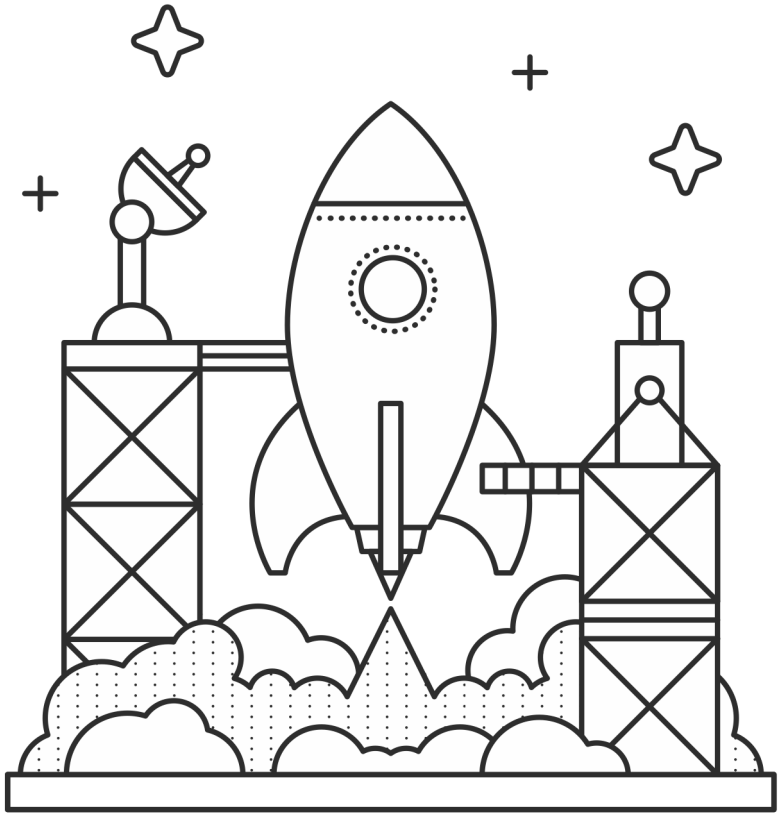
Deployment



We're finally ready to show our app to the world. It's time to deploy. This process can be a pain because there are so many moving parts. There are a lot of choices to make when it comes to our production stack as well. In this chapter, we're going to talk about some of the important pieces and some of the options we have with each.

The Host

We're going to need a server somewhere. There are thousands of providers out there, but these are the three that I personally recommend. I'm not going to go over the details of how to get started with them, because that's out of the scope of this book. Instead I'll talk about their benefits with regards to hosting Flask applications.

Amazon Web Services EC2

Amazon Web Services is a collection of services provided by ... Amazon! There's a good chance that you've heard of them before as they're probably the most popular choice for new startups these days. The AWS service that we're most concerned with here is EC2, or Elastic Compute Cloud. The big selling point of EC2 is that we get virtual servers - or **instances** as they're called in AWS parlance - that spin up in seconds. If we need to scale our app quickly it's just a matter of spinning up a few more EC2 instances for our app and sticking them behind a load balancer (we can even use the AWS Elastic Load Balancer).

With regards to Flask, AWS is just a regular old virtual server. We can spin it up with our favorite linux distro and install our Flask app and our server stack without much overhead. It means that we're going to need a certain amount of systems administration knowledge though.

Heroku

Heroku is an application hosting service that is built on top of AWS services like EC2. They let us take advantage of the convenience of EC2 without the requisite systems administration experience.

With Heroku, we deploy our application with a `git push` to their server. This is really convenient when we don't want to spend our time SSHing into a server, installing and configuring software and coming up with a sane deployment procedure. This convenience comes at a price of course, though both AWS and Heroku offer a certain amount of free service.

> ***Note***
>
> Heroku has a tutorial on deploying Flask with their service.

> ***Note***
>
> Administrating your own databases can be time consuming and doing it well requires some experience. It's great to learn about database administration by doing it yourself for your side projects, but sometimes you'd like to save time and effort by outsourcing that part to professionals.
>
> Both Heroku and AWS have database management offerings. I don't have personal experience with either yet, but I've heard great things. It's worth considering if you want to make sure your data is being secured and backed-up without having to do it yourself.
>
> - Heroku Postgres
> - Amazon RDS

Digital Ocean

Digital Ocean is an EC2 competitor that has recently begun to take off. Like EC2, Digital Ocean lets us spin up virtual servers - now called **droplets** - quickly. All droplets run on SSDs, which isn't something we get at the lower levels of EC2. The biggest selling point for me personally is an interface that is far simpler and easier to use than the AWS control panel. Digital Ocean is my preference for hosting and I recommend that you take a look at them.

The Flask deployment experience on Digital Ocean is roughly the same as on EC2. We're starting with a clean linux distribution and installing our server stack from there.

> ***Note***
>
> Digital Ocean was nice enough to make a contribution to the Kickstarter campaign for *Explore Flask*. With that said, I promise that my recommendation comes from my own experience as a user. If I didn't like them, I wouldn't have asked them to pledge in the first place.

The stack

This section will cover some of the software that we'll need to install on our server to serve our Flask application to the world. The basic stack is a front server that reverse proxies requests to an application runner that is running our Flask app. We'll usually have a database too, so we'll talk a little about those options as well.

Application runner

The server that we use to run Flask locally when we're developing our application isn't good at handling real requests. When we're actually serving our application to the public, we want to run it with an application runner like Gunicorn. Gunicorn handles requests and takes care of complicated things like threading.

To use Gunicorn, we install the `gunicorn` package in our virtual environment with Pip. Running our app is a simple command away.

```
# app.py

from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
        return "Hello World!"
```

A fine app indeed. Now, to serve it up with Gunicorn, we simply run the `gunicorn` command.

```
(ourapp)$ gunicorn rocket:app
2014-03-19 16:28:54 [62924] [INFO] Starting gunicorn 18.0
2014-03-19 16:28:54 [62924] [INFO] Listening at: http://127.0.0.1:8000 (62924)
2014-03-19 16:28:54 [62924] [INFO] Using worker: sync
2014-03-19 16:28:54 [62927] [INFO] Booting worker with pid: 62927
```

At this point, we should see "Hello World!" when we navigate our browser to *http://127.0.0.1:8000*.

To run this server in the background (i.e. daemonize it), we can pass the `-D` option to Gunicorn. That way it'll run even after we close our current terminal session.

If we daemonize Gunicorn, we might have a hard time finding the process to close later when we want to stop the server. We can tell Gunicorn to stick the process ID in a file so that we can stop or restart it later without searching through lists of running processess. We use the `-p <file>` option to do that.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -D
(ourapp)$ cat rocket.pid
63101
```

To restart and kill the server, we can run `kill -HUP` and `kill` respectively.

```
(ourapp)$ kill -HUP `cat rocket.pid`
(ourapp)$ kill `cat rocket.pid`
```

By default Gunicorn runs on port 8000. We can change the port by adding the `-b` bind option.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -b 127.0.0.1:7999 -D
```

Making Gunicorn public

> ***Warning***
>
> Gunicorn is meant to sit behind a reverse proxy. If you tell it to listen to requests coming in from the public, it makes an easy target for denial of service attacks. It's just not meant to handle those kinds of requests. Only allow outside connections for debugging purposes and make sure to switch it back to only allowing internal connections when you're done.

If we run Gunicorn like we have in the listings, we won't be able to access it from our local system. That's because Gunicorn binds to 127.0.0.1 by default. This means that it will only listen to connections coming from the server itself. This is the behavior that we want when we have a reverse proxy server that is sitting between the public and our Gunicorn server. If, however, we need to make requests from outside of the server for debugging purposes, we can tell Gunicorn to bind to 0.0.0.0. This tells it to listen for all requests.

```
(ourapp)$ gunicorn rocket:app -p rocket.pid -b 0.0.0.0:8000 -D
```

> ***Note***
>
> - Read more about running and deploying Gunicorn in the documentation.
> - Fabric is a tool that lets you run all of these deployment and management commands from the comfort of your local machine without SSHing into every server.

Nginx Reverse Proxy

A reverse proxy handles public HTTP requests, sends them back to Gunicorn and gives the response back to the requesting client. Nginx can be used very effectively as a reverse proxy and Gunicorn "strongly advises" that we use it.

To configure Nginx as a reverse proxy to a Gunicorn server running on 127.0.0.1:8000, we can create a file for our app: */etc/nginx/sites-available/expl-oreflask.com*.

```
# /etc/nginx/sites-available/exploreflask.com

# Redirect www.exploreflask.com to exploreflask.com
server {
        server_name www.exploreflask.com;
        rewrite ^ http://exploreflask.com/ permanent;
}

# Handle requests to exploreflask.com on port 80
server {
        listen 80;
        server_name exploreflask.com;

                # Handle all locations
        location / {
```

```
                        # Pass the request to Gunicorn
                proxy_pass http://127.0.0.1:8000;

                # Set some HTTP headers so that our app knows where the
                # request really came from
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
}
```

Now we'll create a symlink to this file at */etc/nginx/sites-enabled* and restart Nginx.

```
$ sudo ln -s \
/etc/nginx/sites-available/exploreflask.com \
/etc/nginx/sites-enabled/exploreflask.com
```

We should now be able to make our requests to Nginx and receive the response from our app.

> ***Note***
>
> The Nginx configuration section in the Gunicorn docs will give you more information about setting Nginx up for this purpose.

ProxyFix

We may run into some issues with Flask not properly handling the proxied requests. It has to do with those headers we set in the Nginx configuration. We can use the Werkzeug ProxyFix to ... fix the proxy.

```
# app.py

from flask import Flask

# Import the fixer
from werkzeug.contrib.fixers import ProxyFix

app = Flask(__name__)

# Use the fixer
app.wsgi_app = ProxyFix(app.wsgi_app)

@app.route('/')
def index():
        return "Hello World!"
```

> ***Note***
>
> - Read more about ProxyFix in the Werkzeug docs.

Summary

- Three good choices for hosting Flask apps are AWS EC2, Heroku and Digital Ocean.
- The basic deployment stack for a Flask application consists of the app, an application runner like Gunicorn and a reverse proxy like Nginx.
- Gunicorn should sit behind Nginx and listen on 127.0.0.1 (internal requests) not 0.0.0.0 (external requests).
- Use Werkzeug's ProxyFix to handle the appropriate proxy headers in your Flask application.