# Model Manager

A manager is usually something hidden away from django programmers that django uses to interface between model code and the database backend.

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.

Model code

Database Backend

Model Manager

When you query the django ORM, you do so through calls to

from my\_app.models import MyModel

mms = MyModel.objects.all()

In this case, the objects part of the function is what is returned by the manager. If you wanted MyModel to only ever get blue MyModel instances (the database might contain red models too) then you could create a manager and hack your model thus

class BlueManager(models.Manager):

def get\_query\_set(self):

return super(BlueManager, self).get\_query\_set().filter(colour='Blue')

class MyModel(models.Model):

colour = models.CharField(max\_length=64)

blue\_objects = BlueManager()

and calling

MyModel.blue\_objects.all()

would only return objects with colour as blue. Note, this is a very poor way to filter models!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Manager is some kind of **'gate' between model and database**. One of nice thing is that you can define your own base queryset for model. For example: if you have model 'Book' with 'availability' field, you can prepare own queryset, which filter specific kind of availability type:

class AvailableBookManager(models.Manager):

def get\_query\_set(self):

return super(AvailableBookManager,

self).get\_query\_set().filter(availability=1)

class Book(models.Model):

(...)#fields definition

objects = models.Manager() #default manager

available\_objects = AvailableBookManager() #own manager

and now you can use:

books = Book.available\_objects.all()

instead of:

books = Book.objects.filter(available=1)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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.

# **Managers:**

A Manager is the interface through which database query operations are provided to Django models. By default, [Django](https://micropyramid.com/django-development-services/) adds a Manager with the name “objects” to every Django model class.

# from django.db.models import Model, Manager

# class User(Model): .... .... objects = Manager() # The default Manager.

**Usage:**

User.objects.all() # This returns a list of users.

objects is a special attribute through which you query your database. We briefly identified this as the model’s manager, and it’s an instance of the class django.db.models.Manager; it’s where all the default methods for performing queries against the entire model class — all(), get(), filter(), etc.

A model’s manager is an object through which Django models perform database queries. Each Django model has at least one manager, and you can create custom managers in order to customise database access.

# **Custom Managers:**

Adding extra manager methods(custom managers) is the preferred way to add “table-level” functionality to your models whereas for “row-level” functionality use model methods.

**For example**, let’s give our Book model a manager method title\_count() that takes a keyword and returns the number of books that have a title containing that keyword.

# from django.db import models

# class BookManager(models.Manager):

# def title\_count(self, keyword):

# return self.filter(title\_\_icontains=keyword).count()

# class Book(models.Model):

# title = models.CharField(max\_length=100)

# publication\_date = models.DateField()

# num\_pages = models.IntegerField(blank=True, null=True)

# objects = BookManager() # The Custom Manager.

# def \_\_unicode\_\_(self):

# return self.title

With this manager, we can now do this:

# Book.objects.title\_count('django')

This will return number of books that have a title containing django. Note that the method uses self.filter() where self refers to the manager itself.

# **Model methods:**

Define custom methods on a model to add “row-level” functionality to your objects.

Here’s a model with a few custom methods:

# from django.db import models

# class Person(models.Model):

# first\_name = models.CharField(max\_length=50)

# last\_name = models.CharField(max\_length=50)

# def get\_full\_name(self):

# # Returns the person's full name.

# return '%s %s' % (self.first\_name, self.last\_name)

# full\_name = property(get\_full\_name)

# 

# def \_\_unicode\_\_(self):

# return self.full\_name

**Usage:**

# person = Person.objects.get(first\_name='xyz', last\_name='abc')

# person.full\_name

# # Note this isn't a method -- it's treated as an attribute, It returns 'xyz abc'

The method in this example is a “property.” Properties are a neat way to implement attributes whose usage resembles attribute access, but whose implementation uses method calls. These are sometimes known as “managed attributes”.