Deploy web apps with Docker

Rescue yourself from the **complexity** of **DevOps**

Nick Janetakis

Deploy web apps with Docker

Rescue yourself from the complexity of DevOps

Nick Janetakis

This book is for sale at http://leanpub.com/deploy-web-apps-with-docker

This version was published on 2015-09-15

Tweet This Book!

Hi,

My name is Nick Janetakis and I'm the author of this book. I just wanted to personally say thanks a lot for investing in ***Deploy web apps with Docker****.*

One of the main reasons that this book exists is because a related project on how to build and deploy a Flask application and deploy it with Docker was successfully funded on Kickstarter.

Thanks to everyone who backed the project and thank YOU for showing your support.

Spoiler alert*: You will be able to apply all of the concepts of this book to any programming language and framework.*

Contents

Getting to know Docker and CoreOS

What is Docker and why should you use it?

Docker's slogan is: build, ship and run any app, anywhere. Docker is simply a way to isolate a service from the rest of your system. It's similar to a virtual machine in the sense that, it allows you to run something in isolation from the main host operating system.

Instead of setting up an entirely separate operating system to run the service, it leverages the concept of containers. You only need to understand that Docker runs services in isolation at the operating system level without using virtualization.

It does this through libcontainer, cgroups and a bunch of other lower level Linux kernel components but all of that is abstracted away from you through the Docker API.

Docker containers are extremely light weight

Containers are launched from Docker images. If you are a developer you can think of the image as a class while the container is the result of instantiating that class.

You can launch containers in a matter of milliseconds and the disk space used is tied into the image, not the container. This means that running 10 copies of the same image will only use the disk space of 1 image.

Docker uses a layered file system

It can intelligently determine the difference between 2 versions of an image. If you make a tiny app change and rebuild your image, it will get rebuilt in a matter of seconds because it can pick out and update only the bits that change.

This layered file system is awesome for deployment too. Instead of having to copy over a 300-400MB image every time you deploy, you may end up only transferring a few megabytes.

Docker has an excellent CLI tool set

Docker officially has 1 CLI tool to manage images and containers but there are a number of other tools that aren't quite built into the core Docker CLI, however are actively developed by the Docker team.

There are tools to help provision new machines, link multiple servers together seamlessly and give you a way to declaratively define which containers you want to run for a specific project.

Docker isn't just for production systems

To wrap up the introduction, let's touch on why it's really important to match your development environment to production and how we'll use Docker to do this.

You've probably heard the infamous line, "well, it works on my machine" at least a few times. You may have even encountered issues where your app worked great on your machine, but as soon as you tried to deploy it, it all went down hill.

Docker can help us eliminate this problem.

For example, a Docker image that contains Redis has its own base operating system on the inside. Let's just say for the sake of argument that it's running Debian. Debian is one of the most popular Linux distros that's well known for its stability.

If you run a Redis container on your Ubuntu workstation at home, or through boot2docker on your Mac, Redis is still using the Debian base on the inside. If we deploy our application using CoreOS or through another distribution of Linux it will continue to use its own Debian base.

This is a really powerful thing to have at our disposal. We can get Redis running locally on our workstation without polluting our main OS, and it's guaranteed to work in production no matter what the main OS is.

What is CoreOS and why should you use it?

CoreOS is designed to be a minimal, scalable, consistent, secure and reliable operating system to host containers. It's very different to other Linux distros such as Debian or CentOS.

One of the main differences is that it doesn't even come with a package manager installed. It is expected that you run containers which in turn have a service running inside of them.

The great thing about this philosophy is that if your service is happily running in a container on a specific CoreOS instance, then you can be certain that it will also run on a different instance.

CoreOS runs on nearly every platform such as VMWare, QEMU/KVM and VirtualBox. A lot of cloud providers such as Digital Ocean and Amazon EC2 have pre-built images to use CoreOS. Of course you could use your own hardware too.

CoreOS makes it easy to spin up a new machine that has the tools necessary to create a scalable system. It uses a combination of Docker, etcd and systemd to orchestrate all of this.

CoreOS works well with 1 or 100+ machines

If you have a small application where fault tolerance is not of the utmost importance then you can easily run CoreOS by itself on 1 machine. On the flip side, since it has excellent clustering abilities built in through the support of various tools, CoreOS makes it very easy to run dozens of instances together and act on them as a single cluster.

What are we going to build?

Overview of the technology stack

It will be a simple web application that keeps track of how many times you visited the website.

It will be an extremely basic **Flask** application that leverages **Redis**. Flask is a minimal web framework built with Python but don't worry, you don't need to know Python to understand what we're building. The importance isn't the language or framework that we select for the app.

I picked Flask because just about every system comes with Python, and Flask has close to no boilerplate. Our app will be a simple 1 file app and the concepts we work through will be applicable to whatever language and framework you use.

We'll also use **nginx** to act as a front end web server for our app. It will be in control of serving assets, handling SSL and proxying requests to our Flask app.

Once all of that is in place we'll sign up with Digital Ocean to host our app, then register a domain name and sign up for a free SSL certificate so we can serve that application over HTTPS. We'll also configure everything so that the site receives an A+ grade SSL security rating.

What else will we cover?

Along the way we'll set up a proper development environment to work with Docker on Linux, Mac and Windows. We'll also set up a local staging server with Vagrant so we can test CoreOS without spinning up a Digital Ocean droplet. We'll also have a crash course with systemd and more.

Example web application architecture

If you had a non-trivial workload and wanted to set up the groundwork for a scalable and redundant system you might consider the following architecture:

- 1x Front end / load balancer *(Ex. nginx)*
- 2x Web applications *(Ex. Flask, Ruby on Rails, Express, etc.)*
- 1x Master write database *(Ex. PostgreSQL)*
- 1x Slave read database *(Ex. PostgreSQL)*
- 1x Master write cache *(Ex. Redis)*
- 1x Slave read cache *(Ex. Redis)*
- 1x Build / continuous integration *(Ex. Jenkins)*
- 1x Docker registry *(Ex. Official Docker registry image)*

You could spread everything across a few Digital Ocean droplets and now you're ready to rock but at the same time you could decide to host it all on 1 powerful droplet.

Of course you would lose the fault tolerance if everything was on 1 droplet but the beauty of all of this is that you can design your system to work so that it runs in both scenarios with little changes.

What if you want to dial it down?

Maybe you only need 1 web server, 1 app server, a database server and cache server while leveraging third party services for continuous integration. That's completely reasonable and smart to do early on during the life cycle of developing an application.

Depending on what technology your app server is using, you could easily fit all of that onto a single $10/month Digital Ocean droplet. It's not possible to speculate the amount of traffic you could sustain because it depends on what the app does, but I have a number of Flask apps in production with this set up and they happily serve hundreds of requests a day, and some cases thousands. They are rock solid and I trust them.

There are other alternatives like Heroku which handle scaling for you but you can be sure that you'll wind up paying 10x or more per month for a decently sized application and the costs will climb severely uphill as your technology stack gets more complicated.

This book will cover the $5/month solution

We'll set everything up to run on a single Digital Ocean $5/month droplet. You will be able to practice along for free because you were given $10 in free credit with the purchase of this book.

Get your workstation running Docker

Are you using Windows, Mac or Linux?

Feel free to skip to whatever OS you're running on your workstation.

Windows users

Docker does not run on Windows natively but you can get it running inside of a virtual machine. The 2 contending pieces of software to run a virtual machine are VMWare and VirtualBox.

You will end up running a specific version of Linux in a virtual machine. It will not interfere with your Windows installation and it's something you can turn on/off on demand.

VMWare Player is free for Windows and is closed source. It's what I run personally and actively develop with it nearly every day. I spend 99% of the time inside of a graphical xubuntu 14.04 virtual machine with a Windows 8.1 host.

The other option is VirtualBox which is open source but tends to be less stable in my testing. However, it is certainly worth checking out.

Download and install VMWare or VirtualBox

You will need to grab one or the other. Below are links to download either one straight from the vendor. Software tends to move quickly, so it's worth looking for a newer version of the below packages.

You only need to install one of them, not both!

Download VMWare Player 7.0 for Windows:
https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/7_0[1]

Download VirtualBox 5.0 for Windows:
http://download.virtualbox.org/virtualbox/5.0.0/VirtualBox-5.0.0-101573-Win.exe[2]

[1]https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/7_0
[2]http://download.virtualbox.org/virtualbox/5.0.0/VirtualBox-5.0.0-101573-Win.exe

Download xubuntu 14.04

You can use whatever distro of Linux you want as your main VM OS. If you plan to use it graphically on your workstation you should check out xubuntu. It's a lightweight version of Ubuntu.

Download xubuntu 14.04.x LTS:
http://torrent.ubuntu.com/xubuntu/releases/trusty/release/desktop/[3]

Configure VMWare or VirtualBox

After you install either VMWare Player or VirtualBox and download a Linux distro you will need to create a new virtual machine within either VMWare Player or VirtualBox.

Before continuing you should also figure out what you want to use the virtual machine for because it will ask you to configure the machine specs of the VM. With both solutions you will be able to change the settings later.

If you plan to make it your main development VM you should try and give it about 60 gigs of space on an SSD. You'll also want to give it about 4-8 gigs of memory and 2-3 CPU cores. That will be enough power to run multiple graphical apps as well as Docker and more without issues.

If your workstation does not have that many resources to give out, don't sweat it. You could give the VM 768MB or 1GB of memory with about 20 gigs of HD space and 1 CPU core.

If all you want to do is run Docker through it, you can get by with far fewer resources allocated to the VM.

Install guest additions

Both VMWare Player and VirtualBox have a concept of "guest additions". They allow you to install additional software which allow you to share your clipboard and folders between the VMs and the host. They also unlock features such as Unity mode in VMWare Player and Seamless mode in VirtualBox.

Either of those modes allow you to blend both operating systems together in such a way that it feels like you're running both at once without constraints. It's awesome.

VMWare Tools for an Ubuntu virtual machine:
http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1022525[4]

VirtualBox guest addons for an Ubuntu virtual machine:
http://askubuntu.com/questions/22743/how-do-i-install-guest-additions-in-a-virtualbox-vm[5]

If for some reason one of the above links are no longer accessible then Google for the term in bold above the link. It's hard to predict the internet!

[3]http://torrent.ubuntu.com/xubuntu/releases/trusty/release/desktop/
[4]http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1022525
[5]http://askubuntu.com/questions/22743/how-do-i-install-guest-additions-in-a-virtualbox-vm

Sanity check

At this point you should have your virtualization software of choice installed with xubuntu or another Linux distro of your choosing. You should also have the guest additions installed to make your VM experience much smoother and user friendly, but it's ok if you skipped that step.

You should be able to access a Linux terminal. You can find the terminal inside of the `xubuntu start menu -> system -> xfce terminal`. Every command you see from this point on will be expected to be run from within the VM.

Install software on the VM

Install Docker

Install the latest version of Docker:

```
wget -qO- https://get.docker.com/ | sh
```

Allow us to run Docker without root:

```
sudo usermod -aG docker ${USER}
```

You must restart your virtual machine for the above command to take effect.

Install a few packages

Install curl and Python:

```
sudo apt-get update && sudo apt-get install curl python-dev
```

The above python package gives us the necessary libraries we need to run the example application. We install `curl` because it allows us to make HTTP requests, and we'll be using it later.

Install PIP:

```
curl https://bootstrap.pypa.io/get-pip.py \
> /tmp/get-pip.py sudo python /tmp/get-pip.py
```

PIP is a package manager for Python. It allows you to easily install Python packages onto your system without polluting your system's libraries.

You can now jump to the Install additional tools section.

Mac users

Docker does not natively run on a Mac. However there are a few tools you can use to run Docker on your Mac without too much pain. As of Docker 1.8 there is a new tool called the Docker Toolbox.

Follow the official Mac guide here to get started:
https://www.docker.com/toolbox[6]

You should follow it up until you run the Docker "hello world" example and should be familiar with the differences of running Docker natively and through the Docker Toolbox (covered in the above guide).

Sanity check

At this point you should have the Docker Toolbox installed. In the rest of this book I will always be referencing 'localhost' when it comes time to access the web app example. In your case you will need to use your `docker-machine` IP address instead, which is also covered in the above guide.

Additional software

You'll also want to run `brew install python` to get Python 2.7.x. This version of Python also includes PIP, a package manager for Python. It allows you to easily install Python packages onto our system without polluting your system's libraries.

Linux users

Excellent. You really don't have to do too much because Linux can run Docker natively. However you do need to install a few things.

[6]https://www.docker.com/toolbox

Install software onto your system

Install Docker

Install the latest version of Docker:

```
wget -qO- https://get.docker.com/ | sh
```

Allow us to run Docker without root:

```
sudo usermod -aG docker ${USER}
```

You must completely logout of your session for the above command to take effect.

Install a few packages

Install curl and Python:

```
sudo apt-get update && sudo apt-get install curl python-dev
```

The above python package gives us the necessary libraries we need to run the example application. The `curl` program is a command line utility to make HTTP requests, we will be using it later.

Install PIP:

```
curl https://bootstrap.pypa.io/get-pip.py \
> /tmp/get-pip.py sudo python /tmp/get-pip.py
```

PIP is a package manager for Python. It allows us to easily install Python packages onto our system without polluting the main operating system.

You can now jump to the Install additional tools section.

Install additional tools

Now that your system is prepared, we need to install 2 more tools. The first one is virtualenv and the other one is Docker compose.

Install virtualenv

It is a popular Python package that will let you isolate your projects from each other. It will also allow you to run all Python commands without `sudo`. You can install virtualenv with PIP by running the 2 commands below:

```
sudo pip install virtualenv
sudo pip install virtualenvwrapper
```

Once installed, append the following 2 lines to your `~/.bashrc` file:

```
export WORKON_HOME=$HOME/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

Attention Windows users: you can run this from a terminal to edit that file:
`mousepad ~/.bashrc`

Once that's in place, restart your terminal or simply run `source ~/.bashrc`.

Fix potential PIP issues

Before we install anything else, it is worth mentioning that PIP might give you permission errors in the future, unless you run the command below:

Fix potential PIP permission errors:
`sudo rm -rf ~/.cache/pip`

Install `docker-compose`

Docker commands get very tedious to run and in a real project you might have 5 or 6 different Docker services that need to be started up in development mode.

It would take a while to type them out by hand each time and it would be annoying to hack together a script to do it.

This is where `docker-compose` steps in. It allows you to define Docker container startup commands in YAML. Since it's a file you can even check it into version control. It makes it very easy for other developers to run your project too.

[Windows or Linux users] Download `docker-compose` 1.4.0 by running:

```
curl -L https://github.com/docker/compose/releases/download/1.4.0/docker-compose\
-Linux-x86_64 \
> /tmp/docker-compose
```

Put `docker-compose` on your system path by running:

```
chmod +x /tmp/docker-compose
sudo mv /tmp/docker-compose /usr/local/bin
```

Attention Mac users: you already have docker-compose because of Docker Toolbox!

Sanity check

At this point you should be able to run the following commands successfully:

```
pip --version
mkvirtualenv --version
docker-compose --version
```

Now that we have Docker installed along with `docker-compose` and the Python dependencies we can get rolling. I do want to point out that Python is not a requirement of Docker, it is only a requirement for our example application.

Create the demo application

Set up the project directory structure

Our web application is not the only thing we will be working with. We'll have deploy related scripts and more. It's good practice to separate these things out into their own folders.

Head over to wherever you save source code on your workstation and run:

```
mkdir -p rediscounter/website
```

The -p flag will automatically create sub-folders without errors.

Commands should always be run from the rediscounter path

It is important to note from this point on, all commands that involve creating a file, folder or editing a file is expected to be run from the `rediscounter` path on your workstation unless noted otherwise.

The sub-folders are already included into the command unless noted.

Set up the Flask application

The only Python libraries we'll be using are Flask and the Redis library. We use the Redis library to connect to a Redis server which we will be running in a Docker container.

Create the requirements.txt file to install Flask:

```
touch website/requirements.txt
```

Edit the requirements.txt file to look like this:

```
Flask==0.10.1
redis==2.10.3
```

Create a new virtualenv:

```
mkvirtualenv rediscounter
```

Install the packages from the requirements.txt file:

```
pip install -r website/requirements.txt
```

Create the Flask application file:

```
touch website/app.py
```

Edit the app.py file to look like this:

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000, debug=True)
```

Ensure you can run the Flask app:
`python website/app.py`

You should see an output that resembles this:

```
$ python website/app.py
 * Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
 * Restarting with stat
```

Try going to your browser at the above address. You should be greeted with a web page that says "Hello World!". If you can't get it to run then take a step back and make sure you executed all of the above commands in the proper directory, with an active virtualenv and in the correct order.

Future coding sessions

You only need to run most of the commands listed above once. If you finish a coding session and want to resume it later you only need to run 2 commands.

Activate the virtualenv we created before:
`workon rediscounter`

Move into the rediscounter path and run the Flask app:
`python website/app.py`

Dockerize the Flask application

Now that we have the application running, it would be nice if we could get it to run in a Docker container. The first thing we need to create is a `Dockerfile`. This is the file that controls how our Docker image will be built, you can think of it as a blueprint.

We can Dockerize our application in about a dozen lines of code and we'll also set things up so that the development experience will feel just as fast as if it were running natively on our OS.

That means if you edit the application, the Docker container will get updated nearly instantly without having to restart or build anything.

The Dockerfile

Create a Dockerfile file in the website sub-folder:

```
touch website/Dockerfile
```

Edit the Dockerfile file to look like this:

```
# Use the barebones version of Python 2.7.10.
FROM python:2.7.10-slim
MAINTAINER Nick Janetakis <nick.janetakis@gmail.com>

# Install any packages that must be installed.
RUN apt-get update && apt-get install -qq -y build-essential --fix-missing --no-\
install-recommends

# Set up the install path for this service.
ENV INSTALL_PATH /rediscounter
RUN mkdir -p $INSTALL_PATH

# Update the workdir to be where our app is installed.
WORKDIR $INSTALL_PATH

# Ensure packages are cached and only get updated when necessary. If we didn't d\
o this step then every time we pushed an app change it would also re-run pip ins\
tall, even if no packages changed.
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

# Copy the source from your workstation to the image at the WORKDIR path.
COPY . .

# Create a volume so that nginx can read from it.
VOLUME ["$INSTALL_PATH/build/public"]

# The default command to run if no command is specified.
CMD python app.py
```

Now that we have the `Dockerfile` in place we need to build the image, and this is the point where we'll start using the Docker compose tool.

Using Docker compose

Create a docker-compose.yml file in the website sub-folder:

```
touch website/docker-compose.yml
```

Edit the docker-compose.yml file to look like this:

```
website:
  build: .
  volumes:
    - .:/rediscounter
  ports:
    - 8000:8000
```

What's going on with this file?

build: Tells docker-compose to build the current working directory into a Docker image based on the Dockerfile.

volumes: Allows us to develop in real time with the Docker container. The Docker image will read in the contents of this directory and place it into the /rediscounter path inside of the Docker image.

ports: Ensures port 8000 is accessible on our workstation.

Build and launch the Docker container

Before you run the command below, make sure to kill the existing Flask app, you can do that by pressing CTRL+C.

```
docker-compose build
docker-compose up
```

After running the build command it will take a few minutes for Docker to pull in what it needs to. Once it finishes you can launch the container with `docker-compose up`. You should be able to access it like normal in your browser on your workstation.

You can test the live editing features created by the volume by editing the `app.py` file to say something other than Hello World! Try editing it now and then reload your browser.

Introduce Redis to our Flask application

It wouldn't be fun if all we did was run a single Flask app. Let's add Redis into our stack. In case you didn't know, Redis is an in-memory key/value store. It has a number of excellent data structures and is generally a tool to reach for when you need to save and/or read data quickly.

Adjust our docker-compose.yml file to look like this:

```
 1 redis:
 2   image: redis:2.8.21
 3   ports:
 4     - 6379:6379
 5   volumes:
 6     - ~/.docker-volumes/rediscounter/redis/data:/var/lib/redis/data
 7
 8 website:
 9   build: .
10   links:
11     - redis
12   volumes:
13     - .:/rediscounter
14   ports:
15     - 8000:8000
```

We don't need to build Redis ourself because Docker has an official hub which contains popular Docker repos. They are generally maintained by each vendor. You can find thousands of pre-made Docker images at https://registry.hub.docker.com/[7].

Lines 1-6 set up Redis. Line 2 lets Docker know we want the Redis image at version 2.8.21, which is the latest stable build at the time of writing this book.

Lines 3-4 just ensure the default Redis port is open.

How can we persist data in the running container?

Docker containers are stateless. If you run a container and make changes to it and then kill the container, those changes will not be saved. If you re-run the container, it will be a fresh start based on the original Docker image.

We can add state by mounting volumes from our workstation (or any server) into the container.

Lines 5-6 are a way to persist data. The left side of the volume command is where the volume will be located on your workstation and the right side is where that volume will be injected into the running container.

You may customize the left side to exist anywhere you want on your workstation. If the path does not exist, it will be created for you automatically. I found that for organizational purposes, naming it the same as your project works well.

[7]https://registry.hub.docker.com/

Linking multiple containers together

The last thing we need to do is link our website container to the Redis container. Lines 10-11 handle that for us. All we have to do is use the name of the image as the link name and Docker takes care of the rest. We'll soon see how we can read in this value in `app.py`.

You will need to shut down the containers and do a `docker-compose up` again to restart since we added new links, go ahead and do that now.

Adjust our Flask app to use Redis

Adjust our app.py file to look like this:

```
 1 from flask import Flask
 2 from redis import StrictRedis
 3 
 4 app = Flask(__name__)
 5 redis = StrictRedis(host='redis')
 6 
 7 @app.route('/')
 8 def hello_world():
 9     hits = redis.incr('hits')
10     return 'You visited {0} times!'.format(hits)
11 
12 if __name__ == '__main__':
13     app.run(host='0.0.0.0', debug=True)
```

Go ahead and reload your browser a few times, you should see the counter increase each time.

The implementation of this isn't too important. We have a basic hit counter hooked up through Redis. The important line to look at is line 5 where we set the host to be 'redis'. This name matches the link we set in the `docker-compose.yml` file. It effectively becomes the host name that we can connect to. Docker does this under the hood by adjusting the container's `/etc/hosts` file.

Moving beyond a trivial application

You might not have realized it but what we covered allows us to build a much more complicated system. For instance, you could add PostgreSQL into the mix by replicating what we did for Redis.

You would just add PostgreSQL to your `docker-compose.yml` file, set up the volume and then adjust the application to use PostgreSQL. Now you have an application, a database and a cache server all running together with Docker.

Need to add Elasticsearch into the mix because you want to do faceted navigated, or full text search? No problem, the pattern we used for Redis works here too.

Add real time system health monitoring

What if you wanted to add real time system monitoring to each of your containers? Hey, that's pretty easy. Let's do that now because it'll take less than 60 seconds.

Add this to the bottom of your `docker-compose.yml`:

```
cadvisor:
  image: google/cadvisor:latest
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
  ports:
  - 8080:8080
```

Kill the current running version of Docker compose and re-run `docker-compose up`.

cAdvisor is a tool built by the guys at Google. Head over to http://localhost:8080[8] to check it out. Feel free to explore the app. Don't forget to click the Docker containers link too because it will allow you to see the system stats for each individual container.

Let's make a git repository

Our application is pretty good now. It counts hits and has real time health stats. Clearly you should commit this to git and push it up to GitHub then head over to NYC and ask venture capitalists for funding because you've created the best thing ever created in the history of the interwebs.

Well, how about we just make a git repo from our source for now. The first thing you need to do is make sure you have `git` on your system. You probably do but just to be safe try running `git --version`, if you get a response then you're good to go.

Don't have git installed?

Windows/Linux users: `sudo apt-get update && sudo apt-get install git`
Mac users: `brew update && brew install git`

Prepare the project for git

Create a .gitignore file in the website sub-folder:
`touch website/.gitignore`

Edit the .gitignore file to look like this:

[8] http://localhost:8080

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]

# OS and editor files
.DS_Store
*/**.DS_Store
._*
.*.sw*
*~
.idea/
.mr.developer.cfg
.project
.pydevproject
*.tmproj
*.tmproject
tmtags
```

We're just ignoring .pyc files, a few popular editor and OS files. Depending on what application programming language or editor you use, you may need to adjust this file in your real project. For now, this should be good enough for our simple Python based Flask app.

The next file we need to create is a `.dockerignore` file which is similar to the above file, except this ignores files from being created inside of the Docker image. This isn't technically related to git, but now's a good time to add in this file.

Create a .dockerignore file in the website sub-folder:

```
touch website/.dockerignore
```

Edit the .dockerignore file to look like this:

```
.git
tmp/*
log/*
.dockerignore
```

Create a new git repo

Now we're ready to create the git repo. For the next set of commands move into the website sub-folder of the rediscounter project.

Initialize the git repo:

```
git init
```

Add all files to the git repo

```
git add -A
```

Commit everything to the git repo:

```
git commit -m "Initial commit"
```

We're not going to push anything just yet, we could push this to a popular remote git hosting service such as GitHub or BitBucket but we'll avoid that for now.

Sanity check

Double check your files against mine

Head over to the chapter-4 section on GitHub[9] and make sure your files match mine.

[9]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-4

Create a staging environment

Going from development to production is risky

We're pretty sure our app works but it would be reckless if we just pushed it directly to production. We should create a staging server and deploy it there just to make sure everything as a whole works.

This is a typical pattern found out in the wild. You develop things locally on your workstation, then you push it to a staging server which acts as an intermediary server to catch last minute issues and then finally if things look good you would promote the code to production.

Depending on the complexity of your project or its need for correctness, you might have a quality assurance team to check it over but that's not important for what we're going to tackle next.

What should we use to host our staging environment?

We're going to use Vagrant to handle provisioning our staging environment but if you're working with a larger team you could certainly spin up a Digital Ocean droplet. In fact, I would recommend using Digital Ocean or another cloud provider of your choice to act as a staging server because that's where you will be running your production server.

However, it's useful to understand how Vagrant works at some level since it's commonly used so we'll stick with that for now. It's also very reasonable to use if you're working on a project alone.

What is Vagrant?

Vagrant allows you to configure and launch virtual machines. It's not the same as VirtualBox or VMWare Player, it is something that you can use to create VirtualBox or VMWare Player VMs. It does more than that but that's all you really need to know for now about Vagrant's capabilities.

Often times people joke around and say Vagrant does a better job at creating VirtualBox virtual machines than VirtualBox itself, there is certainly truth to that statement.

We want to use Vagrant because we'll be installing CoreOS inside of it to host our app. This is what we'll be running in production, so we'll try to mimic it as close as possible in the staging server.

Install Vagrant

You can follow the instructions on Vagrant's homepage for downloading and installing it by following this link: http://www.vagrantup.com/downloads[10].

[10]http://www.vagrantup.com/downloads

Attention Windows users: make sure you download the Linux version, you will be running this in your xubuntu VM. If your CPU's architecture does not let you run a VM within a VM then you will need to skip this chapter. You won't be missing out on anything critical, you can use Digital Ocean instead or read through.

The Vagrantfile

Vagrant's configuration is done through a `Vagrantfile`. Rather than show you the file in the book, you can find it in the chapter-5 section on GitHub[11].

The file is commented out to explain what's going on. You don't need to be an expert on Vagrantfiles but it's worth reading through the entire file to become familiar with what's going on.

Launch, test and stop the Vagrant driven VM

For the next set of commands, move into the website sub-folder of the rediscounter project.

Start the Vagrant box:
`vagrant up`

This may take upwards of 2-10 minutes to complete.

SSH into the Vagrant box:
`vagrant ssh`

You should see an output that resembles this:

```
CoreOS beta (695.0.0)
core@core-01 ~ $
```

The version number might be slightly different depending on when you view this book. You can hit CTRL+D now to logout of the Vagrant box and now you should be back on your workstation.

Get the status of the Vagrant box:
`vagrant status`

You should see an output that resembles this:

```
Current machine states:

core-01                        running (virtualbox)
```

[11] https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-5/website/Vagrantfile

Stop the Vagrant box:
`vagrant halt`

If you re-run `vagrant status` you will see it's no longer running.

Destroy (but don't do this step!) the Vagrant box:
`vagrant destroy`

There's going to come a time where you'll want to completely blow out the Vagrant box you created. The above command will do that for you, but do not run that command now.

Update our ignore files

You may have noticed that after you ran `vagrant up`, it created a `.vagrant/` folder in the website path. We should add that to both the `.gitignore` and `.dockerignore` files.

Here's 2 shortcut commands to edit both files:

```
printf "\n.vagrant/" >> .dockerignore
printf "\n.vagrant/" >> .gitignore
```

With unix the >> sign allows us to append to a file. If we used > instead of >> it would overwrite the file, which is not what we want.

Commit the Vagrantfile and the updated ignore files:

```
git add -A
git commit -m "Add Vagrantfile"
```

Sanity check

Double check your files against mine

Head over to the chapter-5 section on GitHub[12] and make sure your files match mine.

[12]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-5

Crash course with systemd

What is systemd?

It does a lot of things but one of its main components is being a service manager. A lot of popular Linux distros are moving towards using it, or have began using it already.

It can be configured to ensure your services start on bootup and remain up and running even if they crash due to things out of your control. It also allows you to restrict resources to a service. For example you could easily tell a service not to use more than 512mb of memory.

In the past you may have written a custom 200 line bash file to manage a process using the ancient SysVinit system or maybe you hacked together an upstart script and then used a separate tool like Supervisord, monit or god to keep the service up and running.

The above approaches do work but why go through the hassle of all of that when you can use 1 unified tool that was cleverly designed to make process management as simple as possible?

Unit files

With systemd you can write unit files. Unit files, also commonly referred to as service files and often end up being about a dozen lines of very easy to reason about code.

We'll be writing a few unit files to manage our Docker based services.

Create our first unit file

It would be sloppy to house our unit files in the website section of our project. As this book progresses you'll soon see that there will be quite a bit more than just unit files too.

Create a unit folder under a new deploy folder in the project:

```
mkdir -p deploy/units
```

You should see an output that resembles this:

```
nick@isengard:~/Development/Sites/rediscounter  tree
.
├── deploy
│   └── units
└── website
    ├── app.py
    ├── build
    │   └── public
    ├── docker-compose.yml
    ├── Dockerfile
    ├── requirements.txt
    └── Vagrantfile

5 directories, 5 files
```

You do not need to install the `tree` command but if you want, you can install it with `sudo apt-get install tree` or `brew install tree`.

Create the redis.service file:

`touch deploy/units/redis.service`

Edit the redis.service file to look like this:

```
1  [Unit]
2  Description=Run %p
3  Requires=docker.service
4  After=docker.service
5
6  [Service]
7  Restart=always
8  ExecStartPre=-/usr/bin/mkdir -p /var/lib/%p/data
9  ExecStartPre=-/usr/bin/docker kill %p
10 ExecStartPre=-/usr/bin/docker rm -f %p
11 ExecStart=/usr/bin/docker run --rm --name %p \
12   -v /var/lib/%p/data:/var/lib/%p/data -p 6379:6379 %p:2.8.21
13 ExecStop=/usr/bin/docker stop %p
14
15 [Install]
16 WantedBy=multi-user.target
```

Let's break this down. Lines 1 to 4 describe what the unit is and what it depends on. In this case it depends on the Docker service to be running. The `%p` gets replaced by the service name.

Line 7 ensures that the service restarts if it goes down. Lines 8 to 13 control the state of the Docker container. Before it starts up we want to make sure the volume directory exists, the old container is killed and removed.

The stop command on 13 just stops the container. That leaves us with the meaty start command on lines 11 and 12. This isn't much different than what we saw in the `docker-compose.yml` file from earlier. It's just converted into a long Docker command instead of yaml.

systemd will take care of daemonizing the container for us. All we need to do now is copy it over to the server, enable it and then start it. We're not going to do that yet because there's still quite a bit of things to do.

Get used to systemd's command line tools

We never set up the Docker service to be running because CoreOS does it for us. Let's look at it with systemd's command line tools.

SSH into the Vagrant box:
```
vagrant ssh
```

Get the status of the Docker service:
```
systemctl status docker
```

You should get a whole bunch of useful information back. We can see its description, if it's loaded or not, if it's active or not along with basic resource consumption stats. It even displays the last 10 lines of log output at the bottom. That's pretty neat.

Stop the service to see what changes:
```
sudo systemctl stop docker
```

We needed to run this command with sudo since we're making a change to the service. CoreOS is configured to work with passwordless sudo already.

Get the new status of the Docker service:
```
systemctl status docker
```

You should see that this service is marked as dead.

Restart the Docker service:
```
sudo systemctl restart docker
```

Now we know how to start, stop, restart and check the status of a running service. You can run `systemctl list-unit-files` to get a long list of all systemd driven services if you're curious.

Check the logs of the Docker service:
```
journalctl -u docker
```

This tool is part of the systemd toolset and it allows us to read log files. It's very powerful. For instance we can re-run the command with `--reverse` to see the latest log entries on top:

Re-run the command with `--reverse` to sort it by newest entries:

```
journalctl -u docker --reverse
```

Follow the log in real time, similar to the tail command:

```
journalctl -u docker --follow
```

You won't notice anything because we're not actively messing with it but if new additions were added to the log output you would see it in real time. Press CTRL+C to kill it.

There are many other flags you can pass into `journalctl`, I recommend running `journalctl --help` to get an idea of what you can do. It also wouldn't hurt to run `systemctl --help` to see what's available too.

A second unit file for our Flask application

We'll create a unit file for each docker container we run.

Create the rediscounter.service file:

```
touch deploy/units/rediscounter.service
```

Edit the rediscounter.service file to look like this:

```
 1  [Unit]
 2  Description=Run %p
 3  Requires=docker.service redis.service
 4  After=docker.service redis.service
 5
 6  [Service]
 7  Restart=always
 8  ExecStartPre=-/usr/bin/docker kill %p
 9  ExecStartPre=-/usr/bin/docker rm -f %p
10  ExecStart=/usr/bin/docker run -t --rm --name %p \
11    --link redis:redis -p 8000:8000 %p
12  ExecStop=/usr/bin/docker stop %p
13
14  [Install]
15  WantedBy=multi-user.target
```

The above unit file is very similar to the `redis.service` file we created earlier in the chapter. Since we want this one to load after the Redis service we add it on lines 3 and 4.

Are you a Flask developer?

Please do not use the Flask development server in production. You should be using gunicorn or uwsgi, you could easily change the ExecStart command to launch one of those instead without having to make a Dockerfile or code change.

Everything else is about the same. Let's quickly jump back to the `redis.service` file to add a WantedBy entry.

On line 16 inside of the `redis.service` file, make this change:

Old version:

```
WantedBy=multi-user.target
```

New version:

```
WantedBy=multi-user.target rediscounter.service
```

This will inform systemd that the rediscounter service wants the Redis service.

Sanity check

Double check your files against mine

Head over to the chapter-6 section on GitHub[13] and make sure your files match mine.

[13]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-6

Harden the Flask app with nginx

What is nginx and why should you use it?

nginx is a web server. It is usually compared with Apache, both of them perform a similar role. We are going to use nginx for 2 things:

- Proxy all connections through nginx back to the Flask app
- Configure and handle SSL so everything is encrypted

If the example application had assets, such as Javascript, CSS or images then we would configure nginx to serve them too. In fact, you'll see an example on how to do that once we get into the nginx config files.

nginx is just another unit file

At the end of the day it's just another Docker container that we'll be running.

Create the nginx.service file:

```
touch deploy/units/nginx.service
```

Edit the nginx.service file to look like this:

```
 1 [Unit]
 2 Description=Run %p
 3 Requires=docker.service rediscounter.service
 4 After=docker.service rediscounter.service
 5
 6 [Service]
 7 Restart=always
 8 ExecStartPre=-/usr/bin/docker kill %p
 9 ExecStartPre=-/usr/bin/docker rm -f %p
10 ExecStart=/usr/bin/docker run -t --rm --name %p \
11   -p 80:80 -p 443:443 \
12   --link rediscounter:rediscounter \
13   -v /etc/ssl/certs:/etc/ssl/certs \
14   -v /etc/ssl/private:/etc/ssl/private %p
15 ExecStop=/usr/bin/docker stop %p
16
17 [Install]
18 WantedBy=multi-user.target
```

Our nginx unit file has a pretty similar pattern to the Redis unit file. In this case on lines 3-4 we're just making sure Docker comes up before nginx.

Line 10 is our main Docker run command where we listen on both ports 80 (HTTP) and 443 (HTTPS). We're also copying in our SSL certificates as a volume. We wouldn't want to bake in the real certificates into the image because if you were to use a third party Docker registry like the Docker hub you wouldn't want to give them access to your real certificates.

A small adjustment to our rediscounter service

Let's quickly jump back to the `rediscounter.service` file to make a few adjustments since we added nginx into the mix.

After line 11 inside of the `rediscounter.service` file, add this line:

```
ExecStartPost=-/usr/bin/docker stop nginx
```

This ensures nginx will restart after we start the rediscounter service.

On line 16 inside of the `rediscounter.service` file, make this change:

Old version:

```
WantedBy=multi-user.target
```

New version:

```
WantedBy=multi-user.target nginx.service
```

This will inform systemd that the rediscounter service wants the nginx service.

Create a custom nginx Docker image

We have our unit file in place but we don't have the actual nginx image yet. We'll create our own which will be a derivative of the official nginx image.

Create an nginx folder under the deploy folder in the project:
`mkdir -p deploy/nginx`

The Dockerfile

Create the Dockerfile file:
`touch deploy/nginx/Dockerfile`

Edit the Dockerfile file to look like this:

```
# Pull in from the official nginx image.
FROM nginx:1.9.3
MAINTAINER Nick Janetakis <nick.janetakis@gmail.com>

# Delete default static pages.
RUN rm /usr/share/nginx/html/*

# Copy over the custom nginx and default configs.
COPY configs/nginx.conf /etc/nginx/nginx.conf
COPY configs/default.conf /etc/nginx/conf.d/default.conf

# Copy over the self signed SSL certificates.
COPY certs/rediscounter.crt /etc/ssl/certs/rediscounter.crt
COPY certs/rediscounter.key /etc/ssl/private/rediscounter.key
COPY certs/dhparam.pem /etc/ssl/private/dhparam.pem

# Allow us to customize the entry point of the image.
COPY docker-entrypoint /
RUN chmod +x /docker-entrypoint
ENTRYPOINT ["/docker-entrypoint"]

# Start nginx in the foreground.
CMD ["nginx", "-g", "daemon off;"]
```

We're baking in our custom config which has yet to be created. We're also baking in the self signed SSL certificates because leaking the self signed certificates isn't a big deal. The `dhparam.pem` file is something we'll create now. It is used to harden your SSL configuration.

Generate the dhparam.pem file:

```
mkdir deploy/certs
openssl dhparam -out deploy/certs/dhparam.pem 2048
```

Continuing on in our Dockerfile, lines 16-19 allow us to create a custom entry point for the Docker image. We want to make a few variable substitutions in our config files before they are baked into the image. Currently nginx doesn't allow you to pass environment variables in so we'll have to wire up a makeshift templating system.

The Docker entrypoint file

Create the docker-entrypoint file:
`touch deploy/nginx/docker-entrypoint`

Edit the docker-entrypoint file to look like this:

```
 1 #!/usr/bin/env bash
 2 set -e
 3 
 4 # Overwrite a few variables, this allows us to use the same template
 5 # for development, staging and production.
 6 CONFIG_PATH="/etc/nginx/conf.d/default.conf"
 7 STAGING_IP="172.17.8.101"
 8 STAGING_HOSTNAME="core-01"
 9 DOMAIN_NAME="yourrealdomain.com"
10 
11 if [[ $(hostname) != "${STAGING_HOSTNAME}" ]]; then
12   sed -i "s/${STAGING_IP}/${DOMAIN_NAME}/g" "${CONFIG_PATH}"
13 fi
14 
15 # Execute the CMD from the Dockerfile.
16 exec "$@"
```

Lines 6 to 9 are custom variables we've set up. If you're using Vagrant for the staging server you won't need to change lines 7 and 8 but if you've set up a staging server somewhere else you may need to adjust those values. Line 9 would be your real domain name in production.

Lines 11 to 13 do the actual variable replacements with the unix tool sed. If the host isn't staging it will go ahead and swap out the staging IP address with the real domain name at the config path. If we needed to add more variables, this would be the place to add them.

The nginx configuration files

Both of the files we'll be creating are a bit too long to paste into this book, go ahead and find them in this chapter's folder on GitHub[14].

nginx.conf

These are settings that would apply to any app or site that will be proxied by nginx. Read the comments in the file for more details if you're curious.

Now that we have the nginx config out of the way we need to make a default configuration.

default.conf

You probably noticed that at the bottom of the `nginx.conf` file we're loading in this default file. This file will be where we'll set up the proxy for our Flask application and configure everything else.

As you look through this file you'll notice that we're referencing SSL certificates that have yet to be created, let's make them now.

Create self signed SSL certificates

The whole basis of certificate validation revolves around a few specific authorities that issue out certificates. Modern browsers automatically trust them by default and sometimes provide shiny green secure icons next to domain names in the URL address bar.

Our self signed certificates were signed by us. We are not an official authority, so our browser will warn us when trying to access the page by telling us this certificate has not been properly signed. It will still work but it's a minor inconvenience to the end user of the web site.

In a real production set up we would obtain properly signed certificates by a real certificate authority and copy over the real certificates onto the server. We'll be doing that shortly.

[14]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-7/deploy/nginx/configs

Generate the self signed certificates:

```
openssl req \
    -newkey rsa:2048 -nodes -sha256 -keyout \
    deploy/certs/rediscounter.key \
    -x509 -days 3650 -out deploy/certs/rediscounter.crt \
    -subj "/C=US/ST=NewYork/L=NewYork/O=IT/CN=fakerediscounter.com"
```

The above command will create certificates that are valid for about 10 years. The contents of the subject flag are not important, we are just satisfying the requirements to create a certificate.

Sanity check

Double check your files against mine

Head over to the chapter-7 section on GitHub[15] and make sure your files match mine.

[15]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-7

Test drive our staging server

Get the code onto the server

Right now we have a situation where our source code is on our workstation but in order for our rediscounter app to be deployed, it needs to end up on our staging server.

Fortunately this is really easy with git hooks. Git has options for things to happen based on certain events. There are about 20 or so different hooks but the one we're concerned about is the `post-receive` hook.

This allows us to automatically perform arbitrary tasks whenever the git repo receives something. It's perfect for our use case.

Ideal work flow

- git push the code to the staging server
- post-receive hook gets executed
- post-receive code we've written will
  - checkout the code to a separate repo local on the staging server
  - build and tag a proper Docker image based off that source
  - restart the service
  - [insert whatever else you want]

All of the above steps except for the git push will be automated!

The post-receive code we'll write is just simple bash. The hooks are language agnostic, meaning you could write them in Python or something else if you choose to do so.

Create the git hooks

In our case we'll want 1 post-receive hook for each git repo we have. At the moment we have 2 git repos. We have one for the Flask app and another for nginx.

Create a git/post-receive folder under the deploy folder in the project:

```
mkdir -p deploy/git/post-receive
```

Create both files:

```
touch deploy/git/post-receive/rediscounter \
deploy/git/post-receive/nginx
```

Edit the rediscounter file to look like this:

```
#!/usr/bin/env bash

# Configuration.
REPO_NAME="rediscounter"

# Check out the newest version of the code.
export GIT_WORK_TREE="/var/git/${REPO_NAME}"
git checkout -f

TAG="$(git log --pretty=format:'%h' -n 1)"
FULL_COMMIT_TAG="${REPO_NAME}:${TAG}"
FULL_LATEST_TAG="${REPO_NAME}:latest"

# Build the image with the proper commit tag.
docker build -t "${FULL_COMMIT_TAG}" "${GIT_WORK_TREE}"

# Get the Docker ID of the last built image.
DOCKER_ID="$(docker images -q $REPO_NAME | head -1)"

# Tag a latest version based off the proper commit tag.
docker tag -f "${DOCKER_ID}" "${FULL_LATEST_TAG}"

echo "Restarting ${REPO_NAME}"
docker stop "${REPO_NAME}"

echo "Removing untagged Docker images (may take a while)"
docker rmi $(docker images --quiet --filter "dangling=true")

echo "Restarting nginx"
docker stop "nginx"
```

Edit the nginx file to look like this:

```
 1  #!/usr/bin/env bash
 2
 3  # Configuration.
 4  REPO_NAME="nginx"
 5
 6  # Check out the newest version of the code.
 7  export GIT_WORK_TREE="/var/git/${REPO_NAME}"
 8  git checkout -f
 9
10  TAG="$(git log --pretty=format:'%h' -n 1)"
11  FULL_COMMIT_TAG="${REPO_NAME}:${TAG}"
12  FULL_LATEST_TAG="${REPO_NAME}:latest"
13
14  # Build the image with the proper commit tag.
15  docker build -t "${FULL_COMMIT_TAG}" "${GIT_WORK_TREE}"
16
17  # Get the Docker ID of the last built image.
18  DOCKER_ID="$(docker images -q $REPO_NAME | head -1)"
19
20  # Tag a latest version based off the proper commit tag.
21  docker tag -f "${DOCKER_ID}" "${FULL_LATEST_TAG}"
22
23  echo "Restarting ${REPO_NAME}"
24  docker stop "${REPO_NAME}"
25
26  echo "Removing untagged Docker images (may take a while)"
27  docker rmi $(docker images --quiet --filter "dangling=true")
```

Both files are nearly identical. We just need to adjust the repo name on line 4. The beauty of this set up is that you're free to do anything else you want in the hook. If your Flask app were non-trivial, perhaps it includes a background worker, then you could have that get restart as well.

You could even go all out and create your own mini-continous integration solution that runs tests automatically and then pushes the code to production if everything passed. The sky is the limit.

The post-receive hook as is works great because everything is hosted on a single machine. We don't even need a Docker registry because all we have to do is build the Docker image and then restart the previous container, and voila we have a new version of our container running in about 1 second.

On line 10, we custom tag each Docker image with the git SHA to keep old versions around in case we need to roll back to a previous version.

Now, our unit files only run the latest version but you could put additional scripting in place to edit the unit file with the git SHA, then do a `sudo systemctl daemon-reload` so that systemd picks up the new unit file and restarts the service you've deployed. Suddenly you have the power to rollback in seconds on demand.

Disk space will get chewed up pretty quickly

If each build you push creates a new image then you will pretty quickly go through disk space. Let's say each image is 200MB and you push 10 updates in 1 day then you've just used 2GB of space.

It would be a very good idea to create a little script that deletes older images automatically. systemd allows you to run timed units that can execute at specific times, completely eliminating the need for cron.

Alternatively you can adjust the git hooks to always use `:latest` instead of a unique SHA tag and it will overwrite the existing image.

Going back to the Vagrantfile

We need to create the git repos on the CoreOS instance so we can push to them. At this point we'll also start moving over the files we need to the staging server. Instead of doing this by hand, we can just add certain commands to our `Vagrantfile` to have these files and directories set up whenever we provision a new Vagrant box.

Browse to this chapter's Vagrantfile[16] on GitHub to see the new additions.

Copy the contents of this new `Vagrantfile` into yours now and review the file.

The main changes are on lines 8 to 11 and lines 63+. We have to forward both port 80 and 443 to non-standard HTTP/HTTPS ports so we can access them on our workstation and we also removed the old port 8000 entry because we're no longer accessing our Flask app directly.

Lines 63+ sets up the system for us.

We won't be using Vagrant in production

In a real production server we won't be using Vagrant, so we will need to run these commands ourselves on our server. It wouldn't be difficult to create a simple shell script that does this automatically. That won't be covered in this book, but I think you'll be able to pull it off! You only need to run them once to set up the initial system, so it's not too huge of a burden.

Another option would be to use Saltstack[17] or Ansible[18] to provision the initial server but unless you're provisioning a lot of servers frequently, it might not be worth the learning curve.

[16]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-8/website/Vagrantfile
[17]http://saltstack.com/
[18]http://www.ansible.com/home

Launch a new Vagrant box

We've done so much that it's a good idea to start from a clean slate. Execute the following commands from within the website sub-folder of the rediscounter project.

Destroy the Vagrant box:

```
vagrant destroy
```

Create a new Vagrant box:

```
vagrant up
```

This is going to take a bit longer than the last time because we're pulling in Docker images inside of the Vagrantfile.

Configure SSH on your workstation

Before we continue on we need to prepare our workstation to be able to ssh into Vagrant without depending on the special `vagrant ssh` command.

Append the output of vagrant ssh-config to your SSH config:

```
vagrant ssh-config >> ~/.ssh/config
```

We need to do the above command because Vagrant does some black magic to allow you to SSH into the box. If you run the command without redirecting the output to the config file you can see what will get written out.

Make sure you only run that command once, otherwise it will keep appending the value to your ~/.ssh/config file.

Configure your git repos to push to the staging server

Now that we have everything set up we just need to configure our git repos by adding a new remote location. It will be the staging server.

Add the new remotes:

```
# Run this from within the nginx directory.
git init && git add -A && git commit -m "Initial commit"
git remote add staging ssh://core-01:/var/git/nginx.git

# Run this from within the rediscounter's website directory.
git remote add staging ssh://core-01:/var/git/rediscounter.git
```

With that in place you will be able to run `git push staging master` to push changes to the staging server after you've added and committed the changes to git. Do it now for both repos.

SSH into the CoreOS instance:
`vagrant ssh`

Enable and start both services inside of CoreOS:

```
sudo systemctl enable rediscounter.service nginx.service
sudo systemctl start rediscounter.service nginx.service
```

At this point you can visit https://localhost:8081[19] in your browser on your workstation and you'll see the application. It is being served over SSL with self signed certificates so you will need to tell your browser to trust the connection.

Sanity check

Double check your files against mine

Head over to the chapter-8 section on GitHub[20] and make sure your files match mine.

[19]https://localhost:8081
[20]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-8

Create a production server

How will we host the application?

Currently the app has only been running inside of a Vagrant driven CoreOS instance. This is fine and dandy for personal use but now we need to get our code onto a publicly accessible network and we're going to do that by using Digital Ocean.

Sign up with Digital Ocean

In order to redeem your free $10 in credit you must sign up with this referral URL:
https://www.digitalocean.com/?refcode=0a14c0d916b3[21]

You will be automatically credited $10 upon sign up as long as you use the URL above. You don't need to enter in a coupon code.

Go ahead and sign up now. After signing up you can create your first droplet. A droplet is how Digital Ocean defines a server or instance. We'll pick the $5/month plan because our app is pretty small at the moment.

An awesome thing about Digital Ocean is you only get billed for what you use. If you spin the server up for an hour and destroy the instance then you will only get billed for 1 hour, not a full month.

512mb of RAM is more than enough to run Redis and our Flask application. When it comes time to select an image, find **CoreOS - Beta** in the list. You'll also want to enable **Private Networking** in the settings.

Add in your SSH keypair to the droplet

It would also be a good idea to **add your SSH public key** because being able to login without a username and password is much more secure and user friendly.

Check to see if you have a public key set up:

```
ls -la ~/.ssh | grep id_
```

If you see an `id_rsa` and `id_rsa.pub` file you're good to go. You'll want to copy the entire contents of the `id_rsa.pub` file into the textarea on the Digital Ocean site for the SSH key.

[21] https://www.digitalocean.com/?refcode=0a14c0d916b3

Don’t have an SSH keypair? Let’s create one

You only need to run these commands if you do NOT have id rsa keys already.

Create an SSH keypair:
`ssh-keygen -t rsa -b 4096`

It will then prompt you to save the file, you can press enter to save it to the default location. You can also skip creating a passphrase in the next input prompt.

Start the SSH agent and add our newly created key to it:
`eval "$(ssh-agent -s)" && ssh-add ~/.ssh/id_rsa`

Once that’s done you can copy/paste the contents of `id_rsa.pub` into the Digital Ocean site for the SSH key.

Prepare your droplet

After the droplet has been created you should SSH into it.

SSH into the droplet:
ssh core@THE_IP_ADDRESS_OF_YOUR_SERVER

You can find the IP address under the server name in the Digital Ocean admin panel for the droplet you just created.

The first thing we’ll want to do is disable the update-engine for CoreOS. Typically you would run multiple CoreOS servers in a cluster, and it will be able to keep itself updated by performing rolling restarts but we only have 1 server and having it reboot on its own is not going to work for us.

Disable auto-updates:

```
sudo systemctl disable update-engine
sudo systemctl stop update-engine

sudo systemctl disable update-engine-stub
sudo systemctl stop update-engine-stub
```

Now that we have that out of the way we need to copy over the files we moved from our workstation to the Vagrant CoreOS instance over to our Digital Ocean droplet.

Set up the remote git repos:

```
sudo mkdir -p /var/git/nginx.git /var/git/nginx
sudo mkdir -p /var/git/rediscounter.git /var/git/rediscounter

sudo git --git-dir=/var/git/nginx.git --bare init
sudo chown -R core:core /var/git/nginx.git
sudo chown -R core:core /var/git/nginx

sudo git --git-dir=/var/git/rediscounter.git --bare init
sudo chown -R core:core /var/git/rediscounter.git
sudo chown -R core:core /var/git/rediscounter
```

Copy files over to the droplet

We're going to be running a bunch of `scp` commands which will allow us to copy files from our workstation to the droplet.

Make sure to run all of these commands from the `deploy/` folder on your workstation.

Create a variable in our terminal for the server's IP address:
`IP="THE_IP_ADDRESS_OF_YOUR_SERVER"`

It will be really annoying to type our IP address a bunch, so let's just put it into a variable that we can use until we close our terminal.

```
# Copy over the certificates
scp nginx/certs/rediscounter.crt core@${IP}:/tmp/rediscounter.crt
scp nginx/certs/rediscounter.key core@${IP}:/tmp/rediscounter.key
scp nginx/certs/dhparam.pem core@${IP}:/tmp/dhparam.pem

# Copy over the unit files
scp units/nginx.service core@${IP}:/tmp/nginx.service
scp units/redis.service core@${IP}:/tmp/redis.service
scp units/rediscounter.service core@${IP}:/tmp/rediscounter.service

# Copy over the git hooks
scp git/post-receive/nginx core@${IP}:/tmp/nginx
scp git/post-receive/rediscounter core@${IP}:/tmp/rediscounter
```

Move the files on the droplet

We can't copy over the files with sudo, so we placed them all into `/tmp`, now we need to move them to their correct location.

```
# Move the SSL certificates to the correct location
sudo mv /tmp/rediscounter.crt /etc/ssl/certs
sudo mv /tmp/rediscounter.key /etc/ssl/private
sudo mv /tmp/dhparam.pem /etc/ssl/private

# Move the unit files to the correct location
sudo mv /tmp/nginx.service /etc/systemd/system
sudo mv /tmp/redis.service /etc/systemd/system
sudo mv /tmp/rediscounter.service /etc/systemd/system

# Move the git hooks to the correct location
sudo mv /tmp/nginx /var/git/nginx.git/hooks/post-receive
sudo mv /tmp/rediscounter /var/git/rediscounter.git/hooks/post-receive

# Make the git hooks executable
chmod +x /var/git/nginx.git/hooks/post-receive
chmod +x /var/git/rediscounter.git/hooks/post-receive
```

Set up the Redis Docker image

Currently our CoreOS instance in production has no Redis Docker image.

Pull, enable and start the Redis service:

```
docker pull redis:2.8.21
sudo systemctl enable redis.service
sudo systemctl start redis.service
```

Add new git remotes on our workstation

Head back over to your workstation and goto each git repo we have set up. Previously we added a staging remote for our staging server, now it's time to create one for production.

Head over to the nginx git repo on your workstation and run:

```
git remote add production ssh://core@IP_ADDRESS:/var/git/nginx.git
git push production master
```

Head over to the rediscounter git repo on your workstation and run:

```
git remote add production ssh://core@IP_ADDRESS:/var/git/rediscounter.git
git push production master
```

Enable and start them in production

SSH back into the production CoreOS server before doing anything.

Enable and start the services with systemd:

```
sudo systemctl enable rediscounter.service nginx.service
sudo systemctl start rediscounter.service nginx.service
```

You should now be able to access https://IP_ADDRESS in your browser. We're still using the self signed certificates so you'll get the security warning but that's fine, we'll fix that soon enough.

Secure your droplet

Right now all of our ports are open and this isn't very secure. We will use the `iptables` tool that comes with CoreOS to lock things down.

On your workstation, create a production folder in the deploy folder:
`mkdir production`

Create the rules-save file:
`touch production/rules-save`

Edit the rules-save file to look like this:

```
 1 *filter
 2
 3 :INPUT DROP [0:0]
 4 :FORWARD DROP [0:0]
 5 :OUTPUT ACCEPT [0:0]
 6
 7 -A INPUT -i lo -j ACCEPT
 8 -A INPUT -i eth1 -j ACCEPT
 9 -A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
10 -A INPUT -p tcp -m tcp --dport 22 -j ACCEPT
11 -A INPUT -p tcp -m tcp --dport 80 -j ACCEPT
12 -A INPUT -p tcp -m tcp --dport 443 -j ACCEPT
13 -A INPUT -p icmp -m icmp --icmp-type 0 -j ACCEPT
14 -A INPUT -p icmp -m icmp --icmp-type 3 -j ACCEPT
15 -A INPUT -p icmp -m icmp --icmp-type 11 -j ACCEPT
16
17 COMMIT
```

This file is a little cryptic but this is the syntax for an `iptables` config. By default access to Redis is locked down so no one except your Flask app can access it.

If you want to give the world access to Redis add this after line 12:

```
-A INPUT -p tcp -m tcp --dport 6379 -j ACCEPT
```

This would be useful to do if let's say you had a database like PostgreSQL running on your server, perhaps you would have a backup script that runs every 12 hours and it would reside on a different server. If that's the case you would want to allow connections to PostgreSQL from the outside world. You could even white list only a specific IP address for added security.

SCP over the file from your workstation to the Digital Ocean droplet:

```
scp production/rules-save core@${IP}:/tmp/rules-save
```

SSH into the droplet and run these commands:

```
# Move the ip tables config to the correct location
sudo mv /tmp/rules-save /var/lib/iptables

# Change ownership to root for the iptables config
sudo chown root:root /var/lib/iptables/rules-save

# Enable and start iptables
sudo systemctl enable iptables-restore
sudo systemctl start iptables-restore

# Restart Docker because we made an iptables change
sudo systemctl restart docker
```

Sanity check

It would be a good idea to restart your droplet using Digital Ocean's admin panel and then make sure you can SSH back into it once it's available. All of the services should come up on their own. You can verify that by trying to visit the site in your browser.

If you cannot connect, SSH into the droplet and run `docker ps`. Make sure all of the services are running. If they are not, then step back and re-trace this chapter carefully.

You can use the `journalctl -u nginx --reverse` command to look at log files to help debug issues, feel free to replace `nginx` with `rediscounter` to help debug the rediscounter service.

Double check your files against mine

Head over to the chapter-9 section on GitHub[22] and make sure your files match mine.

[22]https://github.com/nickjj/deploy-web-apps-with-docker/tree/master/chapter-9

Register a domain name

Accessing your server by IP address is lame

You do not have to follow this chapter right now if you don't want to as it will require making a purchase and perhaps you're not ready to buy your domain yet but you should still read through it.

My recommendation is to sign up at http://internetbs.net/[23]. I have no affiliation with them but their service is quite good. It might not look as fancy as Namecheap[24] but their pricing is competitive. They even support crypto-currencies like Bitcoin if you're into that sort of thing.

You can buy a .com domain name for $8.49 per year and it comes with free WHOIS privacy for life. Most other registrars will charge you about $10 for the domain and then charge you an additional fee ranging from $10 to $30 for WHOIS privacy per year.

WHOIS privacy means that if someone runs a WHOIS on your domain name, they will see the registrar's contact details instead of your own. Without this privacy guard in place then anyone would be able to see your personal address that you used when signing up for the domain name.

Unfortunately we live in a world where consumers like us get gouged by nonsense like this. It should be private by default at no cost for everyone. Another benefit of http://internetbs.net/[25] is that they are hosted offshore from the US. I'm not a lawyer but it's probably an advantage to not have your domain registered in the US.

Configure your domain

After you sign up and pick your domain name, you will need to hook up a few things. Head over to "My Account" on the InternetBS control panel after you've signed in.

Then click your domain name and choose the "DNS Management" icon. Depending on what their design looks like when you read this book, it might be on the bottom left and it looks like a cog wheel. You should now be in a position to make edits to your DNS entries.

[23]http://internetbs.net/
[24]https://www.namecheap.com/
[25]http://internetbs.net/

Goto the "A Records" section and make it look like this:

```
NAME                 TTL    IPv4 Address
[   ].yourdomain.com   3600   YOUR_DROPLET_IP_ADDRESS
[www].yourdomain.com   3600   YOUR_DROPLET_IP_ADDRESS
```

Once that's done, you can save each row by clicking the disk icon. That effectively maps your domain name to your Digital Ocean droplet's IP address.

It usually takes around 24 hours for the changes to kick in. I've seen it done as fast as 2 hours, it really depends on when you've done it in relation to when they update them.

Once your DNS rolls over you should be able to access your site through yourdomain.com rather than by the IP address. Congrats for setting up a domain name.

Configure your e-mail

You might want to go to the "My Account" section again and forward a few e-mail addresses too. Typically you would make a few e-mail addresses that forward to a GMail account. This way you get professional and custom e-mail addresses while leveraging GMail's mail servers.

You will want to create at least this e-mail address: postmaster@yourdomain.com, and forward it to whatever existing e-mail address you have.

We need to make that one because in the next section when we create our free SSL certificate, it will verify your domain name by sending an e-mail to that address. You may need to wait a few hours before it becomes active.

You can test it by periodically sending e-mails to postmaster@yourdomain.com and waiting until you actually see it. In the mean time, go stretch your legs and come back in a bit.

Grab a legit SSL certificate

A rant on SSL certificate vendors

If anything is gouged more than WHOIS privacy in the world of registering and hosting domain names, it's definitely SSL certificates. A lot of certificate authorities will charge about $50 for a single certificate per year.

A lot of these SSL certificate offering companies are not even proper authorities. They are just re-sellers of white listed authorities trying to make as much profit as possible before the market gets more saturated than it is already is.

You can however get a free basic certificate from https://www.startssl.com/[26]. As of writing this book, they are the only place that offers a basic certificate for free without a million strings attached. Unfortunately it doesn't come with wildcard sub-domains, so if you have dynamic sub-domains you're out of luck.

Careful now, incoming opinions

Also, in my personal opinion, they are a very slimy company who will try and nickel / dime you to death. For instance their web UI is absolutely horrible for creating a new free certificate. It's basically designed for you to make a mistake that you can't reverse unless you pay them $25 to revoke the certificate.

Companies like this disgust me, but hey it's our only choice for a free cert and once you have the certificate created you won't have to deal with them until you need to renew it in a year.

Hopefully by the end of 2015 we can all use this amazing service https://letsencrypt.org/[27]. I can't wait for them to ship this out and iron out the kinks. It's heavily sponsored by Mozilla and other popular internet driven companies and will change how SSL certificates are set up for the better.

Until then if you want to avoid StartSSL out of principal then head over to https://sslmate.com/[28]. You can register a certificate in 1 command and it costs $16/year with free re-issues and revokes.

Ok, that's enough ranting for now. I just wanted to make it crystal clear that the only reason I even mentioned StartSSL is because I want to offer you guys a free means of obtaining a basic SSL certificate.

[26]https://www.startssl.com/
[27]https://letsencrypt.org/
[28]https://sslmate.com

I would certainly go with https://sslmate.com/[29] until https://letsencrypt.org/[30] is available. If you end up going with SSLMate then you can skip most of this chapter and save yourself some time. Just watch their 50 second video to see how fast and painless it is to set up and then join me in the Backup your server certificates in a safe place section.

Signing up with StartSSL

Yep, even signing up with StartSSL is a huge headache. Here's a 7 step guide to victory:

- Goto https://www.startssl.com/[31] and click the **control panel** link near the top right
- Click the **sign up** link, assuming you don't already have an account
- Fill in **all the required fields**
- **Check your e-mail** to obtain the verification code they sent
- Complete the registration by entering in the code and **clicking continue**
- Generate a **High Grade** key, this is only to login, not your certificate
- Click the **install button** and it will install the login certificate in your browser

For some reason they thought it was a brilliant idea to use client side certificates just so you can login to their website. Hopefully you never forget to back this up if you change devices.

Prove to StartSSL that you own your domain

Now that we can sign in, we need to begin creating our certificate but first we need to verify we own the domain. Head over to the **control panel** and follow the steps below:

- Click the **Validations Wizard** tab
- Choose **Domain Name Validation** from the dropdown list
- **Enter in your domain name**
- **Select postmaster@yourdomain.com from the list** and hit continue
- **Check your e-mail** to obtain the verification code they sent
- **Complete the validation process** by entering in the code and clicking **continue**

[29]https://sslmate.com
[30]https://letsencrypt.org/
[31]https://www.startssl.com/

Generate the SSL certificate

Save the certificates in your project

Before we create the certificate we should go over where to save them locally. You should put all of your certificates in the `deploy/production/certs` path.

Create the deploy/production/certs path:

```
mkdir deploy/production/certs
```

Be careful with this directory

This spot isn't a git repo nor is it being added to a Docker image so we don't have to ignore it in any files. Just be careful never to share this directory with someone by accident.

Certificate creation steps

With verification out of the way we can go ahead and generate the certificate. Head over to the **control panel** and follow the steps below:

- Click the **Certificates Wizard** tab
- Choose **Web Server SSL/TLS Certificate** from the dropdown list
- Click **skip** when it comes time to generate the private key
  - *We want to generate our own key so StartSSL does not know it*
- **Move into the deploy/production/certs folder**
- **In a terminal**, run the following 2 commands
  - `openssl genrsa 2048 > yourdomain.key`
  - `openssl req -new -key yourdomain.key -out csr.pem`
  - *Enter in the Common Name that you used earlier in the validation process*
- **Open csr.pem in your favorite editor** and **copy it to your clipboard**
- **Paste the contents of your clipboard** into the textarea on StartSSL
- Click **continue** to submit your certificate request file
- **Select the domain name** you validated earlier
- **Enter in www as a subdomain** that you want this to also work on
- **Confirm everything** by clicking **continue**
- **You may or may not need to wait an hour** for StartSSL to verify it
- Copy the contents of that textarea to a file named **yourdomain-raw.crt**
  - *This is your certificate, do not lose it!*

Create a proper certificate chain

In order to get A+ SSL certificate status you need to create a certificate chain. It will be a combination of your certificate and StartSSL's intermediary certificate.

Download StartSSL's intermediary certificate:

```
wget https://www.startssl.com/certs/class1/sha2/pem/sub.class1.server.sha2.ca.pem
```

Concatenate your certificate with theirs:

```
cat yourdomain-raw.crt sub.class1.server.sha2.ca.pem > yourdomain.crt
```

Backup your client certificate from StartSSL

Since they decided to use client side SSL certificates for authentication you will be completely locked out of your account if you ever format your drive or change devices without first backing up your certificate.

You can follow their instructions on how to do this here:
https://www.startssl.com/?app=25#4[32]

Backup your server certificates in a safe place

It would be a really good idea to backup these files to a Flash drive or onto another device that you use for backups. You certainly don't want to lose them. Also make a note in a calendar at this time because 1 year from now you will need to renew your certificates.

Copy your new certificates to the production server

At this point you should have a few files but the ones we'll be using are, `yourdomain.crt` and `yourdomain.key`. Our nginx configuration expects the names to be `rediscounter.crt` and `rediscounter.key`, so we'll need to change the nginx config.

Keep your certificate names the same across environments

I recommend keeping the certificate names the same between staging and production so you don't have to adjust your nginx configuration.

Let's assume for the sake of argument that they are named `foobar`, of course in real life this would be your domain name. Let's copy over the new production certificates.

[32]https://www.startssl.com/?app=25#4

Create a variable in our terminal for the server's IP address:
`IP="THE_IP_ADDRESS_OF_YOUR_SERVER"`

It will be really annoying to type our IP address a bunch, so let's just put it into a variable that we can use until we close our terminal.

```
scp production/certs/foobar.crt core@${IP}:/tmp/foobar.crt
scp production/certs/foobar.key core@${IP}:/tmp/foobar.key
```

SSH into the droplet:
ssh core@THE_IP_ADDRESS_OF_YOUR_SERVER

You can find the IP address under the server name in the Digital Ocean admin panel for the droplet you just created.

Move the files we just SCP'd over:

```
sudo mv /tmp/foobar.crt /etc/ssl/certs
sudo mv /tmp/foobar.key /etc/ssl/private
```

On your workstation, edit the deploy/nginx/configs/default.conf file:

Replace lines 38 and 39 with:

```
ssl_certificate       /etc/ssl/certs/foobar.crt;
ssl_certificate_key   /etc/ssl/private/foobar.key;
```

Push the new version of nginx by going to the nginx git repo and running:

```
git add -A
git commit -m "Change certificate names"
git push production master
```

At this point nginx should get rebuilt and in a few seconds restarted.

Verify that it works and is graded A+

You should be able to access your website over SSL with no security warnings in your browser and you should also see a lock icon in the address bar of your browser.

If everything is working then head over to to see what grade your SSL is:
https://www.ssllabs.com/ssltest/[33]

The above site is a nice little free service to gauge the security of your setup. Between how our nginx configuration is set up and how we created our certificate, you should receive an A+.

[33]https://www.ssllabs.com/ssltest/

Wrapping things up

That brings us to the end of the book. Hopefully you've learned a lot and managed to work through the book in full. I wish you good luck and if you're looking to build a large Flask application you should check out http://buildasaaswithflask.com[34], it could very well save you hundreds of hours of development time.

[34]http://buildasaaswithflask.com