

# 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 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
```

Table: ORMLearning_user						
<u>id</u>	name	email	gender	date_of_birth	created_at	

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
<b>INT</b>	models.IntegerField()	Stores whole numbers (positive or negative).
<b>BIGINT</b>	models.BigIntegerField()	For larger integer values (64-bit).
<b>SMALLINT</b>	models.SmallIntegerField()	For smaller integer range values.
<b>VARCHAR(100)</b>	models.CharField(max_length=100)	Variable-length string up to defined characters. Must include max_length.
<b>TEXT/LONGTEXT</b>	models.TextField()	Stores long text (no fixed limit).
<b>ENUM</b> (‘Male’, ‘Female’, ‘Other’)	models.CharField(choices=[...])	Restricts value to given choices, similar to ENUM.
<b>DATE</b>	models.DateField()	Stores date only (year, month, day).
<b>DATETIME/TIMESTAMP</b>	models.DateTimeField()	Stores both date and time. auto_now_add=True automatically sets the current time when created.
<b>BOOLEAN</b>	models.BooleanField()	Stores True or False. Often used for status flags. You can set default=True
<b>DECIMAL(10,2)</b>	models.DecimalField(max_digits=10, decimal_places=2)	
<b>FLOAT/DOUBLE</b>	models.FloatField()	Stores approximate floating-point numbers.
<b>CHAR(10)</b>	models.CharField(max_length=10)	Fixed-length string (Django doesn’t distinguish CHAR vs VARCHAR — both use CharField).

<b>TIME</b>	<code>models.TimeField()</code>	Stores time only (hour, minute, second).
<b>YEAR</b>	<code>models.IntegerField()</code>	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:

```
# 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

```
User.objects.all()
```

```
In [1]: from ORMLearning.models import User  
  
In [2]: user1 = User(  
...:     name="Alice Smith",  
...:     email="alice@example.com",  
...:     gender="Female",  
...:     date_of_birth="1995-07-21"  
...: )  
...: user1.save()  
...:  
  
In [3]: User.objects.all()  
Out[3]: <QuerySet [<User: Alice Smith>]>  
  
In [4]: for user in User.objects.all():  
...:     print(user.id, user.name, user.email, user.gender, user.date_of_birth)  
...:  
1 Alice Smith alice@example.com Female 1995-07-21
```

New Database

Open Database

Write Changes

Revert Changes

Undo

Open Project

Save Project

Database Structure

Browse Data

Edit Pragmas

Execute SQL

Table: 

ORMLearning\_user

## 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)
```

```
In [5]: # 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)
...:
Out[5]: [<User: Bob Johnson>, <User: Charlie Brown>, <User: Diana Prince>]

In [6]: for user in User.objects.all():
...:     print(user.id, user.name, user.email, user.gender, user.date_of_birth)
...:
1 Alice Smith alice@example.com Female 1995-07-21
2 Bob Johnson bob@example.com Male 1992-03-10
3 Charlie Brown charlie@example.com Male 1990-11-05
4 Diana Prince diana@example.com Female 1994-01-18
```

New Database		Open Database		Write Changes		Revert Changes		Undo		Open Project		Save Proc	
Database Structure		Browse Data		Edit Pragmas		Execute SQL							
Table: ORMLearning_user												Filter in any column	
	id	name	email	gender	date_of_birth	created_at							
	Filter	Filter	Filter	Filter	Filter	Filter							
1	1	Alice Smith	alice@example.com	Female	1995-07-21	2025-10-17 18:00:07.450165							
2	2	Bob Johnson	bob@example.com	Male	1992-03-10	2025-10-17 18:05:41.558252							
3	3	Charlie Brown	charlie@example.com	Male	1990-11-05	2025-10-17 18:05:41.558252							
4	4	Diana Prince	diana@example.com	Female	1994-01-18	2025-10-17 18:05:41.558252							

## 2.Select Operation:

SELECT * FROM users;	# Fetch all users users = User.objects.all() #here all instance are listed as object.
SELECT name, email FROM users;	users = User.objects.values('name', '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: print(user['name'], user['email'])

## Rename Table:

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



```

        ('Male', 'Male'),
        ('Female', 'Female'),
        ('Other', 'Other'),
    ],
    null=True,
    blank=True
)
date_of_birth = models.DateField(null=True, blank=True) # DATE
created_at = models.DateTimeField(auto_now_add=True) # TIMESTAMP
DEFAULT CURRENT_TIMESTAMP
class Meta:
    db_table = 'customers' # rename table
def __str__(self):
    return self.name

```

get the output:

```

PS D:\DjangoRoom\ORMLearning> python manage.py makemigrations
Migrations for 'ORMLearning':
  ORMLearning\migrations\0002_alter_user_table.py
  ~ Rename table for user to customers

```

auth\_group

customers

customers

table: 

customers

</

## 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

```
class User(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=100)
    email = models.EmailField(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)
    is_active = models.BooleanField(default=True) # new field added

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

```
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
```

id	name	gender	date_of_birth	created_at	is_active	email
Filter...	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	<code>makemigrations + migrate</code>
DROP COLUMN	Remove field from model	<code>makemigrations + migrate</code>
RENAME COLUMN	Rename field	<code>makemigrations + migrate</code>
MODIFY COLUMN	Change field type/size	<code>makemigrations + migrate</code>
RENAME TABLE	Change <code>db_table</code> in Meta	<code>makemigrations + migrate</code>

## Querying Data in ORM

### Filtering

```
SELECT * FROM users WHERE gender = 'Male';
```

Equivalent to

```
users = User.objects.filter(gender='Male')
```

```
In [1]: from ORMLearning.models import User

In [2]: users = User.objects.filter(gender='Male')

In [3]: for user in users:
...:     print(user.name, user.email, user.gender)
...:
Bob Johnson bob@example.com Male
Charlie Brown charlie@example.com Male
```

## SQL vs Django ORM (SELECT Queries)

- *filter()* → acts like WHERE
- *exclude()* → acts like WHERE NOT
- *\_\_lt*, *\_\_lte*, *\_\_gt*, *\_\_gte* → less/greater than lookups
- *\_\_isnull=True/False* → check for NULL values
- *\_\_range* → for BETWEEN
- *\_\_in* → for IN list
- *\_\_startswith*, *\_\_endswith*, *\_\_icontains* → for LIKE queries
- *order\_by()* → for ORDER BY
- *[start:end]* → for LIMIT and OFFSET

SQL	ORM
<code>SELECT * FROM users WHERE gender != 'Female';</code>	<code>User.objects.exclude(gender='Female')</code>
<code>SELECT * FROM users WHERE date_of_birth &lt; '1995-01-01';</code>	<code>User.objects.filter(date_of_birth__lt='1995-01-01')</code>
<code>SELECT * FROM users WHERE id &gt; 10;</code>	<code>User.objects.filter(id__gt=10)</code>
<code>SELECT * FROM users WHERE id &gt;= 5;</code>	<code>User.objects.filter(id__gte=5)</code>
<code>SELECT * FROM users WHERE id &lt;= 20;</code>	<code>User.objects.filter(id__lte=20)</code>
<code>SELECT * FROM users WHERE date_of_birth IS NULL;</code>	<code>User.objects.filter(date_of_birth__isnull=True)</code>
<code>SELECT * FROM users WHERE date_of_birth IS NOT NULL;</code>	<code>User.objects.filter(date_of_birth__isnull=False)</code>
<code>SELECT * FROM users WHERE date_of_birth BETWEEN '1990-01-01' AND '2000-12-31';</code>	<code>User.objects.filter(date_of_birth__range=('1990-01-01', '2000-12-31'))</code>
<code>SELECT * FROM users WHERE gender IN ('Male', 'Other');</code>	<code>User.objects.filter(gender__in=['Male', 'Other'])</code>
<code>SELECT * FROM users WHERE name LIKE 'A%'; -- Starts with A</code>	<code>User.objects.filter(name__startswith='A')</code>
<code>SELECT * FROM users WHERE name LIKE '%a'; -- Ends with a</code>	<code>User.objects.filter(name__endswith='a')</code>
<code>SELECT * FROM users WHERE name LIKE '%li%'; -- Contains 'li'</code>	<code>User.objects.filter(name__icontains='li')</code>
<code>select records where the name contains exactly 3 characters:</code>	<code>User.objects.filter(name__regex=r'^.{3}\$')</code>

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