Chapter 7

Docker Orchestration and Service Discovery

Orchestration is a pretty loosely defined term. It's broadly the process of automated configuration, coordination, and management of services. In the Docker world we use it to describe the set of practices around managing applications running in multiple Docker containers and potentially across multiple Docker hosts. Native orchestration is in its infancy in the Docker community but an exciting ecosystem of tools is being integrated and developed.

In the current ecosystem there are a variety of tools being built and integrated with Docker. Some of these tools are simply designed to elegantly "wire" together multiple containers and build application stacks using simple composition. Other tools provide larger scale coordination between multiple Docker hosts as well as complex service discovery, scheduling and execution capabilities.

Each of these areas really deserves its own book but we've focused on a few useful tools that give you some insight into what you can achieve when orchestrating containers. They provide some useful building blocks upon which you can grow your Docker-enabled environment.

In this chapter we will focus on three areas:

- Simple container orchestration. Here we'll look at Docker Compose. Docker Compose (previously Fig) is an open source Docker orchestration tool developed by the Orchard team and then acquired by Docker Inc in 2014. It's written in Python and licensed with the Apache 2.0 license.

- Distributed service discovery. Here we'll introduce Consul. Consul is also open source, licensed with the Mozilla Public License 2.0, and written in Go. It provides distributed, highly available service discovery. We're going to look at how you might use Consul and Docker to manage application service discovery.
- Orchestration and clustering of Docker. Here we're looking at Swarm. Swarm is open source, licensed with the Apache 2.0 license. It's written in Go and developed by the Docker Inc team. As of Docker 1.12 the Docker Engine now has a Swarm-mode built in and we'll be covering that later in this chapter.

TIP We'll also talk about many of the other orchestration tools available to you later in this chapter.

Docker Compose

Now let's get familiar with Docker Compose. With Docker Compose, we define a set of containers to boot up, and their runtime properties, all defined in a YAML file. Docker Compose calls each of these containers "services" which it defines as:

> A container that interacts with other containers in some way and that
> has specific runtime properties.

We're going to take you through installing Docker Compose and then using it to build a simple, multi-container application stack.

Installing Docker Compose

We start by installing Docker Compose. Docker Compose is currently available for Linux, Windows, and OS X. It can be installed directly as a binary and via Docker for Mac or Windows.

To install Docker Compose on Linux we can grab the Docker Compose binary from GitHub and make it executable. Like Docker, Docker Compose is currently only supported on 64-bit Linux installations. We'll need the `curl` command available to do this.

Listing 7.1: Installing Docker Compose on Linux

```
$ sudo curl -L "https://github.com/docker/compose/releases/
    download/$(curl -sL https://api.github.com/repos/docker/
    compose/releases/latest | grep tag_name | cut -d'"' -f 4)/
    docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/
    docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

This will download the `docker-compose` binary from GitHub and install it into the `/usr/local/bin` directory. We've also used the `chmod` command to make the `docker-compose` binary executable so we can run it.

If we're on OS X Docker Compose comes bundled with Docker for Mac or we can install it like so:

Listing 7.2: Installing Docker Compose on OS X

```
$ sudo bash -c "curl -L https://github.com/docker/compose/
    releases/download/1.17.1/docker-compose-Darwin-x86_64 > /usr/
    local/bin/docker-compose"
$ sudo chmod +x /usr/local/bin/docker-compose
```

TIP Replace the 1.17.1 with the release number of the current Docker Compose release.

If we're on Windows Docker Compose comes bundled inside Docker for Windows.

Once you have installed the `docker-compose` binary you can test it's working using the `docker-compose` command with the `--version` flag:

Listing 7.3: Testing Docker Compose is working

```
$ docker-compose --version
docker-compose version 1.17.1, build f3628c7
```

NOTE If you're upgrading from a pre-1.3.0 release you'll need to migrate any existing container to the new 1.3.0 format using the `docker-compose migrate-to-labels` command.

Getting our sample application

To demonstrate how Compose works we're going to use a sample Python Flask application that combines two containers:

- An application container running our sample Python application.
- A container running the Redis database.

Let's start with building our sample application. Firstly, we create a directory and a `Dockerfile`.

Listing 7.4: Creating the composeapp directory

```
$ mkdir composeapp
$ cd composeapp
```

Here we've created a directory to hold our sample application, which we're calling `composeapp`.

Next, we need to add our application code. Let's create a file called `app.py` in the `composeapp` directory and add the following Python code to it.

Listing 7.5: The app.py file

```
from flask import Flask
from redis import Redis
import os

app = Flask(__name__)
redis = Redis(host="redis", port=6379)

@app.route('/')
def hello():
    redis.incr('hits')
    return 'Hello Docker Book reader! I have been seen {0} times'
   .format(redis.get('hits'))

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

TIP You can find this source code on GitHub.

This simple Flask application tracks a counter stored in Redis. The counter is incremented each time the root URL, `/`, is hit.

We also need to create a `requirements.txt` file to store our application's dependencies. Let's create that file now and add the following dependencies.

Listing 7.6: The requirements.txt file

```
flask
redis
```

Now let's populate our Compose `Dockerfile`.

Listing 7.7: The composeapp Dockerfile

```
# Compose Sample application image
FROM python:2.7
LABEL maintainer="james@example.com"
ENV REFRESHED_AT 2016-06-01

ADD . /composeapp

WORKDIR /composeapp

RUN pip install -r requirements.txt
```

Our `Dockerfile` is simple. It is based on the `python:2.7` image. We add our `app.py` and `requirements.txt` files into a directory in the image called `/composeapp`. The `Dockerfile` then sets the working directory to `/composeapp` and runs the `pip` installation process to install our application's dependencies: `flask` and `redis`.

Let's build that image now using the `docker build` command.

Listing 7.8: Building the composeapp application

```
$ sudo docker build -t jamtur01/composeapp .
Sending build context to Docker daemon  16.9 kB
Sending build context to Docker daemon
Step 0 : FROM python:2.7
 ---> 1c8df2f0c10b
Step 1 : LABEL maintainer="james@example.com"
 ---> Using cache
 ---> aa564fe8be5a
Step 2 : ADD . /composeapp
 ---> c33aa147e19f
Removing intermediate container 0097bc79d37b
Step 3 : WORKDIR /composeapp
 ---> Running in 76e5ee8544b3
 ---> d9da3105746d
Removing intermediate container 76e5ee8544b3
Step 4 : RUN pip install -r requirements.txt
 ---> Running in e71d4bb33fd2
Downloading/unpacking flask (from -r requirements.txt (line 1))
. . .
Successfully installed flask redis Werkzeug Jinja2 itsdangerous
   markupsafe
Cleaning up...
 ---> bf0fe6a69835
Removing intermediate container e71d4bb33fd2
Successfully built bf0fe6a69835
```

This will build a new image called `jamtur01/composeapp` containing our sample application and its required dependencies. We can now use Compose to deploy our application.

NOTE We'll be using a Redis container created from the default Redis image on the Docker Hub so we don't need to build or customize that.

The `docker-compose.yml` file

Now we've got our application image built we can configure Compose to create both the services we require. With Compose, we define a set of services (in the form of Docker containers) to launch. We also define the runtime properties we want these services to start with, much as you would do with the `docker run` command. We define all of this in a YAML file. We then run the `docker-compose up` command. Compose launches the containers, executes the appropriate runtime configuration, and multiplexes the log output together for us.

Let's create a `docker-compose.yml` file for our application inside our `composeapp` directory.

Listing 7.9: Creating the docker-compose.yml file

```
$ touch docker-compose.yml
```

Let's populate our `docker-compose.yml` file. The `docker-compose.yml` file is a YAML file that contains instructions for running one or more Docker containers. Let's look at the instructions for our example application.

Listing 7.10: The docker-compose.yml file

```
version: '3'
services:
  web:
    image: jamtur01/composeapp
    command: python app.py
    ports:
     - "5000:5000"
    volumes:
     - .:/composeapp
  redis:
    image: redis
```

Each service we wish to launch is specified as a YAML hash inside a hash called `services`. Here our two services are: `web` and `redis`.

TIP The `version` tag tells Docker Compose what configuration version to use. The Docker Compose API has evolved over the years and each change has been marked by incrementing the version.

For our `web` service we've specified some runtime options. Firstly, we've specified the `image` we're using: the `jamtur01/composeapp` image. Compose can also build Docker images. You can use the `build` instruction and provide the path to a `Dockerfile` to have Compose build an image and then create services from it.

Listing 7.11: An example of the build instruction

```
web:
  build: /home/james/composeapp
. . .
```

This `build` instruction would build a Docker image from a `Dockerfile` found in the `/home/james/composeapp` directory.

We've also specified the `command` to run when launching the service. Next we specify the `ports` and `volumes` as a list of the port mappings and volumes we want for our service. We've specified that we're mapping port 5000 inside our service to port 5000 on the host. We're also creating `/composeapp` as a volume.

If we were executing the same configuration on the command line using `docker run` we'd do it like so:

Listing 7.12: The docker run equivalent command

```
$ sudo docker run -d -p 5000:5000 -v .:/composeapp \
--name jamtur01/composeapp python app.py
```

Next we've specified another service called `redis`. For this service we're not setting any runtime defaults at all. We're just going to use the base `redis` image. By default, containers run from this image launches a Redis database on the standard port. So we don't need to configure or customize it.

TIP You can see a full list of the available instructions you can use in the `docker-compose.yml` file in the Docker Compose documentation.

Running Compose

Once we've specified our services in docker-compose.yml we use the docker-compose up command to execute them both.

Listing 7.13: Running docker-compose up with our sample application

```
$ cd composeapp
$ sudo docker-compose up
Creating network "composeapp_default" with the default driver
Recreating composeapp_web_1 ...
Recreating composeapp_web_1
Recreating composeapp_redis_1 ...
Recreating composeapp_web_1 ... done
Attaching to composeapp_redis_1, composeapp_web_1
web_1    |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to
   quit)
. . .
```

TIP You must be inside the directory with the docker-compose.yml file in order to execute most Compose commands.

Compose has created two new services: composeapp_redis_1 and composeapp_web_1 . So where did these names come from? Well, to ensure our services are unique, Compose has prefixed and suffixed the names specified in the docker-compose.yml file with the directory and a number respectively.

Compose then attaches to the logs of each service, each line of log output is prefixed with the abbreviated name of the service it comes from, and outputs them multiplexed:

Listing 7.14: Compose service log output

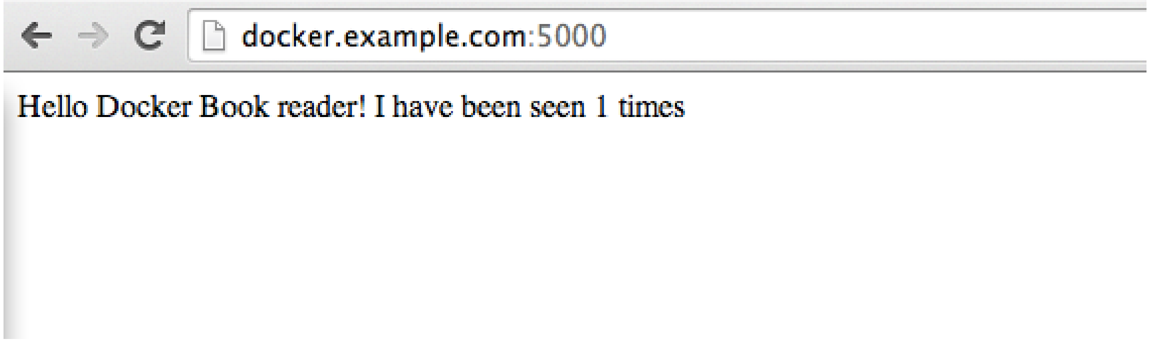
```
redis_1  | 1:M 05 Aug 17:49:17.839 * The server is now ready to
    accept connections on port 6379
```

The services (and Compose) are being run interactively. That means if you use `Ctrl-C` or the like to cancel Compose then it'll stop the running services. We could also run Compose with `-d` flag to run our services daemonized (similar to the `docker run -d` flag).

Listing 7.15: Running Compose daemonized

```
$ sudo docker-compose up -d
```

Let's look at the sample application that's now running on the host. The application is bound to all interfaces on the Docker host on port 5000. So we can browse to that site on the host's IP address or via `localhost`.



Hello Docker Book reader! I have been seen 1 times

Figure 7.1: Sample Compose application.

We see a message displaying the current counter value. We can increment the counter by refreshing the site. Each refresh stores the increment in Redis. The

Redis update is done via the link between the Docker containers controlled by Compose.

TIP By default, Compose tries to connect to a local Docker daemon but it'll also honor the `DOCKER_HOST` environment variable to connect to a remote Docker host.

Using Compose

Now let's explore some of Compose's other options. Firstly, let's use `Ctrl-C` to cancel our running services and then restart them as daemonized services.

Press `Ctrl-C` inside the `composeapp` directory and then re-run the `docker-compose up` command, this time with the `-d` flag.

Listing 7.16: Restarting Compose as daemonized

```
$ sudo docker-compose up -d
Starting composeapp_web_1 ...
Starting composeapp_redis_1 ...
Starting composeapp_redis_1
Starting composeapp_web_1 ... done
$ . . .
```

We see that Compose has recreated our services, launched them and returned to the command line.

Our Compose-managed services are now running daemonized on the host. Let's look at them now using the `docker-compose ps` command; a close cousin of the `docker ps` command.

TIP You can get help on Compose commands by running `docker-compose help` and the command you wish to get help on, for example `docker-compose help ps`.

The `docker-compose ps` command lists all of the currently running services from our local `docker-compose.yml` file.

Listing 7.17: Running the docker-compose ps command

```
$ cd composeapp
$ sudo docker-compose ps
Name                   Command                         State Ports
-----------------------------------------------------------------
   -
composeapp_redis_1 docker-entrypoint.sh redis  Up    6379/tcp
composeapp_web_1   python app.py               Up    0.0.0.0:5000
   ->5000/tcp
```

This shows some basic information about our running Compose services. The name of each service, what command we used to start the service, and the ports that are mapped on each service.

We can also drill down further using the `docker-compose logs` command to show us the log events from our services.

Listing 7.18: Showing a Compose services logs

```
$ sudo docker-compose logs
docker-compose logs
Attaching to composeapp_redis_1, composeapp_web_1
redis_1 |  (    '      ,       .-`  | `,    )     Running in
   stand alone mode
redis_1 |  |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
redis_1 |  |    `-._   `._    /     _.-'    |     PID: 1
. . .
```

This will tail the log files of your services, much as the `tail -f` command. Like the `tail -f` command you'll need to use `Ctrl-C` or the like to exit from it.

We can also stop our running services with the `docker-compose stop` command.

Listing 7.19: Stopping running services

```
$ sudo docker-compose stop
Stopping composeapp_web_1...
Stopping composeapp_redis_1...
```

This will stop both services. If the services don't stop you can use the `docker-compose kill` command to force kill the services.

We can verify this with the `docker-compose ps` command again.

Listing 7.20: Verifying our Compose services have been stopped

```
$ sudo docker-compose ps
Name                 Command          State    Ports
-------------------------------------------------
composeapp_redis_1 redis-server    Exit 0
composeapp_web_1   python app.py   Exit 0
```

If you've stopped services using `docker-compose stop` or `docker-compose kill` you can also restart them again with the `docker-compose start` command. This is much like using the `docker start` command and will restart these services.

Finally, we can remove services using the `docker-compose rm` command.

Listing 7.21: Removing Compose services

```
$ sudo docker-compose rm
Going to remove composeapp_redis_1, composeapp_web_1
Are you sure? [yN] y
Removing composeapp_redis_1...
Removing composeapp_web_1...
```

You'll be prompted to confirm you wish to remove the services and then both services will be deleted. The `docker-compose ps` command will now show no running or stopped services.

Listing 7.22: Showing no Compose services

```
$ sudo docker-compose ps
Name   Command   State   Ports
------------------------------
```

Compose in summary

Now in one file we have a simple Python-Redis stack built! You can see how much easier this can make constructing applications from multiple Docker containers. It's especially a great tool for building local development stacks. This, however, just scratches the surface of what you can do with Compose. There are some more examples using Rails, Django and Wordpress on the Compose website that introduce some more advanced concepts.

TIP You can see a full command line reference in the Docker Compose Reference documentation.

Consul, Service Discovery and Docker

Service discovery is the mechanism by which distributed applications manage their relationships. A distributed application is usually made up of multiple components. These components can be located together locally or distributed across data centers or geographic regions. Each of these components usually provides or consumes services to or from other components.

Service discovery allows these components to find each other when they want to interact. Due to the distributed nature of these applications, service discovery