteView.as\_view(template\_name='users/password\_reset\_complete.html'),

name='password\_reset\_complete'),

*users/templates/users/password\_reset.html-* copy from login.html

change legends and button name. Remove bottom div

*password\_reset\_done.html*

{% extends "blog/base.html" %}

{% block content %}

<div class="alert alert-info">

An email has been sent with instructions to reset your password.

</div>

{% endblock content %}

*password\_reset\_confirm.html*

{% extends "blog/base.html" %}

{% load crispy\_forms\_tags %}

{% block content %}

<div class="content-section">

<form method="POST">

{% csrf\_token %}

<fieldset class="form-group">

<legend class="border-bottom mb-4">Reset Password</legend>

{{ form|crispy }}

</fieldset>

<div class="form-group">

<button class="btn btn-outline-info" type="submit">Reset Password</button>

</div>

</form>

</div>

{% endblock content %}

*password\_reset\_complete.html*

{% extends "blog/base.html" %}

{% block content %}

<div class="alert alert-info">

Your password has been set.

</div>

<a href="{% url 'login' %}">Sign In Here</a>

{% endblock content %}

***login.html*** – add password reset link to login page

<small class="text-muted ml-2">

<a href="{% url 'password\_reset' %}">Forgot Password?</a>

</small>

***Settings.py*** – add email variables

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')

**VIDEO 13**

users/models.py – modify save method

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

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

Sign into AWS, create S3 bucket, CORS configuration, paste code from Heroku,

Create user with less permission using IAM, Permissions, AS3FullAccess, get access and secret key, put in environmental variable,

pip install boto3

pip install django-storages

**settings.py** –

add ‘storages’, to installed apps list. AT BOTTOM:

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\_2')

AWS\_S3\_FILE\_OVERWRITE = False

AWS\_DEFAULT\_ACL = None

DEFAULT\_FILE\_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'

remove pillow file resizing

users/models.py - comment out save method

Something about variables with windows is fucking up. Had to manually reference the bucket.

**Option 1: Deploying to Heroku**

Create heroku account: thedrog email and thedrog password

We need to use git. Already installed.

Install Heroku CLI. Windows program. To test, type in Heroku in cmd.

python

import secrets

secrets.token\_hex(24)

Copy string and set as

SECRET\_KEY environement variable. Also make a DEBUG\_VALUE = True env variable

heroku config:set SECRET\_KEY = “sfdfdf”

dito DEBUG\_VALUE = “False”

ditto AWS access key, aws secret key, email and email pass

**project/settings.py**

import django\_heroku (installed later down)

SECRET\_KEY = os.environ.get(‘SECRET\_KEY’)

DEBUG = (os.environ.get('DEBUG\_VALUE') == 'True')

ALLOWED\_HOSTS = ‘drogrefreshapp.herokuapp.com’

above static\_url:

STATIC\_ROOT = os.path.join(BASE\_DIR, 'staticfiles')

django\_heroku.settings(locals())

create in root: **Procfile**

web: gunicorn refresh.wsgi

web = process type

gunicorn = command for python to talk to web server

wsgi = web service gateway interface

heroku login

go to project directory

pip install gunicorn

create requirements.txt (file that says what dependencies are needed for your app)

pip freeze – tell you all the pip installs you got. Copy and paste that into project folder

git init if you havent used git already. CLI version should be installed

Put .gitignore file in project directory

Commit to git.

heroku create drogrefreshapp

heroku git:remote –a drogrefreshapp

git push heroku master

Install postgres on machine

pip install django-heroku

git –add –A

git commit –m “message here”

git push heroku master

if get missing table: heroku run python manage.py migrate

heroku run python manage.py createsuperuser

App should run now with no posts.

Check django deployment checklist