Models with Django Admin

Chapter 4

Objectives

- In this chapter we will use a database for the first time to build a basic Message Board application (called mbposts)where users can post and read short messages.
- We'll explore Django's powerful built-in admin interface which provides a visual way to make changes to our data.

Introduction

- For Django's MTV framework, where
  - **M stands for "Model,"**
  - **T stands for "Template,"**
  - **V stands for "View,"**

  we have covered the views function (Chapter 2) and templates (Chapter 3).
- In this Chapter, we will discuss Models for making database-driven websites

3

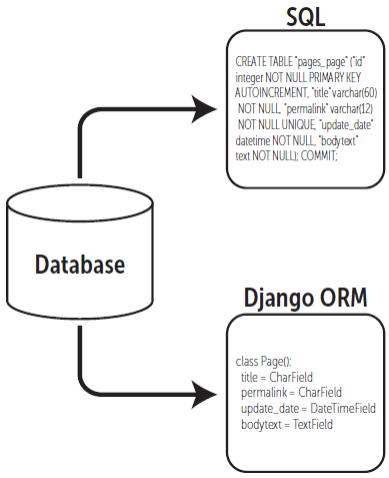
Introduction

- When you think of databases, you will usually think of the *Structured Query Language (SQL)*, the common means with which we query the database for the data we require.
- With Django, querying an underlying database - which can store all sorts of data, such as your website's user details – is taken care of by the *Object Relational Mapper (ORM)*.
- In essence, data stored within a database table can be encapsulated within a *model*.
- A model is a Python object that describes your database table's data. Instead of directly working on the database via SQL, you only need to manipulate the corresponding Python model object.
- This chapter walks you through the basics of data management with Django and its ORM.

4

Django Models

- Django's models provide an Object-relational Mapping (ORM) to the underlying database.
- Most common databases are programmed with some form of Structured Query Language (SQL), however each database implements SQL in its own way.
- An ORM tool on the other hand, provides a simple mapping between an *object* (the 'O' in ORM) and the underlying database, without the programmer needing to know the database structure, or requiring complex SQL to manipulate and retrieve data.



5

Database-driven websites

- Most modern web applications often involves interacting with a database.
- Behind the scenes, a database-driven website connects to a database server, retrieves some data out of it, and displays that data on a web page.
- The site might also provide ways for site visitors to populate the database on their own.

6

"Dumb" way to do database queries in views

- In this example view, we use the MySQLdb library to connect to a MySQL database, retrieve some records, and feed them to a template for display as a web page:

```
from django.shortcuts import render
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb',  passwd='secret',
host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render(request, 'book_list.html', {'names': names})
```

7

Problems of the previous approach

This approach works, but some problems should jump out at you immediately:

- We're hard-coding the database connection parameters. Ideally, these parameters would be stored in the Django configuration.
- We're having to write a fair bit of boilerplate code: creating a connection, creating a cursor, executing a statement, and closing the connection. Ideally, all we'd have to do is specify which results we wanted.
- It ties us to MySQL. If, down the road, we switch from MySQL to PostgreSQL, we'll most likely have to rewrite a large amount of our code. Ideally, the database server we're using would be abstracted, so that a database server change could be made in a single place. (This feature is particularly relevant if you're building an open-source Django application that you want to be used by as many people as possible.)

8

Defining Models in Python

- A Django model is a description of the data in your database, represented as Python code.
- It's your data layout–the equivalent of your SQL CREATE TABLE statements-except it's in Python instead of SQL, and it includes more than just database column definitions.
- SQL is inconsistent across database platforms. If you're distributing a web application, for example, it's much more pragmatic to distribute a Python module that describes your data layout than separate sets of CREATE TABLE statements for MySQL, PostgreSQL, and SQLite.
- To use Django's models, you must create a Django app. Models must live within apps. Hence, we'll create a new app called mbPosts for a Message Boardapplication.

Django: built-in support for database backends

- Django provides built-in support for several types of database backends.
- With just a few lines in our settings.py file it can support PostgreSQL, MySQL, Oracle, or SQLite.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

- But the simplest–by far–to use is SQLite because it runs off a single file and requires no complex installation.
- Django uses SQLite by default for this reason and it’s a perfect choice for small projects

Creating mbPosts app in the existing project

Our initial setup involves the following steps:

Step 1: create a new app called mbPosts

python manage.py startapp mbPosts

Step 2: update settings.py

Add the mbPosts app to our project under INSTALLED_APPS.

Step 3: Creating the database for the app

- Execute the migrate command to create an initial database based on Django's default settings.

  (django2) 05:34 ~/django_projects/mysite $ python manage.py migrate

- If you look inside the directory with the ls command, you'll see there's now a db.sqlite3 file representing our SQLite database.

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Step 4: Create a database model

- Our first task is to create a database model where we can store and display posts from our users.
- Open the posts/models.py file and look at the default code which Django provides.

```
# posts/models.py

from django.db import models

# Create your models here
```

Django imports a module models to build new database models, which will "model" the characteristics of the data in our database.

- We want to create a model to store the textual content of a message board post, which we can do so as follows:

```
class Post(models.Model):

    text = models.TextField()
```

- We've created a new database model called Post which has the database field text and the type of content it will hold is TextField().

Common field types

Field	Default Widget	Description
`AutoField`	N/A	An `IntegerField` that automatically increments according to available IDs.
`BigIntegerField`	`NumberInput`	A 64-bit integer, much like an `IntegerField` except that it is guaranteed to fit numbers from `-9223372036854775808` to `9223372036854775807`
`BinaryField`	N/A	A field to store raw binary data. It only supports `bytes` assignment. Be aware that this field has limited functionality.
`BooleanField`	`CheckboxInput`	A true/false field. If you need to accept `null` values then use `NullBooleanField` instead.
`CharField`	`TextInput`	A string field, for small- to large-sized strings. For large amounts of text, use `TextField`.`CharField` has one extra required argument: `max_length`. The maximum length (in characters) of the field.
`DateField`	`DateInput`	A date, represented in Python by a `datetime.date` instance. Has two extra, optional arguments: `auto_now` which automatically set the field to now every time the object is saved, and `auto_now_add` which automatically set the field to now when the object is first created.

Common field types (cont'd)

Field	Default Widget	Description
`FloatField`	`NumberInput`	A floating-point number represented in Python by a `float` instance. Note when `field.localize` is `False`, the default widget is `TextInput`
`DecimalField`	`TextInput`	A fixed-precision decimal number, represented in Python by a `Decimal` instance. Has two required arguments: `max_digits` and `decimal_places`.
`IntegerField`	`NumberInput`	An integer. Values from `-2147483648` to `2147483647` are safe in all databases supported by Django.
`PositiveIntegerField`	`NumberInput`	An integer . Values from `0` to `2147483647` are safe in all databases supported by Django.
`SmallIntegerField`	`NumberInput`	Like an `IntegerField`, but only allows values under a certain point. Values from `-32768` to `32767` are safe
`NullBooleanField`	`NullBooleanSelect`	Like a `BooleanField`, but allows `NULL` as one of the options.
`TextField`	`Textarea`	A large text field. If you specify a `max_length` attribute, it will be reflected in the `Textarea` widget of the auto-generated form field.

15

Step 5 & 6: Activating models

- Now that our new model is created, we need to activate it.
- Going forward, whenever we create or modify an existing model, we'll need to update Django in a two-step process.
  1. First we create a migration file with the makemigrations command which generate the SQL commands. Note that Migration files do not execute those commands on our database file, rather they are a reference of all new changes to our models. This approach means that we have a record of the changes to our models over time.
     (django2) 05:34 ~/django_projects/mysite $
     python manage.py makemigrations mbPosts

```
Migrations for 'posts':
  posts\migrations\0001_initial.py
    - Create model Post
```

  2. Second we build the actual database with migrate which executes the instructions in our migrations file.
     (django2) 05:34 ~/django_projects/mysite $
     python manage.py migrate mbPosts

```
Operations to perform:
  Apply all migrations: posts
Running migrations:
  Applying posts.0001_initial... OK
```

Make Migrations vs Migrate

- Makemigration will create the migration.
- A migration basically tells your database how it's being changed (i.e new column added, new table, dropped tables etc.).
  - Create the migrations: generate the SQL commands
- Migrate is what pushes your changes to your database. It will run all the migrations created(or the ones that haven't been pushed yet).
  - Run the migrations: execute the SQL commands
- It is necessary to run both the commands to complete the migration of the database tables to be in sync with your models.

17

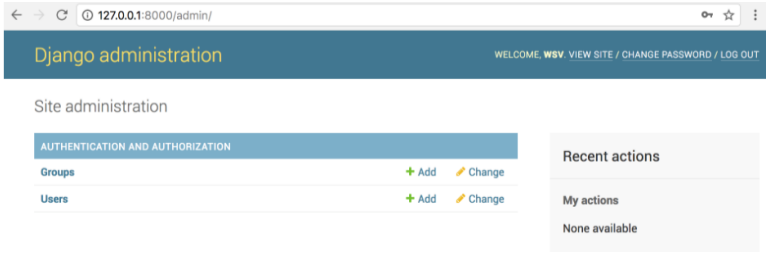
Step 7: Django Admin: create superuser

- Django provides us with a robust admin interface for interacting with our database.
- To use the Django admin, we first need to create a superuser.
- In your command line console, type the following command and respond to the prompts for a username, email, and password:

  (django2) 05:34 ~/django_projects/mysite $ python manage.py createsuperuser
- Login to Django Admin on the browser via yourusername.pythonanywhere.com/admin/ by entering the username and password you just created. You will see the Django admin homepage next.

Step 8: Django Admin: edit admin.py

- Our mbPosts app is not displayed on the main admin page!

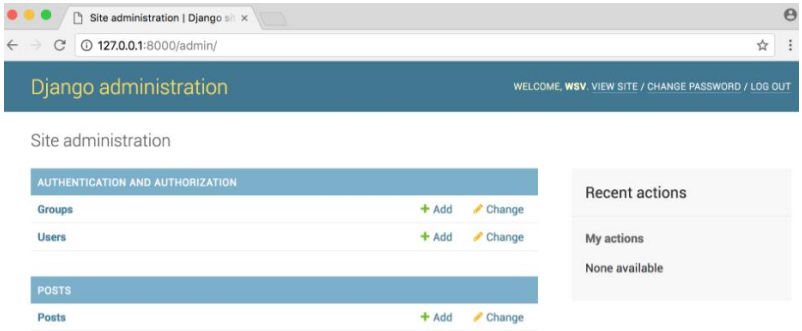


- We need to explicitly tell Django what to display in the admin.
- Edit mbPosts/admin.py file to look like this.

```
from django.contrib import admin
from .models import Post
admin.site.register(Post)
```

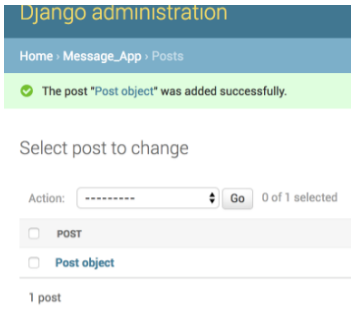

Admin Homepage updated

- Django now knows that it should display our posts app and its database model Post on the admin page. If you refresh your browser you'll see that it now appears.



Django Admin: Adding a new entry

- Now let's create our first message board post for our database.
- Click on the + Add button opposite Posts. Enter your own text in the Text form field.
- Then click the "Save" button, which will redirect you to the main Post page. However if you look closely, our new entry is called "Post object", which isn't very helpful.



str() method to improve readability of models

- Within the mbPosts/models.py file, add a new function __str__ as follows:

```
class Post(models.Model):
    text = models.TextField()
    def __str__(self):
        """A string representation of the model."""
        return self.text
```

- Refresh your Admin page in the browser, it has changed to a much more descriptive representation of our database entry.

Wrap up of Django Admin

- An **admin interface** is a web-based interface, limited to trusted site administrators, that enables the adding, editing and deletion of site content.
- Add a superuser with the command:
  python manage.py createsuperuser
- Edit admin.py to register the model(s) to be displayed in the Django Admin

Display database content on our webpage: Views/Templates/URLs

- In order to display our database content on our homepage, we have to wire up our views, templates, and URLConfs.
- Earlier in the book we used the built-in generic TemplateView to display a template file on our homepage.
- Now we want to list the contents of our database model with the generic class-based ListView.

Views with generic class-based ListView

- In the mbPosts/views.py file enter the Python code below:

```
from django.views.generic import ListView
from .models import Post
class HomePageView(ListView):
    model = Post
    template_name = 'home.html'
```

  - First, import ListView
  - In the second line we define which model we're using.
  - In the view, we subclass ListView, specify the model name and template reference.
  - Internally, ListView returns an object called object_list that we want to display in our template.

Templates (cont'd)

- In our templates file home.html we can use the Django Templating Language's for loop to list all the objects in object_list returned by ListView.

```
<!-- templates/home.html -->
<h1>Message board homepage</h1>
<ul>
  {% for post in object_list %}
    <li>{{ post }}</li>
  {% endfor %}
</ul>
```

URLConfs

- The last step is to set up our URLConfs. In the project-level urls.py file, include our posts and add include on the second line.
- Then create an app-level urls.py file (mbPosts/urls.py) to include the following code:

```
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('posts.urls')),
]
```

```
from django.urls import path
from . import views
urlpatterns = [
   path('', views.HomePageView.as_view(), name='home'),
]
```


Conclusion

- We've now built our first database-driven app. While it's deliberately quite basic, now we know how to create a database model, update it with the admin panel, and then display the contents on a web page. But something is missing……
- In the real-world, users need forms to interact with our site. After all, not everyone should have access to the admin panel.
- In the next chapter we'll build a blog application that uses forms so that users can create, edit, and delete posts.

Summary: what have you learnt?

- Create a database model for the app
- The makemigrations and migrate command
- Django's powerful built-in admin interface
- Use the built-in generic class-based ListView
- Use for loop to list all the objects in object_list returned by ListView.