Django for Python Developers

Web Application basics

A web application, often referred to as a web app, is an interactive computer program built with web technologies (HTML, CSS, JS), which stores (Database, Files) and manipulates data (CRUD), and is used by a team or single user to perform tasks over the internet. CRUD is a popular acronym and is at the heart of web app development. It stands for Create, Read, Update, and Delete. Web apps are accessed via a web browser such as Google Chrome, and often involve a login/signup mechanism.

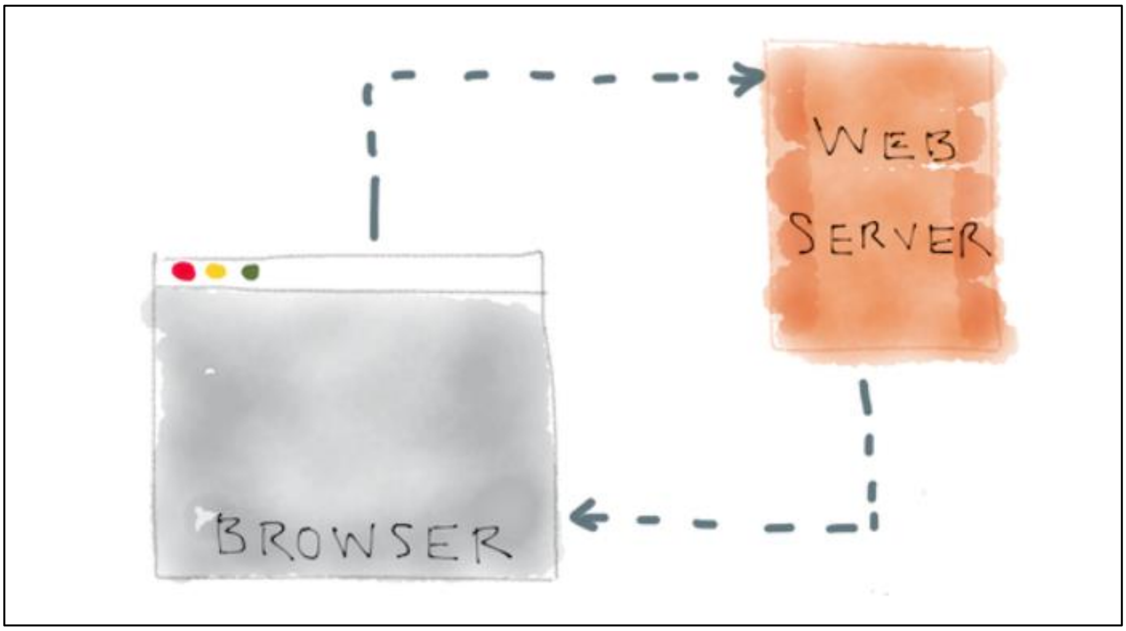
Web applications vs website

The key difference is how we interact with each. Web applications are defined by their **input** - we create, read, update and delete data within a web application. Websites are defined by their **output** - we read the news, marketing information, FAQs on websites.

How websites (and web apps) work

All websites in existence are **hosted**, meaning that all of their code lives on computers, known as web servers. Web servers are basically just computers that have certain software installed on them which enable them to **serve** websites (or send website data back to you, the user, when you visit them). But any computer can be turned into a web server.

When you open your web browser and type in a URL, your request to view the designated website is ultimately sent to a web server, as you can see in the figure below.



That web server packages up all of the necessary information for the website and delivers it to your web browser using **HTTP**: the method used for all web-based communications. Note that the http:// that prefixes all websites' URLs designates that you're making an HTTP **request**.

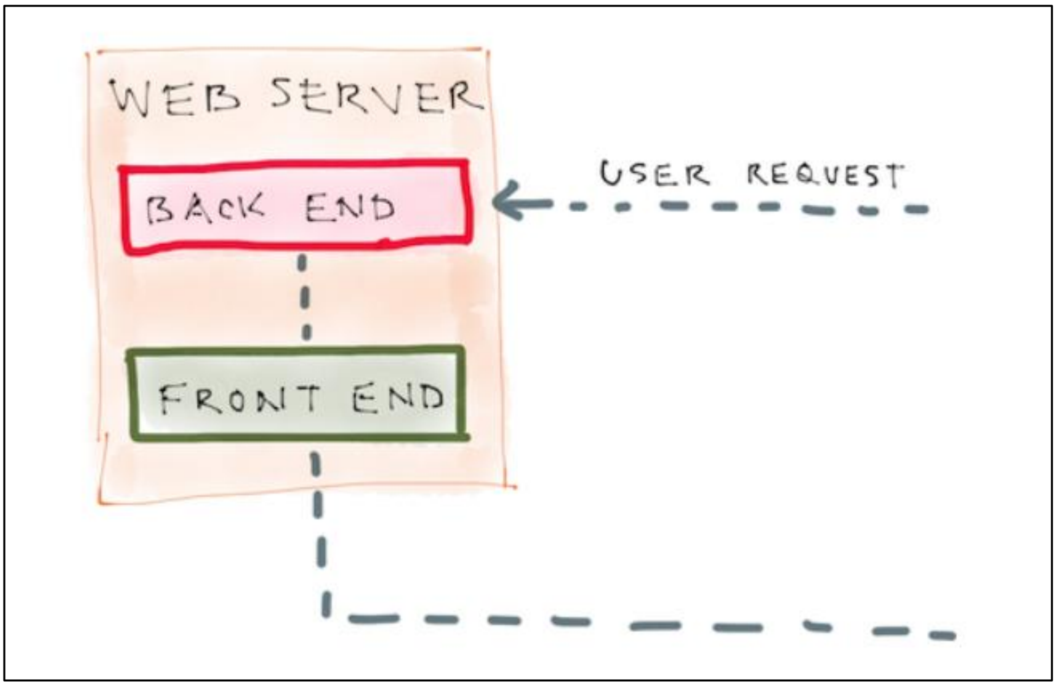
A web app functions just like a restaurant

Web apps are comprised of two parts—the **front end** and the **back end**—that are constantly working together to perform all necessary actions and deliver all desired results.

Just as a restaurant's front-of-the-house is responsible for a presenting the user with an expected experience and results (the food), a web app's front end is responsible for creating the pages and displays a user sees within their web browser.

Just as all customer requests are handled in the front-of-the-house of a restaurant, a web app's front end handles all user interactions and requests. In a restaurant you may be interacting with a waitperson, whereas in a web app you may be interacting with buttons, text boxes, or other controls.

In a restaurant, when an order is placed by a customer in the front-of-the-house, that order is sent to the back-of-the-house so the desired dish can be prepared. As you can see in the figure below, when a user makes a request in a web app, that request is sent to the application's back end to be handled accordingly.

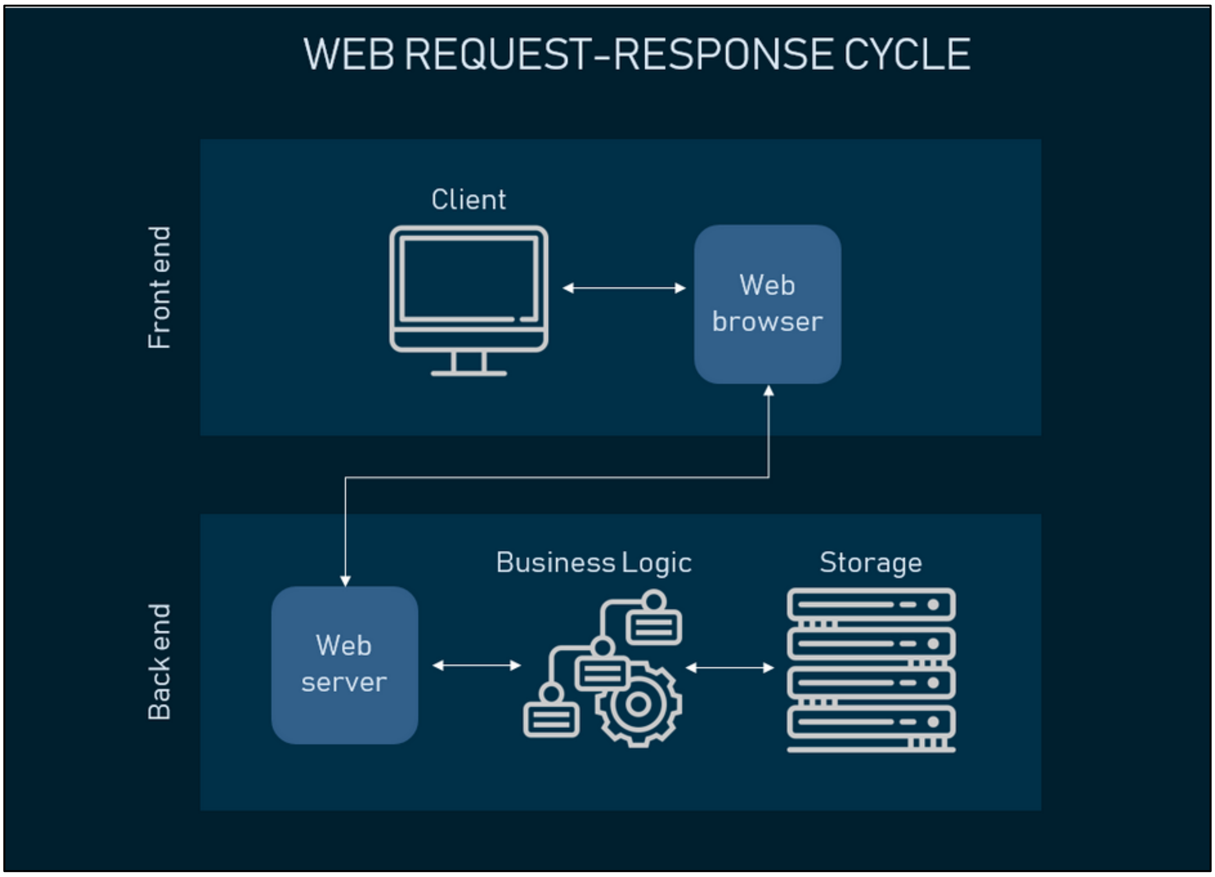


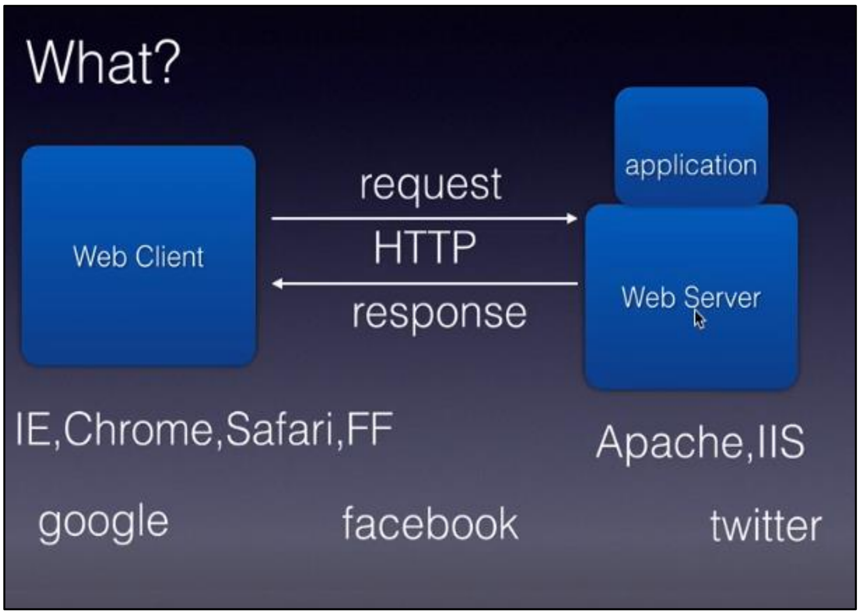
So just as a restaurant’s kitchen, or back-of-the-house, is responsible for preparing all of the food, a web app’s back end is responsible for doing all of the heavy lifting, fulfilling all user requests.

While a restaurant’s kitchen staff chops all of the necessary vegetables and grills all of the steaks, a web app’s back end retrieves data from the database, runs any necessary calculations, and returns that data in a meaningful state so that it can be displayed on the front end.

The role of the database

As such, we need a place to store that data. That’s where a **database** comes in. A database is essentially just a big, organized filing cabinet used for storing and associating a bunch of different pieces of information.





What?

Web Client

request

HTTP

response

application

Web Server

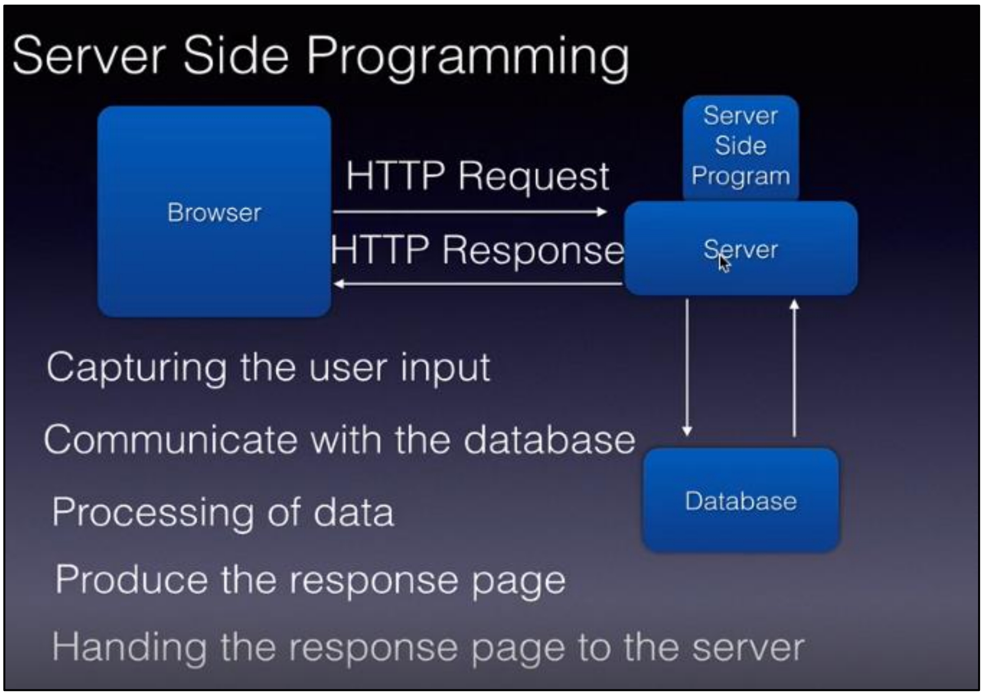
IE,Chrome,Safari,FF

Apache,IIS

google

facebook

twitter



Server Side Programming

Browser

HTTP Request

HTTP Response

Server Side Program

Server

Database

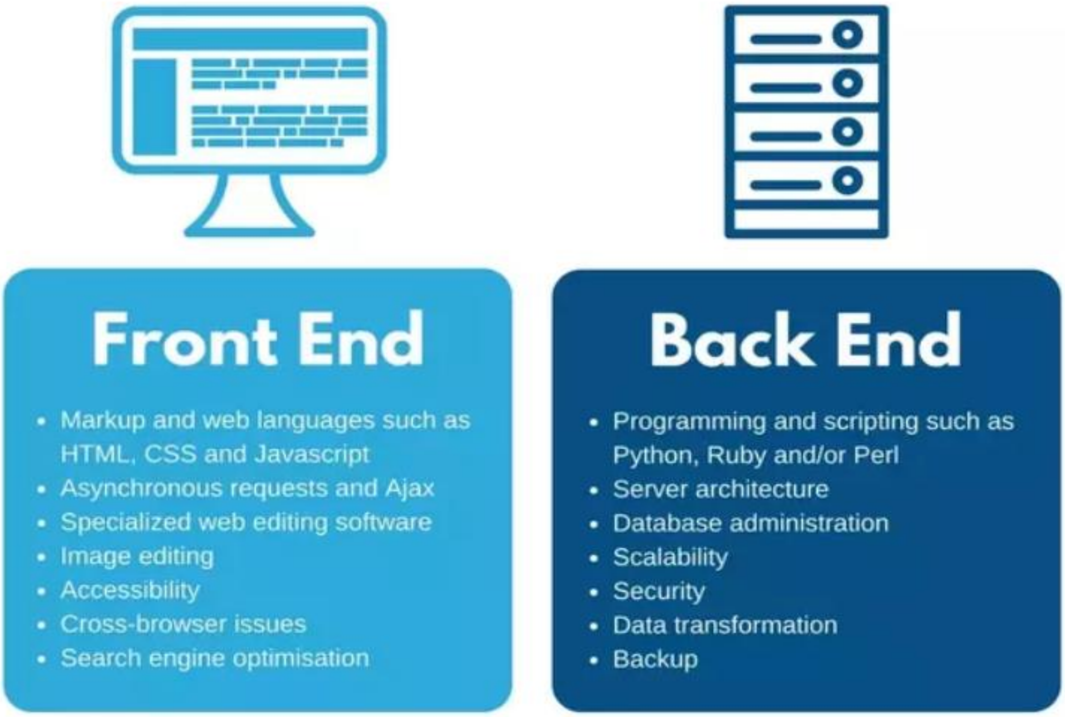
Capturing the user input

Communicate with the database

Processing of data

Produce the response page

Handing the response page to the server



Front End

- Markup and web languages such as HTML, CSS and Javascript
- Asynchronous requests and Ajax
- Specialized web editing software
- Image editing
- Accessibility
- Cross-browser issues
- Search engine optimisation

Back End

- Programming and scripting such as Python, Ruby and/or Perl
- Server architecture
- Database administration
- Scalability
- Security
- Data transformation
- Backup

What and Why Django?

Django is a free and open source web application framework, written in Python. A web framework is a set of components that helps you to develop websites faster and easier.

When you're building a website, you always need a similar set of components: a way to handle user authentication (signing up, signing in, signing out), a management panel for your website, forms, a way to upload files, etc.

Top 5 reason to use Django

1. Easy to Use

Django uses Python programming language which is a popular language in 2015 and now most choosing language by programmers who are learning to code and applications of the Django framework is widely used as it is free and open-source, developed and maintained by a large community of developers. It means we can find answers to the problems easily using Google.

2. It's fast and simple

One of Django's main goals is to simplify work for developers. To do that, the Django framework uses:

- The principles of rapid development, which means developers can do more than one iteration at a time without starting the whole schedule from scratch
- DRY philosophy — Don't Repeat Yourself — which means developers can reuse existing code and focus on the unique one.

3. Excellent Documentation for real-world application

Applications of Django have one of the best documentations for its framework to develop different kinds of real-world applications whereas many other frameworks used an alphabetical list of modules, attributes, and methods. This is very useful for quick reference for developers when we had confused between two methods or modules but not for freshers who are learning for the first time. It's a difficult task for Django developers to maintain the documentation quality as it is one of the best open-source documentations for any framework.

4. It's secure

Security is also a high priority for Django. It has one of the best out-of-the-box security systems out there, and it helps developers avoid common security issues, including

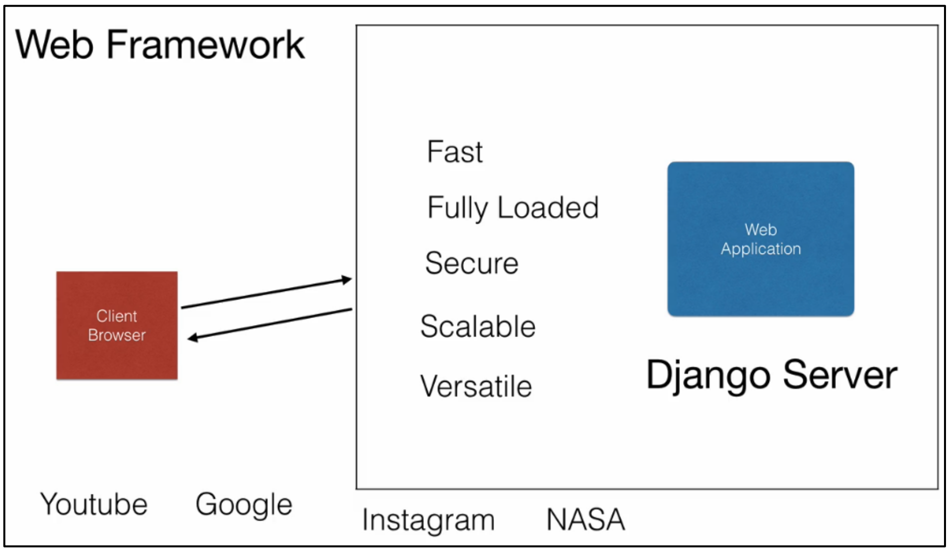
- clickjacking,
- cross-site scripting
- SQL injection.

Django promptly releases new security patches. It's usually the first one to respond to vulnerabilities and alert other frameworks.

5. It suits any web application project

With Django, you can tackle projects of any size and capacity, whether it's a simple website or a high-load web application. Why use Django for your project? Because:

- It's fully loaded with extras and scalable, so you can make applications that handle heavy traffic and large volumes of information
- It is cross-platform, meaning that your project can be based on Mac, Linux or PC
- It works with most major databases and allows using a database that is more suitable in a particular project, or even multiple databases at the same time



MVT Pattern

Django is based on **MVT (Model-View-Template)** architecture. MVT is a software design pattern for developing a web application.

MVT Structure has the following three parts –

Model: Model is going to act as the interface of your data. It is responsible for maintaining data. It is the logical data structure behind the entire application and is represented by a database (generally relational databases such as MySql, Postgres).

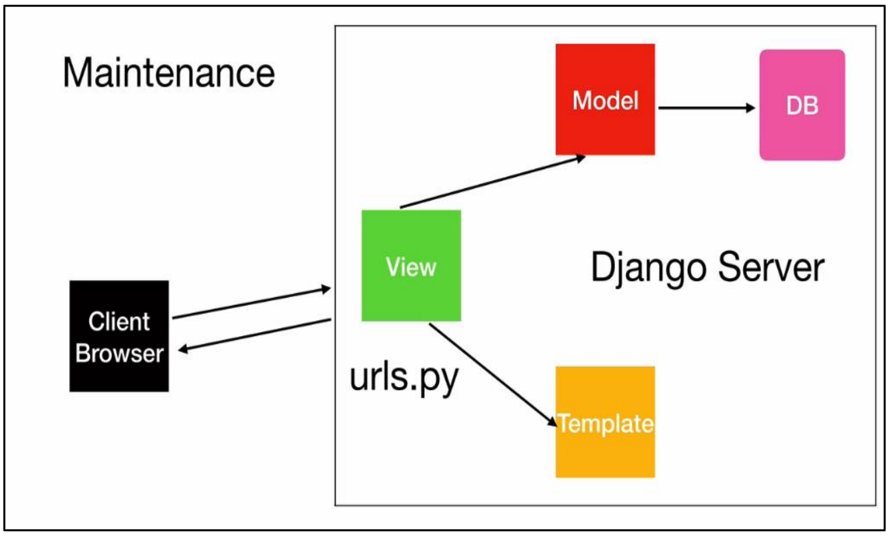
View: The View is the user interface — what you see in your browser when you render a website. It is represented by HTML/CSS/Javascript and Jinja files.

Template: A template consists of static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

When the user request comes in from the web browser to an application running on Django server, Django server will hand that request to over to a particular view. It knows which request is corresponds to which view based on the configuration information we provide in the **urls.py.** Once the view takes the request, it will then process it , use a model if there are databases interaction that are required, and model component is responsible for performing the CRUD operations in database.

The view will then take whatever the model gives and hands it over the template that will render the response to the web browser. So, the template is where all the HTML content lives and will also have some dynamic tags that Django provides, and it is responsible for rendering the final output.

We'll be writing all our logic inside the view and view is responsible for processing the incoming request using the model and then using the template to render the response back.



Software Setup

Before start working on a Django project, please install the below requirements:

1 - Install Django - To install Django, run below command:

```
        $ pip install django
        $ python -m django --version (To check the version of django)
```

2 - Install any database. In our case, we are using MySQL and MySQL workbench.

3 - Install python mysqlclient (it's a connector).

First Django Web Application

Once the Django is installed, you can use Django commands in terminal. 'django-admin' is one of the commands of Django and it has sub-commands as well which can be seen typing 'django-admin' in terminal.

Create a Django project

To create a Django project, you can create a directory and go inside that. After that in terminal, run below command:

```
$ django-admin startproject <project_name>
```

It will create the project for you. A Django Project when initialised contains basic files by default such as manage.py, view.py, etc. A simple project structure is enough to create a single page application. Here are the major files and their explanations.

manage.py - This file is used to interact with your project via the command line (start the server, sync the database… etc). For getting the full list of command that can be executed by manage.py type this code in the command window -

```
        $ python manage.py help
```

folder (<project_name>) - This folder contains all the packages of your project. Initially it contains four files –

- **_init_.py –** It is python package.
- **settings.py –** As the name indicates it contains all the website settings. In this file we register any applications we create, the location of our static files, database configuration details, etc.
- **urls.py –** In this file we store all links of the project and functions to call.
- **wsgi.py –** This file is used in deploying the project in WSGI. It is used to help your Django application communicate with the web server. WSGI stands for Web Server Gateway Interface.

Run the Project

To run your Django project, run below command:

```
        $ python manage.py runserver
```

By default, it uses 8000. In order to change the port:

```
        $ python manager.py runserver 7777
```

It will use port 7777 now.

Create a Django App

Django is famous for its unique and fully managed app structure. For every functionality, an app can be created like a completely independent module.

For example, if you are creating a Blog, Separate modules should be created for Comments, Posts, Login/Logout, etc. In Django, these modules are known as apps. There is a different app for each task.

Benefits of using Django apps

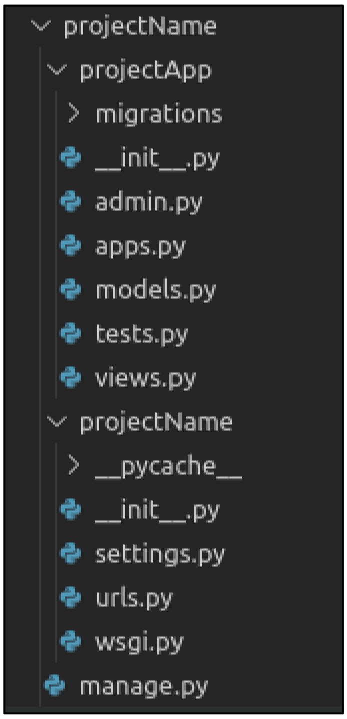
- Django apps are reusable i.e. a Django app can be used with multiple projects.
- We have loosely coupled i.e. almost independent components
- Multiple developers can work on different components
- Debugging and code organisation is easy. Django has excellent debugger tool.

Let us start building an app

To create a basic app in your Django project you need to go to directory containing `manage.py` and from there enter the command:

```
$  python manage.py startapp <appname>
```

Now you can see your directory structure as under:



```
projectName
  projectApp
    migrations
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
  projectName
    __pycache__
    __init__.py
    settings.py
    urls.py
    wsgi.py
  manage.py
```

So, we have finally created an app but to render the app using URLs we need to include the app in our main project so that URLs redirected to that app can be rendered. For that, go to `setting.py` and configure the your inside `INSTALLED_APPS` list.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'AppName'
]
```

Create a View

Django’s views are the information brokers of a Django application. A view sources data from your database (or an external data source or service) and delivers it to a template. For a web application, the view delivers webpage content and templates; for a RESTful API this content could be formatted JSON data.

The view decides what data gets delivered to the template—either by acting on input from the user or in response to other business logic and internal processes. Each Django view performs a specific function and has an associated template.

Views are represented by either a Python function or a method of a Python class. In the early days of Django, there were only function-based views, but, as Django has grown over the years, Django’s developers added class-based views.

We will take a detailed look at the basics of Django's views without getting into any detail on how views interact with models and templates. To help grasp how views work at a fundamental level, we will see how they work using simple HTML responses.

To create our first view, **we need to modify the `views.py` file <u>in our app</u>** (changes are shown in bold):

```
from django.shortcuts import render
from django.http import HttpResponse

def disply(request):
    return HttpResponse("<h1>My First Heading</h1>")
```

We import the `HttpResponse` method. HTTP, the communication protocol used by all web browsers, uses request and response objects to pass data to and from your app and the browser. We need a response object to pass view information back to the browser.

This is your view function. It's an example of a function-based view. It takes a request from your web browser and returns a response. In this simple case, it's just a line of text formatted as an HTML heading.

Configuring the URLs

If you started the development server now, you would notice it still displays the welcome page. For Django to use your new view, you need to tell Django the `display` view is the view you want to display when someone navigates to the site root (home page). We do this by configuring our URLs.

In Django, the `path()` function is used to configure URLs. In its basic form, the `path()` function has a very simple syntax:

```
path(route, view)
```

A practical example of the basic `path()` function would be:

```
path('mypage/', views.myview)
```

n this example, a request to `http://example.com/mypage` would route to the `myview` function in the application's `views.py` file.

The `path()` function also takes an optional `name` argument, and zero or more keyword arguments passed as a Python dictionary.

The `path()` function statements live in a special file called `urls.py`.

When `startproject` created our website, it created a `urls.py` file in our site folder (`\myclub_site\urls.py`). This is the correct place for site-wide navigation but is rarely a good place to put URLs relating to individual applications. Not only is having all our URLs in the one file more complex and less portable, but it can lead to strange behavior if two applications use a view with the same name. To solve this problem, we create a new `urls.py` file for each application. If you are wondering why `startapp` didn't create the file for us, not all apps have public views accessible via URL. For example, a utility program that performs background tasks would not need a `urls.py` file. For this reason, Django lets you decide whether your app needs its own `urls.py` file.

So, in order to configure URL, go to urls.py and add your view there in 'urlpatterns' list. Ensure that you have imported the views from your app.

```
from django.contrib import admin
from django.urls import path
from <appname> import views

urlpatterns = [
     path('admin/', admin.site.urls),
     path('hello/',views.display)
]
```

After this, you can run your application using below command:

```
$ python manage.py runserver
```

Create another View

The views.py can host any number of views that are required for our application. In our new view, we will be returning current server's date and time to client when he/she accesses it. We will use datetime module for that.

```
from django.shortcuts import render
from django.http import HttpResponse
import datetime

# First View
def disply(request):
     return HttpResponse("<h1>My First Heading</h1>")


def displaydatetime(request):

     dt = datetime.datetime.now()

     s = "<b>Current Date and Time: </b>"+str(dt)
     return HttpResponse(s)
```

Once you've added function in views then map it into urls.py like we've done previously and after that, you can start your app.

```
from django.contrib import admin
from django.urls import path
from <appname> import views

urlpatterns = [
     path('admin/', admin.site.urls),
     path('hello/',views.display),
     path('datetime/', views.displaydatetime)
]
```

Task - Create another app inside your project and in that app create a view and then run the app.

Application level URLs

Like we've discussed previously that we can define application level urls.py in order to promote code reusability and then we can use these applications across project.

So, you can just copy paste urls.py from project folder into each of your applications. After pasting the file, you can keep only those URLs inside the file which is specific to your application.

For example, in firstapp, you can update file below (Compare from above what all things have been removed)

```
from django.urls import path
from <appname> import views

urlpatterns = [
     path('admin/', admin.site.urls),
     path('hello/', views.display),
     path('datetime/', views.displaydatetime)
]
```

Similarly, for other apps you can update the urls.py as per their specification.

Now, after doing that, you need to update urls.py inside your project directory such that it will call all other applications urls.py data. For that, we have 'include()' function package django/conf/urls.

```
from django.contrib import admin
from django.urls import path
from django.conf.urls import include

urlpatterns = [
     path('admin/', admin.site.urls),
     path('firstapp/’, include(‘firstapp.urls’)),
     path(‘secondapp/’, include(‘firstapp.urls’))
]
```

Templates

Introduction

Django provides a convenient way to generate dynamic HTML pages by using its template system.

In HTML file, we can't write python code because the code is only interpreted by python interpreter not the browser. We know that HTML is a static markup language, while Python is a dynamic programming language.

Django template engine is used to separate the design from the python code and allows us to build dynamic web pages.

Templates are the third and most important part of Django’s MVT Structure. A template in Django is basically written in HTML, CSS and Javascript in an .html file. Django framework efficiently handles and generates dynamically HTML web pages that are visible to end-user. Django mainly functions with a backend so, in order to provide frontend and provide a layout to our website, we use templates. There are two methods of adding the template to our website depending on our needs. We can use a single template directory which will be spread over the entire project. For each app of our project, we can create a different template directory.

Hands on Steps

- Create a project and app inside it (already completed).
- Setup the templates directory.
- Create a template.
- Create a view and map it to a URL.
- See templates in action.

Create a template directory inside your project

- Once you created your project and app inside it, create a folder named ‘templates’ in the project directory (not in the inside project directory).
- Once templates folder created, create another directory inside it with same name as your app for which you want to create templates. Same name as app is used because it is Django naming convention.
- So, for each application within your project, you are going to create a folder like above we created in step 2 and we are going to store all our templates as per the app.
- Now, go to the ‘settings.py’ in your project and you need to add the path of templates directory inside a dictionary named ‘DIRS’ which is inside a list named ‘TEMPLATES’ in settings.py.
- If you look at the top of the settings.py, there is already a variable ‘BASE_DIR’ which stores the path of your project, so only thing you need to do is to join BASE_DIR with your ‘templates’ using os.path.join() function like below.

```
os.path.join(BASE_DIR, ‘templates’)
```

- In that way, when specific template will be called from view, it will have a path to reach out to it.

Create a Template and View

- Now we are going to create template for an app. For that, go to templates folder and then go inside the folder which has an app name for which you want to create a template.
- Once you are inside the app template folder, then create a html file, as templates are written in html file.
- You can write html as per your requirement and then save it once done.

- After that you need to write code inside views.py of that app to call it as per your requirement.

```
from django.shortcuts import render

def renderTemplate(request):

     return render(request,
'<App_dir_inside_templates>/<template_created.html>')
```

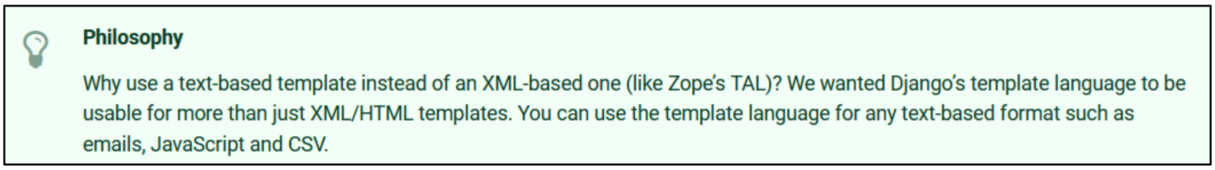
Configure the URL and test

- Coming to URL configuration, you can do it on application level as well as on project level inside 'urls.py' file.
- We will do it on project level.
- It is similar like we did before when we configured urls on app level.

```
from <App_dir> import views
url_patterns = [

         path('firsttemplate/', views.renderTemplate)

      ]
```

Django Template language

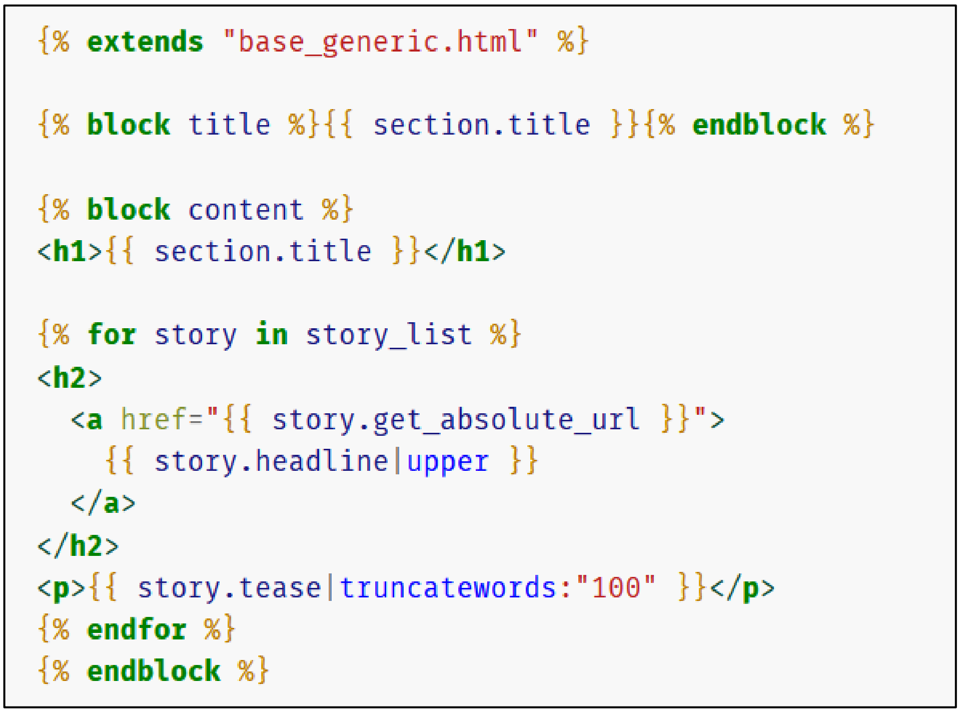
A template is a text file. It can generate any text-based format (HTML, XML, CSV, etc.). A template contains **variables**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.



Philosophy

Why use a text-based template instead of an XML-based one (like Zope's TAL)? We wanted Django's template language to be usable for more than just XML/HTML templates. You can use the template language for any text-based format such as emails, JavaScript and CSV.

Below is a minimal template that illustrates a few basics. Each element will be explained below as we progress.



```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

1 - Variables

Variables look like this: {{ variable }}. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore ("_")

but may not start with an underscore. The dot (".") also appears in variable sections, although that has a special meaning, as indicated below. Importantly, *you cannot have spaces or punctuation characters in variable names.*

Use a dot (.) to access attributes of a variable.

In the above example, {{ section.title }} will be replaced with the title attribute of the section object.

If you use a variable that doesn't exist, the template system will insert the value of the string_if_invalid option, which is set to '' (the empty string) by default.

Note that "bar" in a template expression like {{ foo.bar }} will be interpreted as a literal string and not using the value of the variable "bar", if one exists in the template context. Means if we have variable name bar already exist in a present template, it's value will not be used.

Variable attributes that begin with an underscore may not be accessed as they're generally considered private.

2 - Filters

You can modify variables for display by using **filters**.

Filters look like this: {{ name|lower }}. This displays the value of the {{ name }} variable after being filtered through the lower filter, which converts text to lowercase. Use a pipe (|) to apply a filter.

Filters can be "chained." The output of one filter is applied to the next. {{ text|escape|linebreaks }} is a common idiom for escaping text contents, then converting line breaks to <p> tags.

Some filters take arguments. A filter argument looks like this: {{ bio|truncatewords:30 }}. This will display the first 30 words of the bio variable.

Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaces you'd use {{ list|join:", " }}.

Django provides about sixty built-in template filters. You can read all about them in the built-in filter reference. To give you a taste of what's available, here are some of the more commonly used template filters:

default

If a variable is false or empty, use given default. Otherwise, use the value of the variable. For example:

```
{{ value|default:"nothing" }}
```

If value isn't provided or is empty, the above will display "nothing".

length

Returns the length of the value. This works for both strings and lists. For example:

```
{{ value|length }}
```

If value is ['a', 'b', 'c', 'd'], the output will be 4.

filesizeformat

Formats the value like a "human-readable" file size (i.e. '13 KB', '4.1 MB', '102 bytes', etc.). For example:

```
{{ value|filesizeformat }}
```

If value is 123456789, the output would be 117.7 MB

3 - Tags

Tags look like this: {% tag %}. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. {% tag %} ... tag contents ... {% endtag %}).

Django ships with about two dozen built-in template tags. You can read all about them in the built-in tag reference. To give you a taste of what's available, here are some of the more commonly used tags:

for

Loop over each item in an array. For example, to display a list of athletes provided in athlete_list:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

if, elif, and else

Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

In the above, if athlete_list is not empty, the number of athletes will be displayed by the {{ athlete_list|length }} variable. Otherwise, if athlete_in_locker_room_list is not empty, the message "Athletes should be out..." will be displayed. If both lists are empty, "No athletes." will be displayed.

You can also use filters and various operators in the 'if' tag:

```
{% if athlete_list|length > 1 %}
   Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
   Athlete: {{ athlete_list.0.name }}
{% endif %}
```

While the above example works, be aware that most template filters return strings, so mathematical comparisons using filters will generally not work as you expect. length is an exception.

3 - Comments

To comment-out part of a line in a template, use the comment syntax: {# #}.

For example, this template would render as 'hello':

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

This syntax can only be used for single-line comments (no newlines are permitted between the {# and #} delimiters). If you need to comment out a multiline portion of the template, see the comment tag.

4 - Template Inheritance

The most powerful – and thus the most complex – part of Django's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

Let's look at template inheritance by starting with an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

This template, which we'll call base.html, defines an HTML skeleton document that you might use for a two-column page. It's the job of "child" templates to fill the empty blocks with content.

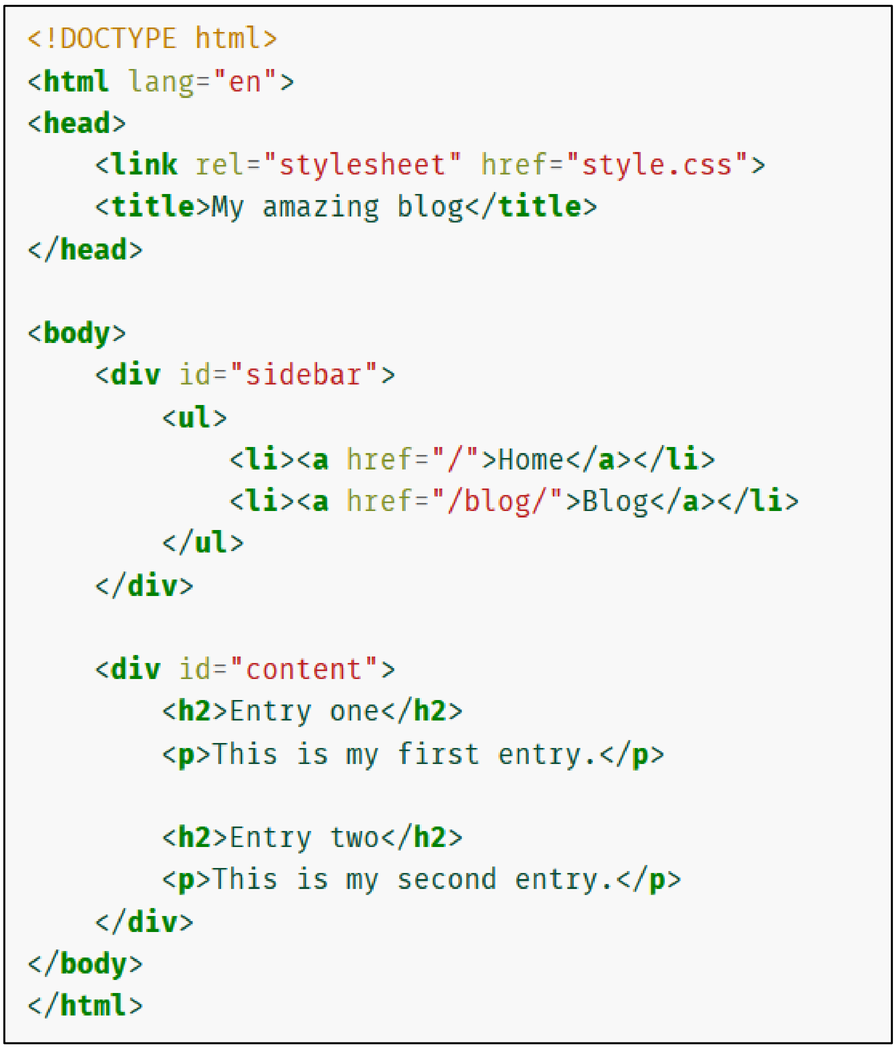
In this example, the `block` tag defines three blocks that child templates can fill in. All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

The `extends` tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent – in this case, "base.html".

At that point, the template engine will notice the three `block` tags in `base.html` and replace those blocks with the contents of the child template. Depending on the value of `blog_entries`, the output might look like:



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css">
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

Note that since the child template didn't define the sidebar block, the value from the parent template is used instead. Content within a {% block %} tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a base.html template that holds the main look-and-feel of your site.
- Create a base_SECTIONNAME.html template for each "section" of your site. For example, base_news.html, base_sports.html. These templates all extend base.html and include section-specific styles/design.
- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and helps to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

- If you use {% extends %} in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.
- More {% block %} tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a {% block %} in a parent template.
- If you need to get the content of the block from the parent template, the {{ block.super }} variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using {{ block.super }} will not be automatically escaped (see the next section), since it was already escaped, if necessary, in the parent template.

- By using the same template name as you are inheriting from, {% extends %} can be used to inherit a template at the same time as overriding it. Combined with {{ block.super }}, this can be a powerful way to make small customizations. See Extending an overridden template in the Overriding templates How-to for a full example.
- Variables created outside of a {% block %} using the template tag as syntax can't be used inside the block. For example, this template doesn't render anything:

```
{% translate "Title" as title %}
{% block content %}{{ title }}{% endblock %}
```

- For extra readability, you can optionally give a *name* to your {% endblock %} tag. For example:

```
{% block content %}
...
{% endblock content %}
```

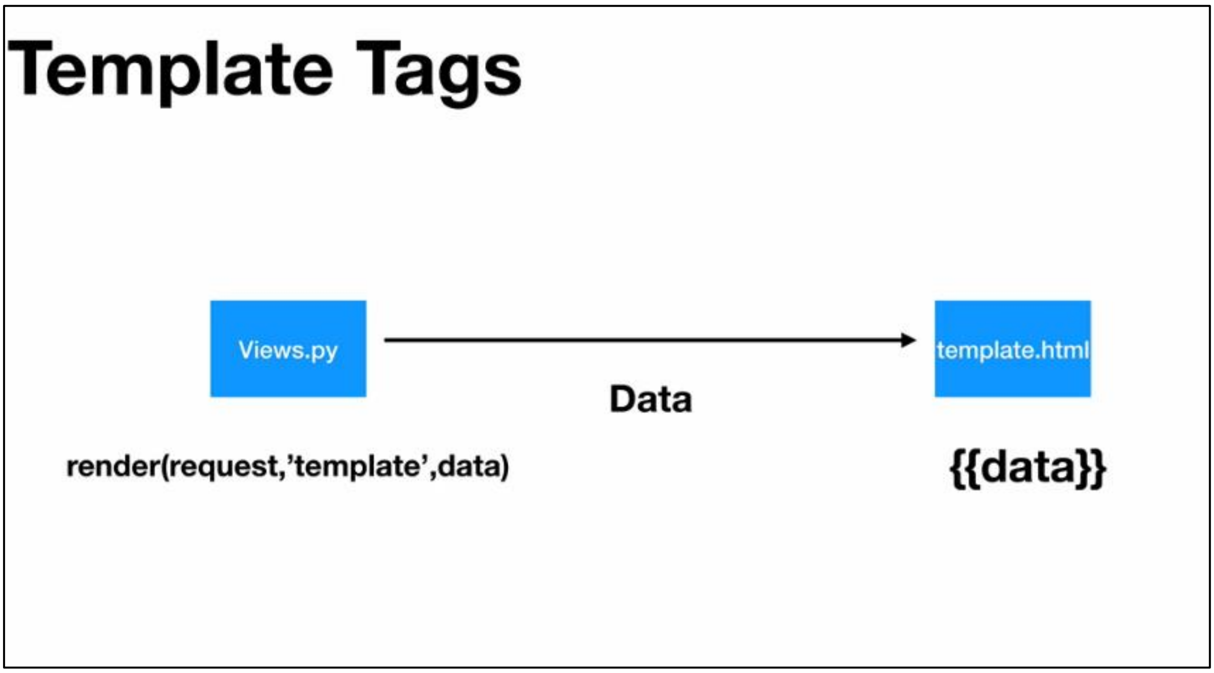
In larger templates, this technique helps you see which {% block %} tags are being closed.
Finally, note that you can't define multiple `block` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill – it also defines the content that fills the hole in the *parent*. If there were two similarly-named `block` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

For more info on tags, please consult official Django documentation.

Using Template Tags

Here, we will to see about how to ask data dynamically from views.py to template.html in 2 simple steps:

- First step is to pass the data to the render method in views.py. render() accept 3 arguments, request, template name, and data which needs to be passed in template. **Data should be defined in dictionary.**
- Once data is passed, it will be retrieved inside the template.html using template tags that Django provides, which is in variable.



Template Tags

Views.py

Data

template.html

render(request,'template',data)

{{data}}

Using Static files

Here, we will see how to insert the static files like images or style sheet. This can be accomplished in below steps:

- First step is to setup the static directory where all the images and style sheets will live for our project. It is similar to templates setup.
- We will use the template tags to insert the static content into the template which is a html page.

Create a folder and name it as ‘static’. Make sure that this folder will be created inside root project folder as we’ve created the template folder earlier. (static folder can be created at app level also)

Now, go to the settings.py and create a variable name ‘STATICFILE_DIRS’ and add the path of static folder in inside a list like below:

```
STATICFILES_DIRS = [os.path.join(BASE_DIR,  <path of static folder>)]
```

Inside static folder, you can also create an image folder where all your app related image will live which can be called by the respective template.

Now, to call the image inside a template, you will have to use template tag `{{% load static %}}` inside the calling template. This will be at top of your template.

Afterwards, you can use image tag in html and inside it pass the path of the image like below:

```
<img src=”{% static “image/<image_name.jpeg>”  %}”/ >
```

So, this is how you load static files using Django templates.

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Photo Demo</title>
</head>
<body>
<h2> Sheldon Cortrell - The Military Man </h2>
    <img src="{% static 'image/image.jpg' %}"/>
</body>
</html>
```

Use a CSS

To load a style sheet inside a template is similar as how we loaded an image. Firstly, create a ‘css’ folder inside static directory.
Inside css folder, you can keep your stylesheets. Now, you can add your stylesheet like below in a template:

```
<link rel=”stylesheet” href=”{% static ‘css/<css_file_name>’ %}”>
```

Models

The next import component in the MVT architecture is Model. It represents database operations and it is with that our view perform all database operation.

A model is the single, definitive source of information about your data. It contains the essential fields and behaviours of the data you’re storing. Generally, each model maps to a single database table.

The basics:

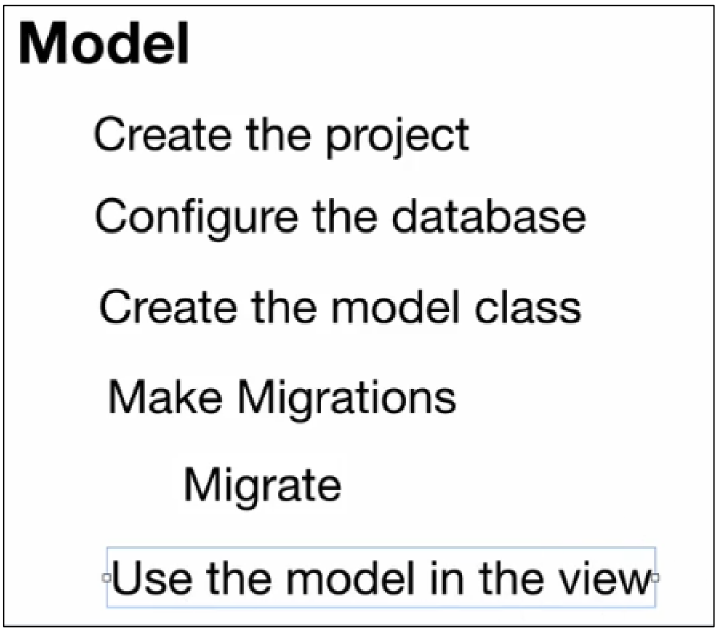
- Each model will represent database table.
- Each model is a Python class which inherits `django.db.models.Model`.
- Each attribute of the model represents a database field.

- With all of this, Django gives you an automatically generated database-access API.

We can create database structure from our models using commands like **‘makemigrations’** which will generate the SQL code from model class and using that code we can create database table by using **‘migrate’** command and from that point you can maintain all your database table through these commands.

Along with application table, Django also creates many other tables like admin, security, session management etc.

We will follow below task in order to understand the model:



Model

Create the project

Configure the database

Create the model class

Make Migrations

Migrate

Use the model in the view

1 - We will create a Django project, or you can work in your existing one.
2 - Afterwards, go to settings.py inside the project and search for ‘DATABASES’ variable.
3 - You will see by default, every Django web application comes with sqlite3 db for free.
4 - DATABASES variable will be configured with sqlite3 by default and to test whether the connection is established with sqlite3, go to your terminal and be in your project directory and type below command and then interactive python shell will get open. Now import connection from django.db and run method cursor(). If it does not throw error then it means the connection is successful.

```
python3 manage.py shell
[1]: from django.db import connection
[2]: c = connection.cursor()
```

5 - Sqlite3 is good for testing purpose but when it comes to production environment it is not mostly used.

<u>Using MySql database (Configuring MySql with Django)</u>

In this section, we will configure MySql database in our project. We can do that using the following configuration or properties exist inside settings.py:

- ENGINE
- NAME (Name of the database)
- USER (Name of the user through which we will connect to database)
- PASSWORD
- HOST (If db is running on remote server)
- PORT (port on the remote server)

If host or port is not provided, then it will consider local machine as host and default port number on which db runs.

We will update above parameters in our settings.py. In it, search for DATABASES variable and set value of ENGINE as ‘django.db.backends.mysql’ and remaining required parameters as below.

```
        DATABASES = {
            'default': {
                'ENGINE': 'django.db.backends.mysql',
                'NAME': '<Name of your DB>',
                'USER': 'root',
                'PASSWORD': 'test1234'
            }
        }
```

Create Model Class

Here we will create the model class (for example - Employee) and for that, go to your relevant app and open `models.py`. You will find that models module is imported already in it and from it create our class(Employee) which will inherit models.Model class.

So, inside class, whatever the fields are required can be defined as attributes and their datatypes will be defined as below:

```
from django.db import models

class Employee(models.Model):
   firstName = models.CharField(max_length=30)
   lastName = models.CharField(max_length=30)
   salary = models.FloatField()
   email = models.CharField(max_length=40)
```

Now, we will generate database table from this model.

Converting Model to DB Tables

We have created our model class above, now with that, we will create db code that will responsible for creating tables.

1 - Go to terminal and be on project path.
2 - Type command `python3 manage.py makemigrations`. It will generate the database code for us.
3 - As you hit enter, a python file will generate in the migration directory of your relevant app. Name of that file starts with number sequence like 0001, 0002 etc.
4 - If you want to see the SQL code, go to terminal and type this command `python3 manage.py sqlmigrate <app_name> 0001` (Note - This command is not used much)

5 - Till now only SQL code/query is generated from the model, but table is yet not created. So, to create the table now based on the query generated, we type below command:

```
python3 manage.py migrate
```

You can verify whether in your db, table has been created by going to MySql workbench and using query `show tables.`

You will find other tables as well along with yours.

Use the Model in View

In this section, we will use the model inside the view. So, let's create a view in our application.

```
from django.shortcuts import render
from <app_name>.model import Employee  # Employee is a class we have defined under
model.py in our app

def employeedata(request):
   emp = Employee.objects.all()  # Basically, it will execute, query SELECT * from
<employee_table> to get all emp details
   emp_dict = {'employees': emp}
   return render(request, <template_path_in_which_we_are_passing_data>, emp_dict)
```

So, basically, we are passing details from db to view which is calling template where this data is dynamically added.

Create the Template

Below is the template created which will render the employee data received from DB:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Employee Information</title>
</head>
<body>
    <h1>Employee List</h1>
    {% if employees %}
    <table>
        <thead>
        <th>First Name</th>
        <th>Second Name</th>
        <th>Salary</th>
        <th>Email</th>
        </thead>
        {% for emp in employees %}
        <tr>
            <td>{{emp.firstName}}</td>
            <td>{{emp.lastName}}</td>
            <td>{{emp.salary}}</td>
            <td>{{emp.email}}</td>
        </tr>
        {% endfor %}
    </table>
        {% else %}
        <p> No Records Exist</p>
        {% endif %}
</body>
</html>
```

Django Admin UI

Django by default provides admin interface and you can see it's url mapping inside project's urls.py. You can checkout this admin page by typing `localhost:8000/admin`.

To login on this UI, you need to create a superuser which can be done by this command:

```
python3 manage.py createsuperuser
```

It will then ask for usernames, password and other parameters. Fill all those and your ID will be created. Now, you can try to login.

Once you login, you will find 'Users' and 'Groups' which will be discussed during Security topic.

In upcoming sessions, we will also see that how we can use this admin UI to view the object of the model without much work as we did before.

Adding Model to the Admin UI

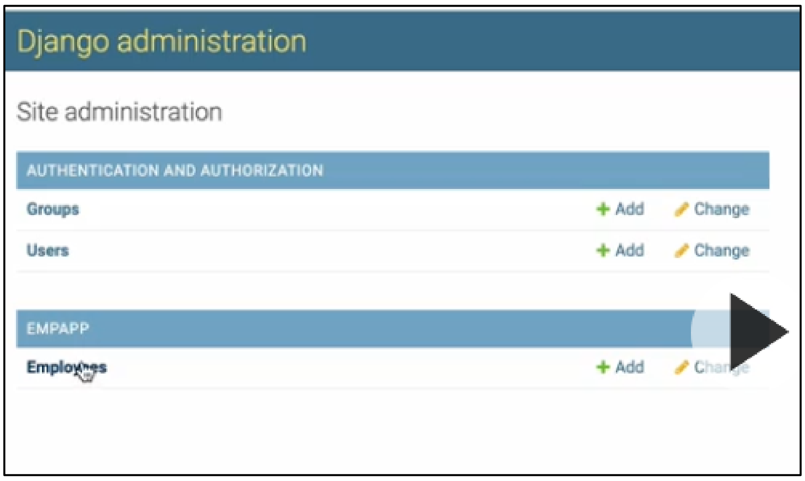
In this session, we will see how to add our model to the admin UI. For that, go to **admin.py** of your app and register your model. This will be done in below way:

```
from django.contrib import admin
from <app_name>.models import Employee  # This model we have created for learning purpose

admin.site.register(Employee)
```

Once done, now go to the admin UI and login with you ID and password you've created. You will find now employee model and now you can add, modify or delete data from UI. Basically, you can perform the CRUD operations.

Displaying Model Fields on the UI

In previous section, we added model to the admin UI. It was reflecting as below:



Django administration

Site administration

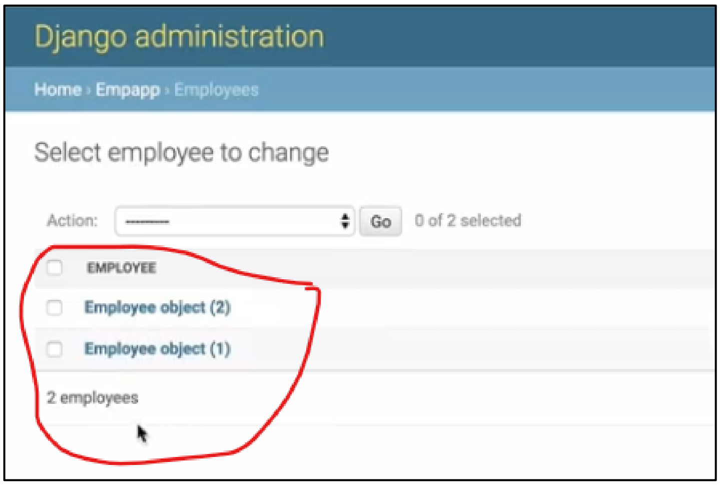
AUTHENTICATION AND AUTHORIZATION

Groups + Add Change

Users + Add Change

EMPAPP

Employees + Add Change



Django administration

Home › Empapp › Employees

Select employee to change

Action: --------- Go 0 of 2 selected

□ EMPLOYEE

□ Employee object (2)

□ Employee object (1)

2 employees

So, we have added our Employee model and when you click on it you got employee object as highlighted in the screenshot above instead of fields.

So, if you want to make this UI even better, you can do that in simple steps:

1 - Go to `admin.py` of your app and create a class with name say, `EmployeeAdmin` which will inherit `admin.ModelAdmin`.
2 - Within this class we have to provide a list of parameters you want to get display.

```
    list_display = ['firstName', 'lastName']
```

Remember, name of the list must be same as **list_display,** it's a Django convention.

then register the class as well like below:

```
admin.site.register(Employee, EmployeeAdmin)
```

Using SqlLite DB

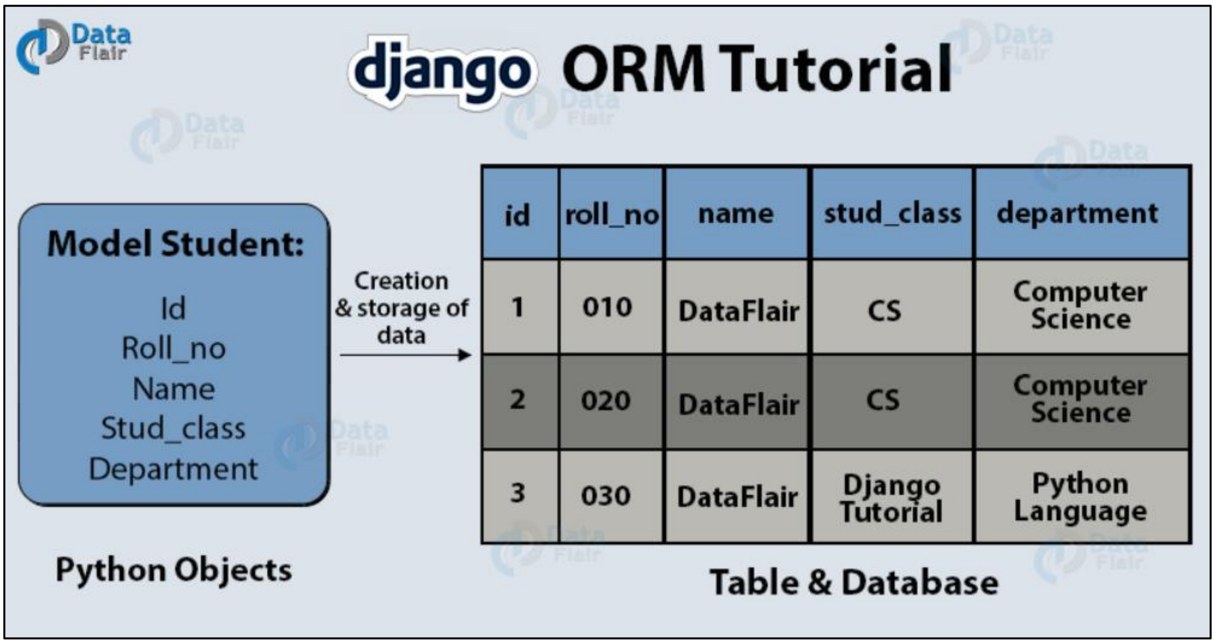
As we know that already that, In Django, by default, SqlLite DB is configured. Operations performed in SqlLite is similar to what we have discussed in MySql.

Django ORM

Django ORM provides an elegant and powerful way to interact with the database. ORM stands for Object Relational Mapper. It is just a fancy word describing how to access the data stored in the database in Object Oriented fashion.

ORM's main goal is to transmit data between a relational database and application model. The ORM automates this transmission, such that the developer need not write any SQL.

ORM, as from the name, maps objects attributes to respective table fields. It can also retrieve data in that manner. This makes the whole development process fast and error-free.



id	roll_no	name	stud_class	department
1	010	DataFlair	CS	Computer Science
2	020	DataFlair	CS	Computer Science
3	030	DataFlair	Django Tutorial	Python Language

In the above image, we have some Python objects and a table with corresponding fields. The object's attributes are stored in corresponding fields automatically. An ORM will automatically create and store your object data in the database. You don't have to write any SQL for the same.

As in one of the previous example where we were defining views for model we have used **Employee.objects.all(),** this is nothing but Django ORM.

In this section, we will explore about Django ORM. For that, we don't require to write codes in IDE, rather we will use Django shell to learn about it.

- Make sure you are in the project directory.
- Open shell by using this command `python3 manage.py shell`
- We have to import necessary models like your model class from the relevant app.

```
>>> from <app_name>.models import Employee
```

- Once you import your class from model, you can then perform all type of operations. Let's call all the element of from a existing table.

```
>>> qs = Employee.objects.all()
>>> print(type(qs))  # To check the kind of object (It is of class QuerrySet)
<class 'django.db.models.querry.QuerySet'>
```

- If you want one record at a time, you can write like below where we are calling records by their unique ID.

```
>>> emp = Employee.objects.get(id=1)
>>> print(emp.firstName)
```

- If you check the type of emp, it will be of class Employee but when we are getting full table, it will be the instance of class QuerySet.

- For any object of QuerySet which basically in background writing query to get the data. So, if you want to see the query, you can type like below and its output would be the query written to perform the dB operations.

```
>>> print(qs.query)
```

Little about QuerySet…

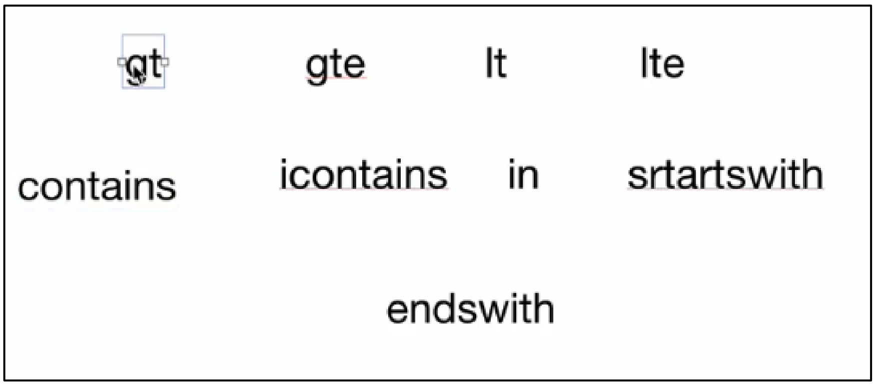
As we know that Django's models are written in Python and provide a mapping to the underlying database structure. Django uses a model to execute SQL behind the scenes to return Python data structures—which Django calls *QuerySets*.
We all use queries to retrieve data from the database. Querysets are Django's way to retrieve data from the database. The Django ORM lets us use Querysets.
A Queryset is a list of objects of a model. We use Querysets to filter and arrange our data. These make our work as a Python developer easier. Querysets are generally associated with CRUD applications.

Filtering Data

In previous section, we were fetching all the data of table and there was not any filtering used. The QuerySet returned by all() describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects.
To create such a subset, you refine the initial QuerySet, adding filter conditions. The two most common ways to refine a QuerySet are:

filter(**kwargs) - Returns a new QuerySet containing objects that match the given lookup parameters.
exclude(**kwargs) - Returns a new QuerySet containing objects that do not match the given lookup parameters.

Django ORM offers various clauses that you can use inside a `filter()` method.
Below are some of the clauses:



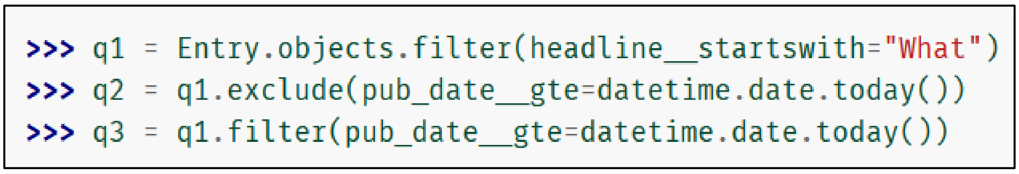
gt gte lt lte

contains icontains in srtartswith

endswith

gt - greater than
gte - greater than equal to
lt - less than
lte - less than equal to
contains - search option
icontains - It is same as contains except it searches irrespective of the case, lowercase or uppercase (case insensitive)

Filtered QuerySets are unique - Each time you refine a QuerySet, you get a brand-new QuerySet that is in no way bound to the previous QuerySet. Each refinement creates a separate and distinct QuerySet that can be stored, used and reused.



```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

Let's see an example:

```
# All the employees whom salary are greater than 5000. Notice 2 underscore used
between salary and gt. This is way of writing.
```

```
emps = Employee.objects.filter(salary__gt=5000)
```

Output would be the Employee object which has salary greater than 5000.

```
emps = Employee.objects.filter(id__in=[1,2])
```

Output would be the Employee object which has the ID 1 and 2.

Retrieving a single object with get() - filter() will always give you a QuerySet, even if only a single object matches the query - in this case, it will be a QuerySet containing a single element. If you know there is only one object that matches your query, you can use the get() method which returns the object directly.

```
emps = Employee.objects.get(firstName__icontain='ABC1')
print(emps)
```

Output will be the object which have the passed string.

Using Logical Operators

Here, we will discuss about the logical operators like AND, OR or NOT inside filter method. Let's see the examples below:

# Demonstration of OR

```
>>> emps = Employee.objects.filter(firstName__startswith='ABC1') |
Employee.objects.filter(lastName__endswith='XYZ1')
>>> print(emps)
```

Output would be Employee object which has one of these criteria or both.

The above way of filtering using OR seems lengthy. We can also write the short version of it. For that, we will use the class 'Queue' from module django.db.models.

```
from django.db.models import Q

>>> emps = Employee.objects.filter(Q(firstName__startswith='ABC1') |
Q(lastName__endswith='XYZ1'))

>>> print(emps)
```

# Demonstration of AND

```
>>> emps = Employee.objects.filter(Q(firstName__startswith='ABC1') &
Q(lastName__endswith='XYZ1'))

>>> print(emps)
```

You can also use AND in below way where we are separating conditions with a comma:

```
>>> emps =
Employee.objects.filter(firstName__startswith='ABC1',lastName__endswith='XYZ1')

>>> print(emps)
```

For implementing NOT, we can use exclude method like below:

```
>>> emps = Employee.objects.exclude(salary__gt=20000)
```

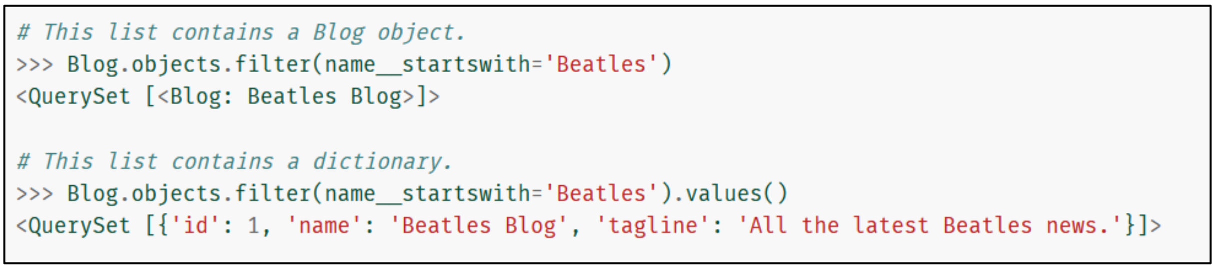
Selective Columns

To get the selective data from db, we can use below methods:

1 - values()

`values`(**fields*, ***expressions*)

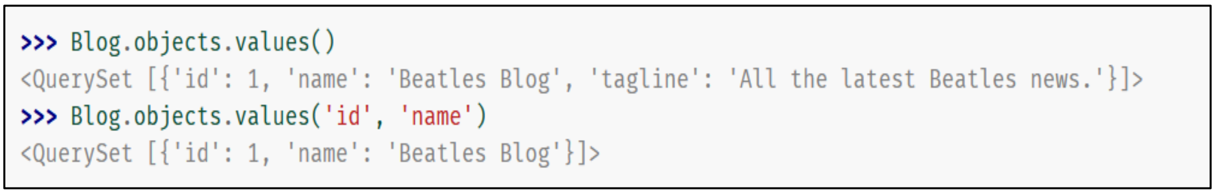
Returns a QuerySet that returns dictionaries, rather than model instances, when used as an iterable. Each of those dictionaries represents an object, with the keys corresponding to the attribute names of model objects. This example compares the dictionaries of values() with the normal model objects:



```
# This list contains a Blog object.
>>> Blog.objects.filter(name__startswith='Beatles')
<QuerySet [<Blog: Beatles Blog>]>

# This list contains a dictionary.
>>> Blog.objects.filter(name__startswith='Beatles').values()
<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]>
```

The values() method takes optional positional arguments, *fields, which specify field names to which the SELECT should be limited. If you specify the fields, each dictionary will contain only the field keys/values for the fields you specify. If you don't specify the fields, each dictionary will contain a key and value for every field in the database table.

Example:

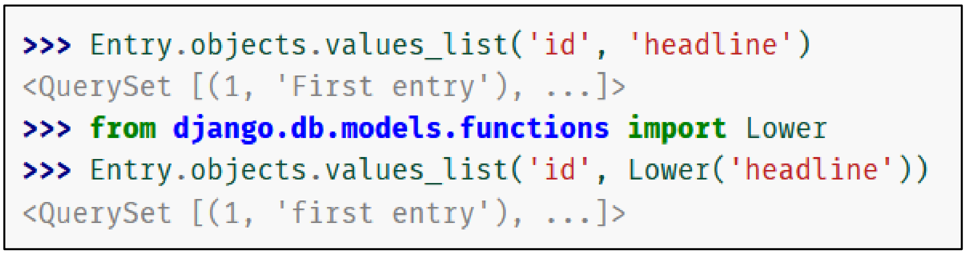


```
>>> Blog.objects.values()
<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]>
>>> Blog.objects.values('id', 'name')
<QuerySet [{'id': 1, 'name': 'Beatles Blog'}]>
```

2 - values_list()

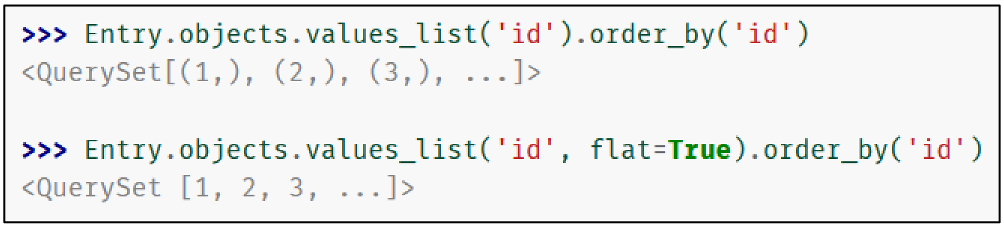
`values_list`(**fields*, *flat=False*, *named=False*)

This is similar to values() except that instead of returning dictionaries, it returns tuples when iterated over. Each tuple contains the value from the respective field or expression passed into the values_list() call — so the first item is the first field, etc. For example:



```
>>> Entry.objects.values_list('id', 'headline')
<QuerySet [(1, 'First entry'), ...]>
>>> from django.db.models.functions import Lower
>>> Entry.objects.values_list('id', Lower('headline'))
<QuerySet [(1, 'first entry'), ...]>
```

If you only pass in a single field, you can also pass in the flat parameter. If True, this will mean the returned results are single values, rather than one-tuples. An example should make the difference clearer:



```
>>> Entry.objects.values_list('id').order_by('id')
<QuerySet[(1,), (2,), (3,), ...]>

>>> Entry.objects.values_list('id', flat=True).order_by('id')
<QuerySet [1, 2, 3, ...]>
```

It is an error to pass in flat when there is more than one field.

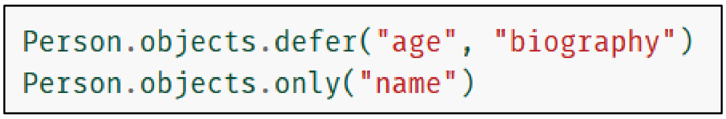
values() and values_list() are both intended as optimizations for a specific use case: retrieving a subset of data without the overhead of creating a model instance. This metaphor falls apart when dealing with many-to-many and other multivalued relations (such as the one-to-many relation of a reverse foreign key) because the “one row, one object” assumption doesn’t hold.

3 - only()

`only`(**fields*)

The only() method is more or less the opposite of defer(). You call it with the fields that should not be deferred when retrieving a model. If you have a model where almost all the fields need to be deferred, using only() to specify the complementary set of fields can result in simpler code.

Suppose you have a model with fields name, age and biography. The following two querysets are the same, in terms of deferred fields:



```
Person.objects.defer("age", "biography")
Person.objects.only("name")
```

Difference between values() and only() is that only() will by default include the ID field along with passed fields. You can also check it in the query.

Aggregate Functions

Django offers several functions to perform aggregate operation like average, maximum, minimum, sum, count etc. To use those, we need to import **models** module and use all like below:

```
>>> from django.db.models import Avg, Sum, Min, Max, Count
>>> avg = Employee.objects.all().aggregate(Avg(‘salary’))
>>> print(avg)
>>> sum = Employee.objects.all().aggregate(Sum(‘salary’))
>>> print(sum)
>>> c = Employee.objects.all().aggregate(Count(‘salary’))
>>> print(c)
```

Try these in the terminal.

Create

We will now see about how to create entry in db table using Django. There are 2 ways to go:

1 - To create the instance of your model class, in our example, we will create the instance of Employee class and save it.

```
>>> from <app_name>.models import Employee

#  Let’s check the count of records present currently in our table in db
>>> e = Employee.objects.all().count()

# Now create the instance of Employee class and then save it
>>> e1 = Employee(firstName=’ABC4’, lastName=’XYZ4’, salary=’45000’)
>>> e1.save()
#  You can check the count to check if it gets increased
```

2 - Another way of creating it is to use the create() method.

```
>>> Employee.objects.create(firstName='ABC5', lastName='XYZ5', salary='80000')
```

Output would be created objects.

Bulk Create

To bulk create, we use the method `bulk_create()`. This takes the list of data as per the fields as an argument. Example:

```
>>> Employee.objects.bulk_create([Employee(firstName='ABC6', lastName='XYZ6',
salary='20000'), Employee(firstName='ABC7', lastName='XYZ7', salary='90000')])
```

It will create those entries in the table.

Delete

To delete a record from a database, we will first get the access of the employee object we want to delete.

```
>>> e = Employee.objects.get(id=1)
>>> e.delete()
```

# Employee which has id 1 has been deleted.

If you want to do bulk delete or delete everything in table, do it like below:

```
>>> qs = Employee.objects.filter(salary__gt=5000)
>>> qs.delete()
```

# It will delete all records which met the criteria. If you want to delete full table, you can do it like below:

```
>>> Employee.objects.all().delete()
```

Update

To perform the update operation, you will first need to fetch the record, update it and save it. Check out below example to understand about it.

```
>>> emp = Employee.object.get(id=2)
>>> emp.firstName= "ZZZZ"
>>> emp.save()
```

Order by

Here we will learn about how to sort or order the data when it comes from database. To do this we are going to use the order_by()

```
>>> emp = Employee.object.all().order_by('salary')
```

Objects will be sorted in ascending order with respect to salary.

If you want to reverse the sorting order, call like below:

```
>>> emp = Employee.object.all().order_by('-salary')
```

You can also sort alphabetically. Let’s sort with respect to first name.

```
>>> emp = Employee.object.all().order_by(‘firstName’)
```

but it would be case sensitive. If you want to do case insensitive, you need to use lower() function from module `functions`.

```
>>> from django.db.models.functions import Lower
>>> emp = Employee.object.all().order_by(Lower(‘firstName’))
```

Forms

Introduction

HTML forms

In HTML, a form is a collection of elements inside <form>...</form> that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.
Some of these form interface elements - text input or checkboxes - are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form <input> elements to achieve these effects.

As well as its <input> elements, a form must specify two things:

- *where*: the URL to which the data corresponding to the user’s input should be returned
- *how*: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several <input> elements: one of type="text" for the username, one of type="password" for the password, and one of type="submit" for the “Log in” button. It also contains some hidden text fields that the user doesn’t see, which Django uses to determine what to do next.
It also tells the browser that the form data should be sent to the URL specified in the <form>’s action attribute - /admin/ - and that it should be sent using the HTTP mechanism specified by the method attribute - post.
When the <input type="submit" value="Log in"> element is triggered, the data is returned to /admin/.

GET and POST are the only HTTP methods to use when dealing with forms.

Django’s role in forms

Handling forms is a complex business. Consider Django’s admin, where numerous items of data of several different types may need to be prepared for display in a form, rendered as HTML, edited using a convenient interface, returned to the server, validated and cleaned up, and then saved or passed on for further processing.
Django’s form functionality can simplify and automate vast portions of this work and can also do it more securely than most programmers would be able to do in code they wrote themselves.
Django handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client

It is *possible* to write code that does all of this manually, but Django can take care of it all for you.

The Django Form class - At the heart of this system of components is Django’s Form class. In much the same way that a Django model describes the logical structure of an object, its behaviour, and the way its parts are represented to us, a Form class describes a form and determines how it works and appears.

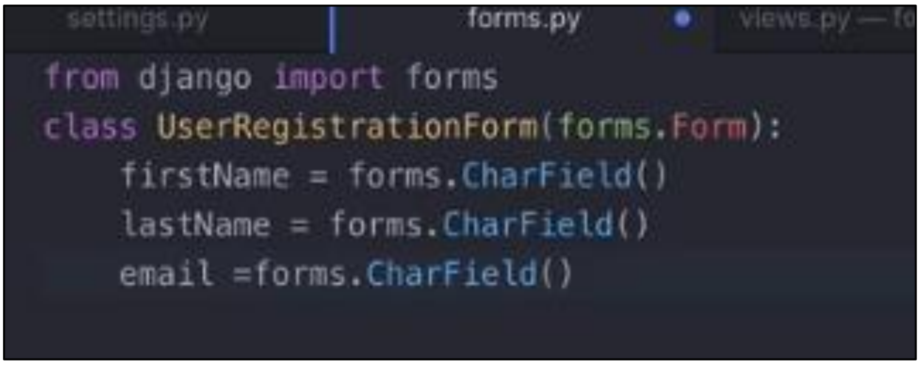
In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form <input> elements. (A ModelForm maps a model class's fields to HTML form <input> elements via a Form; this is what the Django admin is based upon.)
A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A DateField and a FileField handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML "widget" - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required. (Widgets are more explained later)

<u>Create the Form</u>

We will create our first Django form for User registration where user will provide data such as first name, last name and email and submit it.

- Create a forms.py under the application you want to proceed with.
- Go to forms.py and import forms module from Django package.
- Create a class let's say, UserRegistrationForm and it should extend form.Form.
- Within this class we add all the fields we want on that form.
- When we work on model forms later on, we can simply use all the fields as we have defined in that respective model class. We will see it in coming sections.
- For now, we will define first name, last name and email like below which is very similar as we had defined in model class previously.



```
from django import forms
class UserRegistrationForm(forms.Form):
    firstName = forms.CharField()
    lastName = forms.CharField()
    email =forms.CharField()
```

<u>CSRF Token</u>

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

The level of the attack is based upon the level of privileges that the victim possessed. Because attacker will use the authentication that has gained in the current session to do the malicious task. This is the reason why this attack termed as Session Riding too. CSRF attack will exploit the concept that if the user is authenticated all the requests that come from that user must be originated by the user. The attacker will exploit this concept by identifying the session cookie of the session and use that to send his own payload to run on the application.

<u>Preventing CSRF vulnerabilities</u>

The most popular implementation to prevent Cross-site Request Forgery (CSRF), is to make use of a token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a *CSRF Token* or a *Synchronizer Token*, works as follows:

1. The client requests an HTML page that contains a form.

2. The server includes two tokens in the response. One token is sent as a cookie. The other is placed in a hidden form field. The tokens are generated randomly so that an adversary cannot guess the values.
3. When the client submits the form, it must send both tokens back to the server. The client sends the cookie token as a cookie, and it sends the form token inside the form data. (A browser client automatically does this when the user submits the form.)
4. If a request does not include both tokens, the server disallows the request.

This protects the form against CSRF attacks, because an attacker forging a request will also need to guess the anti-CSRF token. Unless they won’t successfully trick a victim into sending a valid request. This token should be invalidated after some time and after the user logs out. Anti-forgery tokens work because the malicious page cannot read the user’s tokens, due to same-origin policy.

Django provides CSRF token feature and you can check it as configured in settings.py under middleware section.

<u>Use the form in the View</u>

We will use form inside our view. View is responsible for sending the form to template as well as processing the data once the form is submitted.
Let’s create view.

Go to your relevant app and open view.py.
Define a function which will send form to a template like below.

```
from <app_name> import forms
from django.shortcuts import render

def userRegistration(request):
   form = forms.UserRegistrationForm()    # Class we have created inside forms.py
   return render(request, ‘<app_name>/userRegistration.html’, {‘form’: form})
```

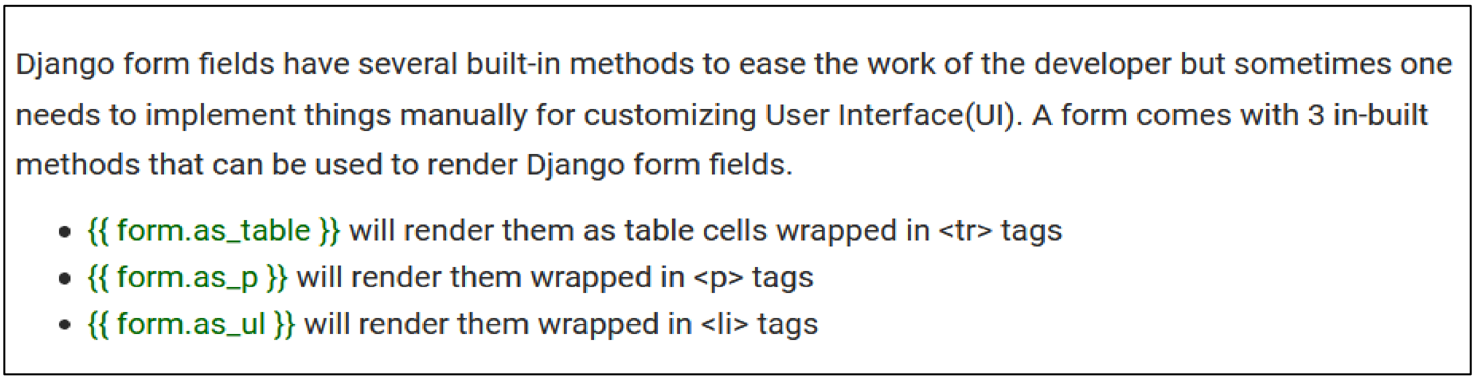
In the next section, we will define template userRegistration.html

<u>Create the Template</u>

- Go to template directory and inside it, go to the relevant app in which we will be writing our template.
- Create ‘userRegistration.html’ template like below and use template tag under form tag:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>User Registration</title>
</head>
<body>
    <h1>User Registration</h1>
    <form method="post">
    {{ form.as_p }}
    </form>
</body>
</html>
```

- Inside <form> tag, we’ve used `form.as_p` which means that Django will render form as paragraph. Other than this option, we can also use `‘as_table’` or `‘as_ul’` (unordered list).



Django form fields have several built-in methods to ease the work of the developer but sometimes one needs to implement things manually for customizing User Interface(UI). A form comes with 3 in-built methods that can be used to render Django form fields.

- {{ form.as_table }} will render them as table cells wrapped in <tr> tags
- {{ form.as_p }} will render them wrapped in <p> tags
- {{ form.as_ul }} will render them wrapped in <li> tags

Configure the CSRF token and URLs

To configure the CSRF token, go to template in which you are rendering django form and add `{% csrf_token %}`. Django will automatically add html hidden field and value of it will have a token. Remember, without CSRF token, Django form will not work.

```
 <!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>User Registration</title>
</head>
<body>
    <h1>User Registration</h1>
    <form method="post">
    {{ form.as_p }}
    {% csrf_token %}
    </form>
</body>
</html>
```

Next task is to configure the URL which we have discussed previously. So, do it yourself.

Processing the form data

The next step is to collect the form data in the view, that is when the user hits the submit button, we should process the data. To do that, go to views.py and add a condition as below:

```
from <app_name> import forms
from django.shortcuts import render


def userRegistration(request):

   form = forms.UserRegistrationForm()    # Class we have created inside forms.py

   if request.method==’POST’:
       form = forms.UserRegistrationForm(request.POST)  # It will take all the
incoming parameters and create a user registration form object for us and it will
have all the data that user submitted in a dictionary called cleaned_data

   if form.is_valid():

       # here validation of data is done automatically by Django
       print(“Form is valid”)
       print(“First Name”, form.cleaned_data[‘firstName’])
       print(“Last Name”, form.cleaned_data[‘lastName’])
       print(“Email”, form.cleaned_data[‘email’])

   return render(request, ‘<app_name>/userRegistration.html’, {‘form’: form})
```

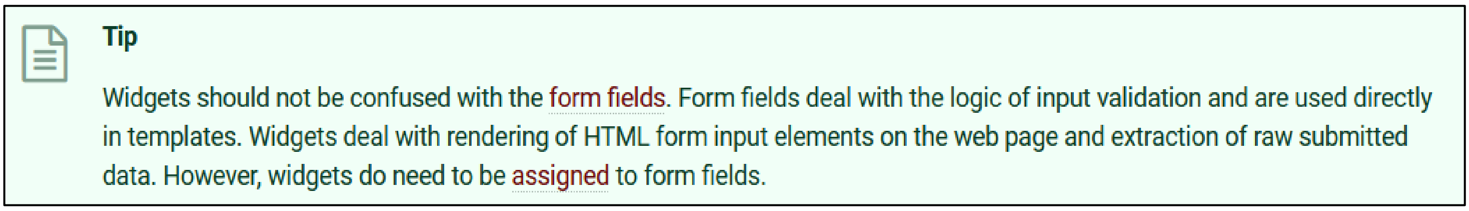
form.is_valid() is basically validating the data submitted by user which is one of the inbuilt feature of Django. All the print statement will be visible in the terminal.

Different types of form fields

When we create a django form, by default, it creates input type of fields as texts. However, if you want different input types, django provides many.

Let’s add some more fields with different input type in our existing form but before that, we will see about widgets.

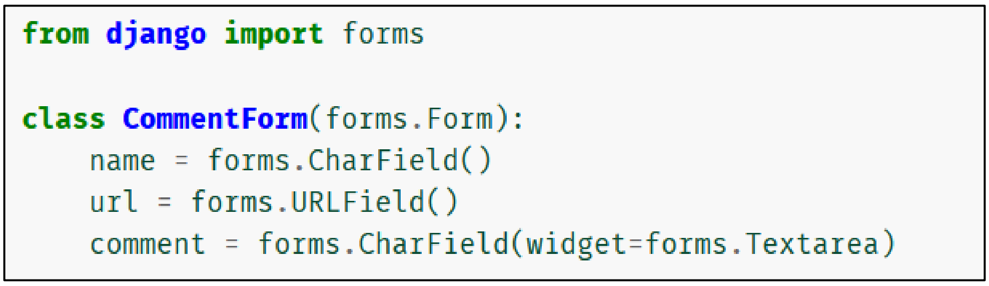
A widget is Django’s representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.



Tip

Widgets should not be confused with the form fields. Form fields deal with the logic of input validation and are used directly in templates. Widgets deal with rendering of HTML form input elements on the web page and extraction of raw submitted data. However, widgets do need to be assigned to form fields.

Whenever you specify a field on a form, **Django will use a default widget that is appropriate to the type of data that is to be displayed.** To find which widget is used on which field, see the documentation about Built-in Field classes.



```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

This would specify a form with a comment that uses a larger `Textarea` widget, rather than the default `TextInput` widget.

Coming to our example:

```
from django import forms

class UserRegistrationForm(forms.Form):

    GENDER = [(‘male’, ‘MALE’), (‘female’, ‘FEMALE’)]

    firstName = forms.CharField()
    lastName = forms.CharField()
    email = forms.CharField()
    gender = forms.CharField(widget=forms.Select(choices=GENDER))
    password = forms.CharField(widget=forms.PasswordInput)
    feedback = forms.CharField(widget=forms.Textarea)
```

For creating drop down, we need to following things:

1 - Create list of tuples of contents which will be the part of drop down.
2 - In a tuple, first element represent values in HTML tag and second element will be displayed in dropdown.
3 - Once list is created, you can pass create dropdown as shown above for ‘gender’ form field.

If you see, for ‘feedback’ form field, its input is created as text area and to do so, widget is changed for class CharField.

Default Django Validation

In this section, we will see default form fields validation that Django provides to every form that we create. In order to verify it, load you form in browser and without entering data, click on submit button, then you immediately see validation error.

So, by default, all the fields in the form are required. However, if you want to make optional, you can make `required` argument as `False` as shown below:

```
feedback = forms.CharField(required=False)
```

So, this is the default validation that you get from Django for free.

Let's add another field like 'Aadhar Card' just to see numeric validation:

```
feedback = forms.IntegerField()
```

Once you refresh, you will find that you can only enter integers and nothing else. So that's how the validation is taken care by Django.

Writing Custom Clean Methods

Form validation means verification of form data. Form validation is an important process when we are using model forms. When we validate our form data, we are actually converting them to python datatypes. Then we can store them in the database. This validation is necessary since it adds that layer of security where unethical data cannot harm our database.

NOTE:

There are certain JavaScript Validation methods too, that can be easier to implement, but they are not preferred at all as JavaScript is client-side scripting. If the client browser does not have JavaScript enabled, the data will not be validated. And, it causes some serious issues for developers.

Therefore, data validation on the server side is always necessary. Though, you can have JavaScript validation on the client side.

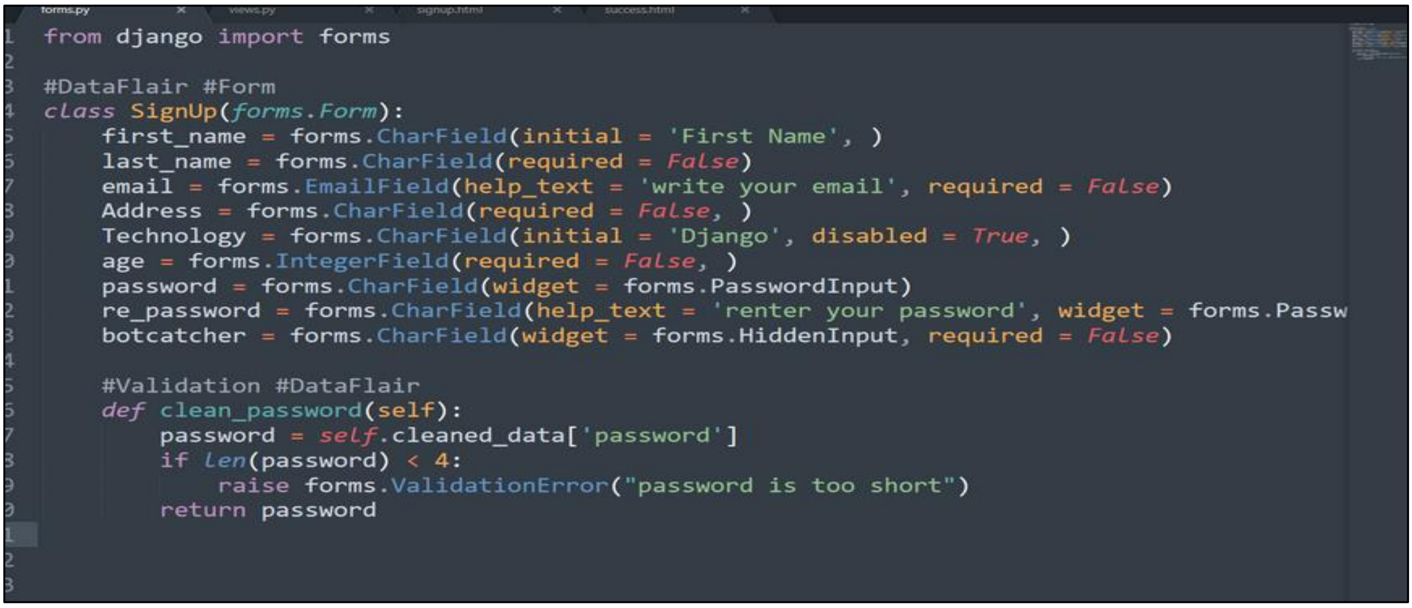
There are different ways of form validation we can use:

1 - Validation of a Particular Field

When we want certain fields to be validated, this method is used. We have to follow some Django conventions at this point to define our methods and other things.

In our forms.py file, add these lines of code:

```
#Validation #DataFlair
def clean_password(self):
password = self.cleaned_data['password']
if len(password) < 4:
raise forms.ValidationError("password is too short")
return password
```



```
from django import forms

#DataFlair #Form
class SignUp(forms.Form):
    first_name = forms.CharField(initial = 'First Name', )
    last_name = forms.CharField(required = False)
    email = forms.EmailField(help_text = 'write your email', required = False)
    Address = forms.CharField(required = False, )
    Technology = forms.CharField(initial = 'Django', disabled = True, )
    age = forms.IntegerField(required = False, )
    password = forms.CharField(widget = forms.PasswordInput)
    re_password = forms.CharField(help_text = 'renter your password', widget = forms.Passw
    botcatcher = forms.CharField(widget = forms.HiddenInput, required = False)

    #Validation #DataFlair
    def clean_password(self):
        password = self.cleaned_data['password']
        if len(password) < 4:
            raise forms.ValidationError("password is too short")
        return password
```

This is a method of your form class.

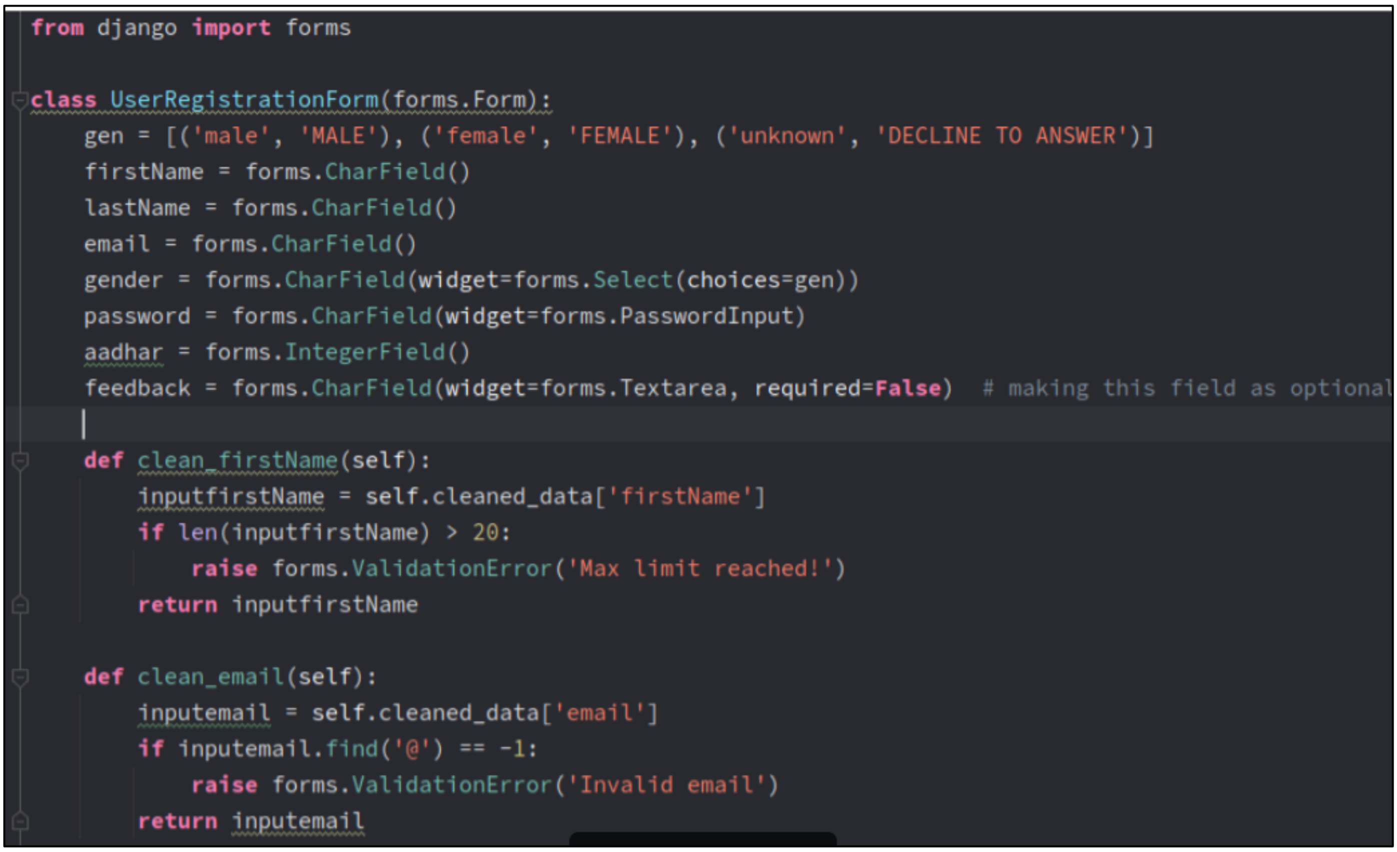
Some important points to note:

- The name of the function is special. You have to define it for each field you want to validate.

```
Syntax - clean_fieldname()
```

- Then we used **self.cleaned_data['field_name']**. This is also specific to a particular field. This list contains the cleaned data received from the form. The cleaned data means it's in python data-type format. It allows us to use python functions on the data we stored in a variable.
- After that, we used some conditions. The more important part is that we raised an error. The error messages can be customized.
- **ValidationErrors** are built in methods and subclasses of Django. They are used in models too.
- Always remember to return the data. It doesn't matter if the data is changing or not, this function should always return the data on which it is operated. That data may or may not be changed.
- You can run this code, but it is highly not recommended for professional use. This method requires much more code, which is actually against the Django principles. We have alternatives which are much more efficient.

So, in our User Registration example, made few changes with respect to custom validation:



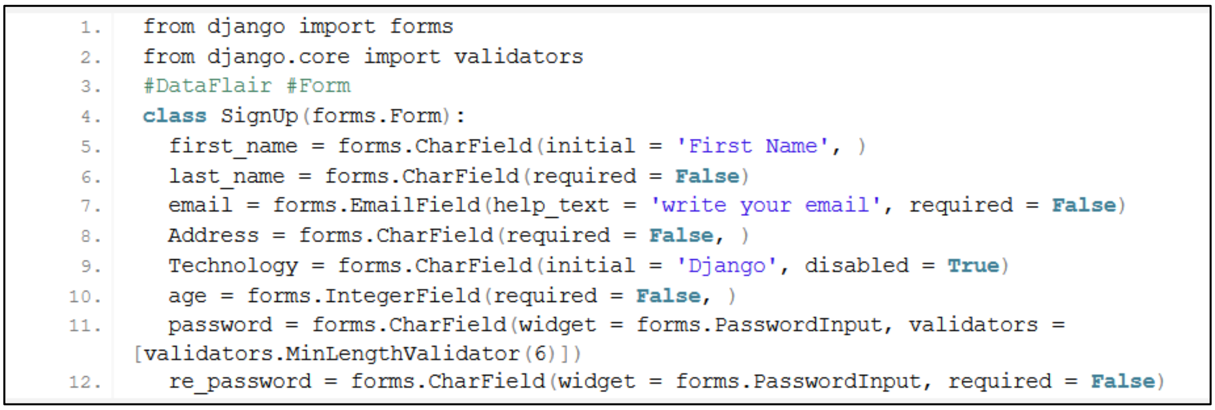
```
from django import forms

class UserRegistrationForm(forms.Form):
    gen = [('male', 'MALE'), ('female', 'FEMALE'), ('unknown', 'DECLINE TO ANSWER')]
    firstName = forms.CharField()
    lastName = forms.CharField()
    email = forms.CharField()
    gender = forms.CharField(widget=forms.Select(choices=gen))
    password = forms.CharField(widget=forms.PasswordInput)
    aadhar = forms.IntegerField()
    feedback = forms.CharField(widget=forms.Textarea, required=False)  # making this field as optional

    def clean_firstName(self):
        inputfirstName = self.cleaned_data['firstName']
        if len(inputfirstName) > 20:
            raise forms.ValidationError('Max limit reached!')
        return inputfirstName

    def clean_email(self):
        inputemail = self.cleaned_data['email']
        if inputemail.find('@') == -1:
            raise forms.ValidationError('Invalid email')
        return inputemail
```

2. Using is_valid() - You can use is_valid() when required to validate complete form-data. This validation will check for Python-datatypes. This function will return True or False in return. We often use this method when we just want our data to be python type. It validates all data. We use is_valid() in view functions as it returns values directly as True and False.

Example has been shown previously.

3 - Validation using Validators

We will now learn a professional way of form validation. It is very easy to use and provide much more control over fields. Let's modify our forms.py again.



```
from django import forms
from django.core import validators
#DataFlair #Form
class SignUp(forms.Form):
  first_name = forms.CharField(initial = 'First Name', )
  last_name = forms.CharField(required = False)
  email = forms.EmailField(help_text = 'write your email', required = False)
  Address = forms.CharField(required = False, )
  Technology = forms.CharField(initial = 'Django', disabled = True)
  age = forms.IntegerField(required = False, )
  password = forms.CharField(widget = forms.PasswordInput, validators = [validators.MinLengthValidator(6)])
  re_password = forms.CharField(widget = forms.PasswordInput, required = False)
```

Here we used something called validators. A validator is like a function which is callable. We basically use them to raise `ValidationErrors` if some conditions aren't fulfilled.

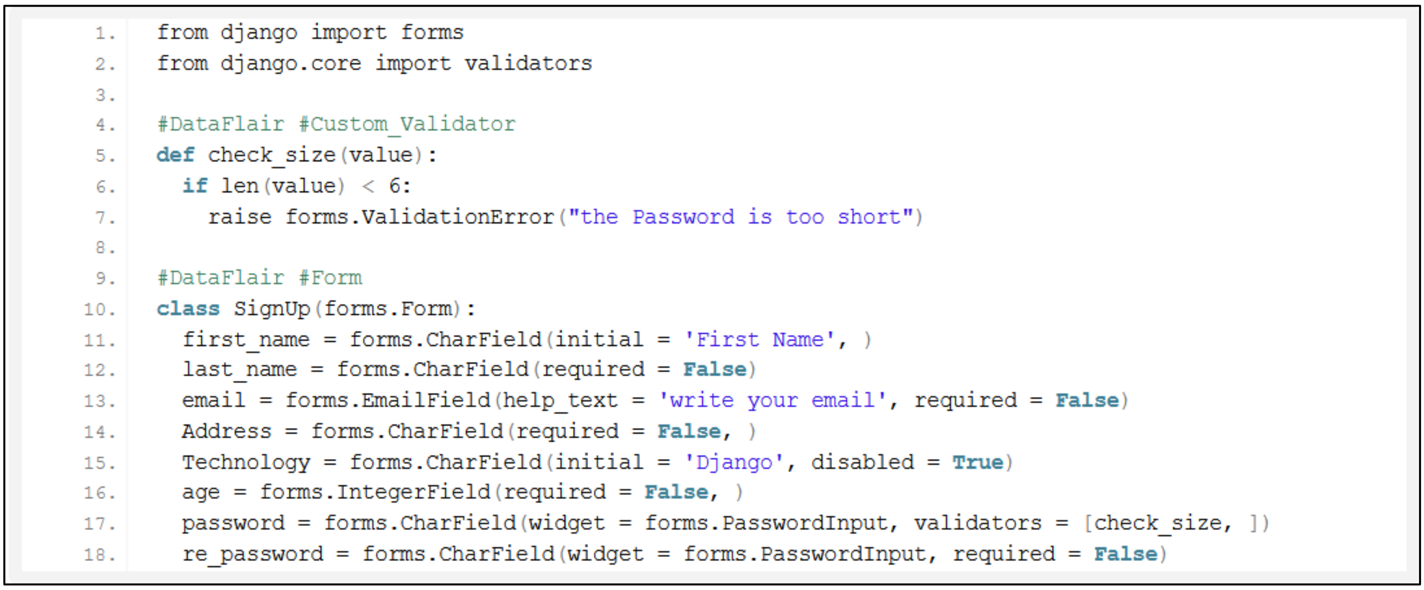
Validators is a general attribute to all the fields. There are certain validators for each field. Here we are using **MinLengthValidator** and passed an argument which is 6. This makes our password fields size to be at least 6 characters.

4 - Custom Validators

Suppose you have some fields which you want to validate and there may be no built-in validator of your use-case. Then, you can create your own custom validators.

- Just declare a function with the parameter value. The parameter name shall be that.
- Then, perform your desired validations on value. When value satisfies any criteria raise a ValidationError
- Add that function_name in validators argument of the chosen field, just as we used validators.

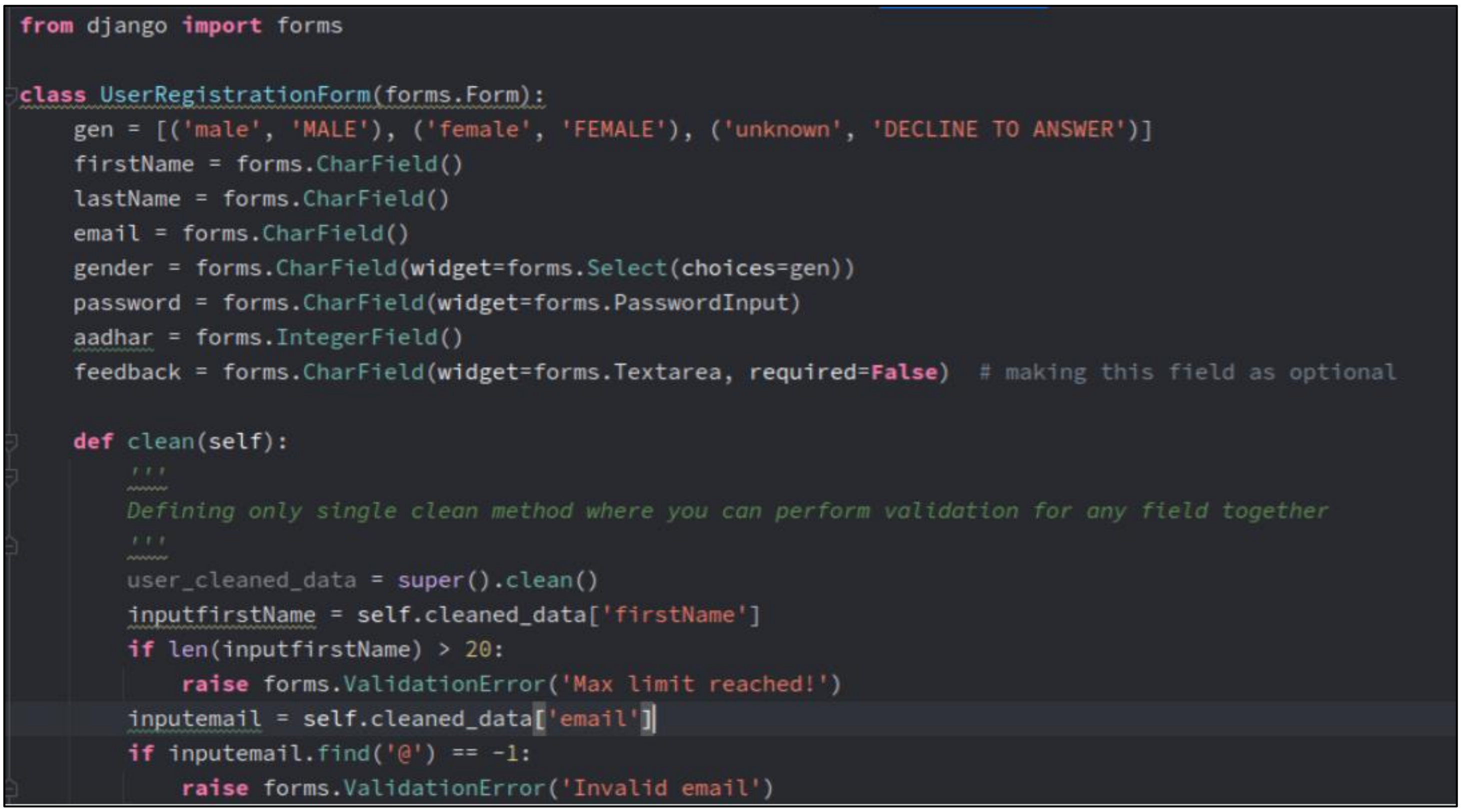
For example, edit your forms.py



```
from django import forms
from django.core import validators

#DataFlair #Custom_Validator
def check_size(value):
  if len(value) < 6:
    raise forms.ValidationError("the Password is too short")

#DataFlair #Form
class SignUp(forms.Form):
  first_name = forms.CharField(initial = 'First Name', )
  last_name = forms.CharField(required = False)
  email = forms.EmailField(help_text = 'write your email', required = False)
  Address = forms.CharField(required = False, )
  Technology = forms.CharField(initial = 'Django', disabled = True)
  age = forms.IntegerField(required = False, )
  password = forms.CharField(widget = forms.PasswordInput, validators = [check_size, ])
  re_password = forms.CharField(widget = forms.PasswordInput, required = False)
```

Single Clean method

We discussed defining clean method for each field which will perform the validation as per the code written inside these methods and return the fields. However, there is one more way in which we define a single clean method and perform validation for any of form fields together. Here, we don't need to return anything as cleaned data dictionary will be available to view by default and we simply need to raise the 'ValidationError'.



```
from django import forms

class UserRegistrationForm(forms.Form):
    gen = [('male', 'MALE'), ('female', 'FEMALE'), ('unknown', 'DECLINE TO ANSWER')]
    firstName = forms.CharField()
    lastName = forms.CharField()
    email = forms.CharField()
    gender = forms.CharField(widget=forms.Select(choices=gen))
    password = forms.CharField(widget=forms.PasswordInput)
    aadhar = forms.IntegerField()
    feedback = forms.CharField(widget=forms.Textarea, required=False)  # making this field as optional

    def clean(self):
        '''
        Defining only single clean method where you can perform validation for any field together
        '''
        user_cleaned_data = super().clean()
        inputfirstName = self.cleaned_data['firstName']
        if len(inputfirstName) > 20:
            raise forms.ValidationError('Max limit reached!')
        inputemail = self.cleaned_data['email']
        if inputemail.find('@') == -1:
            raise forms.ValidationError('Invalid email')
```

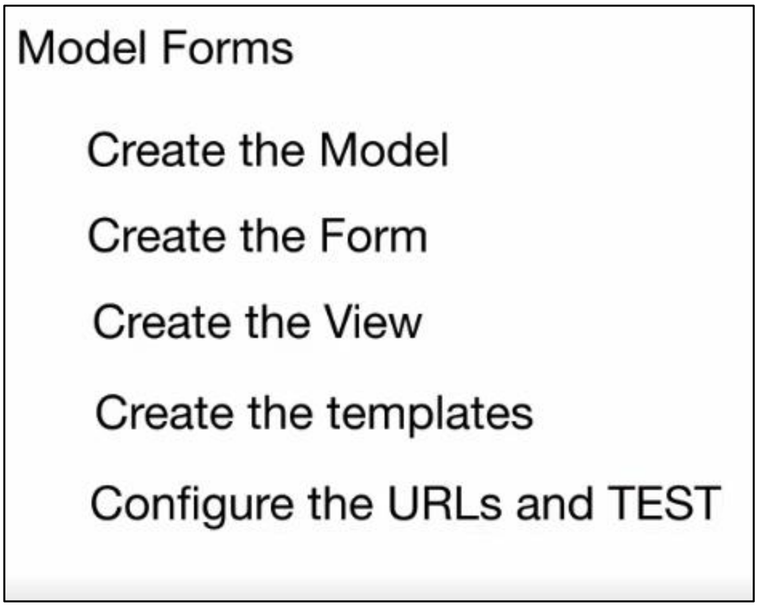
Model Forms

Introduction

Django ModelForm is a class that is used to directly convert a model into a Django form. If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For example, a User Registration model and form would have same quality and quantity of model fields and form fields. So instead of creating a redundant code to first create a form and then map it to the model in a view, we can directly use ModelForm. It takes as argument the name of the model and converts it into a Django Form. Not only this, ModelForm offers a lot of methods and features which automate the entire process and help remove code redundancy.

To create a ModelForm class, we need to add a class Meta inside it. Inside the Meta class, we instantiate the Django Model class of our choice. For example, we have created Employee model class previously and now we want to create a form based on that model class so inside meta, we will pass the same and number of fields can be limited or added inside meta class.

This is what we are going to do in this section.



Model Forms

Create the Model

Create the Form

Create the View

Create the templates

Configure the URLs and TEST

Create the Project and Model

I have already created project and below are the fields defined in the models.py inside one of my app. We are created form that basically assigned tasks with the start and end date and comes with the priority. Other than that, we will also create the list of all assigned tasks.

```
from django.db import models

class Project(models.Model):
    startDate = models.DateField()
    endDate = models.DateField()
    name = models.CharField(max_length = 40)
    assignedTo = models.CharField(max_length = 40)
    priority = models.IntegerField()
```

Create the Model Form

Next step is to create the form using the model which we have already developed. So, for that, go to your app and create `forms.py`, you know like we had discussed earlier.

Once this done, create a class, let's say `ProjectForm(forms.ModelForm)` which is extending class `forms.ModelForm.`

Within this class create a Meta class and within meta class, provide 2 things -
- the model name from which you want to create model form
- number of fields you want from that model

If you want to include all fields set `fields = '__all__'`

For now, we are including every field of our model class Project. Below is the code inside forms.py

```
from django import forms
from .models import Project

class ProjectForm(forms.ModelForm):
    class Meta:
        model = Project
        fields = '__all__'
```

Create the Views

We will be creating 3 views for our purpose:

1 - index.html - In this page, there will be option for user, which would be 2 links, to list all projects and to add project.

2 - listProjects.html - It will list all projects from the database.

3 - addProject.html - Here user can add the data, in this case, assign a project.

```
from django.shortcuts import render
from .forms import ProjectForm
from .models import Project

def index(request):
    return render(request, 'firstapp/index.html')

def listProject(request):
    projectList = Project.objects.all()
    return render(request, 'firstapp/listProject.html', {'projects':
projectList})

def addProject(request):
    form = ProjectForm()
    if form.method == "POST":
        form = ProjectForm(request.POST)
        if form.is_valid():
            form.save()
        return index(request)
    return render(request, 'firstapp/addProject.html', {'form':
form})
```

Create the Index Template

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Projects</title>
</head>
<body>
    <a href="/listProjects"> List Projects </a>
    <a href="/addProject"> Add Project </a>
</body>
</html>
```

Create the List Projects Template

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Project List</title>
</head>
<body>
    {% if projects %}
    <table>
        <thead>
            <th>Name</th>
            <th>Start Date</th>
            <th>End Date</th>
            <th>Assigned To</th>
            <th>Priority</th>
        </thead>
       {% for project in projects %}
        <tr>
            <td>{{ project.name }}</td>
            <td>{{ project.startDate }}</td>
            <td>{{ project.endDate }}</td>
            <td>{{ project.assignedTo }}</td>
            <td>{{ project.priority }}</td>
        </tr>
        {% endfor %}
    </table>
    {% else %}
    <p>No Projects are active</p>
    {% endif %}
</body>
</html>
```

Create the Add Project Template

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Add Project</title>
</head>
<body>
    <h1>Add Project</h1>
    <form method="POST">
        <table>
        {{ form.as_table }}
        {% csrf_token %}
        </table>
        <br/><input type="submit" name="" value="Add Project">
    </form>
</body>
</html>
```

After creating all these templates, configure the URLs accordingly.

Configure MySQL

- Make sure database is configured correctly in `settings.py`
- If this is first time you are running database with django, make sure to run makemigration and migrate commands.
- Once done, db will be created in MySql and you are good to go.

Django Tables from migrate

When you run the migrate command, along with our application table, there are lot of other tables get created. These tables have the auth and django prefix.

auth prefixes basically related to security point of view.

CRUD Using Function Based View

Django is a Python-based web framework which allows you to quickly create web application without all of the installation or dependency problems that you normally will find with other frameworks. Django is based on MVT (Model View Template) architecture and revolves around CRUD (Create, Retrieve, Update, Delete) operations. CRUD can be best explained as an approach to building a Django web application. In general CRUD means performing Create, Retrieve, Update and Delete operations on a table in a database. Let’s discuss what actually CRUD means,

Create – create or add new entries in a table in the database.
Retrieve – read, retrieve, search, or view existing entries as a list(List View) or retrieve a particular entry in detail (Detail View)
Update – update or edit existing entries in a table in the database
Delete – delete, deactivate, or remove existing entries in a table in the database

Before proceeding further, prepare your project and app for demonstration purpose. Make sure all the configurations are completed.

After you have a project and an app, let’s create a model of which we will be creating instances through our view.

```
from django.db import models

class Student(models.Model):
    firstName = models.CharField(max_length=40)
    lastName = models.CharField(max_length=40)
    testScore = models.FloatField()
```

Now we will create a Django ModelForm for this model.

```
from django import forms
from .models import Studentclass StudentForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = '__all__'
```

Implement Read

Once model and modelform is created, we will then create a view to read the data from the database and template that will display all the student information.

View to read data:

```
from django.shortcuts import render
from .models import Student


def getStudent(request):
        students = Student.objects.all()
        return render(request, 'fbvCRUDApp/index.html', {'student':
students})
```

Template to display the data:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Student Information</title>
</head>
<body>
    <h1>Students</h1>
    <table border="1">
        <thead>
            <th>Id</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Test Score</th>
        </thead>
        {% for student in students %}
        <tr>
            <td>{{ student.id }}</td>
            <td>{{ student.firstName }}</td>
            <td>{{ student.lastName }}</td>
            <td>{{ student.testScore }}</td>
        </tr>
        {% endfor %}
    </table></body>
</html>
```

Note - There will be changes made to this template but for now, this fulfils our purpose.

Run the Migrations and Test our Read functionality

Model class we have created above, we have to migrate it so that table is created inside our database. As we have discussed before already, use commands 'makemigration' and 'migrate' for it.

Also, configure the URLs.

Once, all done, you start the server and test it.

Implement Create

Now, we will perform create operation which means that from our web application itself, we will allow user to create entry for students.

Let's create the view for it.

```
from django.shortcuts import render, redirect
from .models import Student
from .forms import StudentForm


def createStudent(request):
    form = StudentForm()
    if request.method == 'POST':
        form = StudentForm(request.POST)
        if form.is_valid():
            form.save()
        return redirect('/')
    return render(request, 'fbvCRUDApp/create.html', {'form':form})
```

Let's create template create.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Create Student</title>
</head>
<body>
    <h2>Create Student</h2>
    <form method="Post">
        <table>
            {{ form.as_table }}
            {% csrf_token %}
        </table>
        <br/>
        <input type="submit" name="" value="Save">
    </form>
</body>
</html>
```

Now after that, in index.html, add a link 'Add Student' which will add student like below:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Student Information</title>
</head>
<body>
    <h1>Students</h1>
    <table border="1">
        <thead>
            <th>Id</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Test Score</th>
        </thead>
        {% for student in students %}
        <tr>
            <td>{{ student.id }}</td>
            <td>{{ student.firstName }}</td>
            <td>{{ student.lastName }}</td>
            <td>{{ student.testScore }}</td>
        </tr>
        {% endfor %}
    </table>
<br/><br/><br/>
<a href=”/create”> Add Student</a>
</body>
</html>
```

Please take care of URL configurations.

Delete

To delete, you need to write a view like below:

```
def deleteStudent(request, id):
    student = Student.objects.get(id=id)
    student.delele()
    return redirect('/fbvCRUDapp/')
```

Afterwards, configure the URL for this operation and update the index.html to add delete link like below:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Student Information</title>
</head>
<body>
    <h1>Students</h1>
    <table border="1">
        <thead>
            <th>Id</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Test Score</th>
        </thead>
        {% for student in students %}
        <tr>
            <td>{{ student.id }}</td>
            <td>{{ student.firstName }}</td>
            <td>{{ student.lastName }}</td>
            <td>{{ student.testScore }}</td>
            <td><a href =
“/fbvCRUDapp/delete/{{student.id}}”>Delete</a></td>
        </tr>
        {% endfor %}
    </table>
<br/><br/><br/>
<a href=”/create”> Add Student</a>
</body>
</html>
```

Update View and Template

Update View refers to a view (logic) to update a particular instance of a table from the database with some extra details. It is used to update enteries in the database.

Let’s start with creating link of update in index.html

```
<td><a href = “/fbvCRUDapp/update/{{student.id}}”>Update</a></td>
```

Just add this in index.html below delete link.

Now, create view to update the entry like below:

```
def updateStudent(request, id):
    student = Student.objects.get(id=id)
    if request.method == 'POST':
        form = StudentForm(request.POST, instance=student)
        if form.is_valid():
            form.save()
            return redirect('/fbvCRUDapp/getStudent')
    return render(request, 'fbvCRUDApp/update.html', {'student':
student}
```

See in bold text, as per the id, we are selecting the object align to it.

For templates, using Django form is a bit different process which we will see in next section, however, for now, creating form fields manually in template.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Update</title>
</head>
<body>
    <h2>Update Student</h2>
    <form method="post">
        {% csrf_token %}
        First Name:<input type="text" name="firstName" value="{{student.firstName}}"><br/>
        Last Name:<input type="text" name="lastName" value="{{student.lastName}}"><br/>
        Test Score:<input type="text" name="testScore" value="{{student.testScore}}"><br/>
        <input type="submit" name="" value="Update">
    </form>
</body>
</html>
```

Using Django form for update

When we had implemented the update operation view, we were fetching the student data and then we were sending that date to template update.html where we were creating form manually using the data passed. However, when we had created the 'createStudent' view, in that we were sending the form to template 'create.html' so we don't need to create form manually in HTML file. Also, since it was ModelForm, data entered by user were directly updating in the db.

Now, we try to pass form in the template so that we don't need to update manually and entered data by user got updated in db directly. We'll just need to create few changes shown below:

```
def updateStudent(request, id):

    student = Student.objects.get(id=id)
    form = StudentForm(instance=student)
    if request.method == 'POST':
        form = StudentForm(request.POST, instance=student)
        if form.is_valid():
            form.save()
            return redirect('/fbvCRUDapp/getStudent')
    return render(request, 'fbvCRUDApp/update.html', {'form': form})
```

Update the update.html template accordingly like below:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

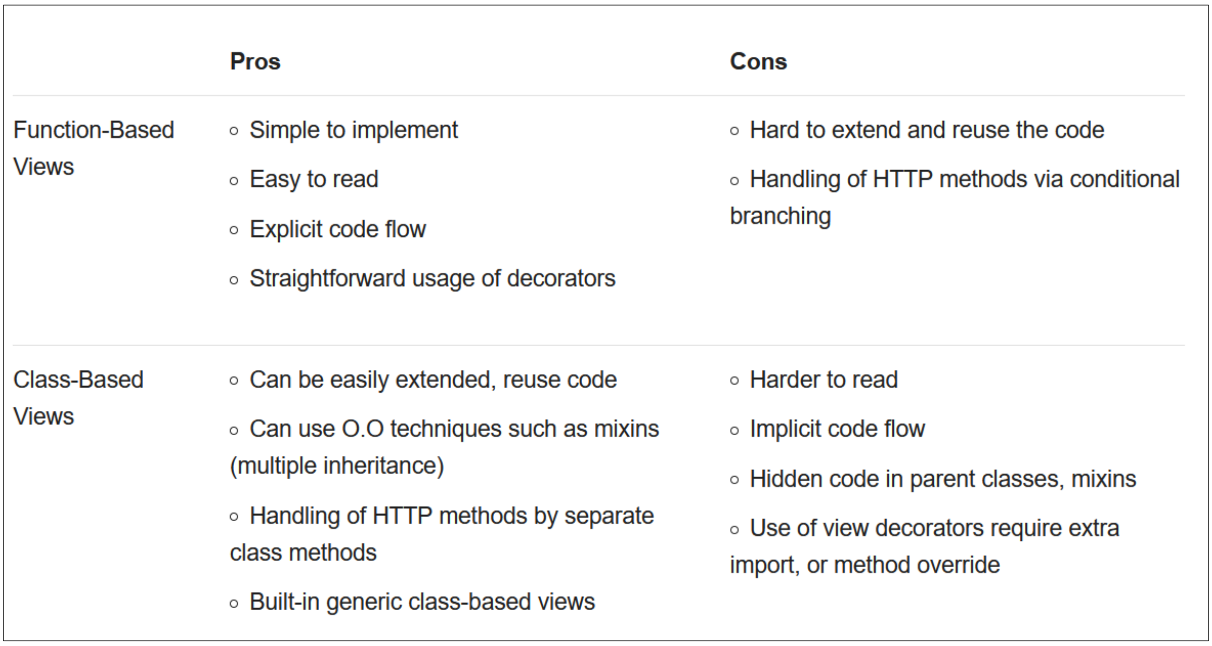
```
        <meta charset="UTF-8">
        <title>Update</title>
    </head>
    <body>
        <h2>Update Student</h2>
        <form method="post">
            {% csrf_token %}
            <table>
            {{form.as_table}}
             </table><br/>
            <input type="submit" name="" value="Update">
        </form>
    </body>
    </html>
```

Class Based Views

Introduction

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

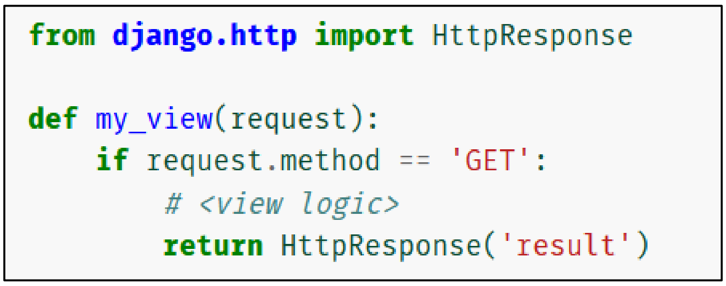
- Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.



	Pros	Cons
Function-Based Views	○ Simple to implement ○ Easy to read ○ Explicit code flow ○ Straightforward usage of decorators	○ Hard to extend and reuse the code ○ Handling of HTTP methods via conditional branching
Class-Based Views	○ Can be easily extended, reuse code ○ Can use O.O techniques such as mixins (multiple inheritance) ○ Handling of HTTP methods by separate class methods ○ Built-in generic class-based views	○ Harder to read ○ Implicit code flow ○ Hidden code in parent classes, mixins ○ Use of view decorators require extra import, or method override

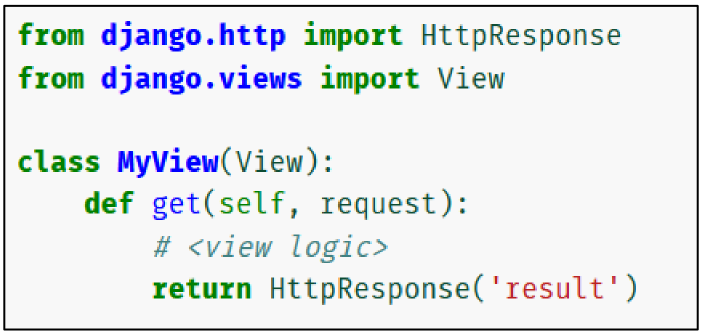
At its core, a class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

So, where the code to handle HTTP GET in a view function would look something like:



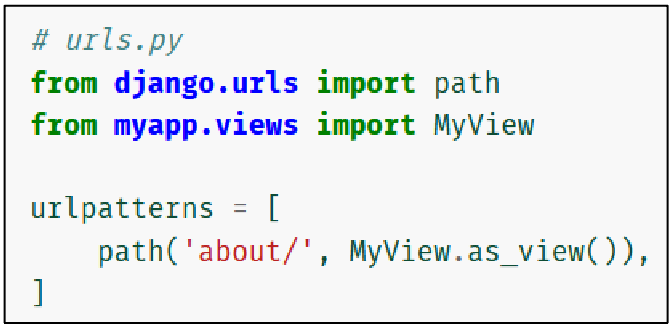
```
from django.http import HttpResponse

def my_view(request):
    if request.method == 'GET':
        # <view logic>
        return HttpResponse('result')
```

In a class-based view, this would become:



```
from django.http import HttpResponse
from django.views import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

Because Django’s URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an `as_view()` class method which returns a function that can be called when a request arrives for a URL matching the associated pattern. The function creates an instance of the class, calls `setup()` to initialize its attributes, and then calls its `dispatch()` method. dispatch looks at the request to determine whether it is a `GET`, `POST`, etc, and relays the request to a matching method if one is defined, or raises `HttpResponseNotAllowed` if not:

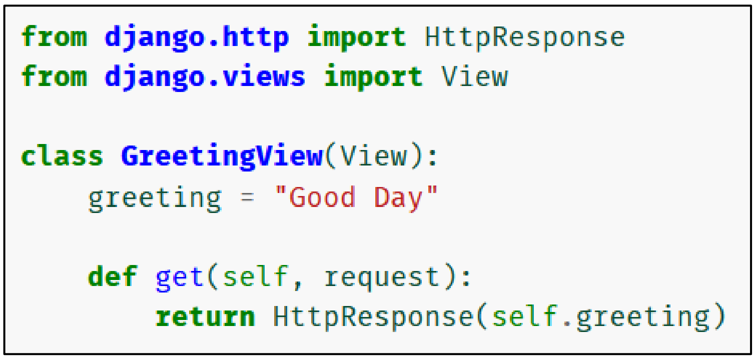


```
# urls.py
from django.urls import path
from myapp.views import MyView

urlpatterns = [
    path('about/', MyView.as_view()),
]
```

It is worth noting that what your method returns is identical to what you return from a function-based view, namely some form of HttpResponse. This means that http shortcuts or TemplateResponse objects are valid to use inside a class-based view.

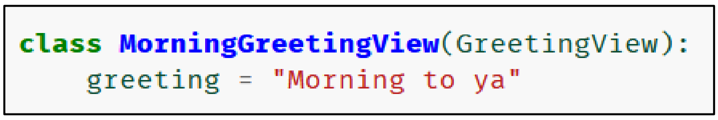
While a minimal class-based view does not require any class attributes to perform its job, class attributes are useful in many class-based designs, and there are two ways to configure or set class attributes.

The first is the standard Python way of subclassing and overriding attributes and methods in the subclass. So that if your parent class had an attribute greeting like this:



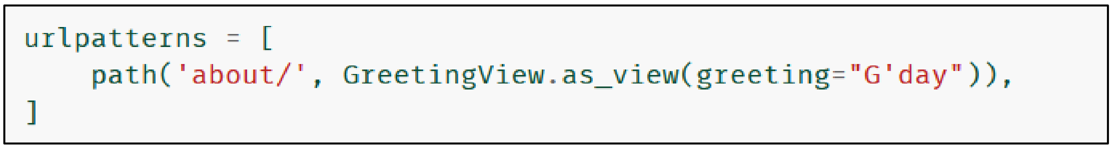
```
from django.http import HttpResponse
from django.views import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

You can override that in a subclass:

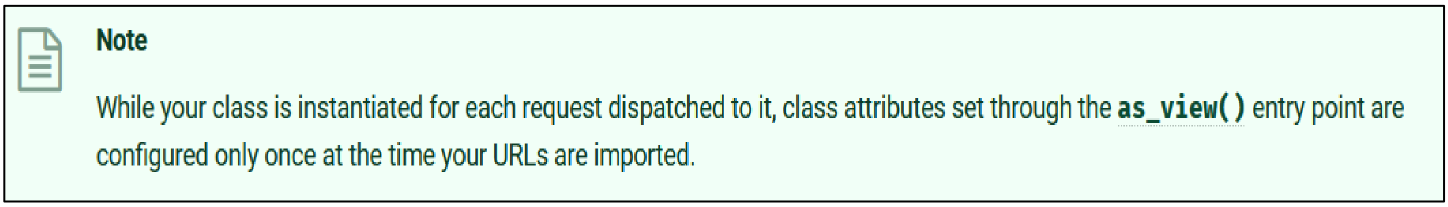


```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

Another option is to configure class attributes as keyword arguments to the `as_view()` call in the URLconf:



```
urlpatterns = [
    path('about/', GreetingView.as_view(greeting="G'day")),
]
```



Note

While your class is instantiated for each request dispatched to it, class attributes set through the `as_view()` entry point are configured only once at the time your URLs are imported.

Hands on Steps

We will be implementing the student example like we have discussed above using class-based view.

Create your project and relevant apps in advance. Also, copy and paste the Student class in your app you are currently using to demonstrate the class-based view from our function-based view example.

Create list view

To create the list view, you need to extend your StudentListView (which we have created in views) to ListView and set the model (predefined variable in django) variable with model class Student like below:

*So, our class extending ListView which will fetch all the students for us and send that data to a template and default name for that template will be <model_name>_list.html. So, we will be creating with this name. Also, by default, the context parameter (data) will be available with default name **context_object_name.** So, in our example, student data will be available in variable 'student_list'. You can loop through it as per your requirement.*

If you want to change the name, you can use this variable like 'template_name' to change the template name by assigning the new to it like below:

template_name= 'new_template_name'

and for data variable, use 'context_object_name' in a similar manner.

```
From django.views.generic. import ListView

class StudentListView(ListView):
    model = Student
    # default template_name is student_list.html
    # default contect_object_name is student_list
```

Create List Template

To create template (in this context, student_list.html), we have to create a directory with name 'templates' and under that create one more directory with the name same as of your current app and all this we will be created under your current working app.

Once all this done, create a student_list.html file and you can copy and paste the data from our previous example of student implemented with function-based view. We will make changes later.

Implement Student Details

In this section, we will learn how to use the 'detail view' from 'generic'. We will make ID of student as link and when you click it, it will generate all the information of that student on a separate page. Now for all that, create a class say, StudentDetailView and extend it with DetailView like below.

In this case, default template name would be student_detail.html and
default context_object_name would be student.

Now, create the template under your 'templates' directory of your app with name 'student_detail.html'

```
from django.views.generic import DetailView

class StudentDetailViews(DetailView):
    model = Student
    # default template_name is student_detail.html
    # default contect_object_name is student
```

Template would be like this when you click on the ID of student:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Student Details</title>
</head>
<body>
    <h1>Student Details</h1>
    <ol>
        <h3><li>First Name: {{student.firstName}}</li></h3>
        <h3><li>First Name: {{student.lastName}}</li></h3>
        <h3><li>First Name: {{student.testScore}}</li></h3>
    </ol>
</body>
</html>
```

Create

In this section, we will work on the create operation. For that, import CreateView from generic and inherit your class StudentCreateView.

Inside your StudentCreateView, you will mention the **model** like we have done in StudentListView and StudentDetailViews class. In addition to model, you also need to mention the fields inside the tuple, as in which all field you want to create.

In this case, default template name would be student_form.html and
default context_object_name would be form.

```
from django.views.generic import CreateView
class StudentCreateView(CreateView):
    model = Student
    fields = ('firstName', 'lastName', 'testScore')
```

student.form.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Create Student Entry</title>
</head>
<body>
    <h1>Create Student</h1>
    <form method="post">
        <table>
            {% csrf_token %}
            {{form.as_table}}
        </table>
        <input type='Submit' name='' value='Save'>
    </form>
</body>
</html>
```

Now, once user enters all the field and submitted it, after submitting, redirecting can be defined inside this class or in the student model class.

1 - Defining redirecting inside model class student (This is the preferred way)

For that, create the method inside model class Student which will return the view once the data will be submitted by user using the **reverse() function which take you back to the url name which you are passing in it**. URL name you can pass in urls.py like shown below. Since this URL expected ID, for that you also have to define keyword argument like below.

urls.py

```
from django.contrib import admin
from django.urls import path
from . import views


urlpatterns = [
    path('', views.GreetingView.homepage),
    path('getview/', views.GreetingView.as_view(greeting="<b>Overriding from as_view()
function</b>")),
    path('studentproject', views.StudentListView.as_view()),
    path('<int:pk>/', views.StudentDetailView.as_view(), name='detail'),
    path('create/', views.StudentCreateView.as_view())
]
```

models.py

```
from django.db import models
from django.urls import reverse


class Student(models.Model):
    firstName = models.CharField(max_length=40)
    lastName = models.CharField(max_length=40)
    testScore = models.FloatField()


def get_absolute_url(self):
        return reverse('detail', kwargs={'pk': self.pk})
```

2 - Second way is to use ‘success_url’ attribute. We will see it when we discuss delete operation as in that case, above of redirecting will not work as it redirecting to the created or updated entry, however, after deleting, record will not exist to redirect and it will throw error.

Update

For updating any record of student, we will create a class which will inherit UpdateView class like below.

```
from django.views.generic import UpdateView
class StudentUpdateView(UpdateView):
    model = Student
    fields = ('testScore',)
```

student_details.html (added update and delete link in this template and removed from student_list.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Student Details</title>
</head>
<body>
    <h1>Student Details</h1>
    <ol>
        <h3><li>First Name: {{student.firstName}}</li></h3>
        <h3><li>First Name: {{student.lastName}}</li></h3>
        <h3><li>First Name: {{student.testScore}}</li></h3>
        <h3><li><a
href="/cbvapp/delete/{{student.id}}">Delete</a></li></h3>
        <h3><li><a
href="/cbvapp/update/{{student.id}}">Update</a></li></h3>
    </ol>
</body>
</html>
```

Delete

For deleting any record of student, we will create a class which will inherit DeleteView class like below:

```
from django.views.generic import DeleteView


class StudentDeleteView(DeleteView):
    model = Student
    success_url = reverse_lazy(‘student’) #redirecting to list of student page.
```

Here student is the name passed in the URL mapping of StudentListView in urls.py file like below:

```
from django.contrib import admin
from django.urls import path
from . import views


urlpatterns = [
    path('', views.GreetingView.homepage),
    path('getview/', views.GreetingView.as_view(greeting="<b>Overriding from
as_view() function</b>")),
    path('studentproject', views.StudentListView.as_view(), name='student'),
    path('<int:pk>/', views.StudentDetailView.as_view(), name='detail'),
    path('create/', views.StudentCreateView.as_view()),
    path('update/<int:pk>', views.StudentUpdateView.as_view())
]
```

Next thing we are implementing is to provide confirmation template of delete, if user is sure about deleting this record. For that, we will be creating the template `student_confirm_delete.html`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Delete Student</title>
</head>
<body>
    <h2>Delete Student {{student.firstName}} ?</h2>
    <form method="post">
        <input type="submit" value="Confirm">
        <a href="/cbvapp/{{student.id}}">Cancel</a>
        {%csrf_token%}
    </form>
</body>
</html>
```

More About Templates

Use template Inheritance on CBV

In this section, we will apply template inheritance concept in our class-based views example. You will see that beginning part is common in all the template. So, we will create a base template and put this common part and other templates will inherit this template.

base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>

{% block body_block %}
{% endblock %}

</body>
</html>
```

Let’s extend the student_confirm_delete.html like below:

```
<!DOCTYPE html>
{% extends 'cbvAPP/base.html' %}
{% block body_block %}
<body>
    <h2>Delete Student {{student.firstName}} ?</h2>
    <form method="post">
        <input type="submit" value="Confirm">
        <a href="/cbvapp/{{student.id}}">Cancel</a>
        {%csrf_token%}
    </form>
{% endblock %}
```

Filters in Action

We have discussed earlier but this is how we can use filter in our student project. In student_list.html, we can pass filters like below:

```
<h3><li>First Name: {{student.firstName}}</li></h3>
<h3><li>First Name: {{student.lastName | lower}}</li></h3>
<h3><li>First Name: {{student.testScore| add:’2’}}</li></h3>
```

lower - to convert to lowercase
add - to perform addition operation. In this case, 2 is getting added to each test score. Here, 2 is parameter passed to filter add.

User Defined Filter

To create a custom filter, first create a folder under your working app with name `‘templatetags’` and you can make it a python package by creating a __init__.py inside that folder.

Now, create a python file and let’s say it `custFilters.py`. Inside this file we will be defining our custom filters.

To start writing your custom filter, you need to first import template module from django and create the instance of Library class which is a part of template module like below. These 2 are mandatory steps:

```
from django import template

register = template.Library()
```

Afterwards, we can define our customer filters. Below we are defining our own lower filter which lowers the first 3 elements of a string.

```
def custLower(value):

   result = value[0:3].lower()+value[3:]
   return result
```

Once you create your custom filter, you need to register it using instance of Library class we have defined above.

```
register.filter('myLower', custLower)
```

Now, you use this custom filter with name 'myLower'

Using Custom Filter

In order to your custom filter in a template, you need to first load python file inside which you have defined your custom filters like below:

```
{% load custFilters %}
```

Once you've loaded your custom filter file, you can use your filter with the name you register it.

```
        <h3><li>First Name: {{student.firstName}}</li></h3>
        <h3><li>First Name: {{student.lastName | myLower}}</li></h3>
```

Note - If you are getting error, try restarting your server again.

Registering a filter using a decorator

In this section, we will learn another easy way to register our filter using decorator. For that, you need to use the decorator `register.filter(name=<desired name you want to give to your filter>)`

```
@register.filter(name = 'mylower')
def custLower(value):
    result = value[0:3].lower()+value[3:]
    return result
```

Passing Arguments in your filter

In this section, we will learn how to pass argument to the filters you create. Below example demonstrate the same:

```
@register.filter(name = 'myappend')
def custAppend(value, arg):
    result = str(arg)+value
    return result
```

In template, you can use this filter like:

```
<h3><li>First Name: {{student.firstName | myappend:'Mr'}}</li></h3>
```

Session Management

Statelessness of HTTP

Network Protocols for web browser and servers are categorized into two types: Stateless Protocol, and Stateful protocol.

These two protocols are differentiated based on the requirement of server or server-side software to save status or session information.

1. Stateless Protocol:
Stateless Protocols are the type of network protocols in which Client send request to the server and server response back according to current state. It does not require the server to retain session information or a status about each communicating partner for multiple request.

HTTP (Hypertext Transfer Protocol), UDP (User Datagram Protocol), DNS (Domain Name System) are the example of Stateless Protocol.

Silent features of Stateless Protocols:

- Stateless Protocol simplify the design of Server.
- The stateless protocol requires less resources because system do not need to keep track of the multiple link communications and the session details.
- In Stateless Protocol each information packet travel on its own without reference to any other packet.
- Each communication in Stateless Protocol is discrete and unrelated to those that precedes or follow.

2. Stateful Protocol:

In Stateful Protocol If client send a request to the server then it expects some kind of response, if it does not get any response then it resends the request. FTP (File Transfer Protocol), Telnet are the example of Stateful Protocol.

Silent features of Stateful Protocol:

- Stateful Protocols provide better performance to the client by keeping track of the connection information.
- Stateful Application require Backing storage.
- Stateful request are always dependent on the server-side state.
- TCP session follow stateful protocol because both systems maintain information about the session itself during its life.

Comparisons between Stateless and Stateful Protocol

Stateless Protocol	Stateful Protocol
Stateless Protocol does not require the server to retain the server information or session details.	Stateful Protocol require server to save the status and session information.
In Stateless Protocol, there is no tight dependency between server and client.	In Stateful protocol, there is tight dependency between server and client
The Stateless protocol design simplify the server design.	The Stateful protocol design makes the design of server very complex and heavy.
Stateless Protocols works better at the time of crash because there is no state that must be restored, a failed server can simply restart after a crash.	Stateful Protocol does not work better at the time of crash because stateful server have to keep the information of the status and session details of the internal states.
Stateless Protocols handle the transaction very fast.	Stateful Protocols handle the transaction very slowly.
Stateless Protocols are easy to implement in Internet.	Stateful protocols are logically heavy to implement in Internet.

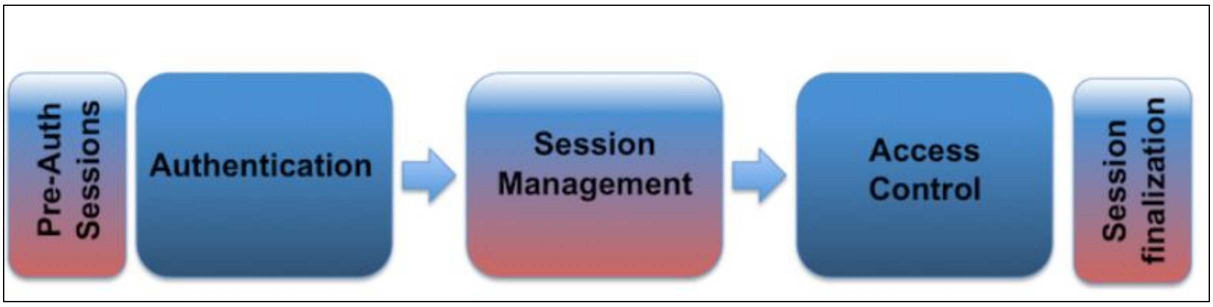
Session Management

A web session is a sequence of network HTTP request and response transactions associated to the same user. Modern and complex web applications require the retaining of information or status about each user for the duration of multiple requests. Therefore, sessions provide the ability to establish variables – such as access rights and localization settings – which will apply to each and every interaction a user has with the web application for the duration of the session.

Web applications can create sessions to keep track of anonymous users after the very first user request. An example would be maintaining the user language preference. Additionally, web applications will make use of sessions once the user has authenticated. This ensures the ability to identify the user on any subsequent requests as well as being able to apply security access controls, authorized access to the user private data, and to increase the usability of the application. Therefore, current web applications can provide session capabilities both pre and post authentication.

Once an authenticated session has been established, the session ID (or token) is temporarily equivalent to the strongest authentication method used by the application, such as username and password, passphrases, one-time passwords (OTP), client-based digital certificates, smartcards, or biometrics (such as fingerprint or eye retina).

HTTP is a stateless protocol where each request and response pair are independent of other web interactions. Therefore, in order to introduce the concept of a session, it is required to implement session management capabilities that link both the authentication and access control (or authorization) modules commonly available in web applications:



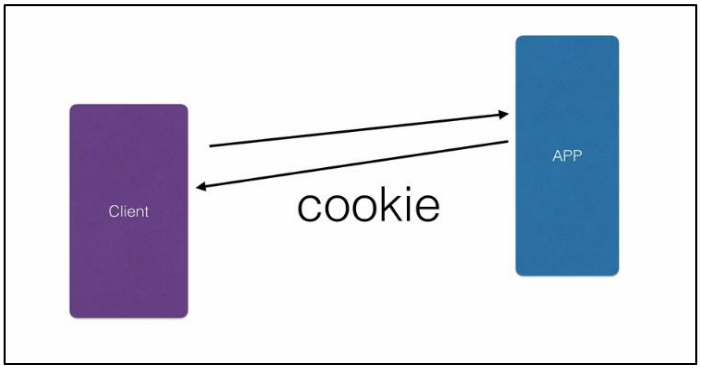
The session ID or token binds the user authentication credentials (in the form of a user session) to the user HTTP traffic and the appropriate access controls enforced by the web application. The complexity of these three components (authentication, session management, and access control) in modern web applications, plus the fact that its implementation and binding resides on the web developer's hands (as web development framework do not provide strict relationships between these modules), makes the implementation of a secure session management module very challenging.

The disclosure, capture, prediction, brute force, or fixation of the session ID will lead to session hijacking (or side jacking) attacks, where an attacker is able to fully impersonate a victim user in the web application. Attackers can perform two types of session hijacking attacks, targeted or generic. In a targeted attack, the attacker's goal is to impersonate a specific (or privileged) web application victim user. For generic attacks, the attacker's goal is to impersonate (or get access as) any valid or legitimate user in the web application.

Django Session Management

In this section we will learn how to manage or track sessions in spite of HTTP being stateless. One of the ways to do it is to use **cookies** which is a HTTP feature.

So, when client request comes to our app, app will process the request, send back the response and in the response, it will set a cookie.



Let's say if you are working on the shopping cart app, this cookie can have the entire shopping cart information. So, the entire shopping cart will go back to the client's browser and client browser is responsible for sending that shopping cart or that cookie back to our app when the next user request will come in. This is like a work around we can use but this is not the preferred way, we will be using the `Session API` provides by Django.

So, when the user request comes in, if we use the Django session API, by simply using `request.session,` the django server will internally create a session for us and it will generate a unique ID and that ID will be sent back to the client as a cookie and that session ID can be map to a particular dictionary and we can store our shopping cart or whatever session information we want on the server inside this dictionary object. So, the session can be ended at a later point when the user logs out or whenever we want using the API method. So, it's the client responsibility or the browser's responsibility to send the session the ID back as a cookie.

If the cookies are disabled for some reason on the client's browser, the django server will fall back to `URL Rewriting` where it will append this session ID to every URL in the application.

If it is a HTML form, django form will automatically include a hidden field and it will send this session ID as hidden field. The key here is that since HTTP does not have support to remember things, we have to pass in a token like session ID so that the client keep sending that back and our server knows that it is associated with a particular session by looking at that unique value.

Another good explanation from Data Flair

As we have seen in the previous tutorial, HTTP is a stateless protocol, where every request made is always new to the server. The request on the server is always treated as if the user is visiting the site for the first time. This poses some problems like you can't implement user login and authentication features. These problems were actually solved by Cookies.

What are Cookies?

Cookies are small text files stored and maintained by the browser. It contains some information about the user and every time a request is made to the same server, the cookie is sent to the server so that the server can detect that the user has visited the site before or is a logged in user.

The cookies also have their drawbacks and a lot of times they become a *path for the hackers and malicious websites to damage the target site.*

Drawbacks of Cookies

Since cookies store locally, the browser gives control to the user to accept or decline cookies. Many websites also provide a prompt to users regarding the same.

Cookies are plain text files, and those cookies which are not sent over HTTPS can be easily caught by attackers. Therefore, it can be dangerous for both the site and the user to store essential data in cookies and returning the same again and again in plain text.

These are some of the more common problems that web developers were facing regarding cookies.

What are Sessions?

After observing these problems of cookies, the web-developers came with a new and more secure concept, Sessions.

The session is a semi-permanent and two-way communication between the server and the browser.

Let's understand this technical definition in detail. Here semi means that session will exist until the user logs out or closes the browser. The two-way communication means that every time the browser/client makes a request, the server receives the request and cookies containing specific parameters and a unique Session ID which the server generates to identify the user. The Session ID doesn't change for a particular session, but the website generates it every time a new session starts.

Generally, Important Session Cookies containing these Session IDs deletes when the session ends. But this won't have any effect on the cookies which have fix expire time.

Checking Cookie Support

In this section, we will learn how to check if the user's browser support cookies. We will do that using the `request.session` which uses three methods, `set_test_cookie(), test_cookie_worked() and delete_test_cookie().`

`set_test_cookie()` method will set the random cookie on the web browser. When user access the next `view,` we will use the `test_cookie_worked().` If the browser sends the cookie we have initially set back, this method will validate it. This means cookies are working in the client's web browser and then we will delete it using `delete_test_cookie().`

Let's test this. Go to current working app and define a method in view like below:

```
from django.shortcuts import render
from django.http import HttpResponse


def setCookie(request):
    request.session.set_test_cookie()
    return HttpResponse("<h2>Testing cookie option is enabled on
browser or not</h2>")


def validateCookie(request):
    if request.session.test_cookie_worked():
        print("Cookies are enabled")  # This will be print on
console
        request.session.delete_test_cookie()

    return HttpResponse("<h2>Cookies Validation Completed</h2>")
```

Before running your server, make sure you have configured the urls correctly. Also, configure your db and run makemigration and migrate command

Page hit Count using Cookies

In this section, we will learn how to set your own cookie and how to read cookies which are coming from browser.

For that, we are going to use `response.set_cookie()` method to set a cookie and `request.COOKIES` which will return the dictionary of all the cookies that come from browser.

Now, we will create a page/view counter that will count number of times a particular view is accessed. For that, we will define a new view like below:

```
def countView(request):
    if 'count' in request.COOKIES:
        count = int(request.COOKIES['count']) + 1
    else:
        count = 1
    response = render(request, 'cookiesApp/count.html', {'count':
count})
    response.set_cookie('count', count)
    return response
```

Now, let's create template 'count.html'

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Page Count</title>
</head>
<body>
```

```
<b>Page Count: {{count}}</b>
</body>
</html>
```

Shopping Cart - Create Form and Views

In this section, we will create a form where user enters any product name and quantity, we will be saving that in a cookie and display the same in a cookie. For that, we will create a `index.html` which shows the 2 links, one for adding items and another for displaying it in a cart (basically we are looking into the cookie) and view for displaying this template.

Afterwards, we will create the add item view and template and similarly for displaying item from the cookie. Code is written below:

Note - Please configure the URLs by yourself.

forms.py

```
from django import forms


class ItemForm(forms.Form):
    name = forms.CharField(max_length=50)
    quantity = forms.IntegerField(max_value=1000)
```

views.py

```
def index(request):
    return render(request, 'cookiesApp/index.html')


def addItem(request):
    form = ItemForm()
    response = render(request, 'cookiesApp/addItem.html', {'form':
form})
    if request.method == 'POST':
        form = ItemForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['name']
            quantity = form.cleaned_data['quantity']
            response.set_cookie(name, quantity, 120)
# In cookie, add item is getting saved and setting also the cookie
timeout to 120 sec
    return response


def displayCart(request):
    return render(request, 'cookiesApp/displayItems.html')
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Shopping Cart</title>
</head>
<body>
    <h1>Shopping Cart</h1>
    <a href="/cookiesapp/addItem">Add Item</a><br/>
    <a href="/cookiesapp/displayItem">Display Cart</a>
</body>
</html>
```

addItem.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Adding Item</title>
</head>
<body>
    <form method="post">
        <table>
            {{form.as_table}}
            {% csrf_token %}
        </table>
        <br/>
        <input type="submit" value="Add">
        <a href="/cookiesapp/displayItem">Display Cart</a>
    </form>
</body>
</html>
```

displayItem.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Items Available</title>
</head>
<body>
    <h1>Items in the Cart</h1>
    {% if request.COOKIES %}
    <table>
        <thead>
            <th>Product Name</th>
            <th>Quantity</th>
        </thead>
        {% for key, value in request.COOKIES.items %}
        <tr>
            <td>{{key}}</td>
            <td>{{value}}</td>
        </tr>
        {% endfor %}
    </table>
    {% else %}
    <b>Cart is empty!!</b>
    {% endif %}
</body>
</html>
```

Note - When you click on the display cart link, it will show you all the other cookie information as well. We will see later how to extract our desired data from cookie when implement this through session API

Page Count Using Session API

In this section, we will implement the page count app with session API which we had earlier implemented using cookies.

Let’s define a view for it.

```
def pageCount(request):
    count = request.session.get('count', 0) # We are initializing
count with 0 inside get method
    count += 1
    request.session['count'] = count
    return render(request, 'sessionAPI/count.html', {'count':
count})
```

count.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Page Count</title>
</head>
<body>
<b>Page Count: {{count}}</b>
</body>
</html>
```

Implementing Shopping Cart using Session API

form is same as we have defined duing cookie example.

views.py

```
def index(request):
    return render(request, 'sessionAPI/index.html')


def addItem(request):
    form = ItemForm()

    if request.method == 'POST':

        name = request.POST['name']
        quantity = request.POST['quantity']
        # sending data into session

        # key is the name and value of name is quantity
        request.session[name] = quantity

    return render(request, 'sessionAPI/addItem.html', {'form':
form})


def displayCart(request):
    return render(request, 'sessionAPI/displayItem.html')
```

Index.html and addItem.html is same.

displayItem.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Items Available</title>
</head>
<body>
    <h1>Items in the Cart</h1>
    {% if request.session %}
    <table>
        <thead>
            <th>Product Name</th>
            <th>Quantity</th>
        </thead>
        {% for key, value in request.session.items %}
        <tr>
            <td>{{key}}</td>
```

```
                <td>{{value}}</td>
            </tr>
            {% endfor %}
        </table>
        {% else %}
        <b>Cart is empty!!</b>
        {% endif %}
    </body>
    </html>
```

Other methods on Session

In this section, we will learn some other methods used in session API.

get_expiry_age() - This returns the session expiry time. By default, expiry age is set to 14 days.

set_expiry_age() - This is to set the session expiry time. You use this function at the start of view for which you want to create a session.

If you want to delete any session, you can do it using del() function. Remember, session is nothing but a dictionary.

Deleting out page count session which we have defined during page count example as it is also coming with shopping cart session. So, for that we will add below line in index() view:

```
del request.session['count']
```

Middleware

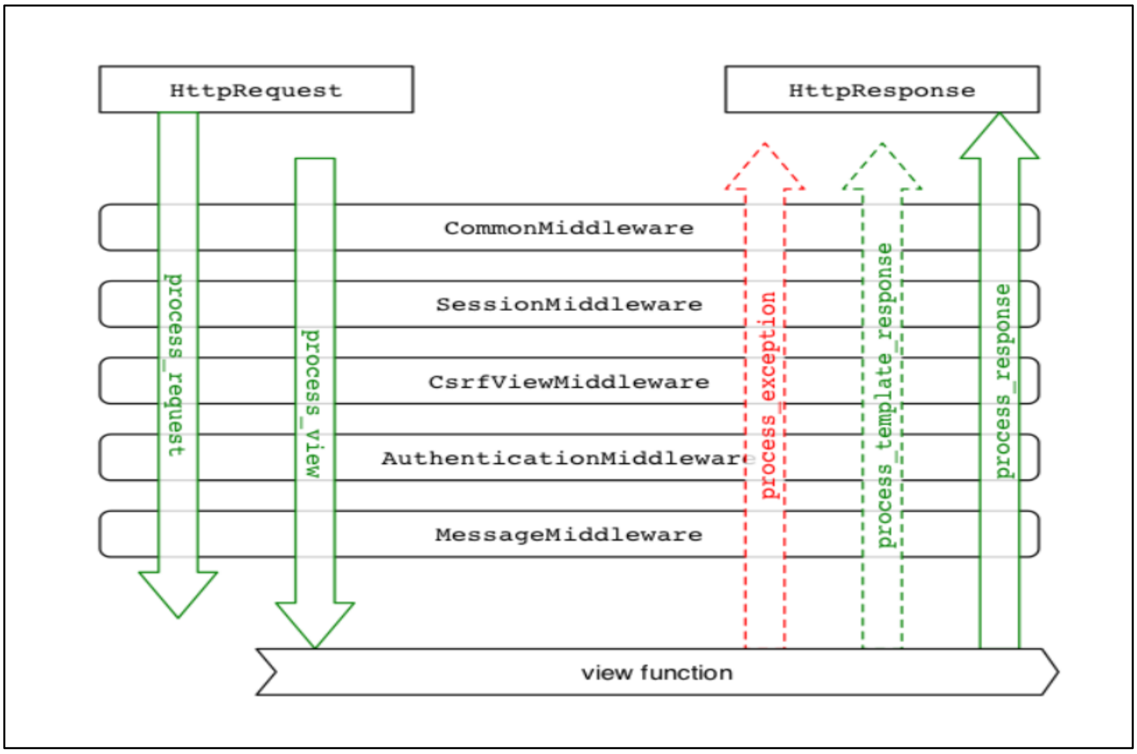
Introduction

Middleware is a framework of hooks into Django's request/response processing. It's a light, low-level "plugin" system for globally altering Django's input or output.

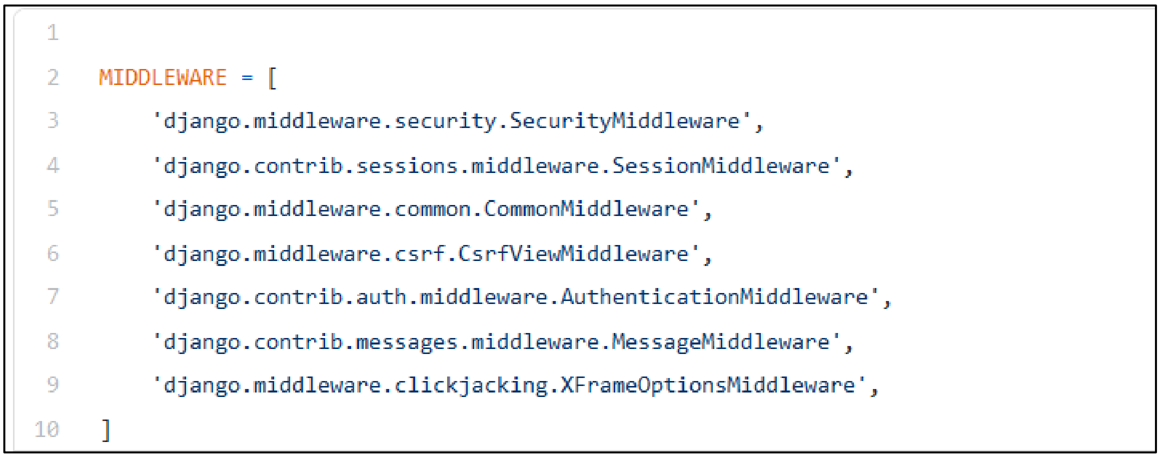
Middleware is a lightweight plugin that processes during request and response execution. Middleware is used to perform a function in the application. The functions can be a security, session, csrf protection, authentication etc.

Each middleware component is responsible for doing some specific function. For example, Django includes a middleware component, `AuthenticationMiddleware`, that associate users with requests using sessions.

To understand how a **Django Middleware** works we need to remember that the basic architecture of Django is composed of a **request** and a **response**. A middleware is something that stays in the middle, Let's give a look to the next diagram, taken from official Django documentation:



The **order** where you place your middleware in **settings.py** is important: middlewares are processed from top to bottom during a request and from bottom to top during a response



```

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

How it works?

In a nutshell, a Middleware is a regular Python class that hooks into Django's request/response life cycle. Those classes holds pieces of code that are processed upon *every* request/response your Django application handles.

The Middleware classes doesn't have to subclass anything, and it can live anywhere in your Python path. The only thing Django cares about is the path you register in the project settings `MIDDLEWARE_CLASSES`.

Your Middleware class should define at least one of the following methods:

Called during request:

- process_request(request)
- process_view(request, view_func, view_args, view_kwargs)

Called during response:

- process_exception(request, exception) (only if the view raised an exception)
- process_template_response(request, response) (only for template responses)
- process_response(request, response)

The Middleware classes are called twice during the request/response life cycle. For that reason, the order you define the Middlewares in the `MIDDLEWARE_CLASSES` configuration is important.

During the **request** cycle, the Middleware classes are executed top-down, meaning it will first execute `SecurityMiddleware`, then `SessionMiddleware` all the way until `XFrameOptionsMiddleware`. For each of the Middlewares it will execute the **process_request()** and **process_view()** methods.

At this point, Django will do all the work on your view function. After the work is done (e.g. querying the database, paginating results, processing information, etc), it will return a **response** for the client.

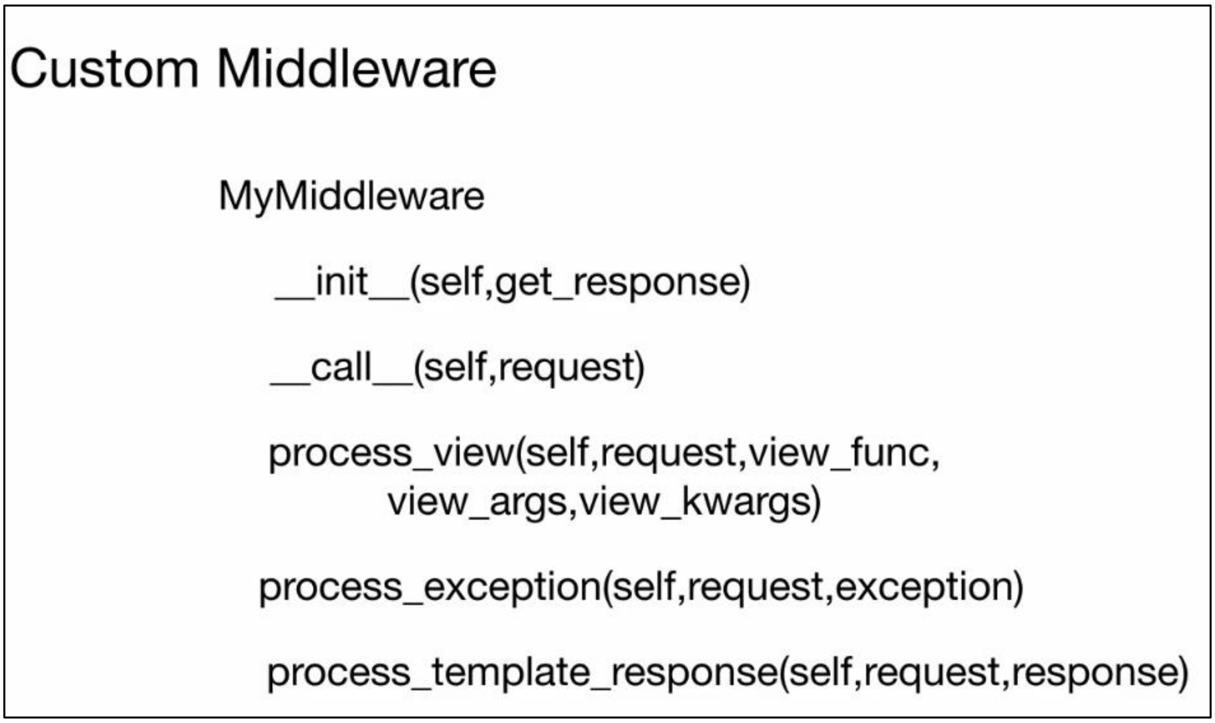
During the **response** cycle, the Middleware classes are executed bottom-up, meaning it will first execute `XFrameOptionsMiddleware`, then `MessageMiddleware` all the way until `SecurityMiddleware`. For each of the Middlewares it will execute the **process_exception()**, **process_template_response()** and **process_response()** methods.

Finally, Django will deliver the response for the client. It is important to note that **process_exception()** is only executed if a exception occurs inside the view function and **process_template_response()** is only executed if there is a template in the response.

Custom Middleware

To create a custom middleware, we will create a class, let says, MyMiddleware within which we will be creating or implementing these methods as shown in below image. Here, __init__() and __call__() are mandatory to create and last three are optional and required as per the requirement.

In __init__() method we get the access to response, that is, `get_response` will point to the next middleware or the view where the request should go. So, if you see the request and response flow above, every middleware's init method will have the information to the next middleware where the request will go in the `get_response` variable.



Custom Middleware

MyMiddleware

```
__init__(self,get_response)

__call__(self,request)

process_view(self,request,view_func,
        view_args,view_kwargs)

process_exception(self,request,exception)

process_template_response(self,request,response)
```

In __call__() method, we can play around the request/response received.

Create Custom Middleware

We will be creating custom middleware in session API app. First, we will create **the middleware.py** python file. In this file, create a class with name MiddleWareLifeCycle (say) and implement methods we have discussed above. As we know, init method is only called once when an object is created for a class. Call method will be invoked for every incoming request and will be called before it goes to actual view or the next middleware in the line.

```
class MiddleWareLifeCycle:

def __init__(self, get_response):  #name should be get_response only
        self.get_response = get_response

def __call__(self, request):
        print("Before the view is executed")
        response = self.get_response(request)
        print("After the view is executed")
        return response
```

Now, you see `self.get_response(request)` in call method which is passing the request to the next middleware defined under MIDDLEWARE list in settings.py. If there is no middleware after this middleware, then request is passing to the view.

Afterwards, add your middleware in the MIDDLEWARE list in settings.py.

Exception Handling Middleware

In this section, we will learn 2 things:

1 - How to handle exceptions by using Middleware? So, no matter which exception our application throws, instead of showing the entire exception to user, we can show the user friendly message. We can do that using method process_exception().

2 - We will also learn how to configure multiple custom middleware inside the MIDDLEWARE list.

```
class ExceptionHandlingMiddleware:

        def __init__(self, get_response):
                self.get_response = get_response

        def __call__(self, request):
                return self.get_response(request)

        def process_exception(self, request, exception):
                return HttpResponse('<b>We are currently facing technical Issue.
It will be fixed soon</b>')
```

Security

Introduction

By default, Django prevents most common security mistakes:

- Cross site request forgery (CSRF) protection
- Cross site scripting (XSS)
- SQL injection protection
- Clickjacking protection
- Enforcing SSL/HTTPS
- Host header validation

We will use 2 more key security features which are provided by Django:

1 - Authentication (Identification of User)
2 - Authorization (Permissions Identification)

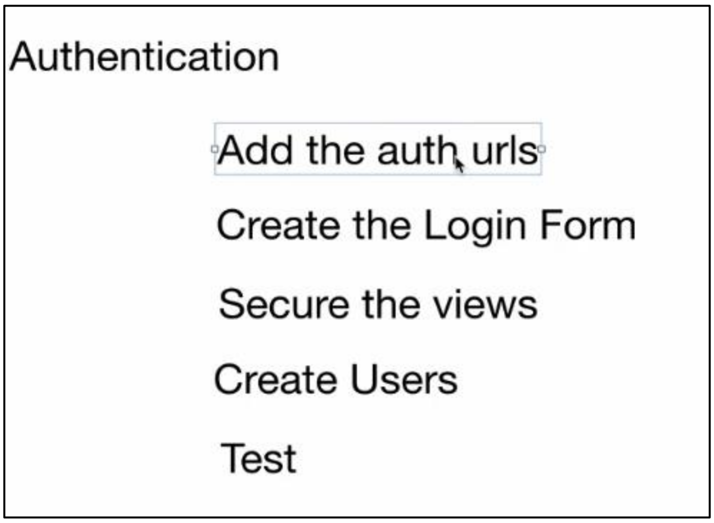
All the user information is stored in your db under the table with name ‘auth_user’ which is mapped to `User` model object in django security. So, in our application, we can directly use this model object and access the user information. So, we need not connect to database and do all the authentication work as Django does it for us.

Once the user is authenticated, he/she can have permission to features depending on group he/she is part of, or user can have individual level permission to any features.

Explanation with an Example

To understand and implement authentication and authorization, we will be using our example while learning function-based views.

This is what we are going to do:



Authentication

Add the auth urls

Create the Login Form

Secure the views

Create Users

Test

Adding the Auth URLs and Creating the Login Form

To do that, go to your urls.py and add this path like below:

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

`django.contrib.auth.urls` module already has lot of URLs which are mapped to certain views like login, logout etc. We have to configure this URL with `account/`. So, account/login will take us to login page and similarly, account/logout will redirect us to logout page and so on.

Next step is to create the form that is mapped to account/login. For that, we will be creating a `registration` directory under `templates` directory and inside `registration` directory we will be creating `login.html`.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Sign In</title>
</head>
<body>
    <form method="post">
        {% csrf_token %}
        {{form.as_p}}
        <button type="submit" name="button">Login</button>
    </form>
</body>
</html>
```

`form` object is available by default for login.html. It is a model form based on internal model called `User`. So, internally, `django_auth` has a model called User and this form is instance of that model.

Secure Views

To securing our views (means only logged in user can use it), we will import `login_required(decorator)` from `django.contrib.auth.decorator.`

We will decorator all our views with this decorator.

```
from django.shortcuts import render, redirect
from .models import Student
from .forms import StudentForm
from django.contrib.auth.decorators import login_required



@login_required
def getStudent(request):
    students = Student.objects.all()
    return render(request, 'fbvCRUDApp/index.html', {'students': students})



@login_required
def createStudent(request):
    form = StudentForm()
    if request.method == 'POST':
        form = StudentForm(request.POST)
        if form.is_valid():
            form.save()
        return redirect('/fbvCRUDapp')
    return render(request, 'fbvCRUDApp/create.html', {'form': form})



@login_required
def deleteStudent(request, id):
    student = Student.objects.get(id=id)
    student.delete()
    return redirect('/fbvCRUDapp/getStudent')
```

```
@login_required
def updateStudent(request, id):
    student = Student.objects.get(id=id)
    form = StudentForm(instance=student)
    if request.method == 'POST':
        form = StudentForm(request.POST, instance=student)
        if form.is_valid():
            form.save()
            return redirect('/fbvCRUDapp/getStudent')
    return render(request, 'fbvCRUDApp/update.html', {'form': form})
```

Create Users

In this section, we will create users using with which we can login to our application. There are multiple ways to do it. But, first, we will first create admin user from command line and then with admin user, we will create app level user.

In earlier section, where we learnt about how to create superuser, so that was nothing but to create admin user. Now, login to localhost:8000/admin with your admin ID and password and then you can create user. (Make this user a 'Staff User')

With this user you can login to your application.

Logout Redirect

Let's say you are logged in your app using your credentials and you have not implemented the logout feature in it. However, if you go to the localhost:8000/account/logout, it will logout you and redirect to the default django administration page of logout. (As discussed earlier, `django.contrib.auth.urls` module already has lot of URLs which are mapped to certain views like login, logout etc.). However, you don't want that, you want to redirect to your desired location, so for that do the following steps:

1 - Go to the settings.py and add the variable LOGOUT_REDIRECT_URL(defined in the same way in django) and assign the 'URL' where you want to redirect like below:

```
LOGOUT_REDIRECT_URL = '/'
```

Here it will take it your root URL.

2 - However, if you want to redirect to some logout template you created you can assign that URL here. Let's define one.

For that, we will create our logout view and then the template.

settings.py

```
LOGOUT_REDIRECT_URL = '/'
```

views.py (logout view)

```
def logout(request):
   return render(request, 'fbvCRUDApp/logout.html')
```

logout.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Logout</title>
</head>
<body>
<h2>You've logged out successfully!!</h2>
</body>
</html>
```

Add login and logout links

As we have implemented login and logout, let's add the link for it in our templates. For that, we don't need to much more rather just create in templates.

Implement Authorization

In this section, we will implement authorization.

So, in our app, let's restrict the delete operation only to some users. So, for that, login to the django administration link as admin user. Click on the user for which you want to restrict the delete operation and scroll down once the page is loaded. You will find available User permissions which shows all type of permissions. Select now those permissions which you want your user to have and then add it to the User Permission queue and then save it.

Afterwards, we need to modify our delete view as well in such a way that it checks first that current user has the permission to delete operation or not like below:

```
from django.contrib.auth.decorators import login_required,
permission_required


@login_required
@permission_required('fbvCRUDApp.delete_student')  # You need to
define app name and operation with your model class in lowercase
def deleteStudent(request, id):
    student = Student.objects.get(id=id)
    student.delete()
    return redirect('/fbvCRUDapp/getStudent')
```

So, reload your server and login to your app with the user for which you have modified the permissions. Now, as soon as you click on delete link, it will logout and redirect you to the login page which actually means that in order to delete entry, user needs to login with admin level ID or the UID which has the permission to delete operation.

User Groups

Let's say if there are 100s of user and to all, you want to assign a similar permission, so instead of assigning one by one, you can create the group, let's say in our example, we want all teachers to give permission to add and see the list of students and not the delete operation permission. After creating the group, you can assign those permission to group which will then apply to all members of that group.

Creating group similar to creating user. In Django administration page, you will find add button for group. By clicking on it, you can enter your group details and play with the permissions accordingly.

Afterwards, you can add your desired users into your groups.

ORM Relationship

Django ORM provides a level of abstraction which makes it easy to work with objects. ORM will automatically relate the object's attributes to corresponding table fields. We are going to make some models and check their field relationships in other tables.

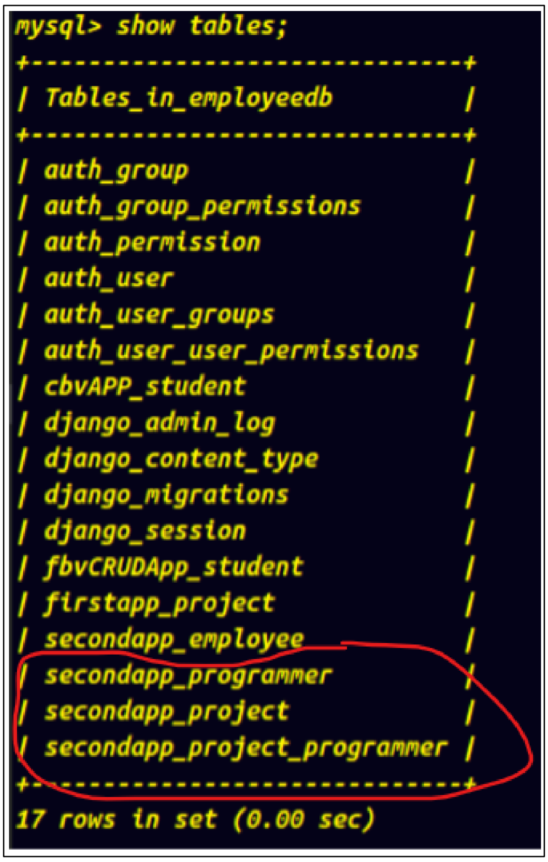
Now, you can create models by pasting the code in models.py of your application. Let's get started.

ManytoMany Relationship

Here, we will be using our employee example, where we learnt to create Models. So, go to the models.py and create 2 model classes to understand many to many relationships. One class is Programmer and another class is Project. So basically, we are implementing many to many relationships between a programmer and project as a programmer can work on multiple projects and a project can have multiple programmers.

```
class Programmer(models.Model):
    name = models.CharField(max_length=30)
    sal = models.IntegerField()

class Project(models.Model):
    name = models.CharField(max_length=40)
    programmer = models.ManyToManyField(Programmer)
```

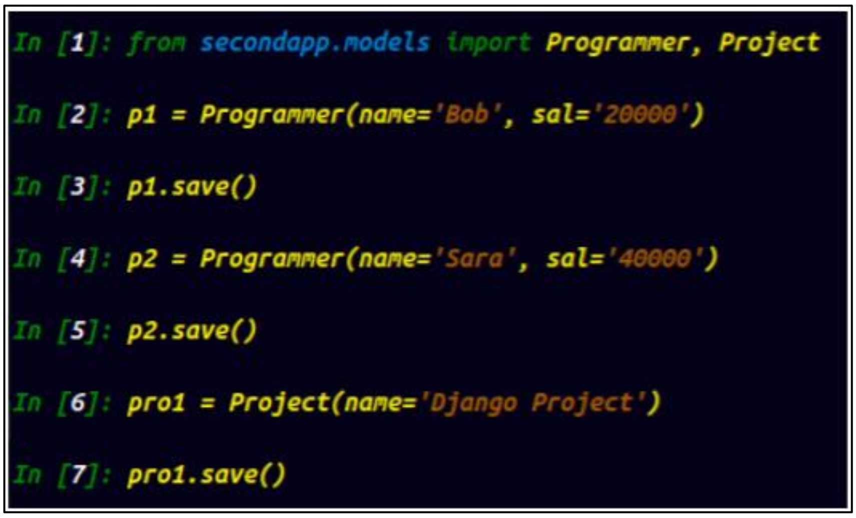
After creating model class, make sure to run makemigration and migrate command to create the table in db. Below will be the tables created in my db:



```
mysql> show tables;
+-------------------------------+
| Tables_in_employeedb          |
+-------------------------------+
| auth_group                    |
| auth_group_permissions        |
| auth_permission               |
| auth_user                     |
| auth_user_groups              |
| auth_user_user_permissions    |
| cbvAPP_student                |
| django_admin_log              |
| django_content_type           |
| django_migrations             |
| django_session                |
| fbvCRUDApp_student            |
| firstapp_project              |
| secondapp_employee            |
| secondapp_programmer          |
| secondapp_project             |
| secondapp_project_programmer  |
+-------------------------------+
17 rows in set (0.00 sec)
```

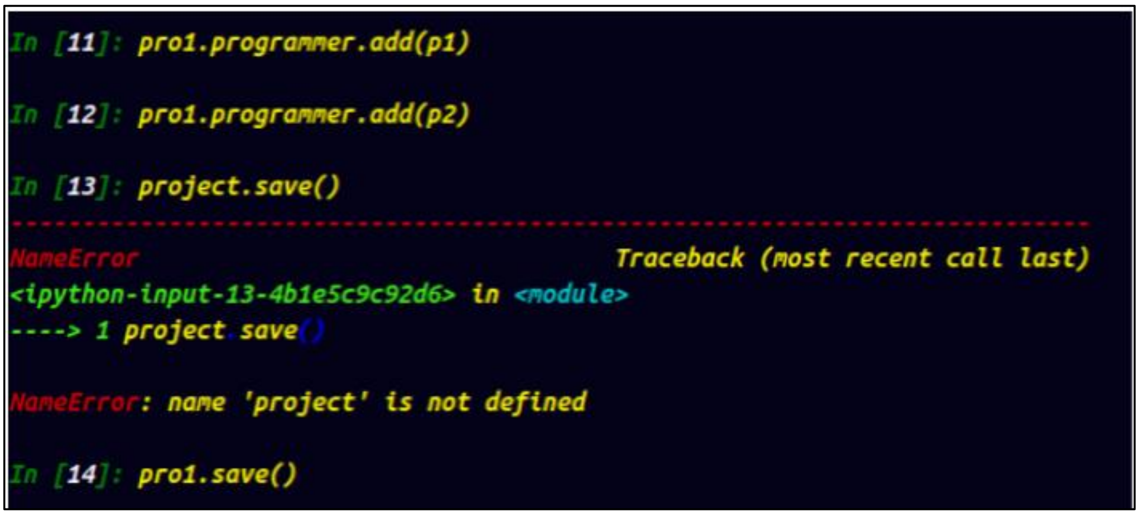
Once all this done, we will check how to use Django ORM API against these 2 model objects. To do that, go to command line and execute this command `python3 manage.py shell` to open the Django shell.

Once your shell will open, import your model classes programmer and project and then create bunch of programmer and project and see the implementation. Below code lines will demonstrate the implementation of manytomany relationship:



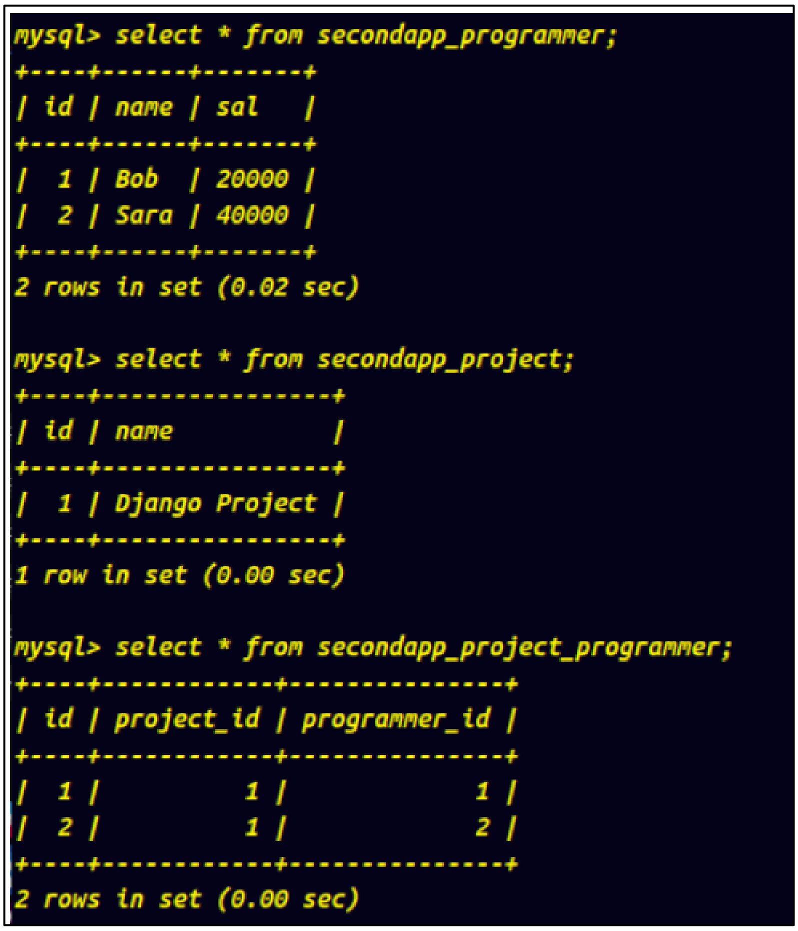
```
In [1]: from secondapp.models import Programmer, Project

In [2]: p1 = Programmer(name='Bob', sal='20000')

In [3]: p1.save()

In [4]: p2 = Programmer(name='Sara', sal='40000')

In [5]: p2.save()

In [6]: pro1 = Project(name='Django Project')

In [7]: pro1.save()
```

Now, we will assign our programmers to this project we have created. Remember, we have created the `programmer` variable in `Project` class, using this only we will now create the manytomany relationship.



```
In [11]: pro1.programmer.add(p1)

In [12]: pro1.programmer.add(p2)

In [13]: project.save()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-4b1e5c9c92d6> in <module>
----> 1 project.save()

NameError: name 'project' is not defined

In [14]: pro1.save()
```

Now, **if you check the table in db**, you will see relationship will between project and programmer will exist in our `secondapp_project_programmer`.

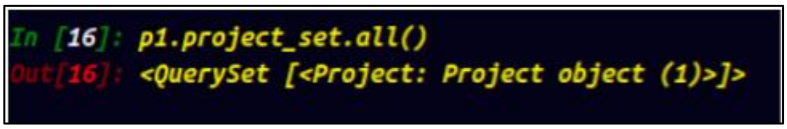


```
mysql> select * from secondapp_programmer;
+----+------+-------+
| id | name | sal   |
+----+------+-------+
|  1 | Bob  | 20000 |
|  2 | Sara | 40000 |
+----+------+-------+
2 rows in set (0.02 sec)

mysql> select * from secondapp_project;
+----+----------------+
| id | name           |
+----+----------------+
|  1 | Django Project |
+----+----------------+
1 row in set (0.00 sec)

mysql> select * from secondapp_project_programmer;
+----+------------+---------------+
| id | project_id | programmer_id |
+----+------------+---------------+
|  1 |          1 |             1 |
|  2 |          1 |             2 |
+----+------------+---------------+
2 rows in set (0.00 sec)
```

Another different thing is that we haven't define relationship in `Programmer` class. So, for a `Programmer` object, if you want to know all the projects he is assigned to, we need not to define a relationship for that in `Programmer` class, instead Django gives us a set automatically. We can see it using this attribute which has a defined in this way,

`<object_you_want_see_all_other_manytomany_relationship>.<Model_Class_against_which_you_are_looking >_set.all()`

So, in our example, like this:



```
In [16]: p1.project_set.all()
Out[16]: <QuerySet [<Project: Project object (1)>]>
```

ManytoOne Relationship

In this section, we will implement many to one relationship. For that, we will take the example of Customer and Phone number where one customer has multiple phone number.

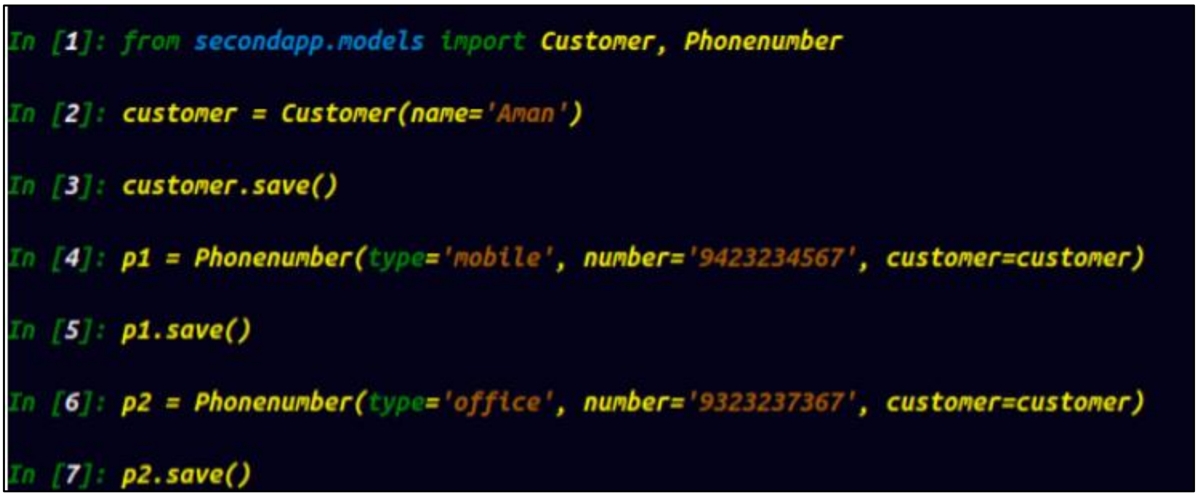
Let's create the model class Customer and PhoneNumber like below:

```
class Customer(models.Model):
    name = models.CharField(max_length=40)


class Phonenumber(models.Model):
    type = models.CharField(max_length=10)
    number = models.CharField(max_length=15)
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
```

on_delete=models.CASCADE means that if any Customer object is getting deleted, then all of his/her contact details will get deleted also.

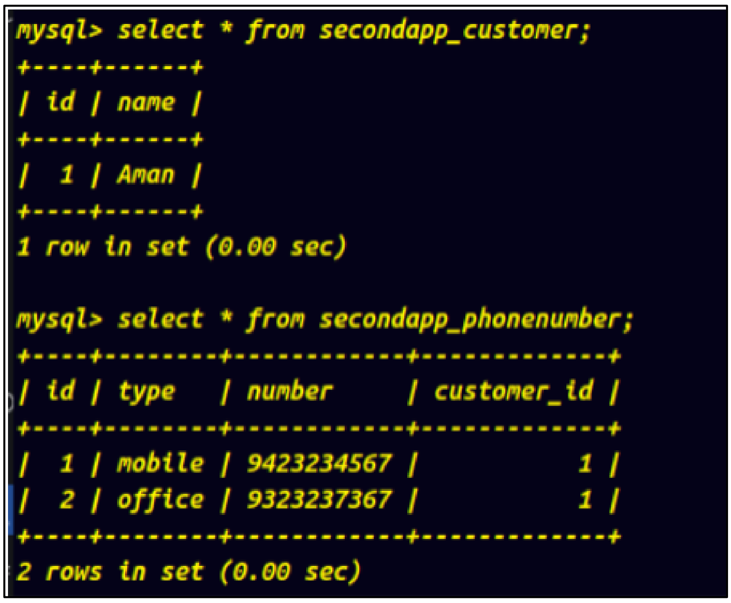
Now, do the migrations.

Once migration is done, open the Django shell and let's add multiple phone number for one customer. So, first, we import our model class Customer and Phonenumber and then create instance for both. In below screenshot, one Customer Instance is created and 2 of his/her Phonenumber instances are created. So, we are implementing onetomany relationship.



```
In [1]: from secondapp.models import Customer, Phonenumber

In [2]: customer = Customer(name='Aman')

In [3]: customer.save()

In [4]: p1 = Phonenumber(type='mobile', number='9423234567', customer=customer)

In [5]: p1.save()

In [6]: p2 = Phonenumber(type='office', number='9323237367', customer=customer)

In [7]: p2.save()
```

Let's see the in the dB also:



```
mysql> select * from secondapp_customer;
+----+------+
| id | name |
+----+------+
|  1 | Aman |
+----+------+
1 row in set (0.00 sec)

mysql> select * from secondapp_phonenumber;
+----+--------+------------+-------------+
| id | type   | number     | customer_id |
+----+--------+------------+-------------+
|  1 | mobile | 9423234567 |           1 |
|  2 | office | 9323237367 |           1 |
+----+--------+------------+-------------+
2 rows in set (0.00 sec)
```

Again, since no relationship is defined in class Customer, you can still checkout all of its phone number, using below attribute used as per the syntax (we have discussed in manytomany relationship section):

```
customer.phonenumber_set.all()
```

You can checkout delete operation. If you delete the customer, then all of its phone number will also gets deleted.

```
customer.delete()
```

<u>OnetoOne Relationship</u>

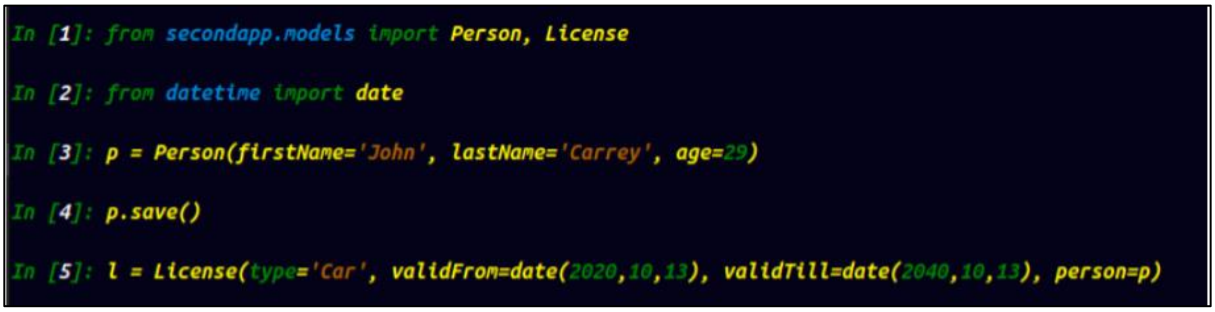
In this section, we will learn about one to one relationship. In this section, we will define Person and License model class.

```
class Person(models.Model):
    firstName = models.CharField(max_length=30)
    lastName = models.CharField(max_length=30)
    age = models.IntegerField()


class License(models.Model):
    type = models.CharField(max_length=30)
    validFrom = models.DateField()
    validTill = models.DateField()
    person = models.OneToOneField(Person, on_delete=models.CASCADE)
```

Perform the migration operation. So, after this operation, two table will be created in dB, one for Person and another for License.

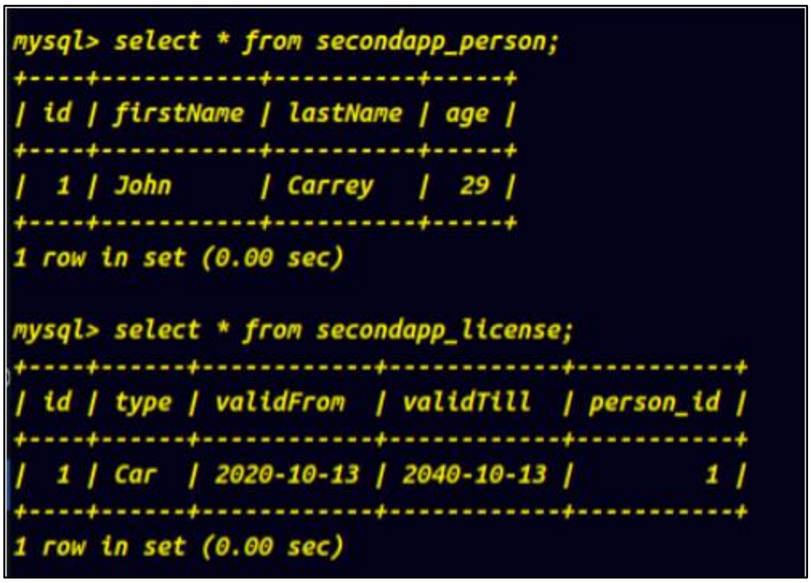
Let’s test it out:

Open the Django shell and import these classes and create a Person and a Licence object to test one to one relationship:



```
In [1]: from secondapp.models import Person, License

In [2]: from datetime import date

In [3]: p = Person(firstName='John', lastName='Carrey', age=29)

In [4]: p.save()

In [5]: l = License(type='Car', validFrom=date(2020,10,13), validTill=date(2040,10,13), person=p)
```

Let’s check out in the database:



```
mysql> select * from secondapp_person;
+----+-----------+----------+-----+
| id | firstName | lastName | age |
+----+-----------+----------+-----+
|  1 | John      | Carrey   |  29 |
+----+-----------+----------+-----+
1 row in set (0.00 sec)

mysql> select * from secondapp_license;
+----+------+------------+------------+-----------+
| id | type | validFrom  | validTill  | person_id |
+----+------+------------+------------+-----------+
|  1 | Car  | 2020-10-13 | 2040-10-13 |         1 |
+----+------+------------+------------+-----------+
1 row in set (0.00 sec)
```