# Important Prelogue

In the below documentation, there are lot of examples in code and in the code there are lot of comments which explains what is happening in the code, which will give the better understanding possible, please go through it if possible

# What is Django

* A python-based framework for creating web applications
* This framework provides rules, structures and functionality that allow us to use python code for backend application development.
* Django can interact with our web application and send information to the user of web application
* Key features or pros:
  + Versatile with python
  + Scalable
  + Updated frequently and secure
  + Supports fast development
  + Many common features included
    - Administration
    - Authentication
    - Database connectivity
    - Security
  + Django is often heard as “Batteries included” or “fully loaded”, that means many of the common web application features in the form of readily available packages/python modules
* Cons of Django
  + Requires steep learning curve
  + Has extra unwanted features (an overkill for simple websites)
  + Built with python (needs python as pre-requisite)
* Who uses Django
  + Instagram, spotify, Youtube, Eventbrite, Dropbox, Pintrest and many more
* Web Application:
  + A web application at a high level simply transfers data to and fro between end user and database (server) with some interaction
* MVT structure:
  + Model: which connects to the database
  + View: connection between model and template
  + Template: connects to the user browser
  + Features that allow us to follow ORM
    - Object Relational Mapper (ORM)
    - Models (python models written with Django)
    - URLs and views
    - Templates

## Flow of MVT

The flow of data happens like template <-> view <-> Model (when you move like template -> view -> model then views need to store data or retrieve data from database, to do that views must be aware of the structure and types, to achieve that functionality django has created models, each model is a representation of a table in database, and all the operations you perform with sql queries will be done by django, so that you just have to interconnect views with models so that all crud operations that are to be done on databases will be done by django using models)

The flow is bi directional (not unidirectional as shown below)

Diagram

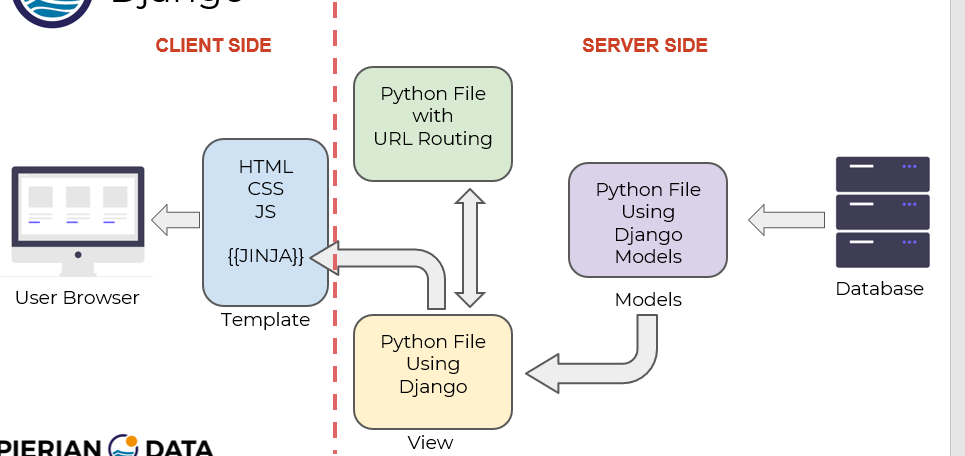
Description automatically generated

models (models.py file – a subset of django framework, which can have more than one class and each class indirectly is related to table on one on one basis, i.e., each class has an individual table) are python representation of database and database tables

The models are used in views (views.py file – using this we can fetch data from database, manipulate or process or analyze or do whatever we want to do with the python packages (knowledge) we have.), which are used to inject information directly in to template, on a short note you can think view as analogous to a particular page on your website, the views are also used to communicate with URL routing (URLs.py) for routing information of URL’s i.e., which view goes to which URL route, ex: a particular view named profile should be routed to an URL [www.abc.com/user-profile](http://www.abc.com/user-profile)

The template (HTML file) is what you get as input to browser which in turn will be displayed on user screen. This takes input from the views.py and data from (a python file) views.py is injected in to template using JINJA, this is called **jinja templating.**

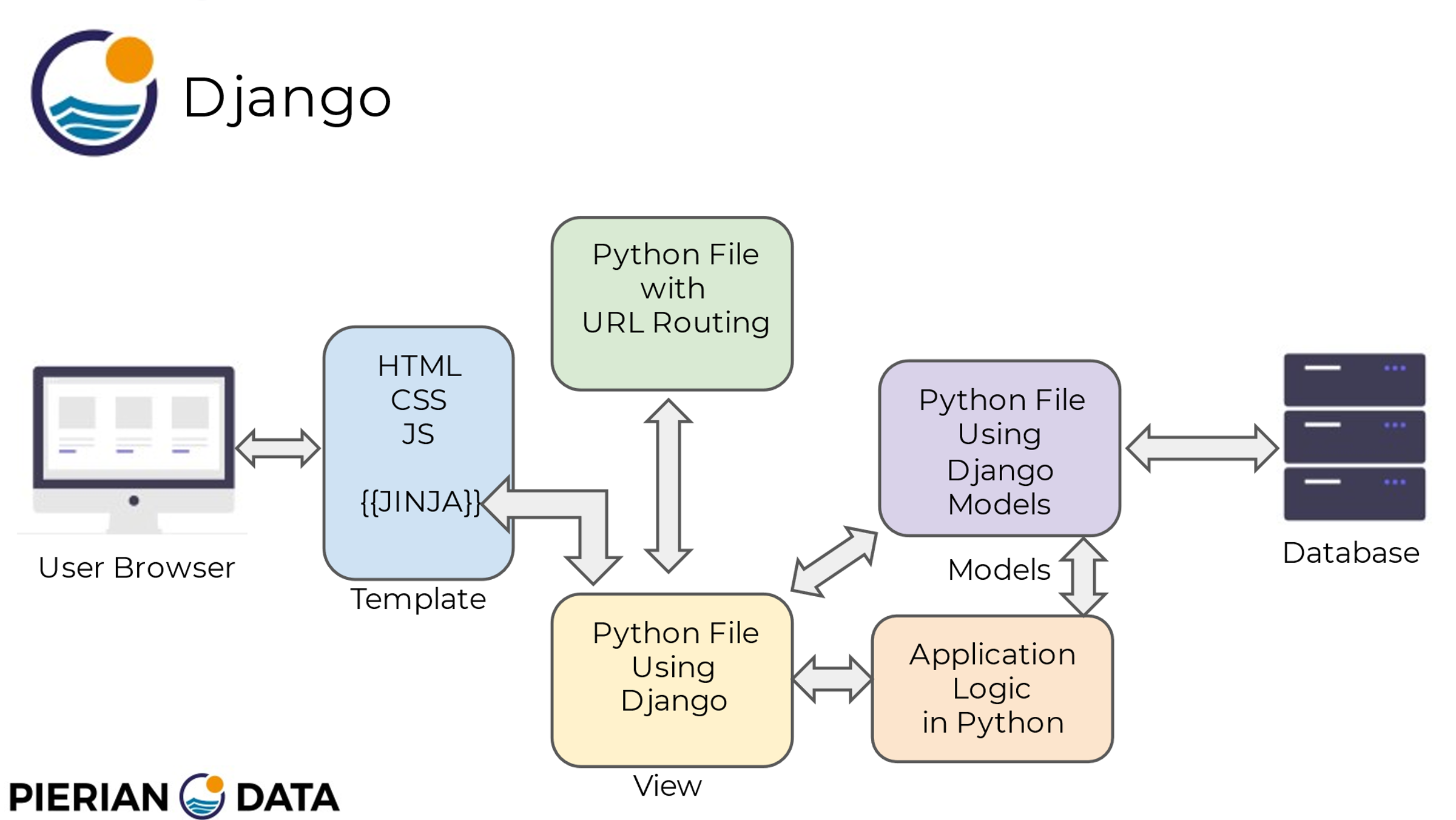
### Client side and server-side partition of MVT

****

### Two-way flow implemented in MVT

Here in the below diagram, you can see the new component Application logic in python. This component will let you use custom python code to do all the code implementations you want to do, to meet the requirements, get met

Ex: you might want to replace the middle name as a single alphabet (Pratap sai Prahalad -> Pratap s Prahalad) you can use an python script to do that and then send that data to view.py



## Drawbacks of Django

* Heavily reliant on the idea of Models (models.py), which is a python/django representation of table in the database. This makes it easy to work with querying data, but **does add the requirement of understanding models and setting them up for views**

## Commands used in Django

### django-admin startproject <project\_name>

* + django-admin tool comes in automatically when you install django
  + project in terms of django means a web application or a website (as a whole) which has the ability be spinned up and running
  + the project name used here should reflect the name that you want to give to the project
  + this command will (upon executing) will create a set of subdirectories and files which are needed to get started with a project structure and later do customized changes that you want to implement as per need
  + sample tree (project template) structure you get is like this:

└── demo\_proj

├── demo\_proj

│ ├── \_\_init\_\_.py

│ ├── asgi.py

│ ├── settings.py

│ ├── URLs.py

│ └── wsgi.py

└── manage.py

* settings.py – has settings that are going to be used along the entire project
* asgi.py – this acts as an entrypoint for asgi compatible web servers
* wsgi.py – entrypoint for WSGI compatible web servers
* note: whenever you are running a command of executing manage.py and passing some command line arguments **you need to be in the folder where you are having manage.py do not pass relative path from some other remote location as some commands- startapp command below for example will create app folder in the directory where you ran the command, instead it should create app folder inside the project app where the manage.py script resides**

### python manage.py runserver

* + note: manage.py (a python file) accepts different command line arguments (to start a server or to create an application and so on and so forth) used to perform different set of steps like running a server of the project we created, or creating a application to cover a specific functionality more details [here](#_why_django_apps)
  + this above command is going to launch the website i.e., spins up a (template) project instantly (most possibly this will run at a default address localhost:8000)
  + if you want to run your project at a different port give **python manage.py runserver <new port>** 
    - **python manage.py runserver 8181**

### python manage.py startapp < app name >

* + this command will create a new app with which you can cover a specific functionality, more details [here](#_why_django_apps)

### python manage.py collectstatic

this command will collect all static files

### python manage.py makemigrations

* Whenever you make an **update to a model or create a model or delete a model** (i.e., if you make any changes to models.py file) you need to run **python manage.py makemigrations <name of the app>** 
  + Here if you don’t pass the name of the app it will run makemigrations command for list of INSTALLED\_APPS in settings.py
* Generally makemigrations will generate SQL commands for mentioned apps

### python manage.py migrate

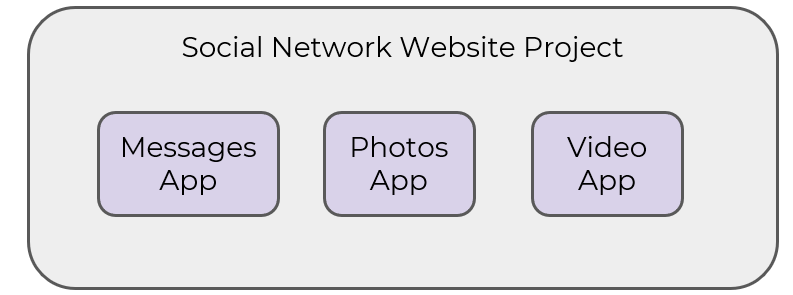
this command will run the generated migrate

## difference between Django project and Django apps

the django project is nothing but the overall website, django apps (don’t get confused with webapps because a webapp means complete website or the mobile application on the web) are subcomponents of that website i.e., django projects have separated components called **apps which (at an individual level) should cover a particular key functionality.**

### why django apps

* it often becomes much easier to organize code through these apps
* each django app should cover a key functionality for your website
* django apps are suggested when django is being used in an organization (not some personal project or a project with one person) because it will be much clearer when other or yourself (see code after a considerable time) to understand which components are referring to which key functionalities
* if you start a social networking website which has image sharing, video sharing, and texting features, it is suggested to have 3 different apps for 3 different features because, in future if users perform a lot of video sharing then scaling up the video app alone seems easy than looking at all the features under a single code base and then trying to scale it up



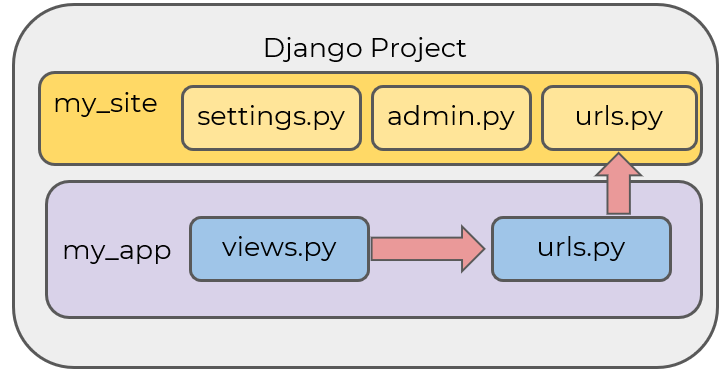
* a

### single app vs multiple apps

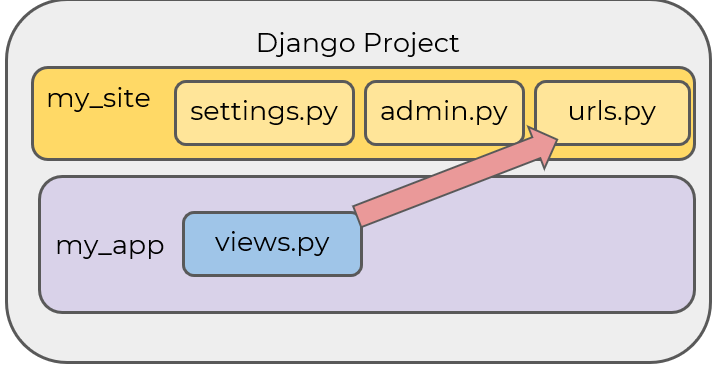
single app is enough if the functionality is limited or if it is just a single person managing the application.

Multiple apps are useful when there are lot of functionalities or multiple people working on multiple functionalities because it will be much clearer when other or yourself (see code after a considerable time) to understand which components are referring to which key functionalities

#### How things are handled in multiple apps:

When the project is very large and there are very large number of apps, then it is not ideal to have views.py (each app can have separate views.py) at an app level and all URLs (each view has a particular URL and all such URL’s are mentioned in URLs.py file) at project level i.e., <project\_name>/<project\_name>/URLs.py , to overcome this we can have URLs.py related to each view in the same app and later linked to URLs.py at project level

if the project is small, then it is okay to have views at app level and URLs.py at project level



# Views

**Views dictate what information is shown to the client**, and **URLs dictate where that information is shown on the website**

**We can think of each view/URL pairing as a webpage on the website**

For a blog related website or an social media platform it is almost impossible to predict how many blogs/posts might get created over time, but django handles this type of scenarios as well, Django supports a lot of dynamic and logic features to help with this sort of task

Ex: you can add new view and link it to corresponding URL (where you want to route to) in URLs.py and make that information available as a blog/social media post, also you can reroute to 404 page for a blog/post that has not been created yet

It is mandatory to connect URLs.py at app level with URLs.py at project level (more information on project level and app level [here](#_How_things_are)) and this is done using django functions called **path()** and **include()**

**Path()** object takes 3 parameters:

1. sub domain url (if the domain is [www.abc.com](http://www.abc.com) and subdomain is def, then the view is displayed at url [www.abc.com/def](http://www.abc.com/def))
2. view function: to display data at [www.abc.com/def](http://www.abc.com/def) you need a logic (view function in which the logic is implemented) and a (html) template. **The second parameter should be a view function**
3. name : let’s say there is a remote url and if you want to redirect from that url to this view function containing URL, then it is needed to identify this path object uniquely and name parameter gets a string assigned and this name is used to identify this path object uniquely

A list of view routes is defined in a list variable called **urlpatterns**, where you link the created views with corresponding URL paths

## Connecting View to URL

* we use path object to connect a view to URL and it accepts 4 parameters (first 2 mandatory, last 2 optional)
  1. string code – this contains the URL pattern, django will scan the relevant **urlpatterns** (a list object in URLs.py file) list until it finds a matching string route (this describes **where is the information that is going to be shown**) (actual route to show the template or the URL)



* 1. **a function or a view** that you typically defined in views.py file of the relevant django app (this describes **what** **is the information that is going to be shown**)
  2. **kwargs** (optional) **–** allows us to pass in keyword arguments as a dictionary to the view
  3. **name** (optional)**–** allow us to **reference** a view and an URL by using this custom defined name (afaik it is used to reference a view and URL in django)

## adding logic to python view

we can add simple logic to a view by connecting python code rather than just sending back a http response

in the below piece of code, you are actually creating 2 dynamic views

1: in the news\_view we are actually using a python dictionary with 3 topics, using python dictionary enables us to not to create different URLs for different topics, instead you can just mention the dynamic value you are passing similar to what is passed in angular brackets in the below URLs.py code

2: the add\_view view function is taking 2 numbers in views.py and returning an f-string to show the adding equation, in URLs.py we are adding an url with <int:num1>/<int:num2>/

views.py

from django.shortcuts import render

from django.http.response import HttpResponse

# Create your views here.

articles = {

    'sports' : 'sports page',

    'finance' : 'finance page',

    'politics' : 'politics page'

}

def news\_view(request,topic):

    return HttpResponse(articles[topic])

def add\_view(request,num1,num2):

    return HttpResponse(f'{num1}+{num2} = {num1+num2}')

URLs.py

from django.URLs import path

from . import views

urlpatterns = [

    path('<str:topic>/',views.news\_view,name = 'news\_view'),

    path('<int:num1>/<int:num2>/', views.add\_view)

]

### Importance of name parameter

When you assign a string to name parameter, then you can use that name to access that particular path inside of urlpatterns list and also in html templates as well.

# Httpresponse and other objects

## HttpResponseNotFound and Http404

Sometimes when you are creating views dynamically there might be some topics/views which are not yet created or sometimes the client might mistype the URL or try to search for a new URL, then we need to use HttpResponseNotFound if you want to **return** a generic response like “page not exist” or you might want to raise an Http404 error incase if you want to raise an exception

Sample piece of code in view.py implementing above both stated methods

from django.shortcuts import render

from django.http.response import HttpResponse, HttpResponseNotFound, Http404

# Create your views here.

articles = {

    'sports' : 'sports page',

    'finance' : 'finance page',

    'politics' : 'politics page'

}

def news\_view(request,topics):

    try:

        result = articles[topics]

        return HttpResponse(result)

    except:

        # raise Http404('404 generic error') #you can link a custom 404 html template here

        return HttpResponseNotFound('page is not created/found')

### When to use what

In the beginning it is suggested to use HttpResponseNotFound, later on you can use Http404 and also you can use a custom 404 html template to display

## HttpResponseRedirect

Let’s say you have moved a particular page from one name to another (ex: [www.google.com/finance](http://www.google.com/finance) is changed as www.google.com/finance.edu), but still some people will try to use the old URL to access the website, or maybe you want to route your client to a new URL when he types some other URL, in such cases we can use **HttpResponseRedirect** or **reverse**.

What does a reverse function do?

It will take the url and return the path object of the corresponding URL in urls.py

You can pass an absolute path or relative path as the target route URL

Ex: if your site name is **temp\_app** and your app name is **app1** and the target url is **end\_result** you need to return HttpResponseRedirect object (from views.py file from **app1**)with a string

1. absolute path - /app1/end\_result and the returing HttpResponseObject syntax should be HttpResponseRedirect(‘/app1/end\_result/’)
   1. note: **the first slash before app1 is important**, else the path that is being passed will be treated as a relative path and that leads to an error
2. relative path - /end\_result and the returing HttpResponseObject syntax should be HttpResponseRedirect(‘endresult/’)
   1. note: using relative path is erroneous and it is suggested to have high priority to use reversal method (which will be discussed later) and as a second priority use this absolute path discussed in above step 1

### sample code implementing above step 1

views.py

from django.shortcuts import render

from django.http.response import HttpResponse, HttpResponseNotFound, Http404, HttpResponseRedirect

# Create your views here.

articles = {

    'sports' : 'sports page',

    'finance' : 'finance page',

    'politics' : 'politics page'

}

def news\_view(request,topic):

    try:

        result = articles[topic]

        return HttpResponse(articles[topic])

    except:

        raise Http404('404 generic error')

        # return HttpResponseNotFound('page is not created/found')

def num\_page\_view(request,num\_page):

    articles\_list = list(articles.keys())

    topic = articles\_list[num\_page]

    return HttpResponseRedirect(f'/first\_app/{topic}')

URLs.py

from django.URLs import path

from . import views

urlpatterns = [

    path('<int:num\_page>/', views.num\_page\_view),

    path('<str:topic>/', views.news\_view),

]

#### important note:

the order of priority in the urlpatterns list has to be in a way that the URL (initial URL)which the client types should be declared first and the URL (target URL) to *which we want to route the initial URL to* should be declared latter

## Reverse URLs and URL names

As website grows, we need more URLs, views and bunch of other stuff such as hyperlinks, buttons, URL redirections

This leads to more instances where we need to refer existing pages

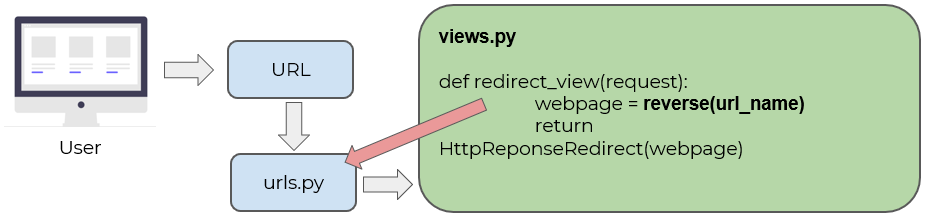
Few points to note

* we need to pass **name** parameter inside **path** object of urlpatterns list in URLs.py, **so that it can be referenced across django and inside of template**
* django has **reverse()** function to find the corresponding url path if provided with a **name**

this way it is more robust

### reverse() function

this function is called reverse function because a typical flow involves a user typing an url which will go to URLs.py and then in turn executes what is inside the views.py, but **when you are redirecting from a view using reverse() function**, the interpreter url name inside the view is taken and is used to search in URLs.py and corresponding url is returned to the view and is redirected



#### Parameters of reverse function

The reverse object has to be sent to HttpsResponseRedirect Object

Reverse object has the following parameters:

First parameter is the **name of the target path** object in URLspattern object in URLs.py

Second parameter is **args** where you need to send the parameters as a list

### Pre-requisites to complete declaring templates at application level:

* Create an app using **python manage.py startapp <application name>**
* Create relevant URLs in URLs.py and views in views.py
* Map the app URLs to the project URLs
* Run **python** **manage.py migrate** command
  + This will take a look at settings.py and will create any necessary database tables if needed by looking at **installed\_apps** list in settings.py if needed (**a typical step when you begin to create models in workflow**)
* Inside of Django App check the **apps.py** created automatically for you and also has a class which is accepting **AppConfig** method, we need to use that class to make an entry in the **INSTALLED\_APPS** list inside of settings.py, syntax:<name of the app>.apps.<name of the class in apps.py file of corresponding application(which is automatically generated when you use startapp command)>
  + Name of the app is the one that you have used to create the app using python manage.py startapp <app name>
    - This will create a configuration class automatically for the above created application in apps.py folder in the same directory
  + This is needed because you need to let the project be aware of the app directories which will in turn let it be aware of the templates directory inside the app
* Register the app and any database changes within django by running **python manage.py makemigrations <name of the app>**
  + This is not relevant to us until we have any models created
  + Whenever you make an **update to a model or create a model or delete a model** (i.e., if you make any changes to models.py file) you need to run **python manage.py makemigrations <name of the app>** and **python manage.py migrate**
* Run **python manage.py migrate** again to create the model tables in our database.
  + Again, not necessary for us yet, but will be once we create models.
* Create a templates directory inside your app directory with this structure:
  + **my\_site**
    - **my\_app**
      * **templates**
        + **my\_app**

**example.html**

* + why do we need to create my app directory twice
    - Often you’ll have multiple template files with the same name (multiple index.html files, one for each app index view page).
    - Because of the way Django searches for matching template names, to make sure we get the relevant template for an app, we create the app subdirectory underneath the template folder.
    - **Official documentation** - Now we might be able to get away with putting our templates directly in **myapp/templates** (rather than creating another **myapp** subdirectory), but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a different application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the best way to ensure this is by namespacing them. That is, by putting those templates inside another directory named for the application itself.

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

]

### An example

Views.py

from django.shortcuts import render

from django.http.response import HttpResponse, HttpResponseNotFound, Http404, HttpResponseRedirect

from django.URLs import reverse

# Create your views here.

articles = {

    'sports' : 'sports page',

    'finance' : 'finance page',

    'politics' : 'politics page'

}

def news\_view(request,topic):

    try:

        result = articles[topic]

        return HttpResponse(articles[topic])

    except:

        raise Http404('404 generic error')

        # return HttpResponseNotFound('page is not created/found')

def num\_page\_view(request,num\_page):

    articles\_list = list(articles.keys())

    topic = articles\_list[num\_page]

    return HttpResponseRedirect(reverse('topic-page', args = [topic]))

URLs.py

from django.URLs import path

from . import views

urlpatterns = [

    path('<int:num\_page>/', views.num\_page\_view),

    path('<str:topic>/', views.news\_view, name = 'topic-page'),

]

# Templates

## Need of templates

We need templates because, realistically you cannot type out HTML code inside a function in view.py file

Instead, we would like to separate out all our html files as templates in a separate directory and have views communicate with this directory and render the templates

Is the relation between views and templates two way? (#need\_to\_know)

We can store html templates at application level and at a project level, it is always suggested to have templates at an application level, also you need to edit settings and add information about the location of templates, so that views.py functions can render them.

We also know that django supports admin and authorization features, and there are admin and authorization templates made readily available with django something like below (if you closely observe, you can see that there are admin and auth templates stored in the python environment named django-course that I created for this course), if you go traverse further you can see a lot of html templates

* /home/saivinil\_pratap/miniconda3/envs/django-course/lib/python3.10/site-packages/django/contrib/admin/templates/
* /home/saivinil\_pratap/miniconda3/envs/django-course/lib/python3.10/site-packages/django/contrib/auth/templates/

## Templates at project level

Templates at project level are handled in the list named templates in settings.py file

It consist of a json which has

* BACKEND : this is the place where django will look for templates by default
* DIRS : this is the list where django will search for all specified folders for templates, **if there are any other folders (having) other than templates at project level (project\_name/templates) or at an app level (project\_name/<app\_name>/templates/<app\_name>) then you need to specify so that django will try to look at them in case if it does not find them in relevant places**
* APPDIRS : this Boolean value will specify whether we have to look for templates at in the available applications
* OPTIONS

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [],

        'APP\_DIRS': True,

        'OPTIONS': {

            'context\_processors': [

                'django.template.context\_processors.debug',

                'django.template.context\_processors.request',

                'django.contrib.auth.context\_processors.auth',

                'django.contrib.messages.context\_processors.messages',

            ],

        },

    },

]

## Templates at application level

Templates at application level are declared in the list named INSTALLED\_APPS in the settings.py file, a sample is shown below, **only then django will be aware of the app’s template existence**

## Using templates in views

To include templates in views, we need to render the templates , we need to import render from django.shortcuts and then pass the request and relative template location (relative location because you already configure the directory structure in above [templates at project level](#_Templates_at_project) and [templates at application level](#_Templates_at_application)))

views.py

from django.shortcuts import render

from django.http.response import HttpResponse, HttpResponseNotFound, Http404, HttpResponseRedirect

from django.URLs import reverse

# Create your views here.

def sample\_view(request):

    return render(request,'first\_app/example.html')

URLs.py

from django.URLs import path

from . import views

urlpatterns = [

    path('',views.sample\_view)

]

### Important note

In the above views.py see the path that is getting passed to the render object in the return statement i.e., first\_app/example.html. regardless of the path it will try to look inside templates directory, if you look at the template path of example.html, that says it will look for **a example.html file inside first\_app which is inside templates folder of application which loosely translates to the path <projectname>/<app\_name>/<templates>/<app\_name>/<html file>**

If you just pass /example.html as path, then you can save html at **templates folder of application which loosely translates to the path <projectname>/<app\_name>/<templates>/<html file>**

**Basically the point I want to drive home is that based on the path that is passed through the rendcer object, it should have relevant file at corresponding location**

## Why separate templates at an app level

Because apps can be reused

If you have an app which deals with storing and retrieving photos, which you created for a social media app, that app can be reused for a marketplace where you can retrieve stored photos

## Jinja templating

When working at app level After you connect views with urls, and then you connect urls at app level with urls at project level, you will add templates in project/<app name>/templates/<app name>/<name of html file>, then you can pass the dictionary from view function to the parameter **context** to the render object (which you return) and then parse that dictionary in the HTML file.

Ex:

views.py

from django.shortcuts import render

from django.shortcuts import render

# Create your views here.

def variable\_view(request):

    my\_var = {'first\_name':'vinil','last\_name':'pratap',

    'some\_list':[1,2,3],'some\_dict':{'inside\_key':'inside\_value'}}

    return render(request,'my\_app/variable.html', context = my\_var) #this actually after turning in to absolute path will look like my\_app/templates/my\_app/variable.html

variables.html located at my\_app/templates/my\_app/

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>Variable.html</h1>

    <h1>my first name is {{first\_name}}, my last name is {{last\_name}}</h1>

<h3>accessing the list {{some\_list}}</h3>

<h3>accessing the element at second index {{some\_list.2}}</h3>

    <h3>accessing the sub dictionary {{some\_dict}}<h3>

    <h6>accessing the key in the sub dictionary {{some\_dict.inside\_key}}</h6>

{# this is a comment#}

</body>

</html>

### Accessing the elements using jinja templating

List : list.index to fetch the value at a particular index

Dict: dict.key to fetch a particular value related to a particular key

Note: in the above example you can notice how they are accessing keys, list, dictionary and comments

## Filters

You will try to manipulate the data you can do that in a view function or in a html template, to manipulate data in a view function, you use python code, to manipulate data in a template you use filters

Filters are built-in modifiers in Django templating that allow you to quickly apply a change to a variable on the template side, rather than in your Python script.

To apply filter, you will take a dynamic variable that you got using jinja templating **{{}}** by passing a pipe symbol

Check the below views, example where you use **upper** and **length** filters

Documentation link: https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#built-in-filter-reference

variables.html file

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>Variable.html</h1>

    <h1>my first name is {{first\_name | upper}}, my last name is {{last\_name}}</h1>

    <h6>accessing the key in the sub dictionary {{some\_dict|length}}</h6>

    {# this is a comment#}

</body>

</html>

Also you can stack one filter on the other

<h1>my first name is {{first\_name | lower | capfirst}}, my last name is {{last\_name | capfirst}}</h1>

Here you take the first\_name make all the letters in it to lowest, then capitalize the first letter

### When to use filters and when to use python code

It depends on choice and complexity, if the code is looking complex (either in html or python) you might want to stop and look and choose a way which might help you have better understanding , **also you can use filters as a last minute check, for example you want to print the name that you get from the data base with first letter capitalized, to double check you can use *capitalize* filter function,** in this way you can make sure functionality is met, even if the data source is not properly entered in to the database

## Tags

Represented as {% %}

* Django Tags can provide further logic at the template in the rendering process.
* This includes a lot of functionalities, such as for loops, if-else statements, and linking to URLs.
* There are many tags which help us achieve functionalities similar to iterating dictionaries or lists using for loops

### Important points to note while using python logic, django templating language and tags

* the programming syntaxes slightly differ from normal python syntax
* it is always suggested to give indentation even when HTML doesn’t need it, because we will get a very good understanding with tagging structures as, they are like python
* you always need to maintain spacing for operators when it comes to loops, else you will get errors, this is because you are coding in
  + ex: the below example will raise an error, because
* {% if len(sample\_list)==3 %}
* <h1>you have 3 items in your list </h1>
* {% endif %}

### For loop in tags

#### Iterating through dictionary

<body>

    <h1>Variable.html</h1>

    <ul>

    {% for k,v in some\_dict.items%}

    <li>{{k}}</li>

    {%endfor%}

    </ul>

    {# this is a comment#}

</body>

#### Iterating through list

<body>

    <h1>Variable.html</h1>

    <ul>

    {% for foo in some\_list %}

    <li>{{foo}}</li>

    {%endfor%}

    </ul>

    {# this is a comment#}

</body>

### If loop

If loop is **powerful,** because you already have the facility to fetch data from database, imagine, you have a streaming platform and if you want to give an premium user the early access for recently released movie, then as you already have the idea of who is a premium user, by using if loop you can give access by checking whether the particular user is an premium user or not

#### One more point why django is powerful

In the below example, by using **python logic, django templating and django tagging** we are **displaying different things to different users on the same page using the exact same template.** Also all the logic exists at the backend even if someone inspects the page **they will not get the login message** if they are not logged in, **because it is never sent to the client, if he is not logged in**

views.py

from django.shortcuts import render

from django.shortcuts import render

# Create your views here.

def variable\_view(request):

    my\_var = {'first\_name':'vinIl','last\_name':'pratap',

    'some\_list':[1,2,3],'user\_logged\_in':True}

    return render(request,'my\_app/variable.html', context = my\_var)

variable.html

<body>

    <h1>Variable.html</h1>

    {% if user\_logged\_in %}

        <h1>hi {{first\_name | lower | capfirst}} thanks for logging in</h1>

    {% endif %}

    {# this is a comment#}

</body>

## Using Tags and URL names in templates

We know how we can assign names to path object in urls.py file, so that they can be retrieved later, what is the need to retrieve the urls later, we see hyperlinks in a page, where on clicking it you will be routed to url (of same or different website), to achieve this scenario, we will assign names to path objects

To use URL names, i.e., To have another web page (html template) in the current url page (html template) as a hyperlink,

You need to have/do 3 things (here only second step is mandatory)

1. Define the app name in urls.py like below
   1. app\_name = <name of the app>
2. name parameter assigned in path object of urls.py like this
   1. path('variable/',views.variable\_view, name = 'variable\_view')
3. the template can be at application level or at an project level, both need different syntaxes
   1. **at project level:** use the path name in the template where we want to embed the html template (where we have got the above view function being used) **name of the path object are both in urls.py** (tried to work this out, but it is not working, #need\_to\_know)
      1. syntax: <h1><a href= "{% url '< name parameter of path obj>':%}"> click here to go to variable</h1>
      2. ex: <h1><a href= "{% url 'variable\_view'%}"> click here to go to variable</h1>
   2. **at application level:** use the above app\_name and path name in the template where we want to embed the html template (where we have got the above view function being used) **both app\_name and name of the path object are both in urls.py**
      1. syntax: <h1><a href= "{% url '<app name:name parameter of path obj>':%}"> click here to go to variable</h1>
      2. ex: <h1><a href= "{% url 'my\_app:variable\_view'%}"> click here to go to variable</h1>

### note (if you want to create re-routings at home page)

if you want to use URL names and tags for pages as home page (ex: [www.abc.com/home](http://www.abc.com/home)) you can use this above mentioned **tags and URL names at a project level** or make a app called **home** and **link the URLs of app level in the project level urls.py as below**

from django.contrib import admin

from django.urls import path,include

from . import views

urlpatterns = [

   path('',include('my\_app.urls'))

]

What happened in the above piece of code is that you created an app which is working at an app level and as it does not have a **extra path** specified (empty string in above path object) in urlpatterns list in urls.py file at project level, therefore all the views and URLs you create at app level will be displayed from home page (ex: [www.abc.com/](http://www.abc.com/))

## Template inheritance

Python follows DRY principle, so it feels natural to not hold any html information which is repetitive, example: navigation bar on the website.

A solution to achieve DRY principle here is to use {%block%} tag by which we can inherit repetitive templates

You can have content which is (most) repetitive in a file called **base.html** which you can use in other files (ex: example.html)

**It is suggested to have repetitive html at project level and normal html at an app level**

### How does it work?

You need to use the below mentioned {% block <name of the block>%} and {% endblock %} as a placeholder in **base.html** file, so that you can include navbar code above **block content** and footer note **below** endblock. look the first box in the below image

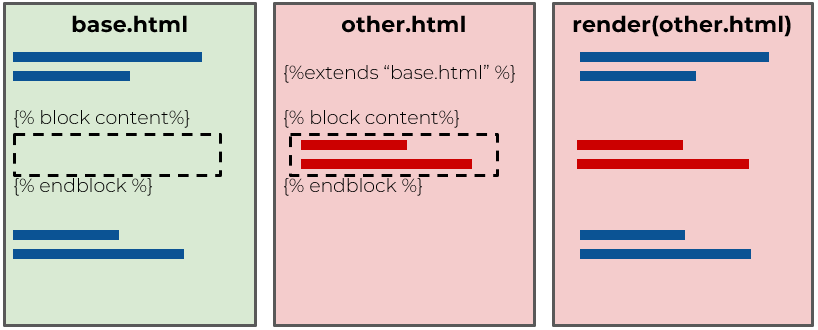
Ex:

{% block content%}

{% endblock content%} #content is the same name you gave to the above line and it is optional to declare name ad the end of the block

Now you can use that {%extends “base.html”%} in the other.html so that you can use that contents here, and you use the code that you want to fit in between navbar and header inside the {% block content %} and {% endblock %}. look the second box in the below image

So that when it is rendered it will look like the third box in the below image



example.html at <project\_name>/<app\_name>/templates>/<app\_name>/example.html

{% extends 'base.html' %} <!--here the html template to be inherited is at a project level, if it is at a app level, then you need to give <app\_name:template\_name> -->

{% block content %}

<h1>inside the block in example.html</h1>

{% endblock content %}

base.html file described above will be at <project\_name>/templates/

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

<h1>above the content block</h1>

{%block content%}

{% endblock content%}

<h1>below the content block</h1>

</body>

</html>

### Important note

Inheritance is not limited to just using one block, you can daisy chain and linkup multiple inheritances or extensions, **one main thing is that keep everything well defined in settings so that django is looking in correct places for templates**

## Custom error templates (like 404,500 and …)

* Many pages, such as admin or 404 pages have built-in templates provided by Django for your convenience (i.e., whenever you type a wrong URL address the custom 404 error page).
* However, we have the ability to overwrite any of these built-in templates
* If you want to override that custom page, you need to create a html template named 404.html in <project\_name>/<project\_name>/templates/404.html
* If you mention by default the page as 404.html you don’t have the need to specify handlers, and django will automatically use the 404.html template in case the server responds with 404 error, **it is** **always suggested to use 404.html (not any custom name) as it is more robust,** but If you want to have a custom name for your error then you need to do the following steps

1. Create a error template and store it in in <project\_name>/<project\_name>/templates/

<!--skeleton html-->

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>my custom 404 page</h1>

</body>

</html>

1. Create a view in views.py

from django.shortcuts import render

# Create your views here.

def variable\_view(request):

    return render(request,'my\_site/demo.html')

def my\_custom\_page\_not\_found\_view(request,exception):

    return render(request,'404\_page\_with\_custom\_name.html',status=404)

1. Use **corresponding handler name (they are specific and used the way they are, ex: handler404 in this case)** in urls.py and you need to assign view function to this variable, the list of available handler names is here: https://docs.djangoproject.com/en/4.1/topics/http/views/#customizing-error-views

from django.contrib import admin

from django.urls import path,include

from . import views

urlpatterns = [

    path('demo/',views.variable\_view, name = 'demo'),

    path('admin/', admin.site.urls),

    path('my\_app/',include('my\_app.urls'))

]

handler404 = 'my\_site.views.my\_custom\_page\_not\_found\_view'

## static tags

* Most projects will have static files (static files are the files which does not change with user actions/inputs), such as images, JS, or CSS.
* Django can serve these static files through the use of Tags, instead of having to refer to a full file path.
* This is similar to the {% url %} tag, but using a {% static %} tag.

Before we move ahead, make sure you have 'django.contrib.staticfiles' in **installed\_apps** list in **settings.py** file

Similar to templates are stored in templates folder, you will have static items getting stored in STATIC\_URL

#folder structure: <project\_name>/<app\_name>/static/<project\_name>/

STATIC\_URL = 'static/'

### Setting static file locally

Example:

{% extends 'base.html' %} <!--here the html template to be inherited is at a project level, if it is at a app level, then you need to give <app\_name:template\_name> -->

{% load static %} <!--this will load the static content-->

{% block content %}

<h1>inside the block in example.html</h1>

<!--below both approaches will work, in the second approach even though, the folder is different and you are not specifying the folder it is working because even though its grandparent folder is different, their parent is bearing same name -->

<img src="{% static '../static/my\_app/vinil.jpg'%}" alt = "my image">

<img src="{% static '/my\_app/vinil.jpg'%}" alt = "my image">

{% endblock content %}

Folder structure:

Text

Description automatically generated

Note: if the images do not load for the first time, try reloading the server

### Setting static file globally

Link <https://docs.djangoproject.com/en/4.1/howto/static-files/>

# Databases

Django models allow us to interact with a database with python

They provide CRUD interactions

Django supports models of SQL and NoSQL databases, SQL stores data in tabular format, NoSQL stores data in key value pairs, Django works pretty well with SQL databases, you might need to install a couple of libraries to make django work with NoSQL Db’s .

SQL db’s:

1. MySQL
2. PostgreSQL
3. SQLite
4. MS SQL
5. And few others

**Django is pretty agnostic to most major SQL engines with the use of its Django Models system, so switching to another SQL engine is more a matter of updating settings.py rather than rewriting the actual Python Django code**

## Db settings in settings.py file

In below code we the ENGINE specifies the type of engine you will be using, and the name specifies (afaik) where is the actual database

**By default** it uses the sqlite3, because

1. When you work with django, you are working with python, and sqlite comes readily with python, so it is readily configured with django in **settings.py file as below**
2. Also it is small, compact and free

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': BASE\_DIR / 'db.sqlite3',

    }

}

As sqlite3 comes by default it needs less configurations, if you want to use another DB, you might want to install few packages and do few configurations, more details here: <https://docs.djangoproject.com/en/4.1/ref/settings/#databases>

## Models

* Django Models are defined inside a Django app (or project) models.py file.
* The models class operates on a system which directly converts Python based code into SQL commands.
* This makes it much easier to work with the backend database.
* Creating a Model is similar to creating a new table in a database.
* Each database table has a name and then columns, where each column will have a specific data type, for example: **character strings** for names or **integers** for ages in years.

Let’s now explore how Django Models works in conjunction with these structures

* Django Model Key Concepts:
  + Inherits from **model’s** class.
  + Uses **fields** to define both data **types** and data **constraints**.
    - * For example, you may want to **require** information, like a user’s email address, in which case you can add a **NOT NULL** constraint
* Django Models can also be connected through keys:

Text

Description automatically generated

### Points to note

* When you are creating a class, try to get all the attributes (columns of a table) at the beginning to avoid further issues, but django allows you to add columns in further when you need to create new fields
* When you are deciding on inputs regarding attributes, for ex- take last\_name length, if you decide last\_name should not be more than of size 10, and if you start accepting a data from a country where they have possibly larger last\_names, then you might want to reconfigure appropriate fields and run **python manage.py migrate**
  + **So it is always better to go long over short, or to think well before deciding constraints of attributes or features of attributes**

## Migrations

* Migrations is the act of connecting changes in your django project or app to a database
* This includes things like adding new models within an application, adding a new application, updating models with a new column/attribute, and more.
* You typically see these commands done through the manage.py file.
* Let’s discuss the migrate based commands you can run:
  + makemigrations
  + migrate
  + sqlmigrate

### python manage.py makemigrations <app\_name>

when you run makemigrations for an app, then it will check if the app is registered the application as configuration (i.e., you need to add that in to the INSTALLED\_APPS list in the settings.py file)

this above command actually creates the set of instructions (**but does not run**) the set of instructions that will apply changes to the database

note, the default apps like auth and admin have already their makemigration code ready and **just not run yet**

when you run the makemigrations command, a directory named migrations (similar to having templates in template directory and static items in static directory, you will find python scripts inside the migrate directory) will be created and the structure is as follows

* app
  + migrations
    - 001\_initial.py

The above python file is human readable and is editable, but that is advanced stuff for now

### Python manage.py migrate

* Runs any existing migrations (typically created through the makemigrations command).
* This is actually running the files under the migrations directory from the previous command.

#### python manage.py sqlmigrate <application name> <number part of the python file that is generated under migrate directory>

if the file name is 001\_initial.py, then the command will look like

python manage.py sqlmigrate <application name> 001

when you run makemigrations command, then you generate .py files in migrate directory, to see how these files will view when converted to sql queries you can run, python mange.py sqlmigrate command

* Note that typically we won’t review the files created under the migrations directory or run sqlmigrate, we’ll simply run makemigrations and migrate.
* You can think of the very first migrate command we run as executing the default (admin, auth, contenttypes sessions) makemigrations that was already created for you upon creating the project.

#### Steps for migrations

* Initial project migrate command
* Create app and create models
* Register app in INSTALLED\_APPS(mention config class name which follows the syntax <name of the app>.apps.<name of the class in apps.py file of corresponding application(which is automatically generated when you use startapp command)>) list in settings.py
* Run makemigrations for new app
* Run migrate for new migrations

## Data interaction

We can perform crud operations in to the database using django models

### Insertion using models

Inserting new data into a SQL table is easy with Django Models.

* Since the models are represented by a class, we can easily create a new instance of the class object in Python, and then call the .save() method to create an INSERT call to the SQL database.
* Alternatively, you can use the built-in .objects.create() method to both create and save the new data entry in a single line.
* In instances where you want to create multiple new data entries in bulk, you can use the .objects.bulk\_create() method to pass in a list of newly created objects.
* Let’s explore these 3 methods of creating new data entries with a model:
  + Create Object and .save()
  + objects.create()
  + objects.bulk\_create()

in general the idea behind performing crud operations is that, data can be sent to and fro from front end to back end with the ability to perform CRUD operations

but, while developing if you just want to check the functionalities, or if you are just taking baby steps with CRUD operations using Django, developing front end for CRUD will seem like a lot, so you can use a command **python manage.py shell (this will open a normal interpreter, if you want to use ipython you need to install it using conda install ipython relevant installation command)** which will provide a interactive shell so that you can initiate models and perform crud operations

#### 3 methods of creating new entries

* Create Object and .save() – here you will create an object and then you have the privilege of checking the data we store in that object and reassign if needed and then save it to db by using .save() method

In [2]: from office.models import Patient

In [3]: vinil = Patient('vinil','pratap','30')

In [4]: vinil.age #this will not work as you did not pass keyword parameters like we did in below

In [5]: sinil = Patient(first\_name='vinil',last\_name='pratap',age='30')

In [6]: sinil.age

Out[6]: '30'

In [7]: sinil = Patient(first\_name='vinil',last\_name='pratap',age=30)

In [8]: sinil.age

Out[8]: 30

In [9]: sinil.name = 'sinil'

In [10]: sinil.name

Out[10]: 'sinil'

In [11]: sinil.save()

* objects.create()

In [12]: Patient.objects.create(first\_name='satya',last\_name='vankadari',age='1000')

Out[12]: <Patient: Patient object (2)>

In [13]: Patient.objects.create(first\_name='ranjith',last\_name='chitrala',age='1')

Out[13]: <Patient: Patient object (3)>

* objects.bulk\_create()
* In [3]: from office.models import Patient
* In [4]: l = [Patient(first\_name='1',last\_name='1',age='1'),Patient(first\_name='2',last\_name='2',age='1')]
* In [5]: Patient.objects.bulk\_create(l)
* Out[5]: [<Patient: Patient object (4)>, <Patient: Patient object (5)>]

### Reading and querying db (models)

Each Model you create comes with a Manager that allows you to create a QuerySet which can then be used to retrieve entries from the database.

Keep in mind that the QuerySet is actually lazily evaluated, meaning that it doesn’t hit the database until its explicitly asked to grab the information.

* Recall we used something like:
  + **<Class name in models.py file>.objects**
* This is the Django Model Manager.
* This Manager can then actually read the database through the use of method calls, like .**all()** and **.get()** and narrow down results with **.filter()** and **.exclude()**
* Great resource on queries with examples:
  + **docs.djangoproject.com/en/4.0/ topics/db/queries/**

#### all() method

* patients.objects.all method
* check the outputs of the last objects.bulk\_create() method, you can see that objects that are inserted are getting displayed. This is because **Patient class** is getting inherited from **model class** and whenever you try to retrieve or print an object **the address of the object will be returned, in order to avoid this and make the output more readable, you can use the \_\_str\_\_ method in the Patient class, so that whenever you try to retrieve an object or print it, it will print the custom method that you are trying to return, check the below str method which will explain the above theory clearly**

from django.db import models

# Create your models here.

class Patient(models.Model):

    first\_name = models.CharField(max\_length=30)

    last\_name = models.CharField( max\_length=30)

    age = models.IntegerField()

    def \_\_str\_\_(self):

        return f'{self.last\_name}.{self.first\_name} is {self.age} years old'

output

In [1]: from office.models import Patient

In [2]: Patient.objects.all()

Out[2]: <QuerySet [<Patient: pratap.vinil is 30 years old>, <Patient: vankadari.satya is 1000 years old>, <Patient: chitrala.ranjith is 1 years old>, <Patient: 1.1 is 1 years old>, <Patient: 2.2 is 1 years old>]>

#### filter() and get() methods

* The **.get()** operation allows us to grab a **single** item from the Model table.
* This is typically reserved for something where you are sure there is only a single unique entry, like the default primary key that is automatically created by Django (**pk=N)**.
* If we want to further filter our results (rather than grab all or get a single item), we can use the **.filter()** method to narrow down based on conditions.
* The **.filter()** methods can be chained together.
* Let’s explore the following topics:
  + Using **.get()**
  + Using **.filter()**
  + Using **operators with filter**

##### get method

this method is used when you know that with the input parameters you are going to get only one record, because fetching more than one record or fetching 0 records is considered an error by get method

so typically pass parameters such that you should get only one record or pass a primary key (which by default is unique) which will return only one record

In [3]: Patient.objects.get(pk=1)

Out[3]: <Patient: pratap.vinil is 30 years old>

In [5]: Patient.objects.filter(last\_name='pratap')

Out[5]: <QuerySet [<Patient: pratap.vinil is 30 years old>]>

In [6]: Patient.objects.get(last\_name='vinil')

---------------------------------------------------------------------------

DoesNotExist                              Traceback (most recent call last)

Input In [6], in <cell line: 1>()

----> 1 Patient.objects.get(last\_name='vinil')

In [8]: Patient.objects.get(age=1)

---------------------------------------------------------------------------

MultipleObjectsReturned

MultipleObjectsReturned: get() returned more than one Patient -- it returned 3!

###### Important note

In SQLite indexing starts from 1 not 0

In [11]: Patient.objects.filter(age=1)

Out[11]: <QuerySet [<Patient: chitrala.ranjith is 1 years old>, <Patient: 1.1 is 1 years old>, <Patient: 2.2 is 1 years old>]>

##### filter method

filter method is used when you want to retrieve some records which meet a criterion. Ex: patients of age above 80

In [11]: Patient.objects.filter(age=1)

Out[11]: <QuerySet [<Patient: chitrala.ranjith is 1 years old>, <Patient: 1.1 is 1 years old>, <Patient: 2.2 is 1 years old>]>

In [12]: Patient.objects.filter(age=1).all().filter(first\_name = 'ranjith')

Out[12]: <QuerySet [<Patient: chitrala.ranjith is 1 years old>]>

You can also stack one filter method on the other, here you stacked second filter over the first filter and then a get method on top of it.

##### Operator on filter method

And now let's talk about operators within a query set in order to use this,We have to import what's known as a **Q** object inside of Django. as a quick note, **You should definitely check out the documentation on this.**

**It is kind of this weird beast, but once you get used to it**, it allows you to have a lot more functionality

that is more skill based in order to filter out results.

More details here: <https://docs.djangoproject.com/en/4.1/topics/db/queries/#complex-lookups-with-q-objects>

**Q object helps us work on “or”-> | and “and” -> & operations**

In [17]: from django.db.models import Q

In [19]: Patient.objects.filter(Q(last\_name = 'chitrala') & Q(age = 1))

Out[19]: <QuerySet [<Patient: chitrala.ranjith is 1 years old>]>

In [20]: Patient.objects.filter(Q(last\_name = 'chitrala') | Q(age = 1))

Out[20]: <QuerySet [<Patient: chitrala.ranjith is 1 years old>, <Patient: 1.1 is 1 years old>, <Patient: 2.2 is 1 years old>]>

##### Filtering with field lookups

* So far we’ve had to use equality statements in our filtering (age=30 or last\_name= ‘smith’).
* But what about more general comparison operators:
  + Greater than or less than
  + Starts with
* For more complex filtering operations we use **field lookups** with a **filter()** call:

**Model.objects.filter()**

* For more complex filtering operations we use **field lookups** with a **filter()** call:

**Model.objects.filter(name\_\_startswith= “s”)**

* There are **many** field lookup calls available:
* **docs.djangoproject.com/en/4.0/ref/ models/querysets/#field-lookups**

In [24]: Patient.objects.filter(first\_name\_\_startswith = 's')

Out[24]: <QuerySet [<Patient: vankadari.satya is 1000 years old>]>

In [25]: Patient.objects.filter(age\_\_in = [1000,30,1])

Out[25]: <QuerySet [<Patient: pratap.vinil is 30 years old>, <Patient: vankadari.satya is 1000 years old>, <Patient: chitrala.ranjith is 1 years old>, <Patient: 1.1 is 1 years old>, <Patient: 2.2 is 1 years old>]>

In [26]: Patient.objects.filter(age\_\_gte = 30)

Out[26]: <QuerySet [<Patient: pratap.vinil is 30 years old>, <Patient: vankadari.satya is 1000 years old>]>

In [27]: Patient.objects.order\_by('age')

Out[27]: <QuerySet [<Patient: chitrala.ranjith is 1 years old>, <Patient: 1.1 is 1 years old>, <Patient: 2.2 is 1 years old>, <Patient: pratap.vinil is 30 years old>, <Patient: vankadari.satya is 1000 years old>]>

In [28]: Patient.objects.filter().order\_by('age')

Out[28]: <QuerySet [<Patient: chitrala.ranjith is 1 years old>, <Patient: 1.1 is 1 years old>, <Patient: 2.2 is 1 years old>, <Patient: pratap.vinil is 30 years old>, <Patient: vankadari.satya is 1000 years old>]>

###### .values on filter

To get a particular field from a filter method result, we can use .values, .values can be applied on a QuerySet object, and here the filter method also returns a QuerySet object

An example: if you want to fetch the names alone of patients whose age is greater than eighty

In [41]: Patient.objects.filter(age\_\_gt=80).values('first\_name')

Out[41]: <QuerySet [{'first\_name': 'satya'}]>

###### note

it is not needed to remember all the functions (because there are many in number) and they are already available at the above link.

The names of the function are pretty self-explanatory and most of the time they have their name resembling their corresponding functionality in python or sql

### Updating models

* There may come a time when you need to create a new column or attribute for a model.
* You can easily update existing models by simply adding a new model class attribute and then migrating those changes.
* You should note that when adding new fields, the existing entries will need to have some default value inserted (even if it’s just null).
* In fact, when we attempt to run migrations without taking care of these (ex:default value) issues, Django will specifically request us to make a decision.
* You’ll be given two options:
  + Choose a default value on the spot when making the migrations file.
  + Cancel the migration and create a default value within the model.

The error message looks something like this:

(django-course) saivinil\_pratap@TIGER02143:/mnt/d/Django/models-databases-fields/my\_site$ python manage.py makemigrations office

It is impossible to add a non-nullable field 'heart\_rate' to patient without specifying a default. This is because the database needs something to populate existing rows.

Please select a fix:

1) Provide a one-off default now (will be set on all existing rows with a null value for this column)

2) Quit and manually define a default value in models.py.

Select an option: 2

Patients class in models.py

from django.db import models

# Create your models here.

class Patient(models.Model):

    first\_name = models.CharField(max\_length=30)

    last\_name = models.CharField( max\_length=30)

    age = models.IntegerField()

    heart\_rate = models.IntegerField(default=60) #if you want to assign some default value

    # heart\_rate = models.IntegerField(null=True) #if you want to assign a null value

    def \_\_str\_\_(self):

        return f'{self.last\_name}.{self.first\_name} is {self.age} years old'

* It’s usually more robust to have the default live in the model, but each case is different.
* We can also use **validators** with fields, which add hard-coded constraints that will reject non valid entries!

#### Validators

You can have a set of criteria to be met, because you might want to control few inputs, ex: age cannot be more than 200, and sex category only needs male, female and others as inputs, **you can achieve that constraints allocation to fields in python models using Validators**

Ex: validators for age and heart rate are below

from django.db import models

from django.core.validators import MaxValueValidator, MinValueValidator

# Create your models here.

class Patient(models.Model):

    first\_name = models.CharField(max\_length=30)

    last\_name = models.CharField( max\_length=30)

    age = models.IntegerField(validators=[MinValueValidator(0),MaxValueValidator(120)])

    heartrate = models.IntegerField(default=60,validators=[MinValueValidator(1),MaxValueValidator(300)])

    def \_\_str\_\_(self):

        return f'{self.last\_name}.{self.first\_name} is {self.age} years old'

### Updating existing entries

* Django makes it very easy to update existing entries.
* You simply grab the existing data entry using **.get() method**, update any attributes, then .save() the changes to write the update to the database table.

In [6]: Patient.objects.get(pk=1)

Out[6]: <Patient: pratap.vinil is 30 years old>

In [7]: prat = Patient.objects.get(pk=1)

In [8]: prat.age

Out[8]: 30

In [9]: prat.age=27

In [10]: prat.save()

In [11]: prat=Patient.objects.get(pk=1)

In [12]: prat.age

Out[12]: 27

In [13]: prat

Out[13]: <Patient: pratap.vinil is 27 years old>

### Deleting existing entries

Deleting existing entries is similar to updating/creating entries, for creating/updating, we use .save() and for deleting we use .delete

We will grab the existing data entry as an object using .get() method and then apply .delete() method on the obtained object

Note: you need to be careful with the command because it will remove the entries from the database, so it is needed to be careful with the data we are fetching and is suggested to check before executing .delete() method

Ex:

In [20]: prat = Patient.objects.get(pk=4)

In [21]: prat

Out[21]: <Patient: 1.1 is 1 years old>

In [22]: prat.delete()

Out[22]: (1, {'office.Patient': 1})

In [23]: Patient.objects.all()

Out[23]: <QuerySet [<Patient: pratap.vinil is 27 years old>, <Patient: vankadari.satya is 1000 years old>, <Patient: chitrala.ranjith is 1 years old>, <Patient: 2.2 is 1 years old>, <Patient: p.vini is 2 years old>, <Patient: p.vini is 2 years old>, <Patient: p.vini is 0 years old>, <Patient: p.vini is 0 years old>, <Patient: s.vinil is 0 years old>]>

# Django Admin

* One of the most powerful features of Django is its ability to automatically create an Admin interface.
* This is a feature meant to be used by the website manager, to have a graphical interface for interacting with data and users on the site.
* We’ve already seen pre-built admin paths in our site urls.py file (**“/admin”**) as well as indications of an existing Django Admin app (**“django.contrib.admin”**).
* In this section we’ll explore how to access the admin panel view and how to configure admin settings.
* Keep in mind that the admin panel is really meant for a manager of the website, we won’t expect normal users to access the Django Administration interface.
* With each app you make, you can see an admin.py (this is where you register your models) file getting created, there you can register the models, it does not make sense to register every single model, it is only **needed to register models which super user should be able to see and have access to it inside the admin interface**

Ex:

from django.contrib import admin

from cars.models import Car

# Register your models here.

admin.site.register(Car)

Graphical user interface, application

Description automatically generated

* **ModelAdmin** class which gives us additional functionality with the fields presented in the Admin Interface, you inherit a new class from the existing admin (from django.contrib import admin) and override existing behaviors if needed, like changing the order of priority of the model class parameters using fields parameter, or grouping parameters of model class using fieldsets parameter

from django.contrib import admin

from cars.models import Car

# Register your models here.

# # simple registration of app

# admin.site.register(Car)

# overriding existing registration

class CarAdmin(admin.ModelAdmin):

    fields = ['year','brand']

    # fieldsets = [

    #     ('TIME INFORMATION',{'fields':['year']}),

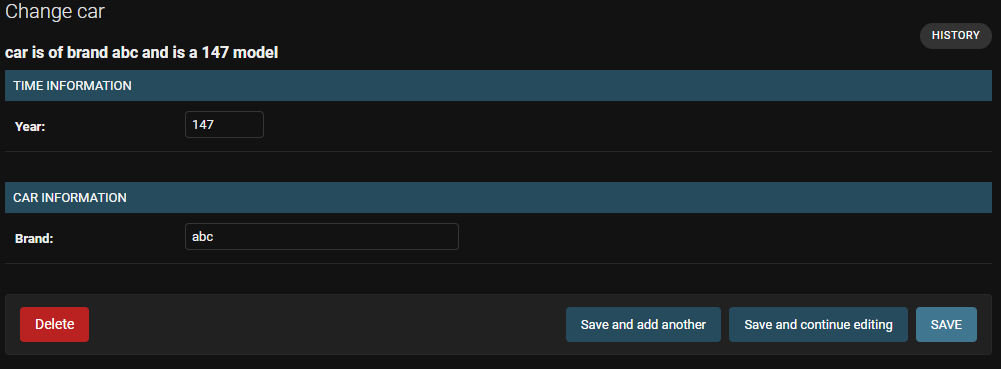
    #     ('CAR INFORMATION',{'fields':['brand']})

    # ]

admin.site.register(Car, CarAdmin)

Graphical user interface, application

Description automatically generated



# Django forms

* We have been able to use HTML forms to allow client users send information in their browser to the backend of our Django application.
* The Django website can then Create/Read/Update/Delete information in the database based on the HTML forms.
* User interactions based on HTML forms are extremely common across the internet.
* However, HTML forms require a lot of processing to connect with Django, especially when we want to later connect these inputs to Models.
* Fortunately, Django comes with a built-in Forms class which can be used with Django and Python to create forms and then send that form to the template through a **simple Tag call** **{{form}}**.
* This allows us to rapidly develop forms for the client while only needing to work with Django and Python! This will be a **huge productivity improvement** and make our overall website code more readable.

## get, post and CSRF

* We’ve already seen that HTTP (Hyptertext Transfer Protocol) is the foundation for the method of sending and receiving data over the world wide web.
* Recall, HTTPS is simply an encrypted version of HTTP
* HTTP defines a variety of methods for interactions.
* The key methods we need to understand are **GET** and **POST** methods, which we’ve already seen used in HTML forms.
* **GET**
  + Requests data from a specified resource.
* **POST**
  + Requests to send data to a server to create/update a resource.
* Note the tricky terminology that can be confusing, both GET and POST are HTTP **request methods**, even though you will commonly see GET as “*requesting*” information and POST as “*sending*” information, they are technically **HTTP Requests**.

### Get: Request data from resource

Diagram

Description automatically generated

* Notice how the GET request is sent in the URL. This means a few things:
  + GET request can be bookmarked
  + GET request saved in history
  + GET request can be cached
  + GET request has length limits
* Also GET request can only request data, not modify or update anything.

### Post Request

* What if we want to send information for the specific purpose of updating some backend information?
* **We would** not **want** that information in the URL and our main concern is no receiving information back, but instead **sending information**

Diagram, text

Description automatically generated

If you notice above, you are not sending information along with the URL(like you did in get request) instead you are sending information in HTTP body

Diagram

Description automatically generated

### The need for CSRF token

When hackers try to steal information/money/any valuable resource, they can do it in 2 ways

1. Forgery
   1. By faking a website and luring end user to use it and then steal
2. Phishing
   1. Even here you will be lured to click on a link, thereby you are sending a form which is allowing hackers to steal data

#### How CPRF comes to the rescue

Common thing in above 2 issues is that they need more than one session and if you have a validator which validates the authenticity of user and has a lifetime of one session, then that problem is solved if we have a token which is valid for that particular session and we have CPRF token in django which will take care of this

* We simply remember to provide:
  + **{% csrf\_token %}**
* For more information on Django’s built-in CSRF protection system:
  + **docs.djangoproject.com/en/4.0/ref/csrf**

## how forms work in django

forms work much in a similar way of how models operate

* you will create a forms.py object in app
* you will import forms Class
* then use methods of that class to assign different type of objects that you want to see in html page
  + ex: if you want to use an **input field (first\_name)**with **label (First Name)** with **length 100** in html you can do it in a single line

first\_name = forms.CharField(label = 'First Name', max\_length=100)

* what happens with above code is that it will create a input field of type textInput and the label parameter (‘First Name’ in this example) is used to create a label in html and the max\_length is an additional parameter
* **objects like CharField bundle the label and corresponding input field (such as textarea or radio button) together,** which can be styled at an individual level
* more details about CharField here: <https://docs.djangoproject.com/en/4.1/ref/forms/fields/#charfield>
* similarly there are many such fields (email,date,integer,json…) available at <https://docs.djangoproject.com/en/4.1/ref/forms/fields/>
* the above first\_name is treated as a widget TextInput (input tag in html)

### an example

let us take a car review website where you have a form to take review for users, and redirect them to thank you page

the code of view,forms and html looks like below

views.py

from django.shortcuts import render,redirect

from django.urls import reverse

from .forms import ReviewForm

# Create your views here.

def rental\_review(request):

    if request.method == 'POST':

        form = ReviewForm(request.POST)

        if form.is\_valid():

            print(form.cleaned\_data)

            return redirect(reverse('cars:thank\_you'))

    else:

        form = ReviewForm()

    return render(request,'cars/rental\_review.html',context = {'form':form})

def thank\_you(request):

    return render(request,'cars/thank\_you.html')

forms.py

from django import forms

class ReviewForm(forms.Form):

    first\_name = forms.CharField(label = 'First Name', max\_length=100)

    last\_name = forms.CharField(label = 'Last Name', max\_length = 100)

    email = forms.EmailField(label = 'Email')

    review = forms.CharField(label = 'Please enter your review here')

car\_review.html

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>Rental Review</h1>

    <form action="" method="post">

        {% csrf\_token %}

        {{form}}

        <input type="submit" name="submit" id="submit">

    </form>

</body>

</html>

### Customizing the django form according to need in html template

There are 3 important points/divisons related to it:

1. initially the form does not look appealing as all elements are treated sequential, and upon resizing, input field and tag break in to different elements, to avoid that you can use jinja templating to teach every pair as a html models such as **table, para, or an ul**
   1. {{form.as\_table}}
   2. {{form.as\_ul}}
   3. {{form.as\_p}}
2. Only fetching important/required elements is also possible with form element in html using jinja templating

<body>

    <h1>Rental Review</h1>

    <form action="" method="post">

        {% csrf\_token %}

        {{form.first\_name.label\_tag}} {{form.first\_name}}

        <div>

        {{form.last\_name.label\_tag}} {{form.last\_name}}

    </div>

        <input type="submit" name="submit" id="submit">

    </form>

</body>

1. This is perhaps the important division/approach, where you can use for loop and apply same **html** **model such as para or table or any other model.** This is not styling and styling is dealt later

<body>

    <h1>Rental Review</h1>

    <div class="container">

        <form action="" method="post">

            {% csrf\_token %}

            {% for field in form%}

                <div class="mb-3">

                    {{field.label\_tag}}

                </div>

                {{field}}

            {% endfor %}

            <input type="submit" name="submit" id="submit">

        </form>

    </div>

</body>

## Django form widgets and styling

Widget: a django’s representation of HTML input (a beautiful concise explanation of widget). i.e, to get all the functionalities of an input tag in html, there are wide variety of methods that are available using form, [here](#widget_example) you can see that first\_name, last\_name and review are of CharField, therefore they are interpreted as input tags of type text, but you need review to be a text area, for that you can create a Textarea widget for that CharField method of forms class and to further pass different parameters you can use **attrs** keyword (a dictionary), detailed explanation [here](#customize_style_widget)

* Recall that a Form Field inside forms.py ends up generating a Django **widget** which in turn renders the actual HTML form input/label tags.
* To have more control over styling and presentation, we can access widget attributes.
* We’ll begin by linking a static files directory to hold our custom CSS files:
  + Create **app/static/app/custom.css** file
  + Load static directory in .html
  + Link static CSS file connection
  + Run migrate to load new app in settings.py file
* How we link this css file to the template is a bit different and can be easily understood with the below example

Example:

forms.py

from django import forms

class ReviewForm(forms.Form):

    first\_name = forms.CharField(label = 'First Name', max\_length=100)

    last\_name = forms.CharField(label = 'Last Name', max\_length = 100)

    email = forms.EmailField(label = 'Email')

    review = forms.CharField(label = 'Please enter your review here')

custom.css

.myform{

    border: 5px dashed red;

}

rental\_review.html

note: notice that to link custom.css to this html you need to do a {%load static%} and then use link tag in head using static keyword

<link rel="stylesheet" href="{% static 'cars/custom.css' %}">

{%load static %}

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <!-- CSS only -->

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">

<!-- JavaScript Bundle with Popper -->

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p" crossorigin="anonymous"></script>

<link rel="stylesheet" href="{% static 'cars/custom.css' %}">

    <title>Document</title>

</head>

<body>

    <h1>Rental Review</h1>

    <div class="container myform">

        <form action="" method="post">

            {% csrf\_token %}

            {% for field in form%}

                <div class="mb-3">

                    {{field.label\_tag}}

                </div>

                {{field}}

            {% endfor %}

            <input type="submit" name="submit" id="submit">

        </form>

    </div>

</body>

</html>

Styling and customizing particular widgets without unpacking them like we did in [step 2](#_Customizing_the_django):

forms.py

look at the review variable, you can see that particular CharField object is transformed in to widget by using the **widget** keyword and then you are passing ‘myform’ styling for class key and you can also see rows and cols, that loosely translates to the tag below the code

from django import forms

class ReviewForm(forms.Form):

    first\_name = forms.CharField(label = 'First Name', max\_length=100)

    last\_name = forms.CharField(label = 'Last Name', max\_length = 100)

    email = forms.EmailField(label = 'Email')

    review = forms.CharField(label = 'Please enter your review here'

                            ,widget = forms.Textarea(attrs = {'class':'myform','rows':'5','cols':'5'}))

Above review variable assigned object loosely translates to something like below

<textarea name="temp" id="temp" cols="30" rows="10" class="myform"></textarea>

## ModelForms

The flow of control that happens is client <-> UI <-> Template <-> Views.py & urls.py <->models <->DB

Browser controls:

client <-> UI <-> Template

But the flow we control inside of django is:

Template(template/ folder and forms.py) <-> Views.py and urls.py <-> models

Python controls:

models <->DB

the flow controlled using django is a bit repetitive for every project so that django came up with a class called ModelForm class **which creates a form with fields which are connected to each model field.** (that means it gets fields of form directly from models.py without you having the need to explicitly mention them in forms.py file, check the “\_\_all\_\_” related code below)

Linking fields in a model, so that they can be viewed/updated dynamically in front end is a tedious task initially because you need to declare those fields in models.py and also you needed to declare them in views.py and you needed to link them so that they are viewed in front end and when user sends data, it is updated in the backend. But using django ModelForms will let you declare fields in models.py and update them dynamically by using a class called Meta in forms.py

Meta class is used to set custom view of fields in admin via classbased functions (discussed later) in views.py, it is also used to set custom view in fields of forms.py, Custom view means ordering the fields as per our choice in admin view or setting error messages or including only particular fields in forms.py

forms.py

from django import forms

from .models import Review

from django.forms import ModelForm

class ReviewForm(ModelForm):

    class Meta:

        model = Review

        fields = ("first\_name","last\_name",'stars')

        # fields = ["first\_name","last\_name","stars"]

models.py

from django.db import models

# Create your models here.

class Review(models.Model):

    first\_name = models.CharField(max\_length=30)

    last\_name = models.CharField(max\_length=30)

    stars = models.IntegerField()

views.py

from django.shortcuts import render,redirect

from django.urls import reverse

from .forms import ReviewForm

# Create your views here.

def rental\_review(request):

    if request.method == 'POST':

        form = ReviewForm(request.POST)

        if form.is\_valid():

            # print(form.cleaned\_data)

            form.save()

            return redirect(reverse('cars:thank\_you'))

    else:

        form = ReviewForm()

    return render(request,'cars/rental\_review.html',context = {'form':form})

def thank\_you(request):

    return render(request,'cars/thank\_you.html')

### Django model form

#### Django model form customization

There are lot of customizations to django model forms and we will only discuss about a couple of the available customizations below

Check the below example:

models.py

from django.db import models

from django.core.validators import MinValueValidator, MaxValueValidator

# Create your models here.

class Review(models.Model):

    first\_name = models.CharField(max\_length=30)

    last\_name = models.CharField(max\_length=30)

    stars = models.IntegerField(validators = [MinValueValidator(1),MaxValueValidator(5)]) #so, if you try to give rating above 5, it will not take it and will give a Http200 and reloads to the same page again as it is how it is declared in views.py, also as it is a wrong entry, it wont be admitted to database  or admin view

forms.py

from django import forms

from .models import Review

from django.forms import ModelForm

class ReviewForm(ModelForm):

    class Meta:

        model = Review

        fields = "\_\_all\_\_"

        labels = {

            'first\_name' : 'enter your first name',

            'last\_name' : 'your last name here',

            'stars' : 'please leave us with a rating'

        }

        error\_messages = {

            'stars' : {

                'min\_value' : 'min value cannot be less than 1',

                'max\_value' : 'max value cannot be more than 5'

            }

        }

rental\_review.html

    <body>

        <h1>Rental Review</h1>

        <div class="container">

            <form action="" method="post">

                {% csrf\_token %}

                {% for field in form%}

                    <div class="mb-3">

                        {{field.label\_tag}}

                    </div>

                    {{field}}

                    {{field.errors}}

                {% endfor %}

                <br><br>

                <input type="submit" name="submit" id="submit">

            </form>

        </div>

    </body>

views.py

from django.shortcuts import render,redirect

from django.urls import reverse

from .forms import ReviewForm

# Create your views here.

def rental\_review(request):

    if request.method == 'POST':

        form = ReviewForm(request.POST)

        if form.is\_valid():

            # print(form.cleaned\_data)

            form.save()

            return redirect(reverse('cars:thank\_you'))

    else:

        form = ReviewForm()

    return render(request,'cars/rental\_review.html',context = {'form':form})

def thank\_you(request):

    return render(request,'cars/thank\_you.html')

when you look at all the files mentioned above, you would have seen the use of validators in models.py and models work very well with forms and does lot of the heavy lifting like displaying an automatic error message if you try to give a value beyond validator range

fields = “\_\_all\_\_” :

listing the fields in UI is our choice, we can only specify a list of fields that you want and pass as a list to above fields parameter or just mention “\_\_all\_\_”, so that every field in models.py is automatically listed in UI

labels (in forms.py) will take a dictionary as an input which will have field\_names in models.py as keys and **alternate text you want to display as label will be a value**

error\_messages is a dictionary of dictionaries, **which will have field name as a key at outer level and validator name as a key at inner level and corresponding value will be the custom message that you want to display on UI for end user**

field.errors will let the custom/automatic error messages of validators get displayed on the screen

#### built in field documentations

there’s a lot more than what the above example explains, there are a lot of builtin fields available and also lot of validations available for each field, and they are very much easy to understand explanations

documentation link: https://docs.djangoproject.com/en/4.1/ref/forms/fields/

ex link for above discussed example: <https://docs.djangoproject.com/en/4.1/ref/forms/fields/#integerfield>

# Class Based Views (CBV)

Similar to classes of forms and models (which you use to create custom objects to achieve you requirements based functionalities), view class system is there and it is very powerful for **quickly rendering class based views**

## Django developer’s own reasoning from the documentation on why to use class based views

* *Writing web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but web developers also experience this boredom at the view level.*
* *Django’s generic views were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.*

Django CBVs (Class Based Views) come with many pre-built generic class views for common tasks, such as listing all the values for a particular model in a database (ListView) or creating a new instance of a model object (CreateView).

To put in simple words, you create a class, then through attributes, you connect it to templates or forms or models

## Template view

This involves connecting view to a template

This is a generic view under a class-based views

### When to use template views

If you know that most of the work is to be done on template side of things, then we will use TemplateView

### Example:

In the below you are creating a class of HomeView and Inheriting TemplateView and passing template path to template\_name variable (the variable name should be as it is and we should not be using other variable)

In urls.py you need to import the class and then use HomeView.as\_view() because it can only accept functions and as\_view() method converts class to function

Views.py

from django.shortcuts import render

from django.views.generic import TemplateView

# Create your views here.

# def home\_view(request):

#     return render(request,'classroom/home.html')

class HomeView(TemplateView):

    template\_name = 'classroom/home.html'

Urls.py

from django.urls import path

from .views import HomeView,ThankYou

app\_name = 'classroom'

urlpatterns =[

    path('',HomeView.as\_view(),name='home')

]

## Form View using class-based views

Earlier you saw how you can use view functions (of forms) connect to models and get to display all the forms of the fields automatically [here](#_ModelForms) , also you can implement that with the help of class based views

Consider the following example

views.py

from django.shortcuts import render

from django.views.generic import TemplateView,FormView

from classroom.forms import ContactForm

from django.urls import reverse, reverse\_lazy

# Create your views here.

class ContactFormView(FormView):

    form\_class = ContactForm

    template\_name = 'classroom/contact.html'

    #if the data is filled and submitted successfully you may have to route another html page, so you need success\_url

    # success\_url = '/classroom/thank\_you' #an url not a html

    success\_url = reverse\_lazy('classroom:thank\_you')

    #when the form is success you need to save or print the data

    def form\_valid(self,form):

        #here form is nothing but the object which holds the data that is typed in form, this is the same form that is used to render data in html file {{form.as\_p}}

        # form.save() #if you want to save the data (submitted in the form) to backend

        print(form.cleaned\_data)

        #similar to ContactForm(request.POST) in function based view

        return super().form\_valid(form) # this is passing POST request to the form and create a new instance of the form

forms.py

from django import forms

class ContactForm(forms.Form):

    name = forms.CharField()

    message = forms.CharField(widget = forms.Textarea)

urls.py

### points to note

1. here we used class based views (we declared a class in views.py and used in urls.py)
2. note the reverselazy() function, it is similar to reverse() function and only difference is that when we are using variable success\_url we use reverselazy, if we want to implement reverse function inside a function, then we use reverse()
3. we can also assign url to success\_url variable directly an url, ex: success\_url = '/classroom/thank\_you'
4. the difference between using function based view and class based view in forms is that, class based view comes with a lot of automation and it needs very less coding
5. also forms with model views with generic templates is going to be much lightweight than this and we can see in later parts of this documentation

## Model based CBV’s

No matter which type of data object (for example person object: it has characteristics like name, age, address, number), you have to **create instance (**creating person object with details**),** **Detail instance (**detailed view for a person object**), Update instance** (update person details of instance)**, Delete instance (**remove data of person**)and list instance (**list a bunch of people details**)**

**Django provides CBV’s that automatically create forms, views and context objects for pre-defined template names by simply being connected to a model.**

**These classes require just a few attributes and they automatically do the work for you**

**IMPORTANT POINT:**

**CBV’s require to follow a particular naming convention for templates, if you are about to create a teacher model, then you need to name the template as teacher\_form.html, if you are about to create a employee model, then you need to name the template as employee\_form.html**

### CreateView

**CreateView creates a view automatically** to **create an instance of a model** by the client user

The below example has a detailed explanation with comments of what is needed and how it is working

Views.py

from django.shortcuts import render

from classroom.forms import ContactForm

from django.urls import reverse, reverse\_lazy

from django.views.generic import TemplateView,FormView,CreateView

from classroom.models import Teacher

# Create your views here.

class TeacherCreateView(CreateView): #it can have any name it wants, but having naming convention of <class name in models.py><type of view> ex: TeacherCreateView will let us understand what model and what class based view are we using

    # a "model create view" requires model, template, fields that you want in template, successURL to redirect to

    model = Teacher

    #when you pass data to model variable python automatically looks for model\_form.html (all lower case) in this case is teacher\_form.html

    fields = "\_\_all\_\_" #else you can also pass a list of field names from the class in models.py

    success\_url = reverse\_lazy('classroom:thank\_you')

    #when you hit submit button, it automatically performs .save() and loads the entry to the database

models.py

from django.db import models

# Create your models here.

class Teacher(models.Model):

    first\_name = models.CharField(max\_length=30)

    last\_name = models.CharField(max\_length=30)

    subject = models.CharField( max\_length=30)

def \_\_str\_\_(self):

    return f'{self.first\_name} {self.last\_name} teaches {self.subject}'

urls.py

from django.urls import path

from .views import HomeView,ThankYou,ContactFormView, TeacherCreateView

app\_name = 'classroom'

urlpatterns =[

    path('',HomeView.as\_view(),name='home'),

    path('thank\_you/',ThankYou.as\_view(),name='thank\_you'),

    path('contact/',ContactFormView.as\_view(),name='contact'),

    path('create\_teacher/', TeacherCreateView.as\_view(), name='create\_teacher')

]

teacher\_form.html

<h1>Teacher form (FormView)</h1>

<form method="post">

    {% csrf\_token %}

    {{form.as\_p}}

    <input type="submit" name="submit" id="submit" value="submit">

</form>

#### Important points to note

1. you have created a form without mentioning anything in forms.py
2. you only wrote 4 lines of code in views.py file and had completed the requirement of creating a new instance
3. Here the html template that is being used under the hood is teacher\_form.html and it is a mandatory thing to follow the naming convention of <object>\_form.html
4. **On clicking submit in <object>\_ form.html it will by default redirect to <object>\_detail.html if success\_url is not defined**
   1. **But for <object>\_detail to load you need to have a field which will help you load that particular objects detail view, it can also be done automatically by defining the below piece of code in corresponding model in models.py**

def get\_absolute\_url(self):

        return reverse("<object>\_detail", kwargs={"pk": self.pk})

## ListView

**You can fetch data entries in database as objects via models in the form of list by using list view, also you can POST (add) an object to the list**

It is easy and few lines of code will do the work

models.py

from django.db import models

# Create your models here.

class Teacher(models.Model):

    first\_name = models.CharField(max\_length=30)

    last\_name = models.CharField(max\_length=30)

    subject = models.CharField( max\_length=30)

def \_\_str\_\_(self):

    return f'{self.first\_name} {self.last\_name} teaches {self.subject}'

views.py

from django.shortcuts import render

from classroom.forms import ContactForm

from django.urls import reverse, reverse\_lazy

from django.views.generic import TemplateView,FormView,CreateView,ListView

from classroom.models import Teacher

# Create your views here.

class TeacherListView(ListView):

    model = Teacher

    #it looks for a html template in the form of <model>\_list.html (all alphabets in lower case) in this case, it is teacher\_list.html under templates/<app\_name>/ directory

    context\_object\_name = 'teacher\_list' #this parameter is optional and is used mainly for better understanding

    # by default context\_object name = object\_list, it holds all the object instantiations of Teacher model (each object will have parameters first\_name,last\_name,subject - parameters which are declared in model class this object\_name is used in html template to fetch list objects, if it is reassigned like above you can use either object\_list or teacher\_list)

    queryset = Teacher.objects.order\_by('subject') #by default queryset will have Teacher.objects.all() which will fetch all entries in database via models, whenever you want to fetch results based on some condition like list all teacher objects in their ascending order of firstnames, you can use Teachers.objects.order\_by('first\_name)

urls.py

from django.urls import path

from .views import (HomeView,ThankYou,ContactFormView, TeacherCreateView,

                    TeacherListView)

app\_name = 'classroom'

urlpatterns =[

    path('',HomeView.as\_view(),name='home'),

    path('thank\_you/',ThankYou.as\_view(),name='thank\_you'),

    path('contact/',ContactFormView.as\_view(),name='contact'),

    path('create\_teacher/', TeacherCreateView.as\_view(), name='create\_teacher'),

    path('list\_teacher/', TeacherListView.as\_view(),name = 'list\_teacher')

]

teacher\_list.html

<h1>List of Teachers (ListView)</h1>

<ul>

    {% for teacher in teacher\_list %}

        <li>{{teacher.first\_name}}  {{teacher.last\_name}} teaches {{teacher.subject}}</li>

    {%endfor%}

</ul>

### Points to note

## DetailView

The main purpose of a detailed view is to view single instance of a particular entry in a model, you can **get** the details of an object, **put** (update) update the details of object, **delete** deletea particular object

To view a particular instance you need to pass some data(a unique identifier such as a primary key) related to that particular instance so that django can identify and return data

In the below example to achieve the usecase of detailview we are listing all the objects in the database will be listed as a hyperlinks in list\_teacher html page which is using ListView (discussed above) under the hood and upon clicking each link(list item), it will take the corresponding id and will reroute to teacher\_detail/<id>

Views.py

from django.shortcuts import render

from classroom.forms import ContactForm

from django.urls import reverse, reverse\_lazy

from django.views.generic import TemplateView,FormView,CreateView,ListView,DetailView

from classroom.models import Teacher

# Create your views here.

class TeacherCreateView(CreateView): #it can have any name it wants, but having naming convention of <class name in models.py><type of view> ex: TeacherCreateView will let us understand what model and what class based view are we using

    # a "model create view" requires model,template,fields that you want in template, successURL to redirect to

    model = Teacher

    #when you pass data to model variable python automatically looks fot model\_form.html (all lower case) in this case is teacher\_form.html

    fields = "\_\_all\_\_" #else you can also pass a list of field names from the class in models.py

    success\_url = reverse\_lazy('classroom:thank\_you')

    #when you hit submit button,it automatically performs .save() and loads the entry to the database

class TeacherListView(ListView):

    model = Teacher

    #it looks for a html template in the form of <model>\_list.html (all alphabets in lower case) in this case, it is teacher\_list.html under templates/<app\_name>/ directory

    context\_object\_name = 'teacher\_list' #this parameter is optional and is used mainly for better understanding

    # by default context\_object name = object\_list, it holds all the object instantiations of Teacher model (each object will have parameters first\_name,last\_name,subject - parameters which are declared in model class this object\_name is used in html template to fetch list objects, if it is reassigned like above you can use either object\_list or teacher\_list)

    # queryset = Teacher.objects.order\_by('subject') #by default queryset will have Teacher.objects.all() which will fetch all entries in database via models, whenever you want to fetch results based on some condition like list all teacher objects in their ascending order of firstnames, you can use Teachers.objects.order\_by('first\_name')

class TeacherDetailView(DetailView):

    #return only one model entry using PK

    model = Teacher

    #it needs a html and it looks for <model>\_detail.html (lowercase) in this case, it is teacher\_detail.html

    # PK --> for every PK send its corresponding data as a context object, in this case {{teacher}}

urls.py

from django.urls import path

from .views import (HomeView,ThankYou,ContactFormView, TeacherCreateView,

                    TeacherListView,TeacherDetailView)

app\_name = 'classroom'

urlpatterns =[

    path('',HomeView.as\_view(),name='home'),

    path('thank\_you/',ThankYou.as\_view(),name='thank\_you'),

    path('contact/',ContactFormView.as\_view(),name='contact'),

    path('create\_teacher/', TeacherCreateView.as\_view(), name='create\_teacher'),

    path('list\_teacher/', TeacherListView.as\_view(),name = 'list\_teacher'),

    path('teacher\_detail/<int:pk>', TeacherDetailView.as\_view(),name='detail\_teacher') #as you are going for a detailed view of a particular instance, you need a differentiating input which will lead you to that particlar instance

    #the above path object translates to host:port/classroom/teacher\_detail/1

]

teacher\_list.html

<h1>List of Teachers (ListView)</h1>

<ul>

    {% for teacher in teacher\_list %}

        <li>

            <a href="/classroom/teacher\_detail/{{teacher.id}}">{{teacher.first\_name}}  {{teacher.last\_name}}</a>

            <a href="../../classroom/teacher\_detail/{{teacher.id}}">{{teacher.first\_name}}  {{teacher.last\_name}}</a>

            <a href="../teacher\_detail/{{teacher.id}}">{{teacher.first\_name}}  {{teacher.last\_name}}</a>

            <a href="classroom/teacher\_detail/{{teacher.id}}">{{teacher.first\_name}}  {{teacher.last\_name}}</a>

        </li>

    {%endfor%}

</ul>

teacher\_detail.html

<h1>detail view for a teacher</h1>

{{teacher}}

<!-- here above you are passing the model name i.e., Teacher in lower case -->

### Points to note

1. We know that in ListView we will list all the **objects** (which will have id, teacher first and last\_name, subject)in the model (which are fetched from backend database by django automatically), we display them as list items <li> and convert them in to anchor tags(hyperlinks) which upon clicking(id will be fetched from object) will be converted as a link and will be re routed to DetailedView and its html template will be rendered
   1. Upon using createView to put some detail in to database
2. In teacher\_list.html you can see 4 anchor tags which serve the same purpose, try to observer the differences between them, 1st and 4th will look the same except for starting forward slash, that forwardslash makes the 1st anchor tag an absolute path, and 4th anchor tag a relative path and the first anchor tag works and the 4th anchor tag wont

## UpdateView

This is kind of a mix between DetailView and CreateView,

Common thing between CreateView and UpdateView is, it shares some of the attributes of the CreateView (fields attribute)

And the common thing between DetailView and UpdateView is that, it needs a particular integer so that it can fetch the data and also send the updated data

An example:

Views.py

from django.shortcuts import render

from classroom.forms import ContactForm

from django.urls import reverse, reverse\_lazy

from django.views.generic import TemplateView,FormView,CreateView,ListView,DetailView,UpdateView

from classroom.models import Teacher

# Create your views here.

class TeacherUpdateView(UpdateView):

    model = Teacher

    # fields = ['last\_name'] #you can restrict the end user to only update a certain fields(which are in models.py), by passing them as a list of elements or you can use \_\_all\_\_

    fields = "\_\_all\_\_" #this lists all fields in models.py up for updation

    success\_url = reverse\_lazy('classroom:list\_teacher')

form.py

<h1>Teacher form (FormView)</h1>

<form method="post">

    {% csrf\_token %}

    {{form.as\_p}}

    <input type="submit" name="submit" id="submit" value="submit">

</form>

teacher\_list.html

<h1>List of Teachers (ListView)</h1>

<ul>

    {% for teacher in teacher\_list %}

        <li>

            <a href="/classroom/teacher\_detail/{{teacher.id}}">{{teacher.first\_name}}  {{teacher.last\_name}}</a>

            <ul>

                <li>

                    <a href="/classroom/update\_teacher/{{teacher.id}}">update teacher information for {{teacher.first\_name}}</a>

                </li>

            </ul>

        </li>

    {%endfor%}

</ul>

urls.py

from django.urls import path

from .views import (HomeView,ThankYou,ContactFormView, TeacherCreateView,

                    TeacherListView,TeacherDetailView,TeacherUpdateView)

app\_name = 'classroom'

urlpatterns =[

    path('',HomeView.as\_view(),name='home'),

    path('thank\_you/',ThankYou.as\_view(),name='thank\_you'),

    path('contact/',ContactFormView.as\_view(),name='contact'),

    path('create\_teacher/', TeacherCreateView.as\_view(), name='create\_teacher'),

    path('list\_teacher/', TeacherListView.as\_view(),name = 'list\_teacher'),

    path('teacher\_detail/<int:pk>', TeacherDetailView.as\_view(),name='detail\_teacher'), #as you are going for a detailed view of a particular instance, you need a differentiating input which will lead you to that particlar instance

    #the above path object translates to host:port/classroom/teacher\_detail/1

    path('update\_teacher/<int:pk>',TeacherUpdateView.as\_view(),name='update\_teacher')

]

### An overview of above example

You will create a hyperlinks for update\_teacher/<int> in teacher\_list.html, upon clicking, it redirects to TeacherUpdateView will use forms.py and lists all the fields the programmer want the end user to update and then upon clicking submit, the user is again redirected to teacher\_list.html

### Point to note

1. Here the html template that is being used is teacher\_form.html and it is a mandatory thing to follow the naming convention of <object>\_form.html

## DeleteView

It is similar to DetailView where we will be dealing with one object of a model by fetching an attribute which will uniquely identifies the object

### Overview of the below example

We start from list page, where we give user the option to delete object with a hyperlink to the teacher\_confirm\_delete.html from where we link it to TeacherDeleteView class in views.py

Urls.py

from django.urls import path

from .views import (HomeView,ThankYou,ContactFormView, TeacherCreateView,

                    TeacherListView,TeacherDetailView,TeacherUpdateView,

                    TeacherDeleteView)

app\_name = 'classroom'

urlpatterns =[

    path('',HomeView.as\_view(),name='home'),

    path('thank\_you/',ThankYou.as\_view(),name='thank\_you'),

    path('contact/',ContactFormView.as\_view(),name='contact'),

    path('create\_teacher/', TeacherCreateView.as\_view(), name='create\_teacher'),

    path('list\_teacher/', TeacherListView.as\_view(),name = 'list\_teacher'),

    path('teacher\_detail/<int:pk>', TeacherDetailView.as\_view(),name='detail\_teacher'), #as you are going for a detailed view of a particular instance, you need a differentiating input which will lead you to that particlar instance

    #the above path object translates to host:port/classroom/teacher\_detail/1

    path('update\_teacher/<int:pk>',TeacherUpdateView.as\_view(),name='update\_teacher'),

    path('delete\_teacher/<int:pk>',TeacherDeleteView.as\_view(),name='delete\_view')

]

teachers\_list.html

<h1>List of Teachers (ListView)</h1>

<ul>

    {% for teacher in teacher\_list %}

        <li>

            <a href="/classroom/teacher\_detail/{{teacher.id}}">{{teacher.first\_name}}  {{teacher.last\_name}}</a>

            <ul>

                <li>

                    <a href="/classroom/update\_teacher/{{teacher.id}}">update teacher information for {{teacher.first\_name}}</a>

                </li>

                <li>

                    <a href="/classroom/delete\_teacher/{{teacher.id}}">delete teacher information for {{teacher.first\_name}} permanently</a>

                </li>

            </ul>

        </li>

    {%endfor%}

</ul>

views.py

from django.shortcuts import render

from classroom.forms import ContactForm

from django.urls import reverse, reverse\_lazy

from django.views.generic import TemplateView,FormView,CreateView,ListView,DetailView,UpdateView,DeleteView

from classroom.models import Teacher

# Create your views here.

class TeacherListView(ListView):

    model = Teacher

    #it looks for a html template in the form of <model>\_list.html (all alphabets in lower case) in this case, it is teacher\_list.html under templates/<app\_name>/ directory

    context\_object\_name = 'teacher\_list' #this parameter is optional and is used mainly for better understanding

    # by default context\_object name = object\_list, it holds all the object instantiations of Teacher model (each object will have parameters first\_name,last\_name,subject - parameters which are declared in model class this object\_name is used in html template to fetch list objects, if it is reassigned like above you can use either object\_list or teacher\_list)

    # queryset = Teacher.objects.order\_by('subject') #by default queryset will have Teacher.objects.all() which will fetch all entries in database via models, whenever you want to fetch results based on some condition like list all teacher objects in their ascending order of firstnames, you can use Teachers.objects.order\_by('first\_name')

class TeacherDeleteView(DeleteView):

    # it is a form which will have delete button, whether to delete it or not

    model = Teacher

    success\_url = reverse\_lazy('classroom:list\_teacher')

    # default template it looks for is <model>\_confirm\_delete.html -> teacher\_confirm\_delete.html

teacher\_confirm\_delete.html

<h1>are you sure you want to delete?</h1>

<h2>{{teacher}}</h2>

<form method="post">

    {% csrf\_token %}

    <input type="submit" value="Confirm delete">

</form>

### An important note

It uses <model>\_confirm\_delete.html file which has to be with same name

# User Authentication and Sessions

Here we see how to handle Users and Sessions with Django’s built-in tools.

Django provides almost everything you need to handle user authentication pages to handle things like **login, logout and password** management all out of the box

urls.py at project level

from django.contrib import admin

from django.urls import path,include

from django.views.generic import RedirectView

urlpatterns = [

    path('admin/', admin.site.urls),

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

    path('',RedirectView.as\_view(url = 'catalog/')),

    path('accounts/',include('django.contrib.auth.urls')) #

]

with this accounts/ app you have already few authentication pages readily available like login, logout, change password, reset password ex: accounts/login is readily available without us coding anything for it, **but you just need to add html template (page) for it,** you have to **save the templates at <project folder>/templates/<name of choice>** because **registration and logging in, really happens at a site level** not at application level

Below are the routes that are to be added with templates to get corresponding functionality, down the lane in this documentation we are discussing about login template, steps needed for its connection and few other things

* url: accounts/ login/ [name='login'] should have html file with name login.html
* url: accounts/ logout/ [name='logout'] should have html file with name logged\_out.html (users will be here when they click on logout button on a html page)
* url: accounts/ password\_change/ [name='password\_change']
* url: accounts/ password\_change/done/ [name='password\_change\_done']
* url: accounts/ password\_reset/ [name='password\_reset']
* url: accounts/ password\_reset/done/ [name='password\_reset\_done']
* url: accounts/ reset/<uidb64>/<token>/ [name='password\_reset\_confirm']They will be under accounts sub domain in project level urls.py

to make it visible at settings.py file level we need to include that in APP\_DIRS in settings.py file

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [os.path.join(BASE\_DIR,'templates')],

        'APP\_DIRS': True,

        'OPTIONS': {

            'context\_processors': [

                'django.template.context\_processors.debug',

                'django.template.context\_processors.request',

                'django.contrib.auth.context\_processors.auth',

                'django.contrib.messages.context\_processors.messages',

            ],

        },

    },

]

<project folder>/templates/registration/login.html – used to check whether a user is logged in or not, if not logged in, I need them to have a form to login, we need to have a couple of things on login form

1. Check errors
2. If the user is **logged in** and is not authenticated to visit a page (he got a stream query from DJANGO called **next**)
   1. If not authenticated, he will be displayed a message- something like, “*you don’t have permission to view this page”*
3. If the user directly tries to access a page **without logging** in, then he is redirected to a login form

{% if form.errors %}

<p> your username or password was incorrect, please try again later</p>

{% endif %}

{% if next %}

    {% if user.is\_authenticated %}

        <p> you dont have permission for this page</p>

    {% else %}

        <p> please login to see the page</p>

    {% endif %}

{% endif %}

<form method="post" action="{% url 'login' %}">

    {% comment %} 'login' will redirect to localhost:8000/accounts/profile, this redirect is called post login redirect, if you want to have a custom url, you can go to settings.py and give LOGIN\_REDIRECT\_URL = '/' {% endcomment %}

    {% csrf\_token %}

    {{form.username.label\_tag}}

    {{form.username}}

    {{form.password.label\_tag}}

    {{form.password}}

    <input type="submit" value="login">

    <input type="hidden" name="next" value="{{next}}">

</form>

## (applying) User authentication to views (ability to access html page)

Also can be called as **authentication-based restriction**

Till now without logging in, we are able to see all the pages that are associated with views, but generally many of the pages on any website needs the user to be logged in

To apply user authentication protocols to views, there are 2 ways of doing this

* Using decorators for function based views
* Using mixin for class based views

### Letting custom view based on user logged in status - a template side capability

Before we go into authentication-based restriction, look at the below example where we are letting user to access home view, **but it will have different information for users who are logged in and users who did not login**

<!--skeleton html-->

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>home page</h1>

    <p>Total books {{num\_books}}</p>

    <p>Num available {{num\_instances\_available}}</p>

    {% if user.is\_authenticated %}

    <p> you are logged in</p>

    <p> welcome {{user.get\_username}}</p>

    {% else %}

    <p> you are not logged in</p>

    {% endif %}

</body>

</html>

**Index.html**

In the below example we are viewing home page, where we will display different information to both logged in and logged out users and also we have a hyper link (anchor tag) which has the field next={{request.path}} which will let the user logout and get redirected to the same page, because request is an object and it holds the same path it has loaded(therefore gets reloaded)

<!--skeleton html-->

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>home page</h1>

    <p>Total books {{num\_books}}</p>

    <p>Num available {{num\_instances\_available}}</p>

    {% if user.is\_authenticated %}

    <p> you are logged in</p>

    <p> welcome {{user.get\_username}}</p>

    {{request.path}}

    <a href="{% url 'logout' %}?next={{request.path}}">logout here</a>

    {% comment %} here request is the request object that loaded this HTML file, therefore request.path means the (current) url that has loaded this html page  {% endcomment %}

    {% comment %} also space around equal sign "next = {{request.path}}" will lead to error as it will treat it as an actual space in the URL  {% endcomment %}

    {% else %}

    <p> you are not logged in</p>

    <a href="{% url 'login' %}?next={{request.path}}">login here</a>

    {% endif %}

</body>

</html>

#### Why is it powerful

Just based on user login status, we are changing the data **from the html template itself, we** **are not even going in to views or deeper**

### Restricting a function-based view to a user who is not logged in- a view side capability

There might be a few scenarios (like profile page or orders list) where we need the user to be logged in so that the view can generate data based on user login information

Restricting a function-based view to a user using django is as simple as assigning a decorator to it

Views.py

from django.shortcuts import render

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

# Create your views here.

@login\_required

def my\_view(request):

    return render(request,'catalog/my\_view.html')

#### how does it work

when we use a decorator @login\_required, it adds current view after next parameter [localhost:8000/accounts/login/?next=/catalog/my\_view/](http://localhost:8000/accounts/login/?next=/catalog/my_view/) , which will redirect based on the login status of the user, it will redirect

### Restricting a class-based view to a user who is not logged in – a view side capability

Similar to function-based view restriction, a class-based view restriction is also as simple as importing a module and passing it as a parameter

from .models import Book, Genre, Language, Author, BookInstance

from django.views.generic import CreateView,DetailView

from django.contrib.auth.mixins import LoginRequiredMixin

# Create your views here.

class BookCreate(LoginRequiredMixin,CreateView):

    model = Book

    fields = "\_\_all\_\_"

## user authentication forms

**we can use the user object to create users manually using our own forms**

**luckily, Django provides quite a few built in forms and views based off the users class, that’s built in with django**

for now, we know that an object creates using a model is typically an entry in the table (which the current model is representing), an ex: let say there is a model called Person (a class) you defined in models.py so when you do person.objects.get(pk=1) it will return a person object whose ID is 1 and you can almost change attributes of that object or print those details like first\_name or other fields that are mentioned in Person class in models.py

similarly we can get objects of user and do almost all operations we discussed in above example explanation

>>> from django.contrib.auth.models import User

>>> user = User.objects.create\_user('john', 'lennon@thebeatles.com', 'johnpassword')

# At this point, user is a User object that has already been saved

# to the database. You can continue to change its attributes

# if you want to change other fields.

>>> user.last\_name = 'Lennon'

>>> user.save()

>>> u.set\_password('new password')

>>> u.save()

More details on authentication here: <https://docs.djangoproject.com/en/4.1/topics/auth/default/#using-the-django-authentication-system>

Much of code in above link can be used inside the views

There are **forms available to achieve the case of creating views by simply importing corresponding forms**

Ex: if you want to have a user creation (user sign up) page, you can use simply

from django.contrib.auth.forms import UserCreationForm

**but to use forms in views, the approach we followed till now is to create a form in forms.py and then import in views.py and use it in the function you declare in views.py** a detailed explanation [here](#_Django_model_form_1)

but we can use [CreateView](#_CreateView) to piggyback the current process of using imported form by mentioning a few more parameters than a create view already has (ex: we override the form\_class parameter below)

views.py

from django.contrib.auth.forms import UserCreationForm

from django.urls import reverse\_lazy

class SignUpView(CreateView):

    form\_class = UserCreationForm

    success\_url = reverse\_lazy('login')

    template\_name = 'catalog/signup.html'

signup.html

<h1>Signup form</h1>

<form method="post">

    {% csrf\_token %}

    {{form.as\_p}}

    <input type="submit" value="signup" >

</form>

## Using user as a model class

What is the use of using user and his information as a model class?

As we know, model is a direct connection to a database and if you are faced with a scenario where you need to display items that are there in the wishlist of a user, then we need information of user and item objects (because user has items (object) in his Wishlist ) to get that information, we need user and item as models in models.py so that we can fetch the data and display it on web page

You can import user model as a package which will have all user information, and use parameters of admin (ex: username) in models like BookInstance (as a borrower, because, obviously user is the borrower), it is implemented as a piece of code below

It goes like this

1. You will alter BookInstance model class to have a field called Borrower, which will get assigned admin as an value
2. You will create a view in views.py, which will use ListView and LoginRequiredMixin (because it is needed as we want to list the books took by user, so user needs to be logged in) and filter the list of books by user
3. Create a html page mentioned by the view in step 2 and fetch elements returned by corresponding view
4. Create path object in urls.py

models.py

from django.db import models

from django.contrib.auth.models import User

import uuid

class BookInstance(models.Model):

    id = models.UUIDField(primary\_key=True,default=uuid.uuid4)

    book = models.ForeignKey('Book',on\_delete=models.RESTRICT,null=True)#before you delete a book from class Book, you need to retrieve the copies that are already lent to library members, so setting on\_delete=models.RESTRICT suits here

    imprint = models.CharField(max\_length=200)

    due\_back = models.DateField(null=True,blank=True)

    borrower = models.ForeignKey(User, on\_delete = models.SET\_NULL, null=True, blank=True)

    LOAN\_STATUS = ( #this is a tuple of tuples which can be used as a parameter for variable where you want to have input from only a set of choices, check the status attribute below

        ('m','Maintenance'),

        ('o','On Loan'),

        ('a','Available'),

        ('r','Reserved')

    )

    status = models.CharField(max\_length=1,choices=LOAN\_STATUS,blank=True,default = 'm')

    #for the above status attribute, we can only have any one of (m,o,a,r)

    def \_\_str\_\_(self):

        return f'{self.id} ({self.book.title})' #here in this class we are referring to

views.py

from .models import Book, Genre, Language, Author, BookInstance

from django.views.generic import CreateView,DetailView,ListView

from django.contrib.auth.mixins import LoginRequiredMixin

from django.contrib.auth.forms import UserCreationForm

# Create your views here.

class CheckedOutBooksByUser(LoginRequiredMixin,ListView):

    model = BookInstance

    template\_name = 'catalog/profile.html'

    paginate\_by = 5 #list 5 instances per page and have pagination widget at bottom of table

    def get\_queryset(self):

        return BookInstance.objects.filter(borrower = self.request.user) #here we are fetching the list of books and then further filtering them by user ID and then returning it to linked html template, each entry will bear information which will be filled as per the \_\_str\_\_ method in BookInstance class

profile.html

<h1>WElcome to your profile</h1>

<h2>

    {% for book in bookinstance\_list %}

{% comment %} here bookinstance\_list is the inbuilt varaible that we get from ListView in views.py {% endcomment %}

    <p> {{book}} </p>

    {% endfor %}

</h2>

Urls.py

from django.urls import path

from . import views

urlpatterns = [

    path('profile/',views.CheckedOutBooksByUser.as\_view(),name='profile')

]

# Django Linode Deployment

Key requirements for deploying django application to web:

* Anyone can visit our website online.
* We don’t want to concern ourselves about uptime or resiliency.
* Need to support Python/Django.
* Need to connect to cloud server.
* Need to be able to push updates to our code and have version control

# DRF – django Rest Framework

An add on you install in to django, which adds features for building REST API’s

# Using docker with django

## Pros of using docker with django

* Consistent dev and prod environment
* Easier collaboration
* Capture all dependencies as a code
* Easier to clean (when you want to delete all project related code)
* Saves a lot of time

## cons of using docker with django

* VS code unable to access interpreter (as code runs inside a container)
* More difficult to use integrated features

To overcome cons, it is suggested to use terminal

## An example command for running docker related commands in django

docker-compose run --rm app sh -c “python manage.py collectstatic”

### breakdown:

docker-compose – runs a docker command

run – will start a specific container defined in config

--rm – removes a container after it finishes running, recommended when you are running a single command, because that means you don’t have to build up lingering containers

app – is the name of the service

sh -c – passes the command in quotes as a shell command, the command in the quotes is the django command and it runs inside the container

# JWT

JSON Web Token (JWT) is an **open standard** that defines a compact and **self-contained way for securely transmitting information between parties as a JSON object**. This **information can be verified and trusted because it is digitally signed**. JWT **used to create access tokens for an application**. JWT is **good for API authentication**, and **server-to-server authorization**

# Tiger Django KT

## apps

a folder where you can have multiple applications, **each app is a folder under apps**, and has its own set of files (urls.py, apps.py models.py etc)

### testing

we know that each app is a separate folder under apps.py, we need to do testing for each app, so at an app level we have testing folder which will have all the testing python scripts, conftest.py is used to configure all the data that is needed to perform testing, conftest.py file is the first triggered file, if you start testing and other testing files in the folder are executed after it.

## config

folder where the settings.py (a file where you do all the changes related to the project using django) file and other configuration files resides, you will have to go to this folder if you have any configuration changes or

## core

a folder which has generic data (reusable components which can be/is used repeatedly) which is used across the project, to edit a reusable component, core folder is the first destination

## drf(Django rest framework) spectacular

looks at all of your code, aggregates all API’s which you will be able to see and use as a client (called as swaggerUI) to send requests (like you do in postman), using drf spectacular, you can also properly document information related to each API call

we can use @extend\_schema which is a decorator from drf spectacular, which is used in views.py to add nice documentation to API so that we can fetch it and display on client page (for API’s)

to use drf spectacular, you need to update settings.py file by updating installed\_apps list and REST\_FRAMEWORK dict

## project level settings.py

also we will add details of additional db in settings.py so that it can be used for testing purposes

# misc:

## path converter

this is to specify to django that you want to treat a particular variable as string or int or slug or uuid or path

urlpatterns = [

    path('<str:topic>/',views.news\_view,name = 'news\_view'),

]

Here we are using path converter str to inform django to treat topic as a string

## Flow involved from client end when you have views, html templates, models and urls.py involved

Client types url in browser-> that url is checked for corresponding view in urls.py-> in corresponding views.py that function will be executed, and data will be operated (crud operation) and corresponding html template will be loaded in the client browser

## Redirecting a view, so that it can be used as a particular URL

Suppose home page of any website will be www.<website>.com, but you want to use the same view which is being used at the url www.<website>.com/abc (here abc is typically the appeig)to be displayed at www.<website>.com, for this you need to use RedirectView, look at the below code

from django.contrib import admin

from django.urls import path,include

from django.views.generic import RedirectView

urlpatterns = [

    path('admin/', admin.site.urls),

    path(abc/',include(abc.urls')),

    path('',RedirectView.as\_view(url = abc/')),

]

## Path parameter and query parameter

**Query Parameter:** These are appended to the end of the request URL, Query parameters are appended to the end of the request URL, following '?' and listed in key-value pairs, separated by '&' Syntax:

1. ?id=1&type=**new**

**Path Parameters:** These are part of the request URL, which are accessed using the placeholders preceded by ':' Example:

1. /customer/:id

There are two ways of setting **query parameters** on a request in [Postman](https://www.javatpoint.com/postman).

## Error: That port is already in use.

if a particular port is not running, but if terminal is saying, port is already in use, then you need to run the command- sudo fuser -k <port number>/tcp

# Hashtags

#need\_to\_know : topic that needs further research and/or exploration