Django Tutorial Part 11: Deploying Django to production

Now you've created (and tested) an awesome LocalLibrary website, you're going to want to install it on a public web server so that it can be accessed by library staff and members over the Internet. This article provides an overview of how you might go about finding a host to deploy your website, and what you need to do in order to get your site ready for production.

Prerequisites:	Complete all previous tutorial topics, including Django Tutorial Part 10: Testing a Django web application.
Objective:	To learn where and how you can deploy a Django app to production.

Overview

Once your site is finished (or finished "enough" to start public testing) you're going to need to host it somewhere more public and accessible than your personal development computer.

Up to now you've been working in a development environment, using the Django development web server to share your site to the local browser/network, and running your website with (insecure) development settings that expose debug and other private information. Before you can host a website externally you're first going to have to:

- Make a few changes to your project settings.
- Choose an environment for hosting the Django app.
- Choose an environment for hosting any static files.
- Set up a production-level infrastructure for serving your website.

This tutorial provides some guidance on your options for choosing a hosting site, a brief overview of what you need to do in order to get your Django app ready for production, and a worked example of how to install the LocalLibrary website onto the Heroku cloud hosting service.

What is a production environment?

The production environment is the environment provided by the server computer where you will run your website for external consumption. The environment includes:

- Computer hardware on which the website runs.
- Operating system (e.g. Linux, Windows).
- Programming language runtime and framework libraries on top of which your website is written.
- Web server used to serve pages and other content (e.g. Nginx, Apache).
- Application server that passes "dynamic" requests between your Django website and the webserver.
- Databases on which your website is dependent.

> **Note**
>
> Depending on how your production is configured you might also have a reverse proxy, load balancer, etc.

The server computer could be located on your premises and connected to the Internet by a fast link, but it is far more common to use a computer that is hosted "in the cloud". What this actually means is that your code is run on some remote computer (or possibly a "virtual" computer) in your hosting company's data center(s). The remote server will usually offer some guaranteed level of computing resources (e.g. CPU, RAM, storage memory, etc.) and Internet connectivity for a certain price.

This sort of remotely accessible computing/networking hardware is referred to as *Infrastructure as a Service (IaaS)*. Many IaaS vendors provide options to preinstall a particular operating system, onto which you must install the other components of your production environment. Other vendors allow you to select more fully-featured environments, perhaps including a complete Django and web-server setup.

> **Note**
>
> Pre-built environments can make setting up your website very easy because they reduce the configuration, but the available options may limit you to an unfamiliar server (or other components) and may be based on an older version of the OS. Often it is better to install components yourself, so that you get the ones that you want, and when you need to upgrade parts of the system, you have some idea of where to start!

Other hosting providers support Django as part of a *Platform as a Service* (PaaS) offering. In this sort of hosting you don't need to worry about most of your production environment (web server, application server, load balancers) as the host platform takes care of those for you (along with most of what you need to do in order to scale your application). That makes deployment quite easy, because you just need to concentrate on your web application and not all the other server infrastructure.

Some developers will choose the increased flexibility provided by IaaS over PaaS, while others will appreciate the reduced maintenance overhead and easier scaling of PaaS. When you're getting started, setting up your website on a PaaS system is much easier, and so that is what we'll do in this tutorial.

> **Note**
>
> If you choose a Python/Django-friendly hosting provider they should provide instructions on how to set up a Django website using different configurations of webserver, application server, reverse proxy, etc (this won't be relevant if you choose a PaaS). For example, there are many step-by-step guides for various configurations in the Digital Ocean Django community docs.

Choosing a hosting provider

There are well over 100 hosting providers that are known to either actively support or work well with Django (you can find a fairly exhaustive list at DjangoFriendly hosts). These vendors provide different types of environments (IaaS, PaaS), and different levels of computing and network resources at different prices.

Some of the things to consider when choosing a host:

- How busy your site is likely to be and the cost of data and computing resources required to meet that demand.
- Level of support for scaling horizontally (adding more machines) and vertically (upgrading to more powerful machines) and the costs of doing so.
- Where the supplier has data centres, and hence where access is likely to be fastest.
- The host's historical uptime and downtime performance.
- Tools provided for managing the site — are they easy to use and are they secure (e.g. SFTP vs FTP).

- Inbuilt frameworks for monitoring your server.
- Known limitations. Some hosts will deliberately block certain services (e.g. email). Others offer only a certain number of hours of "live time" in some price tiers, or only offer a small amount of storage.
- Additional benefits. Some providers will offer free domain names and support for SSL certificates that you would otherwise have to pay for.
- Whether the "free" tier you're relying on expires over time, and whether the cost of migrating to a more expensive tier means you would have been better off using some other service in the first place!

The good news when you're starting out is that there are quite a few sites that provide "evaluation", "developer", or "hobbyist" computing environments for "free". These are always fairly resource constrained/limited environments, and you do need to be aware that they may expire after some introductory period. They are however great for testing low traffic sites in a real environment, and can provide an easy migration to paying for more resources when your site gets busier. Popular choices in this category include Heroku, Python Anywhere, Amazon Web Services, Microsoft Azure, etc.

Many providers also have a "basic" tier that provides more useful levels of computing power and fewer limitations. Digital Ocean and Python Anywhere are examples of popular hosting providers that offer a relatively inexpensive basic computing tier (in the $5 to $10USD per month range).

> **Note**
>
> Remember that price is not the only selection criteria. If your website is successful, it may turn out that scalability is the most important consideration.

Getting your website ready to publish

The Django skeleton website created using the *django-admin* and *manage.py* tools are configured to make development easier. Many of the Django project settings (specified in **settings.py**) should be different for production, either for security or performance reasons.

> **Tip**
>
> It is common to have a separate **settings.py** file for production, and to import sensitive settings from a separate file or an environment variable. This file should then be protected, even if the rest of the source code is available on a public repository

even if the rest of the source code is available on a public repository.

The critical settings that you must check are:

- `DEBUG`. This should be set as `False` in production (`DEBUG = False`). This stops the sensitive/confidential debug trace and variable information from being displayed.
- `SECRET_KEY`. This is a large random value used for CSRF protection etc. It is important that the key used in production is not in source control or accessible outside the production server. The Django documents suggest that this might best be loaded from an environment variable or read from a server-only file.

```
# Read SECRET_KEY from an environment variable
import os
SECRET_KEY = os.environ['SECRET_KEY']

# OR

# Read secret key from a file
with open('/etc/secret_key.txt') as f:
    SECRET_KEY = f.read().strip()
```

Let's change the *LocalLibrary* application so that we read our `SECRET_KEY` and `DEBUG` variables from environment variables if they are defined, but otherwise use the default values in the configuration file.

Open **/locallibrary/settings.py**, disable the original `SECRET_KEY` configuration and add the new lines as shown below in **bold**. During development no environment variable will be specified for the key, so the default value will be used (it shouldn't matter what key you use here, or if the key "leaks", because you won't use it in production).

```
# SECURITY WARNING: keep the secret key used in production secret!
# SECRET_KEY = "cg#p$g+j9tax!#a3cup@1$8obt2_+&k3q+pmu)5%asj6yjpkag"
import os
SECRET_KEY = os.environ.get('DJANGO_SECRET_KEY', 'cg#p$g+j9tax!#a3cup(
```

Then comment out the existing `DEBUG` setting and add the new line shown below.

```
# SECURITY WARNING: don't run with debug turned on in production!
# DEBUG = True
DEBUG = os.environ.get('DJANGO_DEBUG', '') != 'False'
```

The value of the `DEBUG` will be `True` by default, but will only be `False` if the value of the `DJANGO_DEBUG` environment variable is set to `False`. Please note that environment variables are strings and not Python types. We therefore need to compare strings. The only way to set the `DEBUG` variable to `False` is to actually set it to the string `False`

You can set the environment variable to False by issuing the following command:

```
export DJANGO_DEBUG=False
```

A full checklist of settings you might want to change is provided in Deployment checklist (Django docs). You can also list a number of these using the terminal command below:

```
python3 manage.py check --deploy
```

Example: Installing LocalLibrary on Heroku

This section provides a practical demonstration of how to install *LocalLibrary* on the Heroku PaaS cloud.

Why Heroku?

Heroku is one of the longest running and popular cloud-based PaaS services. It originally supported only Ruby apps, but now can be used to host apps from many programming environments, including Django!

We are choosing to use Heroku for several reasons:

- Heroku has a free tier that is *really* free (albeit with some limitations).
- As a PaaS, Heroku takes care of a lot of the web infrastructure for us. This makes it much easier to get started, because you don't worry about servers, load balancers, reverse proxies, or any of the other web infrastructure that Heroku provides for us under the hood.
- While it does have some limitations these will not affect this particular application. For example:
  - Heroku provides only short-lived storage so user-uploaded files cannot safely be stored on Heroku itself.
  - The free tier will sleep an inactive web app if there are no requests within a half hour period. The site may then take several seconds to respond when it is woken up.
  - The free tier limits the time that your site is running to a certain amount of hours

every month (not including the time that the site is "asleep"). This is fine for a low use/demonstration site, but will not be suitable if 100% uptime is required.
    - Other limitations are listed in Limits (Heroku docs).
- Mostly it just works, and if you end up loving it, scaling your app is very easy.

While Heroku is perfect for hosting this demonstration it may not be perfect for your real website. Heroku makes things easy to set up and scale, at the cost of being less flexible, and potentially a lot more expensive once you get out of the free tier.

How does Heroku work?

Heroku runs Django websites within one or more "Dynos", which are isolated, virtualized Unix containers that provide the environment required to run an application. The dynos are completely isolated and have an *ephemeral* file system (a short-lived file system that is cleaned/emptied every time the dyno restarts). The only thing that dynos share by default are application configuration variables. Heroku internally uses a load balancer to distribute web traffic to all "web" dynos. Since nothing is shared between them, Heroku can scale an app horizontally by adding more dynos (though of course you may also need to scale your database to accept additional connections).

Because the file system is ephemeral you can't install services required by your application directly (e.g. databases, queues, caching systems, storage, email services, etc). Instead Heroku web applications use backing services provided as independent "add-ons" by Heroku or 3rd parties. Once attached to your web application, the dynos access the services using information contained in application configuration variables.

In order to execute your application Heroku needs to be able to set up the appropriate environment and dependencies, and also understand how it is launched. For Django apps we provide this information in a number of text files:

- **runtime.txt**: the programming language and version to use.
- **requirements.txt**: the Python component dependencies, including Django.
- **Procfile**: A list of processes to be executed to start the web application. For Django this will usually be the Gunicorn web application server (with a `.wsgi` script).
- **wsgi.py**: WSGI configuration to call our Django application in the Heroku environment.

Developers interact with Heroku using a special client app/terminal, which is much like a Unix Bash shell. This allows you to upload code that is stored in a git repository, inspect the running processes, see logs, set configuration variables and much more!

In order to get our application to work on Heroku we'll need to put our Django web application into a git repository, add the files above, integrate with a database add-on, and make changes to properly handle static files.

Once we've done all that we can set up a Heroku account, get the Heroku client, and use it to install our website.

> **Note**
>
> The instructions below reflect how to work with Heroku at time of writing. If Heroku significantly change their processes, you may wish to instead check their setup documents: Getting Started on Heroku with Django.

That's all the overview you need in order to get started (see How Heroku works for a more comprehensive guide).

Creating an application repository in Github

Heroku is closely integrated with the **git** source code version control system, using it to upload/synchronise any changes you make to the live system. It does this by adding a new heroku "remote" repository named *heroku* pointing to a repository for your source on the Heroku cloud. During development you use git to store changes on your own repository. When you want to deploy your site, you sync your changes to the Heroku repository.

> **Note**
>
> If you're used to following good software development practices you are probably already using git or some other SCM system. If you already have a git repository, then you can skip this step.

There are a lot of ways to work with git, but one of the easiest is to first set up an account on Github, create the repository there, and then sync to it locally:

1. Visit https://github.com/ and create an account.
2. Once you are logged in, click the **+** link in the top toolbar and select **New repository**.
3. Fill in all the fields on this form. While these are not compulsory, they are strongly recommended.
    - Enter a new repository name (e.g. *django_local_library*), and description (e.g.

"Local Library website written in Django".
- Choose **Python** in the *Add .gitignore* selection list.
- Choose your preferred license in the *Add license* selection list.
- Check **Initialize this repository with a README**.

4. Press **Create repository**.
5. Click the green "**Clone or download**" button on your new repo page.
6. Copy the URL value from the text field inside the dialog box that appears (it should be something like: **https://github.com/<*your_git_user_id*>/django_local_library.git**).

Now that the repository ("repo") is created we are going to want to clone it on our local computer:

1. Install *git* for your local computer (you can find versions for different platforms here).
2. Open a command prompt/terminal and clone your repository using the URL you copied above:
```
git clone https://github.com/<your_git_user_id>/django_local_libr
```
This will create the repository in a new folder in the current working directory.
3. Navigate into the new repo.
```
cd django_local_library
```

The final steps are to copy your application into this local project directory and then add (or "push", in git lingo) the local repository to your remote Github repository:

1. Copy your Django application into this folder (all the files at the same level as **manage.py** and below, **not** their containing locallibrary folder).
2. Open the **.gitignore** file, copy the following lines into the bottom of it, and then save (this file is used to identify files that should not be uploaded to git by default).
```
# Text backup files
*.bak

# Database
*.sqlite3
```
3. Open a command prompt/terminal and use the `add` command to add all files to git. This adds the files which aren't ignored by the **.gitignore** file to the "staging area".
```
git add -A
```
4. Use the `status` command to check that all files you are about to `commit` are correct (you want to include source files, not binaries, temporary files etc.). It should look a bit

(you want to include source files, not binaries, temporary files etc). It should look a bit like the listing below.

```
> git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   .gitignore
        new file:   catalog/__init__.py
        ...
        new file:   catalog/migrations/0001_initial.py
        ...
        new file:   templates/registration/password_reset_form.html
```

5. When you're satisfied, `commit` the files to your local repository. This is essentially equivalent to signing off on the changes and making them an official part of the local repository.

   ```
   git commit -m "First version of application moved into github"
   ```

6. At this point, the remote repository has not been changed. Synchronise (`push`) your local repository to the remote Github repository using the following command:

   ```
   git push origin main
   ```

> **Warning**
>
> In 2020 Github change the default repo branch name to "main" (from "master"). If using an older/existing repository you might call `git push origin master` instead.

When this operation completes, you should be able to go back to the page on Github where

When this operation completes, you should be able to go back to the page on Github where you created your repo, refresh the page, and see that your whole application has now been uploaded. You can continue to update your repository as files change using this add/commit/push cycle.

> **Tip**
>
> This is a good point to make a backup of your "vanilla" project — while some of the changes we're going to be making in the following sections might be useful for deployment on any platform (or development) others might not.
>
> The *best* way to do this is to use *git* to manage your revisions. With *git* you can not only go back to a particular old version, but you can maintain this in a separate "branch" from your

back to a particular old version, but you can maintain this in a separate "branch" from your production changes and cherry-pick any changes to move between production and development branches. Learning Git is well worth the effort, but is beyond the scope of this topic.

The *easiest* way to do this is to just copy your files into another location. Use whichever approach best matches your knowledge of git!

Update the app for Heroku

This section explains the changes you'll need to make to our *LocalLibrary* application to get it to work on Heroku. While Heroku's Getting Started on Heroku with Django instructions assume you will use the Heroku client to also run your local development environment, our changes are compatible with the existing Django development server and the workflows we've already learned.

Procfile

Create the file `Procfile` (no extension) in the root of your GitHub repository to declare the application's process types and entry points. Copy the following text into it:

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

The "`web:`" tells Heroku that this is a web dyno and can be sent HTTP traffic. The process to start in this dyno is *gunicorn*, which is a popular web application server that Heroku recommends. We start Gunicorn using the configuration information in the module `locallibrary.wsgi` (created with our application skeleton: **/locallibrary/wsgi.py**).

Gunicorn

Gunicorn is the recommended HTTP server for use with Django on Heroku (as referenced in the Procfile above). It is a pure-Python HTTP server for WSGI applications that can run multiple Python concurrent processes within a single dyno (see Deploying Python applications with Gunicorn for more information).

While we won't need *Gunicorn* to serve our LocalLibrary application during development, we'll install it so that it becomes part of our requirements for Heroku to set up on the remote server.

Install *Gunicorn* locally on the command line using *pip* (which we installed when setting up the development environment):

> **Note**
>
> Make sure that you're in your Python virtual environment (use the `workon [name-of-virtual-environment]` command) before you install *Gunicorn* and further modules with *pip*, or you might experience problems with importing these modules in your **/locallibrary/settings.py** file in the later sections.

```
pip3 install gunicorn
```

Database configuration

We can't use the default SQLite database on Heroku because it is file-based, and it would be deleted from the *ephemeral* file system every time the application restarts (typically once a day, and every time the application or its configuration variables are changed).

The Heroku mechanism for handling this situation is to use a database add-on and configure the web application using information from an environment configuration variable, set by the add-on. There are quite a lot of database options, but we'll use the hobby tier of the *Heroku postgres* database as this is free, supported by Django, and automatically added to our new Heroku apps when using the free hobby dyno plan tier.

The database connection information is supplied to the web dyno using a configuration variable named `DATABASE_URL`. Rather than hard-coding this information into Django, Heroku recommends that developers use the dj-database-url package to parse the `DATABASE_URL` environment variable and automatically convert it to Django's desired configuration format. In addition to installing the *dj-database-url* package we'll also need to install psycopg2, as Django needs this to interact with Postgres databases.

dj-database-url (Django database configuration from environment variable)

Install *dj-database-url* locally so that it becomes part of our requirements for Heroku to set up on the remote server:

```
$ pip3 install dj-database-url
```

settings.py

Open **/locallibrary/settings.py** and copy the following configuration into the bottom of the file:

```
# Heroku: Update database configuration from $DATABASE_URL.
import dj_database_url
db_from_env = dj_database_url.config(conn_max_age=500)
```

```
db_from_env = dj_database_url.config(conn_max_age=500)
DATABASES['default'].update(db_from_env)
```

> **Note**
>
> - We'll still be using SQLite during development because the `DATABASE_URL` environment variable will not be set on our development computer.
> - The value `conn_max_age=500` makes the connection persistent, which is far more efficient than recreating the connection on every request cycle. However, this is optional and can be removed if needed.

psycopg2 (Python Postgres database support)

Django needs *psycopg2* to work with Postgres databases and you will need to add this to the requirements.txt for Heroku to set this up on the remote server (as discussed in the requirements section below).

Django will use our SQLite database locally by default, because the `DATABASE_URL` environment variable isn't set in our local environment. If you want to switch to Postgres completely and use our Heroku free tier database for both development and production then you can. For example, to install psycopg2 and its dependencies locally on a Debian-flavoured Linux system you would use the following Bash/terminal commands:

```
sudo apt-get install python-pip python-dev libpq-dev postgresql postgr
pip3 install psycopg2-binary
```

Installation instructions for the other platforms can be found on the psycopg2 website here.

However, you don't need to do this — you don't need PostgreSQL active on the local computer, as long as you give it to Heroku as a requirement, in `requirements.txt` (see below).

Serving static files in production

During development we used Django and the Django development web server to serve our static files (CSS, JavaScript, etc.). In a production environment we instead typically serve static files from a content delivery network (CDN) or the web server.

> **Note**

> Serving static files via Django/web application is inefficient because the requests have to pass through unnecessary additional code (Django) rather than being handled directly by the web server or a completely separate CDN. While this doesn't matter for local use
>
> during development, it would have a significant performance impact if we were to use the same approach in production.

To make it easy to host static files separately from the Django web application, Django provides the *collectstatic* tool to collect these files for deployment (there is a settings variable that defines where the files should be collected when *collectstatic* is run). Django templates refer to the hosting location of the static files relative to a settings variable (`STATIC_URL`), so that this can be changed if the static files are moved to another host/server.

The relevant setting variables are:

- `STATIC_URL`: This is the base URL location from which static files will be served, for example on a CDN. This is used for the static template variable that is accessed in our base template (see Django Tutorial Part 5: Creating our home page).
- `STATIC_ROOT`: This is the absolute path to a directory where Django's *collectstatic* tool will gather any static files referenced in our templates. Once collected, these can then be uploaded as a group to wherever the files are to be hosted.
- `STATICFILES_DIRS`: This lists additional directories that Django's *collectstatic* tool should search for static files.

settings.py

Open **/locallibrary/settings.py** and copy the following configuration into the bottom of the file. The `BASE_DIR` should already have been defined in your file (the `STATIC_URL` may already have been defined within the file when it was created. While it will cause no harm, you might as well delete the duplicate previous reference).

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.1/howto/static-files/

# The absolute path to the directory where collectstatic will collect static files for
STATIC_ROOT = BASE_DIR / 'staticfiles'  #. os.path.join(BASE_DIR, 'staticfiles')

# The URL to use when referring to static files (where they will be served from)
STATIC_URL = '/static/'
```

We'll actually do the file serving using a library called WhiteNoise, which we install and

configure in the next section.

For more information, see Django and Static Assets (Heroku docs).

Whitenoise

There are many ways to serve static files in production (we saw the relevant Django settings in the previous sections). Heroku recommends using the WhiteNoise project for serving of static assets directly from Gunicorn in production.

> **Note**
>
> Heroku automatically calls *collectstatic* and prepares your static files for use by WhiteNoise after it uploads your application. Check out WhiteNoise documentation for an explanation of how it works and why the implementation is a relatively efficient method for serving these files.

The steps to set up *WhiteNoise* to use with the project are given here (and reproduced below):

WhiteNoise

Install whitenoise locally using the following command:

```
$ pip3 install whitenoise
```

settings.py

To install *WhiteNoise* into your Django application, open **/locallibrary/settings.py**, find the `MIDDLEWARE` setting and add the `WhiteNoiseMiddleware` near the top of the list, just below the `SecurityMiddleware`:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Optionally, you can reduce the size of the static files when they are served (this is more

efficient). Just add the following to the bottom of **/locallibrary/settings.py**:

```
# Simplified static file serving.
# https://warehouse.python.org/project/whitenoise/
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

Requirements

The Python requirements of your web application must be stored in a file **requirements.txt** in the root of your repository. Heroku will then install these automatically when it rebuilds your environment. You can create this file using *pip* on the command line (run the following in the repo root):

```
pip3 freeze > requirements.txt
```

After installing all the different dependencies above, your **requirements.txt** file should have *at least* these items listed (though the version numbers may be different). Please delete any other dependencies not listed below, unless you've explicitly added them for this application.

```
dj-database-url==0.5.0
Django==3.1.2
gunicorn==20.0.4
psycopg2-binary==2.8.6
whitenoise==5.2.0
```

> **Note**
>
> Make sure that a **psycopg2** line like the one above is present! Even if you didn't install this locally then you should still add it to **requirements.txt**.

Runtime

The **runtime.txt** file, if defined, tells Heroku which programming language to use. Create the file in the root of the repo and add the following text:

```
python-3.8.6
```

> **Note**
>
> Heroku only supports a small number of Python runtimes (at time of writing, this includes the one above). Heroku will use a supported runtime irrespective of the value specified in this file.

Re-test and save changes to Github

Before we proceed, lets test the site again locally and make sure it wasn't broken by any of our changes above. Run the development web server as usual and then check the site still works as you expect on your browser.

```
python3 manage.py runserver
```

Next, lets `push` our changes to Github. In the terminal (after having navigated to our local repository), enter the following commands:

```
git add -A
git commit -m "Added files and changes required for deployment to hero
git push origin main
```

We should now be ready to start deploying LocalLibrary on Heroku.

Get a Heroku account

To start using Heroku you will first need to create an account:

- Go to www.heroku.com and click the **SIGN UP FOR FREE** button.
- Enter your details and then press **CREATE FREE ACCOUNT**. You'll be asked to check your account for a sign-up email.
- Click the account activation link in the signup email. You'll be taken back to your account on the web browser.
- Enter your password and click **SET PASSWORD AND LOGIN**.
- You'll then be logged in and taken to the Heroku dashboard: https://dashboard.heroku.com/apps.

Install the client

Download and install the Heroku client by following the instructions on Heroku here.

After the client is installed you will be able run commands. For example to get help on the client:

```
heroku help
```

Create and upload the website

To create the app we run the "create" command in the root directory of our repository. This creates a git remote ("pointer to a remote repository") named *heroku* in our local git environment.

```
heroku create
```

> **Note**
>
> You can name the remote if you like by specifying a value after "create". If you don't then you'll get a random name. The name is used in the default URL.

We can then push our app to the Heroku repository as shown below. This will upload the app, package it in a dyno, run *collectstatic*, and start the site.

```
git push heroku main
```

If we're lucky, the app is now "running" on the site, but it won't be working properly because we haven't set up the database tables for use by our application. To do this we need to use the `heroku run` command and start a " one off dyno" to perform a migrate operation. Enter the following command in your terminal:

```
heroku run python manage.py migrate
```

We're also going to need to be able to add books and authors, so lets also create our administration superuser, again using a one-off dyno:

```
heroku run python manage.py createsuperuser
```

Once this is complete, we can look at the site. It should work, although it won't have any books in it yet. To open your browser to the new website, use the command:

```
heroku open
```

Create some books in the admin site, and check out whether the site is behaving as you expect.

Managing addons

You can check out the add-ons to your app using the `heroku addons` command. This will list

all addons, and their price tier and state.

```
> heroku addons


Add-on                                            Plan        Price  State
────────────────────────────────────────────────  ──────────  ─────  ───────
heroku-postgresql (postgresql-flat-26536)  hobby-dev  free   created
 └─ as DATABASE
```

Here we see that we have just one add-on, the postgres SQL database. This is free, and was created automatically when we created the app. You can open a web page to examine the database add-on (or any other add-on) in more detail using the following command:

```
heroku addons:open heroku-postgresql
```

Other commands allow you to create, destroy, upgrade and downgrade addons (using a similar syntax to opening). For more information see Managing Add-ons (Heroku docs).

Setting configuration variables

You can check out the configuration variables for the site using the `heroku config` command. Below you can see that we have just one variable, the `DATABASE_URL` used to configure our database.

```
> heroku config

=== locallibrary Config Vars
DATABASE_URL: postgres://uzfnbcyxidzgrl:j2jkUFDF6OGGqxkgg7Hk3ilbZI@ec2
```

If you recall from the section on getting the website ready to publish, we have to set environment variables for `DJANGO_SECRET_KEY` and `DJANGO_DEBUG`. Let's do this now.

> **Note**
>
> The secret key needs to be really secret! One way to generate a new key is to use the Django Secret Key Generator.

We set `DJANGO_SECRET_KEY` using the `config:set` command (as shown below). Remember to use your own secret key!

```
> heroku config:set DJANGO_SECRET_KEY="eu09(ilk6@4sfdofb=b_2ht@vad*$el

Setting DJANGO_SECRET_KEY and restarting locallibrary... done, v7
DJANGO_SECRET_KEY: eu09(ilk6@4sfdofb=b_2ht@vad*$ehh9-)3u_83+y%(+phh
```

We similarly set `DJANGO_DEBUG`:

```
> heroku config:set DJANGO_DEBUG='False'

Setting DJANGO_DEBUG and restarting locallibrary... done, v8
```

If you visit the site now you'll get a "Bad request" error, because the ALLOWED_HOSTS setting is *required* if you have `DEBUG=False` (as a security measure). Open **/locallibrary/settings.py** and change the `ALLOWED_HOSTS` setting to include your base app url (e.g. 'locallibrary1234.herokuapp.com') and the URL you normally use on your local development server.

```
ALLOWED_HOSTS = ['<your app URL without the https:// prefix>.herokuapp
# For example:
# ALLOWED_HOSTS = ['fathomless-scrubland-30645.herokuapp.com', '127.0
```

Then save your settings and commit them to your Github repo and to Heroku:

```
git add -A
git commit -m 'Update ALLOWED_HOSTS with site and development server U
git push origin main
git push heroku main
```

> **Note**
>
> After the site update to Heroku completes, enter a URL that does not exist (e.g. **/catalog/doesnotexist/**). Previously this would have displayed a detailed debug page, but now you should just see a simple "Not Found" page.

Debugging

The Heroku client provides a few tools for debugging:

```
# Show current logs
```

```
heroku logs

# Show current logs and keep updating with any new results
heroku logs --tail

# Add additional logging for collectstatic (this tool is run automati
heroku config:set DEBUG_COLLECTSTATIC=1

# Display dyno status
heroku ps
```

If you need more information than these can provide you will need to start looking into Django Logging.

Summary

That's the end of this tutorial on setting up Django apps in production, and also the series of tutorials on working with Django. We hope you've found them useful. You can check out a fully worked-through version of the source code on Github here.

The next step is to read our last few articles, and then complete the assessment task.

See also

- Deploying Django (Django docs)
  - Deployment checklist (Django docs)
  - Deploying static files (Django docs)
  - How to deploy with WSGI (Django docs)
  - How to use Django with Apache and mod_wsgi (Django docs)
  - How to use Django with Gunicorn (Django docs)
- Heroku
  - Configuring Django apps for Heroku (Heroku docs)
  - Getting Started on Heroku with Django (Heroku docs)
  - Django and Static Assets (Heroku docs)
  - Concurrency and Database Connections in Django (Heroku docs)
  - How Heroku works (Heroku docs)
  - Dynos and the Dyno Manager (Heroku docs)
  - Configuration and Config Vars (Heroku docs)
  - Limits (Heroku docs)
  - Deploying Python applications with Gunicorn (Heroku docs)

    - Deploying Python and Django apps on Heroku (Heroku docs)
    - Other Heroku Django docs
- Digital Ocean

    - How To Serve Django Applications with uWSGI and Nginx on Ubuntu 16.04
    - Other Digital Ocean Django community docs

In this module

- Django introduction
- Setting up a Django development environment
- Django Tutorial: The Local Library website
- Django Tutorial Part 2: Creating a skeleton website
- Django Tutorial Part 3: Using models
- Django Tutorial Part 4: Django admin site
- Django Tutorial Part 5: Creating our home page
- Django Tutorial Part 6: Generic list and detail views
- Django Tutorial Part 7: Sessions framework
- Django Tutorial Part 8: User authentication and permissions
- Django Tutorial Part 9: Working with forms
- Django Tutorial Part 10: Testing a Django web application
- Django Tutorial Part 11: Deploying Django to production
- Django web application security
- DIY Django mini blog

Last modified: Feb 24, 2021, by MDN contributors

Change your language

English (US) Change language