
بسم الله

ADVANCED TOPICS IN MODELS

HOSSEIN FORGHANI

MAKTAB SHARIF

Contents

- Meta Options
- Raw SQL
- Model Inheritance

Meta Options

- Give your model metadata by using an inner class Meta
- Model metadata is “anything that’s not a field”

```python
from django.db import models

class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

Meta Options – cont.

- abstract : used to make the model abstract
- app_label: If a model is defined outside of an application in INSTALLED_APPS
- db_table: table name
- ordering: The default ordering
- permissions: Extra permissions to enter into the permissions table
- indexes: A list of indexes that you want to define on the model
- unique_together
- verbose_name, verbose_name_plural: singular and plural human-readable names

Overriding predefined model methods

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super().save(*args, **kwargs)  # Call the "real" save()
method.
        do_something_else()
```

Overriding predefined model methods – cont.

- You can also prevent saving:

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super().save(*args, **kwargs)  # Call the "real"
save() method.
```

Performing raw SQL queries

- The Django ORM provides many tools to express queries without writing raw SQL such as annotate and aggregate
- But Django gives you the possibility of performing raw SQL queries
- You should be very careful whenever you write raw SQL. You should properly escape any parameters that the user can control to avoid SQL injection

Performing raw queries

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

Index lookups

- raw() supports indexing

```
>>> first_person = Person.objects.raw('SELECT * FROM
myapp_person')[0]
```

- However, the indexing and slicing are not performed at the database level. So it is more efficient to write:

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person
LIMIT 1')[0]
```

Deferring model fields

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM
myapp_person'):
...     print(p.first_name, # This will be retrieved by the
original query
...           p.last_name) # This will be retrieved on demand
...
John Smith
Jane Jones
```

- Only the primary key field cannot be left out

Adding annotations

- For example, we could use PostgreSQL's age() function:

```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS age
FROM myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```

Passing parameters into raw()

- If you need to perform parameterized queries, you can use the params argument to raw():

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE
last_name = %s', [lname])
```

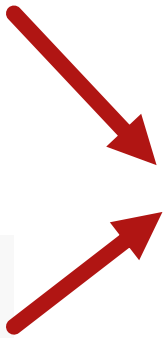
- params is a list or dictionary of parameters
- Use %(key)s placeholders for a dictionary (where key is replaced by a dictionary key)

SQL injection protection

- Do not use string formatting on raw queries or quote placeholders in your SQL strings!

```
>>> query = 'SELECT * FROM myapp_person WHERE last_name
= %s' % lname
>>> Person.objects.raw(query)
```

```
>>> query = "SELECT * FROM myapp_person WHERE last_name
= '%s'"
```



mistakes!

Executing custom SQL directly

- Very similar to psycopg2 and mysql. Use fetchone() or fetchall() :

```
from django.db import connection

def my_custom_sql(self):
    with connection.cursor() as cursor:
        cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s",
[self.baz])
        cursor.execute("SELECT foo FROM bar WHERE baz = %s",
[self.baz])
        row = cursor.fetchone()

    return row
```

- If you are using more than one database, you can use django.db.connections to obtain the connection (and cursor) for a specific database:

```
from django.db import connections
with connections['my_db_alias'].cursor() as cursor:
    # Your code here...
```

Model Inheritance

3 Styles of Inheritance

1. The abstract parent class: when we don't want the parent model have any tables

2. Multi-table inheritance: when we want to both parent and child have their own tables

3. Proxy models: if you only want to modify the Python-level behavior of a model, without changing the models fields in any way

1: Abstract Base Classes

- The Student model will have three fields: name, age and home_group
- CommonInfo will then not be used to create any database table

```python
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

Abstract and Meta

- The Meta class is inherited (but the child is abstract=false by default)
- Also you can subclass it in the child model:

```python
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

Multiple Inheritance of Abstract Models

```python
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True
        ordering = ['name']

class Unmanaged(models.Model):
    class Meta:
        abstract = True
        managed = False

class Student(CommonInfo, Unmanaged):
    home_group = models.CharField(max_length=5)

    class Meta(CommonInfo.Meta, Unmanaged.Meta):
        pass
```

Must be inherited explicitly

related_name on Abstract Model

- If you are using related_name or related_query_name on a ForeignKey or ManyToManyField, you must always specify a unique reverse name and query name for the field.
- Use %(app_label)s and %(class)s
- Then the reverse name of the common.ChildA.m2m field will be common_childa_related

```python
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(
        OtherModel,
        related_name="%(app_label)s_%(class)s_related",
        related_query_name="%(app_label)s_%(class)ss",
    )

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

2: Multi-table inheritance

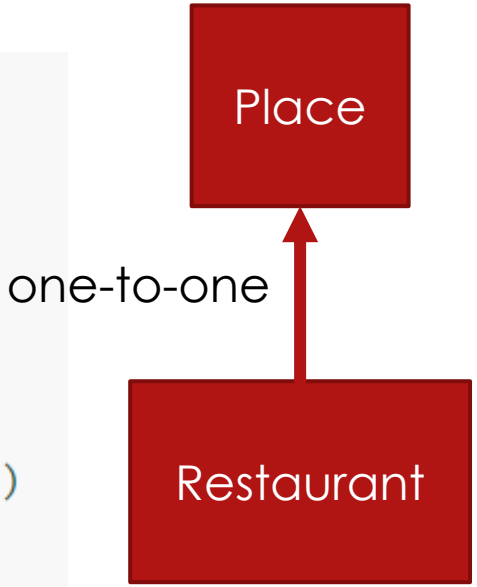
- The second type of model inheritance supported by Django is when each model in the hierarchy is a model all by itself
- An automatically-created OneToOneField established between the child model and each of its parents

Relation of Parents and Children

```python
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```



id	name	address

id	place_ptr_id	serves_hot_dogs	serves_pizza

The OneToOneField

- This is the implicit OneToOneField which you can override it:

```
place_ptr = models.OneToOneField(
    Place, on_delete=models.CASCADE,
    parent_link=True,
    primary_key=True,
)
```

Parents and Children Fields

- All of the fields of Place will also be available in Restaurant, although the data will reside in a different database table

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

- If you have a Place that is also a Restaurant, you can get from the Place object to the Restaurant:

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

Inheritance and Reverse Relations

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

Will raise error

```
Reverse query name for 'Supplier.customers' clashes with reverse
query
name for 'Supplier.place_ptr'.

HINT: Add or change a related_name argument to the definition for
'Supplier.customers' or 'Supplier.place_ptr'.
```

Solution

```
models.ManyToManyField(Place, related_name='provider')
```

Meta and Multi-table Inheritance

- In contrast with the abstract base class, a child model does not have access to its parent's Meta class
- Except a few limited cases such as ordering or get_latest_by

3: Proxy models

- If you only want to modify the Python-level behavior of a model, without changing the models fields in any way
- You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model
- The difference is that you can change things like the default model ordering or the default manager in the proxy, without having to alter the original

Adding Python-level Functionality

- For example, suppose you want to add a method to the Person model. You can do it like this

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

Using in Querying

- The MyPerson class operates on the same database table as its parent Person class. In particular, any new instances of Person will also be accessible through MyPerson, and vice-versa:

```
>>> p = Person.objects.create(first_name="foobar")
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

Customizing Ordering

- You could also use a proxy model to define a different default ordering on a model:

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

- Now normal Person queries will be unordered and OrderedPerson queries will be ordered by last_name
- Proxy models inherit Meta attributes in the same way as regular models

References

- https://docs.djangoproject.com/en/3.1/topics/db/models/
- https://docs.djangoproject.com/en/3.1/topics/db/sql/

Any Question?