**WHAT IS DJANGO?**

Django is a python package that uses a server/client communication protocol to dynamically update a given webpage. Django offers a way to manipulate both the frontend end backend to create fully dynamic web applications that can take the user input, perform a computation, and spit out a graphic in accordance.

**STARTING A NEW APP**

***steps***

* run : python manage.py startapp newapp to create a new application in your project under the name newapp
* Navigate to the **settings.py** file and add the name of the app to the **INSTALLED\_APPS** list
* Navigate to **urls.py** file and add the directory of the app to the path
* Go to the new app directory and create a **urls.py** file which will set up the paths that are accessed from it.
  + Use variable urlpatterns to specify a list of path(“directory”, views.index, name = “index”) where:
    - directory is the extra directory path used to get to this page
    - views is a python file that contains function index which is responsible for generating the page
    - name is a unique name to this directory that can be referenced in code via {% url name %}
* As mentioned above, views.index is a call to a function that points us to the right page. Some options are:
  + render(request, “dir/file.html”, {}) where:
    - dir is the directory inside the **‘templates’** folder that contains the html file that is to generate this page.
    - {} is a python dictionary which holds the context argument which specifies which variables from the template you would want python to have access to.
  + HttpResponse(“text”) which will simply place the text on the page.
* For static parameters, such as style of the page, place the files into **‘static’** folder
  + Inside the html file that uses these style, place {% load static %} at the top
  + Place in <head> - <link href {% static ‘/newyear/styles.css’ %} rel = ‘stylesheet’> to place a css style sheet.

**GENERAL TIPS**

127.0.0.1 is an IP address associated with your computer – like ‘home’

***project level folder structure***

**urls.py** (in the project) – contains a list of apps included in the project that a user might visit. This serves as a table of contents for all available apps.

**settings.py** (in the project) – contains a list of all directories associated with the apps in the project

***app level folder structure***

[**/templates**](https://docs.djangoproject.com/en/4.1/topics/templates/#:~:text=A%20Django%20template%20is%20a,is%20rendered%20with%20a%20context.)– contains the static templates of the webpage

**views.py** – contain the ‘views’ which are pages a user might want to see.

**urls.py** (in the app) – contains the list of urlpatterns that a user might visit while using a given website

* **urlpatterns –** list of urls that a user might visit when on the website. The urls are listed using **path()** functions, to which we pass:

1. A string representing the URL path
2. Function from utls.py which we want to call when we visit that given url
3. And an optional name for this path

**“”** – empty route

! when we request a page:

1. First the program will look at urls.py to see if it’s in there.
2. If it is, it will follow to the views.py to apply the associated function
3. If the function references a template, it will go to it, otherwise it may just perform a specific function without ever needing to go to a set page.

method POST : means we are submitting data

method GET : means we are getting data

**TERMINAL COMMANDS**

**django-admin startnewproject projectname**

* will start a new project with the name projectname in the directory that you are located in

**python manage.py runserver**

* will run the server on your local machine and you can visit the ip address to look at the webpage generated by the project

**python manage.py startapp newapp**

* will create a new application in your project under the name newapp

**python manage.py migrate**

* this will apply all the migrations. For a model database migrations, will create sql file for the database.

**python manage.py makemigrations**

* will create the 0001\_initial.py file that will allow for database migrations. This file will have instructions to the database of how to manipulate the models in models.py.

**python manage.py shell**

* will allow us to run shell commands that will be executed in the application

**python manage.py test** will run the tests in test.py file

**python manage.py createsuperuser** will create a user with admin privileges

**GENERAL DJANGO SPECIFIC COMMANDS AND NOTATION**

**{{ variable }}** : will embed a **python** variable inside the HTTP code calling back to one defined inside the django project that is passed over to the rendered HTTP page.

**{% %}** : will run some piece of code, s.a. a logic statement. Some example are:

* {% if var %} : where var is a variable passed to the page
* {% else %}
* {% endif %}

**{% block body%} {% endblock %}** can be used to specify a section where you’d like to insert a block piece of code defined in another .html file.

* Inside layout.html file (general layout) we have {% block body %} {% endblock %} which sandwiches a unique piece of code that will be inherit the rest of the file.
* Inside the file inheriting layout.html we have {% extends “/layout.html” %} which signals where we are inheriting from. {% block body %} {% endblock %} will sandwich the code which we actually want to be plugged into the ‘boiler plate’ code inside the layout.html

Linking files:

* When we embed urls on a page, we can use the **href = “/app/webpage”** structure, or we can simply use **href = “%url ‘webpagename’%,** in which case Django will go in our urls.py file and look for that name in the urlpatterns. This is useful because we can reference a route with a name rather than its path.
* One has to be careful in using this approach to avoid namespace collisions. To avoid unpleasant surpises, such as linking a webpage from a separate app, use the apps name when linking, like this: **href = “%url ‘app:webpagename’%.**
* The above is true when linking in .html file. To link in python files, we use HttpResponseRedirect(reverse(“tasks:index”)) notations.

**{% csrf\_token %}** when placed in form will represent a token to ensure that data can securely be transferred to our server. This a solution to 403 error.

**FUNCTIONS**

**reverse()** – takes a views name as input and returns the associated url path from views file. Inputs to this file can be passed as args = (input1,input2,..) where the structure is essentially that of a tuple.

**logout(request)** will log the user out

**user = authenticate(request, username,password)** will check if the user under the username with password exists in the system

**login(request,user)** will log the user in

**DJANGO / SQL**

**Creating a database**

* **Create a class in models.py associated with the database structure**
* e.g. origin = models.CharField(max\_length = 64)
* **python manage.py makemigrations** will look for changes made in model.py
* **python manage.py migrate** will apply the migrations which will allow us to access data entries in the .sql file
* **python manage.py shell** to enter python shell
* **from flights.models import flight** (where flight is the name of the class inside models.py)
* **f = Flight(origin=”New York”, destination = “London”, durations = 415)** will create an object
* **f.save()** will save the object
* **flight.objects.all()** will return all the objects in the database

**Interacting with a database**

We can access entries in the table by typing **f.duration, f.destination,** etc.

**f.delete()** will delete the entry

**models.ForeignKey** : allows to reference a data entry via a foreign class key.

* Argument 1 : class name that provides key reference
* Argument 2: on\_delete, which provides the rule for how to deal with an entry to this model if they key reference is deleted
* Argument 3: related\_name, using this name you can find all the model entries which have a matching related\_name

[**Forms.Form:**](https://docs.djangoproject.com/en/4.1/topics/forms/)

* Provides a way to generate a form that is easily editable and offers client-side validation, i.e. the webpage does the necessary checks to make sure the submission is correct.

**Referencing a field within object** given a model object A, to reference all of the entries in a given field write A.fieldname.all().

Other/Testing

PYTHON

***functions***

**assert** is a way to check if a statement is True. If it is True, the program will continue. If it is False, assert an assertion error specifying where the assertion failed.

***libraries***

**unittest** library will allow unit testing in python.

**selenium** can simulate the webpage interactivity which aids in performing web browser based tests

* [**Locating elements**](https://selenium-python.readthedocs.io/locating-elements.html)allows the user to add a test to what happens when the user interacts with them

**CI/CD** – refers to continuous integration continuous delivery of code by a software team

* **CI** : frequent merges to main branch + automated unit testing
  + ensures no two or more branches become irreparably different
  + ensures that no branches get merged unless they pass the tests
* **CD** : short release schedules
  + If something goes wrong you know quicker what it is
  + Releasing features is more useful to the users
* **GitHub Actions**
  + A way to enable smooth **CI/CD**

YAML

**Structured like a dictionary:**

Key1 : value1

Key2 : value2

Key3 :

* Item1
* Item2
* Item3

Other/Docker

DOCKER

**Containerization** software that allows the code to be run in a container that has its own configurations. This allows multiple developers to develop under the same server conditions.

* These are similar to virtual machines but are lighter. Unlike VMs Docker containers use base OS but have a Docker layer separating it.
* **Dockerfile** is a file in the main directory which specifies the instructions to setup the container, with all the necessary dependencies installed.
* **Docker-compose.yml**

POSTGRESQL

Database system use by many applications.