Chapter-12

Working with Advanced Model Concepts

Model Inheritance

It is very useful and powerful feature of django.

There are 4 types of Model Inheritance.

1) **Abstract Base Class Model Inheritance**
2) **Multi table Inheritance**
3) **Proxy Model Inheritance**

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

4) Multiple Inheritance

1) Abstract Base Class Model Inheritance

- If several Model classes having common fields, then it is not recommended to write these fields seperately in every Model class. It increases length of the code and reduces readability.
- We can seperate these common fields into another Model class,which is also known as Base Class. If we extend Base class automatically common fields will be inherited to the child classes.

Without Inheritance

```
1)  class Student(models.Model):
2)      name=models.CharField(max_length=64)
3)      email=models.EmailField()
4)      address=models.CharField(max_length=256)
5)      rollno=models.IntegerField()
6)      marks=models.IntegerField()
7)
8)  class Teacher(models.Model):
9)      name=models.CharField(max_length=64)
10)     email=models.EmailField()
11)     address=models.CharField(max_length=256)
12)     subject=models.CharField(max_length=64)
13)     salary=models.FloatField()
```

With Inheritance

```
1)  class ContactInfo(models.Model):
2)      name=models.CharField(max_length=64)
3)      email=models.EmailField()
4)      address=models.CharField(max_length=256)
5)      class Meta:
6)          abstract=True
7)
8)  class Student(ContactInfo):
9)      rollno=models.IntegerField()
10)     marks=models.IntegerField()
11)
12) class Teacher(ContactInfo):
13)     subject=models.CharField(max_length=64)
14)     salary=models.FloatField()
```

In this case only Student and Teacher tables will be created which includes all the fields of ContactInfo.

Note: CotnactInfo class is an abstract class and hence table won't be created.
It is not possible to register abstract model classes to the admin interface.If we are trying to do then we will get error.

2) Multi Table Inheritance:

- If the base class is not abstract then such type of inheritance is called multi table inheritance.
- In Multitable inheritance, inside database tables will be created for both Parent and Child classes. Multi table inheritance uses an implicit OneToOneField to link Parent and Child. i.e by using one-to-one relationship multi table inheritance is internally implemented.
- Django hides internal structure and creates feeling that both tables are independent.

```
1)  class BasicModel(models.Model):
2)     f1=models.CharField(max_length=64)
3)     f2=models.CharField(max_length=64)
4)     f3=models.CharField(max_length=64)
5)
6)  class StandardModel(BasicModel):
7)     f4=models.CharField(max_length=64)
8)     f5=models.CharField(max_length=64)
```

Corresponding Database Tables are:

mysql> desc testapp_basicmodel;

```
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| f1    | varchar(64) | NO   |     | NULL    |                |
| f2    | varchar(64) | NO   |     | NULL    |                |
| f3    | varchar(64) | NO   |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
```

4 rows in set (0.00 sec)

mysql> desc testapp_standardmodel;

```
+-------------------+-------------+------+-----+---------+-------+
| Field             | Type        | Null | Key | Default | Extra |
+-------------------+-------------+------+-----+---------+-------+
| basicmodel_ptr_id | int(11)     | NO   | PRI | NULL    |       |
| f4                | varchar(64) | NO   |     | NULL    |       |
```

```
| f5                 | varchar(64) | NO   |     | NULL    |       |
+--------------------+-------------+------+-----+---------+-------+
```

3 rows in set (0.01 sec)

3) Multi Level Inheritance:

Inheritance at multiple levels.

```
1)  class Person(models.Model):
2)      name=models.CharField(max_length=64)
3)      age=models.IntegerField()
4)
5)  class Employee(Person):
6)      eno=models.IntegerField()
7)      esal=models.FloatField()
8)
9)  class Manager(Employee):
10)     exp=models.IntegerField()
11)     team_size=models.IntegerField()
```

Note: Multilevel inheritance internally multitable inheritance only.

4) Multiple Inheritance:

If model class extends multiple parent classes simultaneously then such type of inheritance is called Multiple Inheritance.

```
1)  class Parent1(models.Model):
2)      f1=models.CharField(max_length=64)
3)      f2=models.CharField(max_length=64)
4)
5)  class Parent2(models.Model):
6)      f3=models.CharField(max_length=64)
7)      f4=models.CharField(max_length=64)
8)
9)  class Child(Parent1,Parent2):
10)     f5=models.CharField(max_length=64)
11)     f6=models.CharField(max_length=64)
```

Note:

1. Multiple inheritance is also internally multi table inheritance only.
2. In multiple inheritance Parent classes should not contain any common field, otherwise we will get error.

Model Manager:

Model Manager can be used to interact with the database. By default Model Manager is available through the Model.objects property.i.e Model.objects is of type django.db.models.manager.Manager.

1) What is the purpose of Model Manager?

To interact with database

2. How to get Default Model Manager?

By using Model.objects property

3. Model Manager is what type?

django.db.models.manager.Manager

```
>>> from testapp.models import Employee
>>> type(Employee.objects)
<class 'django.db.models.manager.Manager'>
```

We can customize the defualt behaviour of Model Manager by defining our own Customer Manager.

How to define our own Custom Manager:

We have to write child class for models.Manager.
Whenever we are using all() method,internally it will call get_queryset() method.
To customize behaviour we have to override this method in our Custom Manager class.

Eg: To retrieve all employees data according to ascending order of eno, we have to define Custom Manager class as follows.

models.py

```
1)  from django.db import models
2)  class CustomManager(models.Manager):
3)      def get_queryset(self):
4)          return super().get_queryset().order_by('eno')
5)  # Create your models here.
6)
7)  class Employee(models.Model):
8)      eno=models.IntegerField()
9)      ename=models.CharField(max_length=64)
10)     esal=models.FloatField()
11)     eaddr=models.CharField(max_length=256)
12)     objects=CustomManager()
```

When ever we are using all() method it will always get employees in ascending order of eno

Based on our requirement we can define our own new methods also inside Custom Manager class.

```
class CustomManager(models.Manager):
   def get_queryset(self):
      return super().get_queryset().order_by('eno')

   def get_emp_sal_range(self,esal1,esal2):
      return super().get_queryset().filter(esal__range=(esal1,esal2))

   def get_employees_sorted_by(self,param):
      return super().get_queryset().order_by(param)
```

Q) To Customize all() Method Behaviour, which Method we have to override inside Custom Manager Class? get_queryset() Method

Q) In Custom Manager Class, is it Possible to define New Methods?
Yes

views.py

```
1)  from django.shortcuts import render
2)  from testapp.models import Employee
3)
4)  # Create your views here.
5)  def display_view(request):
6)      #employees=Employee.objects.get_emp_sal_range(12000,20000)
7)  employees=Employee.objects.get_employees_sorted_by('esal')
8)      my_dict={'employees':employees}
9)      return render(request,'testapp/index.html',my_dict)
```

index.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)      {%block body_block%}
4)      <h1>Employee Information Dash Board</h1><hr>
```


```
5)      <table border='2'>
6)        <thead>
7)          <th>Employee Number</th>
8)          <th>Employee Name</th>
9)          <th>Employee Salary</th>
10)         <th>Employee Address</th>
11)
12)       </thead>
13)       {%for  emp in employees %}
14)       <tr>
15)         <td>{{emp.eno}}</td>
16)         <td>{{emp.ename}}</td>
17)         <td>{{emp.esal}}</td>
18)         <td>{{emp.eaddr}}</td>
19)
20)       </tr>
21)       {%endfor%}
22)     </table><br><br><br>
23)  {%endblock%}
```

5) Proxy Model Inheritance:

- For the same Model we can provide a customized view without touching the database is possible by using Proxy Model Inheritance.
- In this inheritance a seperate new table won't be created and the new model also pointing to the same old table.

```
1)  class Employee(models.Model):
2)     fields
3)
4)  class ProxyEmployee(Employee):
5)     class Meta:
6)        proxy=True
```

- Both Employee and ProxyEmployee are pointing to the same table only.
- In the admin interface if we add a new record to either Employee or ProxyEmployee, then automatically those changes will be reflected to other model view.

Demo Application:

models.py

```
1)  from django.db import models
```

```
2)
3)  class CustomManager1(models.Manager):
4)      def get_queryset(self):
5)          return super().get_queryset().filter(esal__gte=15000)
6)
7)  class CustomManager2(models.Manager):
8)      def get_queryset(self):
9)          return super().get_queryset().order_by('ename')
10)
11) class CustomManager3(models.Manager):
12)     def get_queryset(self):
13)         return super().get_queryset().filter(eno__lt=1000)
14)
15) # Create your models here.
16) class Employee(models.Model):
17)     eno=models.IntegerField()
18)     ename=models.CharField(max_length=64)
19)     esal=models.FloatField()
20)     eaddr=models.CharField(max_length=256)
21)     objects=CustomManager1()
22)
23) class ProxyEmployee(Employee):
24)     objects=CustomManager2()
25)     class Meta:
26)         proxy=True
27)
28) class ProxyEmployee2(Employee):
29)     objects=CustomManager3()
30)     class Meta:
31)         proxy=True
```

admin.py

```
1)  from django.contrib import admin
2)  from testapp.models import Employee,ProxyEmployee,ProxyEmployee2
3)
4)  # Register your models here.
5)  class EmployeeAdmin(admin.ModelAdmin):
6)      list_display=['eno','ename','esal','eaddr']
7)  class ProxyEmployeeAdmin(admin.ModelAdmin):
8)      list_display=['eno','ename','esal','eaddr']
9)
10) class ProxyEmployee2Admin(admin.ModelAdmin):
11)     list_display=['eno','ename','esal','eaddr']
12)
```

```
13) admin.site.register(Employee,EmployeeAdmin)
14) admin.site.register(ProxyEmployee,ProxyEmployeeAdmin)
15) admin.site.register(ProxyEmployee2,ProxyEmployee2Admin)
```

views.py

```
1)  from django.shortcuts import render
2)  from testapp.models import Employee,ProxyEmployee,ProxyEmployee2
3)  # Create your views here.
4)  def display_view(request):
5)      # employees=Employee.objects.all()
6)      # employees=ProxyEmployee.objects.all()
7)      employees=ProxyEmployee2.objects.all()
8)      my_dict={'employees':employees}
9)      return render(request,'testapp/index.html',my_dict)
```

index.html

```
1)  <!DOCTYPE html>
2)  {% extends 'testapp/base.html'%}
3)      {%block body_block%}
4)      <h1>Employee Information Dash Board</h1><hr>
5)      <table border='2'>
6)        <thead>
7)          <th>Employee Number</th>
8)          <th>Employee Name</th>
9)          <th>Employee Salary</th>
10)         <th>Employee Address</th>
11)       </thead>
12)       {%for  emp in employees %}
13)       <tr>
14)         <td>{{emp.eno}}</td>
15)         <td>{{emp.ename}}</td>
16)         <td>{{emp.esal}}</td>
17)         <td>{{emp.eaddr}}</td>
18)       </tr>
19)       {%endfor%}
20)     </table><br><br><br>
21)     {%endblock%}
```