CODING NOMADS

19) Deploying Your App On The Internet Lesson

Introduction to Flask Deployment

24 min to complete · By Brandon Gigous

Contents

If you've been following along with this course, you've learned a ton about Flask, way to go! :) Now you want to be able to show off your hard work and put your webapp on the internet. For that you will need to learn about **deployment**. And as you might have already gleaned: *deployment can be hard*.

But that's okay, and of course it's manageable and gets easier with training. Coming up, you'll walk through two different deployment options, and you will get a good overview of what options are out there. After that it comes again down to: *training, training, training*.

Keep going even if something doesn't work quite right the first time. When deploying your webapp, there are many different cogwheels that need to fit together *just right* for everything to function. Remember: the internet is a mess and you're about to leave your safe, local development environment in order to venture out into the world wild web. Out here, we're all greenhorns forever, but with practice you'll get better at being a greenhorn. :)

So get started. Ready... set... DEPLOY!

First off, the engine of your website: the server. Let's get started by learning about what a server is.

What is a Server?

A server is a computer or software system that provides services, resources, and/or data to other computers over a network, known as "clients". Servers perform a number of functions, including hosting websites, managing databases, enabling file sharing, and email services. They operate continuously to manage and respond to multiple client requests simultaneously, ensuring efficient communication and resource allocation in a networked environment.

When people talk about a Web Server, it usually refers to one or both of two things:

- **Hardware**

- **Software**

Let's explain each of those in a little more detail.

Server Hardware

Server *hardware* relates to a physical computer that is set up somewhere in the world and connected to a *network*. While this could be any network, what people usually mean when they talk about web servers as computers, are those connected to the *internet*. While you could set up a computer at your home as a server, industrially, servers are usually located in large server farms these days.

Photo by İsmail Enes Ayhan on Unsplash

It helps, however, to keep in mind that these are nothing but specialized computers and that even your laptop could be used as web server hardware. Only industrial web servers are usually far, far more powerful than consumer grade electronics.

To get your Flask app to be accessible via the internet, you will need to run it on a web server that is *always on*, so that users can access it at any time.

Server Software

When talking about web servers, it can also mean *software* that is specifically designed to handle web requests, which usually arrive as HTTP requests.

Info: When you read about a "web server" moving in forward in this course, it will refer to web server *software* rather than the physical machines.

Web Server vs. Web Framework

A web *server* is typically thought of as different from a web *framework*, such as Flask, although some may say that the line is blurry. Let's look at the difference and define how the terms are used in this course:

- **Web Server**: A web server listens for web connections and either serves static content from files, or passes requests onto other programs or processes.
- **Web Framework**: A web framework gives you scaffolding for writing your own custom logic in such a way that it can work together with a web server to serve your content.

Most frameworks handle common details such as HTTP header parsing, URL routing, templating, database interactions via an ORM, etc. They act like glue between server, database, other services, and your custom code.

In practical terms, you only need to *define some settings* for your web server to handle your app's communication with the rest of the internet.

Web Servers Examples

Some of the best web servers that are popular today are:

- Apache
- Nginx
- Gunicorn

In this section, you will learn to interact with `nginx` and `gunicorn`. Apache is another common web server that could be used for the same objective that you'll be using Nginx for.

You learned that while you've been using Flask to develop your apps, you will *also* need to run a web server to serve it on the internet. How will these two programs communicate with each other? Let's find out.

What is WSGI?

In the multi-language-tech environment that is the internet, your Flask app needs a clear language to communicate with the servers it will run on.

Years back, the Python community developed a protocol for communication between Python programs and web servers called **WSGI**, which stands for **"Web Server Gateway Interface"**. The WSGI standard defines clear rules on how a Python webapp can interact with a web server, such as Nginx, Apache, or Gunicorn.

> **Note**: A ***protocol*** is a standard language or specification for communication between different programs, languages, or systems.
>
> For example, TCP is a protocol that allows for communication between different computers on the internet. HTTP is a protocol that allows for communication between web servers and web clients.

Most Python web frameworks implement the WSGI standard, and it has good support for all major web server software. The key concept of deploying with WSGI is an `application` callable which the web server uses to communicate with your Python code. It's commonly provided as an object named `application` in a Python module accessible to the web server.

If you're using a UNIX-based system like Linux or MacOS, for your Flask project you'll use **Gunicorn** as the WSGI server. If you're on Windows, you'll instead need to use uWSGI. Either way, your Flask app, through the *application instance*, will receive any requests from this server and then perform all the application specific stuff it does. You know, the `Flask` object, just like you've been working with all this time! But we'll get to those details later.

For now, that's really all you need to know. If you want to read more, check out Flask's docs on WSGI, but keep in mind that for now a very high-level understanding of WSGI is sufficient.

Python Flask Production Server

So you've come this far. Why all this talk about "production" servers? Why should you not simply stick with using `flask run` also in production? It seems to have worked fine for you so far!

Hang on a second, haven't you seen the warnings in your console whenever you've launched your app?

```
WARNING: This is a development server.
Do not use it in a production deployment.
Use a production WSGI server instead

* Restarting with stat
* Debugger is active!
* Debugger PIN: 123-456-789
* Running on http://127.0.0.1:5000/
```

Flask's Development server is just that: a lightweight *development* server. Flask includes a lightweight server for development purposes, but it's not suitable for production. It's not designed to be particularly fast, stable, or secure. It's not designed to handle a lot of traffic, and it's not designed to be exposed to the internet and handle all the security risks that come with that.

Why does Flask not just include a production server? Well, if Flask was to include a production server, it would then be responsible for maintaining and updating that server. This would be a huge task, and it would take away from the Flask team's ability to focus on the core of Flask. This is a typical decision for many web frameworks, such as Django or Spring. They all come with a development server, but leave the difficult decisions regarding serving stuff over the web to the professionals in that area, such as Nginx.

Throughout this module you will get to know a couple of different solutions for serving your Flask app in production in a safe and stable way. If you're curious in learning more about the complexities of deployment, check out this great talk by Katie McLaughlin: What is deployment, anyway?

Coming up, you'll learn about two different ways of deploying your Flask webapp. They differ significantly in how much effort they require to get up and running, and connected with that, how much opportunity you have to customize the deployment yourself.

Mentorship Makes the Difference!

You don't have to learn alone. Join the CodingNomads' Mentorship Bootcamp Program and get:

- A team of mentors and advisors dedicated to your success
- Weekly 1-on-1 screen share meetings with your professional mentor
- Constant, personal, in-depth support 7 days a week via Discord
- Accountability, feedback, encouragement, and success

Get Mentorship

PaaS vs IaaS

There are two main ways to deploy your applications whether you are using Flask or any other web framework. There are:

- **PaaS** (Platform-as-a-Service) - worry only about your app and let them deal with the rest, but you give up some control. Some popular PaaS providers are Heroku, Google App Engine, and AWS Elastic Beanstalk.
- **IaaS** (Infrastructure-as-a-Service) - you have more control, but also more responsibility to manage servers and infrastructure. Some popular IaaS providers are AWS, Google Cloud, and Azure.

You'll note that IaaS providers often provide PaaS services as well running on their IaaS!

You can also of course set up your own hardware server, but that's a lot of work and not recommended for most situations.

In this course you will focus on PaaS to keep things simple.

PaaS Deployment

Using a Platform-as-a-Service is the recommended and easier way to get started deploying your webapp. You will use **Heroku** as an example of a PaaS. With a PaaS you can approach deployment on a higher level of abstraction. This means there is less that you need to take care of by yourself. Your provider manages for you:

- Hardware
- Storage
- Networking
- Operating System
- Web Server
- Database

Essentially, all that you need to worry about is your **webapp** and your **data**. Using a PaaS gives you the convenience to focus on what is probably your main interest: your app. Difficult decisions, time-consuming setup processes, as well as maintenance tasks are automatically taken care of for you. This comes at a loss of flexibility and offers a lower level of customization, as well as often a higher price tag.

Heroku, the cloud PaaS provider you will be using in the chapter on PaaS deployment, offers a free hobby tier that is sufficient for your apps, so no cost will be incurred. Many of these services allow you to start for free and decide to *scale up* into a tier that has fees, if you choose that you need it.

Now that you know what kind of environment you'll be deploying your app to, what do you need to do to get your Flask app ready for deployment? Let's find out.

Python Flask Production Environment

As you have seen, deployment doesn't have a strict rule that will be applicable across all your deployments. As any app will be unique, your deployment choices will vary. Still, there are certain tools and scenarios that will help you prepare on a general level. There are a few topics to always keep in mind:

1. Do Necessary Flask Production Adjustments
2. Use A Production Web Server
3. Collect Your Static Files
4. Change To A Production Database

A PaaS such as Heroku will mostly handle Step 2-4 for you. However, there are some changes that you need to do to your Flask codebase to prepare your project for *any* kind of deployment.

Flask Production Settings

Flask provides some production guidelines in their tutorial for a simple webapp. It mentions a few settings that are important to update to safely run your webapp in production. In particular for your Flask app, you'd want to update these environment variables:

- `SECRET_KEY`: It's used for cryptographic signing, and needs to be a unique, unpredictable value. For security, you should replace the default value used in development and keep it out of your app's code base.

- `FLASK_DEBUG` : This setting allows you to see all your error message friends during development. Showing them in production, however, represents a security risk.
- `FLASK_CONFIG` : In development, you've been using your `development` configuration. In production, you will definitely want to change this to at least `production` . But, in fact, you will probably need to tweak a few more things depending on which service you choose to deploy on, so you might need to make a new configuration based on the `production` configuration.
- `SQLALCHEMY_DATABASE_URI` : Here you can change what database Flask connects to. By default this is a SQLite file, which is appropriate and handy for development, but needs to be changed for something more robust in production.

Info: You will learn more about each of these settings later in the course when you will need to change them. This is a quick overview that you will expand on, so don't worry if not everything makes sense yet.

Some of the changes necessary can be automated, and in fact Heroku will handle a few of them for you. With this overview in mind, let's start exploring Heroku. In the upcoming chapter you will get to know this popular PaaS and learn how to deploy your Flask app on it.