Click here to code!

PyBites

A Community that Masters Python through Code Challenges

Articles Blog Challenges Python Exercises #100DaysOfCode Join Our Slack

Search

Join our community and grab our *Become a Better Python Developer* cheat sheet. Learn Python. Receive bonus material. Challenge yourself! (Privacy Policy)

Email Address

Join Us

Step-by-Step Guide on Deploying a Simple Flask App to Heroku

Posted by Julian on Fri 21 July 2017 in Flask • 5 min read

Something I've always wanted to do since I first started working in IT was make my own application and have it running on the internet/intranet somewhere. It took years (procrastination yo!) but this week I finally made it happen! I deployed my first app to Heroku! (Note, not exactly mobile friendly - yet!).

In hindsight the process was simple but there were definitely some gotchas. The Heroku documentation, while well written and detailed, makes some assumptions *and* is based on a Django app.

Enter this Flask tutorial!

Step by Step

1. The first thing you have to do is download and install the Heroku CLI for your OS. I won't cover this here as it's extremely straightforward and the documentation is great.

2. Log into your Heroku account using the CLI:

```
$ heroku login
Enter your Heroku credentials:
Email: pybites_rocks@fakemail.com
Password: **********
Logged in as pybites_rocks@fakemail.com
```

3. Initialise the directory with your app in it as a git repo:

```
$ cd ../flaskapps/timezone_printer
$ git init
```

4. We need to install `gunicorn`. This is a Python web server for UNIX based OS's. It's required to have it in your code venv in order to launch the Flask app on Heroku. Initialise the virtualenv in your dir and install it.

```
flaskapps/timezone_printer$ source venv/bin/activate
(venv)flaskapps/timezone_printer$ pip install gunicorn
```

> If you're not sure what a virtualenv is, check out our article!

5. Next, if you haven't done so already, create your requirements.txt file. Heroku relies on the `requirements.txt` file to exist in order to deploy your app. When you deploy Heroku uses pip and requirements.txt to install everything required to run your app:

```
(venv)$ pip freeze > requirements.txt
```

6. Use the normal git commands to add and commit the code (note, repo has not been synced with Heroku yet):

```
$ git add .
$ git commit -m "Initial push of timezone printer flask app code t
```

7. Now we actually create your app instance on the Heroku servers. This is where you specify the app name. I'm calling mine `timezone-printer`:

```
$ heroku create timezone-printer
Creating timezone-printer... done
https://timezone-printer.herokuapp.com/ | https://git.heroku.com/t
```

8. You can now start pushing your code to the app you've just created. To confirm you're pushing to the correct remote app repo run the following:

```
$ git remote -v
git remote -v
heroku  https://git.heroku.com/timezone-printer.git (fetch)
heroku  https://git.heroku.com/timezone-printer.git (push)
```

This information comes from the git config file. If I `cd` into my other directory that houses the code for a different Heroku app, I could run the `git remote -v` command and the output would change. Eg:

```
$ cd ../BMI_calc
$ git remote -v
heroku  https://git.heroku.com/pybites-bmi.git (fetch)
heroku  https://git.heroku.com/pybites-bmi.git (push)
```

9. Let's push the timezone-printer code to Heroku. This push not only pushes the code but also installs all dependencies using the `requirements.txt` file we created in step 4. Ignore the message regarding the Procfile. I'll get to that in the next step:

```
$ git push heroku master
Counting objects: 1783, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (1736/1736), done.
Writing objects: 100% (1783/1783), 10.13 MiB | 353.00 KiB/s, done.
Total 1783 (delta 161), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote:  !     Warning: Your application is missing a Procfile. Th
remote:  !     Learn more: https://devcenter.heroku.com/articles/p
remote: -----> Installing python-3.6.2
remote: -----> Installing pip
remote: -----> Installing requirements with pip
<snip>
remote:        Successfully installed Flask-0.12.2 Jinja2-2.9.6 Ma
remote:
remote: -----> Discovering process types
remote:        Procfile declares types -> (none)
remote:
remote: -----> Compressing...
remote:        Done: 68.9M
remote: -----> Launching...
remote:        Released v3
remote:        https://timezone-printer.herokuapp.com/ deployed to
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/timezone-printer.git
 * [new branch]      master -> master
```

10. Yay! The code is deployed on Heroku! It won't work, but it's deployed! Why won't it work? Well, think of it this way. Your code *exists* on the Heroku servers but Heroku has no idea what to do with it.

This is where the Procfile comes in. The Procfile will be the command Heroku runs to initiate your code. It'd be like you running `python app.py` or other. Create a file in the root directory of your app and enter the following:

```
$ vim Procfile
web: gunicorn app:app
```

Substitute the first "app" in the above code with the name of the script you want to execute. The Python script that launches my Flask app is called `app.py` thus the Procfile contains app:app. If the script was titled `tz_lister.py` the Procfile would read:

```
web: gunicorn tz_lister:app
```

11. While you don't *need* to, you can specify which version of Python to run. At the time of

writing, by default Heroku uses Python 3.6.2. If you need another version you can specify which one to use by creating a `runtime.txt` file:

```
$ vim runtime.txt
python-3.6.0
```

12. Now we can git add, commit and push the code to Heroku:

```
$ git add .
$ git commit -m "Added Procfile and runtime.txt files"
$ git push heroku master
```

Notice that the push doesn't just push the changes but it also results in a rebuild. The requirements.txt file is checked once more and so forth. You should also notice the Profile error is no longer apparent.

13. With all of our files in place, it's time to spin up the app!

```
$ heroku ps:scale web=1
Scaling dynos... done, now running web at 1:Free
```

If you're on the free tier of Heroku you can only have a limited amount of apps running at a time so keep that in mind.

14. Bam! The app should be up and running! You can use the Heroku cli to launch the app in your default browser:

```
$ heroku open
```

Handy Tips

A couple of tips and learning points from my experience running my Flask app on Heroku.

1. You can check the logs of your Heroku app using `heroku logs`. If you want a live view of the logs then run `heroku logs --tail`.

2. Check to see if you have any other Heroku dynos running with `heroku ps`. It'll also tell you how many hours you have left of your quota. Handy if you're on the free tier!

3. If you find an error in your logs relating to the IP address being in use, it may be the Flask app code itself. I found that in one of my older Flask apps I hadn't included the below line of code which resulted in Heroku failing to launch the app:

```
if __name__ == '__main__':
    app.run()
```

Conclusion

This was one of **the most satisfying** things I've ever done within the realms of programming. I can't stress how great it feels to see my own app up on the Internet.

I actually added a second app just to confirm not all of my work formats terribly on mobile. Check it out here.

Given it's completely free and simple to do, I strongly urge everyone to give it a go. Not only is it a great morale/motivation boost but it's something to add to your portfolio.

I'm ticking this one off the bucket list!

Keep Calm and Code in Python!

-- Julian

See an error in this post? Please submit a pull request on Github.

Flask python beginner sessions learning code Heroku

Like this article? Share it with your friends!