PROJECT 3

WEB APPLICATIONS

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.

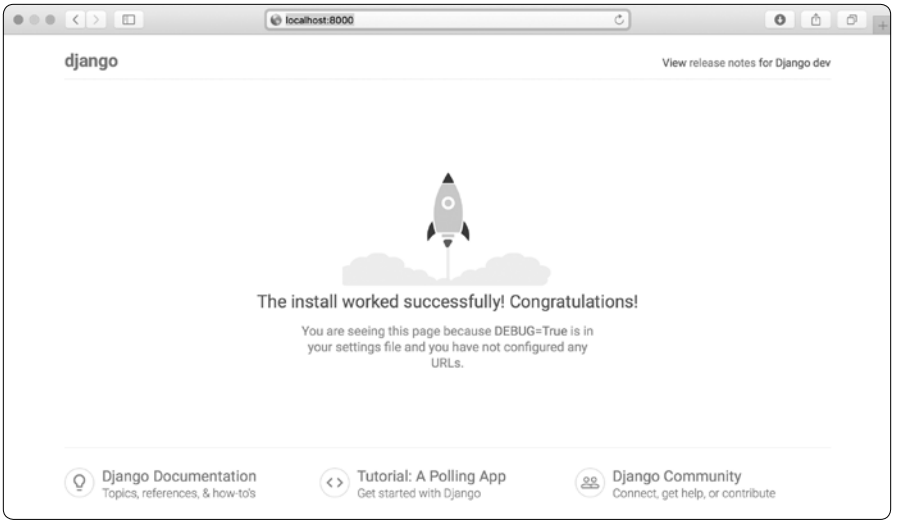


Figure 18-1: Everything is working so far.

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
❶ (ll_env)learning_log$ ls
db.sqlite3  learning_log  learning_logs  ll_env  manage.py
❷ (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* ❶. Open that folder to see what Django has created ❷. 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
❶ Username (leave blank to use 'eric'): ll_admin
❷ Email address:
❸ 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 ❶. 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 ❷. You'll need to enter your password twice ❸.

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.

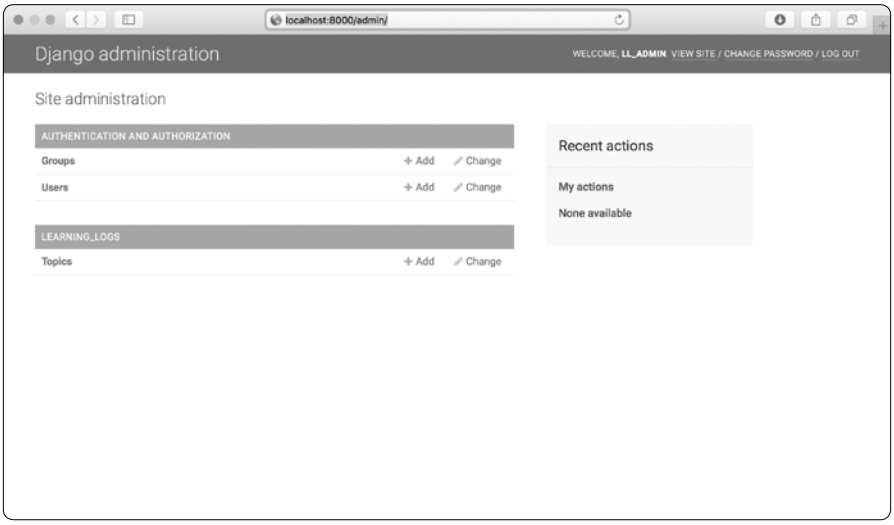


Figure 18-2: The admin site with `Topic` included

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

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

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

The first two lines import a module and a function to manage URLs for the admin site ❶. The body of the file defines the `urlpatterns` variable ❷. 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 ❸ 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 tags signify paragraphs. The tag opens a paragraph, and the 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.

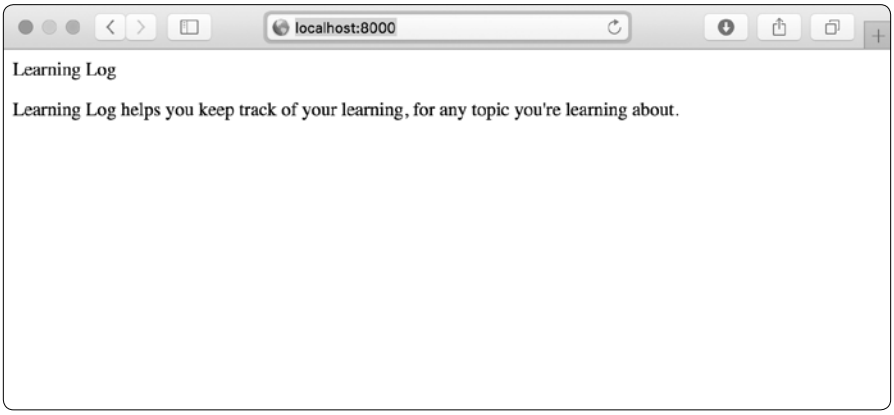


Figure 18-3: The home page for Learning Log

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.

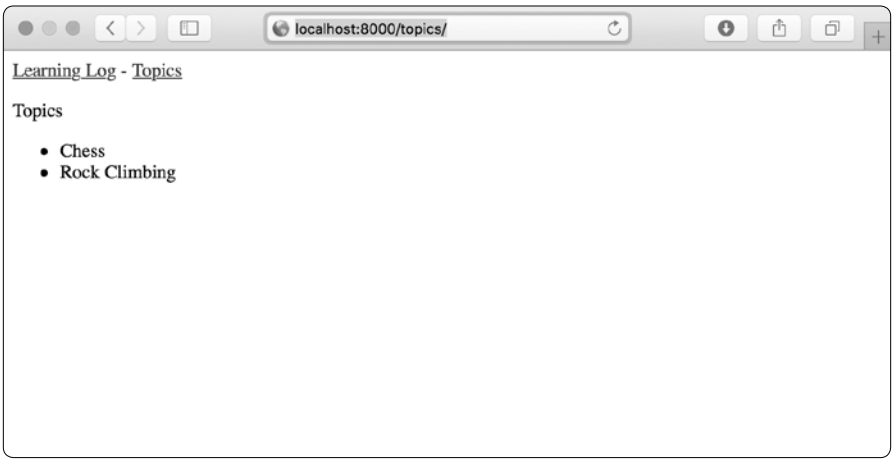


Figure 18-4: The topics page

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.*

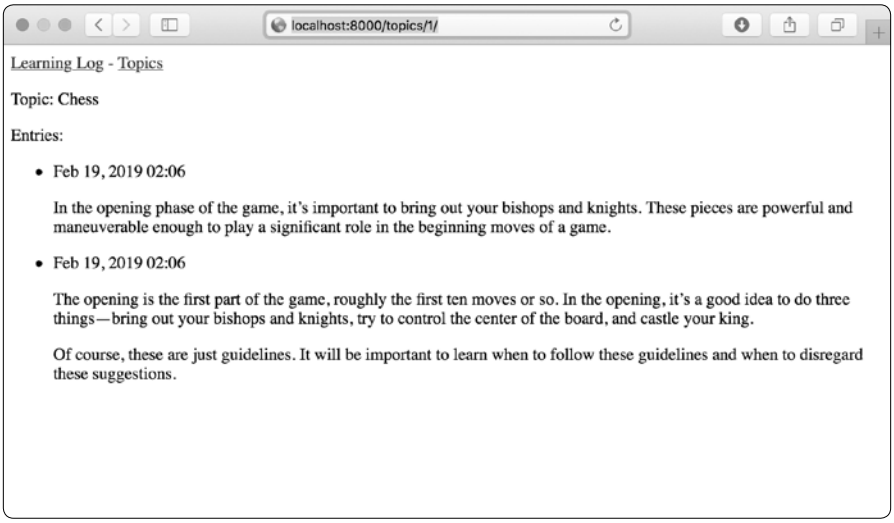


Figure 18-5: The detail page for a single topic, showing all entries for a topic

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

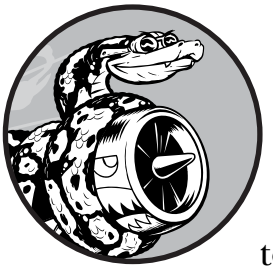
In this chapter, you learned how to build simple web applications using the Django framework. You wrote a brief project specification, installed Django to a virtual environment, set up a project, and checked that the project was set up correctly. You set up an app and defined models to represent the 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.

19

USER ACCOUNTS



At the heart of a web application is the ability for any user, anywhere in the world, to register an account with your app and start using it. In this chapter, you'll build forms so users can add their own topics and entries, and edit existing entries. You'll also learn how Django guards against common attacks to form-based pages so you don't have to spend much time thinking about securing your apps.

You'll also implement a user authentication system. You'll build a registration page for users to create accounts, and then restrict access to certain pages to logged-in users only. Then you'll modify some of the view functions so users can only see their own data. You'll learn to keep your users' data safe and secure.

Allowing Users to Enter Data

Before we build an authentication system for creating accounts, we'll first add some pages that allow users to enter their own data. We'll give users the ability to add a new topic, add a new entry, and edit their previous entries.

Currently, only a superuser can enter data through the admin site. We don't want users to interact with the admin site, so we'll use Django's form-building tools to build pages that allow users to enter data.

Adding New Topics

Let's start by allowing users to add a new topic. Adding a form-based page works in much the same way as the pages we've already built: we define a URL, write a view function, and write a template. The one major difference is the addition of a new module called *forms.py*, which will contain the forms.

The Topic ModelForm

Any page that lets a user enter and submit information on a web page is a *form*, even if it doesn't look like one. When users enter information, we need to *validate* that the information provided is the right kind of data and is not malicious, such as code to interrupt our server. We then need to process and save valid information to the appropriate place in the database. Django automates much of this work.

The simplest way to build a form in Django is to use a *ModelForm*, which uses the information from the models we defined in Chapter 18 to automatically build a form. Write your first form in the file *forms.py*, which you should create in the same directory as *models.py*:

forms.py

```
from django import forms

from .models import Topic

❶ class TopicForm(forms.ModelForm):
      class Meta:
❷         model = Topic
❸         fields = ['text']
❹         labels = {'text': ''}
```

We first import the `forms` module and the model we'll work with, called `Topic`. At ❶ we define a class called `TopicForm`, which inherits from `forms.ModelForm`.

The simplest version of a `ModelForm` consists of a nested `Meta` class telling Django which model to base the form on and which fields to include in the form. At ❷ we build a form from the `Topic` model and include only the `text` field ❸. The code at ❹ tells Django not to generate a label for the `text` field.

The new_topic URL

The URL for a new page should be short and descriptive. When the user wants to add a new topic, we'll send them to *http://localhost:8000/new_topic/*. Here's the URL pattern for the `new_topic` page, which you add to *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Page for adding a new topic
    path('new_topic/', views.new_topic, name='new_topic'),
]
```

This URL pattern sends requests to the view function `new_topic()`, which we'll write next.

The new_topic() View Function

The `new_topic()` function needs to handle two different situations: initial requests for the `new_topic` page (in which case it should show a blank form) and the processing of any data submitted in the form. After data from a submitted form is processed, it needs to redirect the user back to the `topics` page:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm

--snip--
def new_topic(request):
    """Add a new topic."""
❶   if request.method != 'POST':
        # No data submitted; create a blank form.
❷       form = TopicForm()
    else:
        # POST data submitted; process data.
❸       form = TopicForm(data=request.POST)
❹       if form.is_valid():
❺           form.save()
❻           return redirect('learning_logs:topics')

    # Display a blank or invalid form.
❼   context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

We import the function `redirect`, which we'll use to redirect the user back to the `topics` page after they submit their topic. The `redirect()` function takes in the name of a view and redirects the user to that view. We also import the form we just wrote, `TopicForm`.

GET and POST Requests

The two main types of request you'll use when building web apps are GET requests and POST requests. You use *GET* requests for pages that only read data from the server. You usually use *POST* requests when the user needs to submit information through a form. We'll be specifying the POST method for processing all of our forms. (A few other kinds of requests exist, but we won't use them in this project.)

The `new_topic()` function takes in the request object as a parameter. When the user initially requests this page, their browser will send a GET request. Once the user has filled out and submitted the form, their browser will submit a POST request. Depending on the request, we'll know whether the user is requesting a blank form (a GET request) or asking us to process a completed form (a POST request).

The test at ❶ determines whether the request method is GET or POST. If the request method isn't POST, the request is probably GET, so we need to return a blank form (if it's another kind of request, it's still safe to return a blank form). We make an instance of `TopicForm` ❷, assign it to the variable `form`, and send the form to the template in the context dictionary ❼. Because we included no arguments when instantiating `TopicForm`, Django creates a blank form that the user can fill out.

If the request method is POST, the `else` block runs and processes the data submitted in the form. We make an instance of `TopicForm` ❸ and pass it the data entered by the user, stored in `request.POST`. The `form` object that's returned contains the information submitted by the user.

We can't save the submitted information in the database until we've checked that it's valid ❹. The `is_valid()` method checks that all required fields have been filled in (all fields in a form are required by default) and that the data entered matches the field types expected—for example, that the length of `text` is less than 200 characters, as we specified in *models.py* in Chapter 18. This automatic validation saves us a lot of work. If everything is valid, we can call `save()` ❺, which writes the data from the form to the database.

Once we've saved the data, we can leave this page. We use `redirect()` to redirect the user's browser to the `topics` page, where the user should see the topic they just entered in the list of topics.

The `context` variable is defined at the end of the view function, and the page is rendered using the template *new_topic.html*, which we'll create next. This code is placed outside of any `if` block; it will run if a blank form was created, and it will run if a submitted form is determined to be invalid. An invalid form will include some default error messages to help the user submit acceptable data.

The new_topic Template

Now we'll make a new template called *new_topic.html* to display the form we just created.

new_topic.html

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

{% block content %}
  <p>Add a new topic:</p>

❶ <form action="{% url 'learning_logs:new_topic' %}" method='post'>
❷   {% csrf_token %}
❸   {{ form.as_p }}
❹   <button name="submit">Add topic</button>
  </form>

{% endblock content %}
```

This template extends *base.html*, so it has the same base structure as the rest of the pages in Learning Log. At ❶ we define an HTML form. The `action` argument tells the browser where to send the data submitted in the form; in this case, we send it back to the view function `new_topic()`. The `method` argument tells the browser to submit the data as a POST request.

Django uses the template tag `{% csrf_token %}` ❷ to prevent attackers from using the form to gain unauthorized access to the server (this kind of attack is called a *cross-site request forgery*). At ❸ we display the form; here you see how simple Django can make certain tasks, such as displaying a form. We only need to include the template variable `{{ form.as_p }}` for Django to create all the fields necessary to display the form automatically. The `as_p` modifier tells Django to render all the form elements in paragraph format, as a simple way to display the form neatly.

Django doesn't create a submit button for forms, so we define one at ❹.

Linking to the new_topic Page

Next, we include a link to the `new_topic` page on the `topics` page:

topics.html

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

{% block content %}

  <p>Topics</p>

  <ul>
    --snip--
  </ul>

  <a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a>

{% endblock content %}
```

Place the link after the list of existing topics. Figure 19-1 shows the resulting form. Use the form to add a few new topics of your own.

Figure 19-1: The page for adding a new topic

Adding New Entries

Now that the user can add a new topic, they'll want to add new entries too. We'll again define a URL, write a view function and a template, and link to the page. But first, we'll add another class to *forms.py*.

The Entry ModelForm

We need to create a form associated with the `Entry` model but this time with a bit more customization than `TopicForm`:

forms.py

```
from django import forms

from .models import Topic, Entry

class TopicForm(forms.ModelForm):
    --snip--

class EntryForm(forms.ModelForm):
    class Meta:
        model = Entry
        fields = ['text']
❶       labels = {'text': 'Entry:'}
❷       widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

We update the `import` statement to include `Entry` as well as `Topic`. We make a new class called `EntryForm` that inherits from `forms.ModelForm`. The `EntryForm` class has a nested `Meta` class listing the model it's based on and the field to include in the form. We again give the field `'text'` a blank label ❶.

At ❷ we include the `widgets` attribute. A *widget* is an HTML form element, such as a single-line text box, multi-line text area, or drop-down list. By including the `widgets` attribute, you can override Django's default widget choices. By telling Django to use a `forms.Textarea` element, we're customizing

the input widget for the field 'text' so the text area will be 80 columns wide instead of the default 40. This gives users enough room to write a meaningful entry.

The new_entry URL

New entries must be associated with a particular topic, so we need to include a topic_id argument in the URL for adding a new entry. Here's the URL, which you add to *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Page for adding a new entry
    path('new_entry/<int:topic_id>/', views.new_entry, name='new_entry'),
]
```

This URL pattern matches any URL with the form http://localhost:8000/new_entry/*id*/, where *id* is a number matching the topic ID. The code <int:topic_id> captures a numerical value and assigns it to the variable topic_id. When a URL matching this pattern is requested, Django sends the request and the topic's ID to the new_entry() view function.

The new_entry() View Function

The view function for new_entry is much like the function for adding a new topic. Add the following code to your *views.py* file:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm, EntryForm

--snip--
def new_entry(request, topic_id):
    """Add a new entry for a particular topic."""
❶   topic = Topic.objects.get(id=topic_id)

❷   if request.method != 'POST':
        # No data submitted; create a blank form.
❸       form = EntryForm()
    else:
        # POST data submitted; process data.
❹       form = EntryForm(data=request.POST)
        if form.is_valid():
❺           new_entry = form.save(commit=False)
❻           new_entry.topic = topic
            new_entry.save()
❼           return redirect('learning_logs:topic', topic_id=topic_id)

    # Display a blank or invalid form.
    context = {'topic': topic, 'form': form}
    return render(request, 'learning_logs/new_entry.html', context)
```

We update the `import` statement to include the `EntryForm` we just made. The definition of `new_entry()` has a `topic_id` parameter to store the value it receives from the URL. We'll need the topic to render the page and process the form's data, so we use `topic_id` to get the correct topic object at ❶.

At ❷ we check whether the request method is POST or GET. The `if` block executes if it's a GET request, and we create a blank instance of `EntryForm` ❸.

If the request method is POST, we process the data by making an instance of `EntryForm`, populated with the POST data from the `request` object ❹. We then check whether the form is valid. If it is, we need to set the entry object's `topic` attribute before saving it to the database. When we call `save()`, we include the argument `commit=False` ❺ to tell Django to create a new entry object and assign it to `new_entry` without saving it to the database yet. We set the `topic` attribute of `new_entry` to the topic we pulled from the database at the beginning of the function ❻. Then we call `save()` with no arguments, saving the entry to the database with the correct associated topic.

The `redirect()` call at ❼ requires two arguments—the name of the view we want to redirect to and the argument that view function requires. Here, we're redirecting to `topic()`, which needs the argument `topic_id`. This view then renders the topic page that the user made an entry for, and they should see their new entry in the list of entries.

At the end of the function, we create a `context` dictionary and render the page using the *new_entry.html* template. This code will execute for a blank form or for a submitted form that is evaluated as invalid.

The new_entry Template

As you can see in the following code, the template for `new_entry` is similar to the template for `new_topic`:

new_entry.html

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

{% block content %}

❶  <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

  <p>Add a new entry:</p>
❷  <form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
    <button name='submit'>Add entry</button>
  </form>

{% endblock content %}
```

We show the topic at the top of the page ❶, so the user can see which topic they're adding an entry to. The topic also acts as a link back to the main page for that topic.

The form's `action` argument includes the `topic_id` value in the URL, so the view function can associate the new entry with the correct topic ❷. Other than that, this template looks just like *new_topic.html*.

Linking to the new_entry Page

Next, we need to include a link to the `new_entry` page from each topic page in the topic template:

topic.html

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

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
  </p>

  <ul>
  --snip--
  </ul>

{% endblock content %}
```

We place the link to add entries just before showing the entries, because adding a new entry will be the most common action on this page. Figure 19-2 shows the `new_entry` page. Now users can add new topics and as many entries as they want for each topic. Try out the `new_entry` page by adding a few entries to some of the topics you've created.

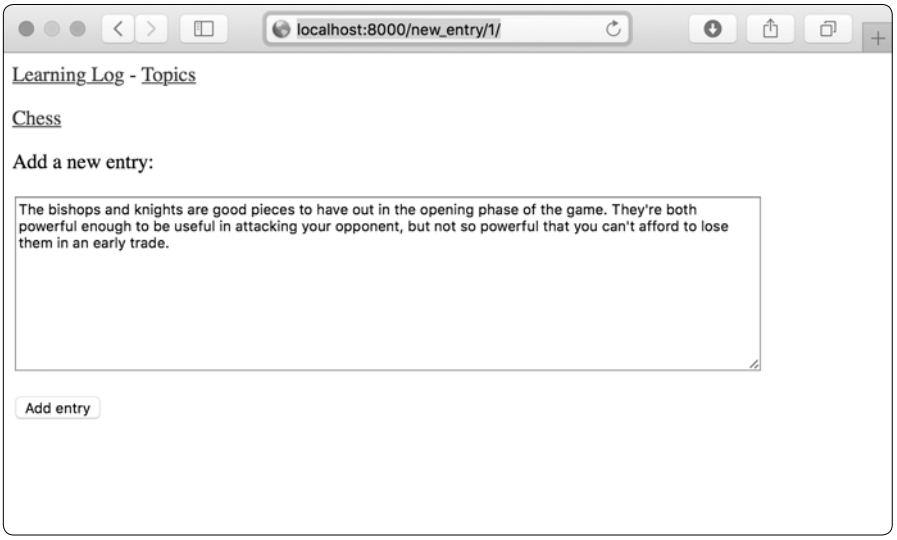


Figure 19-2: The new_entry page

Editing Entries

Now we'll make a page so users can edit the entries they've added.

The edit_entry URL

The URL for the page needs to pass the ID of the entry to be edited. Here's *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Page for editing an entry.
    path('edit_entry/<int:entry_id>/', views.edit_entry, name='edit_entry'),
]
```

The ID passed in the URL (for example, *http://localhost:8000/edit_entry/1/)* is stored in the parameter `entry_id`. The URL pattern sends requests that match this format to the view function `edit_entry()`.

The edit_entry() View Function

When the `edit_entry` page receives a GET request, the `edit_entry()` function returns a form for editing the entry. When the page receives a POST request with revised entry text, it saves the modified text into the database:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic, Entry
from .forms import TopicForm, EntryForm
--snip--

def edit_entry(request, entry_id):
    """Edit an existing entry."""
❶   entry = Entry.objects.get(id=entry_id)
    topic = entry.topic

    if request.method != 'POST':
        # Initial request; pre-fill form with the current entry.
❷       form = EntryForm(instance=entry)
    else:
        # POST data submitted; process data.
❸       form = EntryForm(instance=entry, data=request.POST)
        if form.is_valid():
❹           form.save()
❺           return redirect('learning_logs:topic', topic_id=topic.id)

    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

We first import the `Entry` model. At ❶ we get the entry object that the user wants to edit and the topic associated with this entry. In the `if` block, which runs for a GET request, we make an instance of `EntryForm` with the argument `instance=entry` ❷. This argument tells Django to create the form prefilled with information from the existing entry object. The user will see their existing data and be able to edit that data.

When processing a POST request, we pass the `instance=entry` argument and the `data=request.POST` argument ❸. These arguments tell Django to create a form instance based on the information associated with the existing entry object, updated with any relevant data from `request.POST`. We then check whether the form is valid; if it is, we call `save()` with no arguments because the entry is already associated with the correct topic ❹. We then redirect to the `topic` page, where the user should see the updated version of the entry they edited ❺.

If we're showing an initial form for editing the entry or if the submitted form is invalid, we create the context dictionary and render the page using the *edit_entry.html* template.

The edit_entry Template

Next, we create an *edit_entry.html* template, which is similar to *new_entry.html*:

edit_entry.html

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

{% block content %}

  <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

  <p>Edit entry:</p>

❶ <form action="{% url 'learning_logs:edit_entry' entry.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
❷   <button name="submit">Save changes</button>
  </form>

{% endblock content %}
```

At ❶ the `action` argument sends the form back to the `edit_entry()` function for processing. We include the entry ID as an argument in the `{% url %}` tag, so the view function can modify the correct entry object. We label the submit button as *Save changes* to remind the user they're saving edits, not creating a new entry ❷.

Linking to the edit_entry Page

Now we need to include a link to the `edit_entry` page for each entry on the topic page:

topic.html

```
--snip--
  {% for entry in entries %}
    <li>
      <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
      <p>{{ entry.text|linebreaks }}</p>
      <p>
        <a href="{% url 'learning_logs:edit_entry' entry.id %}">Edit entry</a>
      </p>
    </li>
--snip--
```

We include the edit link after each entry's date and text has been displayed. We use the `{% url %}` template tag to determine the URL for the named URL pattern `edit_entry`, along with the ID attribute of the current entry in the loop (`entry.id`). The link text *Edit entry* appears after each entry on the page. Figure 19-3 shows what the topic page looks like with these links.

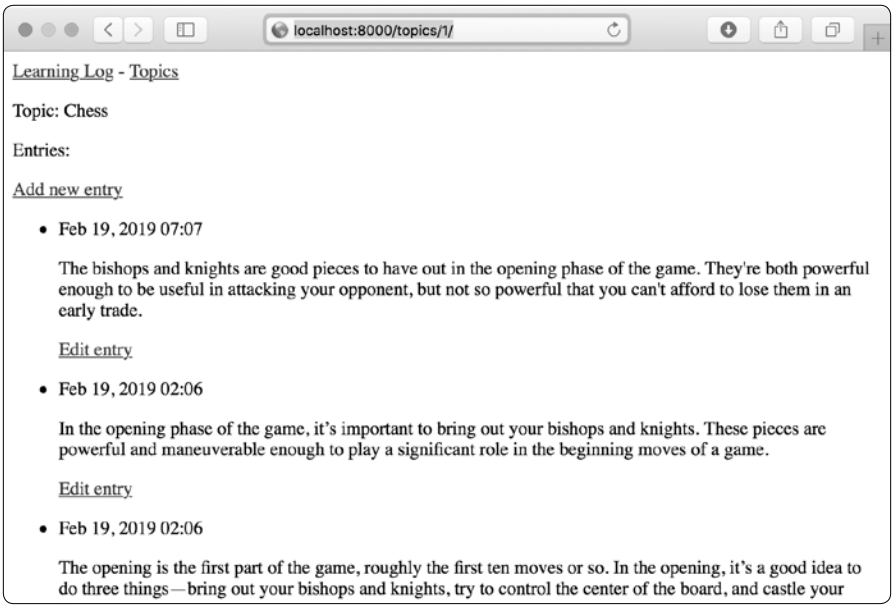


Figure 19-3: Each entry now has a link for editing that entry.

Learning Log now has most of the functionality it needs. Users can add topics and entries, and read through any set of entries they want. In the next section, we'll implement a user registration system so anyone can make an account with Learning Log and create their own set of topics and entries.

TRY IT YOURSELF

19-1. Blog: Start a new Django project called *Blog*. Create an app called *blogs* in the project and a model called `BlogPost`. The model should have fields like `title`, `text`, and `date_added`. Create a superuser for the project, and use the admin site to make a couple of short posts. Make a home page that shows all posts in chronological order.

Create a form for making new posts and another for editing existing posts. Fill in your forms to make sure they work.

Setting Up User Accounts

In this section, we'll set up a user registration and authorization system so people can register an account and log in and out. We'll create a new app to contain all the functionality related to working with users. We'll use the default user authentication system included with Django to do as much of the work as possible. We'll also modify the `Topic` model slightly so every topic belongs to a certain user.

The users App

We'll start by creating a new app called `users`, using the `startapp` command:

```
(ll_env)learning_log$ python manage.py startapp users
(ll_env)learning_log$ ls
❶ db.sqlite3  learning_log  learning_logs  ll_env  manage.py  users
(ll_env)learning_log$ ls users
❷ __init__.py admin.py apps.py migrations models.py tests.py views.py
```

This command makes a new directory called *users* ❶ with a structure identical to the `learning_logs` app ❷.

Adding users to settings.py

We need to add our new app to `INSTALLED_APPS` in *settings.py*, like so:

settings.py

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

    # Default django apps.
    --snip--
]
--snip--
```

Now Django will include the users app in the overall project.

Including the URLs from users

Next, we need to modify the root *urls.py* so it includes the URLs we'll write for the users app:

urls.py

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

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

We add a line to include the file *urls.py* from users. This line will match any URL that starts with the word *users*, such as *http://localhost:8000/users/login/*.

The Login Page

We'll first implement a login page. We'll use the default login view Django provides, so the URL pattern for this app looks a little different. Make a new *urls.py* file in the directory *learning_log/users/*, and add the following to it:

urls.py

```
"""Defines URL patterns for users"""

from django.urls import path, include

❶ app_name = 'users'
urlpatterns = [
    # Include default auth urls.
❷     path('', include('django.contrib.auth.urls')),
]
```

We import the path function, and then import the include function so we can include some default authentication URLs that Django has defined. These default URLs include named URL patterns, such as 'login' and 'logout'. We set the variable app_name to 'users' so Django can distinguish these URLs from URLs belonging to other apps ❶. Even default URLs provided by Django, when included in the users app's *urls.py* file, will be accessible through the users namespace.

The login page's pattern matches the URL *http://localhost:8000/users/login/* ❷. When Django reads this URL, the word *users* tells Django to look in *users/urls.py*, and *login* tells it to send requests to Django's default login view.

The login Template

When the user requests the login page, Django will use a default view function, but we still need to provide a template for the page. The default

authentication views look for templates inside a folder called *registration*, so we'll need to make that folder. Inside the *learning_log/users/* directory, make a directory called *templates*; inside that, make another directory called *registration*. Here's the *login.html* template, which you should save in *learning_log/users/templates/registration*:

login.html

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

{% block content %}

❶  {% if form.errors %}
     <p>Your username and password didn't match. Please try again.</p>
   {% endif %}

❷  <form method="post" action="{% url 'users:login' %}">
     {% csrf_token %}
❸    {{ form.as_p }}

❹    <button name="submit">Log in</button>
❺    <input type="hidden" name="next"
       value="{% url 'learning_logs:index' %}" />
   </form>

{% endblock content %}
```

This template extends *base.html* to ensure that the login page will have the same look and feel as the rest of the site. Note that a template in one app can inherit from a template in another app.

If the form's `errors` attribute is set, we display an error message ❶, reporting that the username and password combination don't match anything stored in the database.

We want the login view to process the form, so we set the `action` argument as the URL of the login page ❷. The login view sends a form to the template, and it's up to us to display the form ❸ and add a submit button ❹. At ❺ we include a hidden form element, `'next'`; the `value` argument tells Django where to redirect the user after they've logged in successfully. In this case, we send the user back to the home page.

Linking to the Login Page

Let's add the login link to *base.html* so it appears on every page. We don't want the link to display when the user is already logged in, so we nest it inside an `{% if %}` tag:

base.html

```
<p>
  <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
  <a href="{% url 'learning_logs:topics' %}">Topics</a> -
❶ {% if user.is_authenticated %}
❷   Hello, {{ user.username }}.
  {% else %}
❸   <a href="{% url 'users:login' %}">Log in</a>
  {% endif %}
```

```
</p>

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

In Django's authentication system, every template has a `user` variable available, which always has an `is_authenticated` attribute set: the attribute is `True` if the user is logged in and `False` if they aren't. This attribute allows you to display one message to authenticated users and another to unauthenticated users.

Here we display a greeting to users currently logged in ❶. Authenticated users have an additional `username` attribute set, which we use to personalize the greeting and remind the user they're logged in ❷. At ❸ we display a link to the login page for users who haven't been authenticated.

Using the Login Page

We've already set up a user account, so let's log in to see if the page works. Go to *http://localhost:8000/admin/*. If you're still logged in as an admin, look for a logout link in the header and click it.

When you're logged out, go to *http://localhost:8000/users/login/*. You should see a login page similar to the one shown in Figure 19-4. Enter the username and password you set up earlier, and you should be brought back to the index page. The header on the home page should display a greeting personalized with your username.

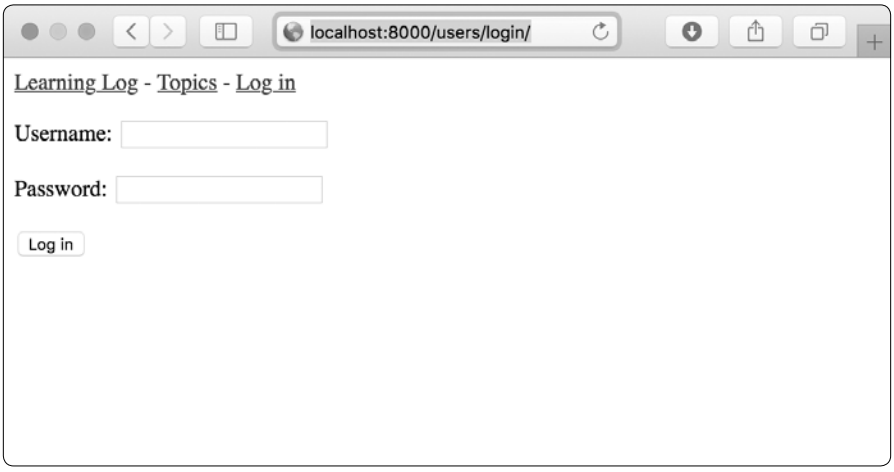


Figure 19-4: The login page

Logging Out

Now we need to provide a way for users to log out. We'll put a link in *base .html* that logs out users; when they click this link, they'll go to a page confirming that they've been logged out.

Adding a Logout Link to base.html

We'll add the link for logging out to *base.html* so it's available on every page. We'll include it in the `{% if user.is_authenticated %}` portion so only users who are already logged in can see it:

base.html

```
--snip--
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">Log out</a>
  {% else %}
    --snip--
```

The default named URL pattern for logging out is simply `'logout'`.

The Logout Confirmation Page

Users will want to know that they've successfully logged out, so the default logout view renders the page using the template *logged_out.html*, which we'll create now. Here's a simple page confirming that the user has been logged out. Save this file in *templates/registration*, the same place where you saved *login.html*:

logged_out.html

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

{% block content %}
  <p>You have been logged out. Thank you for visiting!</p>
{% endblock content %}
```

We don't need anything else on this page, because *base.html* provides links back to the home page and the login page if the user wants to go back to either page.

Figure 19-5 shows the logged out page as it appears to a user who has just clicked the *Log out* link. The styling is minimal because we're focusing on building a site that works properly. When the required set of features works, we'll style the site to look more professional.

Figure 19-5: The logged out page confirms that a user has successfully logged out.

The Registration Page

Next, we'll build a page so new users can register. We'll use Django's default `UserCreationForm` but write our own view function and template.

The register URL

The following code provides the URL pattern for the registration page, again in *users/urls.py*:

urls.py

```
"""Defines URL patterns for users"""

from django.urls import path, include

from . import views

app_name = 'users'
urlpatterns = [
    # Include default auth urls.
    path('', include('django.contrib.auth.urls')),
    # Registration page.
    path('register/', views.register, name='register'),
]
```

We import the `views` module from `users`, which we need because we're writing our own view for the registration page. The pattern for the registration page matches the URL *http://localhost:8000/users/register/* and sends requests to the `register()` function we're about to write.

The register() View Function

The `register()` view function needs to display a blank registration form when the registration page is first requested and then process completed registration forms when they're submitted. When a registration is successful, the function also needs to log in the new user. Add the following code to *users/views.py*:

views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

def register(request):
    """Register a new user."""
    if request.method != 'POST':
        # Display blank registration form.
❶       form = UserCreationForm()
    else:
        # Process completed form.
❷       form = UserCreationForm(data=request.POST)

❸       if form.is_valid():
❹           new_user = form.save()
            # Log the user in and then redirect to home page.
```

```
❺             login(request, new_user)
❻             return redirect('learning_logs:index')

    # Display a blank or invalid form.
    context = {'form': form}
    return render(request, 'registration/register.html', context)
```

We import the `render()` and `redirect()` functions. Then we import the `login()` function to log the user in if their registration information is correct. We also import the default `UserCreationForm`. In the `register()` function, we check whether or not we're responding to a POST request. If we're not, we make an instance of `UserCreationForm` with no initial data ❶.

If we're responding to a POST request, we make an instance of `UserCreationForm` based on the submitted data ❷. We check that the data is valid ❸—in this case, that the username has the appropriate characters, the passwords match, and the user isn't trying to do anything malicious in their submission.

If the submitted data is valid, we call the form's `save()` method to save the username and the hash of the password to the database ❹. The `save()` method returns the newly created user object, which we assign to `new_user`. When the user's information is saved, we log them in by calling the `login()` function with the `request` and `new_user` objects ❺, which creates a valid session for the new user. Finally, we redirect the user to the home page ❻, where a personalized greeting in the header tells them their registration was successful.

At the end of the function we render the page, which will either be a blank form or a submitted form that is invalid.

The register Template

Now create a template for the registration page, which will be similar to the login page. Be sure to save it in the same directory as *login.html*:

register.html

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

{% block content %}

  <form method="post" action="{% url 'users:register' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">Register</button>
    <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
  </form>

{% endblock content %}
```

We use the `as_p` method again so Django will display all the fields in the form appropriately, including any error messages if the form isn't filled out correctly.

Linking to the Registration Page

Next, we'll add the code to show the registration page link to any user who isn't currently logged in:

base.html

```
--snip--
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">Log out</a>
  {% else %}
    <a href="{% url 'users:register' %}">Register</a> -
    <a href="{% url 'users:login' %}">Log in</a>
  {% endif %}
--snip--
```

Now users who are logged in see a personalized greeting and a logout link. Users who aren't logged in see a registration page link and a login link. Try out the registration page by making several user accounts with different usernames.

In the next section, we'll restrict some of the pages so they're available only to registered users, and we'll make sure every topic belongs to a specific user.

NOTE *The registration system we've set up allows anyone to make any number of accounts for Learning Log. But some systems require users to confirm their identity by sending a confirmation email the user must reply to. By doing so, the system generates fewer spam accounts than the simple system we're using here. However, when you're learning to build apps, it's perfectly appropriate to practice with a simple user registration system like the one we're using.*

TRY IT YOURSELF

19-2. Blog Accounts: Add a user authentication and registration system to the Blog project you started in Exercise 19-1 (page 421). Make sure logged-in users see their username somewhere on the screen and unregistered users see a link to the registration page.

Allowing Users to Own Their Data

Users should be able to enter data exclusive to them, so we'll create a system to figure out which data belongs to which user. Then we'll restrict access to certain pages so users can work with only their own data.

We'll modify the `Topic` model so every topic belongs to a specific user. This will also take care of entries, because every entry belongs to a specific topic. We'll start by restricting access to certain pages.

Restricting Access with @login_required

Django makes it easy to restrict access to certain pages to logged-in users through the `@login_required` decorator. A *decorator* is a directive placed just before a function definition that Python applies to the function before it runs, to alter how the function code behaves. Let's look at an example.

Restricting Access to the Topics Page

Each topic will be owned by a user, so only registered users can request the topics page. Add the following code to *learning_logs/views.py*:

views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
--snip--

@login_required
def topics(request):
    """Show all topics."""
    --snip--
```

We first import the `login_required()` function. We apply `login_required()` as a decorator to the `topics()` view function by prepending `login_required` with the `@` symbol. As a result, Python knows to run the code in `login_required()` before the code in `topics()`.

The code in `login_required()` checks whether a user is logged in, and Django runs the code in `topics()` only if they are. If the user isn't logged in, they're redirected to the login page.

To make this redirect work, we need to modify *settings.py* so Django knows where to find the login page. Add the following at the very end of *settings.py*:

settings.py

```
--snip--

# My settings
LOGIN_URL = 'users:login'
```

Now when an unauthenticated user requests a page protected by the `@login_required` decorator, Django will send the user to the URL defined by `LOGIN_URL` in *settings.py*.

You can test this setting by logging out of any user accounts and going to the home page. Click the **Topics** link, which should redirect you to the login page. Then log in to any of your accounts, and from the home page click the **Topics** link again. You should be able to access the topics page.

Restricting Access Throughout Learning Log

Django makes it easy to restrict access to pages, but you have to decide which pages to protect. It's best to think about which pages need to be unrestricted first, and then restrict all the other pages in the project. You can easily correct overrestricting access, and it's less dangerous than leaving sensitive pages unrestricted.

In Learning Log, we'll keep the home page and the registration page unrestricted. We'll restrict access to every other page.

Here's *learning_logs/views.py* with `@login_required` decorators applied to every view except `index()`:

views.py

```
--snip--
@login_required
def topics(request):
    --snip--

@login_required
def topic(request, topic_id):
    --snip--

@login_required
def new_topic(request):
    --snip--

@login_required
def new_entry(request, topic_id):
    --snip--

@login_required
def edit_entry(request, entry_id):
    --snip--
```

Try accessing each of these pages while logged out: you'll be redirected back to the login page. You'll also be unable to click links to pages such as `new_topic`. But if you enter the URL *http://localhost:8000/new_topic/*, you'll be redirected to the login page. You should restrict access to any URL that's publicly accessible and relates to private user data.

Connecting Data to Certain Users

Next, we need to connect the data to the user who submitted it. We need to connect only the data highest in the hierarchy to a user, and the lower-level data will follow. For example, in Learning Log, topics are the highest level of data in the app, and all entries are connected to a topic. As long as each topic belongs to a specific user, we can trace the ownership of each entry in the database.

We'll modify the `Topic` model by adding a foreign key relationship to a user. We'll then have to migrate the database. Finally, we'll modify some of the views so they only show the data associated with the currently logged in user.

Modifying the Topic Model

The modification to *models.py* is just two lines:

models.py

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
   """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

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

class Entry(models.Model):
    --snip--
```

We import the `User` model from `django.contrib.auth`. Then we add an `owner` field to `Topic`, which establishes a foreign key relationship to the `User` model. If a user is deleted, all the topics associated with that user will be deleted as well.

Identifying Existing Users

When we migrate the database, Django will modify the database so it can store a connection between each topic and a user. To make the migration, Django needs to know which user to associate with each existing topic. The simplest approach is to start by giving all existing topics to one user—for example, the superuser. But first we need to know that user's ID.

Let's look at the IDs of all users created so far. Start a Django shell session and issue the following commands:

```
(ll_env)learning_log$ python manage.py shell
❶ >>> from django.contrib.auth.models import User
❷ >>> User.objects.all()
<QuerySet [<User: ll_admin>, <User: eric>, <User: willie>]>
❸ >>> for user in User.objects.all():
...     print(user.username, user.id)
...
ll_admin 1
eric 2
willie 3
>>>
```

At ❶ we import the `User` model into the shell session. We then look at all the users that have been created so far ❷. The output shows three users: *ll_admin*, *eric*, and *willie*.

At ❸ we loop through the list of users and print each user's username and ID. When Django asks which user to associate the existing topics with, we'll use one of these ID values.

Migrating the Database

Now that we know the IDs, we can migrate the database. When we do this, Python will ask us to connect the `Topic` model to a particular owner temporarily or to add a default to our *models.py* file to tell it what to do. Choose option 1:

```
❶ (ll_env)learning_log$ python manage.py makemigrations learning_logs
❷ You are trying to add a non-nullable field 'owner' to topic without a default;
  we can't do that (the database needs something to populate existing rows).
❸ Please select a fix:
   1) Provide a one-off default now (will be set on all existing rows with a
      null value for this column)
   2) Quit, and let me add a default in models.py
❹ Select an option: 1
❺ Please enter the default value now, as valid Python
  The datetime and django.utils.timezone modules are available, so you can do
  e.g. timezone.now
  Type 'exit' to exit this prompt
❻ >>> 1
  Migrations for 'learning_logs':
    learning_logs/migrations/0003_topic_owner.py
  - Add field owner to topic
  (ll_env)learning_log$
```

We start by issuing the `makemigrations` command ❶. In the output at ❷, Django indicates that we're trying to add a required (`non-nullable`) field to an existing model (`topic`) with no default value specified. Django gives us two options at ❸: we can provide a default right now, or we can quit and add a default value in *models.py*. At ❹ we've chosen the first option. Django then asks us to enter the default value ❺.

To associate all existing topics with the original admin user, *ll_admin*, I entered the user ID of 1 at ❻. You can use the ID of any user you've created; it doesn't have to be a superuser. Django then migrates the database using this value and generates the migration file *0003_topic_owner.py*, which adds the field `owner` to the `Topic` model.

Now we can execute the migration. Enter the following in an active virtual environment:

```
(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.0003_topic_owner... OK
(ll_env)learning_log$
```

Django applies the new migration, and the result is `OK` ❶.

We can verify that the migration worked as expected in the shell session, like this:

```
❶ >>> from learning_logs.models import Topic
❷ >>> for topic in Topic.objects.all():
...     print(topic, topic.owner)
...
Chess ll_admin
Rock Climbing ll_admin
>>>
```

We import `Topic` from `learning_logs.models` ❶, and then loop through all existing topics, printing each topic and the user it belongs to ❷. You can see that each topic now belongs to the user *ll_admin*. (If you get an error when you run this code, try exiting the shell and starting a new shell.)

NOTE *You can simply reset the database instead of migrating, but that will lose all existing data. It's good practice to learn how to migrate a database while maintaining the integrity of users' data. If you do want to start with a fresh database, issue the command* `python manage.py flush` *to rebuild the database structure. You'll have to create a new superuser, and all of your data will be gone.*

Restricting Topics Access to Appropriate Users

Currently, if you're logged in, you'll be able to see all the topics, no matter which user you're logged in as. We'll change that by showing users only the topics that belong to them.

Make the following change to the `topics()` function in *views.py*:

views.py

```
--snip--
@login_required
def topics(request):
    """Show all topics."""
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
--snip–
```

When a user is logged in, the request object has a `request.user` attribute set that stores information about the user. The query `Topic.objects.filter(owner=request.user)` tells Django to retrieve only the `Topic` objects from the database whose `owner` attribute matches the current user. Because we're not changing how the topics are displayed, we don't need to change the template for the topics page at all.

To see if this works, log in as the user you connected all existing topics to, and go to the topics page. You should see all the topics. Now log out, and log back in as a different user. The topics page should list no topics.

Protecting a User's Topics

We haven't restricted access to the topic pages yet, so any registered user could try a bunch of URLs, like *http://localhost:8000/topics/1/*, and retrieve topic pages that happen to match.

Try it yourself. While logged in as the user that owns all topics, copy the URL or note the ID in the URL of a topic, and then log out and log back in as a different user. Enter that topic's URL. You should be able to read the entries, even though you're logged in as a different user.

We'll fix this now by performing a check before retrieving the requested entries in the `topic()` view function:

views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
❶ from django.http import Http404

--snip--
@login_required
def topic(request, topic_id):
    """Show a single topic and all its entries."""
    topic = Topic.objects.get(id=topic_id)
    # Make sure the topic belongs to the current user.
❷     if topic.owner != request.user:
        raise Http404

    entries = topic.entry_set.order_by('-date_added')
    context = {'topic': topic, 'entries': entries}
    return render(request, 'learning_logs/topic.html', context)
--snip--
```

A 404 response is a standard error response that's returned when a requested resource doesn't exist on a server. Here we import the `Http404` exception ❶, which we'll raise if the user requests a topic they shouldn't see. After receiving a topic request, we make sure the topic's user matches the currently logged in user before rendering the page. If the current user doesn't own the requested topic, we raise the `Http404` exception ❷, and Django returns a 404 error page.

Now if you try to view another user's topic entries, you'll see a *Page Not Found* message from Django. In Chapter 20, we'll configure the project so users will see a proper error page.

Protecting the edit_entry Page

The `edit_entry` pages have URLs in the form `http://localhost:8000/edit_entry/entry_id/`, where the *entry_id* is a number. Let's protect this page so no one can use the URL to gain access to someone else's entries:

views.py

```
--snip--
@login_required
def edit_entry(request, entry_id):
    """Edit an existing entry."""
    entry = Entry.objects.get(id=entry_id)
```

```
    topic = entry.topic
    if topic.owner != request.user:
        raise Http404

    if request.method != 'POST':
        --snip--
```

We retrieve the entry and the topic associated with this entry. We then check whether the owner of the topic matches the currently logged in user; if they don't match, we raise an `Http404` exception.

Associating New Topics with the Current User

Currently, our page for adding new topics is broken, because it doesn't associate new topics with any particular user. If you try adding a new topic, you'll see the error message `IntegrityError` along with `NOT NULL constraint failed: learning_logs_topic.owner_id`. Django's saying you can't create a new topic without specifying a value for the topic's `owner` field.

There's a straightforward fix for this problem, because we have access to the current user through the `request` object. Add the following code, which associates the new topic with the current user:

views.py

```
--snip--
@login_required
def new_topic(request):
    """Add a new topic."""
    if request.method != 'POST':
        # No data submitted; create a blank form.
        form = TopicForm()
    else:
        # POST data submitted; process data.
        form = TopicForm(data=request.POST)
        if form.is_valid():
❶           new_topic = form.save(commit=False)
❷           new_topic.owner = request.user
❸           new_topic.save()
            return redirect('learning_logs:topics')

    # Display a blank or invalid form.
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
--snip--
```

When we first call `form.save()`, we pass the `commit=False` argument because we need to modify the new topic before saving it to the database ❶. We then set the new topic's `owner` attribute to the current user ❷. Finally, we call `save()` on the topic instance just defined ❸. Now the topic has all the required data and will save successfully.

You should be able to add as many new topics as you want for as many different users as you want. Each user will have access only to their own data, whether they're viewing data, entering new data, or modifying old data.

TRY IT YOURSELF

19-3. Refactoring: There are two places in *views.py* where we make sure the user associated with a topic matches the currently logged in user. Put the code for this check in a function called `check_topic_owner()`, and call this function where appropriate.

19-4. Protecting new_entry: Currently, a user can add a new entry to another user's learning log by entering a URL with the ID of a topic belonging to another user. Prevent this attack by checking that the current user owns the entry's topic before saving the new entry.

19-5. Protected Blog: In your Blog project, make sure each blog post is connected to a particular user. Make sure all posts are publicly accessible but only registered users can add posts and edit existing posts. In the view that allows users to edit their posts, make sure the user is editing their own post before processing the form.

Summary

In this chapter, you learned to use forms to allow users to add new topics and entries, and edit existing entries. You then learned how to implement user accounts. You allowed existing users to log in and out, and used Django's default `UserCreationForm` to let people create new accounts.

After building a simple user authentication and registration system, you restricted access to logged-in users for certain pages using the `@login_required` decorator. You then attributed data to specific users through a foreign key relationship. You also learned to migrate the database when the migration requires you to specify some default data.

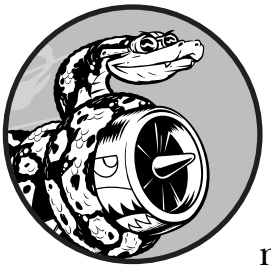
Finally, you learned how to make sure a user can only see data that belongs to them by modifying the view functions. You retrieved appropriate data using the `filter()` method and compared the owner of the requested data to the currently logged in user.

It might not always be immediately obvious what data you should make available and what data you should protect, but this skill will come with practice. The decisions we've made in this chapter to secure our users' data also illustrate why working with others is a good idea when building a project: having someone else look over your project makes it more likely that you'll spot vulnerable areas.

You now have a fully functioning project running on your local machine. In the final chapter, you'll style Learning Log to make it visually appealing, and you'll deploy the project to a server so anyone with internet access can register and make an account.

20

STYLING AND DEPLOYING AN APP



Learning Log is fully functional now, but it has no styling and runs only on your local machine. In this chapter, you'll style the project in a simple but professional manner and then deploy it to a live server so anyone in the world can make an account and use it.

For the styling we'll use the Bootstrap library, a collection of tools for styling web applications so they look professional on all modern devices, from a large flat-screen monitor to a smartphone. To do this, we'll use the django-bootstrap4 app, which will also give you practice using apps made by other Django developers.

We'll deploy Learning Log using Heroku, a site that lets you push your project to one of its servers, making it available to anyone with an internet connection. We'll also start using a version control system called Git to track changes to the project.

When you're finished with Learning Log, you'll be able to develop simple web applications, make them look good, and deploy them to a live server. You'll also be able to use more advanced learning resources as you develop your skills.

Styling Learning Log

We've purposely ignored styling until now to focus on Learning Log's functionality first. This is a good way to approach development, because an app is useful only if it works. Of course, once it's working, appearance is critical so people will want to use it.

In this section, I'll introduce the django-bootstrap4 app and show you how to integrate it into a project to make it ready for live deployment.

The django-bootstrap4 App

We'll use django-bootstrap4 to integrate Bootstrap into our project. This app downloads the required Bootstrap files, places them in an appropriate location in your project, and makes the styling directives available in your project's templates.

To install django-bootstrap4, issue the following command in an active virtual environment:

```
(ll_env)learning_log$ pip install django-bootstrap4
--snip--
Successfully installed django-bootstrap4-0.0.7
```

Next, we need to add the following code to include django-bootstrap4 in `INSTALLED_APPS` in *settings.py*:

settings.py

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

    # Third party apps.
    'bootstrap4',

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

Start a new section called *Third party apps* for apps created by other developers and add `'bootstrap4'` to this section. Make sure you place this section after `# My apps` but before the section containing Django's default apps.

Using Bootstrap to Style Learning Log

Bootstrap is a large collection of styling tools. It also has a number of templates you can apply to your project to create an overall style. It's much easier to use these templates than it is to use individual styling tools. To see the templates Bootstrap offers, go to *https://getbootstrap.com/*, click **Examples**, and look for the *Navbars* section. We'll use the *Navbar static* template, which provides a simple top navigation bar and a container for the page's content.

Figure 20-1 shows what the home page will look like after we apply Bootstrap's template to *base.html* and modify *index.html* slightly.

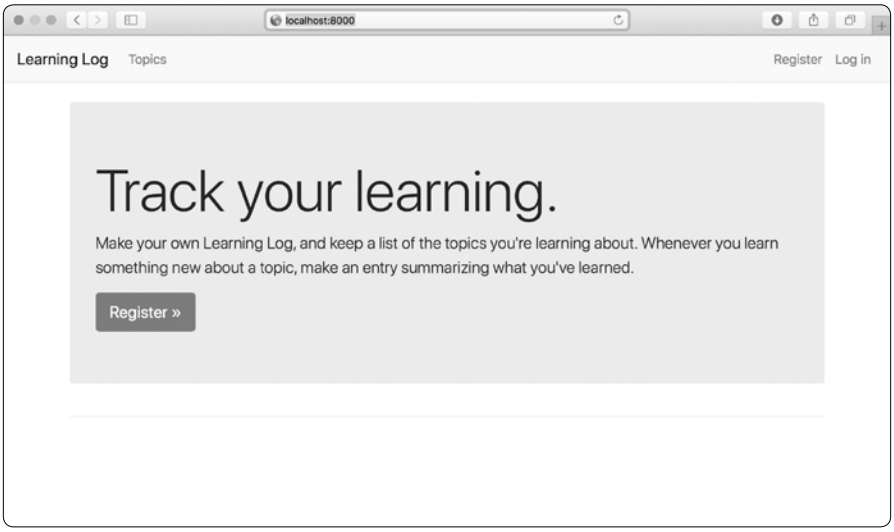


Figure 20-1: The Learning Log home page using Bootstrap

Modifying base.html

We need to modify the *base.html* template to accommodate the Bootstrap template. I'll introduce the new *base.html* in parts.

Defining the HTML Headers

The first change we'll make to *base.html* defines the HTML headers in the file, so whenever a Learning Log page is open, the browser title bar displays the site name. We'll also add some requirements for using Bootstrap in our templates. Delete everything in *base.html* and replace it with the following code:

base.html

```
❶ {% load bootstrap4 %}

❷ <!doctype html>
❸ <html lang="en">
❹ <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
      shrink-to-fit=no">
❺   <title>Learning Log</title>

❻   {% bootstrap_css %}
    {% bootstrap_javascript jquery='full' %}

❼ </head>
```

At ❶ we load the collection of template tags available in django-bootstrap4. Next, we declare this file as an HTML document ❷ written in English ❸. An HTML file is divided into two main parts, the *head* and the *body*—the head of the file begins at ❹. The head of an HTML file doesn't contain any content: it just tells the browser what it needs to know to display the page correctly. At ❺ we include a `title` element for the page, which will display in the browser's title bar whenever Learning Log is open.

At ❻ we use one of django-bootstrap4's custom template tags, which tells Django to include all the Bootstrap style files. The tag that follows enables all the interactive behavior you might use on a page, such as collapsible navigation bars. At ❼ is the closing `</head>` tag.

Defining the Navigation Bar

The code that defines the navigation bar at the top of the page is fairly long, because it has to work well on narrow phone screens and wide desktop monitors. We'll work through the navigation bar in sections.

Here's the first part of the navigation bar:

base.html

```
    --snip--
    </head>
❶ <body>

❷   <nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border">

❸     <a class="navbar-brand" href="{% url 'learning_logs:index'%}">
          Learning Log</a>

❹     <button class="navbar-toggler" type="button" data-toggle="collapse"
          data-target="#navbarCollapse" aria-controls="navbarCollapse"
          aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span></button>
```

The first element is the opening `<body>` tag ❶. The *body* of an HTML file contains the content users will see on a page. At ❷ is a `<nav>` element that indicates the page's navigation links section. Everything contained in this element is styled according to the Bootstrap style rules defined by the selectors `navbar`, `navbar-expand-md`, and the rest that you see here. A *selector* determines which elements on a page a certain style rule applies to. The `navbar-light` and `bg-light` selectors style the navigation bar with a light-themed background. The `mb` in `mb-4` is short for *margin-bottom*; this selector ensures that a little space appears between the navigation bar and the rest of the page. The `border` selector provides a thin border around the light background to set it off a little from the rest of the page.

At ❸ we set the project's name to appear at the far left of the navigation bar and make it a link to the home page; it will appear on every page in the project. The `navbar-brand` selector styles this link so it stands out from the rest of the links and is a way of branding the site.

At ❹ the template defines a button that appears if the browser window is too narrow to display the whole navigation bar horizontally. When the

user clicks the button, the navigation elements will appear in a drop-down list. The `collapse` reference causes the navigation bar to collapse when the user shrinks the browser window or when the site is displayed on mobile devices with small screens.

Here's the next section of code that defines the navigation bar:

base.html

```
        --snip--
      <span class="navbar-toggler-icon"></span></button>
❶   <div class="collapse navbar-collapse" id="navbarCollapse">
❷     <ul class="navbar-nav mr-auto">
❸       <li class="nav-item">
          <a class="nav-link" href="{% url 'learning_logs:topics'%}">
              Topics</a></li>
      </ul>
```

At ❶ we open a new section of the navigation bar. The term *div* is short for division; you build a web page by dividing it into sections and defining style and behavior rules that apply to that section. Any styling or behavior rules that are defined in an opening div tag affect everything you see until the next closing div tag, which is written as `</div>`. This is the beginning of the part of the navigation bar that will be collapsed on narrow screens and windows.

At ❷ we define a new set of links. Bootstrap defines navigation elements as items in an unordered list with style rules that make it look nothing like a list. Every link or element you need on the bar can be included as an item in one of these lists. Here, the only item in the list is our link to the Topics page ❸.

Here's the next part of the navigation bar:

base.html

```
            --snip--
      </ul>
❶     <ul class="navbar-nav ml-auto">
❷       {% if user.is_authenticated %}
          <li class="nav-item">
❸           <span class="navbar-text"}">Hello, {{ user.username }}.</span>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:logout' %}">Log out</a>
          </li>
        {% else %}
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:register' %}">Register</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:login' %}">Log in</a></li>
        {% endif %}
      </ul>
❹   </div>

  </nav>
```

At ❶ we begin a new set of links by using another opening `<ul>` tag. You can have as many groups of links as you need on a page. This will be the group of links related to login and registration that appears on the right side of the navigation bar. The selector `ml-auto` is short for *margin-left-automatic*: this selector examines the other elements in the navigation bar and works out a left margin that pushes this group of links to the right side of the screen.

The `if` block at ❷ is the same conditional block we used earlier to display appropriate messages to users depending on whether or not they're logged in. The block is a little longer now because some styling rules are inside the conditional tags. At ❸ is a `<span>` element. The *span element* styles pieces of text, or elements of a page, that are part of a longer line. Whereas div elements create their own division in a page, span elements are continuous within a larger section. This can be confusing at first, because many pages have deeply nested div elements. Here, we're using the span element to style informational text on the navigation bar, such as the logged-in user's name. We want this information to appear different from a link, so users aren't tempted to click these elements.

At ❹ we close the div element that contains the parts of the navigation bar that will collapse on narrow screens, and at the end of this section we close the navigation bar overall. If you wanted to add more links to the navigation bar, you'd add another `<li>` item to any of the `<ul>` groups that we've defined in the navigation bar by using identical styling directives as what you've seen here.

There's still a bit more we need to add to *base.html*. We need to define two blocks that the individual pages can use to place the content specific to those pages.

Defining the Main Part of the Page

The rest of *base.html* contains the main part of the page:

base.html

```
      --snip--
    </nav>

❶   <main role="main" class="container">
❷     <div class="pb-2 mb-2 border-bottom">
        {% block page_header %}{% endblock page_header %}
      </div>
❸     <div>
        {% block content %}{% endblock content %}
      </div>
    </main>

  </body>

  </html>
```

At ❶ we open a `<main>` tag. The *main* element is used for the most significant part of the body of a page. Here we assign the bootstrap selector

container, which is a simple way to group elements on a page. We'll place two div elements in this container.

The first div element ❷ contains a page_header block. We'll use this block to title most pages. To make this section stand out from the rest of the page, we place some padding below the header. *Padding* refers to space between an element's content and its border. The selector pb-2 is a bootstrap directive that provides a moderate amount of padding at the bottom of the styled element. A *margin* is the space between an element's border and other elements on the page. We want a border only on the bottom of the page, so we use the selector border-bottom, which provides a thin border at the bottom of the page_header block.

At ❸ we define one more div element, which contains the block content. We don't apply any specific style to this block, so we can style the content of any page as we see fit for that page. We end the *base.html* file with closing tags for the main, body, and html elements.

When you load Learning Log's home page in a browser, you should see a professional-looking navigation bar that matches the one shown in Figure 20-1. Try resizing the window so it's very narrow; a button should replace the navigation bar. Click the button, and all the links should appear in a drop-down list.

Styling the Home Page Using a Jumbotron

To update the home page, we'll use a Bootstrap element called a *jumbotron*, which is a large box that stands out from the rest of the page and can contain anything you want. Typically, it's used on home pages to hold a brief description of the overall project and a call to action that invites the viewer to get involved.

Here's the revised *index.html* file:

index.html

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

❶ {% block page_header %}
❷   <div class="jumbotron">
❸     <h1 class="display-3">Track your learning.</h1>

❹     <p class="lead">Make your own Learning Log, and keep a list of the
          topics you're learning about. Whenever you learn something new
          about a topic, make an entry summarizing what you've learned.</p>

❺     <a class="btn btn-lg btn-primary" href="{% url 'users:register' %}"
          role="button">Register &raquo;</a>
    </div>
❻ {% endblock page_header %}
```

At ❶ we tell Django that we're about to define what goes in the page_header block. A jumbotron is just a div element with a set of styling directives applied to it ❷. The jumbotron selector applies this group of styling directives from the Bootstrap library to this element.

Inside the jumbotron are three elements. The first is a short message, *Track your learning*, that gives first-time visitors a sense of what Learning Log does. The `h1` class is a first-level header, and the `display-3` selector adds a thinner and taller look to this particular header ❸. At ❹ we include a longer message that provides more information about what the user can do with their learning log.

Rather than just using a text link, we create a button at ❺ that invites users to register their Learning Log account. This is the same link as in the header, but the button stands out on the page and shows the viewer what they need to do to start using the project. The selectors you see here style this as a large button that represents a call to action. The code `&raquo;` is an *HTML entity* that looks like two right angle brackets combined (>>). At ❻ we close the `page_header` block. We aren't adding any more content to this page, so we don't need to define the `content` block on this page.

The index page now looks like Figure 20-1 and is a significant improvement over our unstyled project.

Styling the Login Page

We've refined the overall appearance of the login page but not the login form yet. Let's make the form look consistent with the rest of the page by modifying the *login.html* file:

login.html

```
{% extends "learning_logs/base.html" %}
❶ {% load bootstrap4 %}

❷ {% block page_header %}
    <h2>Log in to your account.</h2>
  {% endblock page_header %}

  {% block content %}
❸   <form method="post" action="{% url 'users:login' %}" class="form">
      {% csrf_token %}
❹     {% bootstrap_form form %}
❺     {% buttons %}
        <button name="submit" class="btn btn-primary">Log in</button>
      {% endbuttons %}

      <input type="hidden" name="next"
        value="{% url 'learning_logs:index' %}" />
    </form>

  {% endblock content %}
```

At ❶ we load the bootstrap4 template tags into this template. At ❷ we define the `page_header` block, which tells the user what the page is for. Notice that we've removed the `{% if form.errors %}` block from the template; django-bootstrap4 manages form errors automatically.

At ❸ we add a `class="form"` attribute, and then we use the template tag `{% bootstrap_form %}` when we display the form ❹; this replaces the `{{ form .as_p }}` tag we were using in Chapter 19. The `{% booststrap_form %}` template

tag inserts Bootstrap style rules into the form's individual elements as the form is rendered. At ❺ we open a bootstrap4 template tag `{% buttons %}`, which adds Bootstrap styling to buttons.

Figure 20-2 shows the login form now. The page is much cleaner and has consistent styling and a clear purpose. Try logging in with an incorrect username or password; you'll see that even the error messages are styled consistently and integrate well with the overall site.

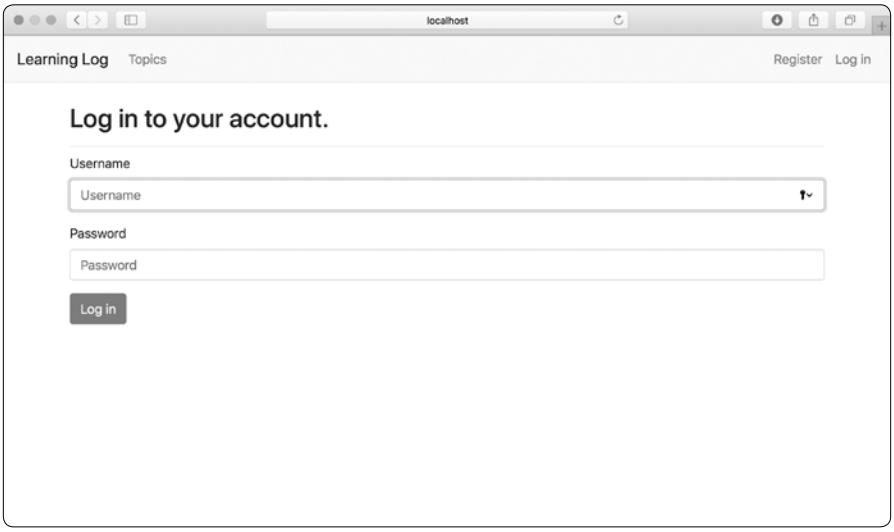


Figure 20-2: The login page styled with Bootstrap

Styling the Topics Page

Let's make sure the pages for viewing information are styled appropriately as well, starting with the topics page:

topics.html

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

❶ {% block page_header %}
    <h1>Topics</h1>
  {% endblock page_header %}

  {% block content %}
    <ul>
      {% for topic in topics %}
❷       <li><h3>
          <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
        </h3></li>
      {% empty %}
        <li><h3>No topics have been added yet.</h3></li>
      {% endfor %}
    </ul>

❸   <h3><a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a></h3>
  {% endblock content %}
```

We don't need the `{% load bootstrap4 %}` tag, because we're not using any custom bootstrap4 template tags in this file. We move the heading *Topics* into the `page_header` block and give it a header styling instead of using the simple paragraph tag ❶. We style each topic as an `<h3>` element to make them a little larger on the page ❷ and do the same for the link to add a new topic ❸.

Styling the Entries on the Topic Page

The topic page has more content than most pages, so it needs a bit more work. We'll use Bootstrap's card component to make each entry stand out. A *card* is a div with a set of flexible, predefined styles that's perfect for displaying a topic's entries:

topic.html

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

❶ {% block page_header %}
    <h3>{{ topic }}</h3>
  {% endblock page_header %}

  {% block content %}
    <p>
      <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
    </p>

    {% for entry in entries %}
❷     <div class="card mb-3">
❸       <h4 class="card-header">
          {{ entry.date_added|date:'M d, Y H:i' }}
❹         <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">
              edit entry</a></small>
        </h4>
❺       <div class="card-body">
          {{ entry.text|linebreaks }}
        </div>
      </div>
    {% empty %}
      <p>There are no entries for this topic yet.</p>
    {% endfor %}

  {% endblock content %}
```

We first place the topic in the `page_header` block ❶. Then we delete the unordered list structure previously used in this template. Instead of making each entry a list item, we create a div element with the selector `card` at ❷. This card has two nested elements: one to hold the timestamp and the link to edit the entry, and another to hold the body of the entry.

The first element in the card is a header, which is an `<h4>` element with the selector `card-header` ❸. This card header contains the date the entry was made and a link to edit the entry. The `<small>` tag around the `edit_entry` link

makes it appear a little smaller than the timestamp ❹. The second element is a div with the selector `card-body` ❺, which places the text of the entry in a simple box on the card. Notice that the Django code for including the information on the page hasn't changed; only the elements that affect the appearance of the page have changed.

Figure 20-3 shows the topic page with its new look. Learning Log's functionality hasn't changed, but it looks more professional and inviting to users now.

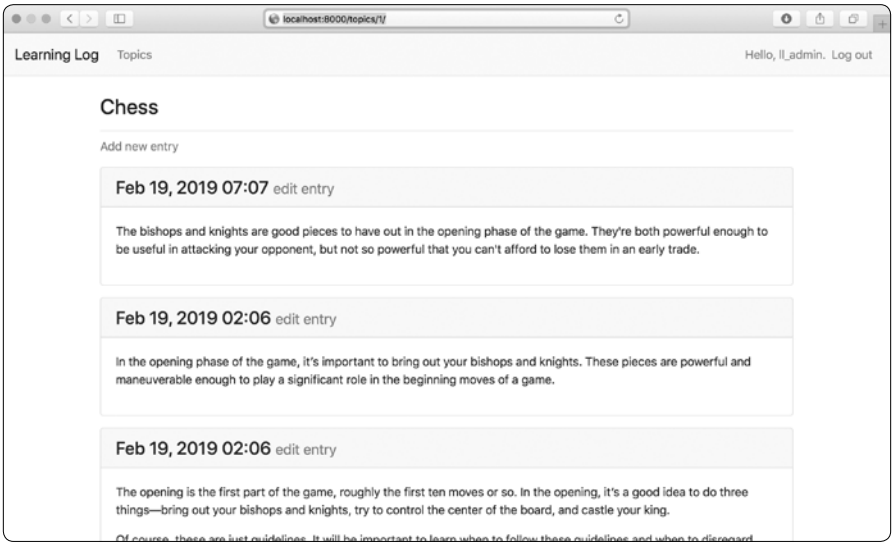


Figure 20-3: The topic page with Bootstrap styling

NOTE *If you want to use a different Bootstrap template, follow a similar process to what we've done so far in this chapter. Copy the template you want to use into* base.html, *and modify the elements that contain actual content so the template displays your project's information. Then use Bootstrap's individual styling tools to style the content on each page.*

TRY IT YOURSELF

20-1. Other Forms: We applied Bootstrap's styles to the `login` page. Make similar changes to the rest of the form-based pages including `new_topic`, `new_entry`, `edit_entry`, and `register`.

20-2. Stylish Blog: Use Bootstrap to style the Blog project you created in Chapter 19.

Deploying Learning Log

Now that we have a professional-looking project, let's deploy it to a live server so anyone with an internet connection can use it. We'll use Heroku, a web-based platform that allows you to manage the deployment of web applications. We'll get Learning Log up and running on Heroku.

Making a Heroku Account

To make an account, go to *https://heroku.com/* and click one of the signup links. It's free to make an account, and Heroku has a free tier that allows you to test your projects in live deployment before properly deploying them.

NOTE *Heroku's free tier has limits, such as the number of apps you can deploy and how often people can visit your app. But these limits are generous enough to let you practice deploying apps without any cost.*

Installing the Heroku CLI

To deploy and manage a project on Heroku's servers, you'll need the tools available in the Heroku Command Line Interface (CLI). To install the latest version of the Heroku CLI, visit *https://devcenter.heroku.com/articles/heroku-cli/* and follow the instructions for your operating system. The instructions will include either a one-line terminal command or an installer you can download and run.

Installing Required Packages

You'll also need to install three packages that help serve Django projects on a live server. In an active virtual environment, issue the following commands:

```
(ll_env)learning_log$ pip install psycopg2==2.7.*
(ll_env)learning_log$ pip install django-heroku
(ll_env)learning_log$ pip install gunicorn
```

The `psycopg2` package is required to manage the database that Heroku uses. The `django-heroku` package handles almost the entire configuration our app needs to run properly on Heroku servers. This includes managing the database and storing static files in a place where they can be served properly. *Static files* contain style rules and JavaScript files. The `gunicorn` package provides a server capable of serving apps in a live environment.

Creating a requirements.txt File

Heroku needs to know which packages our project depends on, so we'll use pip to generate a file listing them. Again, from an active virtual environment, issue the following command:

```
(ll_env)learning_log$ pip freeze > requirements.txt
```

The `freeze` command tells pip to write the names of all the packages currently installed in the project into the file *requirements.txt*. Open this file to see the packages and version numbers installed in your project:

requirements.txt

```
dj-database-url==0.5.0
Django==2.2.0
django-bootstrap4==0.0.7
django-heroku==0.3.1
gunicorn==19.9.0
psycopg2==2.7.7
pytz==2018.9
sqlparse==0.2.4
whitenoise==4.1.2
```

Learning Log already depends on eight different packages with specific version numbers, so it requires a specific environment to run properly. (We installed four of these packages manually, and four of them were installed automatically as dependencies of these packages.)

When we deploy Learning Log, Heroku will install all the packages listed in *requirements.txt*, creating an environment with the same packages we're using locally. For this reason, we can be confident the deployed project will behave the same as it does on our local system. This is a huge advantage as you start to build and maintain various projects on your system.

NOTE *If a package is listed on your system but the version number differs from what's shown here, keep the version you have on your system.*

Specifying the Python Runtime

Unless you specify a Python version, Heroku will use its current default version of Python. Let's make sure Heroku uses the same version of Python we're using. In an active virtual environment, issue the command `python --version`:

```
(ll_env)learning_log$ python --version
Python 3.7.2
```

In this example I'm running Python 3.7.2. Make a new file called *runtime.txt* in the same directory as *manage.py*, and enter the following:

runtime.txt

```
python-3.7.2
```

This file should contain one line with your Python version specified in the format shown; make sure you enter `python` in lowercase, followed by a hyphen, followed by the three-part version number.

NOTE *If you get an error reporting that the Python runtime you requested isn't available, go to* https://devcenter.heroku.com/categories/language-support/ *and look for a link to* Specifying a Python Runtime. *Scan through the article to find the available runtimes, and use the one that most closely matches your Python version.*

Modifying settings.py for Heroku

Now we need to add a section at the end of *settings.py* to define some specific settings for the Heroku environment:

settings.py

```
--snip--
# My settings
LOGIN_URL = 'users:login'

# Heroku settings.
import django_heroku
django_heroku.settings(locals())
```

Here we import the `django_heroku` module and call the `settings()` function. This function modifies some settings that need specific values for the Heroku environment.

Making a Procfile to Start Processes

A *Procfile* tells Heroku which processes to start to properly serve the project. Save this one-line file as *Procfile*, with an uppercase *P* and no file extension, in the same directory as *manage.py*.

Here's the line that goes in *Procfile*:

Procfile

```
web: gunicorn learning_log.wsgi --log-file -
```

This line tells Heroku to use gunicorn as a server and to use the settings in *learning_log/wsgi.py* to launch the app. The `log-file` flag tells Heroku the kinds of events to log.

Using Git to Track the Project's Files

As discussed in Chapter 17, Git is a version control program that allows you to take a snapshot of the code in your project each time you implement a new feature successfully. If anything goes wrong, you can easily return to the last working snapshot of your project; for example, if you accidentally introduce a bug while working on a new feature. Each snapshot is called a *commit*.

Using Git, you can try implementing new features without worrying about breaking your project. When you're deploying to a live server, you need to make sure you're deploying a working version of your project. To read more about Git and version control, see Appendix D.

Installing Git

Git may already be installed on your system. To find out if Git is already installed, open a new terminal window and issue the command `git --version`:

```
(ll_env)learning_log$ git --version
git version 2.17.0
```

If you get an error message for some reason, see the installation instructions for Git in Appendix D.

Configuring Git

Git keeps track of who makes changes to a project, even when only one person is working on the project. To do this, Git needs to know your username and email. You must provide your username, but you can make up an email for your practice projects:

```
(ll_env)learning_log$ git config --global user.name "ehmatthes"
(ll_env)learning_log$ git config --global user.email "eric@example.com"
```

If you forget this step, Git will prompt you for this information when you make your first commit.

Ignoring Files

We don't need Git to track every file in the project, so we'll tell it to ignore some files. Create a file called *.gitignore* in the folder that contains *manage.py*. Notice that this filename begins with a dot and has no file extension. Here's the code that goes in *.gitignore*:

.gitignore

```
ll_env/
__pycache__/
*.sqlite3
```

We tell Git to ignore the entire *ll_env* directory, because we can re-create it automatically at any time. We also don't track the *__pycache__* directory, which contains the *.pyc* files that are created automatically when Django runs the *.py* files. We don't track changes to the local database, because it's a bad habit: if you're ever using SQLite on a server, you might accidentally overwrite the live database with your local test database when you push the project to the server. The asterisk in `*.sqlite3` tells Git to ignore any file that ends with the extension *.sqlite3*.

NOTE *If you're using macOS, add* `.DS_Store` *to your* .gitignore *file. This is a file that stores information about folder settings on macOS, and it has nothing to do with this project.*

Making Hidden Files Visible

Most operating systems hide files and folders that begin with a dot, such as *.gitignore*. When you open a file browser or try to open a file from an application such as Sublime Text, you won't see these kinds of files by default. But as a programmer, you'll need to see them. Here's how to view hidden files, depending on your operating system:

- On Windows, open Windows Explorer, and then open a folder such as *Desktop*. Click the **View** tab, and make sure **File name extensions** and **Hidden items** are checked.

- On macOS, you can press ⌘-SHIFT-. (dot) in any file browser window to see hidden files and folders.
- On Linux systems such as Ubuntu, you can press CTRL-H in any file browser to display hidden files and folders. To make this setting permanent, open a file browser such as Nautilus and click the options tab (indicated by three lines). Select the **Show Hidden Files** checkbox.

Committing the Project

We need to initialize a Git repository for Learning Log, add all the necessary files to the repository, and commit the initial state of the project. Here's how to do that:

```
❶ (ll_env)learning_log$ git init
  Initialized empty Git repository in /home/ehmatthes/pcc/learning_log/.git/
❷ (ll_env)learning_log$ git add .
❸ (ll_env)learning_log$ git commit -am "Ready for deployment to heroku."
  [master (root-commit) 79fef72] Ready for deployment to heroku.
   45 files changed, 712 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 Procfile
   --snip--
   create mode 100644 users/views.py
❹ (ll_env)learning_log$ git status
  On branch master
  nothing to commit, working tree clean
  (ll_env)learning_log$
```

At ❶ we issue the `git init` command to initialize an empty repository in the directory containing Learning Log. At ❷ we use the `git add .` command, which adds all the files that aren't being ignored to the repository. (Don't forget the dot.) At ❸ we issue the command `git commit -am` *`commit message`*: the `-a` flag tells Git to include all changed files in this commit, and the `-m` flag tells Git to record a log message.

Issuing the `git status` command ❹ indicates that we're on the *master* branch and that our working tree is *clean*. This is the status you'll want to see any time you push your project to Heroku.

Pushing to Heroku

We're finally ready to push the project to Heroku. In an active virtual environment, issue the following commands:

```
❶ (ll_env)learning_log$ heroku login
  heroku: Press any key to open up the browser to login or q to exit:
  Logging in... done
  Logged in as eric@example.com
❷ (ll_env)learning_log$ heroku create
  Creating app... done, ⬢ secret-lowlands-82594
  https://secret-lowlands-82594.herokuapp.com/ |
      https://git.heroku.com/secret-lowlands-82594.git
```

```
❸ (ll_env)learning_log$ git push heroku master
  --snip--
  remote: -----> Launching...
  remote:        Released v5
❹ remote:        https://secret-lowlands-82594.herokuapp.com/ deployed to Heroku
  remote: Verifying deploy... done.
  To https://git.heroku.com/secret-lowlands-82594.git
   * [new branch]      master -> master
  (ll_env)learning_log$
```

First you issue the `heroku login` command, which will take you to a page in your browser where you can log in to your Heroku account ❶. Then you tell Heroku to build an empty project ❷. Heroku generates a name made up of two words and a number; you can change this later on. Next, we issue the command `git push heroku master` ❸, which tells Git to push the master branch of the project to the repository Heroku just created. Then Heroku builds the project on its servers using these files. At ❹ is the URL we'll use to access the live project, which we can change along with the project name.

When you've issued these commands, the project is deployed but not fully configured. To check that the server process started correctly, use the `heroku ps` command:

```
(ll_env)learning_log$ heroku ps
❶ Free dyno hours quota remaining this month: 450h 44m (81%)
  Free dyno usage for this app: 0h 0m (0%)
  For more information on dyno sleeping and how to upgrade, see:
  https://devcenter.heroku.com/articles/dyno-sleeping
❷ === web (Free): gunicorn learning_log.wsgi --log-file - (1)
  web.1: up 2019/02/19 23:40:12 -0900 (~ 10m ago)
  (ll_env)learning_log$
```

The output shows how much more time the project can be active in the next month ❶. At the time of this writing, Heroku allows free deployments to be active for up to 550 hours in a month. If a project exceeds this limit, a standard server error page will display; we'll customize this error page shortly. At ❷ we see that the process defined in *Procfile* has been started.

Now we can open the app in a browser using the command `heroku open`:

```
(ll_env)learning_log$ heroku open
(ll_env)learning_log$
```

This command spares you from opening a browser and entering the URL Heroku showed you, but that's another way to open the site. You should see the home page for Learning Log, styled correctly. However, you can't use the app yet because we haven't set up the database.

NOTE *Heroku's deployment process changes from time to time. If you have any unresolvable issues, look at Heroku's documentation for help. Go to* https://devcenter.heroku.com/*, click* ***Python****, and look for a link to* Get Started with Python *or* Deploying Python and Django Apps on Heroku. *If you don't understand what you see there, check out the suggestions in Appendix C.*

Setting Up the Database on Heroku

We need to run `migrate` once to set up the live database and apply all the migrations we generated during development. You can run Django and Python commands on a Heroku project using the command `heroku run`. Here's how to run `migrate` on the Heroku deployment:

```
❶ (ll_env)learning_log$ heroku run python manage.py migrate
❷ Running 'python manage.py migrate' on ⬢ secret-lowlands-82594... up, run.3060
    --snip--
❸ Running migrations:
    --snip--
    Applying learning_logs.0001_initial... OK
    Applying learning_logs.0002_entry... OK
    Applying learning_logs.0003_topic_owner... OK
    Applying sessions.0001_initial... OK
  (ll_env)learning_log$
```

We first issue the command `heroku run python manage.py migrate` ❶. Heroku then creates a terminal session to run the `migrate` command ❷. At ❸ Django applies the default migrations and the migrations we generated during the development of Learning Log.

Now when you visit your deployed app, you should be able to use it just as you did on your local system. But you won't see any of the data you entered on your local deployment, including your superuser account, because we didn't copy the data to the live server. This is normal practice: you don't usually copy local data to a live deployment because the local data is usually test data.

You can share your Heroku link to let anyone use your version of Learning Log. In the next section, we'll complete a few more tasks to finish the deployment process and set you up to continue developing Learning Log.

Refining the Heroku Deployment

Now we'll refine the deployment by creating a superuser, just as we did locally. We'll also make the project more secure by changing the setting `DEBUG` to `False`, so users won't see any extra information in error messages that they could use to attack the server.

Creating a Superuser on Heroku

You've already seen that we can run one-off commands using the `heroku run` command. But you can also run commands by opening a Bash terminal session while connected to the Heroku server using the command `heroku run bash`. Bash is the language that runs in many Linux terminals. We'll use the Bash terminal session to create a superuser so we can access the admin site on the live app:

```
(ll_env)learning_log$ heroku run bash
Running 'bash' on ⬢ secret-lowlands-82594... up, run.9858
❶ ~ $ ls
learning_log learning_logs manage.py Procfile requirements.txt runtime.txt
```

```
staticfiles users
❷ ~ $ python manage.py createsuperuser
Username (leave blank to use ' u47318'): ll_admin
Email address:
Password:
Password (again):
Superuser created successfully.
❸ ~ $ exit
exit
(ll_env)learning_log$
```

At ❶ we run `ls` to see which files and directories exist on the server, which should be the same files we have on our local system. You can navigate this filesystem like any other.

NOTE *Windows users will use the same commands shown here (such as `ls` instead of `dir`), because you're running a Linux terminal through a remote connection.*

At ❷ we run the command to create a superuser, which outputs the same prompts we saw on our local system when we created a superuser in Chapter 18. When you're finished creating the superuser in this terminal session, run the **`exit`** command to return to your local system's terminal session ❸.

Now you can add */admin/* to the end of the URL for the live app and log in to the admin site. For me, the URL is *https://secret-lowlands-82594.herokuapp.com/admin/*.

If others have already started using your project, be aware that you'll have access to all of their data! Don't take this lightly, and users will continue to trust you with their data.

Creating a User-Friendly URL on Heroku

Most likely, you'll want your URL to be friendlier and more memorable than *https://secret-lowlands-82594.herokuapp.com/*. You can rename the app using a single command:

```
(ll_env)learning_log$ heroku apps:rename learning-log
Renaming secret-lowlands-82594 to learning-log-2e... done
https://learning-log.herokuapp.com/ | https://git.heroku.com/learning-log.git
Git remote heroku updated
 ⬢  Don't forget to update git remotes for all other local checkouts of the app.
(ll_env)learning_log$
```

You can use letters, numbers, and dashes when naming your app, and call it whatever you want, as long as no one else has claimed the name. This deployment now lives at *https://learning-log.herokuapp.com/*. The project is no longer available at the previous URL; the `apps:rename` command completely moves the project to the new URL.

NOTE *When you deploy your project using Heroku's free service, Heroku puts your deployment to sleep if it hasn't received any requests after a certain amount of time or if it's been too active for the free tier. The first time a user accesses the site after it's been sleeping, it will take longer to load, but the server will respond to subsequent requests more quickly. This is how Heroku can afford to offer free deployments.*

Securing the Live Project

One glaring security issue exists in the way our project is currently deployed: the setting `DEBUG=True` in *settings.py*, which provides debug messages when errors occur. Django's error pages give you vital debugging information when you're developing a project; however, they give way too much information to attackers if you leave them enabled on a live server.

We'll control whether debugging information is shown on the live site by setting an environment variable. *Environment variables* are values set in a specific environment. This is one of the ways sensitive information is stored on a server, keeping it separate from the rest of the project's code.

Let's modify *settings.py* so it looks for an environment variable when the project is running on Heroku:

settings.py

```
--snip--
# Heroku settings.
import django_heroku
django_heroku.settings(locals())

if os.environ.get('DEBUG') == 'TRUE':
    DEBUG = True
elif os.environ.get('DEBUG') == 'FALSE':
    DEBUG = False
```

The method `os.environ.get()` reads the value associated with a specific environment variable in any environment where the project is running. If the variable we're asking for is set, the method returns its value; if it's not set, the method returns `None`. Using environment variables to store Boolean values can be confusing. In most cases, environment variables are stored as strings, and you have to be careful about this. Consider this snippet from a simple Python terminal session:

```
>>> bool('False')
True
```

The Boolean value of the string `'False'` is `True`, because any non-empty string evaluates to `True`. So we'll use the strings `'TRUE'` and `'FALSE'`, in all capitals, to be clear that we're not storing Python's actual `True` and `False` Boolean values. When Django reads in the environment variable with the key `'DEBUG'` on Heroku, we'll set `DEBUG` to `True` if the value is `'TRUE'` and `False` if the value is `'FALSE'`.

Committing and Pushing Changes

Now we need to commit the changes made to *settings.py* to the Git repository, and then push the changes to Heroku. Here's a terminal session showing this process:

```
❶ (ll_env)learning_log$ git commit -am "Set DEBUG based on environment variables."
[master 3427244] Set DEBUG based on environment variables.
 1 file changed, 4 insertions(+)
❷ (ll_env)learning_log$ git status
On branch master
nothing to commit, working tree clean
(ll_env)learning_log$
```

We issue the `git commit` command with a short but descriptive commit message ❶. Remember that the `-am` flag makes sure Git commits all the files that have changed and records the log message. Git recognizes that one file has changed and commits this change to the repository.

At ❷ the status shows that we're working on the master branch of the repository and that there are now no new changes to commit. It's essential that you check the status for this message before pushing to Heroku. If you don't see this message, some changes haven't been committed, and those changes won't be pushed to the server. You can try issuing the `commit` command again, but if you're not sure how to resolve the issue, read through Appendix D to better understand how to work with Git.

Now let's push the updated repository to Heroku:

```
(ll_env)learning_log$ git push heroku master
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing requirements with pip
--snip--
remote: -----> Launching...
remote:        Released v6
remote:        https://learning-log.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/learning-log.git
   144f020..d5075a1  master -> master
(ll_env)learning_log$
```

Heroku recognizes that the repository has been updated, and it rebuilds the project to make sure all the changes have been taken into account. It doesn't rebuild the database, so we won't have to run `migrate` for this update.

Setting Environment Variables on Heroku

Now we can set the value we want for `DEBUG` in *settings.py* through Heroku. The command `heroku config:set` sets an environment variable for us:

```
(ll_env)learning_log$ heroku config:set DEBUG='FALSE'
Setting DEBUG and restarting ⬢ learning-log... done, v7
DEBUG: FALSE
(ll_env)learning_log$
```

Whenever you set an environment variable on Heroku, it automatically restarts the project so the environment variable can take effect.

To check that the deployment is more secure now, enter your project's URL with a path we haven't defined. For example, try to visit *http://learning-log.herokuapp.com/letmein/*. You should see a generic error page on your live deployment that doesn't give away any specific information about the project. If you try the same request on the local version of Learning Log at *http://localhost:8000/letmein/*, you should see the full Django error page. The result is perfect: you'll see informative error messages when you're developing the project further on your own system. But users on the live site won't see critical information about the project's code.

If you're just deploying an app and you're troubleshooting the initial deployment, you can run `heroku config:set DEBUG='TRUE'` and temporarily see a full error report on the live site. Just make sure you reset the value to `'FALSE'` once you've finished troubleshooting. Also, be careful not to do this once users are regularly accessing your site.

Creating Custom Error Pages

In Chapter 19, we configured Learning Log to return a 404 error if the user requests a topic or entry that doesn't belong to them. You've probably seen some 500 server errors (internal errors) by this point as well. A 404 error usually means your Django code is correct, but the object being requested doesn't exist; a 500 error usually means there's an error in the code you've written, such as an error in a function in *views.py*. Currently, Django returns the same generic error page in both situations. But we can write our own 404 and 500 error page templates that match Learning Log's overall appearance. These templates must go in the root template directory.

Making Custom Templates

In the outermost *learning_log* folder, make a new folder called *templates*. Then make a new file called *404.html*; the path to this file should be *learning_log/templates/404.html*. Here's the code for this file:

404.html

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

{% block page_header %}
  <h2>The item you requested is not available. (404)</h2>
{% endblock page_header %}
```

This simple template provides the generic 404 error page information but is styled to match the rest of the site.

Make another file called *500.html* using the following code:

500.html

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

{% block page_header %}
  <h2>There has been an internal error. (500)</h2>
{% endblock page_header %}
```

These new files require a slight change to *settings.py*.

settings.py

```
--snip--
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        --snip--
    },
]
--snip--
```

This change tells Django to look in the root template directory for the error page templates.

Viewing the Error Pages Locally

If you want to see what the error pages look like on your system before pushing them to Heroku, you'll first need to set `Debug=False` on your local settings to suppress the default Django debug pages. To do so, make the following change to *settings.py* (make sure you're working in the part of *settings.py* that applies to the local environment, not the part that applies to Heroku):

settings.py

```
--snip--
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False
--snip--
```

Now request a topic or entry that doesn't belong to you to see the 404 error page. To test the 500 error page, request a topic or entry that doesn't exist. For example, the URL *http://localhost:8000/topics/999/* should generate a 500 error unless you've generated 999 example topics already!

When you're finished checking the error pages, set the local value of `DEBUG` back to `True` to further develop Learning Log. (Make sure you don't change the way `DEBUG` is handled in the section that manages settings in the Heroku environment.)

NOTE *The 500 error page won't show any information about the user who's logged in, because Django doesn't send any context information in the response when there's a server error.*

Pushing the Changes to Heroku

Now we need to commit the error page changes we just made, and push them live to Heroku:

```
❶ (ll_env)learning_log$ git add .
❷ (ll_env)learning_log$ git commit -am "Added custom 404 and 500 error pages."
 3 files changed, 15 insertions(+), 10 deletions(-)
 create mode 100644 templates/404.html
 create mode 100644 templates/500.html
❸ (ll_env)learning_log$ git push heroku master
--snip--
remote: Verifying deploy.... done.
To https://git.heroku.com/learning-log.git
   d5075a1..4bd3b1c  master -> master
(ll_env)learning_log$
```

We issue the `git add .` command at ❶ because we created some new files in the project, so we need to tell Git to start tracking these files. Then we commit the changes ❷ and push the updated project to Heroku ❸.

Now when an error page appears, it should have the same styling as the rest of the site, making for a smoother user experience when errors arise.

Using the get_object_or_404() Method

At this point, if a user manually requests a topic or entry that doesn't exist, they'll get a 500 server error. Django tries to render the nonexistent page, but it doesn't have enough information to do so, and the result is a 500 error. This situation is more accurately handled as a 404 error, and we can implement this behavior using the Django shortcut function `get_object_or_404()`. This function tries to get the requested object from the database, but if that object doesn't exist, it raises a 404 exception. We'll import this function into *views.py* and use it in place of `get()`:

views.py

```
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.auth.decorators import login_required
--snip--
@login_required
def topic(request, topic_id):
    """Show a single topic and all its entries."""
    topic = get_object_or_404(Topic, id=topic_id)
    # Make sure the topic belongs to the current user.
    --snip--
```

Now when you request a topic that doesn't exist (for example, *http://localhost:8000/topics/999/*), you'll see a 404 error page. To deploy this change, make a new commit and then push the project to Heroku.

Ongoing Development

You might want to further develop Learning Log after your initial push to a live server or develop your own projects to deploy. There's a fairly consistent process for updating projects.

First, you'll make any changes needed to your local project. If your changes result in any new files, add those files to the Git repository using the command `git add .` (be sure to include the dot at the end of the command). Any change that requires a database migration will need this command, because each migration generates a new migration file.

Second, commit the changes to your repository using `git commit -am "commit message"`. Then push your changes to Heroku using the command `git push heroku master`. If you migrated your database locally, you'll need to migrate the live database as well. You can either use the one-off command `heroku run python manage.py migrate`, or open a remote terminal session with `heroku run bash` and run the command `python manage.py migrate`. Then visit your live project, and make sure the changes you expect to see have taken effect.

It's easy to make mistakes during this process, so don't be surprised when something goes wrong. If the code doesn't work, review what you've done and try to spot the mistake. If you can't find the mistake or you can't figure out how to undo the mistake, refer to the suggestions for getting help in Appendix C. Don't be shy about asking for help: everyone else learned to build projects by asking the same questions you're likely to ask, so someone will be happy to help you. Solving each problem that arises helps you steadily develop your skills until you're building meaningful, reliable projects and you're answering other people's questions as well.

The SECRET_KEY Setting

Django uses the value of the `SECRET_KEY` setting in *settings.py* to implement a number of security protocols. In this project, we've committed our settings file to the repository with the `SECRET_KEY` setting included. This is fine for a practice project, but the `SECRET_KEY` setting should be handled more carefully for a production site. If you build a project that's getting meaningful use, make sure you research how to handle your `SECRET_KEY` setting more securely.

Deleting a Project on Heroku

It's great practice to run through the deployment process a number of times with the same project or with a series of small projects to get the hang of deployment. But you'll need to know how to delete a project that's been deployed. Heroku also limits the number of projects you can host for free, and you don't want to clutter your account with practice projects.

Log in to the Heroku website (*https://heroku.com/*); you'll be redirected to a page showing a list of your projects. Click the project you want to delete. You'll see a new page with information about the project. Click the **Settings**

link, and scroll down until you see a link to delete the project. This action can't be reversed, so Heroku will ask you to confirm the request for deletion by manually entering the project's name.

If you prefer working from a terminal, you can also delete a project by issuing the `destroy` command:

```
(ll_env)learning_log$ heroku apps:destroy --app appname
```

Here, *appname* is the name of your project, which is either something like `secret-lowlands-82594` or `learning-log` if you've renamed the project. You'll be prompted to reenter the project name to confirm the deletion.

NOTE *Deleting a project on Heroku does nothing to your local version of the project. If no one has used your deployed project and you're just practicing the deployment process, it's perfectly reasonable to delete your project on Heroku and redeploy it.*

TRY IT YOURSELF

20-3. Live Blog: Deploy the Blog project you've been working on to Heroku. Make sure you set `DEBUG` to `False`, so users don't see the full Django error pages when something goes wrong.

20-4. More 404s: The `get_object_or_404()` function should also be used in the `new_entry()` and `edit_entry()` views. Make this change, test it by entering a URL like *http://localhost:8000/new_entry/999/*, and check that you see a 404 error.

20-5. Extended Learning Log: Add one feature to Learning Log, and push the change to your live deployment. Try a simple change, such as writing more about the project on the home page. Then try adding a more advanced feature, such as giving users the option of making a topic public. This would require an attribute called `public` as part of the `Topic` model (this should be set to `False` by default) and a form element on the new_topic page that allows the user to change a topic from private to public. You'd then need to migrate the project and revise *views.py* so any topic that's public is visible to unauthenticated users as well. Remember to migrate the live database after you've pushed your changes to Heroku.

Summary

In this chapter, you learned to give your projects a simple but professional appearance using the Bootstrap library and the django-bootstrap4 app. Using Bootstrap, the styles you choose will work consistently on almost any device people use to access your project.

You learned about Bootstrap's templates and used the *Navbar static* template to create a simple look and feel for Learning Log. You used a jumbotron to make a home page's message stand out and learned to style all the pages in a site consistently.

In the final part of the project, you learned how to deploy a project to Heroku's servers so anyone can access it. You made a Heroku account and installed some tools that help manage the deployment process. You used Git to commit the working project to a repository and then pushed the repository to Heroku's servers. Finally, you learned to begin securing your app by setting `DEBUG=False` on the live server.

Now that you've finished Learning Log, you can start building your own projects. Start simple, and make sure the project works before adding complexity. Enjoy your continued learning, and good luck with your projects!