WRITE A POST

Prateek FOLLOW

Basics of Django ORM

Published Oct 15, 2017

The article was originally published at Django ORM Basics - OverIQ.com.

This articles provides all the basics you need to know to get started with Django ORM. Django ORM provides an elegant and powerful way to interact with the database. ORM stands for Object Relational Mapper. It is just a fancy word describing how to access the data stored in the database in Object Oriented fashion.

Start Django shell using the `shell` command.

```
(env) C:\Users\Q\TGDB\django_project>python manage.py shell
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 20:20:57) [MSC v.1600 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

To work with database inside the shell, first we have to import necessary models. In this section we will be working on models stored in the blog app, so lets start by importing all the models in the blog app.

Enjoy this post? 3 2

```
>>>
>>> from blog.models import Post, Author, Category, Tag
>>>
```

At this point tables corresponding to these 4 models are empty.

[database_snapshot.png]

Let's start by creating an `Author` object.

```
>>>
>>> a = Author(name='tom', email='tom@email.com', active=True)
>>>
```

Try printing variable `a` you will get the following output:

```
>>>
>>> a
<Author: Author object>
>>>
>>> print(a)
Author object
>>>
```

You can access the attributes of an object using the (`.`) dot operator.

Enjoy this post? 3 2

```
>>>
>>> a.name
'tom'
>>>
>>> a.email
'tom@email.com'
>>>
>>> a.active
True
>>>
```

Notice that at the time of object creating the Author object we didn't provide any value to `created_on` and `last_logged_in` field because these fields have `auto_now_add` and `auto_now` set to `True` respectively. As a result, Django will automatically provides the current date and time at the time when you save the object to the database. However, If we hadn't set `auto_now_add` and `auto_now` parameters then we would have to pass values to `created_on` and `last_logged_in` field as follows:

```
>>>
>>> import datetime
>>>
>>> r = Author(name="root", email="root@mail.com", active=True, created_on=date
ime.datetime.now(), last_logged_in=datetime.datetime.now())
>>>
>>>
```

At this point object pointed to by variable `a` exits only inside the Django shell. To save the object to the database call the `save()` method on the object.

Enjoy this post? 3 2

```
>>>
>>> a.save()
>>>
```

Recall that every model we define inherits from `models.Model` class, this is where the `save()` method comes from.

To view this newly added object open `blog_author` table inside Navicat Premium.

Enjoy this post? 3 2

Similary, `models.Model` class also defines a `delete()` method to delete an object from the database.

Let's delete the object from the database.

Enjoy this post? 3 2

```
>>>
>>> a.delete()
(1, {'blog.Author': 1})
>>>
```

This command removes author `tom` from the database. However, it is still exists inside the shell.

```
>>> a
<Author: Author object>
>>>
```

Sure, the object exists in shell.

Defining __str__() method on model

At this point if you try to print `Author` or any other Model object inside the shell it would output a string that looks something like this:

```
>>>
>>> a
<Author: Author object>
>>>
>>> print(a)
Author object
>>>
```

This `Author object` is not very helpful. Right ? Is there anyway to change it ?

Enjoy this post? ♡ 3 2

learned how to define fields. What about behaviors ? Well behaviors means many things as we will see later. In this case we want to change the behavior of `Author` class - the way it prints an object. We can change this behavior easily by adding a `__str__()` method in the `Author` model.

A `__str__()` is a special method which tells Python how to display an object in human readable form. Open `models.py` inside blog app and make the following changes to the `Author` model.

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField(unique=True)
    active = models.BooleanField(default=False)
    created_on = models.DateTimeField(auto_now_add=True)
    last_logged_in = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.name + " : " + self.email
```

While we are at it lets add `__str__()` method to `Category` , `Tag` and `Post` model too.

Enjoy this post? 3 2

```
class Category(models.Model):
    ...

    def __str__(self):
        return self.name


class Tag(models.Model):
    ...

    def __str__(self):
        return self.name


class Post(models.Model):
    ...

    def __str__(self):
        return self.title
```

Does this ring a bell ? You might say "We are changing our models so we should run `makemigrations` right ?".

Well No! Most of the time, we run `makemigrations` command only in the following two cases:

1. When we add/modify fields in the model.
2. When we adding/modify `Meta` classes.

We will learn what `Meta` classes are in upcoming chapters.

In fact, adding/modifying methods to our models are not even considered as changes. You can test this using the `makemigrations` command.

Enjoy this post? 3 2

```
(env) C:\Users\Q\TGDB\django_project>python manage.py makemigrations
No changes detected

(env) C:\Users\Q\TGDB\django_project>
```

See `makemigrations` returns `"No changes detected"` .

After adding `__str__()` to `models.py` file, if you try to print `Author` object you would get the same output as before.

```
>>>
>>> a
<Author: Author object>
>>>
>>> print(a)
Author object
>>>
```

In order for the changes to take affect, exit the Django shell by hitting Ctrl+Z (Windows) or Ctrl+D (Linux) and start it again using `python manage.py shell` command.

Import necessary models and create a new `Author` object.

Enjoy this post? 3 2

```
(env) C:\Users\Q\my_workspace\django_project>python manage.py shell
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 20:20:57) [MSC v.1600 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
>>> from blog.models import Post, Author, Category, Tag
>>>
>>> a = Author(name='tom', email='tom@email.com')
>>>
```

Now lets try printing the object inside the Django shell.

```
>>> a
<Author: tom : tom@email.com>
>>>
>>> print(a)
tom : tom@email.com

>>>
```

This is much better. isn't ?

Save this object to the database using the `save()` method.

```
>>>
>>> a.save()
>>>
```

When you save an object the primary key is assigned automatically. Once

Enjoy this post? 3 2

`pk` attribute.

```
>>>
>>> a.id
2
>>> a.pk
2
>>>
```

If you want to alter the value of attributes of an object just assign a new value and call the `save()` method again.

```
>>>
>>> a.name = 'Tom'
>>> a.email = 'tom@gmail.com'
>>>
>>> a
<Author: Tom : tom@gmail.com>
>>>
```

These changes are not yet saved to the database, you have to call `save()` to make the changes permanent.

```
>>>
>>> a.save()
>>>
```

Database Access through Managers

According to the Django documentation - Django by default adds a manager

Enjoy this post? 3 2

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` manager type model class name followed by the (`.`) dot operator then the `objects` manager. For example:

```
>>>
>>> Author.objects
<django.db.models.manager.Manager object at 0x00000000042CE978>
>>> type(Author.objects)
<class 'django.db.models.manager.Manager'>
>>>
```

As you can see `objects` is just a instance of `django.db.models.manager.Manager` class. The `objects` manager provides a whole range of methods which allows us to interact with the database easily.

Let's discuss some important methods of `objects` manager.

The 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:

Enjoy this post? 3 2

```
>>>
>>> a2 = Author.objects.create(name='jerry', email='jerry@mail.com')
>>> a2
<Author: jerry : jerry@mail.com>
>>> a2.pk
4
>>>
```

The bulk_create() method

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

```
>>>
>>> Author.objects.bulk_create([
...  Author(name='spike', email='spike@mail.com'),
...  Author(name='tyke', email='tyke@mail.com'),
...  Author(name='droopy', email='droopy@mail.com'),
... ])
[<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: dro
py : droopy@mail.com>]
>>>
```

At this point `blog_author` table should looks like this:

Enjoy this post? 3 2

The all() method

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

```
>>>
>>> Author.objects.all()
<QuerySet [<Author: tom : tom@email.com>, <Author: jerry : jerry@mail.com>, <Au
hor: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: droopy :
droopy@mail.com>]>
>>>
```

The above command fetches all the records from the `Author` 's table.

The `all()` method returns a `QuerySet` object. A `QuerySet` object looks a lot 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.

```
>>>
>>> r = Author.objects.all()
>>> r
<QuerySet [<Author: tom : tom@email.com>, <Author: jerry : jerry@mail.com>, <Au
hor: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: droopy :
droopy@mail.com>]>
>>>

>>>
>>> r[0]
<Author: tom : tom@email.com>
>>>
>>> r[1]
<Author: jerry : jerry@mail.com>
>>>
>>> r[2]
<Author: spike : spike@mail.com>
>>>
```

Enjoy this post? 3 2

Although `r` points to an object of type `QuerySet` but `r[0]`, `r[1]`, `r[2]` and so on, points to an object of type `Author`.

```
>>> type(r[0])
<class 'blog.models.Author'>
>>>
>>> type(r[1])
<class 'blog.models.Author'>
>>>
>>> type(r[3])
<class 'blog.models.Author'>
>>>
```

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.

```
>>>
>>> r = Author.objects.all()
>>> for a in r:
...     print("Author: {0}".format(a.name))
...
Author: tom
Author: jerry
Author: spike
Author: tyke
Author: droopy
>>>
```

The count() method

Enjoy this post? 3 2

```
>>>
>>> Author.objects.count()
5
>>>
```

`Author.objects.all().count()` also returns the same thing.

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.

```
>>>
>>> Author.objects.filter(name='tom')
<QuerySet [<Author: tom : tom@email.com>]>
>>>
>>> Author.objects.filter(name='johnny')
<QuerySet []>
>>>
```

`Author.objects.filter(name='tom')` translates to SQL something like this:

```
SELECT * from blog_author
where name = 'tom'
```

As database has only one record where name is `'tom'`, the `QuerySet` object contains only a single record. If we had two records where name is `'tom'` then `filter()` would have returned a `QuerySet` object containing two `Author`

Enjoy this post? 3 2

Similarly, `Author.objects.filter(name='johnny')` translates to SQL rougly as follows:

```
SELECT * from blog_author
where name = 'johnny'
```

As there are no records where name is `'johnny'` an empty `QuerySet` is returned.

We can also directly print the raw SQL django uses to query the database using the `query` attribute of the `QuerySet` object.

```
>>>
>>> print(Author.objects.filter(name='tom').query)
SELECT "blog_author"."id", "blog_author"."name", "blog_author"."email", "blog_a
thor"."active", "blog_author"."created_on", "blog_author"."last_logged_in" FROM
"blog_author" WHERE "blog_author"."name" = tom
>>>
```

Matching performed using keyword arguments are case-sensitive.

```
>>>
>>> Author.objects.filter(email='jerry@mail.com')
<QuerySet [<Author: jerry : jerry@mail.com>]>
>>>
>>> Author.objects.filter(email='JERRY@mail.com')
<QuerySet []>
>>>
```

Enjoy this post? 3 2

The last query returns an empty `QuerySet` because there are no records where email is `"JERRY@mail.com"` , although there is a record where name is `"jerry@mail.com"` .

You can also pass multiple keyword arguments to the `filter()` method.

```
>>>
>>> Author.objects.filter(name='spike', email='spike@mail.com')
<QuerySet [<Author: spike : spike@mail.com>]>
>>>
```

This traslates to SQL rougly as follows:

```
SELECT * from blog_author
where name = 'spike' and email ='spike@mail.com'
```

Django Field Lookups

In addition to passing field names as keyword arguments. You can also use something called lookups.
Managers and QuerySet objects comes with a feature called lookups. A lookup is composed of a model field followed by two underscores (`__`) which is then followed by lookup name. Let's take some examples.

__contains lookup

Enjoy this post? 3 2

```
>>>
>>> Author.objects.filter(name__contains="ke")
<QuerySet [<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>]>
>>>
>>>
```

Here `__contains` lookup finds all the records where `name` field contains the word `"ke"` .

`Author.objects.filter(name__contains="ke")` translates to SQL roughly as follows:

```
SELECT * from blog_author
where name like '%ke%'
```

Matching performed by `__contains` lookup is case-sensitive. If you want to perform case-insensitive match use `__icontains` . However, SQLite doesn't support case-sensitive `LIKE` statements. As a result `__contains` and `__icontains` returns the same result.

__startswith lookup

```
>>>
>>> Author.objects.filter(name__startswith="t")
<QuerySet [<Author: tom : tom@email.com>, <Author: tyke : tyke@mail.com>]>
>>>
```

`__startswith` lookup finds all the records whose `name` field start with `"t"` .

Enjoy this post? 3 2

```
>>>
>>> Author.objects.filter(email__endswith="com")
<QuerySet [<Author: tom : tom@email.com>, <Author: jerry : jerry@mail.com>, <Au
hor: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <Author: droopy :
droopy@mail.com>]>
>>>
>>>
```

Here `__endswith` lookup finds all the records whose `email` ends with `"com"` . Both `__startswith` and `__endswith` are case-sensitive. Their case-insensitive equivalents are `__istartswith` and `__iendswith` .

__gt lookup

```
>>>
>>> Author.objects.filter(id__gt=3)
<QuerySet [<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>, <A
thor: droopy : droopy@mail.com>]>
>>>
```

here `__gt` lookup finds all the records whose `id` or primary key (`pk`) is greater than `3` . There also exists a complementary lookup called `__lt` .

```
>>>
>>> Author.objects.filter(id__lt=3)
<QuerySet [<Author: tom : tom@email.com>]>
>>>
>>>
```

Enjoy this post? 3 2

Here `__lt` lookups finds all the records whose primary key is less than 3. There are two more similar lookups called `__gte` and `__lte` which finds records which are greater than or equal to and less than or equal to respectively.

To view full list of field lookups check out the Django documentation on lookups.

Retrieving a single record using the get() method

The `filter()` method described in the above section returns a `QuerySet`, sometimes we just want to fetch a single record from the table. To handle these situations `objects` manager provides a `get()` method. 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.

```
>>>
>>> Author.objects.get(name="tom")
<Author: tom : tom@email.com>
>>>
>>> Author.objects.filter(name="tom")
<QuerySet [<Author: tom : tom@email.com>]>
>>>
```

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

Enjoy this post? 3 2

```
>>>
>>> Author.objects.filter(name__contains="ke")
<QuerySet [<Author: spike : spike@mail.com>, <Author: tyke : tyke@mail.com>]>
>>>
```

```
>>>
>>> Author.objects.get(name__contains="ke")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\manager.py", lin
 85, in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\query.py", line
89, in get
    (self.model._meta.object_name, num)
blog.models.MultipleObjectsReturned: get() returned more than one Author -- it
eturned 2!
>>>
```

Here `get()` method raises a `MultipleObjectsReturned` because there are multiple objects in the database that matches the given parameter.

Similarly, if you try to access an object which do not exists then the `get()` method will raise an `DoesNotExist` exception.

Enjoy this post?

3 2

```
>>>
>>> Author.objects.get(name__contains="captain planet")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\manager.py", lin
 85, in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\query.py", line
85, in get
    self.model._meta.object_name
blog.models.DoesNotExist: Author matching query does not exist.
>>>
```

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")
<QuerySet [<Author: droopy : droopy@mail.com>, <Author: tyke : tyke@mail.com>,
Author: spike : spike@mail.com>, <Author: jerry : jerry@mail.com>, <Author: tom
: tom@email.com>]>
>>>
```

This command retrieves all `Author` objects according to id field in ascending order. The above command translates to SQL roughly as follows:

Enjoy this post? 3 2

```
SELECT * from blog_author
order by id
```

It turns out that we can chain methods which returns `QuerySet` objects. Doing so allows us to modify the database query further.

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

This command retrieves only those `Author` objects whose `id` is greater than `3` and orders those objects according to values in the `name` field in ascending order. The above command translates to SQL roughly as follows:

```
SELECT * from blog_author
where id > 3
order by name
```

To reverse the sorting ordering add `-` sign before the field name like this:

Enjoy this post? 3 2

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

The above command traslates to the SQL as follows:

```
SELECT * from blog_author
where id > 3
order by name DESC
```

You can also sort the result by multiple field 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 command will sort the result first by `name` in ascending and then by `email` in descending order

Selecting the fields

When you run a query to database like this:

Enjoy this post? 3 2

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

It returns data from all the fields (columns). 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` . For example:

```
>>>
>>> Author.objects.values_list("id", "name")
<QuerySet [(1, 'tom'), (2, 'jerry'), (3, 'spike'), (4, 'tyke'), (5, 'droopy')]>
>>>
```

Notice that the `values_list()` method returns a `QuerySet` where each element is a tuple. And the tuple only contains data from the fields which we have specified in the `values_list()` .

Enjoy this post? 3 2

```
>>>
>>> Author.objects.filter(id__gt=3).values_list("id", "name")
<QuerySet [(4, 'spike'), (5, 'tyke'), (6, 'droopy')]>
>>>


>>>
>>> r = Author.objects.filter(id__gt=3).values_list("id", "name")
>>> r
<QuerySet [(4, 'spike'), (5, 'tyke'), (6, 'droopy')]>
>>> r[0]
(4, 'spike')
>>> r[0][0]
4
>>> r[0][1]
'spike'
>>>
```

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 tuple.

Enjoy this post? 3 2

```
>>>
>>> r = Author.objects.filter(id__gt=3).values("id", "name")
>>>
>>> r
<QuerySet [{'name': 'spike', 'id': 4}, {'name': 'tyke', 'id': 5}, {'name': 'dro
py', 'id': 6}]>
>>>
>>> type(r[0])
<class 'dict'>
>>>
>>> r[0]
{'name': 'spike', 'id': 4}
>>>
>>> r[0]['name']
'spike'
>>> r[0]['id']
4
>>>
```

Slicing Results

You can use Python list slicing syntax i.e `[start:end]` to limit your `QuerySet` object to certain number of results.

Example 1:

```
>>>
>>> # returns the second record after sorting the result
>>>
>>> Author.objects.order_by("-id")[1]
<Author: tyke : tyke@mail.com>
>>>
```

Enjoy this post? 3 2

```
SELECT * from blog_author
order by -id
limit 1, 1
```

Example 2:

```
>>>
>>> # returns the first three objects after sorting the result
>>>
>>> Author.objects.order_by("-id")[:3]
<QuerySet [<Author: droopy : droopy@mail.com>, <Author: tyke : tyke@mail.com>, 
Author: spike : spike@mail.com>]>
>>>
>>>
```

This command roughly translates to SQL as follows:

```
SELECT * from blog_author
order by -id
limit 0, 3
```

Example 3:

Enjoy this post? 3 2

```
>>>
>>> # returns objects from 3rd index to 5th index after sorting the result
>>>
>>> Author.objects.order_by("-id")[2:5]
<QuerySet [<Author: spike : spike@mail.com>, <Author: jerry : jerry@mail.com>,
Author: tom : tom@email.com>]>
>>>
>>>
```

This command roughly translates to SQL as follows:

```
SELECT * from blog_author
order by -id
limit 2, 3
```

Negative slicing is not supported.

```
>>>
>>> Author.objects.order_by("-id")[-1]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\K\TGDB\env\lib\site-packages\django\db\models\query.py", line
75, in __getitem__
    "Negative indexing is not supported."
AssertionError: Negative indexing is not supported.
>>>
>>>
```

Updating Multiple Objects

Enjoy this post? 3 2

Recall that one way to update an object to call `save()` method after updating it's attributes. For example:

```
>>>
>>>
>>> a = Author.objects.get(pk=2)
>>> a
<Author: tom : tom@email.com>
>>>
>>> a.name = 'tommy'
>>> a.email = 'tommy@mail.com'
>>>
>>> a.save()
>>>
>>> a = Author.objects.get(pk=2)
>>> a
<Author: tommy : tommy@mail.com>
>>>
>>>
```

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 update is successful it returns number of rows updated.

```
>>>
>>> Author.objects.filter(pk=2).update(email='tom@yahoo.com')
1
>>>
```

This command will update the email of author whose `pk` is 2 to `tom@yahoo.com` .

Enjoy this post? 3 2

```
UPDATE blog_author SET
email='tom@mail.com'
WHERE id = 2;
```

Updating all objects

```
>>>
>>>
>>> Author.objects.all().update(active=True)
5
>>>
>>>
```

The above command updates the value of `active` field to `True` for all the records in the `Author` 's table. The above command is equivalent to the following command:

```
Author.objects.update(active=True)
```

The SQL equivalent of the above command is:

```
UPDATE blog_author SET
active=1
```

Deleting records

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

Enjoy this post? 3 2

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})
>>>
>>>
```

You should now have a solid understanding of Django ORM. In the next lesson, we will discuss how to access data from multiple table using Django ORM.

Django Django orm

Enjoy this post? 3 2

Enjoy this post? Give **Prateek** a like if it's helpful.

3 2 SHARE

Prateek

FOLLOW

2 Replies

Leave a reply

vinod 4 months ago

Is it possible to create our own managers in django.

Reply

Wael Salman a year ago

This is an amazing post. Amazing details and details in depth. I want to thank you for the great effort.

I hope you can prepare another post in depth explaining another issues in django. I suggest talking about class based views vs functions based views, and how all of the inheritance is taking place, and how the class methods are called… etc…

Reply

Enjoy this post? 3 2

Django Stars

Merging Django ORM with SQLAlchemy for Easier Data Analysis

Enjoy this post?

3 2

Development of products with Django framework is usually easy and straightforward; great documentation, many tools out of the box, plenty of open source libraries and big community. Django ORM takes full control about SQL layer protecting you from mistakes, and underlying details of queries so you can spend more time on designing and building your application structure

READ MORE

Enjoy this post?

3 2

Enjoy this post?

3 2