# UNIT-4

# Connecting SQLite with Django

Topic: - Database Migrations

## Databases and Django:

- A database is a collection of data arranged in an efficient way. It ensures fast and secure storage, access and update of data. That is, the main concept of every type of database

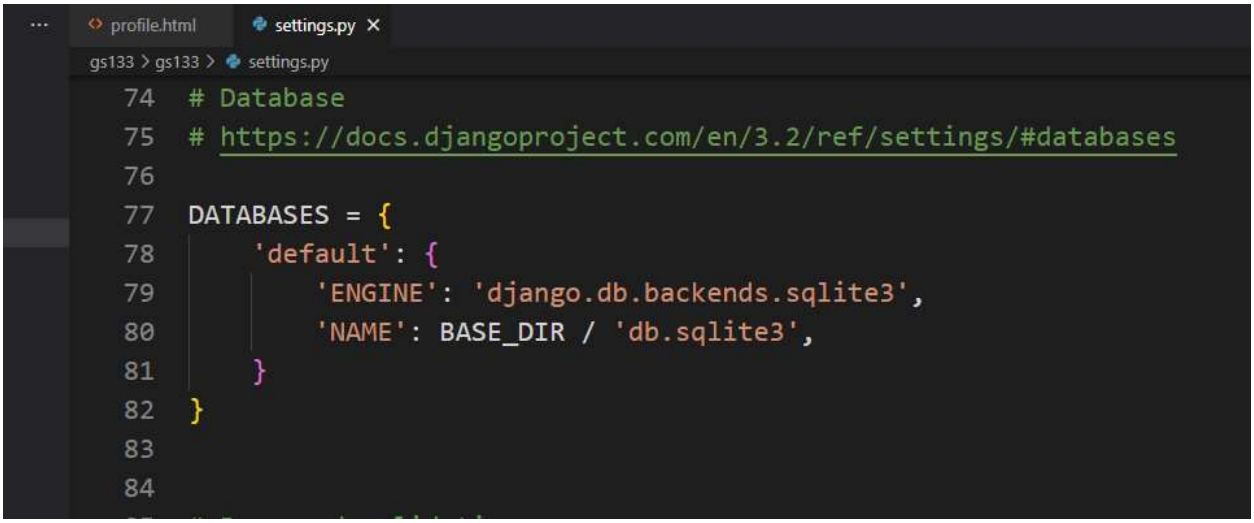
- There are many types of databases out there. We will be narrowing them down to relational databases. Now, what's that? A relational database is a collection of tables having a number of columns and rows.

- SQL (Structured Query Language) is a very popular language for databases. Almost all the popular databases use SQL. The databases like PostgreSQL, MySQL, SQLite, etc. are some of many examples.

- The databases mentioned above are relational databases. Yes, the majority of SQL databases are relational and are extensively used in the industry.

# Configuring Settings for Django Database:

These are common settings you should know to connect database with Django project. This code exists in the **settings.py** file




Basically, database settings for your project reside in a Python Dictionary. The name of the dictionary is DATABASE. This dictionary and some of its attributes are pre-defined.

The **default key** is pre-defined, which contains configuration settings for a default database. Since Django can work with multiple databases, you can have multiple database configurations.

The default key is required to configure a single or multiple databases. Inside the default value, key resides the configuration for the database.

The configuration can vary for different types of the database you are using.

Now, these keys have different values for different databases. This configuration is for SQLite database.

**ENGINE Key** is used to specify the database backend. This will connect the **Django project to SQLite Database**. You can use other backends too. For that, you have to provide additional keys. We will configure these databases in a separate article.

**Note:**

Django comes with these four database backends natively. These databases will require their database software but the backend is in-house.

**PostgreSQL:**

```
1. django.db.backends.postgresql
```

**MySQL:**

```
1. django.db.backends.mysql
```

**SQLite:**

```
1. django.db.backends.sqlite3
```

**Oracle:**

```
1. django.db.backends.oracle
```

## How these Databases Work with Django:-

- When we start our Django project, it loads the settings.py file on the server. This file contains all information and even database configurations. This file contains all information and even database configurations.

- Then Django will load the Database Backend. That backend will connect to the default database configured in settings. It is done by passing appropriate key values to the Database Software.

- The database then verifies the values and the connection is established. The connection is between the Database Backend and Database. The backend becomes a mediator between Database and Django.

- All the built-in backends are very efficient and are used in a production environment.
- Now, the database is connected and we can easily migrate our models and create tables in the same. The backend also handles all the CRUD operations from Django on the database.
- Migrations were not present in Django versions before version 1.7. Let's learn how developers dealt with the situation.

## Problems Solved by Django Migrations:-

- When migrations were introduced for the first time in Django, it became a huge hit. Django's popularity grew exponentially. This proves migrations are a very important feature.
- So, when there were no migrations in Django, developers had some difficulties.
- "They had to define the tables and database schemas according to models".
- Now, this may seem easy in listening but SQL is quite difficult. This became more complex when they had to create models with foreign key relations in them.
- This also led to occasional errors and database inconsistency. The development was much slower than it is today. The developers would also have to keep track of all the changes they made themselves which was very difficult.
- These were some of the many problems faced by developers with no migrations.
- Migrations resolved all of them and also some additional features were provided. Let's learn in-depth about these features and solutions.

# Django Migrations:-

Django migrations are nothing but Python Files. They contain Python code which is used by Django ORM to create and alter tables and fields in the database.

Django implements Models and changes to them on the database via migrations. Django generates migrations automatically. This is a key feature and you should also take care of these.

# Key Benefits of Django Migrations:-

## 1. Creating tables without SQL:-

- If we have used SQL then making tables would become more complex. For beginner developers, this could pose a real problem. Migrations had made Django more fun to use. It lets us bypass the whole SQL writing and makes tables automatically.

## 2. Models and Database Schema are always synchronized:-

- This was the most error-prone zone for backend developers. Since they had to make the Database Schema same as the Models. It would take complex SQL and thus, resulting in errors. Sometimes, they can be real security threats but this is not the case with migrations.

- Since migrations generate SQL automatically, it is always correct. That SQL contains the same constraints, we define in Models.

## 3. Easier Version Control for Database Scheme:-

- Migrations are Python files. Django generates a new migration file every time we modify a model. These files are not deleted and stored inside the app directory.

- These files contain very simple codes. It is easy for a developer to understand what changes were made. This also makes it easy to revert back to previous Models.

- Django migrations have several benefits, these were some important ones.

## We can now learn some very basic operations on migrations

These are some of the common Django migration operations which you must know. These commands will give you an edge in handling Database Management System (DBMS). Migration is a trending topic for Django interviews. You will surely find this knowledge worthwhile.

Now, in **Index/models.py** file paste this code:

```
class Student(models.Model):
   roll_no = models.TextField()
   name = models.TextField(max_length = 40)
   stud_class = models.TextField()
   department = models.TextField()
```

## 1. Creating Migrations

## Just run this command:

```
1. python manage.py makemigrations
```

Thus, we have a Migration file ready. The name of this file is 0001_initial.py. Django automatically serializes the migration files with a numerical value. This file contains the migration class.

## 2. Applying Migration:

We created migrations in the previous file. Now, it's time to apply them. We need to understand the difference. In the previous section, we just created some Python files. These files will not affect the database until we apply them.

## This command will apply migrations in the database.

```
2. python manage.py migrate
```

**"It is important that you execute makemigrations before migrate command. Since a migration file is required before applying it to the database."**

# Topic: - Fetch Data from Database

In Django you primarily interact with databases through the models. Models are generally defined in a Python file creatively called **models.py**. In these files you write Python code that defines the structure and how you interact with the databases.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

In this **Person** class model you defined the Person model, which would map to a **Person table** with two fields: **first_name** and **last_name.**

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

`NOTE- python manage.py sqlmigrate students 0001`

## So, how do I get the data from a database?

- You would make a query in your views, defined creatively in the **views.py**
- Views are what define the logic of your application, so you would interact with your database here.
- Views… will request information from the model you created before and pass it to a template.

- You first have to **import your models** into your views. And now your views can make questions (queries) to the models, which are an interface to your database.

```
from .models import Person

def person_list(request):
    persons = Person.objects.all()
    context = {“persons”: persons}
    return render(request, 'app/person_list.html', context)
```

In this case you’re querying a list of all persons in the database and storing them in a **persons** variable. You will use this variable as the value in the context dictionary. This context is passed to the **template person_list.html**, so the data will be available to render in the template.

# Topic: - How to display API data in Django Template

**The steps to display API data in Django templates (from the beginning) :-**

Step 1. Create Django Project

(django admin startproject portfolio)

Step 2. Create app in that django-project.

(Python manage.py startapp API)

OR

(django admin startapp API)

Step 3. Add your app name in installed apps.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'student.apps.StudentConfig',
    'example',
    'index',
    'students',
    'API',
```

Step 4. Install requests (Request rest framework extends the standard httprequest that provides flexible request parsing which helps user to treat requests with JSON data or other media types in the same way that you would normally deal with form data.)

(pip install requests) on command prompt

Step 5. Add path for API app in the Project's urls.py

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

urlpatterns = [
    path('admin/', admin.site.urls),
 path('API/', include('API.urls')),
```

Step 6. Create a new file in the app with the name urls.py

and write this code

urls.py

```
from django.urls import path
from API import views
urlpatterns = [
    path('',views.index,name="API"),
]
```

Step 7. Create a function **index** in views.py of app where we are going to import requests and fetch the data from the given api and pass it to the html page.

views.py

```python
from django.shortcuts import render
import requests
# Create your views here.
def index(request):
    response=requests.get('https://api.covid19api.com/countries').json()
    return render(request,'index.html',{'response':response})
```

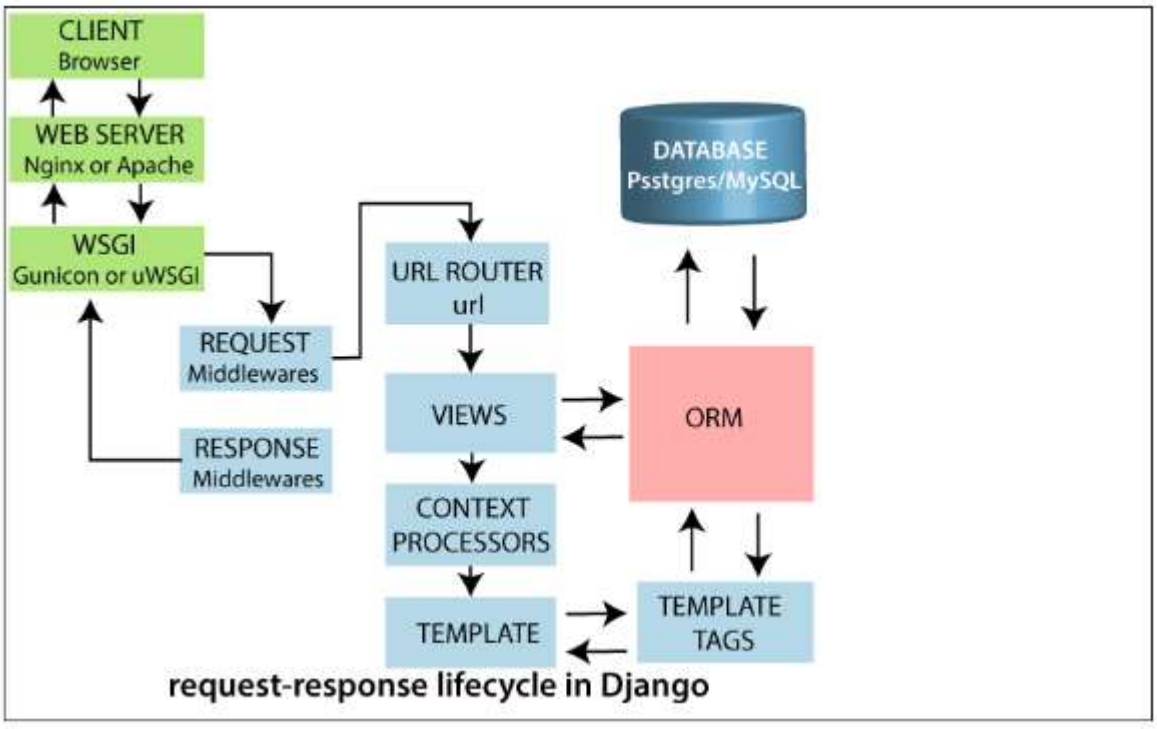
Step 8. Create a html file in the **template folder**

index.html > html

```html
<html>
<head>
    <title>API Display</title>
</head>
<body>
    <h1>List of countries</h1>
    {% for i in response %}
    <strong>{{i.Country}} ,</strong>
    {% endfor %}
</body>
</html>
```

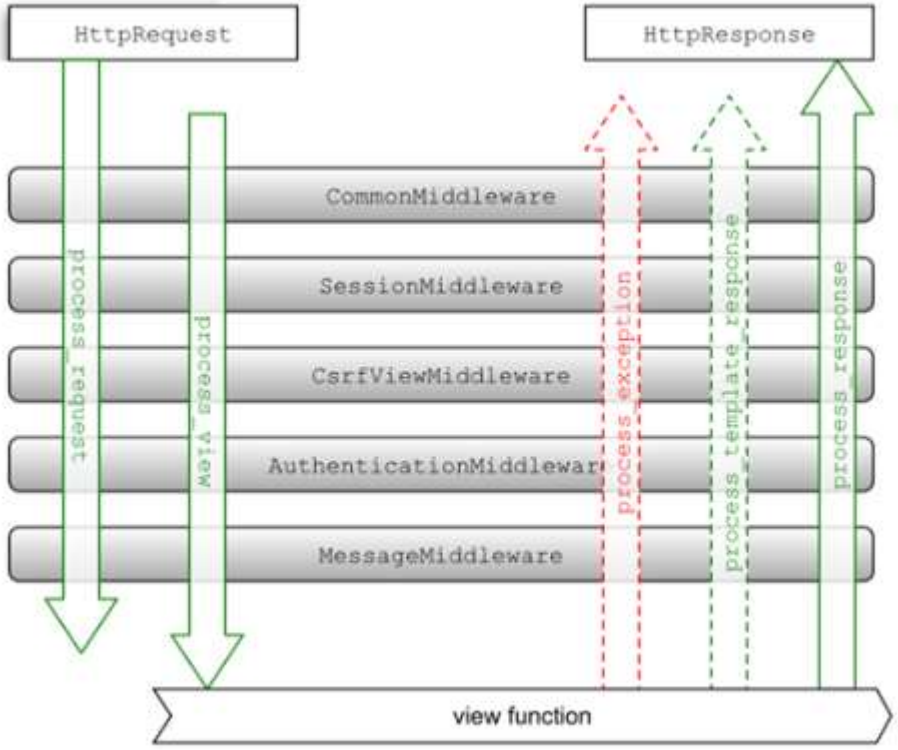
```
ON WITH DJANGO\portfolio> python manage.py runserver
```

## Request- Response Life Cycle of Django :-



# What is a Django middleware and what is used for:-

**In Layman terms** a Middleware is something which acts as a bridge between two parts of a program or the system that enables communication between them. **In technical terms** 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. Each middleware component is responsible for doing some specific function.




Working of Middleware ⬇⬇

## How does Middleware work :-

When a user makes a request from your application, a WSGI handler is instantiated, which handles the following things.

- Imports project's `settings.py` file and Django exception classes.
- Loads all the middleware classes which are written in `MIDDLEWARE` tuple located in `settings.py` file
- Builds list of methods which handle processing of request, view, response & exception.
- Loops through the request methods in order.
- Resolves the requested URL
- Loops through each of the view processing methods
- Calls the view function
- Processes exception methods (if any)
- Loops through each of the response methods in the reverse order from request middleware.
- Builds a return value and makes a call to the callback function.

## What are the types of Middleware:-

There are two types of Middleware in Django:

- Built-in Middleware
- Custom Middleware

**Built-in Middleware** are provided by default in Django when you create your project. You can check the default Middleware in `settings.py` file of your project.

```
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',
]
```

**Custom Middleware** — You can write your own middleware which can be used throughout your project. Custom middleware in Django is created either as **a function** style that takes a get_response callable or a **class-based** style whose call method is used to process requests and responses.

It is created inside a file **middleware.py**. A middleware is activated by adding it to the MIDDLEWARE list in **Django settings.**

A Custom middleware can be written in two ways

1. **As a function based**
2. **As a class based**

## Building the Django middleware:-

1. The first step is to create a file midleware.py inside your app.

### Function based middleware:-

```
def simple_middleware(get_response):
    # One-time configuration and initialization.

    def middleware(request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.
        print("before response")
        response = get_response(request)
        print("After response")
        # Code to be executed for each request/response af
        # the view is called.

        return response

    return middleware
```

## Class based middleware:-

```python
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        print("before response")
        response = self.get_response(request)
        print("After response")
        # Code to be executed for each request/response af
        # the view is called.

        return response
```

## Activating Custom Middleware:-

Activating a middleware is very simple, you have to add it inside the `MIDDLEWARE` list in `settings.py`. In the middleware list, each middleware component is represented by a string that is the full python path of the middleware. You can add your SimpleMiddleware at the last of the middleware list.

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

    'custom_middleware.middleware.SimpleMiddleware', # Cla
    'custom_middleware.middleware.simple_middleware', # Fu
]
```

> *"Order of the middleware is very important because a middleware can depend on the other middleware. For example Authentication Middleware depends on Session Middleware to store the authenticated user in session (`request.session['user']`) therefore, it must be placed after session middleware."*

After adding your middleware, run your server and visit the URL [http://localhost:8080/](http://localhost:8080/)

```
python manage.py runserver 0:8080
```

Now check your terminal for the `print` statements.

```
Django version 3.2, using settings 'conf.settings'
Starting development server at http://0:8080/
Quit the server with CONTROL-C.
before response
After response
```

If you can see the print statements `before response` and `After response`, it means you have successfully created your own custom Django middleware.

Only the first two methods, `__init__` and `__call__`, are required by the Django framework for your middleware to work properly.

## init() in middleware:-

The first method, **init**, is the constructor for our Python class. It is called only once, at the time the server starts up. It takes `get_response` as a parameter, which is passed by the Django itself when we add it inside the **Middleware** list like this: 'custom_middleware.middleware.SimpleMiddleware'

## call() in middleware:-

`__call__()` method is called **once per request**

## Django Middleware Structure:-

```
class SimpleMiddlewareStructure:

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

    def _call_(self, request):

        # Code that is executed in each request before the

        response = self.get_response(request)

        # Code that is executed in each request after the v
        return response

    def process_view(self, request, view_func, view_args, v
        # This code is executed just before the view is cal
        pass

    def process_exception(self, request, exception):
        # This code is executed if an exception is raised
        pass

    def process_template_response(self, request, response)
        # This code is executed if the response contains a
        return response
```

# Django Caching:-

- Every user wants their requests on the website to be responded ASAP. There are certain methods which will make your website faster. Faster the website, greater the users it will attract.

- Caching is one of those methods which a website implements to become faster. It is cost efficient and saves CPU processing time. You must be having some questions in your mind – Is caching a method to make websites faster? The answer is yes!

- Then, what kind of websites will actually benefit from caching? The answer is dynamic websites.

- Dynamic websites generate highly personalized webpages. They come with tons of features. Some examples of websites are Spotify, Google, Facebook, Instagram, etc.

- **Caching is the process of storing recently generated results in memory. Those results can then be used in future when requested again.**

Caching basically means to save the output of an expensive calculation in order to avoid performing the same calculation again. Django provides a robust cache system which in turn helps you save dynamic web pages so that they don't have to be evaluated over and over again for each request. **Some of the caching strategies** of Django are listed down in the following table.

| Strategy | Description |
| --- | --- |
| Memcached | Memory-based cache server which is the fastest and most efficient |
| Filesystem caching | Cache values are stored as separate files in a serialized order |
| Local-memory caching | This is actually the default cache in case you have not specified any other. This type of cache is per-process and thread-safe as well |
| Database caching | Cache data will be stored in the database and works very well if you have a fast and well-indexed database server |

# Django Sessions – How to Create, Use and Delete Sessions:-

## 1. What are Cookies:-

- When we see the word cookies, we expect a chocolate chip baked biscuit, but here we will talk about computer cookies. They do a great piece of work that makes it easier for you to surf the Internet but they can be irritating if you do not know how to remove or delete cookies.

- Cookies, technically called HTTP Cookies are small text files which are created and maintained by your browser on the particular request of Web-Server. They are stored locally by your 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.

## Creating Cookies in Django:-

Django has methods like set_cookie() which we can use to create cookies very easily.

The set_cookie () has these attributes:

Name: It specifies the name of cookie.

Value: It specifies the text or variable you want to store in the cookie.

Max_age: It is the time period of cookie in seconds. After the time period completes, it will expire. It is an optional parameter; if not present then the cookie will exist till the time browser close.

```
def setcookie(request):
    html = HttpResponse("<h1>Dataflair Django Tutorial</h1>")
    html.set_cookie('dataflair', 'Hello this is your Cookies', max_age = None)
    return html
```

## 2. What are Sessions:-

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

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

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.

# Django Sessions:-

Django considers the importance of sessions over the website and therefore provides you with middleware and inbuilt app which will help you generate these session IDs without much hassle.

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

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',
]
```

The middleware. SessionMiddleware is responsible for generating your unique Session IDs. You will also require django.contrib.sessions application, if you want to store your sessions on the database.

## Creating & Accessing Django Sessions:-

Django allows you to easily create session variables and manipulate them accordingly .The request object in Django has a session attribute, which creates, access and edits the session variables. This attribute acts like a dictionary, i.e., you can define the session names as keys and their value as values.

Step 1. We will start by editing our views.py file. Add this section of code.

```
def create_session(request):
    request.session['name'] = 'username'
    request.session['password'] = 'password123'
    return HttpResponse("<h1>dataflair<br> the session is set</h1>")


def access_session(request):
    response = "<h1>Welcome to Sessions of dataflair</h1><br>"
    if request.session.get('name'):
        response += "Name : {0} <br>".format(request.session.get('name'))
    if request.session.get('password'):
        response += "Password : {0} <br>".format(request.session.get('password'))
        return HttpResponse(response)
    else:
        return redirect('create/')
```

Step 2. Add the urls in urlpatterns list.

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('create/', create_session),
    path('access', access_session),
```

**Step 3.** Now run this code -

## Deleting Django Sessions:-

After completing the sessions work, you can delete them quite easily.

Just include this view function inside views.py file.

```
def delete_session(request):
    try:
        del request.session['name']
        del request.session['password']
    except KeyError:
        pass
    return HttpResponse("<h1>dataflair<br>Session Data cleared</h1>")
```

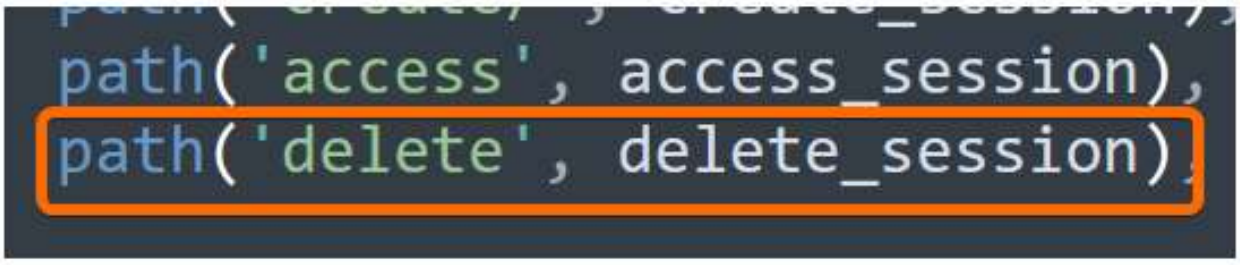
This code also follows pure python; we just used some exception handling.

To delete a session or any particular key of that session, we can use del.

```
1.      del request.session['key_name']
```

Now, to run this code add this to the urlpatterns:

```
1.      path('delete/', delete_session)
```



## What are the different inheritance styles in Django:-

Django has three possible inheritance styles:

***Abstract base classes*** — Used when you want to use the parent class to hold information that you don't want to type for each child model. Here, the parent class is never used in solitude

***Multi-table inheritance*** — Used when you have to subclass an existing model and want each model to have its own database table

***Proxy models*** — Used if you only want to modify the Python-level behavior of a model, without changing the 'models' fields in any way

# Iterators in Django ORM:-

Iterators in Python are basically containers that consist of a countable number of elements. Any object that is an iterator implements two methods which are, the __init__() and the __next__() methods. When you are making use of iterators in Django, the best situation to do it is when you have to process results that will require a large amount of memory space. To do this, you can make use of the iterator() method which basically evaluates a QuerySet and returns the corresponding iterator over the results.

## Customize the functionality of the Django admin:-

There are a number of ways to do this. You can piggyback on top of an add/ change form that is automatically generated by Django, you can add JavaScript modules using the *js parameter*. This parameter is basically a list of URLs that point to the JavaScript modules that are to be included in your project within a <script> tag. In case you want to do more rather than just playing around with from, you can exclusively write views for the admin.

## How django pass data from view to template:-

As we know django is a MVC framework. So, we separate business logic from presentational logic. We write business logic in views and we pass data to templates to present the data. The data that we pass from views to template is generally called as "context" data. Let's get started with an example.

Let's write a simple view that takes user information such as first name, last name and address and renders it in the template.

views.py

```python
from django.shortcuts import render

def user_data(request):
    context = {
        "first_name": "Anjaneyulu",
        "last_name": "Batta",
        "address": "Hyderabad, India"
    }
    template_name="user_template.html"
    return render(request, template_name, context)
```

user_template.html

```html
<html>
<head>
    <title>User Information</title>
</head>
<body>
<p>First Name: {{first_name }}</p>
<p>Last Name: {{last_name }}</p>
<p>Address: {{address}}</p>
</body>
</html>
```

**"render"** is the most used function in django. It combines a given template with a given context dictionary and returns an HttpResponse object with that rendered text. It takes three arguments "**request"**, **"template_name"** and "**context**" dictionary. In template we can access the context dict keys as names or variables and display them like **"{{ <variable/name>}}".**

# How django pass data from url to view:-

You can pass a URL parameter from the URL to a view using a path converter.

Firstly you have to create a path and map it to a view. For this, you have to edit your application's **urls.py** file.

A sample urls.py file will look like this-

```
from django.urls import path
from . import views
urlpatterns = [
    path("URL endpoint">/<path converter: URL parmeter name>,
view_name.function_name, name = "path name")
]
```

For example, if the requested URL is –

```
https://www.shop.tsinfo.com/products/12
```

- Then "**products**" will be the URL endpoint.
- A **path converter** defines which type of data will a parameter store. You can compare path converters with data types. In the above example, the path converter will be **int**.
- You will learn more about various path converters in the upcoming sections.
- **URL parameter name** will be the name that you will use to refer to the parameter.
- The **view_name** will be the view that will handle the request and the function_name is the function that will be executed when the request is made to the specified URL endpoint.
- The **name** will be the name of the path that you are going to create.