Docker Compose Basics

Objective: Install Compose, unless already installed, explore the basics of using Docker Compose with containers.

Preparation: Make sure you have already installed both Docker Engine and Docker Compose. You don't need to install Python or Redis, as both are provided by Docker images.

Outcome: xx

Data Files: Ask Instructor

Step 1. Installing

1. If you're a Mac or Windows user, the best way to install Compose and keep it up-to-date is Docker for Mac and Windows. Docker for Mac and Windows will automatically install the latest version of Docker Engine for you. Check to see if docker-compose is installed already, by using the version flag, like this:

   ```
   $ docker-compose --version
   ```

2. Alternatively, you can use the usual commands to install or upgrade Compose:

   ```
   $ curl -L https://github.com/docker/compose/releases/do
   ```

```
wnload/1.14.0/docker-compose-`uname -s`-`uname -m` > /u
sr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

Compose file format compatibility matrix

Compose File Format	Docker Engine
3.3	17.06.0+
3.0 ; 3.1	1.13.0+
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+
1.0	1.9.1+

> *NOTE: Must use `sudo` and may have to elevate premissions, using `sudo -- su`. Remember, `docker` is root's password.*

Step 2. Setup

1. Create a directory for the project:

```
$ mkdir composetest
$ cd composetest
```

2. Create a file called `app.py` in your project directory and paste this in:

    ```
    from flask import Flask
    from redis import Redis

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

    @app.route('/')
    def hello():
        count = redis.incr('hits')
        return 'Hello World! I have been seen {} times.\n'.
    format(count)

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

3. Create another file called `requirements.txt` in your project directory and paste this in:

    ```
    flask
    redis
    ```

> *NOTE: These define the application’s dependencies.*

Step 3. Create a Dockerfile

In this step, you write a Dockerfile that builds a Docker image. The image contains all the dependencies the Python application requires, including Python itself.

1. In your project directory, create a file named Dockerfile and paste the following:

   ```
   FROM python:3.4-alpine
   ADD . /code
   WORKDIR /code
   RUN pip install -r requirements.txt
   CMD ["python", "app.py"]
   ```

This tells Docker to:

- Build an image starting with the Python 3.4 image.
- Add the current directory `.` into the path /code in the image.
- Set the working directory to `/code`.
- Install the Python dependencies.
- Set the default command for the container to `python` `app.py`.

Step 4. Define Services in Compose File

1. Create a file called `docker-compose.yml` in your project directory and paste the following:

```
version: 'X' //Use the chart above to make your selecti
on.
services:
  web:
    build: .
    ports:
     - 5000:5000
    volumes:
     - .:/code
  redis:
    image: "redis:alpine"
```

This Compose file defines two services, `web` and `redis`. The web service:

- Uses an image that’s built from the `Dockerfile` in the current directory.
- Forwards the exposed port 5000 on the container to port 5000 on the host machine.
- Mounts the project directory on the host to `/code` inside the container, allowing you to modify the code without having to rebuild the image.
- The `redis` service uses a public Redis image pulled from the Docker Hub registry.

Step 5: Build and run your app with

Compose

1. From your project directory, start up your application.

   ```
   $ docker-compose up
   ```

   Output:

   ```
   Pulling image redis...
   Building web...
   Starting composetest_redis_1...
   Starting composetest_web_1...
   redis_1 | [8] 02 Jan 18:43:35.576 # Server started, Red
   is version 2.8.3
   web_1   |  * Running on http://0.0.0.0:5000/
   web_1   |  * Restarting with stat
   ```

Compose pulls a Redis image, builds an image for your code, and start the services you defined.

2. Enter `http://0.0.0.0:5000/` in a browser to see the application running.

   > *NOTE: If you're using Docker on Linux natively, then the web app should now be listening on port 5000 on your Docker daemon host. If `http://0.0.0.0:5000` doesn't resolve, you*

> *can also try `http://localhost:5000`. If you're using Docker Machine on a Mac, use `docker-machine ip MACHINE_VM` to get the IP address of your Docker host. Then, open `http://MACHINE_VM_IP:5000` in a browser.*

You should see a message in your browser saying:

```
Hello World! I have been seen 1 times.
```

3. Refresh the page, and watch the number increment.

Step 6: Updating an App<br>

Because the application code is mounted into the container using a volume, you can make changes to its code and see the changes instantly, without having to rebuild the image.

1. Change the greeting in `app.py` and save it. For example:

```
return 'Hello from Docker! I have been seen {} times.\n
'.format(count)
```

2. Refresh the app in your browser. The greeting should be updated, and the counter should still be incrementing.

3. Also, try it from the terminal using `curl`, like this:

```
$ curl localhost:5000
```

4. Once you are done, before moving to the next step, be sure to bring Compose `down`.

Step 7: Experiment with some other commands

1. If you want to run your services in the background, you can pass the `-d` flag (for "detached" mode) to `docker-compose up` and use `docker-compose ps` to see what is currently running:

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

   Output:

   ```
   Starting composetest_redis_1...
   Starting composetest_web_1...
   ```

   ```
   $ docker-compose ps
   ```

   Output:

   ```
   Name                     Command                   State
          Ports
   ```

```
-------------------------------------------------------
------------
composetest_redis_1   /usr/local/bin/run         Up
composetest_web_1     /bin/sh -c python app.py   Up
5000->5000/tcp
```

2. The `docker-compose run` command allows you to run one-off commands for your services. For example, to see what environment variables are available to the web service:

   ```
   $ docker-compose run web env
   ```

   See `docker-compose --help` to see other available commands.

3. If you started Compose with docker-compose up -d, you'll probably want to stop your services once you've finished with them:

   ```
   $ docker-compose stop
   ```

4. You can bring everything down, removing the containers entirely, with the down command. Pass `--volumes` to also remove the data volume used by the Redis container:

   ```
   $ docker-compose down --volumes
   ```

At this point, you have seen the basics of how Compose works.

Conclusion

This is just the beginning, in Part B you will learn evern more about Compose. In this lab however, we reviewed the basics, deployed an application, updated the application, and looked at some other useful commands. Be sure to cleanup your devlopment area and stop all currently running containers. Do not delete (rmi) any of the images though.