Get started with Docker Compose

*Estimated reading time: 10 minutes*

On this page you build 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.

Prerequisites

Make sure you have already installed both [Docker Engine](https://docs.docker.com/engine/installation/) and [Docker Compose](https://docs.docker.com/compose/install/). You don’t need to install Python or Redis, as both are provided by Docker images.

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)
24. if \_\_name\_\_ == "\_\_main\_\_":
25. 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.

**Handling transient errors**

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.

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:3.4-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.

For more information on how to write Dockerfiles, see the [Docker user guide](https://docs.docker.com/engine/tutorials/dockerimages/#building-an-image-from-a-dockerfile) and the [Dockerfile reference](https://docs.docker.com/engine/reference/builder/).

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. 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](https://registry.hub.docker.com/_/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.
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.

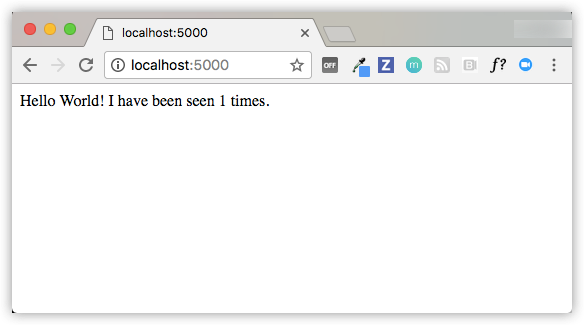
1. Enter http://0.0.0.0:5000/ in a browser to see the application running.

If you’re using Docker natively on Linux, Docker for Mac, or Docker 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 tryhttp://0.0.0.0:5000.

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:

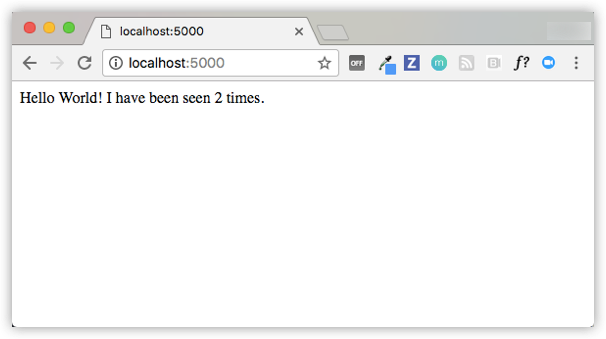
Hello World! I have been seen 1 times.



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](https://docs.docker.com/engine/admin/volumes/bind-mounts/) 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.

$ 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.

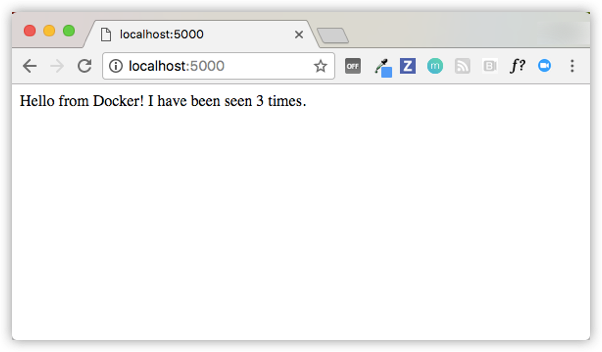
**Shared folders, volumes, and bind mounts**

* If your project is outside of the Users directory (cd ~), then you need to share the drive or location of the Dockerfile and volume you are using. If you get runtime errors indicating an application file is not found, a volume mount is denied, or a service cannot start, try enabling file or drive sharing. Volume mounting requires shared drives for projects that live outside of C:\Users (Windows) or /Users (Mac), and is required for *any* project on Docker for Windows that uses [Linux containers](https://docs.docker.com/docker-for-windows/#switch-between-windows-and-linux-containers-beta-feature). For more information, see [Shared Drives](https://docs.docker.com/docker-for-windows/#shared-drives)on Docker for Windows, [File sharing](https://docs.docker.com/docker-for-mac/#file-sharing) on Docker for Mac, and the general examples on how to [Manage data in containers](https://docs.docker.com/engine/tutorials/dockervolumes/).
* If you are using Oracle VirtualBox on an older Windows OS, you might encounter an issue with shared folders as described in this [VB trouble ticket](https://www.virtualbox.org/ticket/14920). Newer Windows systems meet the requirements for [Docker for Windows](https://docs.docker.com/docker-for-windows/install/) and do not need VirtualBox.

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

See docker-compose --help to see other available commands. You can also install [command completion](https://docs.docker.com/compose/completion/) for the bash and zsh shell, which also shows you available commands.

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

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