Django ORM Concept

- The Django web framework includes a default "object-relational mapper" layer (ORM) that can be used to interact with application data from various relational databases such as SQLite, PostgreSQL and MySQL.
- The Django ORM is an implementation of the object-relational mapping (ORM) concept.
- ORM converting data between incompatible type systems using object oriented programming languages.

converting interacting
Application ORM ----------->> SQL Code ---------->> Database

Structured Query Language(SQL):-

- By using this SQL, we can access and manipulate databases.
- SQL is a standard language for dealing with Relational Databases like MySQL DB, Oracle DB, Postgree SQL DB, etc....

Why ORM?

- By using ORM, developers can write python code insted of SQL to get,create,update and delete data on databases.

What is a QuerySet?

- A QuerySet is a list of objects of a given Model. QuerySets allow you to read the data from the database, filter it and order it.

Database Access through Managers:

- According to the Django documentation - Django by default adds a manager called "objects" to every model class defined inside models.py file.
- This particular manager (i.e objects) helps us to interact with the database in complicated ways.
- The "objects" manager is the most common way Django developers interact with the database.
- To access objects, we need to use manager type model class name followed by the (.) dot operator then the "objects" manager.

For example:

```
>>> from ormapp.models import Employee
>>> Employee.objects
<django.db.models.manager.Manager object at 0x00000000042CE978>
>>> type(Employee.objects)
<class 'django.db.models.manager.Manager'>
```

As you can see objects is just a instance of django.db.models.manager.Manager class.

create() method

- The create() method allows us to create and commit object to the database in one go, instead of separately calling the save() method. For example:
- >>> Employee.objects.create(f1=v1,f2=v2,f3=v3)

To display SQL Query code

>>> **print(Employee.objects.filter(ename='Srinivas').query)**
SELECT `Serialization_App_employee`.`id`, `Serialization_App_employee`.`eno`, `Serialization_App_employee`.`ename`, `Serialization_App_employee`.`address`, `Serialization_App_employee`.`salary` FROM `Serialization_App_employee` WHERE `Serialization_App_employee`.`ename` = Srinivas

Create data into database:

- create data using model class object like bellow

```
emp = Employee(f1=.., f2=...)
emp.save()
>>> emp = Employee(first_name="Srinivas", last_name="Python",salary=20000)
>>> emp.save()
```

create() method

- The create() method allows us to create and commit object to the database in one go, instead of separately calling the save() method.
  For example:
  Sysntax : MonelName.objects.create(field1=value1,field2=value2)

```
>>> Employee.objects.create(first_name="Srinivas",last_name="Python",salary=10000)
```

bulk_create() method

- The bulk_create() method allows us to create and commit multiple objects in a database in one step. It accepts a list of objects.
  For example:

```
>>> Employee.objects.bulk_create(
    [
      Employee(first_name="Srinivas",last_name="Python",salary=10000),
      Employee(first_name="Srinivas",last_name="Python",salary=10000),
      Employee(first_name="Srinivas",last_name="Python",salary=10000),
    ]
)
```

- so here all 3 objects created at a time into a database.

To Get Data from database:

all() method

- The all() method fetches all the records from the table.
  For example:

```
>>> Employee.objects.all()
```

- The above command fetches all the records from the Employee's table.
- The all() method returns a QuerySet object. A QuerySet object looks like a list, but it is not an actual list, in some ways it behaves just like lists.
  For example, you can access individual members in a QuerySet objects using an index number.
- **Note: It is important to note that some methods of objects manager returns QuerySet while some do not.**
- QuerySet is iterable just like a list. You can use a for loop to iterate through all of the objects in a QuerySet object.

indexing:

```
>>> e = Employee.objects.all() #  6 records available
>>> e[0] ---->> returns 0 index related object means first object.
>>> Employee.objects.all()[0]
```

- Above one also returns 0 index related object means first object.

slicing:

- if you want to access the group of records out of all records then use slicing concept.

```
>>> Employee.objects.all()[0:3]
```

- returns first 3 objects only from 0 index onwords.

count() method

- The count() method returns the total number of records in a database table.

```
>>> Employee.objects.count()   # 6 available
```

get():

- get() method can return or access the specific record from table based on condition matching.

  for example,

```
>>> Employee.objects.get(id=1)
<Employee: pythonsrinivas>
```

- get() method returns exception if the condition is not matching.

```
>>> Employee.objects.get(id=10)
```

ORM_App.models.Employee.DoesNotExist: Employee matching query does not exist.

- get() method also returns exception if condition is matching more than one time.

```
>>> Employee.objects.get(location="Mumbai") #  two times matching query.
```

ORM_App.models.Employee.MultipleObjectsReturned: get() returned more than one Employee -- it returned 2!

Filtering records using the filter() method

- Most of the time you would only want to work with a subset of data.
- Django provides a filter() method which returns a subset of data.
- It accepts field names as keyword arguments and returns a QuerySet object.

```
>>> Employee.objects.filter(first_name="Srinivas")
```

- **Here, Employee.objects.filter(first_name='Srinivas') ORM Query Translates to SQL Query something like this: SELECT * from blog_employee where first_name = 'Srinivas'**

Django Field Lookups

- In addition to passing field names as keyword arguments.
- You can also use something called lookups. Managers and QuerySet objects come with a feature called lookups.
- **A lookup is composed of a model field followed by two underscores (__) which are then followed by lookup name. Let's take some examples.**
- **Syntax : ModelName.objects.filter(ModelFieldName__LookupName=LookupValue]**

Examples

__contains lookup finds all the records where first_name field contains the word "Virat".

__startswith lookup finds all the records whose first_name field start with "Virat".

__endswith lookup finds all the records whose first_name field ends with "Virat".

- Both __startswith and __endswith are case-sensitive. Their is case-insensitive equivalents are **__istartswith** and **__iendswith.**

Working with __gt , __gte , __lt , __lte

__gt lookup finds all the records whose id or primary key (pk) is greater than 3.

__gte lookup finds all the records whose id or primary key (pk) is greater than or equal to 3.

__lt lookups find all the records whose primary key is less than 3.

__lte lookups find all the records whose primary key is less than or equal to 3.

__in lookups find all the records whose primary key is belongs to given values if available. If not available then returns empty list object.

__range lookups find all the records whose primary key is available in given range.

get()

- Notice the difference between the output of get() and filter() method. For the same parameter they both will give two different results.
- The get() method returns a instance of Employee while filter() methods returns a QuerySet object.
- The get() method accepts same parameters as filter() method but it returns only a single object.
- If it finds multiple objects it raises a MultipleObjectsReturned exception.
- If it doesn't find any object it raises DoesNotExist exception.

```
>>> Employee.objects.get(ename="Srinivas")
<Employee: Srinivas>
>>> Employee.objects.filter(ename="Srinivas")
<QuerySet [<Employee: Srinivas>]>
```

Notice the difference between the output of get() and filter() method. For the same parameter they both returns two different results. The get() method returns a instance of Author while filter() methods returns a QuerySet object.

Ordering Results

- To order result we use order_by() method, just like filter() it also returns a QuerySet object. It accepts field names that you want to sort by as positional arguments.

```
>>> Author.objects.order_by("id")
```

sort by multiple fields:

- You can also sort the result by multiple fields like this.

```
>>> Author.objects.filter(id__gt=3).order_by("name", "-email")
<QuerySet [<Author: droopy : droopy@mail.com>, <Author: spike : spike@mail.com>,
 <Author: tyke : tyke@mail.com>]>
```

This code will sort the result first by name in ascending and then by email in descending order.

Selecting the fields

```
>>> Author.objects.filter(name__contains='foo').order_by("name")
```

It returns data from all the fields (columns).

values_list()

- What if we want data only from one or two fields ? The objects manager provides a
- values_list() method specially for this job.
- The values_list() accepts optional one or more field names from which we want the data and returns a QuerySet.
- values_list() method returns data in the form of QuerySet tuples and every tuple only contains values list only.

For example:

```
>>> Author.objects.values_list("id", "name")
```

<QuerySet [(1, 'tom'), (2, 'jerry'), (3, 'spike'), (4, 'tyke'), (5, 'droopy')]>
>>> Employee.objects.values_list('id','ename')
<QuerySet [(1, 'Srinivas'), (2, 'Shivam'), (3, 'Shivam 123')]>
>>> Employee.objects.values('id','ename')
<QuerySet [{'id': 1, 'ename': 'Srinivas'}, {'id': 2, 'ename': 'Shivam'}]>

values()

- The objects manager also provides an identical method called values() which works exactly like values_list() but it returns a QuerySet where each element is a dictionary instead of a tuple.
- values() returns data in the form of QuerySet dictionary objects format and every dictionary contains key: value format data.

  >>> Employee.objects.values()
  <QuerySet [{'id': 1, 'eno': 10, 'ename': 'Srinivas', 'address': 'Hyderabad', 'salary': 10000}, {'id': 2, 'eno': 20, 'ename': 'Shivam', 'address': 'Hyd', 'salary': 20000}]>

Slicing Results

>>> # returns the first three objects after sorting the result
>>> Author.objects.order_by("-id")[:3]
>>> Author.objects.order_by("-id")[-1]
AssertionError: Negative indexing is not supported.

Updating Multiple Objects

- The objects manager provides a method called update() to update one or multiple records in one step.
- Just like filter() method it accepts one or more keyword arguments.
- If the update was successful it returns the number of rows updated.
- >>> Author.objects.filter(pk=2).update(email='tom@yahoo.com') # 1
- This code will update the email of author whose pk is equal to 2.

Updating all objects

>>> Author.objects.all().update(active=True) # 5
The above code updates the value of active field to True for all the records in the Author's table.
The code is equivalent to the following:
>>> Author.objects.update(active=True)

Deleting records

- The delete() method is used to delete one or more objects. For example:

Deleting a single object.

>>> a = Author.objects.get(pk=2)
>>> a
<Author: tom : tom@mail.com>
>>> a.delete()
(1, {'blog.Author': 1})

Deleting multiple records.

>>> r = Author.objects.all().delete()
>>> r
(4, {'blog.Author': 4})

Aggregation functions

- Grouping of records in Django ORM can be done using aggregation functions like Max, Min, Avg, Sum, Count, Q, etc.
- Django ORM queries help to create, retrieve, update and delete objects.
- But sometimes we need to get aggregated values from the objects.
- We can get them by example shown below
  >>> from django.db.models import Avg, Max, Min, Sum, Count
- To return total average salary of all employees then we need to use Avg()
  >>> Employee.objects.all().aggregate(Avg('salary'))
  {'salary__avg':20000}
- To return total sum of all employees salaries then we need to use Sum()
  >>> Employee.objects.all().aggregate(Sum('salary'))
  {'salary__sum':120000}
- To return maximum salary of all employees then we need to use Max()
  >>> Employee.objects.all().aggregate(Max('salary'))
- To return minimum salary of all employees then we need to use Min()
  >>> Employee.objects.all().aggregate(Min('salary'))

ORM Queries

- create bulk records

```
>>> Employee.objects.bulk_create(
    [
      Employee(eno=25,ename='Rohit Sharm', address='Bangalore',salary=30000),
      Employee(eno=55,ename='Srini', address='Chennai', salary=35000)
    ]
)
[<Employee: Rohit Sharm>, <Employee: Srini>]
```

- Employee.objects.filter(ename__startswith='Shivam')
  <QuerySet [<Employee: Shivam>, <Employee: Shivam 123>]>
- Employee.objects.filter(ename__endswith='123')
- Employee.objects.filter(id__in=[4,5])
- Employee.objects.filter(id__gte=4).count()
- Employee.objects.filter(salary__in=[10000,60000]).aggregate(total_avg=Avg('salary'))
- Employee.objects.all().aggregate(total_sum=Sum('salary'))
  {'total_sum': 610070}
- Employee.objects.aggregate(total_avg=Avg('salary'))
  {'total_avg': 76258.75}

or , and , negation(~) operaters

- Employee.objects.filter(Q(salary__lt=20000) | Q(address__endswith='Mumbai'))
  <QuerySet [<Employee: Srinivas>, <Employee: Shivam 123>, <Employee: Virat 123>, <Employee: Krishna Shivam>, <Employee: Virat Kohli>]>
- Employee.objects.filter(Q(salary__lt=20000) & Q(address__endswith='Mumbai'))
  <QuerySet [<Employee: Krishna Shivam>]>
- Employee.objects.filter(Q(salary__lt=20000)).order_by('salary')

<QuerySet [<Employee: Krishna Shivam>, <Employee: Virat 123>, <Employee: Srinivas>, <Employee: Shivam 123>]>

- Employee.objects.filter(Q(ename__startswith='S')|Q(address__endswith='Pune'))

NOTE: ~Q(condition)

- excluding given condition it returns because of ~ operator.
- Employee.objects.all().order_by('salary')
  <QuerySet [<Employee: Krishna Shivam>, <Employee: Virat 123>, <Employee: Srinivas>, <Employee: Shivam 123>, <Employee: Shivam>, <Employee: Virat Varam>, <Employee: Virat Kohli>, <Employee: Rohit>]>
- Employee.objects.filter(Q(salary__range=[9000,40000])).order_by('address')
- Employee.objects.filter(salary__range=[9000,40000]).order_by('address')[:2]
  <QuerySet [<Employee: Shivam>, <Employee: Srinivas>, <Employee: Shivam 123>]>
- Employee.objects.values().order_by('salary')
  <QuerySet [{'id': 5, 'eno': 40, 'ename': 'Krishna Shivam', 'address': 'New Mumbai', 'salary': 70}, {'id': 4, 'eno': 25, 'ename': 'Virat 123', 'address': 'mumbai', 'salary': 5000}, {'id': 1, 'eno': 10, 'ename': 'Srinivas', 'address': 'Hyderabad', 'salary': 10000}, {'id': 3, 'eno': 30, 'ename': 'Shivam 123', 'address': 'Pune', 'salary': 15000}, {'id': 2, 'eno': 20, 'ename': 'Shivam', 'address': 'Hyd', 'salary': 20000}, {'id': 7, 'eno': 50, 'ename': 'Virat Varam', 'address': 'Old Pune', 'salary': 50000}, {'id': 6, 'eno': 40, 'ename': 'Virat Kohli', 'address': 'Old Mumbai', 'salary': 60000}, {'id': 8, 'eno': 35, 'ename': 'Rohit', 'address': 'Bangalore', 'salary': 450000}]>
- Employee.objects.values('ename').filter(ename__in=['Shivam','Rohit'])
  <QuerySet [{'ename': 'Shivam'}, {'ename': 'Rohit'}, {'ename': 'Shivam'}]>

ORM Queries Revision practices

To Create data into Database

```
e = Employee(eno=..,ename=...)
e.save()

Employee.objects.create(f1=v1, f2=v2,…)

Employee.objects.bulk_create(
   [
         Employee(eno=..,ename=...),
         Employee(eno=..,ename=...),
         Employee(eno=..,ename=...),
         Employee(eno=..,ename=...),
   ]
)
```

To Get data from Database

```
Employee.objects.all()
Employee.objects.get(condition)
Employee.objects.filter(condition)
Employee.objects.exclude(condition)
Employee.objects.values_list()  -------->>  <QuerySet [ ( v1 , v2 , v3 ) , (…) , (…),…] >
```

Employee.objects.values() ---------->> <QuerySet [{ k1 : v1 , k2 : v2 } , {..} , {....}] >
Employee.objects.all().count()
Employee.objects.all().order_by('salary') ----> Asending order
Employee.objects.all().order_by('-salary') ----> Desending order

Aggregation functions

from django.db.models import Sum, Avg, Max, Min, Count, Q
Employee.objects.aggregate(Sum('salary'))
Employee.objects.aggregate(Avg('salary'))
Employee.objects.aggregate(Max('salary'))
Employee.objects.aggregate(Min('salary'))

Lookups Names

Sysntax: ModelName.objects.filter(fieldName__lookupName='looksup_value')

Employee.objects.filter(first_name__exact='Ravi')
Employee.objects.filter(first_name__contains='Ravi')
Employee.objects.filter(id__exact=2) --->> 1,2,3,4,5,6,7,8,
Employee.objects.filter(id__in=[2,6]) --->> 2 and 6 if available
Employee.objects.filter(id__range=[2,6]) --->> from 2 to 6 returns
Employee.objects.filter(id__gt=3) --->> 4,5,6....
Employee.objects.filter(id__gte=3) --->> 3,4,5...
Employee.objects.filter(id__lt=3) ---->> 2,1,..
Employee.objects.filter(id__lte=3) ---->> 3,2,1...

Employee.objects.filter(location__startswith='Ne')
Mumbai, Navi Mumbai, New Mumbai, nellore ----->> **New Mumbai**

Employee.objects.filter(location__istartswith='Ne')
Mumbai, Navi Mumbai, New Mumbai, nellore ---->> New Mumbai, nellore

Employee.objects.filter(location__endswith='Mumbai')
Mumbai, Navi Mumbai, New Mumbai, nellore ---->> Mumbai, Navi Mumbai, New Mumbai

Q. How we want to count how many Employee objects per distinct ename exist in our Employee table:

>>> Employee.objects.values('ename').order_by('ename').annotate(Count('ename'))
<QuerySet [{'ename': 'John', 'ename__count': 1}, {'ename': 'Kiran Kumar', 'ename__count': 1}, {'ename': 'Ravi', 'ename__count': 1}, {'ename': 'Rohit', 'ename__count': 2}, {'ename': 'Surya Kumar', 'ename__count': 1}, {'ename': 'Virat', 'ename__count': 1}, {'ename': 'Virat Kohli', 'ename__count': 1}]>

Q. Get SQL for Django queryset?

- The query attribute on queryset gives you SQL equivalent syntax for your query.
- >>> queryset = MyModel.objects.all()
- >>> print(queryset.query)
  SELECT "myapp_mymodel"."id", ... FROM "myapp_mymodel"

Lower(), Upper()

```
>>> from django.db.models.functions import Lower,Upper
>>> Employee.objects.values_list('id',Upper('ename'))
```

Questions

Q. How to Select and Output individual fields from Model tables?
Q. How to perform an OR query on a database - including Q objects?
Q. How to perform an AND query on a database - including Q objects?
Q. How to perform a NOT query on a database - including Q Objects?

SQL NOT Examples:

```
mysql> select * from orm_app_employee where not id=2;
```

ORM NOT Example:

--->> exclude(<condition>)
--->> filter(~Q(<condition>))

Q. How to use Q objects for complex queries?
Q. How to group records in Django ORM?
Q. How to create multiple objects in one shot?
Q. How to order a queryset in ascending or descending order?
Q. How to order a queryset in case insensitive manner?
Q. How to Performing raw SQL queries without the ORM?

```
from django.db import connection
cursor = connection.cursor()
cursor.execute(SQL)
cursor.fetchone() or cursor.fetchall()
```

Q. How to use union() function on queries ?
Q. How to find second highest salary using Django ORM ?

- Employee.objects.values().order_by('-salary')[1]["ename"] + " Getting Highest salary"

Q. Find rows which have duplicate field values

- Employee.objects.values('ename').annotate(name_count=Count('ename')).filter(name_count__gt=1)
  <QuerySet [{'ename': 'Rohit', 'name_count': 2}]>