Getting Started with Django on Heroku

Last updated 09 September 2015

Table of Contents

This quickstart will get you going with a Python/Django application that uses a Postgres database, deployed to Heroku. For other Python apps, see Getting Started with Python on Heroku. For general information on how to develop and architect apps for use on Heroku, see Architecting Applications for Heroku.

If you have questions about Python on Heroku, consider discussing it in the **Python on Heroku forums**. Both Heroku and community-based Python experts are available.

Prerequisites

- The Heroku Toolbelt, as described in Getting Started with Python.
- Installed Python and Virtualenv in a unix-style environment. See this guide for guidance.
- An installed version of Postgres to test locally.
- A Heroku user account. Signup is free and instant.

Start a Django app inside a Virtualenv

First, we'll create an empty top-level directory for our project:

```
$ mkdir hellodjango && cd hellodjango
```

Make sure you’re using the latest virtualenv release. If you’re using a version that comes with Ubuntu, you may need to add the `--no-site-packages` flag.

Next, we’ll create a Python Virtualenv (v1.0+):

```
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip...done.
```

To use the new virtualenv, we need to activate it. (You must source the virtualenv environment for each terminal session where you wish to run your app.)

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

Next, install our application’s dependencies with pip. In this case, we will be installing **django-toolbelt**, which includes all of the packages we need:

- Django (the web framework)
- Gunicorn (WSGI server)
- dj-database-url (a Django configuration helper)

- dj-static (a Django static file server)

From your virtualenv:

```
$ pip install django-toolbelt
Installing collected packages: Django, psycopg2, gunicorn, dj-database-url, dj-static
  ...
Successfully installed Django psycopg2 gunicorn dj-database-url dj-static static
Cleaning up...
```

Now that we have a clean Python environment to work in, we'll create our simple Django application.

Don't forget the `.` at the end. This tells Django to put the extract the into the current directory, instead of putting it in a new subdirectory.

```
$ django-admin.py startproject hellodjango .
```

Declare process types with Procfile

Use a Procfile, a text file in the root directory of your application, to explicitly declare what command should be executed to start a web dyno. In this case, you need to execute Gunicorn with a few arguments.

Here's a `Procfile` for our new app. It should be called `Procfile` and live at the root directory of our project:

Procfile

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

You can now start the processes in your Procfile locally using the `heroku local` command, installed as part of the Heroku Toolbelt:

```
$ heroku local
2013-04-03 16:11:22 [8469] [INFO] Starting gunicorn 0.17.2
2013-04-03 16:11:22 [8469] [INFO] Listening at: http://127.0.0.1:8000 (8469)
```

Make sure things are working properly `curl` or a web browser, then Ctrl-C to exit.

Specify dependencies with Pip

Heroku recognizes Python applications by the existence of a `requirements.txt` file in the root of a repository. This simple format is used by most Python projects to specify the external

Python modules the application requires.

Pip has a nice command (`pip freeze`) that will generate this file for us:

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

requirements.txt

```
Django==1.7
dj-database-url==0.3.0
dj-static==0.0.6
gunicorn==19.1.1
psycopg2==2.5.1
static==0.4
wsgiref==0.1.2
```

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

Django settings

Next, configure the application for the Heroku environment, including Heroku's Postgres database. The dj-database-url module will parse the value of the `DATABASE_URL` environment variable and convert them to something Django can understand.

Make sure ‘dj-database-url’ is in your requirements file, then add the following to the bottom of your `settings.py` file:

settings.py

```
# Parse database configuration from $DATABASE_URL
import dj_database_url
DATABASES['default'] =  dj_database_url.config()

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

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

# Static asset configuration
import os
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
STATIC_ROOT = 'staticfiles'
STATIC_URL = '/static/'

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, 'static'),
)
```

With these settings available, you can add the following code to `wsgi.py` to serve static files in production:

wsgi.py

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

application = Cling(get_wsgi_application())
```

Store your app in Git

Now that we've written and tested our application, we need to store the project in a Git repository.

Since our current directory contains a lof of extra files, we'll want to configure our repository to ignore these files with a `.gitignore` file:

GitHub provides an excellent **Python gitignore file** that can be **installed system-wide**.

.gitignore

```
venv
*.pyc
staticfiles
```

Next, we'll create a new git repository and save our changes.

```
$ git init
Initialized empty Git repository in /Users/kreitz/hellodjango/.git/
$ git add .
$ git commit -m "my django app"
[master (root-commit) 2943412] my django app
 7 files changed, 230 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Procfile
 create mode 100644 hellodjango/__init__.py
 create mode 100644 hellodjango/settings.py
 create mode 100644 hellodjango/urls.py
 create mode 100644 hellodjango/wsgi.py
 create mode 100644 manage.py
 create mode 100644 requirements.txt
```

Deploy to Heroku

The next step is to push the application's repository to Heroku. First, we have to get a place to push to from Heroku. We can do this with the `heroku create` command:

```
$ heroku create
Creating simple-spring-9999... done, stack is cedar-14
http://simple-spring-9999.herokuapp.com/ | git@heroku.com:simple-spring-9999.git
Git remote heroku added
```

This automatically added the Heroku remote for our app (`git@heroku.com:simple-spring-9999.git`) to our repository. Now we can do a simple `git push` to deploy our application:

```
$ git push heroku master
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (11/11), 4.01 KiB, done.
Total 11 (delta 0), reused 0 (delta 0)
-----> Python app detected
-----> No runtime.txt provided; assuming python-2.7.6.
-----> Preparing Python runtime (python-2.7.6)
-----> Installing Distribute (0.6.36)
-----> Installing Pip (1.3.1)
-----> Installing dependencies using Pip (1.3.1)
       Downloading/unpacking Django==1.5 (from -r requirements.txt (line 1))
       ...
       Successfully installed Django psycopg2 gunicorn dj-database-url dj-static stat
       Cleaning up...
-----> Collecting static files
       0 static files copied.

-----> Discovering process types
       Procfile declares types -> web

-----> Compiled slug size is 29.5MB
-----> Launching... done, v6
       http://simple-spring-9999.herokuapp.com deployed to Heroku

To git@heroku.com:simple-spring-9999.git
```

```
 * [new branch]      master -> master
```

Visit your application

You've deployed your code to Heroku, and specified the process types in a `Procfile`. You can now instruct Heroku to execute a process type. Heroku does this by running the associated command in a dyno - a lightweight container which is the basic unit of composition on Heroku.

Let's ensure we have one dyno running the `web` process type:

```
$ heroku ps:scale web=1
```

You can check the state of the app's dynos. The `heroku ps` command lists the running dynos of your application:

```
$ heroku ps
=== web: `gunicorn hellodjango.wsgi`
web.1: up for 10s
```

Here, one dyno is running.

We can now visit the app in our browser with `heroku open`.

```
$ heroku open
Opening simple-spring-9999.herokuapp.com... done
```

You should see the satisfying “It worked!” Django welcome page.

Dyno sleeping and scaling

By default, your app is deployed on a free dyno. Free dynos will sleep after a half hour of inactivity and they can be active (receiving traffic) for no more than 18 hours a day before going to sleep. If a free dyno is sleeping, and it hasn’t exceeded the 18 hours, any web request will wake it. This causes a delay of a few seconds for the first request upon waking. Subsequent requests will perform normally.

To avoid dyno sleeping, you can upgrade to a hobby or professional dyno type as described in the Dyno Types article. For example, if you migrate your app to a professional dyno, you can easily scale it by running a command telling Heroku to execute a specific number of dynos, each running your web process type.

View the logs

Heroku treats logs as streams of time-stamped events aggregated from the output streams of all the dynos running the components of your application. Heroku's Logplex provides a single channel for all of these events.

View information about your running app using one of the logging commands, `heroku logs`:

```
$ heroku logs
2012-04-06T19:38:25+00:00 heroku[web.1]: State changed from created to starting
2012-04-06T19:38:29+00:00 heroku[web.1]: Starting process with command `gunicorn hell
2012-04-06T19:38:29+00:00 app[web.1]: Validating models...
2012-04-06T19:38:29+00:00 app[web.1]:
2012-04-06T19:38:29+00:00 app[web.1]: 0 errors found
2012-04-06T19:38:29+00:00 app[web.1]: Django version 1.5, using settings 'hellodjango
2012-04-06T19:38:29+00:00 app[web.1]: Development server is running at http://0.0.0.0
2012-04-06T19:38:29+00:00 app[web.1]: Quit the server with CONTROL-C.
2012-04-06T19:38:30+00:00 heroku[web.1]: State changed from starting to up
2012-04-06T19:38:32+00:00 heroku[slugc]: Slug compilation finished
```

Syncing the database

The `heroku run` command lets you run one-off dynos. You can use this 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 de
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)
```

Using the Django shell

Similarly, 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>]
```

Next steps

Now that you've deployed your first Django application to Heroku, it's time to take the next step! If if you'd like to learn more about Heroku, these articles are a great place to start.

Heroku Reference

- How Heroku Works
- Heroku Reference Documentation

Python Reference

- Django and Static Assets
- Using Celery on Heroku

- Deploying Python Applications with Gunicorn
- Specifying a Python Runtime
- Python Dependencies via Pip
- Background Tasks in Python with RQ

django python

Keep reading

- Python
- Dynos and the Dyno Manager
- Getting Started with Python on Heroku
- Python Dependencies via Pip

Feedback

Log in to submit feedback.

‹ Direct to S3 File Uploads in Python

Getting Started with Python on Heroku ›