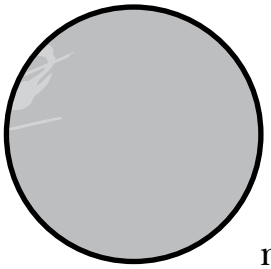
# STYLING AND DEPLOYING AN APP



Learning Log is fully functional now, but it has no styling and runs only on your local machine. In this chapter we'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.

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-bootstrap3 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-bootstrap3 app and show you how to integrate it into a project to make it ready for live deployment.

### The django-bootstrap3 App

We'll use django-bootstrap3 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-bootstrap3, issue the following command in an active virtual environment:

```
(ll_env)learning_log$ pip install django-bootstrap3
--snip--
Successfully installed django-bootstrap3
```

Next, we need to add the following code to include django-boostrap3 in `INSTALLED_APPS` in *settings.py*:

*settings.py*

```
--snip--
INSTALLED_APPS = (
    --snip--
    'django.contrib.staticfiles',

    # Third party apps
    'bootstrap3',

    # My apps
    'learning_logs',
    'users',
)
--snip--
```

Start a new section called *Third party apps* for apps created by other developers and add `'bootstrap3'` to this section. Most apps need to be included in `INSTALLED_APPS`, but to be sure, read the setup instructions for the particular app you're using.

We need django-bootstrap3 to include jQuery, a JavaScript library that enables some of the interactive elements that the Bootstrap template provides. Add this code to the end of *settings.py*:

*settings.py*

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

# Settings for django-bootstrap3
BOOTSTRAP3 = {
    'include_jquery': True,
    }
```

This code spares us from having to download jQuery and place it in the correct location manually.

### Using Bootstrap to Style Learning Log

Bootstrap is basically a large collection of styling tools. It also has a number of templates you can apply to your project to create a particular overall style. If you're just starting out, it's much easier to use these templates than it is to use individual styling tools. To see the templates Bootstrap offers, go to the *Getting Started* section at *http://getbootstrap.com/*; then scroll down to the *Examples* heading, and look for the *Navbars in action* section. We'll use the *Static top navbar* template, which provides a simple top navigation bar, a page header, and a container for the content of the page.

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.

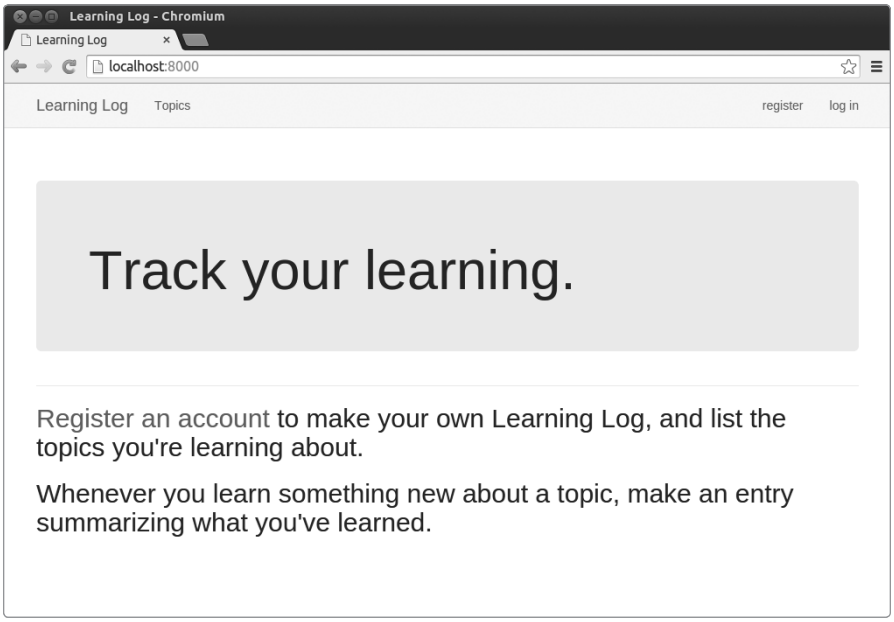



*Figure 20-1: The Learning Log home page using Bootstrap*

Now that you know the result we're after, the sections that follow will be easier to understand.

## 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 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 bootstrap3 %}

❷ <!DOCTYPE html>
❸ <html lang="en">
❹   <head>
      <meta charset="utf-8">
      <meta http-equiv="X-UA-Compatible" content="IE=edge">
      <meta name="viewport" content="width=device-width, initial-scale=1">

❺     <title>Learning Log</title>

❻     {% bootstrap_css %}
      {% bootstrap_javascript %}

❼   </head>
```

At ❶ we load the collection of template tags available in django-bootstrap3. 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 be displayed in the title bar of the browser whenever Learning Log is open.

At ❻ we use one of django-bootstrap3'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

Now we'll define the navigation bar at the top of the page:

```
--snip--
  </head>

  <body>

    <!-- Static navbar -->
```

```
❶   <nav class="navbar navbar-default navbar-static-top">
      <div class="container">

        <div class="navbar-header">
❷         <button type="button" class="navbar-toggle collapsed"
              data-toggle="collapse" data-target="#navbar"
              aria-expanded="false" aria-controls="navbar">
          </button>
❸         <a class="navbar-brand" href="{% url 'learning_logs:index' %}">
              Learning Log</a>
        </div>

❹       <div id="navbar" class="navbar-collapse collapse">
❺         <ul class="nav navbar-nav">
❻           <li><a href="{% url 'learning_logs:topics' %}">Topics</a></li>
          </ul>

❼         <ul class="nav navbar-nav navbar-right">
            {% if user.is_authenticated %}
              <li><a>Hello, {{ user.username }}.</a></li>
              <li><a href="{% url 'users:logout' %}">log out</a></li>
            {% else %}
              <li><a href="{% url 'users:register' %}">register</a></li>
              <li><a href="{% url 'users:login' %}">log in</a></li>
            {% endif %}
❽         </ul>
        </div><!--/.nav-collapse -->

      </div>
    </nav>
```

The first element is the opening `<body>` tag. The body of an HTML file contains the content users will see on a page. At ❶ is a `<nav>` element that indicates the navigation links section of the page. Everything contained in this element is styled according to the Bootstrap style rules defined by the selectors `navbar`, `navbar-default`, and `navbar-static-top`. A *selector* determines which elements on a page a certain style rule applies to.

At ❷ the template defines a button that will appear 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.

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, because it will appear on every page in the project.

At ❹ we define a set of links that lets users navigate the site. A navigation bar is basically a list that starts with `<ul>` ❺, and each link is an item in this list (`<li>`) ❻. To add more links, insert more lines using the following structure:

```
<li><a href="{% url 'learning_logs:title' %}">Title</a></li>
```

This line represents a single link in the navigation bar. The link is taken directly from the previous version of *base.html*.

At ❼ we place a second list of navigation links, this time using the selector `navbar-right`. The `navbar-right` selector styles the set of links so it appears at the right edge of the navigation bar where you typically see login and registration links. Here we'll display the user greeting and logout link or links to register or log in. The rest of the code in this section closes out the elements that contain the navigation bar ❽.

### Defining the Main Part of the Page

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

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

❶   <div class="container">

      <div class="page-header">
❷       {% block header %}{% endblock header %}
      </div>
      <div>
❸       {% block content %}{% endblock content %}
      </div>

    </div> <!-- /container -->

  </body>
</html>
```

At ❶ is an opening div with the class `container`. A *div* is a section of a web page that can be used for any purpose and can be styled with a border, space around the element (margins), space between the contents and the border (padding), background colors, and other style rules. This particular div acts as a container into which we place two elements: a new block called `header` ❷ and the `content` block we used in Chapter 18 ❸. The header block contains information telling the user what kind of information the page holds and what they can do on a page. It has the class `page-header`, which applies a set of style rules to the block. The content block is in a separate div with no specific style classes.

When you load the home page of Learning Log 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 really narrow; the navigation bar should be replaced by a button. Click the button, and all the links should appear in a drop-down list.

**NOTE** *This simplified version of the Bootstrap template should work on most recent browsers. Earlier browsers may not render some styles correctly. The full template, available at* http://getbootstrap.com/getting-started/#examples/*, will work on almost all available browsers.*

### Styling the Home Page Using a Jumbotron

Let's update the home page using the newly defined `header` block and another Bootstrap element called a *jumbotron*—a large box that will stand out from the rest of the page and can contain anything you want. It's typically used on home pages to hold a brief description of the overall project. While we're at it, we'll update the message on the home page as well. Here's *index.html*:

*index.html*

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

❶ {% block header %}
❷   <div class='jumbotron'>
      <h1>Track your learning.</h1>
    </div>
  {% endblock header %}

  {% block content %}
❸   <h2>
      <a href="{% url 'users:register' %}">Register an account</a> to make
      your own Learning Log, and list the topics you're learning about.
    </h2>
    <h2>
      Whenever you learn something new about a topic, make an entry
      summarizing what you've learned.
    </h2>
  {% endblock content %}
```

At ❶ we tell Django that we're about to define what goes in the `header` block. Inside a `jumbotron` element ❷ we place a short tagline, *Track your learning*, to give first-time visitors a sense of what Learning Log does.

At ❸ we add text to provide a little more direction. We invite people to make an account, and we describe the two main actions—add new topics and make topic entries. 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, so let's make the form look consistent with the rest of the page:

*login.html*

```
{% extends "learning_logs/base.html" %}
❶ {% load bootstrap3 %}

❷ {% block header %}
    <h2>Log in to your account.</h2>
  {% endblock 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 bootstrap3 template tags into this template. At ❷ we define the `header` block, which describes what the page is for. Notice that we've removed the `{% if form.errors %}` block from the template; django-bootstrap3 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 individual elements of the form as it's rendered. At ❺ we open a bootstrap3 template tag `{% buttons %}`, which adds Bootstrap styling to buttons.

Figure 20-2 shows the login form as it's rendered 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.

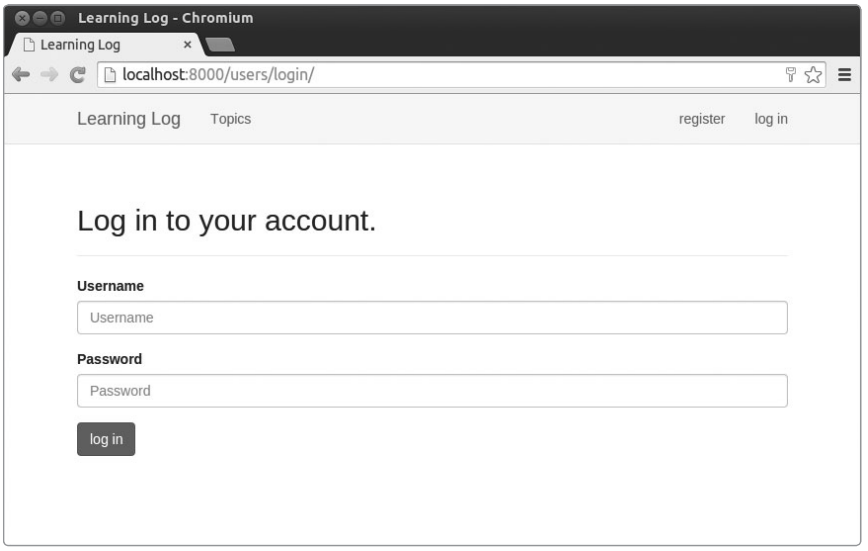



*Figure 20-2: The login page styled with Bootstrap*

### Styling the new_topic Page

Let's make the rest of the pages look consistent as well. We'll update the `new_topic` page next:

*new_topic.html*

```
{% extends "learning_logs/base.html" %}
{% load bootstrap3 %}

❶ {% block header %}
    <h2>Add a new topic:</h2>
  {% endblock header %}

  {% block content %}

❷   <form action="{% url 'learning_logs:new_topic' %}" method='post'
        class="form">

      {% csrf_token %}
❸     {% bootstrap_form form %}

❹     {% buttons %}
        <button name="submit" class="btn btn-primary">add topic</button>
      {% endbuttons %}

    </form>

  {% endblock content %}
```

Most of the changes here are similar to those applied in *login.html*: we load bootstrap3 and add the `header` block with an appropriate message at ❶. Then we add the `class="form"` attribute to the `<form>` tag ❷, use the `{% bootstrap_form %}` template tag instead of `{{ form.as_p }}` ❸, and use the bootstrap3 structure for the submit button ❹. Log in and navigate to the `new_topic` page; it should look similar to the login page now.

### Styling the Topics Page

Now 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 header %}
    <h1>Topics</h1>
  {% endblock header %}

  {% block content %}

    <ul>
      {% for topic in topics %}
        <li>
```

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

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

     {% endblock content %}
```

We don't need the `{% load bootstrap3 %}` tag, because we're not using any custom bootstrap3 template tags in this file. We add the heading *Topics* inside the `header` block 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 panels to make each entry stand out. A *panel* is a div with predefined styling and is perfect for displaying a topic's entries:

*topic.html*

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

❶ {% block header %}
    <h2>{{ topic }}</h2>
  {% endblock 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="panel panel-default">
❸       <div class="panel-heading">
❹         <h3>
            {{ entry.date_added|date:'M d, Y H:i' }}
❺           <small>
              <a href="{% url 'learning_logs:edit_entry' entry.id %}">
                edit entry</a>
            </small>
          </h3>
        </div>
❻       <div class="panel-body">
          {{ entry.text|linebreaks }}
        </div>
      </div> <!-- panel -->
```

```
    {% empty %}
      There are no entries for this topic yet.
    {% endfor %}

{% endblock content %}
```

We first place the topic in the `header` block ❶. We then delete the unordered list structure previously used in this template. Instead of making each entry a list item, we create a `panel` div element at ❷, which contains two more nested divs: a `panel-heading` div ❸ and a `panel-body` div ❻. The `panel-heading` div contains the date for the entry and the link to edit the entry. Both are styled as `<h3>` elements ❹, but we add `<small>` tags around the `edit_entry` link to make it a little smaller than the timestamp ❺.

At ❻ is the `panel-body` div, which contains the actual text of the entry. Notice that the Django code for including the information on the page hasn't changed at all; only the elements that affect the appearance of the page have changed.

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

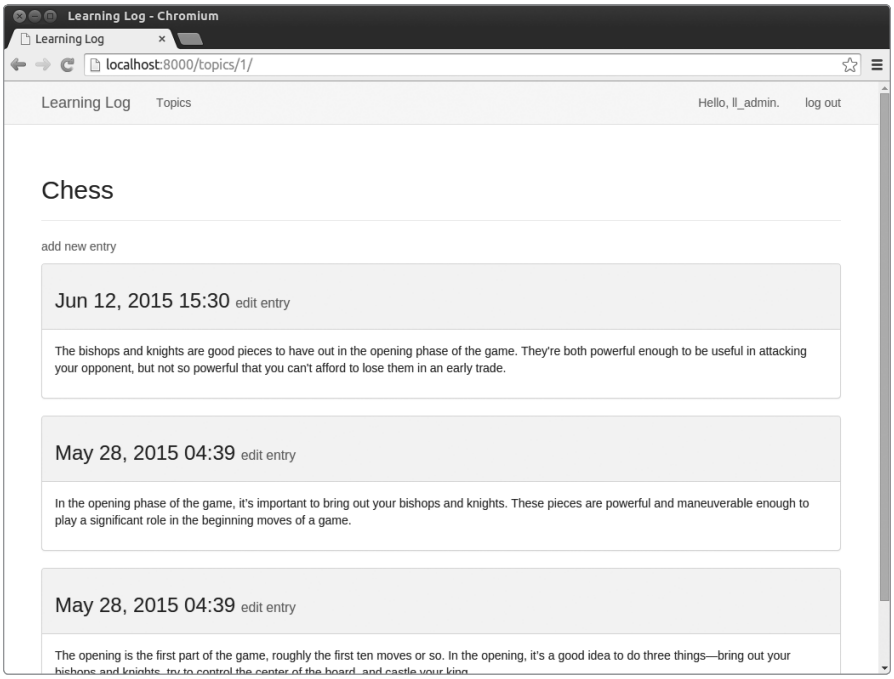



*Figure 20-3: The topic page with Bootstrap styling*

**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 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've applied Bootstrap's styles to the `login` and `add_topic` pages. Make similar changes to the rest of the form-based pages: `new_entry` and `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.

The process is slightly different on Windows than it is on Linux and OS X. If you're using Windows, check for notes in each section that specify what you'll need to do differently on your system.

### 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.

**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 Toolbelt

To deploy and manage a project on Heroku's servers, you'll need the tools available in the Heroku Toolbelt. To install the latest version of the Heroku Toolbelt, visit *https://toolbelt.heroku.com/* and follow the directions for your operating system, which 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 a number of 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 dj-database-url
(ll_env)learning_log$ pip install dj-static
(ll_env)learning_log$ pip install static3
(ll_env)learning_log$ pip install gunicorn
```

Make sure you issue the commands one at a time so you know if any package fails to install correctly. The package `dj-database-url` helps Django communicate with the database Heroku uses, `dj-static` and `static3` help Django manage static files correctly, and `gunicorn` is a server capable of serving apps in a live environment. (*Static files* contain style rules and JavaScript files.)

**NOTE** *Some of the required packages may not install on Windows, so don't be concerned if you get an error message when you try to install some of them. What matters is getting Heroku to install the packages on the live deployment, and we'll do that in the next section.*

### Creating a Packages List with 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 *requirements.txt* to see the packages and version numbers installed in your project (Windows users might not see all of these lines):

*requirements.txt*

```
Django==1.8.4
dj-database-url==0.3.0
dj-static==0.0.6
django-bootstrap3==6.2.2
gunicorn==19.3.0
static3==0.6.1
```

Learning Log already depends on six different packages with specific version numbers, so it requires a specific environment to run properly. 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.

Next, we need to add `psycopg2`, which helps Heroku manage the live database, to the list of packages. Open *requirements.txt* and add the line `psycopg2>=2.6.1`. This will install version 2.6.1 of `psycopg2`, or a newer version if it's available:

*requirements.txt*

```
Django==1.8.4
dj-database-url==0.3.0
dj-static==0.0.6
django-bootstrap3==6.2.2
gunicorn==19.3.0
```

```
static3==0.6.1
psycopg2>=2.6.1
```

If any of the packages didn't install on your system, add those as well. When you're finished, your *requirements.txt* file should include each of the packages shown above. 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.

**NOTE** *If you're using Windows, make sure your version of* requirements.txt *matches the list shown here regardless of which packages you were able to install on your system.*

### Specifying the Python Runtime

Unless you specify a Python version, Heroku will use its own 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 3.5.0
```

In this example I'm running Python 3.5.0. Make a new file called *runtime.txt* in the same directory as *manage.py*, and enter the following:

*runtime.txt*

```
python-3.5.0
```

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 is not available, go to* https://devcenter.heroku.com/ *and click* ***Python****; then 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 settings specifically for the Heroku environment:

*settings.py*

```
--snip--
# Settings for django-bootstrap3
BOOTSTRAP3 = {
    'include_jquery': True,
    }
```

```
# Heroku settings
❶ if os.getcwd() == '/app':
❷     import dj_database_url
      DATABASES = {
          'default': dj_database_url.config(default='postgres://localhost')
      }

      # Honor the 'X-Forwarded-Proto' header for request.is_secure().
❸     SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

      # Allow all host headers.
❹     ALLOWED_HOSTS = ['*']

      # Static asset configuration
❺     BASE_DIR = os.path.dirname(os.path.abspath(__file__))
      STATIC_ROOT = 'staticfiles'
      STATICFILES_DIRS = (
          os.path.join(BASE_DIR, 'static'),
      )
```

At ❶ we use the function `getcwd()`, which gets the *current working directory* the file is running from. In a Heroku deployment, the directory is always */app*. In a local deployment, the directory is usually the name of the project folder (*learning_log* in our case). The `if` test ensures that the settings in this block apply only when the project is deployed on Heroku. This structure allows us to have one settings file that works for our local development environment as well as the live server.

At ❷ we import `dj_database_url` to help configure the database on Heroku. Heroku uses PostgreSQL (also called Postgres), a more advanced database than SQLite, and these settings configure the project to use Postgres on Heroku. The rest of the settings support HTTPS requests ❸, ensure that Django will serve the project from Heroku's URL ❹, and set up the project to serve static files correctly on Heroku ❺.

### Making a Procfile to Start Processes

A *Procfile* tells Heroku which processes to start in order to serve the project properly. This is a one-line file that you should save as *Procfile*, with an uppercase *P* and no file extension, in the same directory as *manage.py*.

Here's what 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.

### Modifying wsgi.py for Heroku

We also need to modify *wsgi.py* for Heroku, because Heroku needs a slightly different setup than what we've been using:

*wsgi.py*

```
--snip--
import os

from django.core.wsgi import get_wsgi_application
from dj_static import Cling

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "learning_log.settings")
application = Cling(get_wsgi_application())
```

We import Cling, which helps serve static files correctly, and use it to launch the application. This code will work locally as well, so we don't need to put it in an `if` block.

### Making a Directory for Static Files

On Heroku, Django collects all the static files and places them in one place so it can manage them efficiently. We'll create a directory for these static files. Inside the *learning_log* folder we've been working from is another folder called *learning_log*. In this nested folder, make a new folder called *static* with the path *learning_log/learning_log/static/*. We also need to make a placeholder file to store in this directory for now, because empty directories won't be included in the project when it's pushed to Heroku. In the *static/* directory, make a file called *placeholder.txt*:

*placeholder.txt*

```
This file ensures that learning_log/static/ will be added to the project.
Django will collect static files and place them in learning_log/static/.
```

There's nothing special about this text; it just reminds us why we included this file in the project.

### Using the gunicorn Server Locally

If you're using Linux or OS X, you can try using the gunicorn server locally before deploying to Heroku. From an active virtual environment, run the command `heroku local` to start the processes defined in *Procfile*:

```
(ll_env)learning_log$ heroku local
Installing Heroku Toolbelt v4... done
--snip--
forego | starting web.1 on port 5000
❶ web.1  | [2015-08-13 22:00:45 -0800] [12875] [INFO] Starting gunicorn 19.3.0
❷ web.1  | [2015-08-13 22:00:45 -0800] [12875] [INFO] Listening at:
    http://0.0.0.0:5000 (12875)
❸ web.1  | [2015-08-13 22:00:45 -0800] [12878] [INFO] Booting worker with pid: 12878
```

The first time you run `heroku local`, a number of packages from the Heroku Toolbelt will be installed. The output shows that gunicorn has been started with a process id of 12875 in this example ❶. At ❷ gunicorn is listening for requests on port 5000. In addition, gunicorn has started a *worker* process (12878) to help it serve requests ❸.

Visit *http://localhost:5000/* to make sure everything is working; you should see the Learning Log home page, just as it appears when you use the Django server (`runserver`). Press CTRL-C to stop the processes started by `heroku local`. You should continue to use `runserver` for local development.

**NOTE** *gunicorn won't run on Windows, so skip this step if you're using Windows. This won't affect your ability to deploy the project to Heroku.*

## Using Git to Track the Project's Files

If you completed Chapter 17, you'll know that 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. This allows you to easily return to the last working snapshot of your project if anything goes wrong; for example, if you accidentally introduce a bug while working on a new feature. Each of these snapshots is called a *commit*.

Using Git means 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. If you want to read more about Git and version control, see Appendix D.

### Installing Git

The Heroku Toolbelt includes Git, so it should already be installed on your system. But terminal windows that were open before you installed the Heroku Toolbelt won't have access to Git, so open a new terminal window and issue the command `git --version`:

```
(ll_env)learning_log$ git --version
git version 2.5.0
```

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

### Configuring Git

Git keeps track of who makes changes to a project, even in cases like this when there's only one person working on the project. To do this, Git needs to know your username and email. You have to provide a username, but feel free to 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 Git to ignore some files. Make 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 what goes in *.gitignore*:

*.gitignore*

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

We tell Git to ignore the entire directory *ll_env*, 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.

**NOTE** *If you're using Python 2.7, replace* `__pycache__` *with* `*.pyc` *because Python 2.7 doesn't create a* `__pycache__` *directory.*

### 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 we 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) dbc1d99] Ready for deployment to heroku.
   43 files changed, 746 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 directory 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 directory 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 terminal session, issue the following commands:

```
❶ (ll_env)learning_log$ heroku login
  Enter your Heroku credentials.
  Email: eric@example.com
  Password (typing will be hidden):
  Logged in as eric@example.com
❷ (ll_env)learning_log$ heroku create
  Creating afternoon-meadow-2775... done, stack is cedar-14
  https://afternoon-meadow-2775.herokuapp.com/ |
      https://git.heroku.com/afternoon-meadow-2775.git
  Git remote heroku added
❸ (ll_env)learning_log$ git push heroku master
  --snip--
  remote: -----> Launching... done, v6
❹ remote:        https://afternoon-meadow-2775.herokuapp.com/ deployed to Heroku
  remote: Verifying deploy.... done.
  To https://git.heroku.com/afternoon-meadow-2775.git
     bdb2a35..62d711d  master -> master
  (ll_env)learning_log$
```

First, log in to Heroku in the terminal session with the username and password you used to create an account at *https://heroku.com/* ❶. Then 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. We then 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. Heroku then builds the project on its servers using these files. At ❹ is the URL we'll use to access the live project.

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 quota left: 17h 40m
❷ === web (Free): `gunicorn learning_log.wsgi __log-file -`
  web.1: up 2015/08/14 07:08:51 (~ 10m ago)
  (ll_env)learning_log$
```

The output shows how much more time the project can be active in the next 24 hours ❶. At the time of this writing, Heroku allows free deployments to be active for up to 18 hours in any 24-hour period. If a project

exceeds these limits, a standard server error page will be displayed; 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
Opening afternoon-meadow-2775... done
```

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 issues you can't resolve, look at Heroku's documentation for help. Go to* https://devcenter.heroku.com/*, click* ***Python****, and look for a link to* Getting Started with Django. *If you can'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 afternoon-meadow-2775... up, run.2435
    --snip--
❸ Running migrations:
    --snip--
    Applying learning_logs.0001_initial... OK
    Applying learning_logs.0002_entry... OK
    Applying learning_logs.0003_topic_user... 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. However, you won't see any of the data you entered on your local deployment, 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

In this section 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 afternoon-meadow-2775... up, run.6244
❶ ~ $ ls
learning_log learning_logs manage.py Procfile requirements.txt runtime.txt
users
staticfiles
❷ ~ $ python manage.py createsuperuser
Username (leave blank to use 'u41907'): 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 file system 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, use 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://afternoon-meadow-2775.herokuapp.com/admin/*.

If other people 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

You'll probably want your URL to be friendlier and more memorable than *https://afternoon-meadow-2775.herokuapp.com/*. You can rename the app using a single command:

```
(ll_env)learning_log$ heroku apps:rename learning-log
Renaming afternoon-meadow-2775 to learning-log... done
https://learning-log.herokuapp.com/ | https://git.heroku.com/learning-log.git
Git remote heroku updated
(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, but they give way too much information to attackers if you leave them enabled on a live server. We also need to make sure no one can get information or redirect requests by pretending to be the project's host.

Let's modify *settings.py* so we can see error messages locally but not on the live deployment:

*settings.py*

```
--snip--
# Heroku settings
if os.getcwd() == '/app':
    --snip--
    # Honor the 'X-Forwarded-Proto' header for request.is_secure().
    SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

    # Allow only Heroku to host the project.
❶   ALLOWED_HOSTS = ['learning-log.herokuapp.com']

❷   DEBUG = False

    # Static asset configuration
    --snip--
```

We need to make only two changes: at ❶ we modify `ALLOWED_HOSTS`, so the only server allowed to host the project is Heroku. You need to use the name of your app, whether it's the name Heroku provided, such as *afternoon-meadow-2775.herokuapp.com*, or the name you chose. At ❷ we set `DEBUG` to `False`, so Django won't share sensitive information when an error occurs.

### 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=False for Heroku."
[master 081f635] Set DEBUG=False for Heroku.
 1 file changed, 4 insertions(+), 2 deletions(-)
❷ (ll_env)learning_log$ git status
# On branch master
nothing to commit, working directory 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
--snip--
remote: -----> Python app detected
remote: -----> Installing dependencies with pip
--snip--
remote: -----> Launching... done, v8
remote:        https://learning-log.herokuapp.com/ deployed to Heroku
remote: Verifying deploy.... done.
To https://git.heroku.com/learning-log.git
   4c9d111..ef65d2b  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.

To check that the deployment is more secure now, enter the URL of your project with an extension 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, but users won't see critical information about the project's code.

## 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 the overall appearance of Learning Log. These templates must go in the root template directory.

### Making Custom Templates

In the *learning_log/learning_log* folder, make a new folder called *templates*. Then make a new file called *404.html* using the following code:

*404.html*

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

{% block header %}
    <h2>The item you requested is not available. (404)</h2>
{% endblock 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 header %}
    <h2>There has been an internal error. (500)</h2>
{% endblock 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, 'learning_log/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 changes 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

ALLOWED_HOSTS = ['localhost']
--snip--
```

You must have at least one host specified in `ALLOWED_HOSTS` when `DEBUG` is set to `False`. Now request a topic or entry that doesn't belong to you to see the 404 error page, and request a URL that doesn't exist (such as *localhost:8000/letmein/*) to see the 500 error page.

When you're finished checking the error pages, set `DEBUG` back to `True` to further develop Learning Log. (Make sure `DEBUG` is still set to `False` in the section of *settings.py* that applies to the Heroku deployment.)

**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 template changes 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 learning_log/templates/404.html
 create mode 100644 learning_log/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
   2b34ca1..a64d8d3  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 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 with 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*

```
--snip--
from django.shortcuts import render, get_object_or_404
from django.http import HttpResponseRedirect, Http404
--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/999999/*), 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.

Then commit the changes to your repository using `git commit -am "commit message"`. Thereafter, 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 might also limit 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/*), and you'll be redirected to a page showing a list of your projects. Click the project you want to delete, and 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 `afternoon-meadow-2775` 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.*