# Docker Compose

Docker Compose is a tool for defining and running **multi-container** Docker applications. With Compose, you use a **YAML** file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Using Compose is basically a three-step process:

1. Define your app's environment with a **Dockerfile** so it can be reproduced anywhere.
2. Define the services that make up your app in **docker-compose.yml** so they can be run together in an isolated environment.
3. Run docker-compose up and Compose starts and runs your entire app.

A **docker-compose.yml** looks like this:

```
version: '3'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

## Install Docker Compose

### Prerequisites

Docker Compose relies on Docker Engine for any meaningful work, so make sure you have Docker Engine installed either locally or remote, depending on your setup.

1. Run this command to download the current stable release of Docker Compose:

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

2. Apply executable permissions to the binary.

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

3. Test the installation.

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

**Docker-compose example**

a simple Python web application running on Docker Compose. The application uses the Flask framework and maintains a hit counter in Redis. While the sample uses Python, the concepts demonstrated here should be understandable even if you're not familiar with it.

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.

**Step 1: 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:

```
import time
import redis
from flask import Flask



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



def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)



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

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

In this example, **redis** is the hostname of the redis container on the application's network. We use the default port for Redis, **6379**.

**Explanation**: Note the way the get_hit_count function is written. This basic retry loop lets us attempt our request multiple times if the redis service is not available. This is useful at startup while the application comes online, but also makes our application more resilient if the Redis service needs to be restarted anytime during the app's lifetime. In a cluster, this also helps handling momentary connection drops between nodes.

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

```
flask
redis
```

**Step 2: 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.

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"]
```

**Explanation**

- 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 3: Define services in a Compose file**

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

```
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"
  redis:
    image: "redis:alpine"
```

**Explanation**

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. We use the default port for the Flask web server, 5000.

The redis service uses a public Redis image pulled from the Docker Hub registry.

**Step 4: Build and run your app with Compose**

1. From your project directory, start up your application by running docker-compose up

```
$ docker-compose up
```

Sample log

```
redis_1   | 1:M 04 Apr 2019 10:59:28.736 * Running mode=standalone, port=6379.
redis_1   | 1:M 04 Apr 2019 10:59:28.736 # WARNING: The TCP backlog setting of 51
1 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower va
lue of 128.
redis_1   | 1:M 04 Apr 2019 10:59:28.736 # Server initialized
redis_1   | 1:M 04 Apr 2019 10:59:28.736 # WARNING overcommit_memory is set to 0!
 Background save may fail under low memory condition. To fix this issue add 'vm.
overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 's
ysctl vm.overcommit_memory=1' for this to take effect.
redis_1   | 1:M 04 Apr 2019 10:59:28.737 # WARNING you have Transparent Huge Page
s (THP) support enabled in your kernel. This will create latency and memory usag
e issues with Redis. To fix this issue run the command 'echo never > /sys/kernel
/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in o
rder to retain the setting after a reboot. Redis must be restarted after THP is
disabled.
redis_1   | 1:M 04 Apr 2019 10:59:28.737 * Ready to accept connections
web_1     |  * Serving Flask app "app" (lazy loading)
web_1     |  * Environment: production
web_1     |    WARNING: Do not use the development server in a production environ
ment.
web_1     |    Use a production WSGI server instead.
web_1     |  * Debug mode: on
web_1     |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
web_1     |  * Restarting with stat
web_1     |  * Debugger is active!
web_1     |  * Debugger PIN: 326-951-687
```

2. open **http://MACHINE_IP:5000** in a browser.

3. Refresh the page.

4. The number should increment.


5. Switch to another terminal window, and type **`docker image ls`** to list local images.

```
[ec2-user@ip-172-31-18-218 ~]$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
composetest_web     latest              01aaaa3628bc        About an hour ago
84.5MB
python              3.4-alpine          c06adcf62f6e        2 weeks ago
72.9MB
redis               alpine              07103bda7d12        2 weeks ago
51.6MB
```

6. Stop the application, either by running **docker-compose down** from within your project directory in the second terminal, or by hitting **CTRL+C** in **the original terminal** where you started the app.

**Step 5: Edit the Compose file to add a bind mount**

Edit docker-compose.yml in your project directory to add a bind mount for the web service:

```
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: "redis:alpine"
```

The new volumes key mounts the project directory (current directory) on the host to /code inside the container, allowing you to modify the code on the fly, without having to rebuild the image.

**Step 6: Re-build and run the app with Compose**

From your project directory, type **docker-compose up** to build the app with the updated Compose file, and run it.

Check the **Hello World** message in a web browser again, and refresh to see the count increment.

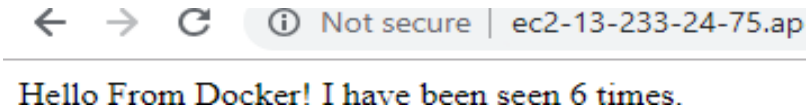
**Step 7: Update the application**

Because the application code is now 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, change the Hello World! message to Hello from Docker!:

```
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.

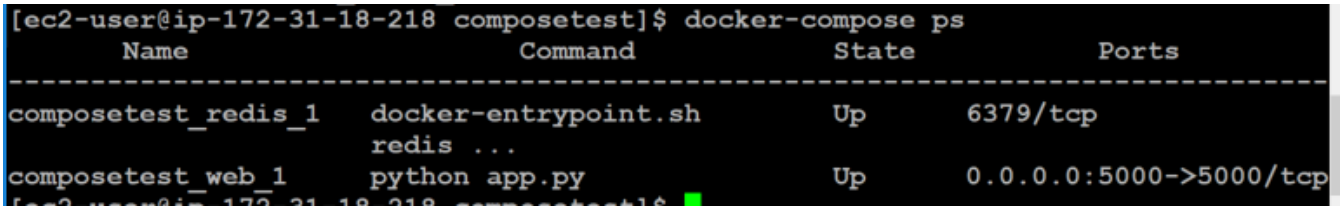



**Step 8: Experiment with some other commands**

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
```


```
$ docker-compose ps
```



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
```

If you started Compose with docker-compose up -d, stop your services once you've finished with them:

```
$ docker-compose stop
```


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
```