Working with Models and Databases in Django:

- Django provides built-in support for database operations, making it easier to manage data without writing SQL queries directly. It uses the **Object-Relational Mapping (ORM)** approach to map **model** classes to database tables.
- Django models define the structure of your database tables using Python classes. Each model class represents a table in the database, and its attributes define the columns.

Default Database (SQLite3):

- **SQLite3** is the default database used by Django, which is suitable for small-scale applications.
- For larger applications, you may need to configure other relational databases like **MySQL**, **PostgreSQL**, or **Oracle**.

Database Configuration in Django:

- Django allows you to configure the database in the `settings.py` file.

  SQLite3 (Default) Configuration:

```
DATABASES = {

    'default': {

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

        'NAME': BASE_DIR / 'db.sqlite3',

    }

}
```

Other Databases Configuration:

- You can change the database engine to **MySQL**, **PostgreSQL**, or **Oracle**. Here are the configurations:

1. **MySQL Configuration**:

```
DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'employeedb',

        'USER': 'root',

        'PASSWORD': 'root',

        'HOST': 'localhost',

        'PORT': '3306',

    }

}
```

2. **PostgreSQL Configuration**:

```
DATABASES = {

    'default': {

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

        'NAME': 'employeedb',

        'USER': 'postgres',

        'PASSWORD': 'password',

        'HOST': 'localhost',

        'PORT': '5432',

    }

}
```

3. **Oracle Configuration**:

```
DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'XE',

        'USER': 'system',

        'PASSWORD': 'system',

        'HOST': 'localhost',

        'PORT': '1521',

    }

}
```

Checking Database Connection:

- Run the following command to validate the project setup, including database connectivity.

```
python manage.py check
```

Working with Models:

- In Django, models are Python classes that define the structure of your database tables. Each model class corresponds to a table, and the attributes represent the table columns.

Example: DBProject

- Create a new project called DBProject inside the workspace (by activating the virtual environment)

```
django-admin startproject DBProject
```

- Move inside the project folder:

```
cd DBProject
```

- Create a new application **StudentApp**

```
python manage.py startapp StudentApp
```

- Register the StudentApp inside the settings.py file in INSTALLED_APPS
- For the sqlite3 database there is no need for the separate database configuration.
- To create a **student** table, we have to write a model class inside the models.py file.

Defining a Model:

- We have to write all the model classes for an application inside the 'models.py' file for that particular application folder.

```
from django.db import models

class Student(models.Model):

    roll = models.IntegerField(unique=True)

    name = models.CharField(max_length=100)

    age = models.IntegerField()

    email = models.EmailField(unique=True)

    address = models.TextField()

    phone_number = models.CharField(max_length=15,
unique=True)

    admission_date = models.DateField(auto_now_add=True)

    is_active = models.BooleanField(default=True)


    def __str__(self):

        return self.name
```

Explanation:

- `roll`: Unique identification number for the student.

- `name`: Name of the student (max 100 characters).
- `age`: Age of the student.
- `email`: Email field with a unique constraint.
- `address`: Text field to store detailed addresses.
- `phone_number`: Stores contact number with uniqueness.
- `admission_date`: Auto-filled when a student is added.
- `is_active`: Boolean field indicating active students.

Note: Django automatically creates an `id` column as a **primary key**, even if it is not explicitly defined in addition with other columns.

- The above **Model** class will be converted into the database table.
- For the above Model class the corresponding table name will be generated in the following format inside the database:

    appname_modelclassname

Example:

StudentApp_student

Applying Migrations to Create the Student Table:

- To convert your model definitions into actual database tables, Django uses migrations.

Creating Migrations:

```
python manage.py makemigrations
```

- The above command generates migration files that describe the changes in the database schema (e.g., creating tables).
- Inside the app\migrations\ folder a new file will be created with the name "**0001_initial.py**"

To View the Generated SQL:

- To view the SQL statements Django will use, run:

```
python manage.py sqlmigrate <app_name> <migration_number>
```

Example:

```
python manage.py sqlmigrate StudentApp 0001
```

Applying Migrations:

- To apply the migrations and create/update the database tables:

  ```
  python manage.py migrate
  ```

- With the above command all the installed app related database tables will be created along with our application related database tables inside the "**`db.sqlite3`**" database.

'id' field:

1. For every table django will generate a special column named with "**id**".
2. id is a primary key.(unique value for every record)
3. It is an auto increment field. While inserting data we are not required to provide value for this field.
4. This field is of type: **BigAutoFeild**
5. We can override the behaviour of the **id** field and we can make our own field as **id**.
6. Every column is by default **not null**.

Note: to make the roll as the primary key: use the following way:

```
roll = models.IntegerField(primary_key=True)
```

- In this case extra **id** fields will not be created.
- We can see all the tables by opening the db.sqlite3 database inside the **db-browser** software.
- In this case roll field will not be auto_incremented value, to make this roll as auto_incremented value we need to make use of:
  - roll = models.AutoField(primary_key=True)

Summary:

1. Perform the db configurations inside the settings.py file.
2. Write the model classes inside the models.py file of our application.
3. python manage.py makemigrations

4. python manage.py migrate

Difference between makemigrations and migrate command:

python manage.py makemigrations:

- **Purpose**: Detects changes in your model definitions (e.g., adding a field, modifying a model) and generates migration files that describe those changes.
- **Output**: Creates Python files (e.g., 0001_initial.py) in the migrations/ directory of your app. These files contain instructions for altering the database schema.
- **Effect**: Does **not** modify the database—it only prepares the migration plan.
- **Example**: If you add a grade field to the Student model, makemigrations generates a migration file to add that column.

python manage.py migrate:

- **Purpose**: Applies the migration files to the database, executing the SQL commands to create, update, or delete tables/columns as needed.
- **Effect**: Updates the actual database schema and creates/updates tables (e.g., testapp_student).
- **Example**: Running migrate after makemigrations will add the grade column to the testapp_student table in the database.

Key Difference: makemigrations is about **planning** changes, while migrate is about **executing** those changes.

Advantage of creating tables by using the "migrate" command:

- In addition to our application tables, default application tables also will be created.

Accessing the tables inside the admin panel:

- Now to see all the tables, access the admin application: check the url for admin inside the urls.py file at project level.
- Run the server :

```
python manage.py runserver
```

- Access the admin interface

```
http://127.0.0.1:8000/admin
```

Note: To access the admin interface, we need to create a super user:

```
python manage.py createsuperuser
```

username: ratan

email: ratan@gmail.com

password: 123

retype password: 123

Now we can access the admin interface by providing the above username and password.

- By Default our application specific created tables are not visible inside the admin interface
- We have to register the model inside the admin interface then only it will be visible.
- We have to do the registration inside the admin.py file of the application folder.

```
from django.contrib import admin

from StudentApp.models import Student

admin.site.register(Student) # to register all the student field (default behaviour)
```

- To register only the specific fields we need to create a separate class:

```
from django.contrib import admin

from StudentApp.models import Student

class StudentAdmin(admin.ModelAdmin):
        list_display = ['roll', 'name', 'age', 'email',
            'address', 'phone_number', 'admission_date', 'is_active']


admin.site.register(Student, StudentAdmin)
```

- Now we can see the Student table related information inside the admin interface and from there we can perform the insert and delete operations also.

Note: for every model class we have can define a separate Admin class inside the admin.py file. In that admin class we need to specify which column should be required to display as a **list_display**

We have to register every model and corresponding ModelAdmin class in admin.site

- Add list_filter = ['is_active', 'admission_date'] to filter students by these fields.
- Add search_fields = ['name', 'email'] to enable searching by name or email. Example:

Example:

```
from django.contrib import admin

from StudentApp.models import Student

class StudentAdmin(admin.ModelAdmin):

        list_display = ['roll', 'name', 'age', 'email',

            'address', 'phone_number', 'admission_date', 'is_active']

       list_filter = ['is_active', 'admission_date']

       search_fields = ['name', 'email']


       # to register all the employee field

       admin.site.register(Student, StudentAdmin)
```

Django ORM: Performing Database Operations:

- Django ORM (Object-Relational Mapping) allows interacting with the database using Python code instead of SQL queries. Below are various ORM methods to perform database operations:

1. Retrieving Data (SELECT Queries):

Retrieve All Records:

```
students = Student.objects.all()  # Returns all student records
```

Retrieve a Single Record by Primary Key (ID):

```
student = Student.objects.get(id=1)  # Fetches the student with ID = 1
```

Note: If no record exists, it raises a DoesNotExist exception.

Retrieve a Single Record by a Non-Primary Key:

```
student = Student.objects.get(roll=101)  # Fetches student with roll 101
```

Note: get() raises an error if multiple records exist. Use filter() for multiple records.

2. Filtering Data:

Retrieve Students Based on Conditions:

Get students with marks less than 500

```
students = Student.objects.filter(marks__lt=500)
```

Get students with marks less than or equal to 500

```
students = Student.objects.filter(marks__lte=500)
```

Get students whose name starts with "A"

```
students = Student.objects.filter(name__startswith="A")
```

Case-Insensitive Search for Students Named "Kumar":

```
students = Student.objects.filter(name__icontains="kumar")
```

Applying multiple conditions:

```
students = Student.objects.filter(marks__gt=500, address__icontains="New York")
```

To apply the OR (|) operation, you must use the Q object:

```
from django.db.models import Q

students = Student.objects.filter(Q(marks__gt=500) | Q(address__icontains="New
York"))
```

3. Sorting the record:

Retrieve Students in Sorted Order:

```
students_asc = Student.objects.all().order_by("marks")  # Ascending order

students_desc = Student.objects.all().order_by("-marks")  # Descending order
```

4. Retrieving First and Last Record:

```
first_student = Student.objects.first()  # Fetches the first student record

last_student = Student.objects.last()  # Fetches the last student record
```

5. Inserting New Records (INSERT Queries):

Method 1: Using create()

```
Student.objects.create(roll=101, name="Ram", address="Delhi", marks=75,
email="ram@example.com", phone=9876543210, dob="2000-01-01")
```

Method 2: Using Object and save()

```
student = Student(roll=102, name="Shyam", address="Mumbai", marks=80,
email="shyam@example.com", phone=9876543211, dob="2001-05-15")

student.save()  # Save to database
```

6. Updating Records (UPDATE Queries):

Updating a Single Record:

```
student = Student.objects.get(id=1)  # Fetch the student by ID

student.address = "Mumbai"  # Modify the address

student.save()  # Save changes
```

Updating multiple records:

```
students = Student.objects.filter(marks__lt=500)

for student in students:

    student.marks += 10

   student.save()

# Student.objects.bulk_update(students, ["marks"])
```

7. **Deleting Records (DELETE Queries):**

   Delete a Single Record by Primary Key:

```
student = Student.objects.get(id=1)

student.delete()
```

   Delete Multiple Records

   - **Delete students who scored less than 30.**

```
Student.objects.filter(marks__lt=300).delete()
```

   Delete All Records:

```
Student.objects.all().delete()
```

8. **Aggregation Functions (SUM, AVG, MAX, MIN, COUNT)**

   - Django provides built-in aggregate functions for database operations:

```
from django.db.models import Sum, Avg, Max, Min, Count

total_marks = Student.objects.aggregate(Sum("marks"))  # Sum of all marks

average_marks = Student.objects.aggregate(Avg("marks"))  # Average marks
```

```
max_marks = Student.objects.aggregate(Max("marks"))  # Maximum marks

min_marks = Student.objects.aggregate(Min("marks"))  # Minimum marks

total_students = Student.objects.aggregate(Count("id"))  # Total number of
students
```

9. Limiting Query Results:

Retrieve First 5 Students

```
students = Student.objects.all()[:5]
```

10. Bulk Insert:

```
students = [

Student(roll=103, name="Alice", address="New York", marks=70,
email="alice@example.com", phone_number=9876543212, dob="2002-03-10"),

Student(roll=104, name="Bob", address="Los Angeles", marks=55,
email="bob@example.com", phone_number=9876543213, dob="2003-07-20"),

]

Student.objects.bulk_create(students)
```

11. Selective column retrieval:

```
students = Student.objects.only("name", "email")  # Fetch only 'name' and 'email'

students = Student.objects.defer("phone_number")  # Fetch all fields except
'phone_number'
```

Note: To test the above ORM methods we can use the **Django shell**:

Step1: Open the Django Shell

- Run the following command inside your Django project directory:

```
python manage.py shell
```

- This opens an interactive Python shell with Django loaded.

Step 2: Import Your Model

- Once inside the shell, import your model:

```
from StudentApp.models import Student
```

Step 3: Run ORM Queries:

- Now, you can run ORM queries and test them live.

```
students = Student.objects.all()

print(student)  # Output: Emma Watson
```

Step 4: Exit the Shell:

- Once you're done testing, exit the Django shell:
    - exit()

      OR

    - quit()

Generate the fake data using django-seed library:

- django-seed is a django based customized application to generate fake data for every model automatically.

  Documentation: https://github.com/brobin/django-seed

Steps to use django-seed:

Step1. pip install django-seed

Step2. Register "django_seed" application inside the INSTALLED_APPS of the settings.py file

Step3. generate and send fake data to the models.

```
python manage.py seed StudentApp --number=5
```

Note: if error comes :

```
pip install psycopg2
```

Assignment:

- **Seed the 10 student records inside the table and display those records inside the template (Bootstrap table)**
- **Make use of the following url:**
  - **students/getallstudents**

Folder structure:

```
StudentProject/
```

```
│-- StudentProject/
│           │-- urls.py  👈 (Includes 'students/')
│-- StudentApp/
│           │-- urls.py  👈 (Defines 'getallstudents/')
│           │-- views.py
│           |-- templates/
│                 │-- students_list.html
```

Classroom Exercise Example:

- Create a new Project : DBProject2

  ```
  django-admin startproject DBProject2
  ```

- Move inside the DBProject folder:

  ```
  cd DBProject2
  ```

- Create a new application called StudentApp

  ```
  python manage.py startapp StudentApp
  ```

- Register StudentApp inside the settings.py file
- Define the following models inside the StudentApp/models.py file

models.py:

```
from django.db import models

# Create your models here.
```

```
class Student(models.Model):

    roll = models.IntegerField(unique=True)

    name = models.CharField(max_length=20)

    address = models.TextField(null=True, blank=True)

    email = models.EmailField(unique=True)

    marks = models.IntegerField()

    def __str__(self):

        return f"Roll is: {self.roll}, Name is: {self.name}"


class Course(models.Model):

    course_id = models.AutoField(primary_key=True)

    course_name = models.CharField(max_length=20, unique=True)

    fee = models.IntegerField()

    duration = models.CharField(max_length=20)

    image = models.URLField()

    def __str__(self):

        return f"Course Name is: {self.course_name}"
```

- **Do the migrations**

  ```
  python manage.py makemigrations

  python manage.py migrate
  ```

- **Register both model classes inside the StudentApp/admins.py file**

admins.py:

```
from django.contrib import admin

from StudentApp.models import Student, Course

# Register your models here.

class StudentAdmin(admin.ModelAdmin):

    list_display = ['roll', 'name', 'address', 'email', 'marks']

    search_fields = ['email', 'address']

    list_filter = ['address']

class CourseAdmin(admin.ModelAdmin):

    list_display = ['course_id', 'course_name', 'fee', 'duration',
'image']

admin.site.register(Student, StudentAdmin)

admin.site.register(Course, CourseAdmin)
```

- **Create a super user to access the admin interface**

  ```
  python manage.py createsuperuser
  ```

- **Run the server and access the admin interface**

  ```
  python manage.py runserver
  ```

  http://127.0.0.1:8000/admin

- **Add few records in both the tables(Student and Course) from the admin interface**

- **Define the following view functions inside StudentApp/views.py file**

views.py:

```
from django.shortcuts import render, redirect

from StudentApp.models import Student, Course

# Create your views here.

def all_student_view(request):

    students = Student.objects.all()

    return render(request, 'allstudents.html',
context={'student_list': students})


def get_student_view(request, roll):

    student = Student.objects.get(roll=roll)

    result = "Pass"

    if student.marks < 700:

        result = "Fail"

    return render(request, 'student.html', context={'studentdata':
student, 'result': result})


def delete_student_view(request, roll):

    student = Student.objects.get(roll=roll)

    student.delete()

    return redirect('allstudents')


def all_course_view(request):

    courses = Course.objects.all()
    return render(request, 'allcourses.html',
context={'course_list': courses})


def delete_course_view(request, course_id):

    course = Course.objects.get(course_id=course_id)

    course.delete()

    return redirect('allcourses')
```

- **Define the mappings for the above view functions inside StudentApp/urls.py file**

urls.py:

```
from django.urls import path

from . import views


urlpatterns = [


    path('', views.all_student_view, name='allstudents'),

    path('getstudent/<int:roll>/', views.get_student_view,
name='getstudent'),

    path('students/<int:roll>/delete/',

         views.delete_student_view, name='deletestudent'),

    path('allcourses/', views.all_course_view, name='allcourses'),

    path('courses/<int:course_id>/delete/',

         views.delete_course_view, name='deletecourse')

]
```

- **Include the above StudentApp.urls.py file inside the Project level urls.py file:**

DBProject/urls.py

```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('StudentApp.urls'))

]
```

- **Define the following html files inside StudentApp/templates folder.**

allstudents.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstra
p.min.css" rel="stylesheet"
integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0J
MhjY6hW+ALEwIH" crossorigin="anonymous">

</head>
```

```
<body>

    <h1 class="text-center">All Student Details</h1>

    <div class="container">

        {% if student_list %}

        <table class="table table-dark">


            <thead>

                <tr>

                    <th>Roll</th>

                    <th>Name</th>

                    <th>Address</th>

                    <th>Email</th>

                    <th>Marks</th>

                    <th>Actions</th>

                </tr>

            </thead>

            <tbody>


                {% for student in student_list %}


                <tr>

                    <td>{{student.roll}}</td>
                    <td>{{student.name}}</td>

                    <td>{{student.address}}</td>

                    <td>{{student.email}}</td>

                    <td>{{student.marks}}</td>

                    <td>

  <a href="{% url 'getstudent' roll=student.roll  %}"

         class="btn btn-primary btn-sm">GETDETAILS</a>

  <a onclick="return confirm('Are You Sure ?')" href="{% url
'deletestudent' roll=student.roll %}"

         class="btn btn-sm btn-danger">DELETE</a>

                    </td>

                </tr>

                {% endfor %}

            </tbody>

        </table>

        {% else %}



        <h3>No Recored Available inside Database</h3>

        <h4>Please add Student Records</h4>



        <a href="/admin" class="btn btn-success">Add New Student</a>



        {% endif %}
```

```
    </div>

    <a href="{% url 'allcourses' %}" class="btn btn-primary">GET
COURSE DETAILS</a>

</body>

</html>
```

student.html:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

</head>

<body bgcolor="wheat">

    <h1 style="text-align: center;">Welcome
{{studentdata.name}}</h1>

    <h4>Roll is: {{studentdata.roll}}</h4>

    <h4>Name is: {{studentdata.name}}</h4>

    <h4>Address is: {{studentdata.address}}</h4>

    <h4>Email is: {{studentdata.email}}</h4>

    <h4>Marks is: {{studentdata.marks}}</h4>
```

```
          <hr>

          <h2>Your Result is: {{result}}</h2>

      </body>

      </html>
```

allcourses.html:

```
<!DOCTYPE html>

{% load static %}

<html lang="en">

<head>

    <meta charset="UTF-8">

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

    <title>Document</title>

    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.
css" rel="stylesheet"
integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6h
W+ALEwIH" crossorigin="anonymous">

    <style>

        body {

            background-image: url("{% static 'images/img1.jpg' %}");

            background-position: center;

        }

    </style>

</head>

<body>
```

```
    <h1 class="text-center">All Course Details</h1>

    <a href="{% url 'allstudents' %}" class="btn btn-success">Back</a>

    <hr>

    {% if course_list %}

    <div class="contianer d-flex gap-4 flex-wrap">

        {% for course in course_list %}



        <div class="card" style="width: 18rem;">

            <img src="{{course.image}}" class="card-img-top" alt="Image
not loaded">

            <div class="card-body">

                <h5 class="card-title">{{course.course_name}}</h5>

                <p class="card-text">{{course.duration}}</p>

                <a href="#" class="btn btn-primary">{{course.fee}}</a>

                <a href="{% url 'deletecourse' course_id=course.course_id
%}" class="btn btn-danger"

                    onclick="return confirm('Are You Sure ?')">DELETE</a>

            </div>

        </div>

        {% endfor %}

    </div>

    {% else %}

    <h2>No Course found inside Database</h2>

    <h4>Please add some course from Admin interface</h4>
    <a href="/admin" class="btn btn-primary">Add New Course</a>

    {% endif %}

</body>

</html>
```

- **Place a background image called img1.jpg inside the StudentApp/static/images folder.**
- **Restart the server and access the application:**

  python manage.py runserver

  http://127.0.0.1:8000/

Image uploading Example:

Modify the above application as follows:

Step1: write the following configurations inside the **settings.py** file .

```
import os

MEDIA_URL = '/media/'

MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

- With this all the uploaded images will be stored inside the **media** folder

Step2: Change the **Course** model class inside the **models.py** file

```
class Course(models.Model):
    course_id = models.AutoField(primary_key=True)

    course_name = models.CharField(max_length=20, unique=True)

    fee = models.IntegerField()

    duration = models.CharField(max_length=20)

    image = models.ImageField(

        upload_to='course_images/', blank=True, null=True)


    def __str__(self):

        return f"Course Name is: {self.course_name}"
```

- This means images will be saved inside:

  /media/course_images/

- If the uploaded file is named `python.jpg`, it will be stored as:

  /media/course_images/python.jpg

- Instead of storing the actual image, Django saves **only the relative file path** inside the database.

Example Database Entry:

course_id	course_name	fee	duration	image
1	DJango	5000	45 days	course_images/python.jpg

Step 3: Modify the Project level **urls.py** file as follows:

Project level urls.py

```
from django.conf.urls.static import static

from django.conf import settings

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('', include('StudentApp.urls'))

]

if settings.DEBUG:

   urlpatterns += 
static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

- These lines **enable Django to serve uploaded media files (like images, PDFs, videos, etc.) during development** when `DEBUG = True`.
- Django **does not** automatically serve media files (uploaded by users) like it does for static files (`STATIC_URL`).
- So, we need this configuration to make uploaded files accessible when running the Django development server (`python manage.py runserver`).
- When `DEBUG = False` (in production), Django **does NOT** serve media files. Instead, you need a web server like **NGINX** or **Apache** to serve them.

Step 4: Display the image inside the HTML template:

```
<img src="{{ course.image.url }}" class="card-img-top" alt="Course Image">
```