Object Relational Mapping (ORM) in Django

Object Relational Mapping (ORM) is a programming technique that allows developers to interact with a relational database using object-oriented concepts instead of writing raw SQL queries.

In Django, the ORM acts as a bridge between the relational database (like MySQL, PostgreSQL, SQLite) and the Python objects (models) in the application. It automatically converts Python class objects into database tables and vice versa.

In a traditional database system, developers use SQL (Structured Query Language) to perform operations such as SELECT, INSERT, UPDATE, and DELETE.

However, in Django ORM:

- Tables are represented as **Python classes** (called *models*).
- Table columns are represented as **class attributes**.
- Rows in the table are represented as **instances** (objects) of those classes.

Why We Need ORM

- ORM allows developers to perform database operations using Python code instead of SQL syntax, making it simpler and more readable.
- Django ORM supports multiple databases (SQLite, MySQL, PostgreSQL, etc.). The same Python code can work with any of them without modification.
- ORM automatically handles SQL injection prevention by safely constructing queries.
- Developers can focus on application logic rather than complex SQL queries.
- ORM ensures the data types and schema stay synchronized between code and database.

Don't waste time Let's go to Actual implementation of ORM.

```
Let's create a table.

CREATE TABLE users (

id INT AUTO_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

gender ENUM('Male', 'Female', 'Other'),

date_of_birth DATE,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);

Corresponding OPM:
```

Corresponding ORM:

```
from django.db import models
class User(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=100, null=False)
    email = models.EmailField(max_length=100, unique=True)
    gender = models.CharField(
        max_length=10,
        choices=[
            ('Male', 'Male'),
            ('Female', 'Female'),
            ('Other', 'Other'),
        ],
        null=True,
        blank=True
    date_of_birth = models.DateField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    def __str__(self):
       return self.name
```

Write given commands to migrate User table to Database

python manage.py makemigrations appname
python manage.py migrate



There are several ways to define primary key while creating table.

- 1) Default Primary Key (Automatic): No need to write any attribute for primary key. It will automatically generate and auto incremental.
- 2) Manual Primary Key Using *AutoField. Integer-based auto-increment primary key.*

student_id = models.AutoField(primary_key=True)

3) Primary Key Using IntegerField

roll_no = models.IntegerField(primary_key=True)

4) Primary Key Using CharField (Custom String Key)

username = models.CharField(max_length=50, primary_key=True)

SQL vs Django ORM Data Type Mapping:

SQL Data Type	ORM Field Type	Description
INT	models.IntegerField()	Stores whole numbers
		(positive or negative).
BIGINT	models.BigIntegerField()	For larger integer
CALATANA	11 6 11 7 110	values (64-bit).
SMALLINT	models.SmallIntegerField()	For smaller integer
VADCHAD(100)	1.1 Cl. E' 11/ 1 (1.100)	range values.
VARCHAR(100)	models.CharField(max_length=100)	Variable-length string
		up to defined characters. Must
		include max_length.
TEXT/LONGTEXT	models.TextField()	Stores long text (no
TEXT/EONGTEXT	models. Textr leid()	fixed limit).
ENUM	models.CharField(choices=[])	Restricts value to
('Male','Female','Other')	`	given choices, similar
		to ENUM.
DATE	models.DateField()	Stores date only (year,
		month, day).
DATETIME/TIMESTAMP	models.DateTimeField()	Stores both date and
		time.
		auto_now_add=True
		automatically sets the current time when
		created.
BOOLEAN	models.BooleanField()	Stores True or False.
	me ueis/2 e e i e uni 1918 ()	Often used for status
		flags. You can set
		default=True
DECIMAL(10,2)	models.DecimalField(max_digits=10,	
	decimal_places=2)	
FLOAT/DOUBLE	models.FloatField()	Stores approximate
		floating-point
CHAD(10)	11 01 5:11(1 4 10)	numbers.
CHAR(10)	models.CharField(max_length=10)	Fixed-length string
		(Django doesn't
		distinguish CHAR vs VARCHAR — both
		use CharField).
		use Charriela).

TIME	models.TimeField()	Stores time only (hour, minute, second).
YEAR	models.IntegerField()	No direct YEAR type; use IntegerField for year numbers.

SQL Constraints and Django ORM Equivalents

1. AUTO_INCREMENT

Automatically generates a unique number for each new row.

id = models.AutoField(primary key=True)

2. PRIMARY KEY

Uniquely identifies each row; cannot be NULL.

student_id = models.IntegerField(primary_key=True)

3. NOT NULL

Ensures a column always has a value.

name = models.CharField(max length=100, null=False)

4. UNIQUE

Ensures all values in a column are distinct.

email = models.EmailField(unique=True)

5. DEFAULT

Sets a default value if none is provided.

created_at = models.DateTimeField(auto_now_add=True) #
Default current timestamp
is active = models.BooleanField(default=True)

Working With Table

1. Insert Operation:

First Open Django Shell:

```
python manage.py shell
```

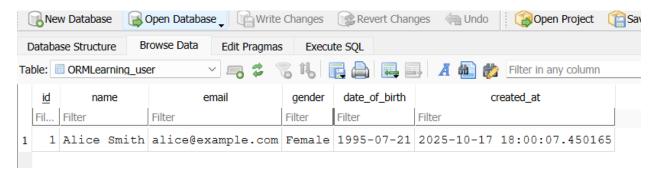
Then import the model

from ORMLearning.models import User

Insert a Single Row:

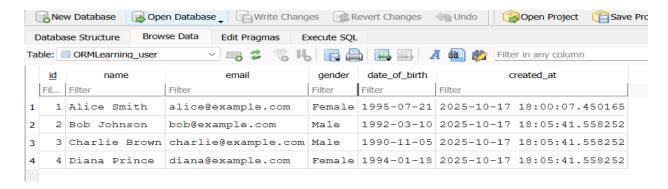
User.objects.all()

```
# Create a single user
user1 = User(
    name="Alice Smith",
    email="alice@example.com",
    gender="Female",
    date_of_birth="1995-07-21"
)
user1.save() # This inserts the row into the database
For checking
```



Insert Multiple Rows at a time:

```
# Create multiple user instances (not saved yet)
users = [
    User(name="Bob Johnson", email="bob@example.com",
gender="Male", date_of_birth="1992-03-10"),
    User(name="Charlie Brown",
email="charlie@example.com", gender="Male",
date_of_birth="1990-11-05"),
    User(name="Diana Prince",
email="diana@example.com", gender="Female",
date_of_birth="1994-01-18"),
]
# Bulk insert all at once
User.objects.bulk create(users)
```

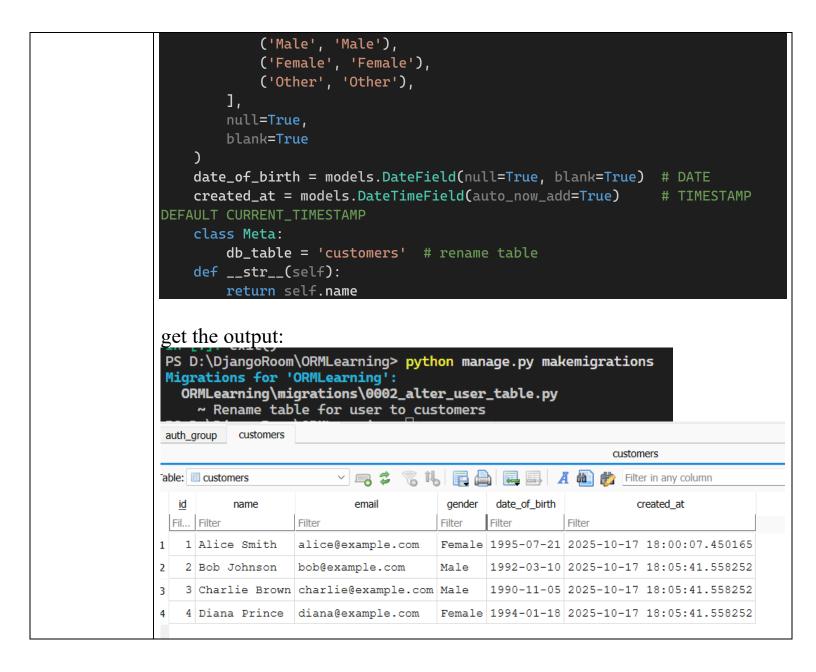


2. Select Operation:

SELECT * FROM	# Fetch all users	
users;	users = User.objects.all() #here all instance are	
	listed as object.	
SELECT name,	users = User.objects.values('name',	
email FROM users;	'email') "here a dictionary is generated like { 'name':	
	'Alice Smith', 'email':	
	'alice@example.com'}	
	that's why while printing the value we need to access attributes	
	like	
	\# To display them	
	for user in users:	
	<pre>print(user['name'], user['email'])</pre>	

Rename Table:

```
RENAME
              # Django ORM does not provide a direct method to rename tables.
              class User(models.Model):
TABLE
                  id = models.AutoField(primary_key=True)
users TO
                                                                          # INT
               AUTO_INCREMENT PRIMARY KEY
customers;
                  name = models.CharField(max_length=100, null=False)
                                                                          # VARCHAR(100)
                                                                          # UNIQUE and
                  email = models.EmailField(max_length=100, unique=True)
                  gender = models.CharField(
                                                                          # ENUM('Male',
                      max_length=10,
                      choices=[
```



Altering a Table:

In SQL:

You can use ALTER TABLE to modify an existing table.

ALTER TABLE users ADD COLUMN is active BOOLEAN DEFAULT TRUE;

IN ORM:

There has specific command like SQL. But you can rewrite schema in models.py

```
PS D:\DjangoRoom\ORMLearning> python manage.py makemigrations ORMLearning
Migrations for 'ORMLearning':
    ORMLearning\migrations\0004_user_is_active_alter_user_email.py
    + Add field is_active to user
    ~ Alter field email on user
PS D:\DjangoRoom\ORMLearning>
```

```
PS D:\DjangoRoom\ORMLearning> python manage.py migrate

Operations to perform:
Apply all migrations: ORMLearning, admin, auth, contenttypes, sessions

Running migrations:
Applying ORMLearning.0004_user_is_active_alter_user_email... OK
```

<u>id</u>	name	gender	date_of_birth	created_at	is_active	email
Fil	Filter	Filter	Filter	Filter	Filter	Filter
1	Alice Smith	Female	1995-07-21	2025-10-17 18:00:07.450165	1	alice@example.com
2	Bob Johnson	Male	1992-03-10	2025-10-17 18:05:41.558252	1	bob@example.com
3	Charlie Brown	Male	1990-11-05	2025-10-17 18:05:41.558252	1	charlie@example.com
4	Diana Prince	Female	1994-01-18	2025-10-17 18:05:41.558252	1	diana@example.com

Dropping a Column:

As like adding a column you can remove a column by just changing the schema in the models.py and run command of makemigrations and migrate.

SQL Operation	Django ORM Action	Command to Apply
ADD COLUMN	Add field in model	makemigrations + migrate
DROP COLUMN	Remove field from model	makemigrations + migrate
RENAME COLUMN	Rename field	makemigrations + migrate
MODIFY COLUMN	Change field type/size	makemigrations + migrate
RENAME TABLE	Change db_table in Meta	makemigrations + migrate

Querying Data in ORM

Filtering

```
SELECT * FROM users WHERE gender = 'Male';
Equivalent to
users = User.objects.filter(gender='Male')
```

SQL vs Django ORM (SELECT Queries)

- $filter() \rightarrow acts like WHERE$
- $exclude() \rightarrow acts like WHERE NOT$
- $_lt$, $_lte$, $_gt$, $_gte \rightarrow less/greater than lookups$
- $isnull=True/False \rightarrow check for NULL values$
- $_range \rightarrow for BETWEEN$
- $\underline{\quad}$ in \rightarrow for IN list
- __startswith, __endswith, __icontains → for LIKE queries
- order by() \rightarrow for ORDER BY
- [start:end] → for LIMIT and OFFSET

SQL	ORM
<pre>SELECT * FROM users WHERE gender != 'Female';</pre>	User.objects.exclude(gender='Female')
SELECT * FROM users WHERE date_of_birth < '1995-01-01';	User.objects.filter(date_of_birthlt='19 95-01-01')
SELECT * FROM users WHERE id > 10;	User.objects.filter(idgt=10)
SELECT * FROM users WHERE id >= 5;	User.objects.filter(idgte=5)
SELECT * FROM users WHERE id <= 20;	User.objects.filter(idlte=20)
SELECT * FROM users WHERE date_of_birth IS NULL;	<pre>User.objects.filter(date_of_birthisnull =True)</pre>
SELECT * FROM users WHERE date_of_birth IS NOT NULL;	<pre>User.objects.filter(date_of_birthisnull =False)</pre>
SELECT * FROM users WHERE date_of_birth BETWEEN '1990-01-01' AND '2000-12-31';	User.objects.filter(date_of_birthrange= ('1990-01-01', '2000-12-31'))
<pre>SELECT * FROM users WHERE gender IN ('Male', 'Other');</pre>	<pre>User.objects.filter(genderin=['Male', 'Other'])</pre>
SELECT * FROM users WHERE name LIKE 'A%'; Starts with A	User.objects.filter(namestartswith='A'
SELECT * FROM users WHERE name LIKE '%a'; Ends with a	User.objects.filter(nameendswith='a')
SELECT * FROM users WHERE name LIKE '%li%'; Contains 'li'	<pre>User.objects.filter(nameicontains='li')</pre>
select records where the name contains exactly 3 characters:	<pre>User.objects.filter(nameregex=r'^.{3}\$')</pre>

SELECT * FROM users WHERE name LIKE '';	
<pre>SELECT * FROM users WHERE gender = 'Female' AND date_of_birth > '1990- 01-01';</pre>	User.objects.filter(gender='Female', date_of_birthgt='1990-01-01')
<pre>SELECT * FROM users WHERE gender = 'Male' OR gender = 'Other';</pre>	`User.objects.filter(Q(gender='Male')
<pre>SELECT * FROM users ORDER BY date_of_birth ASC;</pre>	User.objects.order_by('date_of_birth')
SELECT * FROM users ORDER BY name DESC;	User.objects.order_by('-name')
SELECT * FROM users LIMIT 5;	User.objects.all()[:5]
SELECT * FROM users LIMIT 10 OFFSET 5;	User.objects.all()[5:15]
SELECT * FROM users LIMIT 5, 10; Get 10 rows starting from 6th	User.objects.all()[5:15] (same as above)
SELECT * FROM users ORDER BY created_at DESC LIMIT 10;	User.objects.order_by('-created_at')[:10]