The Flask Mega-Tutorial Part XVIII: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku)

April 3 2018

Posted by Miguel Grinberg (/author/Miguel Grinberg) under Python (/category/Python), Flask (/category/Flask), Programming (/category/Programming), Cloud (/category/Cloud).

This is the eighteenth installment of the Flask Mega-Tutorial series, in which I'm going to deploy Microblog to the Heroku cloud platform.

For your reference, below is a list of the articles in this series.

- Chapter 1: Hello, World! (/post/the-flask-mega-tutorial-part-i-hello-world)
- Chapter 2: Templates (/post/the-flask-mega-tutorial-part-ii-templates)
- Chapter 3: Web Forms (/post/the-flask-mega-tutorial-part-iii-web-forms)
- Chapter 4: Database (/post/the-flask-mega-tutorial-part-iv-database)
- Chapter 5: User Logins (/post/the-flask-mega-tutorial-part-v-user-logins)
- Chapter 6: Profile Page and Avatars (/post/the-flask-mega-tutorial-part-vi-profile-page-and-avatars)
- Chapter 7: Error Handling (/post/the-flask-mega-tutorial-part-vii-error-handling)
- Chapter 8: Followers (/post/the-flask-mega-tutorial-part-viii-followers)
- Chapter 9: Pagination (/post/the-flask-mega-tutorial-part-ix-pagination)
- Chapter 10: Email Support (/post/the-flask-mega-tutorial-part-x-email-support)
- Chapter 11: Facelift (/post/the-flask-mega-tutorial-part-xi-facelift)
- Chapter 12: Dates and Times (/post/the-flask-mega-tutorial-part-xii-dates-and-times)
- Chapter 13: I18n and L10n (/post/the-flask-mega-tutorial-part-xiii-i18n-and-l10n)
- Chapter 14: Ajax (/post/the-flask-mega-tutorial-part-xiv-ajax)
- Chapter 15: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)
- Chapter 16: Full-Text Search (/post/the-flask-mega-tutorial-part-xvi-full-text-search)
- Chapter 17: Deployment on Linux (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux)
- Chapter 18: Deployment on Heroku (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku) (this article)
- Chapter 19: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)
- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)

- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

In the previous article I showed you the "traditional" way to host a Python application, and I gave you two actual examples of deployment to Linux based servers. If you are not used to manage a Linux system, you probably thought that the amount of effort that needs to be put into the task was big, and that surely there must be an easier way.

In this chapter I'm going to show you a completely different approach, in which you rely on a third-party *cloud* hosting provider to perform most of the administration tasks, freeing you to spend more time working on your application.

Many cloud hosting providers offer a managed platform on which applications can run. All you need to provide to have your application deployed on these platforms is the actual application, because the hardware, operating system, scripting language interpreters, database, etc. are all managed by the service. This type of service is called Platform as a Service (https://en.wikipedia.org/wiki/Platform_as_a_service), or PaaS.

Sounds too good to be true, right?

I will look at deploying Microblog to Heroku (http://heroku.com), a popular cloud hosting service that is also very friendly for Python applications. I picked Heroku not only because it is popular, but also because it has a free service level that will allow you to follow me and do a complete deployment without spending any money.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.18), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.18.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.17...v0.18).

Hosting on Heroku

Heroku was one of the first platform as a service providers. It started as a hosting option for Ruby based applications, but then grew to support many other languages like Java, Node.js and of course Python.

Deploying a web application to Heroku is done through the `git` version control tool, so you must have your application in a git repository. Heroku looks for a file called *Procfile* in the application's root directory for instructions on how to start the application. For Python projects, Heroku also expects a *requirements.txt* file that lists all the module dependencies that need to be installed. After the application is uploaded to Heroku's servers through git, you are essentially done and just need to wait a few seconds until the application is online. It's really that simple.

The different service tiers Heroku offers allow you to choose how much computing power and time you get for your application, so as your user base grows you will need to buy more units of computing, which Heroku calls "dynos".

Ready to try Heroku? Let's get started!

Creating Heroku account

Before you can deploy to Heroku you need to have an account with them. So visit heroku.com (https://id.heroku.com/signup) and create a free account. Once you have an account and log in to Heroku, you will have access to a dashboard, where all your applications are listed.

Installing the Heroku CLI

Heroku provides a command-line tool for interacting with their service called Heroku CLI (https://devcenter.heroku.com/articles/heroku-cli), available for Windows, Mac OS X and Linux. The documentation includes installation instructions for all the supported platforms. Go ahead and install it on your system if you plan on deploying the application to test the service.

The first thing you should do once the CLI is installed is login to your Heroku account:

```
$ heroku login
```

Heroku CLI will ask you to enter your email address and your account password. Your authenticated status will be remembered in subsequent commands.

Setting Up Git

The `git` tool is core to the deployment of applications to Heroku, so you must install it on your system if you don't have it yet. If you don't have a package available for your operating system, you can visit the git site (https://git-scm.com/) to download an installer.

There are a lot of reasons why using `git` for your projects makes sense. If you plan to deploy to Heroku, you have one more, because to deploy to Heroku, your application must be in a `git` repository. If you are going to do a test deployment for Microblog, you can clone the application from GitHub:

```
$ git clone https://github.com/miguelgrinberg/microblog
$ cd microblog
$ git checkout v0.18
```

The `git checkout` command selects the specific commit that has the application at the point in its history that corresponds to this chapter.

If you prefer to work with your own code instead of mine, you can transform your own project into a `git` repository by running `git init .` on the top-level directory (note the period after `init`, which tells git that you want to create the repository in the current directory).

Creating a Heroku Application

To register a new application with Heroku, you use the `apps:create` command from the root directory of the application, passing the application name as the only argument:

```
$ heroku apps:create flask-microblog
Creating flask-microblog... done
http://flask-microblog.herokuapp.com/ | https://git.heroku.com/flask-microblog.git
```

Heroku requires that applications have a unique name. The name `flask-microblog` that I used above is not going to be available to you because I'm using it, so you will need to pick a different name for your deployment.

The output of this command will include the URL that Heroku assigned to the application, and also its git repository. Your local git repository will be configured with an extra *remote*, called `heroku`. You can verify that it exists with the `git remote` command:

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

Depending on how you created your git repository, the output of the above command could also include another remote called `origin`.

The Ephemeral File System

The Heroku platform is different to other deployment platforms in that it features an *ephemeral* file system that runs on a virtualized platform. What does that mean? It means that at any time, Heroku can reset the virtual server on which your server runs back to a clean state. You cannot assume that any data that you save to the file system will persist, and in fact, Heroku recycles servers very often.

Working under these conditions introduces some problems for my application, which uses a few files:

- The default SQLite database engine writes data in a disk file
- Logs for the application are also written to the file system
- The compiled language translation repositories are also written to local files

The following sections will address these three areas.

Working with a Heroku Postgres Database

To address the first problem, I'm going to switch to a different database engine. In Chapter 17 (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux) you saw me use a MySQL database to add robustness to the Ubuntu deployment. Heroku has a database offering of its own, based on the Postgres database, so I'm going to switch to that to avoid the file-based SQLite.

Databases for Heroku applications are provisioned with the same Heroku CLI. In this case I'm going to create a database on the free tier:

```
$ heroku addons:add heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on flask-microblog... free
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-parallel-56076 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

The URL for the newly created database is stored in a `DATABASE_URL` environment variable that will be available when the application runs. This is very convenient, because the application already looks for the database URL in that variable.

Logging to stdout

Heroku expects applications to log directly to `stdout`. Anything the application prints to the standard output is saved and returned when you use the `heroku logs` command. So I'm going to add a configuration variable that indicates if I need to log to `stdout` or to a file like I've been doing. Here is the change in the configuration:

config.py: Option to log to stdout.

```
class Config(object):
    # ...
    LOG_TO_STDOUT = os.environ.get('LOG_TO_STDOUT')
```

Then in the application factory function I can check this configuration to know how to configure the application's logger:

app/__init__.py: Log to stdout or file.

```
def create_app(config_class=Config):
    # ...
    if not app.debug and not app.testing:
        # ...

        if app.config['LOG_TO_STDOUT']:
            stream_handler = logging.StreamHandler()
            stream_handler.setLevel(logging.INFO)
            app.logger.addHandler(stream_handler)
        else:
            if not os.path.exists('logs'):
                os.mkdir('logs')
            file_handler = RotatingFileHandler('logs/microblog.log',
                                               maxBytes=10240, backupCount=10)
            file_handler.setFormatter(logging.Formatter(
                '%(asctime)s %(levelname)s: %(message)s '
                '[in %(pathname)s:%(lineno)d]'))
            file_handler.setLevel(logging.INFO)
            app.logger.addHandler(file_handler)

        app.logger.setLevel(logging.INFO)
        app.logger.info('Microblog startup')

    return app
```

So now I need to set the `LOG_TO_STDOUT` environment variable when the application runs in Heroku, but not in other configurations. The Heroku CLI makes this easy, as it provides an option to set environment variables to be used at runtime:

```
$ heroku config:set LOG_TO_STDOUT=1
Setting LOG_TO_STDOUT and restarting flask-microblog... done, v4
LOG_TO_STDOUT: 1
```

Compiled Translations

The third aspect of Microblog that relies on local files is the compiled language translation files. The more direct option to ensure those files never disappear from the ephemeral file system is to add the compiled language files to the git repository, so that they become part of the initial state of the application once it is deployed to Heroku.

A more elegant option, in my opinion, is to include the `flask translate compile` command in the start up command given to Heroku, so that any time the server is restarted those files are compiled again. I'm going to go with this option, since I know that my start up procedure is going to require more than one command anyway, since I also need to run the database migrations. So for now, I will set this problem aside, and will revisit it later when I write the *Procfile*.

Elasticsearch Hosting

Elasticsearch is one of the many services that can be added to a Heroku project, but unlike Postgres, this is not a service provided by Heroku, but by third parties that partner with Heroku to provide add-ons. At the time I'm writing this, there are three different providers of an integrated Elasticsearch service.

Before you configure Elasticsearch, be aware that Heroku requires your account to have a credit card on file before any third party add-on is installed, even if you stay within their free tiers. If you prefer not to provide your credit card to Heroku, then skip this section. You will still be able to deploy the application, but the search functionality is not going to work.

Out of the Elasticsearch options that are available as add-ons, I decided to try SearchBox (https://elements.heroku.com/addons/searchbox), which comes with a free starter plan. To add SearchBox to your account, you have to run the following command while being logged in to Heroku:

```
$ heroku addons:create searchbox:starter
```

This command will deploy an Elasticsearch service and leave the connection URL for the service in a `SEARCHBOX_URL` environment variable associated with your application. Once more keep in mind that this command will fail unless you add your credit card to your Heroku account.

If you recall from Chapter 16 (/post/the-flask-mega-tutorial-part-xvi-full-text-search), my application looks for the Elasticsearch connection URL in the `ELASTICSEARCH_URL` variable, so I need to add this variable and set it to the connection URL assigned by SearchBox:

```
$ heroku config:get SEARCHBOX_URL
<your-elasticsearch-url>
$ heroku config:set ELASTICSEARCH_URL=<your-elasticsearch-url>
```

Here I first asked Heroku to print the value of `SEARCHBOX_URL`, and then I added a new environment variable with the name `ELASTICSEARCH_URL` set to that same value.

Updates to Requirements

Heroku expects the dependencies to be in the *requirements.txt* file, exactly like I defined it in Chapter 15 (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure). But for the application to run on Heroku I need to add two new dependencies to this file.

Heroku does not provide a web server of its own. Instead, it expects the application to start its own web server on the port number given in the environment variable `$PORT`. Since the Flask development web server is not robust enough to use for production, I'm going to use gunicorn (http://gunicorn.org/) again, the server recommended by Heroku for Python applications.

The application will also be connecting to a Postgres database, and for that SQLAlchemy requires the `psycopg2` package to be installed.

Both `gunicorn` and `psycopg2` need to be added to the *requirements.txt* file.

The Procfile

Heroku needs to know how to execute the application, and for that it uses a file named *Procfile* in the root directory of the application. The format of this file is simple, each line includes a process name, a colon, and then the command that starts the process. The most common type of application that runs on Heroku is a web application, and for this type of application the process name should be `web`. Below you can see a *Procfile* for Microblog:

Procfile: Heroku Procfile.

```
web: flask db upgrade; flask translate compile; gunicorn microblog:app
```

Here I defined the command to start the web application as three commands in sequence. First I run a database migration upgrade, then I compile the language translations, and finally I start the server.

Because the first two sub-commands are based on the `flask` command, I need to add the `FLASK_APP` environment variable:

```
$ heroku config:set FLASK_APP=microblog.py
Setting FLASK_APP and restarting flask-microblog... done, v4
FLASK_APP: microblog.py
```

The application also relies on other environment varialbes, such as those that configure the email server or the token for the live translations. Those need to be added with addition `heroku config:set` commands.

The `gunicorn` command is simpler than what I used for the Ubuntu deployment, because this server has a very good integration with the Heroku environment. For example, the `$PORT` environment variable is honored by default, and instead of using the `-w` option to set the number of workers, heroku recommends adding a variable called `WEB_CONCURRENCY`, which `gunicorn` uses when `-w` is not provided, giving you the flexibility to control the number of workers without having to modify the Procfile.

Deploying the Application

All the preparatory steps are complete, so now it is time to run the deployment. To upload the application to Heroku's servers for deployment, the `git push` command is used. This is similar to how you push changes in your local git repository to GitHub or other remote git server.

And now I have reached the most interesting part, where I push the application to our Heroku hosting account. This is actually pretty simple, I just have to use `git` to push the application to the master branch of the Heroku git repository. There are a couple of variations on how to do this, depending on how you created your git repository. If you are using my `v0.18` code, then you need to create a branch based on this tag, and push it as the remote master branch, as follows:

```
$ git checkout -b deploy
$ git push heroku deploy:master
```

If instead, you are working with your own repository, then your code is already in a `master` branch, so you first need to make sure that your changes are committed:

```
$ git commit -a -m "heroku deployment changes"
```

And then you can run the following to start the deployment:

```
$ git push heroku master
```

Regardless of how you push the branch, you should see the following output from Heroku:

```
$ git push heroku deploy:master
Counting objects: 247, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (238/238), done.
Writing objects: 100% (247/247), 53.26 KiB | 3.80 MiB/s, done.
Total 247 (delta 136), reused 3 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing python-3.6.2
remote: -----> Installing pip
remote: -----> Installing requirements with pip
...
remote:
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 57M
remote: -----> Launching...
remote:        Released v5
remote:        https://flask-microblog.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/flask-microblog.git
 * [new branch]      deploy -> master
```

The label `heroku` that we used in the `git push` command is the remote that was automatically added by the Heroku CLI when the application was created. The `deploy:master` argument means that I'm pushing the code from the local repository referenced by the `deploy` branch to the `master` branch on the Heroku repository. When you work with your own projects, you will likely be pushing with the command `git push heroku master`, which pushes your local `master` branch. Because of the way this project is structured, I'm pushing a branch that is not `master`, but the destination branch on the Heroku side always needs to be `master` as that is the only branch that Heroku accepts for deployment.

And that is it, the application should now be deployed at the URL that you were given in the output of the command that created the application. In my case, the URL was *https://flask-microblog.herokuapp.com*, so that is what I need to type to access the application.

If you want to see the log entries for the running application, use the `heroku logs` command. This can be useful if for any reason the application fails to start. If there were any errors, those will be in the logs.

Deploying Application Updates

To deploy a new version of the application, you just need to run a new `git push` command with the new code. This will repeat the deployment process, take the old deployment offline, and then replace it with the new code. The commands in the Procfile will run again as part of the new deployment, so any new database migrations or translations will be updated during the process.