CODING NOMADS

19) Deploying Your App On The Internet Lesson

Part 2: Configuration for Deployment on Heroku

21 min to complete · By Brandon Gigous

Contents

This is Part 2 of our Flask Deployment tutorial series:

Part 1: Setup for Flask Deployment on Heroku
Part 2: Configuration for Deployment on Heroku
Part 3: Using Git & Debugging Your Deployment on Heroku

If you haven't already completed Part 1, visit that lesson and return here, and make sure to check out Part 3 to tie it all together!

In this lesson, you'll learn how to manage environment variables for deployment on Heroku and how to set up security features like TLS and CSRF protection.

Prepare for Deployment on Heroku

Your config vars are set. Now you need to let your Flask app know how to *find* all of them on Heroku. Open up your `config.py` file and make the following changes:

`SECRET_KEY` variable

Change your `SECRET_KEY` setting to the following:

```
import os

SECRET_KEY = os.environ.get('SECRET_KEY') or 'anyRandomLongStringUse
```

Remember that you will want to declare a long random string as your *actual* `SECRET_KEY` and put that value into your Heroku config vars. Flask cannot operate without a `SECRET_KEY` variable being set. This is why you'll keep a *default value* as the second argument in the method call. The default value is used in case Flask can't find a value with the name `SECRET_KEY` that you need to define in your config vars.

Note: Keep in mind that the default value, in this case, `'anyRandomLongStringUseNumbersYo#123'` should *never* be used in production.

`FLASK_CONFIG` variable

You definitely do not want to use your app's `default` configuration for production. That's why you'll make a new configuration (or two) for production. You'll learn more about that soon.

The general adaptations mentioned above are necessary for any Flask deployment. You will *always* need to make sure you are not showing the `SECRET_KEY` in production, and the `FLASK_CONFIG` value must be a configuration created with production in mind. You'll build such a configuration next.

Production Configuration

Here is an example of a production configuration class:

```
class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        f'sqlite:///{os.path.join(basedir, "data.sqlite")}'

    @classmethod
    def init_app(cls, app):
        Config.init_app(app)

        import logging
        from logging.handlers import SMTPHandler
        creds = None
        secure = None
        if getattr(cls, 'MAIL_USERNAME', None) is not None:
            creds = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD)
            if getattr(cls, 'MAIL_USE_TLS', None):
                # logging: to use TLS, must pass tuple (can be empty
                secure = ()
        mail_handler = SMTPHandler(
            mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT),
            fromaddr=cls.RAGTIME_MAIL_SENDER,
            toaddrs=[cls.RAGTIME_ADMIN],
            subject=cls.RAGTIME_MAIL_SUBJECT_PREFIX + " Application
            credentials=creds,
            secure=secure
        )
        mail_handler.setLevel(logging.ERROR)
        app.logger.addHandler(mail_handler)
```

You might remember that the Flask application instance has its own logger attached as `app.logger`. The logger writes any log messages to the console in debug mode (meaning `FLASK_DEBUG=1`), but since production is *not* part of debug, how would you possibly see that your app has any problems? One way is to attach another handler to the app logger, one that can send an email when an error occurs.

This class, also based on the base `Config` class like your other configuration classes, actually has a use for the `init_app()` method that no configuration classes so far have used. You use this method to set up the log handler for sending emails.

The `SMTPHandler` class is imported, but before invoking it to create an instance of `SMTPHandler` class, there's first a bit of setup that needs to be done. Mostly, it's the settings for SMTP that the `app.logger` object will need to be able to send emails about errors, and of course, the username and password for the email are needed.

Later, when the handler is created, the SMTP server settings are also passed. You can check out the `SMTPHandler` class **here** for the details.

Oh, what? I forgot to mention something. Maybe even forgot to tell you about setting up an environment variable? Ah, the `SQLALCHEMY_DATABASE_URI`, set from the environment variable `DATABASE_URL`. Well, if somehow there is no environment variable with that name, then it will be a SQLite file like you've been doing before. However, Heroku doesn't use SQLite but does use PostgreSQL. Now what?

Well, to set `DATABASE_URL` to the proper value AND set with a value reflecting a PostgreSQL database is actually surprisingly easy. Just add Heroku's convenient PostgreSQL addon to your project:

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on <app name>
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-spherical-70199 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

Heroku's cheapest version of the PostgreSQL let's you have up to 10000 rows of data. And do you see that? It looks like `DATABASE_URL` has been set for you. Pretty convenient, huh? That means, when you deploy your app on Heroku, it will take care of the `DATABASE_URL` environment variable for you.

Heroku Specific Configuration

So you're off to a good start with the `ProductionConfig`; in general, it's a good configuration for almost any production environment. Because Heroku has its own "flavor", it's good to create a configuration just for Heroku based on the production configuration.

What is this flavor I'm talking about? Well, Heroku considers log output other than just error messages. You can write more general and informative log messages, and Heroku will capture it. Why would you want to capture messages other than errors? Because knowledge is power! Take a look at the new config:

```
class HerokuConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler
        file_handler.setLevel(file_handler, level=logging.INFO)
        app.logger.addHandler(file_handler)
```

Note that this configuration inherits from the `ProductionConfig` class, and also has a `init_app()` method. This method is used to add a new handler to the app logger, one that will write log messages to the standard error stream. The level handled is `INFO`, which also allows warnings to pass through to the log output.

Now, what about that `heroku` config setting you set earlier? You might have guessed correctly already. You'll need to add it to your `config` dictionary:

```
config = {
    # ...
    'heroku': HerokuConfig,
    # ...
}
```

Heroku requires some additional platform-specific settings that you will adjust next. Next, you will follow Heroku's instructions to ensure your app is ready for Heroku deployment.

Connect Flask to Heroku with a Procfile

First, and most importantly, Heroku web applications require a `Procfile` .

This file is used to explicitly declare your application's process types and entry points, and you can read more detailed information in Heroku's dev center. The file should be located in the root of your repository. The `Procfile` helps the web server software to know how to communicate with your app.

```
web: gunicorn ragtime:app
```

This Procfile can initiate a variety of tasks. For your web app, the `Procfile` defined above will simply start the production web server, in this case, using Gunicorn. What does the `web` mean? It's actually a task name recognized by Heroku to start the web server. Then, Gunicorn takes the name of your module and the name of your application instance separated with a colon.

Info: The Procfile does not have any file extension. You can simply name it `Procfile` .

Mentorship Makes the Difference!

Installing Gunicorn

`gunicorn` is your web server that will handle the communication between your Flask app and the web. You can install it with pip:

```
$ pip install gunicorn
```

That's all you need to do with Gunicorn.

Security

You're about to release your app into the wilds of the internet, and security should be at the top of your priorities. Below, you'll learn how to easily add top-notch security to your Flask app.

Adding Encryption To Requests

While you've been developing your app, you've been using regular ol' HTTP to make everything work. But using *less secure, vanilla HTTP* is not going to cut it for a production webapp, as it is possible for a user's credentials to get intercepted while in transit to the server. There are also other considerations that the Flask development team outlines here. You can easily add secure HTTP, or **HTTPS**, using Flask-Talisman:

```
$ pip install flask-talisman
```

Along with helping you quickly apply protections against some common web security issues, Flask-Talisman will enable TLS to encrypt all communications between clients and server, which is a win for both you and your users. But you only need to activate this when your app is in a production environment, so in your application factory, you'll want to add this code:

```
# app/__init__.py

def create_app(config_name='default'):
    # ...
    if app.config['HTTPS_REDIRECT']:
        from flask_talisman import Talisman
        Talisman(app, content_security_policy={
                'default-src': [
                    "'self'",
                    'cdnjs.cloudflare.com',
                ],
                # allow images from anywhere,
                #   including unicornify.pictures
                'img-src': '*'
            }
        )
```

At this point, adding another extension to your app should be somewhat trivial, so what's up with all this content `security_policy` stuff? By default, Flask-Talisman would clamp down on security by *outright disallowing scripts or styles* from outside your app, known as `'self'` above. Safe, but a little *too* safe. The additions to the policy will allow your app to use Bootstrap styling and download images from anywhere.

Info: If you add new features to your app, you may need to update the `content_security_policy` to allow additional resources from outside your app.

And that's right, you'll need to add `HTTPS_REDIRECT` as a configuration variable:

```
class Config():
    # ...
    HTTPS_REDIRECT = False

class HerokuConfig(ProductionConfig):
    HTTPS_REDIRECT = True if os.environ.get('DYNO') else False
    # ...

    from werkzeug.middleware.proxy_fix import ProxyFix
    app.wsgi_app = ProxyFix(app.wsgi_app)
```

`HTTPS_REDIRECT` is only set to `True` if the `DYNO` environment variable exists, which is set by Heroku in its environment. If you were to test your Heroku configuration locally, your app would not use TLS.

"Hey, wait, what's this `ProxyFix` thing?" You see, Heroku uses a *reverse proxy server* to redirect any client requests meant for your website to *your* app; these client requests don't go directly to your app but instead to this proxy server. Because of how Heroku's proxy server **works**, your app would get confused and would send external links like confirmation tokens through `http://` . To ensure your app *doesn't* get confused, you can use Werkzeug's `ProxyFix` to, well, fix it.

Cross-Site Request Forgery (CSRF) Protection

While you're at, you can easily add global CSRF protection across your app. You already learned a bit about this back in the section about forms, and they are already protected from CSRF attacks, but any requests outside your form may not be. Flask-WTF's CSRFProtect object will do the job:

```
from flask_wtf.csrf import CSRFProtect
# ...
csrf = CSRFProtect()

def create_app(config_name='default'):
    # ...
    csrf.init_app(app)
```

It may be overkill in terms of security, but doing this now will provide peace of mind in case you add certain features to your app later.

Deployment Tasks

When you install your app on a production server, you still need to be able to perform setup tasks like creating your database tables, upgrading your database, and inserting roles. Doing this all manually is tedious because it could lead to errors, so adding a `deploy` command to do these tasks is a great idea:

```
from flask_migrate import upgrade

@app.cli.command()
def deploy():
    """ Run deployment tasks """
    # migrate database
    upgrade()

    Role.insert_roles()

    User.add_self_follows()
```

You'll see how to use this command once you actually deploy your app to Heroku.

Mentorship Makes the Difference!

Top-Level `requirements.txt` File

The last step to get your Flask app ready for deployment is to add a top-level `requirements.txt` file. Heroku looks in the top-level directory of your app for this file in order to install package dependencies. Before you create this file, you'll first want to make a `heroku.txt` file for your Heroku-specific dependencies. This will include Flask-Talisman and Gunicorn, *as well as* psycopg2, which Heroku will need to get your fancy PostgreSQL production database going. The file will look something like this:

```
# requirements/heroku.txt
-r prod.txt
flask-talisman==0.7.0
gunicorn==20.1.0
psycopg2==2.8.6
```

Version numbers may vary, of course. Then, you just need to copy those requirements to the new `requirements.txt` file:

```
-r requirements/heroku.txt
```

That's it. You are now ready to deploy your app. On the next page you will learn how to finally get your webapp deployed on Heroku! You're almost there, so keep going!

Summary: Configuration for Flask Deployment on Heroku

In this lesson, you:

- Discovered how to manage **environment variables** on Heroku using `heroku config:set`, or with the GUI.
- Tackled the application security by setting up **TLS with Flask-Talisman** and enabled CSRF protection on non-form endpoints with **Flask-WTF CSRFProtect**.
- Addressed **deployment tasks**, such as database migrations and role creations, using custom Flask CLI commands.
- Wrote a `Procfile` to declare how Heroku should run your app.
- Wrote a `requirements.txt` for tracking dependencies to be installed by Heroku during deployment.

Previous Next → Part 3: Using Git & Debugging Your Deployment on...