The Flask Mega-Tutorial Part XIX: Deployment on Docker Containers (/post/the-flask-mega-tutorial-part-xix-deployment-on-docker-containers)

April 10 2018

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

This is the nineteenth installment of the Flask Mega-Tutorial series, in which I'm going to deploy Microblog to the Docker container platform.

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

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

- Chapter 20: Some JavaScript Magic (/post/the-flask-mega-tutorial-part-xx-some-javascript-magic)
- Chapter 21: User Notifications (/post/the-flask-mega-tutorial-part-xxi-user-notifications)
- Chapter 22: Background Jobs (/post/the-flask-mega-tutorial-part-xxii-background-jobs)
- Chapter 23: Application Programming Interfaces (APIs) (/post/the-flask-mega-tutorial-part-xxiii-application-programming-interfaces-apis)

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

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

In Chapter 17 (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux) you learned about traditional deployments, in which you have to take care of every little aspect of the server configuration. Then in Chapter 18 (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku) I took you to the other extreme when I introduced you to Heroku, a service that takes complete control of the configuration and deployment tasks, allowing you to fully concentrate on your application. In this chapter you are going to learn about a third application deployment strategy based on *containers*, more particularly on the Docker (https://www.docker.com/) container platform. This third option sits somewhere in between the other two in terms of the amount of deployment work needed on your part.

Containers are built on a lightweight virtualization technology that allows an application, along with its dependencies and configuration to run in complete isolation, but without the need to use a full blown virtualization solution such as virtual machines, which need a lot more resources and can sometimes have a significant performance degradation in comparison to the host. A system configured as a container host can execute many containers, all of them sharing the host's kernel and direct access to the host's hardware. This is in contrast to virtual machines, which have to emulate a complete system, including CPU, disk, other hardware, kernel, etc.

In spite of having to share the kernel, the level of isolation in a container is pretty high. A container has its own file system, and can be based on an operating system that is different than the one used by the container host. For example, you can run containers based on Ubuntu Linux on a Fedora host, or vice versa. While containers are a technology that is native to the Linux operating system, thanks to virtualization it is also

possible to run Linux containers on Windows and Mac OS X hosts. This allows you to test your deployments on your development system, and also incorporate containers in your development workflow if you wish to do so.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.19), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.19.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.18...v0.19).

Installing Docker CE

While Docker isn't the only container platform, it is by far the most popular, so that's going to be my choice. There are two editions of Docker, a free community edition (CE) and a subscription based enterprise edition (EE). For the purposes of this tutorial Docker CE is perfectly adequate.

To work with Docker CE, you first have to install it on your system. There are installers for Windows, Mac OS X and several Linux distributions available at the Docker website (https://www.docker.com/community-edition). If you are working on a Microsoft Windows system, it is important to note that Docker CE requires Hyper-V. The installer will enable this for you if necessary, but keep in mind that enabling Hyper-V prevents other virtualization technologies such as VirtualBox from working.

Once Docker CE is installed on your system, you can verify that the install was successful by typing the following command on a terminal window or command prompt:

```
$ docker version
Client:
 Version:      17.09.0-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:40:09 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.09.0-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   afdb6d4
 Built:        Tue Sep 26 22:45:38 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

Building a Container Image

The first step in creating a container for Microblog is to build an *image* for it. A container image is a template that is used to create a container. It contains a complete representation of the container file system, along with various settings pertaining to networking, start up options, etc.

The most basic way to create a container image for your application is to start a container for the base operating system you want to use (Ubuntu, Fedora, etc.), connect to a bash shell process running in it, and then manually install your application, maybe following the guidelines I presented in Chapter 17 (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux) for a traditional deployment. After you install everything, you can take a snapshot of the container and that becomes the image. This type of workflow is supported with the `docker` command, but I'm not going to discuss it because it is not convenient to have to manually install the application every time you need to generate a new image.

A better approach is to generate the container image through a script. The command that creates scripted container images is `docker build .`. This command reads and executes build instructions from a file called *Dockerfile*, which I will need to create. The Dockerfile is basically an installer script of sorts that executes the installation steps to get the application deployed, plus some container specific settings.

Here is a basic *Dockerfile* for Microblog:

Dockerfile: Dockerfile for Microblog.

```
FROM python:3.6-alpine

RUN adduser -D microblog

WORKDIR /home/microblog

COPY requirements.txt requirements.txt
RUN python -m venv venv
RUN venv/bin/pip install -r requirements.txt
RUN venv/bin/pip install gunicorn

COPY app app
COPY migrations migrations
COPY microblog.py config.py boot.sh ./
RUN chmod +x boot.sh

ENV FLASK_APP microblog.py

RUN chown -R microblog:microblog ./
USER microblog

EXPOSE 5000
ENTRYPOINT ["./boot.sh"]
```

Each line in the Dockerfile is a command. The `FROM` command specifies the base container image on which the new image will be built. The idea is that you start from an existing image, add or change some things, and you end up with a derived image. Images are referenced by a name and a tag, separated by a colon. The tag is used as a versioning mechanism, allowing a container image to provide more than one variant. The name of my chosen image is `python`, which is the official Docker image for Python. The tags for this image allow you to specify the interpreter version and base operating system. The `3.6-alpine` tag selects a Python 3.6 interpreter installed on Alpine Linux. The Alpine Linux distribution is often used instead of more popular ones such as Ubuntu because of its small size. You can see what tags are available for the Python image in the Python image repository (https://hub.docker.com/r/library/python/tags/).

The `RUN` command executes an arbitrary command in the context of the container. This would be similar to you typing the command in a shell prompt. The `adduser -D microblog` command creates a new user named `microblog`. Most container images have `root` as the default user, but it is not a good practice to run an application as root, so I create my own user.

The `WORKDIR` command sets a default directory where the application is going to be installed. When I created the `microblog` user above, a home directory was created, so now I'm making that directory the default. The new default directory is going to apply to any remaining commands in the Dockerfile, and also later when the container is executed.

The `COPY` command transfers files from your machine to the container file system. This command takes two or more arguments, the source and destination files or directories. The source file(s) must be relative to the directory where the Dockerfile is located. The destination can be an absolute path, or a path relative to the directory that was set in a previous `WORKDIR` command. In this first `COPY` command, I'm copying the *requirements.txt* file to the `microblog` user's home directory in the container file system.

Now that I have the *requirements.txt* file in the container, I can create a virtual environment, using the `RUN` command. First I create it, and then I install all the requirements in it. Because the requirements file contains only generic dependencies, I then explicitly install *gunicorn*, which I'm going to use as a web server. Alternatively, I could have added gunicorn to my *requirements.txt* file.

The three `COPY` commands that follow install the application in the container, by copying the *app* package, the *migrations* directory with the database migrations, and the *microblog.py* and *config.py* scripts from the top-level directory. I'm also copying a new file, *boot.sh* that I will discuss below.

The `RUN chmod` command ensures that this new *boot.sh* file is correctly set as an executable file. If you are in a Unix based file system and your source file is already marked as executable, then the copied file will also have the executable bit set. I added an explicit set because on Windows it is harder to set executable bits. If you are working on Mac OS X or Linux you probably don't need this statement, but it does not hurt to have it anyway.

The `ENV` command sets an environment variable inside the container. I need to set `FLASK_APP`, which is required to use the `flask` command.

The `RUN chown` command that follows sets the owner of all the directories and files that were stored in */home/microblog* as the new `microblog` user. Even though I created this user near the top of the Dockerfile, the default user for all the commands remained `root`, so all these files need to be switched to the `microblog` user so that this user can work with them when the container is started.

The `USER` command in the next line makes this new `microblog` user the default for any subsequent instructions, and also for when the container is started.

The `EXPOSE` command configures the port that this container will be using for its server. This is necessary so that Docker can configure the network in the container appropriately. I've chosen the standard Flask port 5000, but this can be any port.

Finally, the `ENTRYPOINT` command defines the default command that should be executed when the container is started. This is the command that will start the application web server. To keep things well organized, I decided to create a separate script for this, and this is the *boot.sh* file that I copied to the container earlier. Here are the contents of this script:

boot.sh: Docker container start-up script.

```
#!/bin/sh
source venv/bin/activate
flask db upgrade
flask translate compile
exec gunicorn -b :5000 --access-logfile - --error-logfile - microblog:app
```

This is a fairly standard start up script that is fairly similar to how the deployments in Chapter 17 (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux) and Chapter 18 (/post/the-flask-mega-tutorial-part-xviii-deployment-on-heroku) were started. I activate the virtual environment, upgrade the database though the migration framework, compile the language translations, and finally run the server with gunicorn.

Note the `exec` that precedes the gunicorn command. In a shell script, `exec` triggers the process running the script to be replaced with the command given, instead of starting it as a new process. This is important, because Docker associates the life of the container to the first process that runs on it. In cases like this one, where the start up process is not the main process of the container, you need to make sure that the main process takes the place of that first process to ensure that the container is not terminated early by Docker.

An interesting aspect of Docker is that anything that the container writes to `stdout` or `stderr` will be captured and stored as logs for the container. For that reason, the `--access-logfile` and `--error-logfile` are both configured with a `-`, which sends the log to standard output so that they are stored as logs by Docker.

With the Dockerfile created, I can now build a container image:

```
$ docker build -t microblog:latest .
```

The `-t` argument that I'm giving to the `docker build` command sets the name and tag for the new container image. The `.` indicates the base directory where the container is to be built. This is the directory where the *Dockerfile* is located. The build process is going to evaluate all the commands in the *Dockerfile* and create the image, which will be stored on your own machine.

You can obtain a list of the images that you have locally with the `docker images` command:

```
$ docker images
REPOSITORY    TAG          IMAGE ID       CREATED              SIZE
microblog     latest       54a47d0c27cf   About a minute ago   216MB
python        3.6-alpine   a6beab4fa70b   3 months ago         88.7MB
```

This listing will include your new image, and also the base image on which it was built. Any time you make changes to the application, you can update the container image by running the build command again.

Starting a Container

With an image already created, you can now run the container version of the application. This is done with the `docker run` command, which usually takes a large number of arguments. I'm going to start by showing you a basic example:

```
$ docker run --name microblog -d -p 8000:5000 --rm microblog:latest
021da2e1e0d390320248abf97dfbbe7b27c70fefed113d5a41bb67a68522e91c
```

The `--name` option provides a name for the new container. The `-d` option tells Docker to run the container in the background. Without `-d` the container runs as a foreground application, blocking your command prompt. The `-p` option maps container ports to host ports. The first port is the port on the host computer, and the one on the right is the port inside the container. The above example exposes port 5000 in the container on port 8000 in the host, so you will access the application on 8000, even though internally the container is using 5000. The `--rm` option will delete the container once it is terminated. While this isn't required, containers that finish or are interrupted are usually not needed anymore, so they can be automatically deleted. The last argument is the container image name and tag to use for the container. After you run the above command, you can access the application at *http://localhost:8000*.

The output of `docker run` is the ID assigned to the new container. This is a long hexadecimal string, that you can use whenever you need to refer to the container in subsequent commands. In fact, only the first few characters are necessary, enough to make the ID unique.

If you want to see what containers are running, you can use the `docker ps` command:

```
$ docker ps
CONTAINER ID  IMAGE             COMMAND      PORTS                   NAMES
021da2e1e0d3  microblog:latest  "./boot.sh"  0.0.0.0:8000->5000/tcp  microblog
```

You can see that even the `docker ps` command shortens container IDs. If you now want to stop the container, you can use `docker stop`:

```
$ docker stop 021da2e1e0d3
021da2e1e0d3
```

If you recall, there are a number of options in the application's configuration that are sourced from environment variables. For example, the Flask secret key, database URL and email server options are all imported from environment variables. In the `docker run` example above I have not worried about those, so all those configuration options are going to use defaults.

In a more realistic example, you will be setting those environment variables inside the container. You saw in the previous section that the `ENV` command in the *Dockerfile* sets environment variables, and it is a handy option for variables that are going to be static. For variables that depend on the installation, however, it isn't convenient to have them as part of the build process, because you want to have a container image that is fairly portable. If you want to give your application to another person as a container image, you would want that person to be able to use it as is, and not have to rebuild it with different variables.

So build-time environment variables can be useful, but there is also a need to have run-time environment variables that can be set via the `docker run` command, and for these variables, the `-e` option can be used. The following example sets a secret key and sends email through a gmail account:

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    microblog:latest
```

It is not uncommon for `docker run` command lines to be extremely long due to having many environment variable definitions.

Using Third-Party "Containerized" Services

The container version of Microblog is looking good, but I haven't really thought much about storage yet. In fact, since I haven't set a `DATABASE_URL` environment variable, the application is using the default SQLite database, which is supported by a file on disk. What do you think is going to happen to that SQLite file when you stop and delete the container? The file is going to disappear!

The file system in a container is *ephemeral*, meaning that it goes away when the container goes away. You can write data to the file system, and the data is going to be there if the container needs to read it, but if for any reason you need to recycle your container and replace it with a new one, any data that the application saved to disk is going to be lost forever.

A good design strategy for a container application is to make the application containers *stateless*. If you have a container that has application code and no data, you can throw it away and replace it with a new one without any problems, the container becomes truly disposable, which is great in terms of simplifying the deployment of upgrades.

But of course, this means that the data must be put somewhere outside of the application container. This is where the fantastic Docker ecosystem comes into play. The Docker Container Registry contains a large variety of container images. I have already told you about the Python container image, which I'm using as a base image for my Microblog container. In addition to that, Docker maintains images for many other languages, databases and other services in the Docker registry and if that isn't enough, the registry also allows companies to publish container images for their products, and also regular users like you or me to publish your own images. That means that the effort to install third party services is reduced to finding an appropriate image in the registry, and starting it with a `docker run` command with proper arguments.

So what I'm going to do now is create two additional containers, one for a MySQL database, and another one for the Elasticsearch service, and then I'm going to make the command line that starts the Microblog container even longer with options that enable it to access these two new containers.

Adding a MySQL Container

Like many other products and services, MySQL has public container images available on the Docker registry. Like my own Microblog container, MySQL relies on environment variables that need to be passed to `docker run`. These configure passwords, database names etc. While there are many MySQL images in the registry, I decided to use one that is officially maintained by the MySQL team. You can find detailed information about the MySQL container image in its registry page: *https://hub.docker.com/r/mysql/mysql-server/*.

If you remember the laborious process to set up MySQL in Chapter 17 (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux), you are going to appreciate Docker when you see how easy it is to deploy MySQL. Here is the `docker run` command that starts a MySQL server:

```
$ docker run --name mysql -d -e MYSQL_RANDOM_ROOT_PASSWORD=yes \
    -e MYSQL_DATABASE=microblog -e MYSQL_USER=microblog \
    -e MYSQL_PASSWORD=<database-password> \
    mysql/mysql-server:5.7
```

That is it! On any machine that you have Docker installed, you can run the above command and you'll get a fully installed MySQL server with a randomly generated root password, a brand new database called `microblog`, and a user with the same name that is configured with full permissions to access the database. Note that you will need to enter a proper password as the value for the `MYSQL_PASSWORD` environment variable.

Now on the application side, I need to add a MySQL client package, like I did for the traditional deployment on Ubuntu. I'm going to use `pymysql` once again, which I can add to the *Dockerfile*:

Dockerfile: Add pymysql to Dockerfile.

```
# ...
RUN venv/bin/pip install gunicorn pymysql
# ...
```

Any time a change is made to the application or the *Dockerfile*, the container image needs to be rebuilt:

```
$ docker build -t microblog:latest .
```

Any now I can start Microblog again, but this time with a link to the database container so that both can communicate through the network:

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    --link mysql:dbserver \
    -e DATABASE_URL=mysql+pymysql://microblog:<database-password>@dbserver/microblog \
    microblog:latest
```

The `--link` option tells Docker to make another container accessible to this one. The argument contains two names separated by a colon. The first part is the name or ID of the container to link, in this case the one named `mysql` that I created above. The second part defines a hostname that can be used in this container to refer to the linked one. Here I'm using `dbserver` as generic name that represents the database server.

With the link between the two containers established, I can set the `DATABASE_URL` environment variable so that SQLAlchemy is directed to use the MySQL database in the other container. The database URL is going to use `dbserver` as the database hostname,

`microblog` as the database name and user, and the password that you selected when you started MySQL.

One thing I noticed when I was experimenting with the MySQL container is that it takes a few seconds for this container to be fully running and ready to accept database connections. If you start the MySQL container and then start the application container immediately after, when the *boot.sh* script tries to run `flask db upgrade` it may fail due to the database not being ready to accept connections. To make my solution more robust, I decided to add a retry loop in *boot.sh*:

boot.sh: Retry database connection.

```
#!/bin/sh
source venv/bin/activate
while true; do
    flask db upgrade
    if [[ "$?" == "0" ]]; then
        break
    fi
    echo Upgrade command failed, retrying in 5 secs...
    sleep 5
done
flask translate compile
exec gunicorn -b :5000 --access-logfile - --error-logfile - microblog:app
```

This loop checks the exit code of the `flask db upgrade` command, and if it is non-zero it assumes that something went wrong, so it waits five seconds and then retries.

Adding a Elasticsearch Container

The Elasticsearch documentation for Docker (https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html) shows how to run the service as a single-node for development, and as a two-node production-ready deployment. For now I'm going to go with the single-node option and use the "oss" image, which only has the open source engine. The container is started with the following command:

```
$ docker run --name elasticsearch -d -p 9200:9200 -p 9300:9300 --rm \
    -e "discovery.type=single-node" \
    docker.elastic.co/elasticsearch/elasticsearch-oss:7.6.2
```

This `docker run` command has many similarities with the ones I've used for Microblog and MySQL, but there are a couple of interesting differences. First, there are two `-p` options, which means that this container is going to listen on two ports instead of just one. Both ports 9200 and 9300 are mapped to the same ports in the host machine.

The other difference is in the syntax used to refer to the container image. For the images that I've been building locally, the syntax was `<name>:<tag>`. The MySQL container uses a slightly more complete syntax with the format `<account>/<name>:<tag>`, which is appropriate to reference container images on the Docker registry. The Elasticsearch image that I'm using follows the pattern `<registry>/<account>/<name>:<tag>`, which includes the address of the registry as the first component. This syntax is used for images that are not hosted in the Docker registry. In this case Elasticsearch runs their own container registry service at *docker.elastic.co* instead of using the main registry maintained by Docker.

So now that I have the Elasticsearch service up and running, I can modify the start command for my Microblog container to create a link to it and set the Elasticsearch service URL:

```
$ docker run --name microblog -d -p 8000:5000 --rm -e SECRET_KEY=my-secret-key \
    -e MAIL_SERVER=smtp.googlemail.com -e MAIL_PORT=587 -e MAIL_USE_TLS=true \
    -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
    --link mysql:dbserver \
    -e DATABASE_URL=mysql+pymysql://microblog:<database-password>@dbserver/microblog \
    --link elasticsearch:elasticsearch \
    -e ELASTICSEARCH_URL=http://elasticsearch:9200 \
    microblog:latest
```

Before you run this command, remember to stop your previous Microblog container if you still have it running. Also be careful in setting the correct passwords for the database and the Elasticsearch service in the proper places in the command.

Now you should be able to visit *http://localhost:8000* and use the search feature. If you experience any errors, you can troubleshoot them by looking at the container logs. You'll most likely want to see logs for the Microblog container, where any Python stack traces will appear:

```
$ docker logs microblog
```

The Docker Container Registry

So now I have the complete application up and running on Docker, using three containers, two of which come from publicly available third-party images. If you would like to make your own container images available to others, then you have to *push* them to the Docker registry from where anybody can obtain images.

To have access to the Docker registry you need to go to *https://hub.docker.com* and create an account for yourself. Make sure you pick a username that you like, because that is going to be used in all the images that you publish.

To be able to access your account from the command line, you need to log in with the `docker login` command:

```
$ docker login
```

If you've been following my instructions, you now have an image called `microblog:latest` stored locally on your computer. To be able to push this image to the Docker registry, it needs to be renamed to include the account, like the image from MySQL. This is done with the `docker tag` command:

```
$ docker tag microblog:latest <your-docker-registry-account>/microblog:latest
```

If you list your images again with `docker images` you are now going to see two entries for Microblog, the original one with the `microblog:latest` name, and a new one that also includes your account name. These are really two alias for the same image.

To publish your image to the Docker registry, use the `docker push` command:

```
$ docker push <your-docker-registry-account>/microblog:latest
```

Now your image is publicly available and you can document how to install it and run from the Docker registry in the same way MySQL and others do.

Deployment of Containerized Applications

One of the best things about having your application running in Docker containers is that once you have the containers tested locally, you can take them to any platform that offers Docker support. For example, you could use the same servers I recommended in Chapter 17 (/post/the-flask-mega-tutorial-part-xvii-deployment-on-linux) from Digital Ocean, Linode or Amazon Lightsail. Even the cheapest offering from these providers is sufficient to run Docker with a handful of containers.

The Amazon Container Service (ECS) (https://aws.amazon.com/ecs/) gives you the ability to create a cluster of container hosts on which to run your containers, in a fully integrated AWS environment, with support for scaling and load balancing, plus the option to use a private container registry for your container images.

Finally, a container orchestration platform such as Kubernetes (https://kubernetes.io/) provides an even greater level of automation and convenience, by allowing you to describe your multi-container deployments in simple text files in YAML format, with load balancing, scaling, secure management of secrets and rolling upgrades and rollbacks.