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
);
```

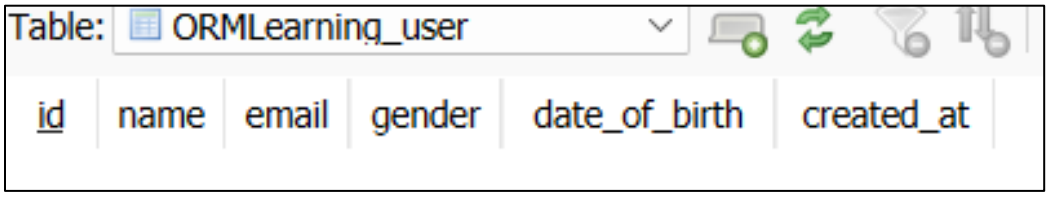
Corresponding ORM:

```python
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 values (64-bit).
SMALLINT	models.SmallIntegerField()	For smaller integer 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 fixed limit).
ENUM ('Male','Female','Other')	models.CharField(choices=[...])	Restricts value to 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. 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 numbers.
CHAR(10)	models.CharField(max_length=10)	Fixed-length string (Django doesn't distinguish CHAR vs VARCHAR — both use CharField).

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
```

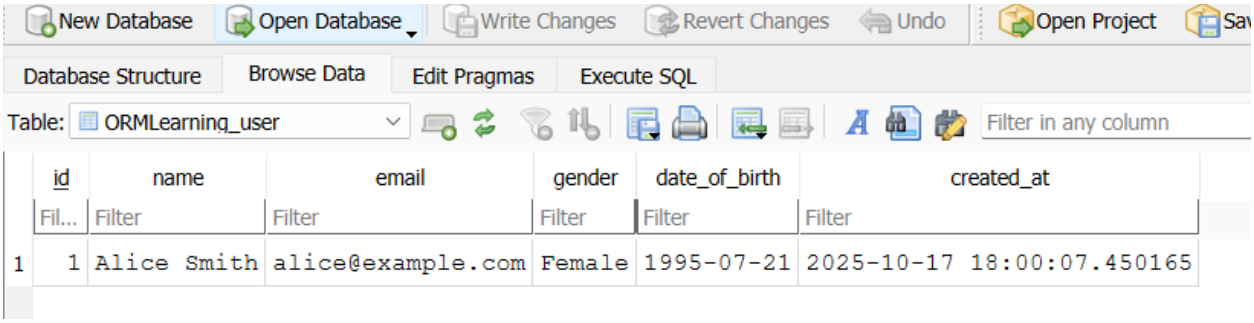
Insert a Single Row:

```
# 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()
```

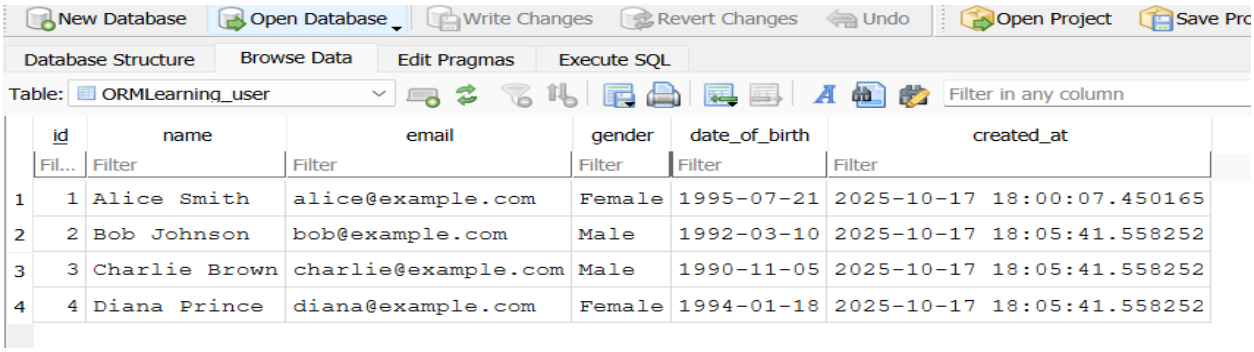
```
In [1]: from ORMLearning.models import User

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
```



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
```



	id	name	email	gender	date_of_birth	created_at
	Fil...	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.
	`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(  # ENUM('Male', 'Female', 'Other')`
	`  max_length=10,`
	`  choices=[`

```
            ('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

able: customers — Filter in any column

	id	name	email	gender	date_of_birth	created_at
	Fil...	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

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

```python
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
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')
```

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

`SELECT * FROM users where name LIKE '___'` `SELECT * FROM users WHERE name LIKE '___';`	
`SELECT * FROM users WHERE gender = 'Female' AND date_of_birth > '1990-01-01';`	`User.objects.filter(gender='Female', date_of_birth__gt='1990-01-01')`
`SELECT * FROM users WHERE gender = 'Male' OR gender = 'Other';`	`` `User.objects.filter(Q(gender='Male')``
`SELECT * FROM users ORDER BY date_of_birth ASC;`	`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]`

Difference Between values() and values_list() in Django ORM

Selection Case	Query Using .values()	Output of .values()	Query Using .values_list()	Output of .values_list()
Select One Column	User.objects.values('name')	[{'name': 'Rahim'}, {'name': 'Karim'}]	User.objects.values_list('name', flat=True)	['Rahim', 'Karim']
Select Two Columns	User.objects.values('name', 'gender')	[{'name': 'Rahim', 'gender': 'Male'}, {'name': 'Mala', 'gender': 'Female'}]	User.objects.values_list('name', 'gender')	[('Rahim', 'Male'), ('Mala', 'Female')]

Select All Columns	User.objects.values()	[{'id':1, 'name':'Rahim', 'gender':'Male', ...}, {...}]	User.objects.values_list() (not recommended unless field order known)	[(1, 'Rahim', 'Male', ...), (...)]

When to use?

Use .values() when:
- You want readable output with field names.
- You are displaying data in templates or debugging.
- Example usage: Rendering dictionary data in HTML templates.

Use .values_list() when:
- You want performance optimization.
- Field names are not required.
- You are working with pure lists or tuples (e.g., exporting data, forming choices in forms).
- Use flat=True only when selecting exactly one field.

Use of Aggregate Function:

SQL	ORM
SELECT COUNT(*) FROM users;	total_users = User.objects.count()
SELECT COUNT(*) FROM users WHERE gender = 'Female';	female_count = User.objects.filter(gender='Female').count()
SELECT MIN(salary) as min_salary FROM users;	result = User.objects.aggregate(min_salary=Min('salary')) value = result['min_salary']
SELECT MAX(salary) as max_salary FROM users;	result = User.objects.aggregate(max_salary=Max('salary')) value=result[‘max_salary’]

SELECT MIN(salary), MAX(salary) FROM users;	result = User.objects.aggregate(Min('salary'), Max('salary')) print(result) Output: {'salary__min': 25000, 'salary__max': 98000}
SELECT AVG(salary) AS avg_salary FROM users;	result = User.objects.aggregate(avg_salary=Avg('salary'))
SELECT SUM(salary) AS total_salary FROM users;	result = User.objects.aggregate(total_salary=Sum('salary'))
SELECT gender, AVG(salary) AS avg_salary FROM users GROUP BY gender;	User.objects.values('gender').annotate(avg_salary=Avg('salary'))
SELECT name, LENGTH(name) AS name_length FROM users;	users = User.objects.annotate(name_length=Length('name'))
SELECT name, LOWER(name) AS lowercase_name FROM users;	users = User.objects.annotate(lowercase_name=Lower('name'))
SELECT name, UPPER(name) AS uppercase_name FROM users;	users = User.objects.annotate(uppercase_name=Upper('name'))
SELECT CONCAT(name, ' <', email, '>') AS user_contact FROM users;	users = User.objects.annotate(user_contact=Concat('name', Value(' <'), 'email', Value('>')))

SELECT
CONCAT(name, ' earns ', CAST(salary AS CHAR), ' BDT') AS info
FROM users;
	users = User.objects.annotate(
	info=Concat(
	'name',
	Value(' earns '),
	Cast('salary', output_field=CharField()), # convert number to text
	Value(' BDT')
	)
	)