Defining Models in Python

Django’s models are written in Python and provide a simple mapping to the underlying database structure. Django uses a model to execute SQL behind the scenes to return Python data structures – which Django calls *QuerySets*.

Writing models in Python has several advantages:

* **Simplicity**. Writing Python is not only easier than writing SQL, but it’s less error prone and more efficient as your brain doesn’t have to keep switching from one language to another.
* **Consistency**. As I metioned earlier, SQL is inconsistent across different databases. You want to describe your data once, not create seperate sets of SQL statements for each database the application is likely to be deployed to.
* **Avoids Introspection**. Any application has to know something about the underlying database structure. There are only two ways to do that: have an explicit description of the data in the application code, or introspect the database at runtime. As introspection has a high overhead, and is not perfect, Django’s creators chose the first option.
* **Version Control**. Storing models in your codebase makes it easier to keep track of design changes.
* **Advanced Metadata**. Having models described in code rather than SQL allows for special datatypes (e.g., email addresses) as well as provide the ability to store much more metadata about a data layout than is generally provided by SQL.

There is a drawback in that the database can get out of sync with your models, but Django takes care of this problem with migrations. You’ve already used migrations in Chapter 2 when you created the My Club application. We’ll be using them again shortly when we create the Eventmodel.

You should also note that it is possible to introspect an existing database with Django using the inspectdb management command. We’ll be diving deeper into introspecting existing databases in Chapter [TODO].

Your First Model

Now that you have an idea what Django models are, it’s time to go ahead and create your first model.

The My Club website application includes an event planner. In the last chapter we created the events app to manage events within out My Club web application. If you haven’t created the events app yet, you need to go back to Page [TODO] and create it now.

**DB Browser for SQLite**

Throughout this chapter, and in a few other chapters, I am going to be using an application called *DB Browser for SQLite*. This is a great tool for checking out what is going on inside your SQLite database as Django changes things.

If you want to install DB Browser for SQLite on your own machine, you can download it from https://sqlitebrowser.org/.

There is lots of information we can record for a club event, but we’re going to start simple with a very basic event model. When you are first designing a data model, it’s always a good idea to map out the table fields before you create the model.

There are many different approaches to mapping out data structures – from simple tables to complex data maps using special markup. As you have probably worked out by now, my preference is to keep things as simple as possible, so I tend to just use tables (Table 4.1).

| Field Name | Field Description | Data Type |
| --- | --- | --- |
| name | Name of the event | Short Text |
| date | Date and time of the event | Date/Time |
| venue | Location of the event | Short Text |
| manager | Name of the person managing the event | Short Text |
| description | Detailed description of the event | Long Text |

**Table 4.1:** A simple mapping of our event model fields

This should be straight forward – we have database-friendly names for the fields, a description of the field and the type of data that will be saved to the field in the database. The description is for your benefit when you come back to the model later and need to remind yourself what the field was for. You can also put the field description in your model as comments or in the model’s docstring. We won’t be adding comments or docstrings to our code in this book; just keep it in mind for when you become a professional progammer – properly documenting your code is a Good Thing.

Now let’s turn our table into a Django model. Open the models.py file in your events folder and add the following model code:

# \myclub\_root\events\models.py

1 from django.db import models

2

3 class Event(models.Model):

4 name = models.CharField('Event Name', max\_length=120)

5 event\_date = models.DateTimeField('Event Date')

6 venue = models.CharField(max\_length=120)

7 manager = models.CharField(max\_length = 60)

8 description = models.TextField(blank=True)

Let’s have a closer look at your first model, as there’s a fair bit going on here:

* **Line 1** imports the models package from django.db. If you used startapp, this line will already be in your file.
* **Line 3** is the Event class definition. Each Django model must inherit from Django’s Modelclass.

Each of our model fields has a related Django *field type* and *field options*. The Event model uses three different field types – CharField, DateTimeField and TextField. Let’s have a look at the field types and options in more detail:

* **Line 4**. The name field is a Django CharField. A CharField is a short line of text (up to 255 characters). In this case, the max\_length option sets the maximum length of the event name to 120 characters. The name field also has a *verbose name* argument. The verbose name is used to create a human-friendly name for the model field. Most model fields accept verbose name, either as the first positional argument, or as a keyword argument (verbose\_name).
* **Line 5**. event\_date is a Django DateTimeField. A DateTimeField records a Python datetime object. The event\_date field has a single argument – the verbose name for the field. Note that I have named the field event\_date, rather than just date. This is because date is a reserved word in Python and, while Django will let you use reserved words in field names, it’s always best not to.
* **Lines 6 and 7**. venue and manager are both Django CharFields. As the max\_lengthargument is required on CharFields, they have both had their max\_length set to limit the size of the field.
* **Line 8**. description is a Django TextField. A TextField is a large text field that can hold many thousands of characters (maximum depends on the database). The final option – blank=True – is set so that we can create an event without a detailed description. The default for this option is False; if you don’t add a description, Django will throw an error.

This simple event model only uses a small subset of the model field types and options available in Django. We’ll be using many more throughout the book. A complete reference to all the model fields and options is in Appendix [TODO].

Now that we’ve created the model, it’s time to add it to the database. Make sure the virtual environment is running and then change into the myclub\_project directory. From your command prompt, run:

python manage.py makemigrations events

Hit enter, and then run:

python manage.py migrate events

Your terminal output should look something like this:

(env\_myclub) ...\myclub\_root> python manage.py makemigrations events

Migrations for 'events':

events\migrations\0001\_initial.py

- Create model Event

(env\_myclub) ...\myclub\_root> python manage.py migrate events

Operations to perform:

Apply all migrations: events

Running migrations:

Applying events.0001\_initial... OK

This is all you need to do to add your new model to the database. Before we go on though, remember how I said Django uses the model to generate SQL? Try this command at your command prompt:

python manage.py sqlmigrate events 0001\_initial

You should get an output that looks like this (I’ve reformatted for clarity):

BEGIN;

--

-- Create model Event

--

CREATE TABLE "events\_event" (

"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,

"name" varchar(120) NOT NULL,

"event\_date" datetime NOT NULL,

"venue" varchar(120) NOT NULL,

"manager" varchar(60) NOT NULL,

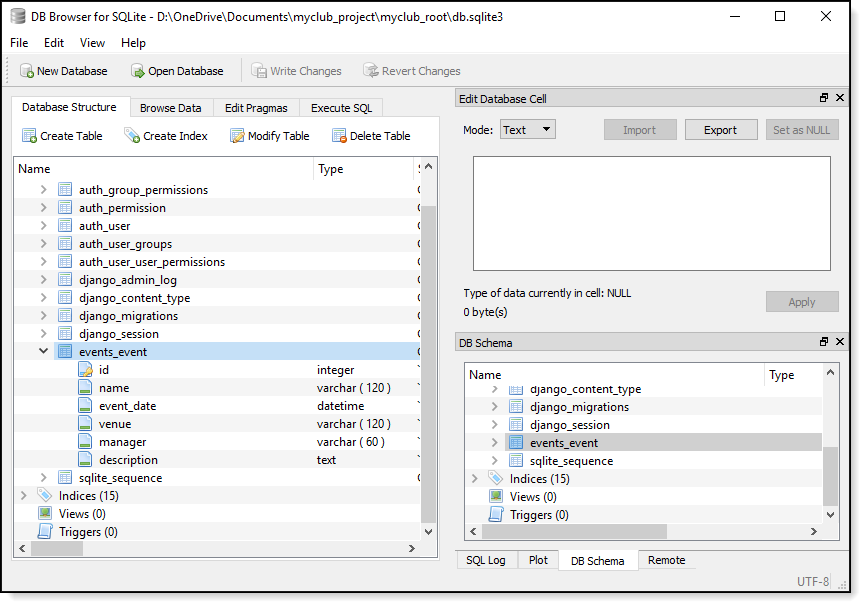
"description" text NOT NULL

);

COMMIT;

The sqlmigrate command prints the SQL for the named migration – in this case it’s printing out the SQL for the initial migration, where Django creates the new table and adds the fields to the table.

We can check out what Django created in the database browser (Figure 4.4).



**Figure 4.4:** The event table in the database

You can see Django’s app\_model naming convention for table names at work in the table name (events\_event). Also note how each of the model fields have been added as table fields with the appropriate data type applied to each field. If you are using DB Browser for SQLite, you will also notice that the schema for the table is almost the same as the SQL that the sqlmigrate command printed out.

Finally, note that the SQL, datatypes and database schema are different for each database type. This is what’s so cool about Django models – you don’t have to worry about the underlying database most of the time unless you need to do something unusual.

Basic Data Access

Django provides the four basic database functions – Create, Read, Update and Delete (CRUD) – that you would expect from a web framework designed for data-driven websites. Django, however, uses a high-level Python API to communicate with your database, rather than SQL.

To learn how to use Django’s database API, we’re going to be using the Django interactive shell. Django’s interactive shell runs just like the regular Python interactive shell, except it loads your project’s settings module and other Django-specific modules, so you can work directly with your Django project. To use the Django interactive shell, you must first be running the virtual environment, and then run the following command from inside your myclub\_root folder:

python manage.py shell

Your terminal output should look like this:

(env\_myclub) ...\myclub\_root> python manage.py shell

Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license" for more information.

(InteractiveConsole)

>>>

While this looks the same as the Python interactive shell, you can do things like this:

>>> from events.models import Event

If you tried to do this in the standard Python interactive shell, you get an error because Python has no knowledge of Django or your project.

Creating Database Records

To create a new event, you must first create a new event object:

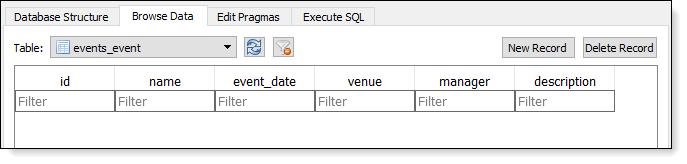
>>> from events.models import Event

>>> event1 = Event(name="Test Event1", event\_date="2018-12-17",

... venue="somewhere", manager="Bob")

>>>

You can see I have imported the Event model class and have created a new Event object and named it event1. Before we move on, have a look at the database (Figure 4.5). You’ll notice that the object hasn’t been saved to the database yet. This is by design. Accessing a database is expensive time-wise, so Django doesn’t hit the database until you explicitly tell Django to save the object.

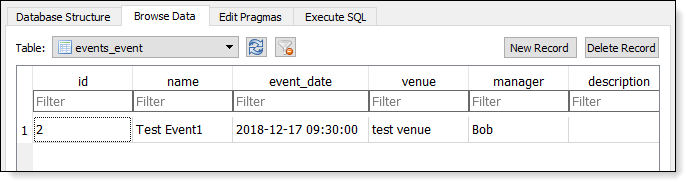


**Figure 4.5:** Django doesn’t hit the database until the record is saved.

To save a record to the database, Django has a model method that is conveniently called save(). So, let’s go ahead and save the record to the database:

>>> event1.save()

You will probably get a warning from Django about saving a date without a timezone, but you can ignore it. Now when you check the database, you can see that the record has now been added to the events\_event table in the database (Figure 4.6).



**Figure 4.6:** The record is added when you call the save() method.

Let’s go ahead now and create some more events:

>>> event2 = Event(name="Xmas Barbeque", event\_date="2018-12-23 12:00",

... venue="Notareal Park", manager="Bob")

>>> event2.save()

>>> event3 = Event(name="NYE Party", event\_date="2018-12-31 18:00",

... venue="McIvor's Bar", manager="Terry")

>>> event3.save()

Retrieving Records

Retrieve All Records

To retrieve all of the records in a database table, you use the all() method:

>>> event\_list = Event.objects.all()

>>> event\_list

<QuerySet [<Event: Event object (1)>, <Event: Event object (2)>, <Event: Event object (3)>]>

As you can see, this is not a very useful output – you have no way of knowing which event ‘Event Object’ is referring to. Luckily, Python has a special *string representation* method that can be added to Python functions and classes. Let’s go back to the Event model class declaration and add a couple of lines of code (changes in bold):

# \myclub\_root\events\models.py

1 from django.db import models

2

3 class Event(models.Model):

4 name = models.CharField('Event Name', max\_length=120)

5 event\_date = models.DateTimeField('Event Date')

6 venue = models.CharField(max\_length=120)

7 manager = models.CharField(max\_length = 60)

8 description = models.TextField(blank=True)

9

10 def \_\_str\_\_(self):

11 return self.name

The magic is in **lines 10 and 11**. The \_\_str\_\_ method generates a string representation of any Python object. For the Event model, we are simply returning the name of the event. You will have to close and restart the terminal for the changes to take effect, but after you have added the \_\_str\_\_method, your terminal output should look like this:

>>> from events.models import Event

>>> event\_list = Event.objects.all()

>>> event\_list

<QuerySet [<Event: Test Event1>, <Event: Xmas Barbeque>, <Event: NYE Party>]>

Much better.

Retrieve a Single Record

You retrieve single records with the get() method. You can retrieve a record using its primary key:

>>> Event.objects.get(id=1)

<Event: Test Event1>

You can also retrieve a record using one of the field names:

>>> Event.objects.get(name="Xmas Barbeque")

<Event: Xmas Barbeque>

However, the get() method only works for single objects. If your search term returns multiple records, you will get an error:

>>> Event.objects.get(manager="Bob")

Traceback (most recent call last):

# Lots of traceback info

... get() returned more than one Event -- it returned 2!

>>>

And if you try to retrieve a record that doesn’t exist, Django will throw a DoesNotExist error:

>>> Event.objects.get(id=999)

Traceback (most recent call last):

# Lots of traceback info

events.models.Event.DoesNotExist: Event matching query does not exist.

>>>

Retrieve Multiple Records

Whereas the get() method is limited to returning a single record, the filter() method allows you to filter your data to return zero or more records.

You can filter with a single search term:

>>> Event.objects.filter(manager="Bob")

<QuerySet [<Event: Test Event1>, <Event: Xmas Barbeque>]>

You can also filter with multiple search terms:

>>> Event.objects.filter(manager="Bob", venue="Notareal Park")

<QuerySet [<Event: Xmas Barbeque>]>

In both the above cases, Django translates the search terms into an SQL WHERE clause.

By default, the filter method uses exact match lookups. For more control, you can search using one of Django’s *field lookups*:

>>> Event.objects.filter(manager="Bob", name\_\_contains="Xmas")

<QuerySet [<Event: Xmas Barbeque>]>

Note that there’s a double underscore between name and contains. If you’re curious, the \_\_contains part gets translated by Django into an SQL LIKE statement. See chapter TODO for an in-depth review of field lookups.

If the search string doesn’t match, filter() will return an empty QuerySet:

>>> Event.objects.filter(manager="Fred")

<QuerySet []>

>>>

If you want to return all records in the database table with filter, you leave the query string blank:

>>> Event.objects.filter()

<QuerySet [<Event: Test Event1>, <Event: Xmas Barbeque>, <Event: NYE Party>]>

>>>

Ordering Data

You’ll notice from the previous examples that records are being retrieved from the database in table order. If you want to sort the records by a field in the table, you use the order\_by() method:

>>> Event.objects.order\_by("name")

<QuerySet [<Event: NYE Party>, <Event: Test Event1>, <Event: Xmas Barbeque>]>

If you want to sort in descending order, you add a minus (-) sign before the field name:

>>> Event.objects.order\_by("-name")

<QuerySet [<Event: Xmas Barbeque>, <Event: Test Event1>, <Event: NYE Party>]>

>>>

Internally, Django translates the order\_by() method into an SQL ORDER BY statement.

You can also sort by multiple fields:

>>> Event.objects.order\_by("manager", "name")

<QuerySet [<Event: Test Event1>, <Event: Xmas Barbeque>, <Event: NYE Party>]>

>>>

**Model Default Ordering**

You can see that this would get quite tedious if you wanted a default ordering for your model and had to use the order\_by() method every time you retrieved records from the database.

Django solves this issue with the ordering attribute of Django models’ class Metamethod. More about the class Meta method in Chapter TODO.

It’s very common to want to set a sort order on a subset of your database records. You achieve this in Django by *chaining lookups*. This is best illustrated with an example:

>>> Event.objects.filter(manager="Bob").order\_by("name")

<QuerySet [<Event: Test Event1>, <Event: Xmas Barbeque>]>

>>>

In this example, the filter() method first retrieves all events managed by Bob, and then the order\_by() method sorts them in alphabetical order by event name.

Slicing Data

You can also return a fixed number of rows from the database using Python’s list slicing syntax. This is handy for when you are paginating data, or want to return the the Top 10, say, of a QuerySet.

For example, you can retrieve the earliest event in the database:

>>> Event.objects.all().order\_by("event\_date")[0]

<Event: Test Event1>

Or the latest:

>>> Event.objects.all().order\_by("-event\_date")[0]

<Event: NYE Party>

You can even specify a subset of records to retrieve:

>>> Event.objects.all()[1:3]

<QuerySet [<Event: Xmas Barbeque>, <Event: NYE Party>]>

>>>

What you can’t do is use negative indexing. So, to retrieve Bob’s last event, this wont work:

>>> Event.objects.filter(manager="Bob")[-1]

Traceback (most recent call last):

# More traceback info

AssertionError: Negative indexing is not supported.

To get around this limitation, you use a reverse sort:

>>> Event.objects.filter(manager="Bob").order\_by("-event\_date")[0]

<Event: Xmas Barbeque>

>>>

Updating Records

Earlier in the chapter, we used Django’s database API to create a new record. Let’s go ahead and add another record now:

>>> event4 = Event(name="Bob's Birthday", event\_date="2019-01-26 15:00",

... venue="McIvor's Bar", manager="Terry")

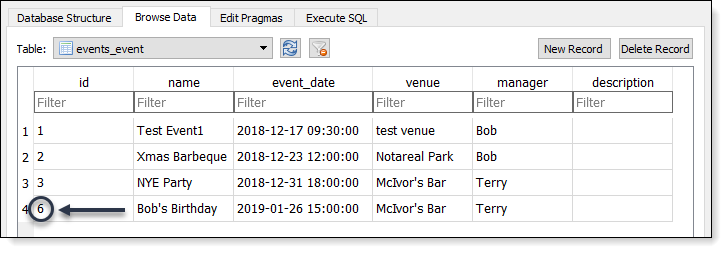
>>> event4.save()

You may have noticed that our Event model class doesn’t have a primary, or unique, key field defined. This is because Django creates a primary key automatically when it adds a record to the database. Django generates the primary key using the AUTOINCREMENT function of your database. In SQLite, AUTOINCREMENT returns an integer which you can access via the objects id field:

>>> event4.id

6 # Note that this will be different in your database

You can see in Figure 4.7 that this is the same key used by the database.



**Figure 4.7:** The Event object primary key is the same as the database record unique ID

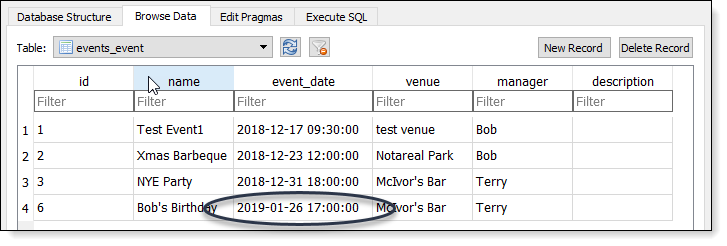
Subsequent calls to the save() method will save the record in place:

>>> event4.event\_date = "2019-01-26 17:00"

>>> event4.save()

>>>

If you check out the database table, you can see that the event data has been updated (Figure 4.8).



**Figure 4.8:** Calling the save() method updates the current record.

While this method of updating a database record is straight forward, it’s inefficient as it will save *all*the field values, not just the event date. To ensure records are updated in the most efficient manner possible, you should use the update() method. To re-write the above change to use the update()method, you would write:

>>> Event.objects.filter(id=6).update(event\_date="2019-01-26 17:00")

1

>>>

update() has an integer return value – the number of records updated.

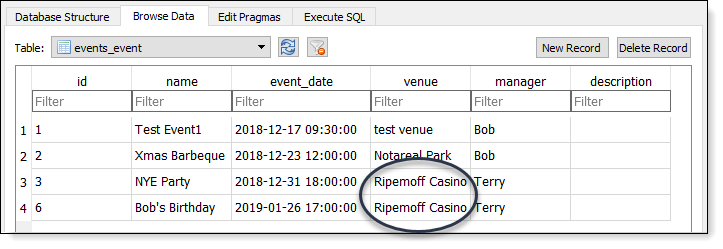
You can also use update to modify multiple records. For example, say you want to move all the events at McIvor’s Bar to Rippemoff Casino, you can use the follwing code:

>>> Event.objects.filter(venue="McIvor's Bar").update(venue="Ripemoff Casino")

2

>>>

You can see by the return value that update() has changed two records in the database. If you check the database, you will also see that all events at the bar have been moved to the casino (Figure 4.9).



**Figure 4.9:** The update() method makes updating multiple records easy.

Deleting Records

To delete an object from the database, you use the delete() method:

>>> Event.objects.filter(name\_\_contains="Test").delete()

(1, {'events.Event': 1})

>>>

The return value for the delete() method lists the total number of records affected (one in this example) and a dictionary listing all the tables that will be affected by the delete operation and the number of records deleted in each table.

You can delete multiple objects by using a filter that returns more than one record:

>>> Event.objects.filter(manager="Terry").delete()

(2, {'events.Event': 2})

>>>

And you can delete all the records in a table with the all() method:

>>> Event.objects.all().delete()

(1, {'events.Event': 1})

>>> Event.objects.all()

<QuerySet []>

>>>

To prevent you accidentally deleting all the data in a table, Django requires you to explicitly use the all() method to delete everything in a table. For example, this code doesn’t work:

>>> Event.objects.delete()

Traceback (most recent call last):

File "<console>", line 1, in <module>

AttributeError: 'Manager' object has no attribute 'delete'

>>>

Creating Relationships

In a data-driven website, it’s rare to have a table of information that is unrelated to information in other tables. This is largely due to well established database design best-practice.

Let’s take the venue field as an example. If you were just saving the venue name in the database, you could probably get away with repeating the name of the venue multiple times in your database records. But what about if you wanted to save more information for the venue?

To be of any use, your venue records would also need to include an address, telephone number, website address and email. If you added these fields to your events table, you can see that you’re going to end up with a lot of repeated information, not to mention the nightmare of ensuring you update all records if some of the venue information changes.

*Database normalization* is the process of designing your tables to minimize or eliminate data repetition. In simple term, normalization is the process of keeping related data in separate tables and linking tables via *relationships* (hence the name *relational* database).

Database normalization is also in keeping with Django’s Don’t Repeat Yourself (DRY) philosophy so, as you would expect, Django makes creating relationships between tables of related information simple.

Let’s make a change to our Event model and add a Venue model to our events app (changes in bold):

# \myclub\_root\events\models.py

1 class Venue(models.Model):

2 name = models.CharField('Venue Name', max\_length=120)

3 address = models.CharField(max\_length=300)

4 zip\_code = models.CharField('Zip/Post Code', max\_length=12)

5 phone = models.CharField('Contact Phone', max\_length=20)

6 web = models.URLField('Web Address')

7 email\_address = models.EmailField('Email Address')

8

9 def \_\_str\_\_(self):

10 return self.name

11

12

13 class Event(models.Model):

14 name = models.CharField('Event Name', max\_length=120)

15 event\_date = models.DateTimeField('Event Date')

16 venue = models.ForeignKey(Venue, on\_delete=models.CASCADE)

17 manager = models.CharField(max\_length = 60)

18 description = models.TextField(blank=True)

19

20 def \_\_str\_\_(self):

21 return self.name

The Venue model is very similar to the Event model, so you should find it easy to understand. Notice how the Venue model uses two new model fields: URLField and EmailField.

This is another cool feature of Django models – while at the database level these fields are no different to Django CharFields (they’re all saved as varchar‘s in SQLite), Django’s models provide built-in validation for special fields like URLs and email addresses.

There is one important change to the Event model – in **Line 16**, I have changed the venue field type from a CharField to a ForeignKey field. The first argument is the name of the related model. The on\_delete argument is required for foreign keys; CASCADE means that if a record is deleted, all related information in other tables will also be deleted.

Let’s go ahead and add our new model to the database:

(env\_myclub) ...\myclub\_root> py manage.py check

System check identified no issues (0 silenced).

(env\_myclub) ...\myclub\_root> py manage.py makemigrations events

Migrations for 'events':

events\migrations\0002\_auto\_20181218\_1346.py

- Create model Venue

- Alter field venue on event

(env\_myclub) ...\myclub\_root> py manage.py migrate

Operations to perform:

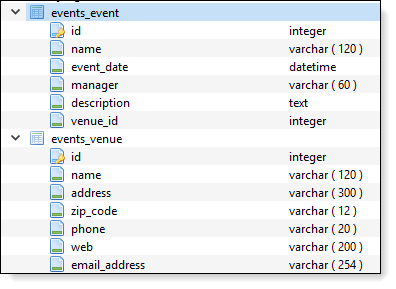
Apply all migrations: admin, auth, contenttypes, events, sessions

Running migrations:

Applying events.0002\_auto\_20181218\_1346... OK

(env\_myclub) ...\myclub\_root>

Once the migration has been applied to the database, you can see that the events\_venue table has been added and the venue field in the events\_event table is now an integer field (Figure 4.10).



**Figure 4.10:** Django has added a new table and created a relationship with the event table.

If you take a look at the event table schema (you can also do this with the sqlmigrate command you used earlier in the chapter) you can see the SQL Django uses to create the relationship:

CREATE TABLE "events\_event" (

"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,

"name" varchar(120) NOT NULL,

"event\_date" datetime NOT NULL,

"manager" varchar(60) NOT NULL,

"description" text NOT NULL,

"venue\_id" integer NOT NULL

REFERENCES "events\_venue" ("id")

DEFERRABLE INITIALLY DEFERRED

)

As I mentioned earlier, this SQL will be different for each database; the key takeaway here is that Django’s migrations take care of creating relationships in your database at the model level, without you having to care about how the underlying database is structured.