

Object Relationship Mapping - ORM

What is ORM

Usually, the type of system used in an Object-Oriented language such as Python contains types that are non-scalar and every time, they cannot be represented only in primitive types such as integers and strings. On the other hand, data types in tables of a relational database are of primary types, that is integer and string.

Therefore, the program needs to convert objects in scalar data to interact with a backend database.

Object-oriented programming languages use Object Relation Mapping (ORM) to interact with Structured Query Language (SQL), both of which are incompatible.

The ORM layer maps a class to a table in a relational database, with its attributes matching the table's field structure.

The ORM library lets you perform the database operations in an object-oriented way instead of executing raw SQL queries.

This allows you to focus more on the programming logic than the backend database operations.

Object Relational Mapping or ORM is the ability to create a SQL query using object-oriented programming language such as Python. This enables a quick turnaround time in fast production environments that need constant updates.

Each model is a Python class that subclasses **django.db.models.Model**

Each attribute of the model represents a database field. Essentially you can think of a model as a Python object, and models help developers create, read, update and delete objects, commonly called the CRUD operations.

Django has its own ORM layer. Its migration mechanism propagates the models in database tables.

You need to construct a **QuerySet** via a Manager of your model class to retrieve objects from your database.

QuerySet

Let's try to understand how **QuerySet** is constructed and handled with an example. To begin with, add the following models in the Django app named **demoapp**.

```
1  from django.db import models
2  class Customer(models.Model):
3      name = models.CharField(max_length=255)
4
5  class Vehicle(models.Model):
6      name = models.CharField(max_length=255)
7      customer = models.ForeignKey(
8          Customer,
9          on_delete=models.CASCADE,
10         related_name='Vehicle'
11     )
```

The idea is to create two tables, Customer and Vehicle, with a one-to-many relationship. Apply the migrations and you will see these two tables in the project's database.

Next, open the Interactive shell.

It is important to know that these commands are the same if you are on Windows or Mac.

(djenv) C:\djenv\demoproject>python manage.py shell

Create an object of the Customer class. Give a string parameter to the constructor. The **save()** method of the **models.Model** class creates a row in the Customer table.

```
1  >>> from demoapp.models import Customer
2  >>> c=Customer(name="Henry")
3  >>> c.save()
```

Adding a model object

The **Customer.objects** gives the Manager of the model. It handles all the CRUD operations on the database table. For example, an object can also be created with the **create()** method as follows:

```
1  >>> Customer.objects.create(name="Hameed")
2  <Customer: Customer object (2)>
```

```
2 <Customer: Customer object (2)>
```

The **create()** method actually performs the **INSERT** operation of SQL.

Now, let us add two objects to the Vehicle model. Note that one of the fields in the Vehicle model refers to an object of the Customer table as **ForeignKey**. Fetch the Customer object with a primary key = 2.

```
1 >>> from demoapp.models import Customer, Vehicle
2 >>> c=Customer.objects.get(pk=2)
3 >>> c.name
4 'Hameed'
```

This object is used as the value of the customer attribute in the Vehicle object.

```
1 >>> v=Vehicle(name="Honda", customer=c)
2 >>> v.save()
3 >>> v=Vehicle(name="Toyota", customer=c)
4 >>> v.save()
```

Similarly, add two vehicles for Customer "Henry".

```
1 >>> c=Customer.objects.get(name="Henry")
2 >>> Vehicle.objects.create(name="Ford", customer=c)
3 <Vehicle: Vehicle object (3)>
4 >>> Vehicle.objects.create(name="Nissan", customer=c)
5 <Vehicle: Vehicle object (4)>
```

Fetch model objects

A **QuerySet** represents a collection of objects from your database. It represents a **SELECT** query. To fetch all the objects, use the **all()** method of the Manager.

```
1 model.objects.all()
```

The **all()** method returns a list of objects. You can iterate over it with the usual for loop or list comprehension technique.

```
1 >> lst=Customer.objects.all()
2 >>> [c.name for c in lst]
3 ['Henry', 'Hameed']
```

You can apply filters to the data fetched from the model. This is used to fetch objects satisfying the given criteria. In SQL terms, a **QuerySet** equates to a **SELECT** statement, it is like applying a **WHERE** clause.

```
1 model.objects.filter(criteria)
```

For example, use the following statement to retrieve all the customers with names starting with 'H'.

```
1 mydata = Customer.objects.filter(name__startswith='H')
```

Similarly, you can retrieve the objects of the Vehicle model. Remember that the Vehicle object refers to a customer object. You can retrieve the attributes of the related customer as follows:

```
1 >>> lst=Vehicle.objects.all()
2 >>> for v in lst:
3     print (v.name, " : ", v.customer.name)
4 ...
5 Honda : Hameed
6 Toyota : Hameed
7 Ford : Henry
8 Nissan : Henry
```

Updating and removing a model object

To update an object, such as changing the Customer's name from Henry to Helen, assign a new value to the name attribute and save.




```
1 >>> c=Customer.objects.get(name="Henry")
2 >>> c.name="Helen"
3 >>> c.save()
```

Similarly, the **delete()** method of the model manager physically removes the corresponding row in the model's mapped table.

```
1 >>> c=Customer.objects.get(pk=4)
2 >>> c.delete()
3 (1, {'demoapp.Customer': 1})
```

This way, you can perform the CRUD operations on a database table using the **QuerySet** API instead of executing raw SQL queries.

Mark as completed

 Like  Dislike  Report an issue