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)
```

For the editable field :

```
list_editable = ['name', 'age', 'email',
                 'address', 'phone_number', 'admission_date', 'is_active']
```

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:

Modify allcourses.html file

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

Note: while migration if error comes then install the Pillow library

```
python -m pip install Pillow
```

Association mapping in Django-ORM:

- Association Mapping refers to defining relationships between database tables (or models in Django) using the Object-Relational Mapping (ORM) system. Django ORM provides a high-level abstraction to manage these relationships without writing raw SQL, making it easier to work with related data. In relational databases, associations are implemented using **primary keys** and **foreign keys**, and Django simplifies this through its model fields.

Django supports three primary types of association mappings:

1. **One-to-One Mapping**
2. **One-to-Many Mapping (Foreign Key)**
3. **Many-to-Many Mapping**

1. One-to-One Mapping

- **Definition**: A one-to-one relationship means that one record in a table is associated with exactly one record in another table, and vice versa.
- **Use Case**: Used when you want to split a model into two parts for logical separation or to store additional, optional data.
- **Django Field:** OneToOneField
- **Database Representation:** Implemented as a foreign key with a **UNIQUE** constraint in the database (e.g., **student_id** in the **StudentProfile** table).

Example: Student and **StudentProfile**

- **Create a new Project called : RelationshipProject**

```
django-admin startproject RelationshipProject
```

- **Move inside the project directory:**

```
cd RelationshipProject
```

- **Create a new Application: StudentApp**

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

- **Register the StudentApp inside the settings.py file**
- **Create Student and StudentProfile model class inside the StudentApp/models.py file.**

- A **Student** model might have basic details, while a **StudentProfile** model contains extended information like a bio or experience.

```
from django.db import models

class Student(models.Model):
```

```
    roll = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)

    def __str__(self):
        return self.name


class StudentProfile(models.Model):
    bio = models.TextField(blank=True, null=True)
    experience = models.CharField(max_length=100)
    student = models.OneToOneField(Student,
on_delete=models.CASCADE)

    def __str__(self):
        return f"Profile of {self.student.name}"
```

Explanation:

- student: A OneToOneField linking to the Student model. Each StudentProfile is tied to exactly one Student.
- on_delete=models.CASCADE: If the Student is deleted, the associated StudentProfile is also deleted.

- In the database, student_id is added to the StudentProfile table as a foreign key with a unique constraint.

Applying Migrations

```
python manage.py makemigrations

python manage.py migrate
```

Database Schema for the above model classes:

StudentApp_student table:

roll (PK)	name	email
100	Alice	alice@gmail.com
101	Bob	bob@gmail.com

StudentApp_studentprofile table:

id (PK)	bio	experience	student_id (FK + Unique)
1	Loves coding	2 years	100
2	Loves reading	3 years	101

Register both model classes inside the admin.py:

```python
from django.contrib import admin
from StudentApp.models import Student, StudentProfile
# Register your models here.
class StudentAdmin(admin.ModelAdmin):
    list_display = ["roll", "name", "email"]
class StudentProfileAdmin(admin.ModelAdmin):
    list_display = ["bio", "experience", "student"]

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

- Add a few records inside both models from the admin interface.

Usage in Python Shell:

```python
from StudentApp.models import Student, StudentProfile
# Create a student
student1 = Student.objects.create(roll=101, name="Alice",
email="alice@example.com")

# Create a profile for the student
```

```
profile = StudentProfile.objects.create(student=student1, bio="Loves
coding", experience="2 years")


# Access related data

print(student1.studentprofile.bio)  # Output: "Loves coding"

print(profile.student.name)         # Output: "Alice"


# Get profile by student roll

student = Student.objects.get(roll=101)

profile = student.studentprofile

print(profile)  # Output: <StudentProfile: Profile of Alice>
```

Reverse Access: Django automatically creates a reverse relation (studentprofile) from Student to StudentProfile.

```
# Get profile by student roll:

student = Student.objects.get(roll= 101)

profile = student.studentprofile

print(profile)
```

Assignment: Update the StudentProfile experience with **4 years** for all the students whose **name** is **Alice**:

Solution: Since name isn't a primary key, Student.objects.filter(name="Alice") returns a QuerySet of all matching records. To update all StudentProfile instances for these students, use a loop or a bulk update.

1. **Using a Loop**:

```
alice_students = Student.objects.filter(name="Alice")

print(alice_students) #  <QuerySet [<Student: Alice>, <Student: Alice>]>

 for student in alice_students:

        profile = student.studentprofile

        profile.experience = "4 years"

        profile.save()

       # Updated All  Alice's experience to 4 years
```

2. **Using Bulk Update:**

```
updated_count =
StudentProfile.objects.filter(student__name="Alice").update(experien
ce="4 years")

print(f"Updated {updated_count} profiles")

# Output example: Updated 2 profiles
```

- **Define a view function to render all the students record inside a HTML table:**

StudentApp/views.py

```
from django.shortcuts import render
from StudentApp.models import Student, StudentProfile


# Create your views here.
def get_all_student_view(request):
    students = Student.objects.all()
    return render(request, 'allstudents.html', context={'student_list':
students})


def get_student_profile(request, roll):
    student = Student.objects.filter(roll=roll).first()
    # profile = student.studentprofile
    return render(request, 'profile.html', context={'student': student})
```

- **Define the url path for the above view functions**

StudentApp/urls.py

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.get_all_student_view, name="allstudents"),
```

```
    path('student/<int:roll>/profile/', views.get_student_profile,
name='profile')

]
```

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

Project level urls.py

```
from django.contrib import admin

from django.urls import path, include

urlpatterns = [

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

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

]
```

- **Create the following HTML files inside the 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>

</head>
```

```
<body bgcolor="wheat">
    <h1 style="text-align: center;">All Student Details</h1>
    <table align="center" border="1">
        <thead>
            <tr>
                <th>Roll</th>
                <th>Name</th>
                <th>Email</th>
                <th>Action</th>
            </tr>
        </thead>
        <tbody>
            {% for student in student_list %}
            <tr>
                <td>{{student.roll}}</td>
                <td>{{student.name}}</td>
                <td>{{student.email}}</td>
                <td>
                    <a href="{% url 'profile' roll=student.roll %}">GET
PROFILE</a>
                </td>
            </tr>
            {% endfor %}
        </tbody>
```

```
    </table>
</body>
</html>
```

profile.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="cyan">

    <h1>Welcome {{student.name}}</h1>

    <hr>
    <h4>Student Roll: {{student.roll}}</h4>
    <h4>Student Name: {{student.name}}</h4>
    <h4>Student Email: {{student.email}}</h4>
    <h4>Student Bio: {{student.studentprofile.bio}} </h4>
    <h4>Student Experience: {{student.studentprofile.experience}}</h4>
```

```
    <a href="{% url 'allstudents' %}">Back</a>
</body>
</html>
```

- **Run the server and access the application:**

  python manage.py runserver

  http://127.0.01:8000/

2. One-to-Many Mapping in Django ORM:

Definition

- A one-to-many relationship means one record in a table (the "one" side) can be associated with multiple records in another table (the "many" side), but each record on the "many" side is linked to only one record on the "one" side.

Use Case

- Commonly used for hierarchical or ownership relationships, such as a department having multiple employees, where each employee belongs to exactly one department.

Django Field

- ForeignKey: Establishes the one-to-many relationship from the "many" side (e.g., **Employee**) to the "one" side (e.g., **Department**).

Database Representation

- Implemented using a foreign key column in the "many" side table (e.g., **department_id** in the Employee table), referencing the primary key of the "one" side table. No uniqueness constraint is needed, as multiple records can share the same foreign key value.

Example: Department and Employee

- **Department:** Represents a department with a unique ID, name, and location.
- **Employee:** Represents an employee with an ID, name, salary, and a reference to their department.

Create EmployeeApp inside the above Project:

```
python manage.py startapp EmployeeApp
```

Register the EmployeeApp inside the settings.py file:

Add the following classes inside the models.py file inside EmployeeApp folder

```
# EmployeeApp/models.py

from django.db import models


class Department(models.Model):

    dept_id = models.IntegerField(primary_key=True)

    name = models.CharField(max_length=100)

    location = models.CharField(max_length=100)


    def __str__(self):
```

```
        return self.name


class Employee(models.Model):
    emp_id = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=100)
    salary = models.DecimalField(max_digits=10, decimal_places=2)
    department = models.ForeignKey(Department,
on_delete=models.CASCADE)


    def __str__(self):
        return self.name
```

Explanation

- department: A ForeignKey in Employee linking to the Department. Each Employee belongs to one Department, but a Department can have multiple Employee instances.
- on_delete=models.CASCADE: If a Department is deleted, all associated Employee records are also deleted.
- **Database**: The employee table has a department_id column (foreign key to Department.department_id), allowing multiple employees to reference the same department.

Applying Migrations

- Create and apply migrations to set up the database schema:

```
python manage.py makemigrations

python manage.py migrate
```

Database Schema

EmployeApp_department table:

dept_id (PK)	name	location
100	IT	Mumbai
101	HR	Chennai

EmployeeApp_employee table:

emp_id (PK)	name	salary	department_id (FK)
1	Alice	50000.00	100
2	Bob	60000.00	100
3	Charlie	55000.00	101

Admin Registration

- Register the models in the admin interface for easy management:

Inside **EmployeeApp/admin.py** file

```
# EmployeeApp/admin.py

from django.contrib import admin

from EmployeeApp.models import Department, Employee


class DepartmentAdmin(admin.ModelAdmin):
```

```
    list_display = ["dept_id", "name", "location"]


class EmployeeAdmin(admin.ModelAdmin):
    list_display = ["emp_id", "name", "salary", "department"]


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

- **Add few records in both the models from the admin interface**

Shell Usage:

python manage.py shell

1. Insert Data

```
from EmployeeApp.models import Department, Employee

# Create a department

dept1 = Department.objects.create(dept_id=100, name="IT",
location="Mumbai")

# Create employees in that department

emp1 = Employee.objects.create(emp_id=1, name="Alice",
salary=50000.00, department=dept1)

emp2 = Employee.objects.create(emp_id=2, name="Bob",
salary=60000.00, department=dept1)
```

```
# Create another department

dept2 = Department.objects.create(dept_id="101", name="HR",
location="Chennai")

# Create an employee in the second department

emp3 = Employee.objects.create(emp_id=3, name="Charlie",
salary=55000.00, department=dept2)
```

Access Related Data

```
# Forward access: Employee to Department

print(emp1.department.name)  # Output: "IT"

print(emp3.department.name)  # Output: "HR"


# Reverse access: Department to Employees

print(dept1.employee_set.all())  # Output: <QuerySet [<Employee:
Alice>, <Employee: Bob>]>

print(dept2.employee_set.all())  # Output: <QuerySet [<Employee:
Charlie>]>
```

Reverse Access: Django creates a reverse relation (employee_set) from Department to Employee. You can customize this with **related_name**:

Example:

```
department = models.ForeignKey(Department, on_delete=models.CASCADE,
related_name="employees")
```

Then use dept1.employees.all() instead of dept1.employee_set.all().

Sample Operations

1. Query Employees by Department

```
# Get all employees in the IT department

it_employees = Employee.objects.filter(department__name="IT")

print(it_employees)  # <QuerySet [<Employee: Alice>, <Employee: Bob>]>
```

2. Update Salaries for a Department

```
# Increase salary by 10000 for all employees who is working in IT dept

it_employees = Employee.objects.filter(department__dept_id=100)

for emp in it_employees:

    emp.salary = emp.salary + 10000

    emp.save()

# Verify

for emp in it_employees:

    print(f"{emp.name}: {emp.salary}")  # Alice: 55000.00, Bob: 66000.00
```

3. Add an Employee to a Department

```
dept1 = Department.objects.get(dept_id=100)

new_emp = Employee.objects.create(emp_id=4, name="David", salary=52000.00,
department=dept1)

print(dept1.employee_set.all())  # <QuerySet [<Employee: Alice>,
<Employee: Bob>, <Employee: David>]>
```

4. Delete a Department (Cascades to Employees)

```
dept2 = Department.objects.get(dept_id=100)

dept2.delete()  # Deletes HR and Charlie

print(Employee.objects.filter(department__dept_id=100).exists())  # False
```

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

EmployeeApp.views.py

```
from django.shortcuts import render

from EmployeeApp.models import Employee, Department

# Create your views here.

def get_all_dept_view(request):

    depts = Department.objects.all()

    return render(request, 'alldepartments.html', context={"depts":
depts})

def get_dept_emps_view(request, deptid):

    dept = Department.objects.filter(dept_id=deptid).first()

    emps = dept.employee_set.all()

    return render(request, 'allemployee.html', context={"department":
dept, 'employees': emps})



def get_emp_dept_view(request, empid):

    emp = Employee.objects.filter(emp_id=empid).first()

    dept = emp.department

    return render(request, 'department.html', context={'department':
dept})
```

- Define the url path for the above view functions inside **EmployeeApp/urls.py** file

EmployeeApp/urls.py

```
from django.urls import path

from . import views

urlpatterns = [

    path('', views.get_all_dept_view, name="alldeparments"),

    path('<int:deptid>/employees/', views.get_dept_emps_view,
name='dept_emps'),

    path('employee/<int:empid>/', views.get_emp_dept_view,
name='emp_dept')

]
```

- Include the above urls.py file inside the **project level urls.py** file

project level urls.py file:

```
from django.contrib import admin

from django.urls import path, include


urlpatterns = [

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

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

    path('departments/', include('EmployeeApp.urls'))

]
```

- Create the following HTML files inside the **EmployeeApp/templates** folder:

alldepartments.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="pink">
    <h1 style="text-align: center;">All Department Details</h1>
    <hr>
    <table align="center" border="1">
        <thead>
            <tr>
                <th>Department Id</th>
                <th>Department Name</th>
                <th>Location</th>
                <th>Action</th>
            </tr>
        </thead>
        <tbody>
```

```
            {% for dept in depts %}


            <tr>
                <td>{{dept.dept_id}}</td>
                <td>{{dept.name}}</td>
                <td>{{dept.location}}</td>
                <td>
                    <a href="{% url 'dept_emps' deptid=dept.dept_id
%}">GET ALL EMPLOYEES</a>
                </td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</body>
</html>
```

allemployee.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="pink">
    <h1 style="text-align: center;">All Employees in Department
{{department.name}}</h1>
    <hr>
    {% if employees %}
    <h2>All Emplyees List</h2>
    <table border="1" align="center">
        <thead>
            <tr>
                <th>Employee ID</th>
                <th>Employee Name</th>
                <th>Salary</th>
                <th>Action</th>
            </tr>
        </thead>
        <tbody>
            {% for emp in employees %}

            <tr>
                <td>{{emp.emp_id}}</td>
                <td>{{emp.name}}</td>
```

```
                <td>{{emp.salary}}</td>
                <td>
                    <a href="{% url 'emp_dept' empid=emp.emp_id %}">Get
Depatment</a>
                </td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
    {% else %}
    <h2>No Employee inside this Department</h2>
    {% endif %}
</body>
</html>
```

department.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">


    <h2>Department ID is: {{department.dept_id}}</h2>

    <h2>Department Name is: {{department.name}} </h2>

    <h2>Location is: {{department.location}}</h2>

    <a href="{% url 'dept_emps' deptid=department.dept_id %}">Back</a>

</body>

</html>
```

- Run the server and access the above application:

  python manage.py runserver

  http://127.0.0.1:8000/departmetns

- To access the above path of **EmployeeApp** from the **StudentApp** home page i.e. from the **allstudents.html** file, add the following link inside the **allstudents.html** file:

```
<a href="{% url 'alldeparments' %}">View All Departments</a>
```

3. Many-to-Many Mapping in Django ORM:

Definition

- A many-to-many relationship allows multiple records in one table to be associated with multiple records in another table, and vice versa. For example, a student can enroll in multiple courses, and a course can have multiple students enrolled.

Use Case

- Ideal for scenarios where entities have a mutual, non-exclusive relationship, such as students enrolling in multiple courses and courses being taken by multiple students.

Django Field

- **ManyToManyField:** Defines a many-to-many relationship between two models.

Database Representation

- Implemented using an intermediary (junction) table that contains foreign keys to both related tables. Django automatically generates this table when using ManyToManyField

Example: Student and Course

- **Student**: Represents a student with roll number, name, email, and address.
- **Course**: Represents a course with course ID, name, fee, and duration.
- **Relationship**: A student can enroll in multiple courses, and a course can have multiple students.

- Create a CourseApp inside the RelationshipProject

  python manage.py startapp CourseApp

- Register the **CourseApp** inside the settings.py file
- Define the following model classes inside the models.py of **CourseApp**

```
# CourseApp/models.py (Simplified Version)
```

```python
from django.db import models


class Course(models.Model):
    course_id = models.IntegerField(primary_key=True)
    cname = models.CharField(max_length=100)
    fee = models.DecimalField(max_digits=10, decimal_places=2)
    duration = models.CharField(max_length=10)

    def __str__(self):
        return self.cname

class Student(models.Model):
    roll = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    address = models.TextField()
    courses = models.ManyToManyField(Course)  # Implicit many-to-many
    # courses = models.ManyToManyField('Course')
    def __str__(self):
        return self.name
```

Explanation:

- **courses = models.ManyToManyField(Course)**: Defines the many-to-many relationship directly in Student. Django creates an implicit junction table(3rd table).
- A **ManyToManyField** does **not** use `ForeignKey`, so `on_delete` is **not required or allowed**.
- If a **Course** is deleted, Django will **automatically remove** the corresponding entries in the join table.
- If a **Student** is deleted, their course enrollments are also removed from the join table.
- This behavior is managed by Django without needing `on_delete`.

Applying Migrations

python manage.py makemigrations

python manage.py migrate

Table Structure

1. Courseapp_student:
    - roll (INTEGER, PRIMARY KEY)
    - name (VARCHAR(100), NOT NULL)
    - email (VARCHAR(255), NOT NULL, UNIQUE)
    - address (TEXT, NOT NULL)
2. Courseapp_course:
    - course_id (INTEGER, PRIMARY KEY)
    - cname (VARCHAR(100), NOT NULL)
    - fee (DECIMAL(10,2), NOT NULL)
    - duration (INTEGER, NOT NULL)
3. Courseapp_student_courses (Junction Table):
    - id (INTEGER, PRIMARY KEY, AUTOINCREMENT)
    - student_id (INTEGER, FOREIGN KEY to studentapp_student.roll, NOT NULL)
    - course_id (INTEGER, FOREIGN KEY to studentapp_course.course_id, NOT NULL)
    - **Constraints:** Composite unique index on (student_id, course_id)

Admin Registration

- Register both the model classes inside the admin.py file of **CourseApp**

```
# CourseApp/admin.py

from django.contrib import admin

from CourseApp.models import Student, Course


class StudentAdmin(admin.ModelAdmin):

    list_display = ["roll", "name", "email", "address"]


class CourseAdmin(admin.ModelAdmin):

    list_display = ["course_id", "cname", "fee", "duration"]


admin.site.register(Student, StudentAdmin)

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

Note:

- Inside the above **StudentAdmin** class in **list_display** we should not use **courses** directly,
- The **list_display** attribute expects fields that can be rendered as single values (e.g., strings, integers). A ManyToManyField like **courses** is a relationship field that points to multiple **Course** objects, so Django cannot automatically display it in the admin list view.

- When you try to include **"courses**" in **list_display**, Django raises an error because it doesn't know how to serialize or display the related objects in a list.
- To list all the enrolled courses we need to define **list_display** inside the **StudentAdmin** class as follows:

```
class StudentAdmin(admin.ModelAdmin):

    list_display = ["roll", "name", "email", "address",
"enrolled_courses"]

    def enrolled_courses(self, obj):

        course_names = []

        for course in obj.courses.all():

            course_names.append(course.cname)

        return ", ".join(course_names)
```

- Add a few Courses and Students details from the admin interface.

Shell Usage:

```
from CourseApp.models import Student, Course

# Create students

s1 = Student.objects.create(roll=101, name="Raj", email="raj@gmail.com",
address="Delhi")

s2 = Student.objects.create(roll=102, name="Simran",
email="simran@gmail.com", address="Mumbai")
```

```
# Create courses

c1 = Course.objects.create(course_id=1, cname="Python Programming",
fee=5000.00, duration="45 days")

c2 = Course.objects.create(course_id=2, cname="Django Framework",
fee=600.00, duration=16)



# Enroll students in courses

s1.courses.add(c1, c2)  # Raj takes Python and Django

s2.courses.add(c1)      # Simran takes Python



# Access related data

print(s1.courses.all())  # <QuerySet [<Course: Python Programming>,
<Course: Django Framework>]>

print(c1.student_set.all())  # <QuerySet [<Student: Raj>, <Student:
Simran>]>
```

Sample Operations:

1. **Add a Course:**

   ```
   s2.courses.add(c2)  # Simran takes Django

   print(s2.courses.all())  # <QuerySet [<Course: Python Programming>,
   <Course: Django Framework>]>
   ```

2. **Remove a Course:**

   ```
   s1.courses.remove(c1)  # Raj drops Python

   print(s1.courses.all())  # <QuerySet [<Course: Django Framework>]>
   ```

3. **Query Students by Course:**

   ```
   python_students = Student.objects.filter(courses__course_id=1)
   ```

```
print(python_students)  # <QuerySet [<Student: Raj>, <Student:
Simran>]>
```

4. **Query Courses by Student:**

```
raj_courses = Student.objects.get(roll=101).courses.all()

print(raj_courses)  # <QuerySet [<Course: Python Programming>,
<Course: Django Framework>]>
```