Theory: Django ORM

24 minutes 0 / 5 problems solved

§1. Working With a Database From Python

Chances are, the storage you most often work with is a file system. It works well for HTML pages and templates, but how do you keep small objects like login, age or, say, favorite color for each individual person? Relational databases can help you organize and manipulate such data.

We will start from scratch and learn how to work with databases using only Python.

We define models to describe the schema of our data. To convert Python objects and primitives to database types, we will use adaptor classes, Fields. These abstractions help us pay less attention to the database specifics and focus mainly on what to store and how.

Once we declare the models and the fields in them, we create migrations and apply them to the SQLite3 database. Feel free to change it to another database backend. No matter which database you choose, our code will remain valid.

§2. Relational Databases

If your first thought is *"I need to keep the data with a common structure"*, then your second thought should surely be "databases".

A **relational database** is a collection of multiple data sets organized by tables, records, and columns. It works fine for most types of data. Each implementation provides you the universal language called structured query language (SQL). You can read about it, but as we said, we will work with the database in another way.

The most popular databases are PostgreSQL, Oracle SQL, MS SQL, and MySQL. There is also a simple database that works on your smartphone in many applications: it's called SQLite. It's perfect for one-client use and trying out Django models for the first time. Check whether you have it on your computer:

```
sqlite3 --version
```

If you don't, try to install it with your package manager or download it from the official site.

§3. Object-Relational Mapping

With the fall approaching and clouds getting denser, the new season of Quidditch is starting. As you know, wizards really lack computer science classes in Hogwarts, even though programming is a kind of magic. They want to store the teams, their results and the rosters on the website, and they wonder if there is a way to do it with Django. Well, there sure is! For this purpose, we will make the *quidditch* project with the *tournament* app in it. Let's meet and greet Django Models!

Django Models are classes that map the objects from the real world to the database records. We have teams, so we call our model the *Team*. This approach is called Object-Relational Mapping(ORM).

The ORM is a concept to map one type system to the other. We will work with databases by means of Python classes and methods. Our strong side is the programming language and we are going to make the most of it. The objects are similar to database records and their methods resemble SQL commands. There's no need to know SQL directly as we apply the instruments that imitate it.

Current topic:

Django ORM

Topic depends on:

- Object-Relational Mapping(ORM)
- Inheritance
- Datetime module (Stage 2)
- Django MVC (Stage 1)

Topic is required for:

- Queries and filters
- Using models with templates
- Registration and authentication

Table of contents:

To tell Django that it's a special class which maps its structure to the database table, we inherit the `Team` from `django.models.Model`. Also, we have players and game tables. Let's make the stubs for our classes in *tournament/models.py* module:

```
from django.db import models


class Team(models.Model):
    name = ...


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

    name = ...

    team = ...




class Game(models.Model):

    date = ...

    home_team = ...

    home_team_points = ...

    rival_team = ...

    rival_team_points = ...
```

We gave names to our classes and described their content. The restriction of all relational databases is that we should define the types for all the fields in the Model. So how can we match the types with the fields?

§4. Fields

To get most of the database's features, we use special *Fields* classes. They map the attribute of the class to a particular column in the database table. Does it mean we need the instance of a class for each field? Yes, but don't worry, it's actually easier than it may seem.

To build the whole schema, we start from the core element, the *Team:*

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

`CharField` is similar to Python string but has one restriction: the length limit. "*Wigtown Wanderers*" is the longest team name in the league now, but the league is still open to new teams, so we ensure `max_length` with 64 symbols.

Each team has players. Let's define a model for a player:

```
class Player(models.Model):
    height = models.FloatField()
    name = models.CharField(max_length=64)
    team = models.ForeignKey(Team, on_delete=models.CASCADE)
```

We already know what the `CharField` means, so the `FloatField` should sound familiar to you, too. It's the same as Python's `float` type. What's more interesting is the `ForeignKey` field. It means that the player is bound to a specific *Team* and the restriction `on_delete=models.CASCADE` means that if the Team is deleted from the database, it will be erased with all the players. That sounds unfair, but you should try harder to stay in the league!

```
class Game(models.Model):

    home_team = models.ForeignKey(Team, related_name='game_at_home', on_delete=models.CASCADE)
        home_team_points = models.IntegerField()

    rival_team = models.ForeignKey(Team, related_name='rival_game', on_delete=models.CASCADE)
        rival_team_points = models.IntegerField()
        date = models.DateField()
```

There are no games without teams, so again we set `on_delete=models.CASCADE` for each `ForeignKey`. Also, we add the `related_name` for the *Game* model, by which we can access it from the *Team* model. It's necessary to add such names because we have two foreign keys to the *Team* and you should differ one from another.

Points is an `int` type, so we make it `IntegerField`, and the date is clearly a `DateField`.

You can think of *Fields* as expansions of Python's primitive types for simple cases like `IntegerField`, `CharField`, and `FloatField`. They also have special cases like `ForeignKey` and other relations between objects.

§5. Migrations

At this point, we describe the mappings between Python classes and database tables, but we don't have any tables yet. That's sad news. Should we learn some fancy SQL to create a database and tables in it? No, because we can simply describe to Django what we want and it will do the dirty work for us — again.

Add tournament to `INSTALLED_APPS` in the *quidditch/settings.py* module:

```
INSTALLED_APPS = [
    # other installed apps
    'tournament',
]
```

We have the schema of the league in our code, we are ready to migrate it to the database. It takes two steps:

```
python manage.py makemigrations
python manage.py migrate
```

The first command creates migrations. Migration is a piece of code that describes what actions should be done in the database to synchronize the models with the tables. You can find the created code in the *tournament/migrations/0001_inititial.py* file.

In the second step, we apply the changes and run the generated commands.

> Preceding `manage.py <command>` with `python` is the platform-independent way to launch any django command. It's a valid syntax for both Unix and Windows systems.

If you want to make and then apply migrations to a particular application in your project, you should add the application name after each command:

```
python manage.py makemigrations tournament
python manage.py migrate tournament
```

When you run these commands, your database will finally have the tables to work with. We are ready to fill them with real data!

Report a typo

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

Start practicing

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