

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 psycpg2==2.7.*  
(ll_env)learning_log$ pip install django-heroku  
(ll_env)learning_log$ pip install gunicorn
```

The psycpg2 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
❶ 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.