**Notes on Python Django**

Notes from YouTube course: <https://www.youtube.com/watch?v=UmljXZIypDc>

**PART 1 – Getting Started:**

- First install the Django package:  
pip install django

- Next find the version of django:

python -m django --version

- Create a new Django project (in whatever directory you are currently in) called "django\_project" which will have a predefined structure and files:

django-admin startproject django\_project

- The manage.py file allows us to run command-line commands and is found at the base directory of the Django project.

- There is also a sub-directory called "django\_project" which contains predefined files: init.py, settings.py, urls.py, wsgi.py.

- To run the base website, use the following command, which will provide a URL to access the Django website on the localhost:   
python manage.py runserver

- By default, on the urls.py, there is a predefined admins route.

- To stop running your Django server, press CTRL C on the terminal.

- Django creates an initial website project. You can then add multiple apps to this website. So, you can create a Blog section to this Django website which would be its own app.

- You can also add a Django app to other Django projects, putting an app into multiple different websites.

**PART 2 – Applications and Routes:**

- To create an app called "Blog" within the initial Django project, navigate back to the base directory of the project where the manage.py exists and type:

python manage.py startapp blog

- The structure of this "Blog" app includes files like models.py, test.py, views.py and migrations.

- views.py = logic (as functions) for certain routes. It takes a web request and returns a web response. So, you can create a function "Home" in this views.py which manages the traffic from the Home page of your "Blog" app. The function would take a request argument and return a response.

- A URL pattern needs to be mapped to a function like the "Home" function within the views.py. To do this, create a new file called urls.py in your "Blog" directory. This urls.py is very similar to the urls.py in the main "django\_project" directory.

- One of the imports in your urls.py of your "Blog" app would be the views.py module from the same "Blog" app. So, we are mapping URLS to views within our "Blog" app.

- Likewise, we need to map a route between the main "django\_project" project and the "Blog" app's urls.py. To do this, import the "include" function from django.urls in the urls.py of the main "django\_project" and create a new path to the "Blog" app's urls.py.

- So, the main urls.py in "django\_project" has a path that maps to the "Blog" app's urls.py. Then, the "Blog" app's urls.py has a path that links the "Blog" app to a specific view.

- To manually check what is returned from a route, right-click "source" on a webpage within Google Chrome.

- So, to create a new route (i.e. a new webpage) to an app within a Django project, first add a new function to the views.py of your app. Next, create a mapping between a path in urls.py (in the app directory) and this new function in views.py. You do NOT need to add anything to the urls.py of the main "django\_project" project as that just connects to the "Blog" app's urls.py.

- To make the "Blog" app your homepage of your Django project, set the path as an empty string in the urls.py of the "Blog" app and in the main project's urls.py. This matches the empty path of both the main Django project and the "Blog" app.

**PART 3 - Templates**

- To return more complex HTML code from the views.py in the Blog app instead of small Http Responses => use templates!

- To use templates, first create a "templates" directory within the "Blog" app. By default, Django actually looks for a "templates" sub-directory in installed apps. So, as Django may be looking for other "templates" folders in this Django project, you need to create a specific sub-folder inside this "templates" folder called "blog".

- For each webpage in your "Blog" app, create an html file inside the "templates" folder.

- Next, you need to add the actual "Blog" app into the list of installed apps so Django looks for the templates directory. To do this, you need to copy the name of the class which inherits from the AppConfig inside the apps.py and then paste this path into the INSTALLED\_APPS list of the project's settings.py file.

- Also, point the views of the Blog app to the templates by using the render function (from the Shortcuts module) and passing the template you want to render as arguments to the render function.

- To display posts (which will be updated) on the templates, you first add the posts to the views of the Blog app and then pass these posts to the templates. On the views, the posts would be a list of dictionaries, where each dictionary is a single post. Next, to pass these posts to a template, say the home template, pass the optional "context" argument to the render function in the home function of the views. Send these posts to a dictionary within this "home" function and the "home" function renders these posts to the home page, with the value of this dictionary being the posts passed into the function. The key of this "context" dictionary will be available in the template.

- You can now access posts within the template using the name of the dictionary defined in the views by referencing the key name of the dictionary (and the value of the dictionary is a list of dictionaries).

- Next, on the template for the home page, need to loop through the posts passed to this page. You can loop over the posts with a for loop using {% for post in posts %}, where "posts" is the key of the "posts" dictionary defined in the views. To end the for loop, type {% endfor %}. Furthermore, to access a variable like "Title" in one of the variables in the post dictionary, use

{{ post.title }} .

- To set a title to a webpage in Django, pass a dictionary variable as the context argument of the render function for that webpage in the views.py. The key of this dictionary variable is used in the respective template file. If no title is passed, the Django webpage sets the title with the default "Dango <name of the app>".

- If multiple templates share similar code => use template inheritance. So, in the templates folder of your app, create a new html file "base.html". Copy all repeated code across templates into this "base.html" file. In the place(s) of this "base.html" which are unique to other templates, create a BLOCK. A BLOCK is a section of a template which child templates can override. To create a block, use {% block content %} and {% endblock %} in the "base.html". Note, the name of this BLOCK is "content". Next, on top of the unique templates, first inherit the base template "base.html" with the code {% extends "blog/base.html" %}. Now, since the BLOCK in the base template is called "content", enclose the unique information in the child template with {% block content %) and {% endblock content %}.

- To incorporate Bootstrap into your Django app, go to the Bootstrap Starter Template (<https://getbootstrap.com/docs/4.0/getting-started/introduction/#starter-template>) and copy the meta tags, CSS and Javascript from this template into your base.html. Paste the metatags and CSS code into the head of the base.html and paste the JavaScript code at the end of the body tag of the base.html. Wrap the content block in the body with the div tag of class = "container". (a CSS class which gives the webpage padding and spacing).

- To add a navigation bar, add a standard code snippet to the head of the base.html.

- In Django, static files like CSS and JavaScript need to be placed in a static directory within your app. So, create a folder "static" in the blog app and a subfolder inside this "static" folder called "blog". If you have a CSS file in this subfolder called main.css and you want it to be used by base.html, you must load static files. This is done by typing {% load static %} above the <!DOCTYPE html> in the base.html. So, to specifically load the main.css into the base.html, add a link tag to the head of base.html, with href of this tag equal to "{% static 'blog/main.css' %}", where 'blog/main.css' is the location of the css file.

- To make each individual blog post appear more separate, add more CSS and html code inside the for loop for the posts in the home.html.

- If you have links on one of your templates, like your home.html, and you are currently using hardcoded values for the respective href (Note, href with value = # is a dead link) => then, use the Django URL tag instead. Hardcoded routes are not good and one should use the Django URL tag to get the absolute URL path for a given URL pattern. The URL patterns are in the urls.py of the Blog app. So, instead of setting the href to a hardcoded value, set it to {% url <name of the url pattern> %}. This is why pattern names should be unique in urls.py. Using the Django URL tag prevents one from having to change URLs in multiple locations.

**PART 4 – Admin Page**

- For every Django project, there is a default Admin interface. To access this Admin page of your project, type "/admin" at the end of your localhost home page URL on your web browser. What appears is an Admin login.

- To log in to the Django Admin page, you first need to create an Admin user and password. However, to create such an Admin user and password, you first need to create a database for your Django project.

- So, to create a database on a Django project with default tables, type the following to first prepare the database to be updated:

python manage.py makemigrations

(It will say "no changes detected" unless you wrote your own tables or models)

- Next, to actually apply the migrations, type:

python manage.py migrate

- Once the database is created for your Django project, you can create an Admin user by typing:

python manage.py createsuperuser.

This command will prompt you to enter your username, email address and password. You can use these credentials to sign into the Admin page of your Django project.

- Note, Django does not store a user's "raw" password but rather a "hashed" password (for security) which can be viewed on the Django Admin page.

- You can also add users on the Django Admin page. The default permissions of such a new user is "Active" (NOT "Staff status" or "Superuser status").

**PART 5 – Database and Migrations**

- To work with databases, Django has its own ORM (Object Relational mapper) which allows one to access Django databases in an OOP-way. It allows a user to have different databases for Dev and Prod as there would be different databases defined in the settings but the code for querying the databases would be the same.

- Furthermore, with the Django ORM, databases can be represented as classes called models. So, by default, within the "Blog" app, there is a models.py file and this will define what is actually saved to the database.

- So, for the "Blog" app, to create a "Post" model for the posts stored on the database. This will be a class that will inherit from the Django models class, by default (from django.db import models).

- To create the "Post" class (or post model) in this models.py, first type "class Post(models.Model):". Each class/model will be its own table in the database.

- Next, create attributes for this class/model. Each attribute would be a different field in the table of the database. You can specify arguments for these attributes which set restraints on the respective fields.

- You can import "from django.utils import timezone" which takes our timezone settings into consideration when setting times/dates.

- Note, Django has default tables called "auth.models" which has a list of users and should be imported into models.py if you want to set attributes like authors of posts (from django,contrib.auth.models import User). Here, User is considered a model in Django (equivalent to a table in SQL).

- If there are ANY changes relating to your Django database, you need to rerun migrations to create the files in the "migrations" directory of the "Blog" app:

python manage.py makemigrations

- To actually RUN the files in this "migrations" directory, and these migration changes take effect on the actual database, type:

python manage.py migrate

- To view the actual SQL code with the app name "blog" and migration number (0001):

python manage.py sqlmigrate blog 0001

- Migrations are useful in Django as it allows us to make changes to our Django database, even after the database has been created and data is in the database. Otherwise, complex SQL queries would be needed to update the database structure so it didn't mess with the current data in it.

- To query the Django database with these models, the Django ORM allows us to do this with the classes also. So, you can work with these models interactively using the Django-Python shell by typing:

python manage.py shell

- You can run Python code in this Django-Python shell but also work with our Django objects. For example, you can import both your Post model and user model from the "Blog" app on this shell.

- So, to import the Post model from the "Blog" app, type the following in the Django-Python shell:

from blog.models import Post

- To query the user table for the list of ALL users, type:

User.objects.all()

- To query the user table for the FIRST user, type:

User,objects.first()

- To filter the queried results by a field, say the "username" field, then type:

User,objects.filter(username='POR160893')

- You can store the result of a filter query to a variable. For example:

user = User.objects.first()

- You can also look at the attributes of this "user" variable like the 'id" attribute or "primary key" attribute by typing the following, respectively:

[user.id](http://user.id/)

[user.pk](http://user.pk/)

- To get the user with "id" equal to1 and assign the result to a variable called "user", then type:

user = User.objects.get(id = 1)

- To create a new post, like "post\_1", with field values on the Django-Python shell and using the variable "user", then type:

post\_1 = Post(title = 'Blog 1', content = 'First Post Content!', author = user)

- To now save this new post "post\_1" to the actual Django database, you would need to type:

post\_1.save()

- To actually see the contents of a Post in the Django-Python shell, create a dunder "str" method, i.e. def \_\_str\_\_, in the models.py file of your "Blog" app. Set the return of this method to what you want printed out for the Post such as returning the Post's title, i.e. return self.title.

- Say a user is mentioned as an author on a model "Post" and you want the email address for this user. Now, when we created this model "Post", we did not set one of the attributes of the model to be "email\_address". BUT, "email" is one of the attributes in the default model "User". So, you can access another model (in this case "User") by using the attribute with the ForeignKey, let's say it is the "author" key, in your "Post" model. Hence, to access the email attribute from the "User" model by accessing it with the "author" attribute of your "Post" model, you would type:

Post.author.email

- To get a list of all attributes in the default Django model "User"(same as getting all fields of a SQL table), then type:

[[f.name](http://f.name/) **for** f **in** User.\_meta.get\_fields()]

- In this Blog, a User can have multiple Posts but a Post can have only 1 User related to it. Hence, this is a 1-to-many relationship. To get the User related to a given Post, you would type the following as "author" is the foreign key related to User and has the name of the User anyway:

post.author

Conversely, to get all the Posts related to a given user, it is a reverse lookup and you need to type the following:

user.post\_set.all()

- To create another post from the same user from the previous example, you can simply write:

user.post\_set.create(title = 'Blog 3', content = 'Third Post Content!')

You do not need to add the "author" argument as Django knows already it is the same person from the "user" variable.

- In Django, dummy data can be added to the views.py (in the "Blog" app) or real data can be added to the actual database. So, you can get the data from the database and then pass this data to the views. We could have dummy data defined in this actual views.py and have it passed to the context dictionary of the home method. However, alternatively, you can run a query on Post mode and pass this data to the context dictionary. If you used the 2nd way, you would need to import the Post model into the views.py.

- If you access a date with a Django model (from the actual database), then, by default, the date will be in a long format which includes the actual time. To change to as needs be, you can use the date filter on whatever template this pertains to. Example:

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

- To see a newly created model (like the "Post" model) on the Admin interface, you need to register the model on the Admin page. To register this "Post" model to the Admin page, navigate to the admin.py file in your "Blog" app. First, import the "Post" model into this admin.py (from .models import Post) and then, register this "Post" model with the Admin site (admin.site.register(Post)).  
You can now update each blog post via the Admin page.  
=> You can change Django models and objects from the backend Admin page.

**PART 6 – Forms**

- Next, need to look. at creating front-end Forms on Django. To do this, create a new app to handle all the user logic in your Django project. So, to create a new app called "users", type:

python manage.py startapp users

- As always, when we create a new app within a Django project, add this new app to the installed apps list in settings.py (in the main project folder):  
  
INSTALLED\_APPS = [  
 'users.apps.UsersConfig',…  
  
This is the same UserConfig class defined in apps.py on the User app.

- Next, move onto the logic for the register route in the views.py (in the users folder). Here, write a function that creates a registration form that will be passed to the template that will be created for this view.

def register(request):

- To create such a register form, you can import a Python class which generates a HTML form in the views.py (on the User app):

from django.contrib.auth.forms import UserCreationForm

- To use this imported form in your register function, create an instance of this form inside your register function.  
  
def register(request):  
 form = UserCreationForm()

- To actually use this instance of a form, render it on a template (as the return of the register function). The actual form would be the context passed into this template and you will be able to access this form from within the template.  
  
  
def register(request):  
 form = UserCreationForm()  
 return render(request, 'users/register.html', {'form': form})

- Next, create the template, register.html, which uses this form. Like with any template, you will be extending the base.html and then filling in the content block. You can reference templates from other apps within the same Django project also.

- Within the actual template, you would first need to use a <form> HTML tag. Inside this form tag, you need to add a CSRF (Cross-Site Request Forgery) token which will protect our form from certain attacks. The Django form will NOT work without this CSRF token.

{% csrf\_token %}

- Once the template, register.html, is created, you need to add a URL pattern that uses the register view in order to access this webpage via a browser. To do this, you need to map the "Users" app to the main Django project. Previously, for the "Blog" app, we created a urls.py within the "Blog" app. You can also do this for the "Users" app. However, alternatively, you can just import the register view (in the "User" app) directly into the urls.py file of the main Django project and create a url pattern for it here.

- So, import the register view into the main urls.py with:

from users import views as user\_views

- Next, create the URL pattern which uses the imported register views within the same urls.py:

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

- So, to summarize, for the "Blog" app, the views were imported into the urls.py within the actual "Blog" app directory and then passed into the main project's urls.py with the include() function. However, for the "Users" app, the views were imported directly into the main project's URL and the respective URL pattern was created here too. For the "Blog" app, the URL patterns were created into the ulrs.py in this directory of the actual app.

- Once you have created a form in your Django app, you can render the form with paragraph tags by using the as\_p method on the form variable in the register.html:

{{ form.as\_p }}

- Next, need to add actual functionality to this register form. So, at the moment, when someone adds details onto this form, a user is NOT created on the Admin page. At the moment, any request passed to the register route just creates a blank form. What we need is to actually POST a request on the register route with the form information (as it was POST request specified in the register.html but we didn't specify where to post the form data).

- So, need an IF statement in the register method of the views.py (in the Users app) if it receives a POST request and needs to validate the form data. There will also be an ELSE statement for GET requests and this will return an empty form:

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

- Next, at the moment, the POST data could be associated with anything, not specifically for a form. So, the POST data needs to be validated. To do this, we need to add another IF statement within the previous IF statement to validate the form data with function form.cleaned\_data. This function gives us the validated form data in the form of a dictionary:

if request.method == 'POST':  
 form = UserCreationForm(request.POST)  
 if form.is\_valid():  
 username = form.cleaned\_data.get('username')

- In addition, we will add a flash message which will show that we have received valid form data. Fash messages send 1-time alerts to a template and will disappear on the next request. To do this, first import messages:

from django.contrib import messagesThen, specify what type of message you want to add:

messages.success(request, f'Account created for {username}!')

- Finally, redirect the user to a different webpage once the registration form has been completed successfully. In this scenario, we will redirect the user back to the home page. To do this redirecting, we need to import the redirect function:

from django.shortcuts import render, redirect

- Subsequently, we will simply return a redirect back to the home page after the flash message has disappeared. Note 'blog-home' is the name of the URL pattern for the home page:

return redirect('blog-home')

- Finally, we need to update the template, (base.html as flash messages can come up on ANY template pages then), to show these flash messages. So, a good place to display such messages would be above our content block. Add an IF code block first then relating to the IF message. Within this IF code block, add a FOR loop code block which will loop over the flash messages. Finally, inside this FOR loop, print out the flash message.

(Note, the message types in Bootstrap are the same as the message types in Django. You can access it within Django by message.tags):

{% if messages %}  
 {% for message in messages %}  
 <div class="alert alert-{{ message.tags }}">  
 {{ message }}  
 </div>  
 {% endfor %}  
{% endif %}

- Next, to save a new username after a registration form has been validated, simply add the form.save() after a form has been checked that it is valid. Now this new username can be seen on the Admin page:

if form.is\_valid():  
 form.save()

- To add a new field like an email address field on the registration form, you need to create a new form which will inherit from the original user creation form. So, you need to create a file, forms.py, in the users apps that will house this new form. In this new file, you need to import Django forms, the User model and the user creation form. Create a new class for the new form which will inherit from the user creation form but also have an additional email field. Within this new class, create a subclass called "Meta" which will specify what model this new form will interact with. In this case, the model would be "User" and specify what are the fields to be shown on this new form.

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']

- Now, in the views.py of the Users app, you can now use this new User Register form instead of the User Custom form (making sure you import this User Register form into this file).

- To add better styling to this form, use Crispy forms which allow us to add tags into our templates and style our form in a Bootstrap way. Firstly, add Crispy Forms into the Django's project's setting.py:

INSTALLED\_APPS = [  
  
 'crispy\_forms',

Also, tell Crispy what CSS framework you want to use by specifying Bootstrap 4 at the end of the same settings.py:

CRISPY\_TEMPLATE\_PACK = 'bootstrap4'

Finally, load these Crispy form tags into the register.html template by adding the code:

{% load crispy\_forms\_tags %}  
{% block content %}

This code allows us to use the Crispy filter on the form variable within this template;

{{ form|crispy }}

**PART 7 – Login and Logout Views**

-For login and logout, use Django’s in-built views by importing them in the project’s urls.py:  
from django.contrib.auth import views as auth\_views  
- Next, create URL pattern for these imported views and using these login and logout views, respectively (Note, these are examples of Class-based views):  
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'),  
  
  
  
  
  
  
  
  
  
  
  
  
  
- As the URL paths above are looking for login.html and logout.html templates in the User app, create them. The login.html template will be very similar to the register.html template. However, you need to change the legend so that “Log In” is used:  
<fieldset class="form-group">  
 <legend class="border-bottom mb-4">Log In</legend>  
 {{ form|crispy }}  
</fieldset>  
and create a “Log In” button”  
<div class="form-group">  
 <button class = "btn btn-outline-info" type = "submit">Log In</button>  
</div>  
- If a user currently does not already have an account, create a link on the login page that would navigate them to the register page by using a URL tag:  
<small class="text-muted">  
 Need An Account? <a class="ml-2" href="{% url 'register' %}">Sign Up Now</a>  
</small>  
- Similarly, on the register.html template, add the href to the login.html:  
<small class="text-muted">  
 Already Have An Account? <a class="ml-2" href="{% url 'login' %}">Sign In</a>  
</small>  
- By default, when one uses Django’s built-in login module, it redirects the user to <http://127.0.0.1:8000/accounts/profile/> once logged in. This page is NOT usually created though. To redirect Django to another webpage once a user has logged in, add the following code to the project’s Settings.py:  
LOGIN\_REDIRECT\_URL = 'blog-home'  
The above code now redirects the user to the blog’s home page once logged in (This URL is defined in the blog’s url.py and is called “blog-home”).  
- Once a user is registered, Django should redirect them to the login page. So, in the views.py of the Users app, change the register function by redirecting the user to the login page once they are registered:  
return redirect('login')  
- Similarly, the default logout view in Django’s built-in login/logout module looks like the admin page with the heading “Logged Out” and a link to login. However, you want the website to NOT expose the admin page to users when they have logged out. So, you must create a logout template in the Users app (as already referenced on the “logout” URL pattern on url.py in the project app).  
- The logout view is already looking for the logout.html as it was defined on the url.py of the project app:  
path('logout/', auth\_views.LogoutView.as\_view(template\_name='users/logout.html'), name='logout'),  
  
- Finally, create the logout.html template in the users app. In the template, add the option for the user to login again:  
<div class="border-top pt-3">  
 <small class="text-muted">  
 <a href="{% url 'login' %}">Log In Again</a>  
 </small>  
</div>  
- Next, change the navigation bar of the base template so that it changes depending on whether a user is logged in or logged out. This navigation bar is in the base template in the blog app. Need to add a condition that checked if a user is currently logged in or not. To check whether a user is logged in or not, use the is\_authenticated attribute of Django’s user variable. If a user is logged in, they will see the options to create a post, profile and logout. Otherwise, if a user is logged out, they will see options to login or register:  
<!-- Navbar Right Side -->  
<div class="navbar-nav">  
 {% if user.is\_authenticated %}  
 <a class="nav-item nav-link" href="{% url 'post-create' %}">New Post</a>  
 <a class="nav-item nav-link" 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 %}  
- Finally, need to add restrictions to some routes so that they are accessible only if a user is logged in.

-