Deploying Python and Django Apps on Heroku

Last updated 15 November 2017

Table of Contents

This article describes how to take an existing Python or Django app and deploy it to Heroku.
If you are new to Heroku, you might want to start with the **Getting Started with Python on Heroku** tutorial.

Prerequisites

The best practices in this article assume that you have:

- an existing Python app
- Python installed locally. See the installation guides for OS X, Windows, and Linux
- Setuptools, Pip, and Virtualenv installed. See the Python install guides above for installation instructions

- a free Heroku account
- the Heroku CLI installed locally

Overview

The details of Heroku's Python Support are described in the Heroku Python Support article.
Heroku will recognize a deployed application as a Python application only when the application has a `requirements.txt` file in the root directory. Even if an application has no module dependencies, it should include an empty `requirements.txt` file to document that your app has no dependencies.

Django applications on Heroku

Heroku supports all types of Python applications, including Django applications.
The Configuring Django Apps for Heroku article describes this in detail and explains how to use the Heroku Django Starter Template which is strongly recommended for getting started with Django on Heroku.

Set up a virtual environment

The `virtualenv` tool keeps the dependencies required by different projects isolated, by creating virtual environments for them. To use it, install virtualenv.

```
$ pip install virtualenv
```

In the top-level directory of your project, create a virtual environment for your project.

```
$ virtualenv venv
```

Next, activate the virtual environment. You must source the virtualenv environment for each terminal session where you run your app.
If you are using Windows, run this command.

```
$ venv\Scripts\activate
```

If you are not using Windows, run this command.

```
$ source venv/bin/activate
```

While your virtual environment is active, all packages that you install using pip will be placed in your virtual environment.

Declare app dependencies

The `requirements.txt` file lists the app dependencies together with their versions. When an app is deployed, Heroku reads this file and installs the appropriate Python dependencies.
To generate a `requirements.txt` file, you can use the command `pip freeze`.

```
$ pip freeze > requirements.txt
```

Pip can also be used for advanced dependency management. See Python Dependencies via Pip to learn more.

Specify the version of Python

Optionally, you can specify the version of Python to use to run your application on Heroku. For more information, see Supported Python Runtimes.

The Procfile

A Procfile is a text file in the root directory of your application that defines process types and explicitly declares what command should be executed to start your app.
Your `Procfile` will look something like this:

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

This declares a single process type, `web`, and the command needed to run it. The name, `web`, is important here. It declares that this process type will be attached to the HTTP routing stack of Heroku, and receive web traffic when deployed.
The command in a web process type must bind to the port number specified in the `PORT`environment variable. If it does not, the dyno will not start.

Build your app and run it locally

Run this command in your local directory to install the dependencies locally.

```
$ pip install -r requirements.txt
Downloading/unpacking ...
...
Successfully installed Django dj-database-url dj-static django-toolbelt gunicorn
psycopg2 static3
Cleaning up...
```

Windows users may encounter an error when installing the dependencies locally. The production web server that we recommend, Gunicorn, does not work on Windows. To get your app running locally, use the default web server instead:

- Edit the `requirements.txt` file. Remove the line containing `gunicorn` and add another, `psycopg2==2.5.3`
- Run pip again:`pip install -r requirements.txt` - it should now work
- Edit the `Procfile` so that your app will start using the default web server instead of Gunicorn. The contents of the file should read: `web: python manage.py runserver 0.0.0.0:$PORT`

  While this will get you up and running locally, please ensure you revert to Gunicorn when running in production.
  To run your application locally, use the heroku local command, which was installed as part of the Heroku CLI:

```
$ heroku local web
11:48:19 web.1  | started with pid 36084
11:48:19 web.1  | 2014-07-17 11:48:19 [36084] [INFO] Starting gunicorn 19.0.0
11:48:19 web.1  | 2014-07-17 11:48:19 [36084] [INFO] Listening at:
http://0.0.0.0:5000 (36084)
11:48:19 web.1  | 2014-07-17 11:48:19 [36084] [INFO] Using worker: sync
11:48:19 web.1  | 2014-07-17 11:48:19 [36087] [INFO] Booting worker with pid: 36087
```

  Just like Heroku, `heroku local` examines the `Procfile` to determine what to run.
  Your app should now be running on http://localhost:5000/.

How to keep build artifacts out of git

Prevent build artifacts from going into revision control by creating a `.gitignore` file. Here's a typical `.gitignore` file:

```
venv
*.pyc
staticfiles
.env
```

Deploy your application to Heroku

After you commit your changes to git, you can deploy your app to Heroku.

```
$ git add .
$ git commit -m "Added a Procfile."
$ heroku login
Enter your Heroku credentials.
...
$ heroku create
Creating intense-falls-9163... done, stack is cedar
http://intense-falls-9163.herokuapp.com/ | git@heroku.com:intense-falls-9163.git
Git remote heroku added
$ git push heroku master
...
-----> Python app detected
...
-----> Launching... done, v7
       https://intense-falls-9163.herokuapp.com/ deployed to Heroku
```

To open the app in your browser, type `heroku open`.

One-off dynos

Heroku allows you to run commands in a one-off dyno by using the `heroku run` command. Use this for scripts and applications that only need to be executed when needed, such as maintenance tasks, loading fixtures into a database, or database migrations during application updates.
For debugging purposes, such as to examine the state of your application after a deploy, you can use `heroku run bash` for a full shell into a one-off dyno. But remember that this will not connect you to one of the web dynos that may be running at the same time.

```
$ heroku run bash
Running `bash` attached to terminal... up, run.2365
$ cat Procfile
web: gunicorn gettingstarted.wsgi --log-file -
$ exit
exit
```

You can also use `heroku run` to launch a Python shell attached to your local terminal for experimenting in your app's environment. From there, you can import some of your application modules. Type `quit()` to close the interactive shell.

```
$ heroku run python
Running `python` attached to terminal... up, run.8826
Python 2.7.8 (default, Jul 15 2014, 15:37:51)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> quit()
```

Similarly, if you are using Django, you can use `heroku run` to get a Django shell for executing arbitrary code against your deployed app:

```
$ heroku run python manage.py shell
Running python manage.py shell attached to terminal... up, run.1
Python 2.7.6 (default, Jan 16 2014, 02:39:37)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from django.contrib.auth.models import User
>>> User.objects.all()
[<User: kenneth>]
```

You can also use the `heroku run` command to sync the Django models with the database schema:

```
$ heroku run python manage.py syncdb
Running python manage.py syncdb attached to terminal... up, run.1
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site
```

```
You just installed Django's auth system, which means you don't have any superusers
defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'u53976'): kenneth
Email address: kenneth@heroku.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

Logging

Heroku treats logs as streams and expects them to be written to STDOUT and STDERR and not to its ephemeral filesystem.
Refer to the logging documentation for more general information about logging on Heroku.

Provision add-ons

The add-on marketplace has a large number of add-ons to choose from.
For Django applications, a Heroku Postgres `hobby-dev` database is automatically provisioned. This populates the `DATABASE_URL` environment variable.
No add-ons are automatically provisioned if a pure Python application is detected. If you need a SQL database for your app, add one explicitly:

```
$ heroku addons:create heroku-postgresql:hobby-dev
```

Running a worker

The `Procfile` format lets you run any number of different process types. For example, if you added a worker process to complement your web process, your updated `Procfile` will look something like this:

```
web: gunicorn gettingstarted.wsgi --log-file -
worker: python worker.py
```

Running more than one dyno for an extended period may incur charges to your account. Read more about **dyno-hour costs**.

Push this change to Heroku, then launch a worker:

```
$ heroku ps:scale worker=1
Scaling worker processes... done, now running 1
```

Check `heroku ps` to see that your worker comes up, and `heroku logs` to see your worker doing its work.

Next steps

- Read the Heroku Python Support documentation.
- Explore the Python category on Dev Center.

Keep reading

- Python Support
- Heroku Python Support
- Process Types and the Procfile
- Dynos and the Dyno Manager