

MANAGERS

Django basic to pro in Nepali

Managers

class Manager

A Manager is the interface through which database query operations are provided to Django models. At least one Manager exists for every model in a Django application.

Manager names

By default, Django adds a Manager with the name `objects` to every Django model class. However, if you want to use `objects` as a field name, or if you want to use a name other than `objects` for the Manager, you can rename it on a per-model basis. To rename the Manager for a given class, define a class attribute of type `models.Manager()` on that model. For example:

```
from django.db import models
class Person(models.Model):
    # ...
    people = models.Manager()
```

Using this example model, `Person.objects` will generate an `AttributeError` exception, but `Person.people.all()` will provide a list of all Person objects.

Custom managers

You can use a custom Manager in a particular model by extending the base Manager class and instantiating your custom Manager in your model. There are two reasons you might want to customize a Manager: to add extra Manager methods, and/or to modify the initial QuerySet the Manager returns.

Adding extra manager methods

Adding extra Manager methods is the preferred way to add “table-level” functionality to your models. (For “row-level” functionality – i.e., functions that act on a single instance of a model object – use Model methods, not custom Manager methods.) For example, this custom Manager adds a method `with_counts()`:

```
from django.db import models
from django.db.models.functions import Coalesce

class PollManager(models.Manager):
    def with_counts(self):
        return self.annotate(num_responses=Coalesce(models.Count("response"), 0))

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll, on_delete=models.CASCADE)
    # ...
```

With this example, you’d use `OpinionPoll.objects.with_counts()` to get a QuerySet of OpinionPoll objects with the extra `num_responses` attribute attached. A custom Manager method can return anything you want. It doesn’t have to return a QuerySet. Another thing to note is that Manager methods can access `self.model` to get the model class to which they’re attached.

Modifying a manager's initial QuerySet

A Manager's base QuerySet returns all objects in the system. For example, using this model:

```
from django.db import models
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
```

...the statement `Book.objects.all()` will return all books in the database. You can override a Manager's base QuerySet by overriding the `Manager.get_queryset()` method. `get_queryset()` should return a QuerySet with the properties you require. For example, the following model has two Managers – one that returns all objects, and one that returns only the books by Roald Dahl:

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_queryset(self):
        return
super().get_queryset().filter(author="Roald Dahl")
# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific
manager.
```

With this sample model, `Book.objects.all()` will return all books in the database, but `Book.dahl_objects.all()` will only return the ones written by Roald Dahl. Because `get_queryset()` returns a QuerySet object, you can use `filter()`, `exclude()` and all the other QuerySet methods on it.

This example also pointed out another interesting technique: using multiple managers on the same model. You can attach as many `Manager()` instances to a model as you'd like. This is a non-repetitive way to define common “filters” for your models.

```
class AuthorManager(models.Manager):

    def get_queryset(self):
        return
super().get_queryset().filter(role="A")

class EditorManager(models.Manager):
    def get_queryset(self):
        return
super().get_queryset().filter(role="E")
```

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1,
choices={"A": _("Author"), "E": _("Editor")})
    people = models.Manager()
    authors = AuthorManager()
    editors = EditorManager()
```

Default managers

`Model._default_manager`

If you use custom Manager objects, take note that the first Manager Django encounters (in the order in which

they're defined in the model) has a special status. Django interprets the first Manager defined in a class as the “default” Manager, and several parts of Django (including `dumpdata`) will use that Manager exclusively for that model. As a result, it's a good idea to be careful in your choice of default manager in order to avoid a situation where overriding `get_queryset()` results in an inability to retrieve objects you'd like to work with.

You can specify a custom default manager using `Meta.default_manager_name`. If you're writing some code that must handle an unknown model, for example, in a third-party app that implements a generic view, use this manager (or `_base_manager`) rather than assuming the model has an objects manager.

Base managers

`Model._base_manager`

Using managers for related object access, By default, Django uses an instance of the `Model._base_manager` manager class when accessing related objects (i.e. `choice.question`), not the `_default_manager` on the related object. This is because Django needs to be able to retrieve the related object, even if it would otherwise be filtered out (and hence be inaccessible)

by the default manager. If the normal base manager class (`django.db.models.Manager`) isn't appropriate for your circumstances, you can tell Django which class to use by setting `Meta.base_manager_name`. Base managers aren't used when querying on related models, or when accessing a one-to-many or many-to-many relationship. For example, if the Question model from the tutorial had a deleted field and a base manager that filters out instances with `deleted=True`, a `queryset` like `Choice.objects.filter(question__name__startswith='What')` would include choices related to deleted questions.

Calling custom QuerySet methods from the manager

While most methods from the standard QuerySet are accessible directly from the Manager, this is only the case for the extra methods defined on a custom QuerySet if you also implement them on the Manager:

```
class PersonQuerySet(models.QuerySet):
    def authors(self):
        return self.filter(role="A")
    def editors(self):
        return self.filter(role="E")
class PersonManager(models.Manager):
    def get_queryset(self):
        return PersonQuerySet(self.model, using=self._db)
    def authors(self):
        return self.get_queryset().authors()
    def editors(self):
        return self.get_queryset().editors()
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices={"A": _("Author"), "E":
        _("Editor")})
    people = PersonManager()
```

This example allows you to call both `authors()` and `editors()` directly from the manager ***Person.people***.

Creating a manager with QuerySet methods

In lieu of the above approach which requires duplicating methods on both the QuerySet and the Manager,

QuerySet.as_manager() can be used to create an instance of Manager with a copy of a custom **QuerySet's** methods:

```
class Person(models.Model):  
    ...  
    people = PersonQuerySet.as_manager()
```

The Manager instance created by **QuerySet.as_manager()** will be virtually identical to the **PersonManager** from the previous example. Not every QuerySet method makes sense at the Manager level; for instance we intentionally prevent the **QuerySet.delete()** method from being copied onto the Manager class.

Methods are copied according to the following rules:

- Public methods are copied by default.
- Private methods (starting with an underscore) are not copied by default.
- Methods with a **queryset_only** attribute set to **False** are always copied.
- Methods with a **queryset_only** attribute set to **True** are never copied.

EXAMPLE

```
class CustomQuerySet(models.QuerySet):  
    # Available on both Manager and QuerySet.  
    def public_method(self):  
        return  
  
    # Available only on QuerySet.  
    def _private_method(self):  
        return  
  
    # Available only on QuerySet.  
    def opted_out_public_method(self):  
        return  
  
    opted_out_public_method.queryset_only = True  
    # Available on both Manager and QuerySet.  
    def _opted_in_private_method(self):  
        return  
  
    _opted_in_private_method.queryset_only = False
```

`from_queryset()`

`classmethod from_queryset(queryset_class)`

For advanced usage you might want both a custom Manager and a custom QuerySet. You can do that by calling `Manager.from_queryset()` which returns a subclass of your base Manager with a copy of the custom QuerySet methods:

```
class CustomManager(models.Manager):
    def manager_only_method(self):
        return

class CustomQuerySet(models.QuerySet):
    def manager_and_queryset_method(self):
        return

class MyModel(models.Model):
    objects = CustomManager.from_queryset(CustomQuerySet)()
```

You may also store the generated class into a variable:

```
MyManager = CustomManager.from_queryset(CustomQuerySet)
```

```
class MyModel(models.Model):
    objects = MyManager()
```


Custom managers and model inheritance

Here's how Django handles custom managers and model inheritance:

1. Managers from base classes are always inherited by the child class, using Python's normal name resolution order (names on the child class override all others; then come names on the first parent class, and so on).

2. If no managers are declared on a model and/or its parents, Django automatically creates the objects manager.

3. The default manager on a class is either the one chosen with `Meta.default_manager_name`, or the first manager declared on the model, or the default manager of the first parent model.

These rules provide the necessary flexibility if you want to install a collection of custom managers on a group of models, via an abstract base class, but still customize the default manager. For example, suppose you have this base class:

```
class AbstractBase(models.Model):  
    # ...  
    objects = CustomManager()  
    class Meta:  
        abstract = True
```

If you use this directly in a child class, `objects` will be the default manager if you declare no managers in the child class:

```
class ChildA(AbstractBase):  
    # ...  
    # This class has CustomManager as the default  
    # manager.  
    pass
```

If you want to inherit from `AbstractBase`, but provide a different default manager, you can provide the default manager on the child class:

```
class ChildB(AbstractBase):  
    # ...  
    # An explicit default manager.  
    default_manager = OtherManager()
```

Here, **default_manager** is the default. The objects manager is still available, since it's inherited, but isn't used as the default. Finally for this example, suppose you want to add extra managers to the child class, but still use the default from **AbstractBase**. You can't add the new manager directly in the child class, as that would override the default and you would have to also explicitly include all the managers from the abstract base class. The solution is to put the extra managers in another base class and introduce it into the inheritance hierarchy after the defaults:

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()
    class Meta:
        abstract = True

class ChildC(AbstractBase, ExtraManager):
    # ...
    # Default manager is CustomManager, but
    # OtherManager is
    # also available via the
    # "extra_manager" attribute.
    pass
```

Implementation concerns

Whatever features you add to your custom Manager, it must be possible to make a shallow copy of a Manager instance; i.e., the following code must work:

```
>>> import copy
>>> manager = MyManager()
>>> my_copy = copy.copy(manager)
```

Django makes shallow copies of manager objects during certain queries; if your **Manager** cannot be copied, those queries will fail. This won't be an issue for most custom managers. If you are just adding simple methods to your Manager, it is unlikely that you will inadvertently make instances of your Manager uncopyable. However, if you're overriding **__getattr__** or some other private method of your **Manager** object that controls object state, you should ensure that you don't affect the ability of your Manager to be copied.

Thank you for watching 