Use Docker Compose

## What is Docker Compose?

If your Docker application includes more than one container (for example, a webserver and database running in separate containers), building, running, and connecting the containers from separate Dockerfiles is cumbersome and time-consuming. Docker Compose solves this problem by allowing you to use a YAML file to define multi-container apps. You can configure as many containers as you want, how they should be built and connected, and where data should be stored. When the YAML file is complete, you can run a single command to build, run, and configure all of the containers.

# Docker Compose Example

This is the step by step tutorial to understand uses of Docker compose. In this tutorial, I will create two Docker containers using Docker compose. One docker container will have MySQL database instance and another Docker container have Apache web server with our dummy application file.

Let’s follow step by step tutorial and watch the things happening there.

## Step 1 – Create Directory Structure

First of all, create a directory structure. Here webapp is our web application directory. Also, create a index.html in webapp directory for testing.

|  |  |
| --- | --- |
| **1**  **2**  **3** | **$ mkdir dockercompose && cd dockercompose**  **$ mkdir webapp**  **$ echo "<h2>It Works</h2>" > webapp/index.html** |

## Step 2 – Create Dockerfile for Webapp

Now create a Dockerfile in webapp directory to create a customized image for your application including Apache web server.

$ vim webapp/Dockerfile

add following content

FROM tecadmin/ubuntu-ssh:16.04

RUN apt-get update \

&& apt-get install -y apache2

COPY index.html /var/www/html/

WORKDIR /var/www/html

CMD ["apachectl", "-D", "FOREGROUND"]

EXPOSE 80

## Step 3 – Create Docker Compose File

Finally create a docker compose configuration file (docker-compose.yml) file in current directory. This will define all the containers will be used in your current setup.

$ vim docker-compose.yml

add following content.

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17** | **version: '3'**  **services:**  **db:**  **image: mysql**  **container\_name: mysql\_db**  **restart: always**  **environment:**  **- MYSQL\_ROOT\_PASSWORD="secret"**  **web:**  **image: apache**  **build: ./webapp**  **depends\_on:**  **- db**  **container\_name: apache\_web**  **restart: always**  **ports:**  **- "8080:80"** |

Above docker compose file has settings for two containers. The first container is for mysql database server and the second is for web server. The web container will run our application on Apache server. As this is customized we have defined build directory to webapp.

## Step 4 – Build Webapp Image

Now, build an image using the following command. This will create an image named apache using Dockerfile and contents from webapp directory.

$ docker-compose build

read the below output of above command. I have skipped some part of output which is not required. The first line of below output shows that it skipped building for db container due to no build defined. For web container it uses **webapp/Dockerfile** to build an image.

db uses an image, skipping

Building web

Step 1/6 : FROM tecadmin/ubuntu-ssh:16.04

16.04: Pulling from tecadmin/ubuntu-ssh

b3e1c725a85f: Pull complete

4daad8bdde31: Pull complete

63fe8c0068a8: Pull complete

4a70713c436f: Pull complete

bd842a2105a8: Pull complete

c41407f48fa7: Pull complete

1fcfeb9b5ef4: Pull complete

13195a7d2240: Pull complete

b86be64bbda8: Pull complete

8c951fe917dc: Pull complete

f74bc80103b6: Pull complete

Digest: sha256:523d6fbc97954e9f77231bf54bfcfbbdd4805349887477fbac4a63dc735d777d

Status: Downloaded newer image for tecadmin/ubuntu-ssh:16.04

---> bb63b492da01

Step 2/6 : RUN apt-get update && apt-get install -y apache2

---> Running in 00be0dd717ce

[[[Removed long output from here]]]

---> 41c731590234

Removing intermediate container 00be0dd717ce

Step 3/6 : COPY index.html /var/www/html/

---> 42f84d4c2243

Removing intermediate container 945aaee6cbde

Step 4/6 : WORKDIR /var/www/html

---> 40bebd21e352

Removing intermediate container e13f5f412906

Step 5/6 : CMD apachectl -D FOREGROUND

---> Running in ab0db1ef1c6e

---> 587bf2323289

Removing intermediate container ab0db1ef1c6e

Step 6/6 : EXPOSE 80

---> Running in 7bcbef52d585

---> 8f03d4135394

Removing intermediate container 7bcbef52d585

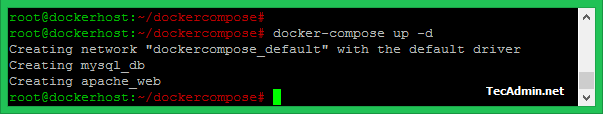
Successfully built 8f03d4135394

Successfully tagged apache:latest

## Step 5 – Launch Docker Containers

Finally launch your containers using docker-compose up command. Use **-d** switch to run them in daemon mode.

$ docker-compose up -d

[](https://tecadmin.net/tutorial/wp-content/uploads/2017/09/docker-compose-up.png)

You can access your web application running on the apache\_web container by accessing your docker host on port 8080. For example, http://dockerhost:8080/ where dockerhost is IP or hostname of your Docker host machine.

## Step 6 – Update Content in Web Application

Let’s make a change in your web application. I have added some more content to webapp/index.html file as following.

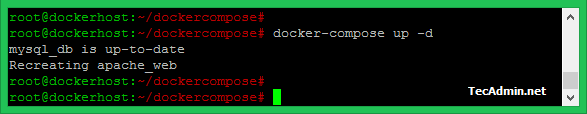
$ echo "Welcome to Docker Compose Tutorial" >> webapp/index.html

Now use the following commands to rebuild webapp container and relaunch using docker-compose.

$ docker-compose build

$ docker-compose up -d

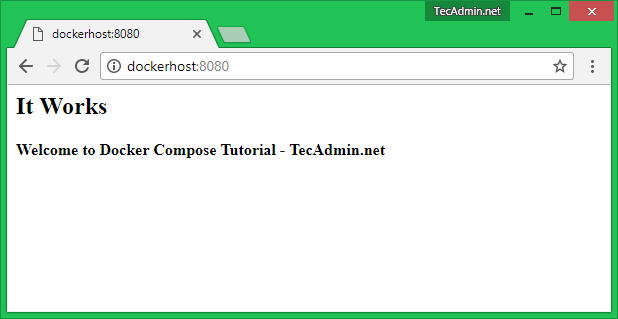
Check the output of the command.

[](https://tecadmin.net/tutorial/wp-content/uploads/2017/09/docker-compose-recreate.png)

You can see that mysql\_db container is showing unchanged as nothing changed there. The only apache\_web container has been recreated due to new build found for the image used for that.

Again access your web application on port 8080 of docker host machine. You will see the updated content here.

http://dockerhost:8080/

[](https://tecadmin.net/tutorial/wp-content/uploads/2017/09/docker-compose-test.png)

Example 2:

# Docker Compose

## Step 1: Setup

Define the application dependencies.

1. Create a directory for the project:
2. $ mkdir composetest
3. $ cd composetest
4. Create a file called app.py in your project directory and paste this in:
5. import time
6. import redis
7. from flask import Flask
8. app = Flask(\_\_name\_\_)
9. cache = redis.Redis(host='redis', port=6379)
10. def get\_hit\_count():
11. retries = 5
12. while True:
13. try:
14. return cache.incr('hits')
15. except redis.exceptions.ConnectionError as exc:
16. if retries == 0:
17. raise exc
18. retries -= 1
19. time.sleep(0.5)
20. @app.route('/')
21. def hello():
22. count = get\_hit\_count()
23. return 'Hello World! I have been seen {} times.\n'.format(count)

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

1. Create another file called requirements.txt in your project directory and paste this in:
2. flask
3. 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:3.7-alpine

WORKDIR /code

ENV FLASK\_APP app.py

ENV FLASK\_RUN\_HOST 0.0.0.0

RUN apk add --no-cache gcc musl-dev linux-headers

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["flask", "run"]

This tells Docker to:

* Build an image starting with the Python 3.7 image.
* Set the working directory to /code.
* Set environment variables used by the flask command.
* Install gcc so Python packages such as MarkupSafe and SQLAlchemy can compile speedups.
* Copy requirements.txt and install the Python dependencies.
* Copy the current directory . in the project to the workdir . in the image.
* Set the default command for the container to flask run.

## 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"

This Compose file defines two services: web and redis.

### Web service

The web service uses an image that’s built from the Dockerfile in the current directory. It then binds the container and the host machine to the exposed port, 5000. This example service uses the default port for the Flask web server, 5000.

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

1. From your project directory, start up your application by running docker-compose up.
2. $ docker-compose up
3. Creating network "composetest\_default" with the default driver
4. Creating composetest\_web\_1 ...
5. Creating composetest\_redis\_1 ...
6. Creating composetest\_web\_1
7. Creating composetest\_redis\_1 ... done
8. Attaching to composetest\_web\_1, composetest\_redis\_1
9. web\_1 | \* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
10. redis\_1 | 1:C 17 Aug 22:11:10.480 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
11. redis\_1 | 1:C 17 Aug 22:11:10.480 # Redis version=4.0.1, bits=64, commit=00000000, modified=0, pid=1, just started
12. redis\_1 | 1:C 17 Aug 22:11:10.480 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
13. web\_1 | \* Restarting with stat
14. redis\_1 | 1:M 17 Aug 22:11:10.483 \* Running mode=standalone, port=6379.
15. redis\_1 | 1:M 17 Aug 22:11:10.483 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
16. web\_1 | \* Debugger is active!
17. redis\_1 | 1:M 17 Aug 22:11:10.483 # Server initialized
18. redis\_1 | 1:M 17 Aug 22:11:10.483 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage 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 order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
19. web\_1 | \* Debugger PIN: 330-787-903
20. redis\_1 | 1:M 17 Aug 22:11:10.483 \* Ready to accept connections

Compose pulls a Redis image, builds an image for your code, and starts the services you defined. In this case, the code is statically copied into the image at build time.

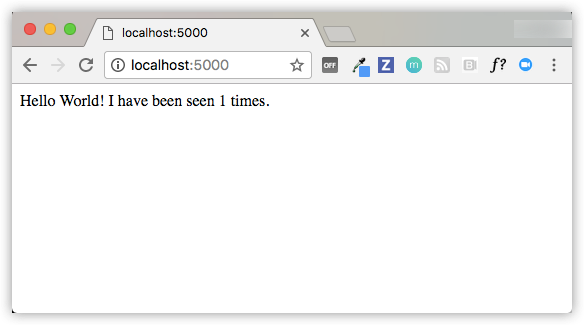
1. Enter http://localhost:5000/ in a browser to see the application running.

If you’re using Docker natively on Linux, Docker Desktop for Mac, or Docker Desktop for Windows, then the web app should now be listening on port 5000 on your Docker daemon host. Point your web browser to http://localhost:5000 to find the Hello World message. If this doesn’t resolve, you can also try http://127.0.0.1:5000.

If you’re using Docker Machine on a Mac or Windows, 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:

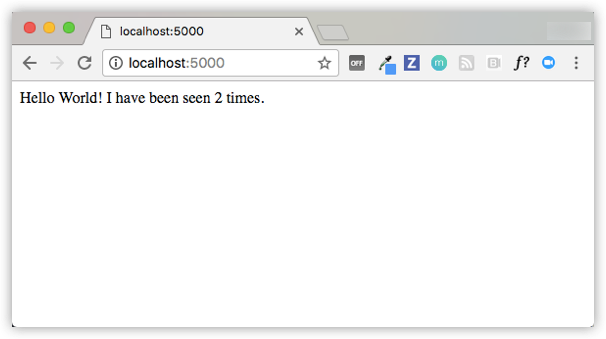
Hello World! I have been seen 1 times.



1. Refresh the page.

The number should increment.

Hello World! I have been seen 2 times.



1. Switch to another terminal window, and type docker image ls to list local images.

Listing images at this point should return redis and web.

$ docker image ls

REPOSITORY TAG IMAGE ID CREATED SIZE

composetest\_web latest e2c21aa48cc1 4 minutes ago 93.8MB

python 3.4-alpine 84e6077c7ab6 7 days ago 82.5MB

redis alpine 9d8fa9aa0e5b 3 weeks ago 27.5MB

You can inspect images with docker inspect <tag or id>.

1. 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

environment:

FLASK\_ENV: development

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. The environment key sets the FLASK\_ENVenvironment variable, which tells flask run to run in development mode and reload the code on change. This mode should only be used in development.

## 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.

$ docker-compose up

Creating network "composetest\_default" with the default driver

Creating composetest\_web\_1 ...

Creating composetest\_redis\_1 ...

Creating composetest\_web\_1

Creating composetest\_redis\_1 ... done

Attaching to composetest\_web\_1, composetest\_redis\_1

web\_1 | \* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

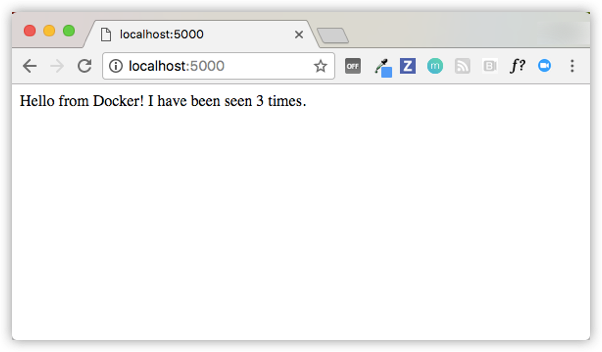
...

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!:
2. return 'Hello from Docker! I have been seen {} times.\n'.format(count)
3. 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

Starting composetest\_redis\_1...

Starting composetest\_web\_1...

$ docker-compose ps

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

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 webservice:

$ docker-compose run web env

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 downcommand. Pass --volumes to also remove the data volume used by the Redis container:

$ docker-compose down --volumes