Theory: Queries and filters

20 minutes 0 / 5 problems solved

There is not much sense in having a database if you don't know how to get data from it. Let's find out how it works in Django!

First, you should get familiar with the Model object manager. We will use it to get and filter the data for a particular model. Once you learn the syntax rules, you'll be able to easily make queries to your database. It will give you the flexibility to retrieve any objects you want.

> Reading the data is the most common operation for a web application. The clients get data from the server more often than modify or delete it.

§1. Model Object Manager

An instance of the Model class represents a single row in the table of your database. To start working with a set of rows you should call the **Model Object Manager** methods.

The Manager is a special class to get object(s) from the database and modify them. To access the Manager of your model, you should get the attribute "*objects*" of the `Model` class.

Right now we are working on a *tournament* application for a Quidditch league. The season is coming, but the website is not ready yet! The wizards from Hogwarts get used to working with the books and papers but know nothing about databases. Fortunately, you don't need magic to start querying and searching. We create models `Team` and `Player` and that's how we define them:

```
from django.db import models


class Team(models.Model):
    name = models.CharField(max_length=64)


class Player(models.Model):
    height= models.FloatField()

    name = models.CharField(max_length=64)

    team = models.ForeignKey(Team, on_delete=models.CASCADE)




team_model_manager = Team.objects

player_model_manager = Player.objects
```

It's not necessary to give an alias name to the Manager, you can use its methods simply like this: `Team.objects.filter(name="Ballycastle Bats")`. You can choose what you like more, but for clarity, we will access it directly in all the examples.

This small snippet helps you fill the tables with the data:

Current topic:

Queries and filters

Topic depends on:

✓ Indexes 7★ (Stage 3)

✓ Dictionary methods (Stage 1)

✓ Exception handling 3★

✗ Django ORM

Topic is required for:

Field lookups

Object modification with ORM

Table of contents:

↑ Queries and filters

§1. Model Object Manager

§2. Get an Object

§3. Filtering Objects

§4. Conclusion

Feedback & Comments

```
falmouth_falcons = Team.objects.create(name="Falmouth Falcons")
montrose_magpies = Team.objects.create(name="Montrose Magpies")


Player.objects.create(name="Karl Broadmoor", height=180, team=falmouth_falcons)
Player.objects.create(name="Kevin Broadmoor", height=183, team=falmouth_falcons)
Player.objects.create(name="Alasdair Maddock", height=175, team=montrose_magpies)
Player.objects.create(name="Lennox Campbell", height=197, team=montrose_magpies)
```

Remember that you should migrate your models before using it!

§2. Get an Object

One step at a time, we will start from getting the team we want and then move on to getting a distinct player.

Unlike Python's `dict` *get* method, the Manager's *get* method may raise an `Exception`. You should keep in mind two rules:

- You can only pass the parameters with the names of the fields of your model or with valid field lookups;
- You should be sure that with this query you will get exactly one object.

We will carefully choose the parameters for our first query. Our `Team` model has two fields: *id* and *name*. The *id* field is generated automatically for every model, though we do not specify it explicitly.

We are sure that we have a team named "Falmouth Falcons". Let's try to get it with the Manager:

```
falcons = Team.objects.get(name="Falmouth Falcons")
```

Looks fine. But what happens if we get a nonexistent team?

```
tornados = Team.objects.get(name="Tutshill Tornados")
```

This call raises a `Team.DoesNotExist` exception. To prevent this situation and keep our program from crashing, you can wrap this call in *try-except* construction:

```
try:
    tornados = Team.objects.get(name="Tutshill Tornados")
except Team.DoesNotExist:
    ...
```

Let's try to get the "*Karl Broadmoor*" player account from the database:

```
karl_broadmoor = Player.objects.get(name="Karl Broadmoor")
```

Karl plays for Falmouth Falcons, so we get his account with no errors, but suppose you want to make a query that returns multiple objects:

```
falcons = Team.objects.get(name="Falmouth Falcons")
falcon_player = Player.objects.get(team=falcons)
```

You will not get a player, but a `Player.MultipleObjectsReturned` exception.

It seems that life is not that easy with the get queries. Sometimes we get an object, sometimes we get an error and we're never sure what happens next. Data may change and our valid call will start raising an `Exception`. You may turn to other Manager's methods and see what they can do for you.

§3. Filtering Objects

Like the standard Python *filter* function, the Manager's *filter* method returns only the objects that match the query. You don't have to know initially how many objects it will return, so it's safer than the *get* method.

The only rule is similar to the first rule for the *get* method:

- You can only pass parameters with names of the fields of your model or with valid field lookups.

Now we'll try to make our queries without fear of `DoesNotExist` and `MultipleObjectReturned` situations. We modify our call to:

```
tornados = Team.objects.filter(name="Tutshill Tornados")
```

Although we do not have Tornados in the database, no exception is raised. So what is the difference between these two methods? The answer is the return type. The *get* method always returns an instance of a particular model, while the *filter* method returns the **QuerySet**.

QuerySet is a wrapper for a set of objects in the database. QuerySet and Manager have much in common, so you can easily convert one into another. You can think of QuerySet as of another type of Manager.

To retrieve an object from the QuerySet you can iterate it over or get the item by the index as you get it from the Python's `list`.

```
tornados = Team.objects.filter(name="Tutshill Tornados")
if len(tornados) == 1:
    tornados_team = tornados[0]
```

This call is safe, so you can change the model and condition and it will still work.

Also, we want to get a "Falmouth Falcons" player. Let's do it with the combination of *filter* and *first* methods:

```
falcons = Team.objects.get(name="Falmouth Falcons")
falcon_player = Player.objects.filter(team=falcons).first()
```

Success!

> The last pitfall you should consider is that the *first* method does not raise any exceptions: if no objects are found, it returns `None`. So before accessing any properties of an object be sure that it's not `None`.

§4. Conclusion

It's likely that getting data from a database is an operation you will frequently use. We started polishing our skills by getting and filtering data. We found out how to retrieve a single object and a QuerySet to work with them as we work with other Python classes. However, the main purpose of Django is to provide the tools to make web services, and you can easily apply your query skills for doing analytics and reports.

Report a typo

44 users liked this theory. **3** didn't like it. **What about you?**

Start practicing

Comments (4) Hints (0) Useful links (0) Show discussion