Docker Compose

Docker Compose is an orchestration tool for Docker that allows you to define a set of containers and their interdependencies in the form of a YAML file. You can then use Docker Compose to bring up part or the whole of your application stack, as well as track application output, etc. Setting up the Docker toolbox on Mac OSX or Windows is fairly easy. Head over to https://www.docker.com/product... to download the installer for your platform. On Linux, you simply install Docker and Docker Compose using your native packaging tools.

An example application

For the sake of this exercise, let's look at a simple Python app that uses a web framework, with Nginx acting as a reverse proxy sitting in front. Our aim is to run this application stack in Docker using the Docker Compose tool. This is a simple "Hello World" application. Let's start off with just the application. This is a single Python script that uses the Pyramid framework. Let's create a directory and add the application code there. Here's what the directory structure looks like:

```
helloworld
└── app.py
```

I have created a directory called helloworld, in which there's a single Python script called app.py. *helloworld* here represents my checked out code tree.

This makes up the contents of my example application app.py:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.response import Response

def hello_world(request):
 print('Incoming request')
 return Response('<body>

<h1>Hello World!</h1>

</body>')

if __name__ == '__main__':
 config = Configurator()
 config.add_route('hello', '/')
 config.add_view(hello_world, route_name='hello')
 app = config.make_wsgi_app()
 server = make_server('0.0.0.0', 5000, app)
 server.serve_forever()
```

It simply listens on port 5000 and responds to all HTTP requests with "Hello World!" If you wanted to run this natively on your Windows or Mac machine, you would need to

install Python, and then the Pyramid module, along with all dependencies. Let's run this under Docker instead.
It's always a good idea to keep the infrastructure code separate from the application code. Let's create another directory here called compose and add files here to containerize this application.

Here's what my file structure now looks like. The text in bold represents new files and folders:

```
├── compose
│  └── docker-compose.yml
└── helloworld
└── app.py
```

This makes up the contents of the docker-compose.yml:

```
version: '2'
services:
 helloworld:
 image: helloworld:1.0
 ports:
 - "5000:5000"
 volumes:
 - ../helloworld:/code
```

Let's break this down to understand what our docker-compose definition means. We start off with the line "version: '2'", which tells Docker Compose we are using the new Docker Compose syntax.

We define a single service called helloworld, which runs from an image called helloworld:1.0. (This of course doesn't exist. We'll come to that later.) It exposes a single port 5000 on the docker host that maps to port 5000 inside the container. It maps the helloworld directory that holds our app.py to /code inside the container.

Now if you tried to run this as-is, using "docker-compose up", docker could complain that it couldn't find helloworld:1.0. That's because it's looking on the docker hub for a container image called helloworld:1.0. We haven't created it yet. So now, let's add the recipe to create this container image. Here's what the file tree now looks like:

```
├── compose
│  ├── docker-compose.yml
│  └── helloworld
│  └── Dockerfile
└── helloworld
└── app.py
```

We've added a new directory called helloworld inside the compose directory and added a file called Dockerfile there. The following makes up the contents of Dockerfile:

```
FROM ubuntu:14.04
MAINTAINER Your Name <your-email@somedomain.com>

ENV HOME /root
ENV DEBIAN_FRONTEND noninteractive

RUN apt-get -yqq update
RUN apt-get install -yqq python python-dev python-pip
RUN pip install pyramid

WORKDIR /code
CMD ["python", "app.py"]
```

This isn't a very optimal Dockerfile, but it will do for us. It's derived from Ubuntu 14.04, and it contains the environment needed to run our Python app. It has the Python interpreter and the Pyramid Python module installed. It also defines /code as the working directory and defines an entry point to the container, namely: "python app.py". It assumes that /code will contain a file called app.py that will then be executed by the Python interpreter.

We'll now change our docker-compose.yml to add a single line that tells Docker Compose to build the application container for us if needed. This is what it now looks like:

```
version: '2'
services:
 helloworld:
 build: ./helloworld
 image: helloworld:1.0
 ports:
 - "5000:5000"
 volumes:
 - ../helloworld:/code
```

We've added a single line "build: ./helloworld" to the helloworld service. It instructs Docker Compose to enter the compose/helloworld directory, run a docker build there, and tag the resultant image as helloworld:1.0. It's very concise. You'll notice that we haven't added the application app.py into the container. Instead, we're actually mapping the helloworld directory that contains app.py to /code inside the container, and asking docker to run it from there. What that means is that you are free to modify the code using the developer IDE or editor of your choice on your host platform, and all you need to do is restart the docker container to run new code. So let's fire this up for the first time.

Before we start, let's find out the IP address of the docker machine so we can connect to our application when it's up. To do that, type "docker-machine ls":

```
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
default * virtualbox Running tcp://192.168.99.100:2376 v1.11.0
```

This tells us that the Docker VM is running on 192.168.99.100.

Inside the Docker terminal, navigate to the compose directory and run:

```
$ docker-compose up
```

We are running docker-compose in the foreground. You should see something similar to this:

```
$ docker-compose up
Building helloworld
Step 1 : FROM ubuntu:14.04
 ---> b72889fa879c
Step 2 : MAINTAINER Your Name <your-email@somedomain.com>
 ---> Running in d40e1c4e45d8
 ---> f0d1fe4ec198
Removing intermediate container d40e1c4e45d8
Step 3 : ENV HOME /root
 ---> Running in d6808a44f46f
 ---> b382d600d584
Removing intermediate container d6808a44f46f
Step 4 : ENV DEBIAN_FRONTEND noninteractive
 ---> Running in d25def6b366b
 ---> b5d310716d1f
Removing intermediate container d25def6b366b
Step 5 : RUN apt-get -yqq update
 ---> Running in 198faaac5c1b
 ---> fb86cbdcbe2e
Removing intermediate container 198faaac5c1b
Step 6 : RUN apt-get install -yqq python python-dev python-pip
 ---> Running in 0ce70f832459
Extracting templates from packages: 100%
Preconfiguring packages ...
Selecting previously unselected package libasan0:amd64.

...

 ---> 4a9ac1adb7a2
Removing intermediate container 0ce70f832459
Step 7 : RUN pip install pyramid
 ---> Running in 0907fb066fce
Downloading/unpacking pyramid

...

Cleaning up...
 ---> 48ef0b2c3674
Removing intermediate container 0907fb066fce
Step 8 : WORKDIR /code
 ---> Running in 5c691ab4d6ec
 ---> 860dd36ee7f6
Removing intermediate container 5c691ab4d6ec
Step 9 : CMD python app.py
 ---> Running in 8230b8989501
 ---> 7b6d773a2eae
```

```
Removing intermediate container 8230b8989501
Successfully built 7b6d773a2eae
Creating compose_helloworld_1
Attaching to compose_helloworld_1
```

… And it stays stuck there. This is now the application running inside Docker. Don't be overwhelmed by what you see when you run it for the first time. The long output is Docker attempting to build and tag the container image for you since it doesn't already exist. After it's built once, it will reuse this image the next time you run it.

Now open up a browser and try navigating to 192.168.99.100:5000 greeted by a page that says Hello World!.

So, that's our first application running under Docker. To stop the application, simply type Ctrl-C at the terminal prompt and Docker Compose will stop the container and exit. You can go ahead and change the code in the helloworld directory, add new code or modify existing code, and test it out using "docker-compose up" again.

To run it in the background: docker-compose up -d.
To tail the container standard output: docker-compose logs -f.

This is a minimal application. Let's now add a commodity container to the mix. Let's pull in Nginx to act as the front-end to our application. Here, Nginx listens on port 80 and forwards all requests to helloworld:5000. This isn't useful in itself, but helps us demonstrate a few key concepts, primarily inter-container communication. It also demonstrates the container dependency that Docker Compose can handle for you, ensuring that your application comes up before Nginx comes up, so it can then forward connections to the application correctly. Here's the new docker-compose.yml file:

```
<a href="http://192.168.99.100:5000.</p><p>We're"
class="redactor-autoparser-object">version: '2'
services:
 helloworld:
 build: ./helloworld
 image: helloworld:1.0
 volumes:
 - ../helloworld:/code
 - ./logs:/var/log
 - ./config:/etc/appconfig

 nginx:
 image: nginx:alpine
 ports:
 - "80:80"
 volumes:
 - ./nginx/conf.d:/etc/nginx/conf.d
 links:
 - helloworld
</a>
```

As you can see, we’ve added a new service here called nginx. We’ve also removed the port’s entry for helloworld, and instead we’ve added a link to it from nginx. What this means is that the nginx service can now communicate with the helloworld service using the name helloworld. Then, we also map the new nginx/conf.d directory to /etc/nginx/conf.d inside the container. This is what the tree now looks like:

```
├── compose
│  ├── docker-compose.yml
│  ├── helloworld
│  │  └── Dockerfile
│  └── nginx
│  └── conf.d
│  └── helloworld.conf
└── helloworld
└── app.py
```

The following makes up the contents of compose/nginx/conf.d/helloworld.conf

```
<a href="http://192.168.99.100:5000.</p><p>We’re"
class="redactor-autoparser-object">server {

 listen 80;
 server_name helloworld.org;
 charset utf-8;

 location / {
 proxy_pass http://helloworld:5000;
 proxy_set_header Host $host;
 proxy_set_header X-Real-IP $remote_addr;
 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
 }
}
</a>
```

This tells nginx to listen on port 80 and forward all requests for / to helloworld:5000. Although port 5000 is no longer being forwarded to by Docker, it’s still exposed on the helloworld container and is accessible from all other containers on the machine. This is how the connections now work:

```
<a href="http://192.168.99.100:5000.</p><p>We're"
class="redactor-autoparser-object">
browser ->
192.168.99.100(docker machine) ->
nginx:80 ->
nginx-process ->
hellworld:5000
</a>
```

Commodity Containers and Docker Hub

The nginx container for this example comes from the official Nginx image on the Docker Hub. This version uses Alpine Linux as its base OS, instead of Ubuntu. Not only is the Alpine Linux version smaller in size, it also demonstrates one of the advantages of dockerization—running commodity containers without worrying about underlying distribution. I could swap it out for the Debian version tomorrow without breaking a sweat.

It's possible that your cloud application actually uses cloud services like Amazon's RDS for the database, or S3 for the object store, etc. You could of course let your local instance of the application talk to the services too, but the latency and the cost involved may beg for a more developer-friendly solution. The easy way out is to abstract the access to these services via some configuration and point the application to local containers that offer the same service instead. So instead of Amazon's RDS, spin up a MySQL container and let your application talk to that. For Amazon S3, use LeoFS or minio.io in containers, for example.

Container configuration

Unless you've created your own images for the commodity services, you might need to pass on configuration information in the form of files or environment variables. This can usually be expressed in the form of environment variables defined in the docker-compose.yml file, or as mapped directories inside the container for configuration files. We've already seen an example of overriding configuration in the nginx section of the docker-compose.yml file.

Managing data, configuration and logs

For a real-world application, it's very likely that you have some persistent storage in the form of RDBMS or NoSQL storage. This will typically store your application state. Keeping this data inside the commodity container would mean you couldn't really swap it out for a different version or entity later without losing your application data. That's where data volumes come in. Data volumes allow you to keep state separately in a different container volume. Here's a snippet from the official Docker Compose documentation about how to use data volumes:

```
<a href="http://192.168.99.100:5000.</p><p>We’re" class="redactor-autoparser-object">version: '2'
services:
 db:
 image: postgres
 volumes:
 - mydata:/var/lib/postgresql/data
 - ./logs:/var/log
volumes:
 mydata: {}
</a>
```

The volume is defined in the top level volumes section as mydata. It's then used in the volumes section of the db service and maps the mydata volume to /var/lib/postgesql/data, so that when the postgres container starts, it actually writes to a separate container volume named mydata.

While our example only mapped code into the application container, you could potentially get data out of the container just as easily. In our data volume example, we map a directory called logs to /var/log inside the postgres container. So all postgres logs should end up in the logs directory, which we could then analyze using our native Windows/Mac tools. The Docker toolbox maps volumes into the VM running the docker daemon using vboxfs, Virtualbox's shared filesystem implementation. It does so transparently, so it's easy to use without any extra setup.