18

GETTING STARTED WITH DJANGO

Behind the scenes, today's websites are rich applications that act like fully developed desktop applications. Python has a great set of tools called Django for building web applications. Django is a *web framework*—a set of tools designed to help you build interactive websites. In this chapter, you'll learn how to use Django (*https://djangoproject.com/*) to build a project called Learning Log—an online journal system that lets you keep track of information you've learned about particular topics.

We'll write a specification for this project, and then we'll define models for the data the app will work with. We'll use Django's admin system to enter some initial data, and then you'll learn to write views and templates so Django can build the site's pages.

Django can respond to page requests and make it easier to read and write to a database, manage users, and much more. In Chapters 19 and 20, you'll refine the Learning Log project and then deploy it to a live server so you (and your friends) can use it.

Setting Up a Project

When beginning a project, you first need to describe the project in a specification, or *spec*. Then you'll set up a virtual environment in which to build the project.

Writing a Spec

A full spec details the project goals, describes the project's functionality, and discusses its appearance and user interface. Like any good project or business plan, a spec should keep you focused and help keep your project on track. We won't write a full project spec here, but we'll lay out a few clear goals to keep the development process focused. Here's the spec we'll use:

We'll write a web app called Learning Log that allows users to log the topics they're interested in and to make journal entries as they learn about each topic. The Learning Log home page will describe the site and invite users to either register or log in. Once logged in, a user can create new topics, add new entries, and read and edit existing entries.

When you learn about a new topic, keeping a journal of what you've learned can be helpful in tracking and revisiting information. A good app makes this process efficient.

Creating a Virtual Environment

To work with Django, we'll first set up a virtual environment. A *virtual environment* is a place on your system where you can install packages and isolate them from all other Python packages. Separating one project's libraries from other projects is beneficial and will be necessary when we deploy Learning Log to a server in Chapter 20.

Create a new directory for your project called *learning_log*, switch to that directory in a terminal, and enter the following code to create a virtual environment:

```
learning_log$ python -m venv ll_env
learning_log$
```

Here we're running the `venv` virtual environment module and using it to create a virtual environment named *ll_env* (note that this is *ll_env* with two lowercase *L*s, not two ones). If you use a command such as **`python3`** when running programs or installing packages, make sure to use that command here.

Activating the Virtual Environment

Now we need to activate the virtual environment using the following command:

```
  learning_log$ source ll_env/bin/activate
❶ (ll_env)learning_log$
```

This command runs the script *activate* in *ll_env/bin*. When the environment is active, you'll see the name of the environment in parentheses, as shown at ❶; then you can install packages to the environment and use packages that have already been installed. Packages you install in *ll_env* will be available only while the environment is active.

NOTE

If you're using Windows, use the command `ll_env\Scripts\activate` (without the word `source`) to activate the virtual environment. If you're using PowerShell, you might need to capitalize `Activate`.

To stop using a virtual environment, enter **`deactivate`**:

```
(ll_env)learning_log$ deactivate
learning_log$
```

The environment will also become inactive when you close the terminal it's running in.

Installing Django

Once the virtual environment is activated, enter the following to install Django:

```
(ll_env)learning_log$ pip install django
Collecting django
--snip--
```

```
Installing collected packages: pytz, django
Successfully installed django-2.2.0 pytz-2018.9 sqlparse-0.2.4
(ll_env)learning_log$
```

Because we're working in a virtual environment, which is its own self-contained environment, this command is the same on all systems. There's no need to use the `--user` flag, and there's no need to use longer commands, such as `python -m pip install` *`package_name`*.

Keep in mind that Django will be available only when the *ll_env* environment is active.

NOTE

Django releases a new version about every eight months, so you may see a newer version when you install Django. This project will most likely work as it's written here, even on newer versions of Django. If you want to make sure to use the same version of Django you see here, use the command `pip install django==2.2.*`. This will install the latest release of Django 2.2. If you have any issues related to the version you're using, see the online resources for the book at https://nostarch.com/pythoncrashcourse2e/.

Creating a Project in Django

Without leaving the active virtual environment (remember to look for *ll_env* in parentheses in the terminal prompt), enter the following commands to create a new project:

```
❶ (ll_env)learning_log$ django-admin startproject learning_log .
❷ (ll_env)learning_log$ ls
  learning_log ll_env manage.py
❸ (ll_env)learning_log$ ls learning_log
  __init__.py  settings.py  urls.py  wsgi.py
```

The command at ❶ tells Django to set up a new project called *learning_log*. The dot at the end of the command creates the new project with a directory structure that will make it easy to deploy the app to a server when we're finished developing it.

NOTE

Don't forget this dot, or you might run into some configuration issues when you deploy the app. If you forget the dot, delete the files and folders that were created (except ll_env*), and run the command again.*

Running the `ls` command (`dir` on Windows) ❷ shows that Django has created a new directory called *learning_log*. It also created a *manage.py* file, which is a short program that takes in commands and feeds them to the relevant part of Django to run them. We'll use these commands to manage tasks, such as working with databases and running servers.

The *learning_log* directory contains four files ❸; the most important are *settings.py*, *urls.py*, and *wsgi.py*. The *settings.py* file controls how Django interacts with your system and manages your project. We'll modify a few of these settings and add some settings of our own as the project evolves. The *urls.py* file tells Django which pages to build in response to browser requests. The *wsgi.py* file helps Django serve the files it creates. The filename is an acronym for *web server gateway interface*.

Creating the Database

Django stores most of the information for a project in a database, so next we need to create a database that Django can work with. Enter the following command (still in an active environment):

```
  (ll_env)learning_log$ python manage.py migrate
❶ Operations to perform:
    Apply all migrations: admin, auth, contenttypes, sessions
  Running migrations:
```

```
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  --snip--
  Applying sessions.0001_initial... OK
❷ (ll_env)learning_log$ ls
db.sqlite3  learning_log  ll_env  manage.py
```

Any time we modify a database, we say we're *migrating* the database. Issuing the `migrate` command for the first time tells Django to make sure the database matches the current state of the project. The first time we run this command in a new project using SQLite (more about SQLite in a moment), Django will create a new database for us. At ❶, Django reports that it will prepare the database to store information it needs to handle administrative and authentication tasks.

Running the `ls` command shows that Django created another file called *db.sqlite3* ❷. SQLite is a database that runs off a single file; it's ideal for writing simple apps because you won't have to pay much attention to managing the database.

NOTE

In an active virtual environment, use the command `python` to run `manage.py` commands, even if you use something different, like `python3`, to run other programs. In a virtual environment, the command `python` refers to the version of Python that created the virtual environment.

Viewing the Project

Let's make sure that Django has set up the project properly. Enter the `runserver` command as follows to view the project in its current state:

```
(ll_env)learning_log$ python manage.py runserver
Watchman unavailable: pywatchman not installed.
Watching for file changes with StatReloader
```

```
  Performing system checks...

❶ System check identified no issues (0 silenced).
  February 18, 2019 - 16:26:07
❷ Django version 2.2.0, using settings 'learning_log.settings'
❸ Starting development server at http://127.0.0.1:8000/
  Quit the server with CONTROL-C.
```

Django should start a server called the *development server*, so you can view the project on your system to see how well it works. When you request a page by entering a URL in a browser, the Django server responds to that request by building the appropriate page and sending it to the browser.

At ❶, Django checks to make sure the project is set up properly; at ❷ it reports the version of Django in use and the name of the settings file in use; and at ❸ it reports the URL where the project is being served. The URL *http://127.0.0.1:8000/* indicates that the project is listening for requests on port 8000 on your computer, which is called a localhost. The term *localhost* refers to a server that only processes requests on your system; it doesn't allow anyone else to see the pages you're developing.

Open a web browser and enter the URL *http://localhost:8000/*, or *http://127.0.0.1:8000/* if the first one doesn't work. You should see something like Figure 18-1, a page that Django creates to let you know all is working properly so far. Keep the server running for now, but when you want to stop the server, press CTRL-C in the terminal where the `runserver` command was issued.

Figure 18-1: Everything is working so far.

NOTE

If you receive the error message That port is already in use, *tell Django to use a different port by entering* `python manage.py runserver 8001`*, and then cycle through higher numbers until you find an open port.*

TRY IT YOURSELF

18-1. New Projects: To get a better idea of what Django does, build a couple of empty projects and look at what Django creates. Make a new folder with a simple name, like *snap_gram* or *insta_chat* (outside of your *learning_log* directory), navigate to that folder in a terminal, and create a virtual environment. Install Django and run the command `django-admin.py startproject snap_gram .` (make sure you include the dot at the end of the command).

Look at the files and folders this command creates, and compare them to Learning Log. Do this a few times until you're familiar with what Django creates when starting a new project. Then delete the project directories if you wish.

Starting an App

A Django *project* is organized as a group of individual *apps* that work together to make the project work as a whole. For now, we'll create just one app to do most of our project's work. We'll add another app in Chapter 19 to manage user accounts.

You should leave the development server running in the terminal window you opened earlier. Open a new terminal window (or tab), and navigate to the directory that contains *manage.py*. Activate the virtual environment, and then run the `startapp` command:

```
  learning_log$ source ll_env/bin/activate
  (ll_env)learning_log$ python manage.py startapp learning_logs
➊ (ll_env)learning_log$ ls
  db.sqlite3  learning_log  learning_logs  ll_env  manage.py
➋ (ll_env)learning_log$ ls learning_logs/
  __init__.py  admin.py  apps.py  migrations  models.py  tests.py  views.py
```

The command `startapp` *`appname`* tells Django to create the infrastructure needed to build an app. When you look in the project directory now, you'll see a new folder called *learning_logs* ➊. Open that folder to see what Django has created ➋. The most important files are *models.py*, *admin.py*, and *views.py*. We'll use *models.py* to define the data we want to manage in our app. We'll look at *admin.py* and *views.py* a little later.

Defining Models

Let's think about our data for a moment. Each user will need to create a number of topics in their learning log. Each entry they make will be tied to a topic, and these entries will be displayed as text. We'll also need to store the timestamp of each entry, so we can show users when they made each entry.

Open the file *models.py*, and look at its existing content:

models.py

```
from django.db import models

# Create your models here.
```

A module called `models` is being imported for us, and we're being invited to create models of our own. A *model* tells Django how to work with the data that will be stored in the app. Code-wise, a model is just a class; it has attributes and methods, just like every class we've discussed. Here's the model for the topics users will store:

```
  from django.db import models

  class Topic(models.Model):
      """A topic the user is learning about."""
❶     text = models.CharField(max_length=200)
❷     date_added = models.DateTimeField(auto_now_add=True)

❸     def __str__(self):
          """Return a string representation of the model."""
          return self.text
```

We've created a class called `Topic`, which inherits from `Model`—a parent class included in Django that defines a model's basic functionality. We add two attributes to the `Topic` class: `text` and `date_added`.

The `text` attribute is a `CharField`—a piece of data that's made up of characters, or text ❶. You use `CharField` when you want to store a small amount of text, such as a name, a title, or a city. When we define a `CharField` attribute, we have to tell Django how much space it should reserve in the database. Here we give it a `max_length` of 200 characters, which should be enough to hold most topic names.

The `date_added` attribute is a `DateTimeField`—a piece of data that will record a date and time ❷. We pass the argument `auto_now_add=True`, which tells

Django to automatically set this attribute to the current date and time whenever the user creates a new topic.

NOTE

To see the different kinds of fields you can use in a model, see the Django Model Field Reference *at* https://docs.djangoproject.com/en/2.2/ref/models/fields/. *You won't need all the information right now, but it will be extremely useful when you're developing your own apps.*

We tell Django which attribute to use by default when it displays information about a topic. Django calls a `__str__()` method to display a simple representation of a model. Here we've written a `__str__()` method that returns the string stored in the `text` attribute ❸.

Activating Models

To use our models, we have to tell Django to include our app in the overall project. Open *settings.py* (in the *learning_log/learning_log* directory); you'll see a section that tells Django which apps are installed and work together in the project:

settings.py

```
--snip--
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
--snip--
```

Add our app to this list by modifying `INSTALLED_APPS` so it looks like this:

```
--snip--
INSTALLED_APPS = [
    # My apps
    'learning_logs',

    # Default django apps.
    'django.contrib.admin',
    --snip--
]
--snip--
```

Grouping apps together in a project helps to keep track of them as the project grows to include more apps. Here we start a section called *My apps*, which includes only `learning_logs` for now. It's important to place your own apps before the default apps in case you need to override any behavior of the default apps with your own custom behavior.

Next, we need to tell Django to modify the database so it can store information related to the model `Topic`. From the terminal, run the following command:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
  learning_logs/migrations/0001_initial.py
    - Create model Topic
(ll_env)learning_log$
```

The command `makemigrations` tells Django to figure out how to modify the database so it can store the data associated with any new models we've defined. The output here shows that Django has created a migration file called *0001_initial.py*. This migration will create a table for the model `Topic` in the database.

Now we'll apply this migration and have Django modify the database for us:

```
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
❶   Applying learning_logs.0001_initial... OK
```

Most of the output from this command is identical to the first time we issued the `migrate` command. The line we need to check appears at ❶, where Django confirms that the migration for `learning_logs` worked `OK`.

Whenever we want to modify the data that Learning Log manages, we'll follow these three steps: modify *models.py*, call `makemigrations` on `learning_logs`, and tell Django to `migrate` the project.

The Django Admin Site

Django makes it easy to work with your models through the *admin site*. Only the site's administrators use the admin site, not general users. In this section, we'll set up the admin site and use it to add some topics through the `Topic` model.

Setting Up a Superuser

Django allows you to create a *superuser*, a user who has all privileges available on the site. A user's *privileges* control the actions that user can take. The most restrictive privilege settings allow a user to only read public information on the site. Registered users typically have the privilege of reading their own private data and some selected information available only to members. To effectively administer a web application, the site owner usually needs access to all information stored on the site. A good administrator is careful with their users' sensitive information, because users put a lot of trust into the apps they access.

To create a superuser in Django, enter the following command and respond to the prompts:

```
  (ll_env)learning_log$ python manage.py createsuperuser
➊ Username (leave blank to use 'eric'): ll_admin
➋ Email address:
➌ Password:
  Password (again):
  Superuser created successfully.
  (ll_env)learning_log$
```

When you issue the command `createsuperuser`, Django prompts you to enter a username for the superuser ➊. Here I'm using *ll_admin*, but you can enter any username you want. You can enter an email address if you want or just leave this field blank ➋. You'll need to enter your password twice ➌.

NOTE

Some sensitive information can be hidden from a site's administrators. For example, Django doesn't store the password you enter; instead, it stores a string derived from the password, called a hash. *Each time you enter your password, Django hashes your entry and compares it to the stored hash. If the two hashes match, you're authenticated. By requiring hashes to match, if an attacker gains access to a site's database, they'll be able to read its stored hashes but not the passwords. When a site is set up properly, it's almost impossible to get the original passwords from the hashes.*

Registering a Model with the Admin Site

Django includes some models in the admin site automatically, such as `User` and `Group`, but the models we create need to be added manually.

When we started the `learning_logs` app, Django created an *admin.py* file in the same directory as *models.py*. Open the *admin.py* file:

admin.py

```
from django.contrib import admin

# Register your models here.
```

To register `Topic` with the admin site, enter the following:

```
  from django.contrib import admin

❶ from .models import Topic

❷ admin.site.register(Topic)
```

This code first imports the model we want to register, `Topic` ❶. The dot in front of `models` tells Django to look for *models.py* in the same directory as *admin.py*. The code `admin.site.register()` tells Django to manage our model through the admin site ❷.

Now use the superuser account to access the admin site. Go to *http://localhost:8000/admin/*, and enter the username and password for the superuser you just created. You should see a screen like the one in Figure 18-2. This page allows you to add new users and groups, and change existing ones. You can also work with data related to the `Topic` model that we just defined.

Figure 18-2: The admin site with `Topic` included

NOTE

If you see a message in your browser that the web page is not available, make sure you still have the Django server running in a terminal window. If you don't, activate a virtual environment and reissue the command `python manage.py runserver`. If you're having trouble viewing your project at any point in the development process, closing any open terminals and reissuing the `runserver` command is a good first troubleshooting step.

Adding Topics

Now that `Topic` has been registered with the admin site, let's add our first topic. Click **Topics** to go to the Topics page, which is mostly empty, because we have no topics to manage yet. Click **Add Topic**, and a form for adding a new topic appears. Enter **`Chess`** in the first box and click **Save**. You'll be sent back to the Topics admin page, and you'll see the topic you just created.

Let's create a second topic so we'll have more data to work with. Click **Add Topic** again, and enter **`Rock Climbing`**. Click **Save**, and you'll be sent back to the main Topics page again. Now you'll see Chess and Rock Climbing listed.

Defining the Entry Model

For a user to record what they've been learning about chess and rock climbing, we need to define a model for the kinds of entries users can make in their learning logs. Each entry needs to be associated with a particular topic. This relationship is called a *many-to-one relationship*, meaning many entries can be associated with one topic.

Here's the code for the `Entry` model. Place it in your *models.py* file:

models.py

```
  from django.db import models

  class Topic(models.Model):
      --snip--

❶ class Entry(models.Model):
      """Something specific learned about a topic."""
❷     topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
❸     text = models.TextField()
      date_added = models.DateTimeField(auto_now_add=True)

❹     class Meta:
          verbose_name_plural = 'entries'

      def __str__(self):
          """Return a string representation of the model."""
❺         return f"{self.text[:50]}..."
```

The `Entry` class inherits from Django's base `Model` class, just as `Topic` did ❶. The first attribute, `topic`, is a `ForeignKey` instance ❷. A *foreign key* is a database term; it's a reference to another record in the database. This is the code that connects each entry to a specific topic. Each topic is assigned a key, or ID, when it's created. When Django needs to establish a connection between two pieces of data, it uses the key associated with each piece of information. We'll use these connections shortly to retrieve all the entries associated with a certain topic. The `on_delete=models.CASCADE` argument tells Django that when a topic is deleted, all the entries associated with that topic should be deleted as well. This is known as a *cascading delete*.

Next is an attribute called `text`, which is an instance of `TextField` ❸. This kind of field doesn't need a size limit, because we don't want to limit the size of individual entries. The `date_added` attribute allows us to present entries in the order they were created and to place a timestamp next to each entry.

At ❹ we nest the `Meta` class inside our `Entry` class. The `Meta` class holds extra information for managing a model; here, it allows us to set a special attribute telling Django to use *Entries* when it needs to refer to more than one entry. Without this, Django would refer to multiple entries as *Entrys*.

The `__str__()` method tells Django which information to show when it refers to individual entries. Because an entry can be a long body of text, we tell Django to show just the first 50 characters of `text` ❺. We also add an ellipsis to clarify that we're not always displaying the entire entry.

Migrating the Entry Model

Because we've added a new model, we need to migrate the database again. This process will become quite familiar: you modify *models.py*, run the command `python manage.py makemigrations` *`app_name`*, and then run the command `python manage.py migrate`.

Migrate the database and check the output by entering the following commands:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
❶ learning_logs/migrations/0002_entry.py
```

```
  - Create model Entry
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  --snip--
❷ Applying learning_logs.0002_entry... OK
```

A new migration called *0002_entry.py* is generated, which tells Django how to modify the database to store information related to the model `Entry` ❶. When we issue the `migrate` command, we see that Django applied this migration, and everything was okay ❷.

Registering Entry with the Admin Site

We also need to register the `Entry` model. Here's what *admin.py* should look like now:

admin.py

```
from django.contrib import admin

from .models import Topic, Entry

admin.site.register(Topic)
admin.site.register(Entry)
```

Go back to *http://localhost/admin/*, and you should see *Entries* listed under *Learning_Logs*. Click the **Add** link for Entries, or click **Entries**, and then choose **Add entry**. You should see a drop-down list to select the topic you're creating an entry for and a text box for adding an entry. Select **Chess** from the drop-down list, and add an entry. Here's the first entry I made:

The opening is the first part of the game, roughly the first ten moves or so. In the opening, it's a good idea to do three things—bring out your bishops and knights, try to control the center of the board, and castle your king.

Of course, these are just guidelines. It will be important to learn when to follow these guidelines and when to disregard these suggestions.

When you click **Save**, you'll be brought back to the main admin page for entries. Here, you'll see the benefit of using `text[:50]` as the string representation for each entry; it's much easier to work with multiple entries in the admin interface if you see only the first part of an entry rather than the entire text of each entry.

Make a second entry for Chess and one entry for Rock Climbing so we have some initial data. Here's a second entry for Chess:

In the opening phase of the game, it's important to bring out your bishops and knights. These pieces are powerful and maneuverable enough to play a significant role in the beginning moves of a game.

And here's a first entry for Rock Climbing:

One of the most important concepts in climbing is to keep your weight on your feet as much as possible. There's a myth that climbers can hang all day on their arms. In reality, good climbers have practiced specific ways of keeping their weight over their feet whenever possible.

These three entries will give us something to work with as we continue to develop Learning Log.

The Django Shell

With some data entered, we can examine that data programmatically through an interactive terminal session. This interactive environment is called the Django *shell*, and it's a great environment for testing and troubleshooting your project. Here's an example of an interactive shell session:

```
  (ll_env)learning_log$ python manage.py shell
❶ >>> from learning_logs.models import Topic
```

```
>>> Topic.objects.all()
<QuerySet [<Topic: Chess>, <Topic: Rock Climbing>]>
```

The command `python manage.py shell`, run in an active virtual environment, launches a Python interpreter that you can use to explore the data stored in your project's database. Here, we import the model `Topic` from the `learning_logs.models` module ❶. We then use the method `Topic.objects.all()` to get all the instances of the model `Topic`; the list that's returned is called a *queryset*.

We can loop over a queryset just as we'd loop over a list. Here's how you can see the ID that's been assigned to each topic object:

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
...  print(topic.id, topic)
...
1 Chess
2 Rock Climbing
```

We store the queryset in `topics`, and then print each topic's `id` attribute and the string representation of each topic. We can see that Chess has an ID of 1, and Rock Climbing has an ID of 2.

If you know the ID of a particular object, you can use the method `Topic.objects.get()` to retrieve that object and examine any attribute the object has. Let's look at the `text` and `date_added` values for Chess:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Chess'
>>> t.date_added
datetime.datetime(2019, 2, 19, 1, 55, 31, 98500, tzinfo=<UTC>)
```

We can also look at the entries related to a certain topic. Earlier we defined the `topic` attribute for the `Entry` model. This was a `ForeignKey`, a connection between each entry and a topic. Django can use this connection to get every entry related to a certain topic, like this:

```
❶ >>> t.entry_set.all()
  <QuerySet [<Entry: The opening is the first part of the game, roughly...>,
  <Entry:
  In the opening phase of the game, it's important t...>]>
```

To get data through a foreign key relationship, you use the lowercase name of the related model followed by an underscore and the word `set` ❶. For example, say you have the models `Pizza` and `Topping`, and `Topping` is related to `Pizza` through a foreign key. If your object is called `my_pizza`, representing a single pizza, you can get all of the pizza's toppings using the code `my_pizza.topping_set.all()`.

We'll use this kind of syntax when we begin to code the pages users can request. The shell is very useful for making sure your code retrieves the data you want it to. If your code works as you expect it to in the shell, you can expect it to work properly in the files within your project. If your code generates errors or doesn't retrieve the data you expect it to, it's much easier to troubleshoot your code in the simple shell environment than within the files that generate web pages. We won't refer to the shell much, but you should continue using it to practice working with Django's syntax for accessing the data stored in the project.

NOTE

Each time you modify your models, you'll need to restart the shell to see the effects of those changes. To exit a shell session, press CTRL*-D; on Windows, press* CTRL*-Z and then press* ENTER*.*

TRY IT YOURSELF

18-2. Short Entries: The `__str__()` method in the `Entry` model currently appends an ellipsis to every instance of `Entry` when Django shows it in the admin site or the shell. Add an `if` statement to the `__str__()` method that adds an ellipsis only if the entry is longer than 50 characters. Use the admin site to add an entry that's fewer than 50 characters in length, and check that it doesn't have an ellipsis when viewed.

18-3. The Django API: When you write code to access the data in your project, you're writing a *query*. Skim through the documentation for querying your data at *https://docs.djangoproject.com/en/2.2/topics/db/queries/*. Much of what you see will look new to you, but it will be very useful as you start to work on your own projects.

18-4. Pizzeria: Start a new project called `pizzeria` with an app called `pizzas`. Define a model `Pizza` with a field called `name`, which will hold name values, such as `Hawaiian` and `Meat Lovers`. Define a model called `Topping` with fields called `pizza` and `name`. The `pizza` field should be a foreign key to `Pizza`, and `name` should be able to hold values such as `pineapple`, `Canadian bacon`, and `sausage`.

Register both models with the admin site, and use the site to enter some pizza names and toppings. Use the shell to explore the data you entered.

Making Pages: The Learning Log Home Page

Making web pages with Django consists of three stages: defining URLs, writing views, and writing templates. You can do these in any order, but in this project we'll always start by defining the URL pattern. A URL pattern describes the way the URL is laid out. It also tells Django what to look for when matching a browser request with a site URL so it knows which page to return.

Each URL then maps to a particular *view*—the view function retrieves and processes the data needed for that page. The view function often renders the page using a *template*, which contains the overall structure of the page. To see how this works, let's make the home page for Learning Log.

We'll define the URL for the home page, write its view function, and create a simple template.

Because all we're doing is making sure Learning Log works as it's supposed to, we'll make a simple page for now. A functioning web app is fun to style when it's complete; an app that looks good but doesn't work well is pointless. For now, the home page will display only a title and a brief description.

Mapping a URL

Users request pages by entering URLs into a browser and clicking links, so we'll need to decide what URLs are needed. The home page URL is first: it's the base URL people use to access the project. At the moment the base URL, *http://localhost:8000/*, returns the default Django site that lets us know the project was set up correctly. We'll change this by mapping the base URL to Learning Log's home page.

In the main *learning_log* project folder, open the file *urls.py*. Here's the code you should see:

urls.py

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

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

The first two lines import a module and a function to manage URLs for the admin site ➊. The body of the file defines the `urlpatterns` variable ➋. In this *urls.py* file, which represents the project as a whole, the `urlpatterns` variable includes sets of URLs from the apps in the project. The code at ➌ includes the module `admin.site.urls`, which defines all the URLs that can be requested from the admin site.

We need to include the URLs for `learning_logs`, so add the following:

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

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

We've added a line to include the module `learning_logs.urls` at ❶.

The default *urls.py* is in the *learning_log* folder; now we need to make a second *urls.py* file in the *learning_logs* folder. Create a new Python file and save it as *urls.py* in *learning_logs*, and enter this code into it:

urls.py

```
❶ """Defines URL patterns for learning_logs."""

❷ from django.urls import path

❸ from . import views

❹ app_name = 'learning_logs'
❺ urlpatterns = [
      # Home page
❻     path('', views.index, name='index'),
  ]
```

To make it clear which *urls.py* we're working in, we add a docstring at the beginning of the file ❶. We then import the `path` function, which is needed when mapping URLs to views ❷. We also import the `views` module ❸; the dot tells Python to import the *views.py* module from the same directory as the current *urls.py* module. The variable `app_name` helps Django distinguish this *urls.py* file from files of the same name in other apps within the project ❹. The variable `urlpatterns` in this module is a list of individual pages that can be requested from the `learning_logs` app ❺.

The actual URL pattern is a call to the `path()` function, which takes three arguments ❻. The first argument is a string that helps Django route the current request properly. Django receives the requested URL and tries to route the request to a view. It does this by searching all the URL patterns we've defined to find one that matches the current request. Django ignores the base URL for the project (*http://localhost:8000/*), so the empty string (`''`) matches the base URL. Any other URL won't match this pattern, and Django will return an error page if the URL requested doesn't match any existing URL patterns.

The second argument in `path()` ❻ specifies which function to call in *views.py*. When a requested URL matches the pattern we're defining, Django calls the `index()` function from *views.py* (we'll write this view function in the next section). The third argument provides the name `index` for this URL pattern so we can refer to it in other code sections. Whenever we want to provide a link to the home page, we'll use this name instead of writing out a URL.

Writing a View

A view function takes in information from a request, prepares the data needed to generate a page, and then sends the data back to the browser, often by using a template that defines what the page will look like.

The file *views.py* in *learning_logs* was generated automatically when we ran the command `python manage.py startapp`. Here's what's in *views.py* right now:

views.py

```
from django.shortcuts import render

# Create your views here.
```

Currently, this file just imports the `render()` function, which renders the response based on the data provided by views. Open the views file and add the following code for the home page:

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request, 'learning_logs/index.html')
```

When a URL request matches the pattern we just defined, Django looks for a function called `index()` in the *views.py* file. Django then passes the `request` object to this view function. In this case, we don't need to process any data for the page, so the only code in the function is a call to `render()`. The `render()` function here passes two arguments—the original `request` object and a template it can use to build the page. Let's write this template.

Writing a Template

The template defines what the page should look like, and Django fills in the relevant data each time the page is requested. A template allows you to access any data provided by the view. Because our view for the home page provided no data, this template is fairly simple.

Inside the *learning_logs* folder, make a new folder called *templates*. Inside the *templates* folder, make another folder called *learning_logs*. This might seem a little redundant (we have a folder named *learning_logs* inside a folder named *templates* inside a folder named *learning_logs*), but it sets up a structure that Django can interpret unambiguously, even in the context of a large project containing many individual apps. Inside the inner *learning_logs* folder, make a new file called *index.html*. The path to the file will be *learning_log/learning_logs/templates/learning_logs/index.html*. Enter the following code into that file:

index.html

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any topic
you're
learning about.</p>
```

This is a very simple file. If you’re not familiar with HTML, the `<p></p>` tags signify paragraphs. The `<p>` tag opens a paragraph, and the `</p>` tag closes a paragraph. We have two paragraphs: the first acts as a title, and the second describes what users can do with Learning Log.

Now when you request the project’s base URL, *http://localhost:8000/*, you should see the page we just built instead of the default Django page. Django will take the requested URL, and that URL will match the pattern `''`; then Django will call the function `views.index()`, which will render the page using the template contained in *index.html*. Figure 18-3 shows the resulting page.

Figure 18-3: The home page for Learning Log

Although it might seem like a complicated process for creating one page, this separation between URLs, views, and templates works quite well. It allows you to think about each aspect of a project separately. In larger projects, it allows individuals working on the project to focus on the areas in which they’re strongest. For example, a database specialist can focus on the models, a programmer can focus on the view code, and a web designer can focus on the templates.

NOTE

You might see the following error message:

```
ModuleNotFoundError: No module named
'learning_logs.urls'
```

If you do, stop the development server by pressing CTRL-C *in the terminal window where you issued the* `runserver` *command. Then reissue the command* `python manage.py runserver`*. You should be able to see the home page. Any time you run into an error like this, try stopping and restarting the server.*

TRY IT YOURSELF

18-5. Meal Planner: Consider an app that helps people plan their meals throughout the week. Make a new folder called *meal_planner*, and start a new Django project inside this folder. Then make a new app called `meal_plans`. Make a simple home page for this project.

18-6. Pizzeria Home Page: Add a home page to the Pizzeria project you started in Exercise 18-4 (page 394).

Building Additional Pages

Now that we've established a routine for building a page, we can start to build out the Learning Log project. We'll build two pages that display data: a page that lists all topics and a page that shows all the entries for a particular topic. For each page, we'll specify a URL pattern, write a view function, and write a template. But before we do this, we'll create a base template that all templates in the project can inherit from.

Template Inheritance

When building a website, some elements will always need to be repeated on each page. Rather than writing these elements directly into each page, you can write a base template containing the repeated elements and then have each page inherit from the base. This approach lets you focus on developing the unique aspects of each page and makes it much easier to change the overall look and feel of the project.

The Parent Template

We'll create a template called *base.html* in the same directory as *index.html*. This file will contain elements common to all pages; every other template will inherit from *base.html*. The only element we want to repeat on each page right now is the title at the top. Because we'll include this template on every page, let's make the title a link to the home page:

base.html

```
  <p>
❶   <a href="{% url 'learning_logs:index' %}">Learning Log</a>
  </p>

❷ {% block content %}{% endblock content %}
```

The first part of this file creates a paragraph containing the name of the project, which also acts as a home page link. To generate a link, we use a *template tag*, which is indicated by braces and percent signs `{% %}`. A template tag generates information to be displayed on a page. Our template tag `{% url 'learning_logs:index' %}` generates a URL matching the URL pattern defined in *learning_logs/urls.py* with the name `'index'` ❶. In this example, `learning_logs` is the *namespace* and `index` is a uniquely named URL pattern in that namespace. The namespace comes from the value we assigned to `app_name` in the *learning_logs/urls.py* file.

In a simple HTML page, a link is surrounded by the *anchor* tag `<a>`:

```
<a href="link_url">link text</a>
```

Having the template tag generate the URL for us makes it much easier to keep our links up to date. We only need to change the URL pattern in *urls.py*, and Django will automatically insert the updated URL the next time the page is requested. Every page in our project will inherit from *base.html*, so from now on, every page will have a link back to the home page.

At ❷ we insert a pair of `block` tags. This block, named `content`, is a placeholder; the child template will define the kind of information that goes in the `content` block.

A child template doesn't have to define every block from its parent, so you can reserve space in parent templates for as many blocks as you like; the child template uses only as many as it requires.

NOTE

In Python code, we almost always use four spaces when we indent. Template files tend to have more levels of nesting than Python files, so it's common to use only two spaces for each indentation level. You just need to ensure that you're consistent.

The Child Template

Now we need to rewrite *index.html* to inherit from *base.html*. Add the following code to *index.html*:

index.html

```
❶ {% extends "learning_logs/base.html" %}

❷ {% block content %}
  <p>Learning Log helps you keep track of your learning, for any topic
you're
  learning about.</p>
❸ {% endblock content %}
```

If you compare this to the original *index.html*, you can see that we've replaced the Learning Log title with the code for inheriting from a parent template ❶. A child template must have an `{% extends %}` tag on the first line to tell Django which parent template to inherit from. The file *base.html* is part of `learning_logs`, so we include *learning_logs* in the path to the parent template. This line pulls in everything contained in the *base.html* template and allows *index.html* to define what goes in the space reserved by the `content` block.

We define the content block at ❷ by inserting a `{% block %}` tag with the name `content`. Everything that we aren't inheriting from the parent template goes inside the `content` block. Here, that's the paragraph describing the Learning Log project. At ❸ we indicate that we're finished defining the content by using an `{% endblock content %}` tag. The `{% endblock %}` tag doesn't require a name, but if a template grows to contain multiple blocks, it can be helpful to know exactly which block is ending.

You can start to see the benefit of template inheritance: in a child template, we only need to include content that's unique to that page. This not only simplifies each template, but also makes it much easier to modify the site. To modify an element common to many pages, you only need to modify the parent template. Your changes are then carried over to every page that inherits from that template. In a project that includes tens or hundreds of pages, this structure can make it much easier and faster to improve your site.

NOTE

In a large project, it's common to have one parent template called base.html *for the entire site and parent templates for each major section of the site. All the section templates inherit from* base.html, *and each page in the site inherits from a section template. This way you can easily modify the look and feel of the site as a whole, any section in the site, or any individual page. This configuration provides a very efficient way to work, and it encourages you to steadily update your site over time.*

The Topics Page

Now that we have an efficient approach to building pages, we can focus on our next two pages: the general topics page and the page to display entries for a single topic. The topics page will show all topics that users have created, and it's the first page that will involve working with data.

The Topics URL Pattern

First, we define the URL for the topics page. It's common to choose a simple URL fragment that reflects the kind of information presented on the page. We'll use the word *topics*, so the URL *http://localhost:8000/topics/* will return this page. Here's how we modify *learning_logs/urls.py*:

urls.py

```
"""Defines URL patterns for learning_logs."""
--snip--
urlpatterns = [
    # Home page.
    path('', views.index, name='index'),
    # Page that shows all topics.
❶   path('topics/', views.topics, name='topics'),
]
```

We've simply added `topics/` into the string argument used for the home page URL ❶. When Django examines a requested URL, this pattern will match any URL that has the base URL followed by *topics*. You can include or omit a forward slash at the end, but there can't be anything else after the word *topics*, or the pattern won't match. Any request with a URL that matches this pattern will then be passed to the function `topics()` in *views.py*.

The Topics View

The `topics()` function needs to retrieve some data from the database and send it to the template. Here's what we need to add to *views.py*:

views.py

```
  from django.shortcuts import render

❶ from .models import Topic

  def index(request):
      --snip--

❷ def topics(request):
      """Show all topics."""
❸     topics = Topic.objects.order_by('date_added')
❹     context = {'topics': topics}
❺     return render(request, 'learning_logs/topics.html', context)
```

We first import the model associated with the data we need ❶. The `topics()` function needs one parameter: the `request` object Django received from the server ❷. At ❸ we query the database by asking for the `Topic` objects, sorted by the `date_added` attribute. We store the resulting queryset in `topics`.

At ❹ we define a context that we'll send to the template. A *context* is a dictionary in which the keys are names we'll use in the template to access the data, and the values are the data we need to send to the template. In this case, there's one key-value pair, which contains the set of topics we'll display on the page. When building a page that uses data, we pass the `context` variable to `render()` as well as the `request` object and the path to the template ❺.

The Topics Template

The template for the topics page receives the `context` dictionary, so the template can use the data that `topics()` provides. Make a file called *topics.html* in the same directory as *index.html*. Here’s how we can display the topics in the template:

topics.html

```
  {% extends "learning_logs/base.html" %}

  {% block content %}

    <p>Topics</p>

❶  <ul>
❷    {% for topic in topics %}
❸      <li>{{ topic }}</li>
❹    {% empty %}
       <li>No topics have been added yet.</li>
❺    {% endfor %}
❻  </ul>

  {% endblock content %}
```

We use the `{% extends %}` tag to inherit from *base.html*, just as the index template does, and then open a `content` block. The body of this page contains a bulleted list of the topics that have been entered. In standard HTML, a bulleted list is called an *unordered list* and is indicated by the tags `<ul></ul>`. We begin the bulleted list of topics at ❶.

At ❷ we have another template tag equivalent to a `for` loop, which loops through the list `topics` from the `context` dictionary. The code used in templates differs from Python in some important ways. Python uses indentation to indicate which lines of a `for` statement are part of a loop. In a template, every `for` loop needs an explicit `{% endfor %}` tag indicating where the end of the loop occurs. So in a template, you’ll see loops written like this:

```
{% for item in list %}
  do something with each item
{% endfor %}
```

Inside the loop, we want to turn each topic into an item in the bulleted list. To print a variable in a template, wrap the variable name in double braces. The braces won't appear on the page; they just indicate to Django that we're using a template variable. So the code `{{ topic }}` at ❸ will be replaced by the value of `topic` on each pass through the loop. The HTML tag `<li></li>` indicates a *list item*. Anything between these tags, inside a pair of `<ul></ul>` tags, will appear as a bulleted item in the list.

At ❹ we use the `{% empty %}` template tag, which tells Django what to do if there are no items in the list. In this case, we print a message informing the user that no topics have been added yet. The last two lines close out the `for` loop ❺ and then close out the bulleted list ❻.

Now we need to modify the base template to include a link to the topics page. Add the following code to *base.html*:

base.html

```
  <p>
❶   <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
❷   <a href="{% url 'learning_logs:topics' %}">Topics</a>
  </p>

  {% block content %}{% endblock content %}
```

We add a dash after the link to the home page ❶, and then add a link to the topics page using the `{% url %}` template tag again ❷. This line tells Django to generate a link matching the URL pattern with the name `'topics'` in *learning_logs/urls.py*.

Now when you refresh the home page in your browser, you'll see a *Topics* link. When you click the link, you'll see a page that looks similar to

Figure 18-4.

Figure 18-4: The topics page

Individual Topic Pages

Next, we need to create a page that can focus on a single topic, showing the topic name and all the entries for that topic. We'll again define a new URL pattern, write a view, and create a template. We'll also modify the topics page so each item in the bulleted list links to its corresponding topic page.

The Topic URL Pattern

The URL pattern for the topic page is a little different than the prior URL patterns because it will use the topic's `id` attribute to indicate which topic was requested. For example, if the user wants to see the detail page for the Chess topic, where the `id` is 1, the URL will be *http://localhost:8000/topics/1/*. Here's a pattern to match this URL, which you should place in *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Detail page for a single topic.
    path('topics/<int:topic_id>/', views.topic, name='topic'),
]
```

Let’s examine the string `'topics/<int:topic_id>/'` in this URL pattern. The first part of the string tells Django to look for URLs that have the word *topics* after the base URL. The second part of the string, `/<int:topic_id>/`, matches an integer between two forward slashes and stores the integer value in an argument called `topic_id`.

When Django finds a URL that matches this pattern, it calls the view function `topic()` with the value stored in `topic_id` as an argument. We’ll use the value of `topic_id` to get the correct topic inside the function.

The Topic View

The `topic()` function needs to get the topic and all associated entries from the database, as shown here:

views.py

```
  --snip--
❶ def topic(request, topic_id):
      """Show a single topic and all its entries."""
❷     topic = Topic.objects.get(id=topic_id)
❸     entries = topic.entry_set.order_by('-date_added')
❹     context = {'topic': topic, 'entries': entries}
❺     return render(request, 'learning_logs/topic.html', context)
```

This is the first view function that requires a parameter other than the `request` object. The function accepts the value captured by the expression `/<int:topic_id>/` and stores it in `topic_id` ❶. At ❷ we use `get()` to retrieve the topic, just as we did in the Django shell. At ❸ we get the entries associated with this topic, and we order them according to `date_added`. The minus sign in front of `date_added` sorts the results in reverse order, which will display the most recent entries first. We store the topic and entries in the context dictionary ❹ and send `context` to the template *topic.html* ❺.

NOTE

The code phrases at ❷ and ❸ are called queries, *because they query the database for specific information. When you're writing queries like these in your own projects, it's helpful to try them out in the Django shell first. You'll get much quicker feedback in the shell than you will by writing a view and template, and then checking the results in a browser.*

The Topic Template

The template needs to display the name of the topic and the entries. We also need to inform the user if no entries have been made yet for this topic.

topic.html

```
  {% extends 'learning_logs/base.html' %}

  {% block content %}

❶ <p>Topic: {{ topic }}</p>

   <p>Entries:</p>
❷ <ul>
❸ {% for entry in entries %}
    <li>
❹     <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
❺     <p>{{ entry.text|linebreaks }}</p>
    </li>
❻ {% empty %}
    <li>There are no entries for this topic yet.</li>
   {% endfor %}
   </ul>

  {% endblock content %}
```

We extend *base.html*, as we do for all pages in the project. Next, we show the topic that's currently being displayed ❶, which is stored in the template variable `{{ topic }}`. The variable `topic` is available because it's included in the `context` dictionary. We then start a bulleted list to show each of the entries ❷ and loop through them as we did the topics earlier ❸.

Each bullet lists two pieces of information: the timestamp and the full text of each entry. For the timestamp ❹, we display the value of the attribute `date_added`. In Django templates, a vertical line (`|`) represents a template *filter*—a function that modifies the value in a template variable. The filter `date:'M d, Y H:i'` displays timestamps in the format *January 1, 2018 23:00*. The next line displays the full value of `text` rather than just the first 50 characters from `entry`. The filter `linebreaks` ❺ ensures that long text entries include line breaks in a format understood by browsers rather than showing a block of uninterrupted text. At ❻ we use the `{% empty %}` template tag to print a message informing the user that no entries have been made.

Links from the Topics Page

Before we look at the topic page in a browser, we need to modify the topics template so each topic links to the appropriate page. Here's the change you need to make to *topics.html*:

topics.html

```
--snip--
  {% for topic in topics %}
    <li>
      <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
    </li>
  {% empty %}
--snip--
```

We use the URL template tag to generate the proper link, based on the URL pattern in `learning_logs` with the name `'topic'`. This URL pattern requires a `topic_id` argument, so we add the attribute `topic.id` to the URL template tag. Now each topic in the list of topics is a link to a topic page, such as *http://localhost:8000/topics/1/*.

When you refresh the topics page and click a topic, you should see a page that looks like Figure 18-5.

NOTE

There's a subtle but important difference between `topic.id` *and* `topic_id`*. The expression* `topic.id` *examines a topic and retrieves the value of the corresponding ID. The variable* `topic_id` *is a reference to that ID in the code. If you run into errors when working with IDs, make sure you're using these expressions in the appropriate ways.*

Figure 18-5: The detail page for a single topic, showing all entries for a topic

TRY IT YOURSELF

18-7. Template Documentation: Skim the Django template documentation at *https://docs.djangoproject.com/en/2.2/ref/templates/*. You can refer back to it when you're working on your own projects.

18-8. Pizzeria Pages: Add a page to the *Pizzeria* project from Exercise 18-6 (page 398) that shows the names of available pizzas. Then link each pizza name to a page displaying the pizza's toppings. Make sure you use template inheritance to build your pages efficiently.

Summary

In this chapter, you learned how to build simple web applications using the Django framework. You wrote a brief project specification, installed Django to a virtual environment, set up a project, and checked that the project was set up correctly. You set up an app and defined models to represent the data for your app. You learned about databases and how Django helps you migrate your database after you make a change to your models. You created a superuser for the admin site, and you used the admin site to enter some initial data.

You also explored the Django shell, which allows you to work with your project's data in a terminal session. You learned to define URLs, create view functions, and write templates to make pages for your site. You also used template inheritance to simplify the structure of individual templates and make it easier to modify the site as the project evolves.

In Chapter 19, you'll make intuitive, user-friendly pages that allow users to add new topics and entries and edit existing entries without going through the admin site. You'll also add a user registration system, allowing users to create an account and make their own learning log. This is the heart of a web app—the ability to create something that any number of users can interact with.