***DJANGO***

* **INSTALLATION & SETUP**
  + **Python -m pip install Django**
  + If error, then **python -m pip install Django –user**
  + To check version of Django--- **python -m django –version**
  + Go to path where Django installed**—cd C:\Users\riygupta\AppData\Roaming\Python\Python310\Scripts**
  + Then Django-admin
  + Navigate to your path where u want to create a project then
  + Then create project--- **Django-admin startproject nameOfProject**
  + If it gives error then go in environment variables **---** Path environment variable add the path where u have Django-admin.py which will be path mentioned before.
  + To run the Django project – go to terminal in pycharm
  + To open terminal-view-tool windows—terminal
  + Then **python manage.py runserver**
  + Then to create folder for your modules(app)—features of project
  + In terminal run—**python manage.py startapp nameOfApp**
* **VIEWS AND URLS:**
* Create urls file in your app folder
* In views file create a function and return the response in form of Http
* In Urls file, create a list of urlPatterns in which it takes two arguments:
  + First is the url string to be attached in url(/January)
  + Second is the function that we want to call when we hit the url that we created in views file
* In urls file that we already have in our project before, in that add the path for this urls file, it takes 2 arguments:
  + First the url which will be appending in all the urls
  + Scend the urls file in ur app folder
* **DYNAMIC PATH SEGMENTS and PATH CONVERTERS:**
  + If we want to dynamically change the url then in urls file in first argument pass parameter that will dynamically change in angular brackets(<month>)
  + Then use this parameter in the views function
  + Also , if we want to specify the datatype for our parameter then we can specify inside angular brackets<str:month>
* path("<int:month>", views.monthly\_challenge\_by\_number)
* **REDIRECTS**:
* Inside this we can redirect one view function to another view function using **HttpResponseRedirect("/challenges/" + redirect\_month)**
  + In this first argument specifies the path of url
  + Second specifies the parameter
* **REVERSE AND NAMED URLS:**
  + This is used in redirecting concept as there is an issue if we change the url at one place we have to change at every place that is something that we don’t want
  + So, Django provides us the reverse function concept and named urls concept
  + In this in url file where we defined the urls in that we can provide name argument and provide a name
* **path("<str:month>", views.monthly\_challenge, name="month-challenge")**
  + then in views file create a variable and use reverse function from djanago.urls import and
  + reverse takes 2 arguments
    - first is the name of named url that we passed in urls file
    - second is the parameters that is takes as args=[]
    - example: **redirect\_path = reverse("month-challenge", args=[redirect\_month])**
* **RETURNING HTML**
  + In this where we are returning httpresponse, there instead we can return html content like :
* **Response\_data = "<h1>{}</h1".format(challenge\_text**) and return this
* If it’s just text then **return HttpResponseNotFound("<h1>This month is not supported</h1>")**
* **TEMPLATES AND STATIC FILES**
  + **ADDING AND REGISTERING TEMPLATES**
* We will not be writing the html code inside the python file
* For that we will create a templates folder and inside that another folder same as name of your app
* Inside that create a html file
* We created the same name folder as if you go in settings file in your project and see a templates dictionary then there you can see the **app\_dirs = True**, this means all your apps are loaded and it will have templates folder which it will automatically load for you
* You just have to **register** your app
* So, in **settings** file, in **INSTALLED\_APPS** section, give the name of your app and its loaded
* Now to use this html file in your view you have to import from **Django.templates.loader render\_to\_string**, and use it to see response
* **response\_data = render\_to\_string("challenges/challenge.html")  
  return HttpResponse(response\_data)**
* We created that challenges folder inside the templates folder because if consider we have multiple apps then it will be difficult for Django to identify the file, so to avoid clashing names its important to have app folder and then its template files
* **RENDERING TEMPLATES:**
  + Instead of using render\_to\_string function Django provides shortcut for it as render, which takes 2 arguments one is request and another is your template name
* **render(request,"challenges/challenge.html")**
* **DYNAMIC HTML/VARIABLE INTERPOLATION**
  + Now we are going to make html file more dynamic, so for that the render function takes another argument which is a dictionary and then Django uses that key in your html file to render
  + In views file**, render(request, "challenges/challenge.html", {  
     "text": challenge\_text  
    })**
  + in html file, **<h2>{{text}}</h2>**
  + If we want to pass parameter then just pass the parameter in function that we were already doing and just inject in html
* **render(request, "challenges/challenge.html", {  
   "text": challenge\_text,  
   "month\_name": month.capitalize()  
  })**
* **FILTERS**
  + We will be doing all business logic separate, so instead of capitalizing in views we will do that in html file
  + Django provides us that as filters, we can use pipe symbol in injecting and then use them
* **<h1>{{month-name|title}} challenge</h1>**
* **TAGS AND “FOR” TAG**
  + So, we will write all our business logic in templates file
  + So, to render list items in html, we have the following syntax
* **{% for month in months %}  
   <li><a href="">{{month-name|title}}</a></li>  
  {% endfor %}**
* **Dynamic urls**
  + For dynamic url’s also, Django Templates language provides us some like this
* **<a href="{% url 'month-challenge' month %}">,**
* Where month-challenge is the named url that we specified in urls.py file and month is the argument that we want to pass, we can pass 2 or more params separated by space
* **THE “IF” TAG**
* **{% if text is not None %}  
   <h2>{{ text }}</h2>  
  {% else %}  
   <p>There is no challenge for this month yet</p>  
  {% endif %}**
* **TEMPLATE INHERITANCE**
  + Now maybe we have base html file which will have footer and header to be included in all the html files
  + So create a templates folder in your project
  + Then create a base.html file in your folder
  + Then refer this file in settings file in templates section as
* **BASE\_DIR / "templates"**
  + Then create blocks inside that like below then we will use these blocks in our html files
* **<title>{% block page-title %} {% endblock%}</title>  
  </head>  
  <body>  
   {% block content %} {% endblock%}  
  </body>  
  </html>**
  + Here page-title and content is of your choice u can give any name
  + To make it base for every file we will need to extend as follow
* **{% extends "base.html" %}  
   {% block page-title %}  
   All Challenges   
   {% endblock %}  
  {% block content %}  
  <ul>  
   {% for month in months %}  
   <li><a href="{% url 'month-challenge' month %}">{{month|title}}</a></li>  
   {% endfor %}  
  </ul>  
  {% endblock %}**
  + Then using the blockname we will make changes accordingly
* **INCLUDE PARTIAL TEMPLATES**
  + If we want to have a template to include in some file but not in all
  + Then we can create another folder in templates/challenges as includes
  + Then use them in the files where want to use
  + Example to include that is in html file:
* **{% include "challenges/includes/header.html" %}**
* **ADDING STATIC FILES**
  + Create a static folder in your app folder
  + Then create challenges folder (app name folder)
  + Then create css file
  + To load in html file below is the example
* **{% load static %}  
  {% block css\_files %}  
   <link rel="stylesheet" href="{% static 'challenges/challenges.css' %}">  
  {% endblock %}**
* **GLOBAL STATIC FILES**
  + We can create a static folder in our project which will be used globally
  + In that we can create css files which can be used by all html files
  + So to load them ,we have to go in settings file
* **STATICFILES\_DIRS = [  
   BASE\_DIR / 'static'  
  ]**
* **DATA MODELS**
  + **DJANGO MODELS**
* To create models we have to go in models.py file
* Then create entities based on our project
* This class should inherit the models.Model class of Django
* Example below
* **class Book(models.Model):  
   title = models.CharField(max\_length=50)  
   rating = models.IntegerField()**
* here we don’t have to take an ID column that in inbuilt in Django
* **MIGRATIONS**:
* To push data to database
* To create migrations go in your project folder in terminal
* And type **python manage.py makemigrations**
* Then u will a folder created and a file
* To migrate them to db **python manage.py migrate**
* **INSERTING DATA**
  + To insert data do, **python manage.py shell**
  + Then there – **from appName.models import className**
  + Create objects
  + **harry\_potter = Book(title = "Harry Potter 1 - The Philosopher's Stone ", rating=5)**
  + **then save, harry\_potter.save()**
  + **lord\_of\_the\_rings = Book(title = "Lord Of The Rings", rating=4)**
  + **lord\_of\_the\_rings.save()**
* **GETTING ALL ENTERIES**
  + **className.objects.all()**
  + **ex: Book.objects.all()**
  + To get output in string format do this
  + Write a function in models.py file
* **def \_\_str\_\_(self):  
   return f"{self.title} ({self.rating})"**
  + So, this suggests whenever the output is printed it should display this
* **UPDATING MODELS AND MIGRATIONS**
  + If we want to update our database by adding new fields or adding validations
  + We can add validations as Django has inbuilt library validators from Django.core.validators
  + Like this we can add field validations as follows
* **rating = models.IntegerField(validators=[MinValueValidator(1), MaxValueValidator(5)])**
* **author = models.CharField(null=True, max\_length=100)  
  isbestselling = models.BooleanField(default=False)**
  + to update do makemigartions and migrate
* **UPDATING DATA**
  + To update data
  + **harry\_potter = Book.objects.all()[0]**

**harry\_potter.title**

**"Harry Potter 1 - The Philosopher's Stone "**

**lotr = Book.objects.all()[1]**

**lotr.title**

**'Lord Of The Rings'**

**harry\_potter.author = "J.K Rowling"**

**harry\_potter.isbestselling = True**

**harry\_potter.save()**

* **DELETING DATA**
  + **harry\_potter.delete()**
* **Creating Objects**
  + In this instead of instantiating again a
  + We can use this also to create objects
  + **Book.objects.create(title="New Story",rating=2, author="Murali",isbestselling=False)**
* **QUERYING AND FILTERING**
  + We can only use get if the condition passed return single value, otherwise use filter
  + For less than we cannot use (<) instead we have methods as below
  + **Book.objects.get(title="My Story")**
  + **Book.objects.filter(isbestselling=True)**
  + **Book.objects.filter(rating\_\_lt=3)**
* **OR CONDITIONS**
  + from django.db.models import Q
  + For or use |, for and use ,
  + **Book.objects.filter(Q(rating\_\_lt=3) | Q(isbestselling=True))**
  + **Book.objects.filter(Q(rating\_\_lt=3) | Q(isbestselling=True), Q(author="J.K Rowling"))**
* **RENDERING QUERIED DATA IN TEMPLATES**
  + In views.py file
  + From .models import Book
* **def index(request):  
   books = Book.objects.all()  
   return render(request, "book\_outlet/index.html", {  
   "books": books  
   })**
  + then in index.html render it as we used to do it
* **{% for book in books %}  
  <li>  
   {{ book.title }} (Rating : {{ book.rating }})  
  </li>  
  {%endfor%}**
* **MODEL URLS**
  + Instead of using that url tag in html file
  + We can also create specific model utility(function) as
* **def get\_absolute\_url(self):  
   return reverse ('book-detail', args=[self.id])**
* and then use it as
* **<a href="{{ book.get\_absolute\_url }}">{{ book.title }} </a>**
* **ADDING SLUGFIELD AND OVERWRITING SAVE**
  + In models file add, to get slug field in url
* **slug = models.SlugField(default="", null=False) # Harry potter 1 => harry-potter-1  
  def save(self, \*args, \*\*kwargs):  
   self.slug = slugify(self.title)  
   super().save(\*args, \*\*kwargs)**
* then do migrations
* also update data as--- **Book.objects.get(title="New Story").save()**
* **AGGREGATE METHODS**
  + To use aggregate methods

**# or order\_by('-rating') --descending order**

* **def index(request):  
   books = Book.objects.all().order\_by("title")  
   num\_of\_books = books.count()  
   avg\_rating = books.aggregate(Avg('rating'))  
   return render(request, "book\_outlet/index.html", {  
   "books": books,  
   "total\_number\_of\_books":num\_of\_books,  
   "average\_rating":avg\_rating  
   })**
* **DJANGO ADMIN**
  + In urls.py file we have a path admin
  + So, in browser, got to /admin page
  + To set username, password
  + In console do- **python manage.py createsuperuser**
  + Then login
  + **ADDING MODELS TO ADMIN AREA**
    - There is an admin file, in that
    - Do this
* **from django.contrib import admin  
  from .models import Book  
  # Register your models here.  
  admin.site.register(Book)**
  + **CONFIGURE ADMIN SETTINGS**
    - In this, we can configure admin settings
    - This means the /admin page where we want how to display and its settings
* **class BookAdmin(admin.ModelAdmin):  
   prepopulated\_fields = {"slug": ("title",)}  
   list\_filter = ("author", "rating",)  
   list\_display = ("title", "author",)**
* **RELATIONSHIPS**
  + **One To Many Relation:**
    - AUTHOR 🡪 BOOK= author can have many books and one book one author
    - Create another class then to use in another model, foreign key.
* **author = models.ForeignKey(Author, on\_delete=models.CASCADE, null=True, related\_name="books")**
* ForeignKey is used for many to one relationship.
* Here on\_delete specifies, if author is deleted then what Django has to do we did CASCADE, which implies that if author deleted delete the related books
* We can choose other options too
  + - Here related\_name specifies that if we query author then how can we find book related to that author in author table, so we can use books to find that
* **OneToOneField is use for one to one relation**
* **class Address(models.Model):  
   street = models.CharField(max\_length=80)  
   postal\_code = models.CharField(max\_length=5)  
   city = models.CharField(max\_length=50)  
    
   def \_\_str\_\_(self):  
   return f"{self.street} {self.postal\_code} {self.city}"**

**class Meta:  
 # is use to change the display name in the admin page for the table  
 verbose\_name\_plural = "Address Entries"**

* **class Author(models.Model):  
   first\_name = models.CharField(max\_length=100)  
   last\_name = models.CharField(max\_length=100)  
   address = models.OneToOneField(Address, on\_delete=models.CASCADE, null=True)**
  + **Many To Many**
    - It does not have the on\_delete attribute
    - Because here Django creates a mapping table behind the scenes, so when we delete the book the mapping row is deleted
* **published\_countries = models.ManyToManyField(Country, null=False)**
* **DATABASE QUERIES**



* **FORMS**
  + In html file we add form tag and if post then we need to specify the method as post
* Also, when specifying post method, do specify **{% csrf\_token %}**
  + For post method, instead of returning a response, we redirect to specific page
* **def review(request):  
   if request.method == 'POST':  
   entered\_username = request.POST['username']  
   print(entered\_username)**

**#here /thank-you is the url that we registerd in urls.py which indirectly calls the thank\_you fxn   
 return HttpResponseRedirect("/thank-you")  
 return render(request, "reviews/review.html")  
  
def thank\_you(request):  
 return render(request, "reviews/thank\_you.html")**

* + **USING DJANGO FORM CLASS**
    - Instead of doing all that, we can create a form.py file
    - In that we will create a class as follows
* **from django import forms  
  class ReviewForm(forms.Form):  
   user\_name = forms.CharField(max\_length=100)**
  + - then in views file, we can directly import this form and use, also Django has built-in validations
* **def review(request):  
   if request.method == 'POST':  
   form = ReviewForm(request.POST)  
   if form.is\_valid():  
   print(form.cleaned\_data)  
   return HttpResponseRedirect("/thank-you")  
   form = ReviewForm()  
   return render(request, "reviews/review.html",{  
   "form" : form  
   })**
* In html file, we don’t need to use label and input we can directly use this form
* **<form action="/" method="POST">  
   {% csrf\_token %}  
   {{ form }}  
   <button type="submit">Send</button>  
   </form>**
  + **VALIDATION USING DJANGO FORMS**
* **if request.method == 'POST':  
   form = ReviewForm(request.POST)  
   if form.is\_valid():  
   print(form.cleaned\_data)  
   # here /thank-you is the url that we registered in urls.py, which indirectly calls the thank\_you fxn  
   return HttpResponseRedirect("/thank-you")  
  else:  
   form = ReviewForm()  
  return render(request, "reviews/review.html", {  
   "form": form  
  })**
  + **CUSTOMIZING FORM CONTROLS:**
* **class ReviewForm(forms.Form):  
   user\_name = forms.CharField(label="Your Name",max\_length=100, error\_messages={  
   "required":"Your name must not be Empty!!",  
   "max\_length": "Please enter a shorter name"  
   })**
  + **CUSTOMIZING RENDERED HTML**
* **<form action="/" method="POST">  
   {% csrf\_token %}  
   <div class="form-control {% if form.user\_name.errors %} errors {% endif %}">  
   {{ form.user\_name.label\_tag }}  
   {{ form.user\_name }}  
   {{ form.error\_messages }}  
   </div>  
   <button type="submit">Send</button>  
  </form>**
  + **STORING FROM DATA IN DATABASE**
    - Create a class in models
    - Then in views get that values from form and save
* **def review(request):  
   if request.method == 'POST':  
   form = ReviewForm(request.POST)  
   if form.is\_valid():  
   review = Review(user\_name=form.cleaned\_data['user\_name'], review\_text=form.cleaned\_data['review\_text'],  
   rating=form.cleaned\_data['rating'])  
   review.save()  
   print(form.cleaned\_data)  
   # here /thank-you is the url that we registered in urls.py, which indirectly calls the thank\_you fxn  
   return HttpResponseRedirect("/thank-you")  
   else:  
   form = ReviewForm()  
   return render(request, "reviews/review.html", {  
   "form": form  
   })**
* **MODEL FORMS**
* We can use this ModelForm class and let Django create a form for us
* So, in that we can create Meta class and use model variable to tell Django which model to use
* Here we are letting django infer our form from our model directly
* **class ReviewForm(forms.ModelForm):  
   class Meta:  
   model = Review  
   # to include all fields do this otherwise fields=[], which u want to include otherwise u can use exclude also  
   fields = '\_\_all\_\_'  
   # exclude = ['owner\_comment']**
* **CONFIGURING MODEL FORMS**
  + If we want to add labels and error\_messages, we can add that also using ModelForms
* **class ReviewForm(forms.ModelForm):  
   class Meta:  
   model = Review  
   # to include all fields do ths otherwise fields=[], whch u wnt to include otherwise u can use exclude also  
   fields = '\_\_all\_\_'  
   # exclude = ['owner\_comment']  
   labels = {  
   "user\_name": " Your Name",  
   "review\_text": "Your Feedback",  
   "rating": "Your Rating"  
   }  
   error\_messages = {  
   "user\_name": {  
   "required": "Your name must not be Empty!!",  
   "max\_length": "Please enter a shorter name"  
   }  
   }**
* **SAVING DATA WITH MODEL FORM**
  + **Using model form we can directly save the form to db**
* **def review(request):  
   if request.method == 'POST':  
   form = ReviewForm(request.POST)  
   if form.is\_valid():  
   form.save()  
   return HttpResponseRedirect("/thank-you")  
   else:  
   form = ReviewForm()  
   return render(request, "reviews/review.html", {  
   "form": form  
   })**
* **UPDATING FORM DATA IN DATABASE**
  + - In this it is same like creating form, only we need to pass instance so that we get all the values from database and then we just update the changes
    - In this, we can use the same template as of create.
* **def updateProject(request, pk):  
   project= Project.objects.get(id=pk)  
   form = ProjectForm(instance=project)  
   if request.method == 'POST':  
   form = ProjectForm(request.POST, instance=project)  
   if form.is\_valid():  
   form.save()  
   return redirect('projects')  
   context = {'form': form}  
   return render(request, 'projects/project\_form.html', context)**
* **DELETING FORM DATA IN DATABASE**
  + **In this, we need to create separate template as**
  + **<form action="" method="post">  
     {% csrf\_token %}  
     <p>Are you sure you want to delete "{{object}}"?</p>  
     <a href="{% url 'projects' %}">Go Back</a>  
     <input type="submit" value="Confirm">  
    </form>**
* **def deleteProject(request, pk):  
   project = Project.objects.get(id=pk)  
   if request.method == 'POST':  
   project.delete()  
   return redirect('projects')  
   context = {'object': project}  
   return render(request, 'projects/delete\_template.html', context)**
* **CLASS BASED VIEWS**
  + We can use class based views, which will have functions for different methods(get, post)
* **class ReviewView(View):  
   def get(self, request):  
   form = ReviewForm()  
   return render(request, "reviews/review.html", {  
   "form": form  
   })  
    
   def post(self, request):  
   form = ReviewForm(request.POST)  
   if form.is\_valid():  
   form.save()  
   return HttpResponseRedirect("/thank-you")  
   return render(request, "reviews/review.html", {  
   "form": form  
   })**
  + also in urls.py file we have to change the path
* **path("", views.ReviewView.as\_view()),**
* **CLASS VIEWS**
  + **TEMPLATE VIEW**
    - This view is specific to render html template
* **template\_name = "reviews/thank\_you.html"  
    
  def get\_context\_data(self, \*\*kwargs):  
   context = super().get\_context\_data(\*\*kwargs)  
   context["message"] = "This Works"  
   return context**
* **LIST VIEW**
* **class ReviewListView(ListView):  
   template\_name = "reviews/review\_list.html"  
   model = Review  
   # this is the name of the object that we use in template to render in for loop  
   context\_object\_name = "reviews"  
    
   # for filtering we can do this  
   def get\_queryset(self):  
   base\_query = super().get\_queryset()  
   data = base\_query.filter(rating\_\_gt=4)  
   return data**
* **DETAIL VIEW**
* **class SingleReviewView(DetailView):  
   template\_name = "reviews/single\_review.html"  
   model = Review**
* If we use detail view then in url change it to pk
* **path("reviews/<int:pk>", views.SingleReviewView.as\_view())**
* **FORM VIEW**
* **class ReviewView(FormView):  
   form\_class = ReviewForm  
   template\_name = "reviews/review.html"  
   success\_url = "/thank-you"  
    
   def form\_valid(self, form):  
   form.save()  
   return super().form\_valid(form)**
* **CREATE VIEW**
* **class ReviewView(CreateView):  
   model = Review  
   form\_class = ReviewForm  
   template\_name = "reviews/review.html"  
   success\_url = "/thank-you"**
* **UPLOAD FILES**
  + Create a form with fileField
  + Create a model to store image in db, but usually we don’t do that as it makes the server slow
  + So, we create a folder in our project not app specific
  + Then we need to specify that folder in settings.py file
* **MEDIA\_ROOT = BASE\_DIR / "uploads"**
* **class UserProfile(models.Model):**

**# fileField is used take all file whereas imagefield only takes images  
 image = models.ImageField(upload\_to="images")**

* + In view:
* **class CreateProfileView(View):  
   def get(self, request):  
   form = ProfileForm  
   return render(request, "profiles/create\_profile.html", {  
   "form": form  
   })  
    
   def post(self, request):**

**#here the request.FILES is used to send the images uploaded in form, don’t forget to add the enctype attribute in form as multipart/form-data  
 submitted\_form = ProfileForm(request.POST, request.FILES)  
 if submitted\_form.is\_valid():**

**# here image is the name of the field in model, columnName in table  
# user\_image is the specified name in the form  
 profile = UserProfile(image=request.FILES["user\_image"])  
 profile.save()  
 # here the key image is the name attribute of the input type=file in html  
 # print(request.FILES['image'])  
 return HttpResponseRedirect("/profiles")  
 return render(request, "profiles/create\_profile.html", {  
 "form": submitted\_form  
 })**

* + **USING CREATE VIEW**
    - Instead of using view, we can use CreateView, no need to create form
    - In create View, we don’t have to do anything as below:
* **class CreateProfileView(CreateView):  
   template\_name = "profiles/create\_profile.html"  
   model = UserProfile  
   fields = "\_\_all\_\_"  
   success\_url = "/profiles"**
* Model
* **class UserProfile(models.Model):  
   # fileField is used take all file whereas imagefield only takes images  
   image = models.ImageField(upload\_to="images")**
  + **SERVING THE FILES**
    - **To view files uploaded in browser, we will create a ListView view**
* **class ProfilesView(ListView):  
   model = UserProfile  
   template\_name = "profiles/user\_profiles.html"  
   context\_object\_name = "profiles"**
  + - **Then we will create a template and also add the url in urlpatterns**
* **<ul>  
   {% for profile in profiles %}  
   <li>  
   <img src="{{ profile.user\_image.url }}">  
   </li>  
   {% endfor %}  
  </ul>**
  + - **But to render this we need to register the url in setting file**
* **# you can give any name between two slashes  
  MEDIA\_URL = "/user-media/"**
  + - **Also then in urls.py file, project specific**
* **urlpatterns = [  
   path('admin/', admin.site.urls),  
   path("", include("reviews.urls")),  
   path("profiles/", include("profiles.urls"))  
   ] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)**
* **SESSIONS**
  + Ongoing connection between server and client
  + Check in setting.py there should be a sessionMisddleware and also in apps sessions
  + By default, as session longs for two weeks
  + We can set the session age also, SESSION\_COOKIE\_AGE in seconds it takes
  + **To set Sessions**
* **class AddFavouriteView(View):  
   def post(self, request):  
   review\_id = request.POST["review\_id"]  
   request.session["favorite\_review"] = review\_id  
   return HttpResponseRedirect("/reviews/" + review\_id)**
  + **GET SESSIONS**
* **def get\_context\_data(self, \*\*kwargs):  
   context = super().get\_context\_data(\*\*kwargs)  
   loaded\_review = self.object  
   request = self.request  
   favorite\_id = request.session.get("favorite\_review")  
   context["is\_favorite"] = favorite\_id == str(loaded\_review.id)  
   return context**
* **SIGNALS**
  + Is used for after saving, before deleting and after deleting all signals
  + Import it from Django.db.models.signals
  + **After saving**
* **# sender will the model name, created is true false, whether its created/or not  
  def profileUpdated(sender, instance, created, \*\*kwargs):  
   print("Profile Updated")  
    
  post\_save.connect(profileUpdated, sender=Profile)**
* **post\_save.connect(profileUpdated, sender=Profile)**
* **Deleting User**
* **def deleteUser(sender, instance, \*\*kwargs):  
   print('Deleting User....')**
* **post\_delete.connect(deleteUser, sender=Profile)**
* **another way of calling this function**
  + **from Django.dispatch import receiver**
  + Then above function add
* **@receiver(post\_save, sender=Profile)**
* **def createProfile(sender, instance, created, \*\*kwargs):  
   if created:  
   user = instance  
   profile = Profile.objects.create(  
   user=user,  
   username=user.username,  
   email=user.email,  
   name=user.first\_name,  
   )  
    
  def deleteUser(sender, instance, \*\*kwargs):  
   user= instance.profile  
   user.delete()  
   print('Deleting User....')  
    
  post\_save.connect(createProfile, sender=User)  
  post\_delete.connect(deleteUser, sender=Profile)**
* **LOGIN/REGISTER/LOGOUT**
* **def loginUser(request):  
   if request.user.is\_authenticated:  
   return redirect('profiles')  
    
   if request.method == 'POST':  
   print(request.POST)  
   username = request.POST['username']  
   password = request.POST['password']  
   try:  
   user = User.objects.get(username=username)  
   except:  
   print('Username does not exists')  
    
   user = authenticate(request, username=username, password=password)  
     
   if user is not None:  
   login(request, user)  
   return redirect("profiles")  
   else:  
   print('Username or password is incorrect')  
   return render(request, 'users/login\_register.html')**
* **def logoutUser(request):  
   logout(request)  
   return redirect('login')**
* **DECORATORS**
  + We can add them if we want to protect some pages so that user cannot navigate to them until they are logged in
* **@login\_required(login\_url="login"),** add this over the functions u want to
* **REGISTER**
* **def registerUser(request):  
   page = 'register'  
  form = UserCreationForm()  
  if request.method == 'POST':  
   form = UserCreationForm(request.POST)  
   if form.is\_valid():  
   user = form.save(commit=False)  
   user.username = user.username.lower()  
   user.save()  
   messages.success(request, "User account was created")  
   login(request,user)**

**return redirect(‘profiles’)  
context = {'page': page, 'form': form}  
return render(request, 'users/login\_register.html', context)**

* **{% block content %}  
   {% if page == 'register' %}  
   <h1>Register User</h1>  
   <p>Already you have an account<a href="{% url 'login' %}">SignUp</a></p>  
   {% else %}  
   <form action="{% url 'login' %}" method="post">  
   {% csrf\_token %}  
   <input type="text" name="username" placeholder="Enter username"/>  
   <input type="password" name="password" placeholder="Enter password"/>  
   <input type="submit" value="Login"/>  
   </form>  
    
   <p>Don't have an account<a href="{% url 'register' %}">SignUp</a></p>  
   {% endif %}  
  {% endblock %}**
* **FLASH MESSAGES**
* **messages.error(request, 'Username or password is incorrect')**
* **{% if messages %}  
   <ul class="messages">  
   {% for message in messages %}  
   <li {% if message.tags %} class="{{ message.tags }}" {% endif %}>{{ message }}</li>  
   {% endfor %}  
   </ul>  
  {% endif %}**
* **ALERTS**
* **{% if messages %}  
   {% for message in messages %}  
   <div class="alert alert--{{message.tags}}">  
   <p class="alert\_\_message">{{message}}</p>  
   <button class="alert\_\_close">x</button>  
   </div>  
   {% endfor %}  
  {% endif %}**
* **SEARCH**
  + This feature is used if in our app there is any feature to search on a page
  + Here we need to keep in mind that the in html template where we use search there the name of the field has to be search\_query, if other then use that
  + We can create a new file utils.py in that we can create a function
* **def searchProjects(request):  
   search\_query = ''  
   if request.GET.get('search\_query'):  
   search\_query = request.GET.get('search\_query')  
    
   tags = Tag.objects.filter(name\_\_icontains=search\_query)  
    
   projects = Project.objects.distinct().filter(  
   Q(title\_\_icontains=search\_query) |  
   Q(description\_\_icontains=search\_query) |  
   Q(owner\_\_name\_\_icontains=search\_query) |  
   Q(tags\_\_in=tags)  
   )**

**return projects, search\_query**

here tags\_\_in is used as it is child object of Project.

* **def projects(request):  
   projects, search\_query = searchProjects(request)  
   context = {  
   'projects': projects,  
   'search\_query': search\_query  
   }  
   return render(request, 'projects/projects.html', context)**
* **PAGINATION**
  + Create a pagination.html file as follows
* **{% if queryset.has\_other\_pages %}  
   <div class="pagination">  
   <ul class="container">  
   {% if queryset.has\_previous %}  
   <li>  
   <a href="?page={{queryset.previous\_page\_number}}" class="btn page-link">&#10094; Prev</a>  
   </li>  
   {% endif %}  
    
   {% for page in custom\_range %}  
   {% if page == queryset.number %}  
   <li>  
   <a href="?page={{page}}" class="btn page-link btn-sub">{{page}}</a>  
   </li>  
   {% else %}  
   <li>  
   <a href="?page={{page}}" class="btn page-link">{{page}}</a>  
   </li>  
   {% endif %}  
   {% endfor %}  
    
   {% if queryset.has\_next %}  
   <li>  
   <a href="?page={{queryset.next\_page\_number}}" class="btn page-link">Next &#10095;</a>  
   </li>  
   {% endif %}  
   </ul>  
   </div>  
   {% endif %}**
* Then in utils file create a specific function to render
* **def paginateProjects(request, projects, results):  
   page = request.GET.get('page')  
   # no of records in one page  
   paginator = Paginator(projects, results)  
    
   try:  
   projects = paginator.page(page)  
   except PageNotAnInteger:  
   page = 1  
   projects = paginator.page(page)  
   except EmptyPage:  
   page = paginator.num\_pages  
   projects = paginator.page(page)  
    
   leftIndex = (int(page) - 4)  
    
   if leftIndex < 1:  
   leftIndex = 1  
    
   rightIndex = (int(page) + 5)  
    
   if rightIndex < paginator.num\_pages:  
   rightIndex = paginator.num\_pages + 1  
    
   custom\_range = range(leftIndex, rightIndex)  
    
   return custom\_range, projects**
* **def projects(request):  
   projects, search\_query = searchProjects(request)  
   # page number we are on  
   custom\_range, projects = paginateProjects(request, projects, 6)  
    
   context = {  
   'projects': projects,  
   'search\_query': search\_query,  
   'custom\_range': custom\_range  
   }  
   return render(request, 'projects/projects.html', context)**
* **DEPLOYMENT**
  + **Settings.py file**
    - In settings.py file, do debug=False, as in prod we can’t show the detailed error
  + **COLLECTING STATIC FILES**
    - For deployment, we will want all static files to be at one place so that our server can collect them and serve them to user
    - So, here we will collect all static files from different apps of your project in one folder
    - In settings.py file add setting as
    - **STATIC\_ROOT = BASE\_DIR / “staticfiles”**
    - **Then run command python manage.py collectstatic**
  + **SERVING STATIC FILES**
    - In urls.py file, add this
* **urlpatterns = [  
   path('admin/', admin.site.urls),  
   path("", include("blog.urls"))  
   ] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT) \  
   + static(settings.STATIC\_URL, document\_root=settings.STATIC\_ROOT)**
  + **MIGRATIONS**
    - Do make sure that you run migrations once
    - **Python manage.py makemigrations**
    - **Python manage.py migrate**
    - Also, make sure that u have the ran **the createsuperuser command**
  + **LOCKING IN DEPENDENCIES**
    - We can run a command
    - **Python -m pip freeze > requirements.txt**
    - Which will create a file for all dependencies of project but that may include other dependencies of another projects
    - Delete the requirements.txt file first, now
    - Instead, we create virtual env for this
    - **Python -m venv django\_my\_site(any name)**
    - Activate this venv
    - **Then Python -m pip freeze > requirements.txt**
  + **USING ENVIRONMENT VARIABLES**
    - This we are doing to specify where we are going to host our application
    - In settings.py file
* **ALLOWED\_HOSTS = [  
   getenv("APP\_HOST")  
  ]**
  + - **From os import getenv**
* **DEBUG = getenv("IS\_PRODUCTION", True), true is by default is that env variable doesn’t exists**
  + - We can do this for the SECRET\_KEY ALSO
  + **AWS DEPLOYMENT**
    - Search for elastic beanstalk
    - Create application
    - Any app name
    - Platform-Python
    - Then in your project create a folder as **.ebextensions**
    - In that folder, create a Django.config file
* **In that write,**
* **option\_settings:  
   aws:elasticbeanstalk:container:python  
   WSGIPath: my\_site.wsgi:application**
  + - here we are telling aws to create wsgi application in this location
    - then zip the folders u want, include dbsqlite also requirements, staticfiles etc
    - then in aws upload your code, choose file
    - Then, configure options, in that software edit
    - In Environment Variables, add the variables thatw e added in settings.py file
    - IS\_PRODUCTION=True
    - APP\_HOST,we don’t know yet, so keep it xxx later we will change
    - After this, u will see a link go to that, copy that domain
    - In left side in configuration, add the APP\_HOST, removing https
    - And is\_production change to False
    - then again go the url, u can see your app running
  + **CONNECTING TO POSTGRESQL**
    - In console, run **pip install psycopg2-binary**
    - Then again do, **Python -m pip freeze > requirements.txt**
    - In aws, search RDS, select PostgresQl, free tier
    - Specify a name, then choose username, password
    - Disable the autoscaling as we don’t need, if u want u can enable
    - Then also, select Public Access
    - VPC group, create new
    - Create database
* DATABASES = {  
   'default': {  
   'ENGINE': 'django.db.backends.postgresql’,  
   'NAME': 'postgres',  
   'USER':'',  
   'PASSWORD':'',  
   'HOST':'',  
   'PORT':''  
   }  
   }
  + - then run migrations, python manage.py migrate
    - createsuperuser, python manage.py createsuperuser
    - Now, we have to upload code again without dbsqlite filder
    - Then deploy our app
  + **SERVING STATIC FILES**
* In .ebextensions folder, add another file **static-files.config file**
* **option\_settings:  
   aws:elasticbeanstalk:environment:proxy:staticfiles:  
   /static: staticfiles**

**/files:uploads**

* here, /files is the media url that u specified in settings.py file and /static is the static url specified
* also, staticfiles and uploads are the static root and media root folder path
* zip them again and upload your code in aws
* the go in /admin, add the posts
* **SERVING FILES VIA S3**
  + IN AWS, s3 service is used to store files
  + In s3 console, create a bucket
  + Remove block all public access
  + In bucket, properties, edit static website hosting, enable it, save
  + Permissions, edit cross-origin, add configuration then
  + Then in bucket policy, edit, copy paste then save
  + Now go to another service IAM service,
  + Create group, in policy select s3 full access, create group
  + Then in that group, add users=> user-name=’’
  + Select AWS access type, Programmatic access
  + See video