Learning Objectives: Models

Learners will be able to...

- **Identify the types of databases that are compatible with Django**
- **Explain the purpose of the object-relational mapper (ORM)**
- **Update and synchronize databases and models**
- **Define Django Admin**

info

Make Sure You Know

You are familiar with Python classes and working with the command line.

Limitations

This project uses a single, simple database. All data entry is done with the Django Admin and not forms.

What are Models?

Models in Django

Databases are a foundational component to Django projects. There are many kinds of databases out there, but Django works with SQL databases like PostgreSQL, MariaDB, MySQL, Oracle, and SQLite. Even though all of these are SQL databases, they do not all function in the same way. Django helps free developers from writing database-specific code. The **object-relational mapper (ORM)** creates a stable interface for developers, their code, and their database.

▼ **How do databases store information?**

> Databases are a collection of tables. You can think of a table as a collections of rows and columns. Columns are called **fields**, which store a category of information. Rows are called **records**. Below is a representation of a table of automobiles. There are three fields, which contain the characteristics of the automobile — make, model, and color. A record can be thought of as the complete representation of an automobile stored in the database.
>
> 
>
>
> The image depicts a three by three grid of rectangles. The top row has the text “Make”, “Model”, and “Color”. The second row has the text “Ford”, “F150”, and “Red”. The third row has the text “Dodge”, “Ram”, and “Silver”. In red, you have the label “Field” for the columns and “Record” for the rows.

Instead of writing code for MariaDB or SQLite, you write your model independent of the database. The ORM then translates it to the correct SQL for the database being used with the project. By default, Django uses SQLite

for local development. We will continue to use the default database. However, there is a `DATABASES` section of the `settings.py` file where you can change the type of database used.

Many Django projects have several interconnected databases used to store information, which means they have several models. We, however, will only use a single model for our database. The models describes the information (both by name and by type) stored in the database. You can edit the model as you see fit, but you must first save these changes and then sync your database with the new model. These steps are discussed later.

Creating the Model

As our website is one about movie reviews, our model needs to reflect this. Every movie review has a movie title, some kind of rating, and some text describing the film. Our rating system will be one to four stars. We are going to add some additional information such as the year the film was released, who starred in it, and who directed it.

As mentioned above, the model represents information by name and by type. However, the traditional Python data types are not used. Categories of information in a database table are referred to as fields. Each field has a type. For example, several fields in our model will store a small to moderate amount of text. We are going to use the `CharField` to store the text (characters). Moreover, we can specify the maximum number of characters for these fields. So the field for the title should be smaller than the field for all of the actors.

Let's start by creating just the field for the movie title. First, activate the `django` virtual environment.

```
conda activate django
```

Open the `models.py` file in the `reviews` app with the link below.

Models are Python classes. Create the class `Review` which inherits from `models.Model`. Each attribute in the class represents a field in the database. Create the `title` attribute which represents a field in the database for storing movie titles. The field needs a type, which will be `CharField`. This field type is good for small to medium sized strings. However, we want to be more precise about this field. We want to limit these strings to 100 characters. Use the named parameter `max_length` and set its value to 100.

```
from django.db import models

class Review(models.Model):
  title = models.CharField(max_length=100)
```

Field types are important, because Django validates the data according to the model. If you put integers into the `title` field, Django will throw an error. It is a good idea to be precise when creating models.

Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black reviews/models.py
```

Deactivate the `django` virtual environment.

```
conda deactivate
```

Building the Model

Movie Fields

Start by activating the `django` virtual environment.

```
conda activate django
```

Open the `models.py` file.

Now that we have seen how to build models, let's add more fields to our database of movie reviews. Here is a representation of the fields needed for our model.

- Title - short-form text with a max length of 100 characters
- Director - short-form text with a max length of 100 characters
- Year - year only
- Actors - short-form text with a max length of 200 characters, as there are several actors per film
- Review - long-form text with no max length
- Stars - short-form text with a max length of 4

Create an attribute for each of the above categories. Use a `CharField` for short-form text, and set the maximum number of characters. Use a `TextField` for long-form text. We do not want to limit the length of a review, so there is no max length. A `PositiveSmallIntegerField` is sufficient for the year. The model should look like this:

```
from django.db import models

class Review(models.Model):
    title = models.CharField(max_length=100)
    director = models.CharField(max_length=100)
    year = models.PositiveSmallIntegerField()
    actors = models.CharField(max_length=200)
    review = models.TextField()
```

The `PositiveSmallIntegerField` accepts only integers from 0 to 32,767. This is the smallest integer field that Django provides. Django has a `DateField` which stores a `datetime` object. This works great if we want to note the date and time that the review was entered into the database. Using it for the year when a movie was released is a little more complicated. For the sake of simplicity we are going to use a `PositiveSmallIntegerField`.

Stars Field

You may have noticed that the number of stars is represented by a `CharField` and not a `PositiveSmallIntegerField`. We could easily take the integer representing the number of stars for a film and display it in a template. However, we want to display a unicode character (`&#9733;`) for the image of a star. If the film has four stars, we see four star images.

We already saw how Django lets you use template tags to insert some Python logic into a template. In fact, we will soon see a template tag for iterating over a sequence. You may think that since Django works with for loops, that there would be a template tag to iterate over the number of stars. It would be similar to this:

```
{% for i in range(review.stars) %}
  &#9733;
{% endfor %}
```

However, Django does not have a template tag that allows for this operation. The easiest way around this is to store the information as a string. Django will let you iterate over a string. So a review with one star will have a string with one character, a review with two stars will have a string with two characters, etc. Add the `stars` field with a max length of `4`.

```
from django.db import models

class Review(models.Model):
    title = models.CharField(max_length=100)
    director = models.CharField(max_length=100)
    year = models.PositiveSmallIntegerField()
    actors = models.CharField(max_length=200)
    review = models.TextField()
    stars = models.CharField(max_length=4)
```

Storing the stars as a `CharField` helps solve a programming problem, but it is not user friendly. We think of star ratings as being a number. What we want is a way to enter a number but have a string stored in the database. We can do exactly that with something called the `choices` field option. This shows the user a dropdown menu with predetermined values. When they select the value, a different valued is stored in the database.

Start by creating `STARS`, which is a tuple of tuples. Each inner tuple has two values. The first value is what is stored in the database. The second value is what is shown to the user. So `('s', 1)` means the user will see `1` as the choice but `s` will be stored in the database. Create a tuple for each of the star ratings. Then modify `stars` so that it still has a max length of 4, sets `choices` to `STARS` and has a default value of `1`.

```
    # previous code omitted
    STARS = (
      ('s', 1),
      ('ss', 2),
      ('sss', 3),
      ('ssss', 4),
    )
    stars = models.CharField(
      max_length=4,
      choices=STARS,
      default=1,
    )
```

Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black reviews/models.py
```

▼ **Code**

> Your code for the `Review` model should look like this:
>
> ```
> from django.db import models
>
> class Review(models.Model):
>     title = models.CharField(max_length=100)
>     director = models.CharField(max_length=100)
>     year = models.PositiveSmallIntegerField()
>     actors = models.CharField(max_length=200)
>     review = models.TextField()
>     STARS = (
>       ("s", 1),
>       ("ss", 2),
>       ("sss", 3),
>       ("ssss", 4),
>     )
>     stars = models.CharField(
>       max_length=4,
>       choices=STARS,
>       default=1,
>    )
> ```

Activating the Database

Because our website is so simple, this is all we need for our model. Before we use it, we need to activate the new model. This is a two-step process and should performed whenever the model changes. The first step is to create a migrations file. These files keep track of all of the changes made to the model over time.

```
python manage.py makemigrations reviews
```

▼ **Did you notice?**

> We specified the `reviews` app with the `makemigrations` command. This is optional. If you do not specify the app, Django will make migration files for all apps in the project.

Next, use the `migrate` command to sync the database to the model. We have already seen this command once before when we first activated the database with the Django defaults.

```
python manage.py migrate
```

Our database is ready to go. Next, we'll take a look at how to view and modify the database using the powerful, built-in tool called Django Admin.

Deactivate the virtual environment.

```
conda deactivate
```

Django Admin

Superuser Account

Start by activating the `django` virtual environment.

```
conda activate django
```

One of the biggest advantages to using Django is the built-in administration interface called Django Admin. If you remember back to editing the `urls.py` file for the Django project, this code was already in the file:

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

This means you can access the Django Admin by appending `/admin` to the URL after starting the dev server. This is a GUI that gives you an incredible amount of control without having to write any code. However, Django Admin requires you to log in to use it. We are going to create a super user account. This will be done via the terminal.

```
python manage.py createsuperuser
```

Python is going to ask you a series of questions for the account creation. We are going to use the following information for our superuser:

- Username - “Calvin”
- Email - “calvin@mail.com”
- Password “bestfriendhobbes”

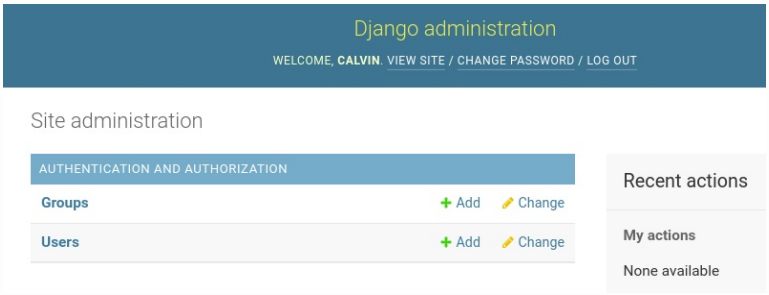
Python will not display the password as you type it in. Feel free to use your own information for the superuser account. **Important**, you need to remember the username and password so you can continue to login to the Django Admin.

Accessing the Django Admin

Once the superuser account has been successfully created, start the dev server.

```
python manage.py runserver 0.0.0.0:8000
```

Then navigate to the Django Admin by adding `/admin` to the end of our Django URL. Press `Enter` or `Return` on the keyboard. Login with your username and password. You will see an image like the one below. The Django Admin gives you the ability to modify users and groups for the Django project. All of this can be done with a mouse a keyboard, which is nice. However, there is no reference to the `review` model we just created.



The image depicts the Django Admin. You can chose from users and groups. There is no reference to the Review model.

Just as we must add the `reviews` app to the `INSTALLED_APPS` in the `settings.py` file, we need to register this model with the Django Admin. Open the `admin.py` file from the `reviews` app. Import the `Review` class from `models.py`. Then register the model with Django Admin.

```
from django.contrib import admin
from .models import Review

admin.site.register(Review)
```

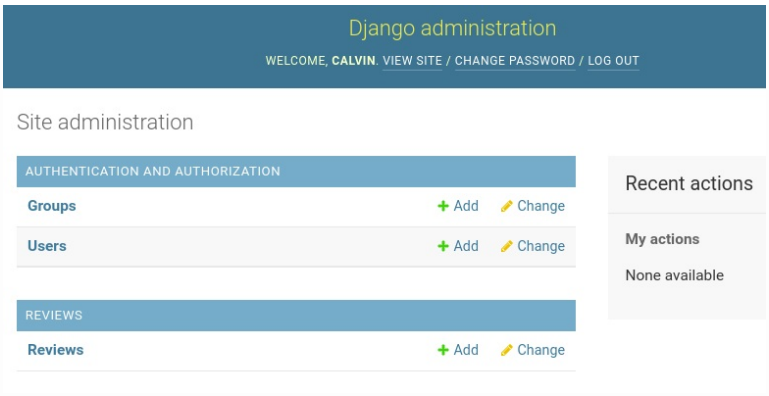
Open the terminal and run the Black formatter to make sure our code has a consistent style.

```
black reviews/admin.py
```

Run the dev server once again. Then open the Django Admin and login.

```
python manage.py runserver 0.0.0.0:8000
```

The `reviews` app is now present in the Django Admin. Once apps are visible in this tool, we can start to enter information into the database.



The image depicts the Django Admin. Below the section on users and groups, there is a section called “Reviews”. This is our Django app.

Switch back to the terminal and stop the dev server with `Ctrl+C` on the keyboard. Then deactivate the virtual environment.

```
conda deactivate
```

Adding to the Database

Creating a Review

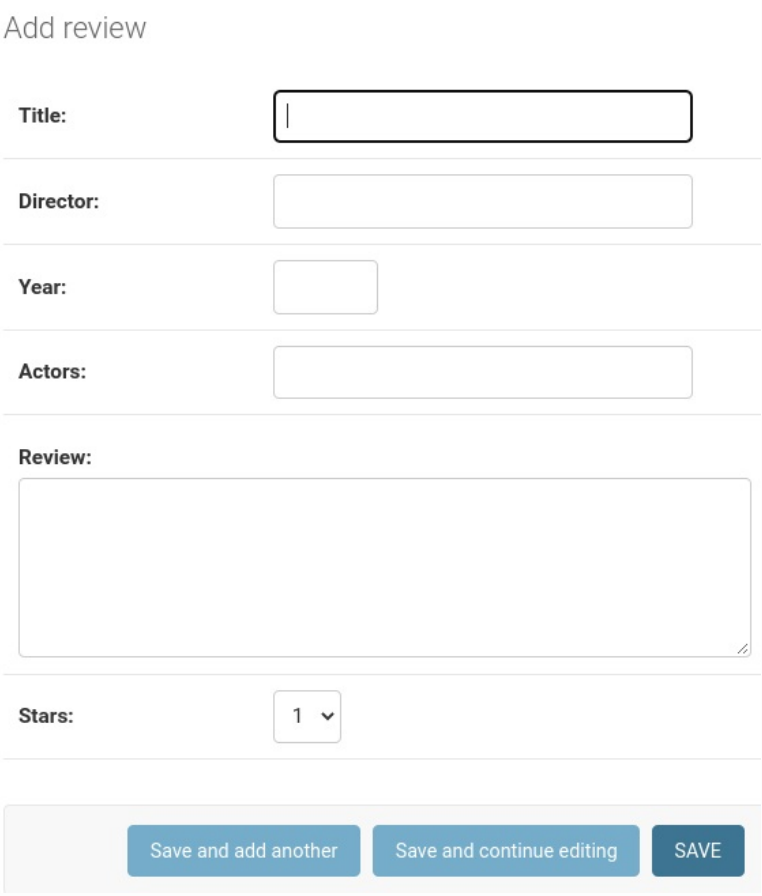
Start by activating the `django` virtual environment.

```
conda activate django
```

Run the dev server and open the Django Admin. Login once more. Click on the button `Add +` under the `Reviews` section to add a movie review to the database.

```
python manage.py runserver 0.0.0.0:8000
```

The Django Admin renders this as a form to fill out. For every field in the `Review` model, you have a text box or drop down menu in which you can enter information into the database. Again, all of this is handled automatically by the Django Admin.



The image depicts a form with all of the fields from the Review model. You have text boxes for all of the fields except for the number of stars. This is a drop down menu with a default value of 1.

▼ **Did you notice?**

> Most of the fields in our database are a `CharField` with a max length. The two exceptions are for the review itself and the number of stars. If you look carefully at the bottom-right corner of the text box for the review, you will see an icon to resize the area. That is because we are using a `TextField` which is for long-form content. The area for the stars is a drop down menu that has the numbers 1 to 4 in it. It defaults to 1.

Copy and paste the following information into the text boxes. Then click the `Save` button.

- Title:

```
The Big Lebowski
```

- Director:

```
Joel Cohen
```

- Year:

```
1998
```

- Actors:

```
Jeff Bridges, John Goodman, Julianne Moore, and Steve Buscemi
```

- Review:

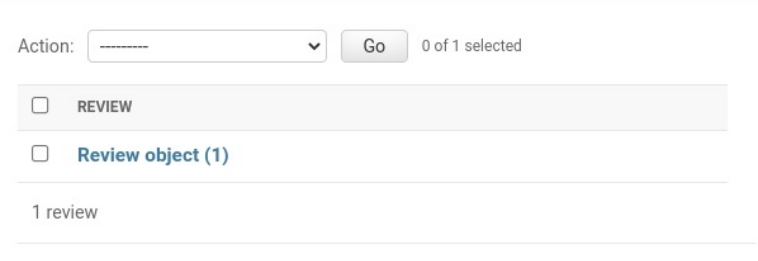
```
This comedy revolves around a case of kidnapping, mistaken
        identity, and bowling. The film has a cast of eccentric
        characters, matched by similar dialogue. Some of the
        jokes are not easy to catch on the first viewing, so it
        is definitely worth watching again. Initial response to
        the film's release was mixed. Over the years, however,
        The Big Lebowski has become a cult hit. Believe it or
        not, the character of The Dude is actually based on a
        real individual named Jeff Dowd.
```

- Stars:

```
4
```

Improving Readability

After saving the movie review, Django takes you to a list of all of the saved records in the database. From here, you can click on the record to make any changes to the information. However, the default name is not helpful. This is especially true when we add more movie reviews to the database.



The image depicts the saved movie review in the database. The default name is “Review object (1)”.

We can address this problem by adjusting our model. Open the `models.py` file and override the `__str__` magic method. This should return the `title` attribute.

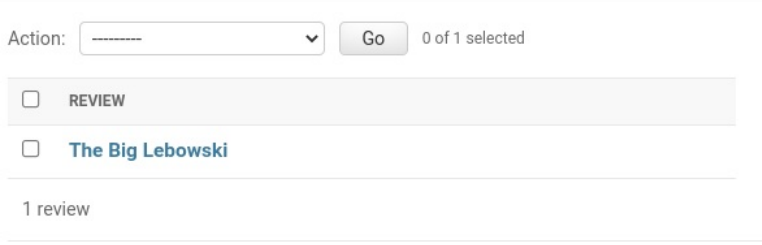
```
class Review(models.Model):
    title = models.CharField(max_length=100)
    director = models.CharField(max_length=100)
    year = models.PositiveSmallIntegerField()
    actors = models.CharField(max_length=200)
    review = models.TextField()
    STARS = (
        ("s", 1),
        ("ss", 2),
        ("sss", 3),
        ("ssss", 4),
    )
    stars = models.CharField(
        max_length=4,
        choices=STARS,
        default=1,
    )

    def __str__(self):
        return self.title
```

Go back to the Django Admin and refresh the page by clicking the icon with two blue arrows in the shape of a circle.

The image depicts two blue icons. One is in the shape of a triangle and the other two circles in the shape of a circle.

The default record name of `Review object(1)` is now replaced with the `The Big Lebowski`, the title of the movie. You should get in the habit of adding a `__str__` method to all of the models you create. This will help with readability.



The image depicts the list of records in the database. The previous title of “Review object(1)” has been replaced with “The Big Lebowski”.

▼ **Did you notice?**

> We did not use the `makemigrations` and `migrate` commands even though the model changed. That is because the fields, the actual information in the database, did not change. The **`str`** method only dictates how the record is displayed.

Switch back to the terminal and stop the dev server with `Ctrl+C` on the keyboard. Then deactivate the virtual environment.

```
conda deactivate
```

Multiple Records

Adding More Reviews

Start by activating the `django` virtual environment.

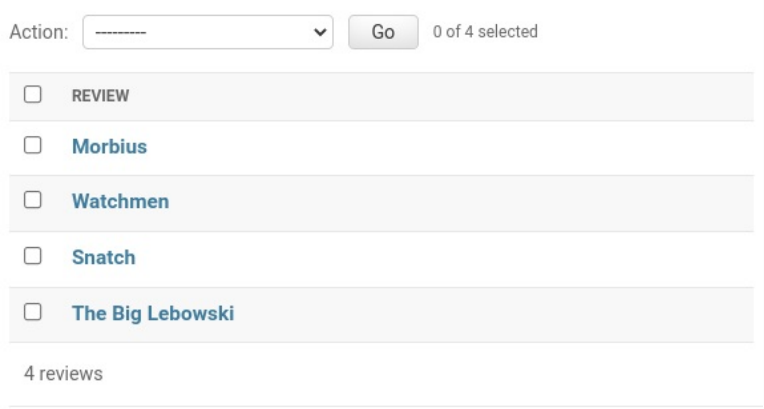
```
conda activate django
```

For this project to function as intended, we need a few more reviews in our database. We could manually create some more reviews. However, we are going to overwrite out database with another SQLite database that contains a total of four reviews.

```
cp .guides/new-db.sqlite3 db.sqlite3
```

Run the dev server and open the Django Admin. Click on the `Reviews` section. You should see four movie reviews in the database.

```
python manage.py runserver 0.0.0.0:8000
```

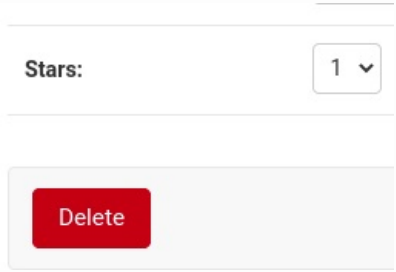


The image depicts the list of records in the database. There are now four reviews. The films are “Morbius”, “Watchmen”, “Snatch”, and “The Big Lebowski”.

Adding Your Own Reviews

Feel free to modify the database by adding reviews for any films you like (or dislike). You can even delete records from the database with Django Admin. Click on a review. You should see a red “Delete” button at the

bottom toward the left.



The image depicts a red button labeled as “Delete”. The button is below the drop down menu for stars.

The one thing we are unable to do is view the reviews outside Django Admin. To do this, we need to set up our views, templates, and URL patterns.

Switch back to the terminal and stop the dev server with `Ctrl+C` on the keyboard. Then deactivate the virtual environment.

```
conda deactivate
```