Docker compose

Raúl Estrada

Octubre 2020

1. Overview

When using Docker extensively, the management of several different containers quickly becomes cumbersome. Docker Compose is a tool that helps us overcome this problem and **easily handle multiple containers at once.** In this tutorial, we'll have a look at its main features and powerful mechanisms.

2. The YAML Configuration Explained

In short, Docker Compose works by applying many rules declared within **a single *docker-compose.yml* configuration file**.

These YAML rules, both human-readable and machine-optimized, provide us an effective way to snapshot the whole project from ten-thousand feet in a few lines.

Almost every rule replaces a specific Docker command so that in the end we just need to run:

```
1 docker-compose up
```

We can get dozens of configurations applied by Compose under the hood. This will save us the hassle of scripting them with Bash or something else.

In this file, we need to specify the *version* of the Compose file format, at least one *service*, and optionally *volumes* and *networks*:

```
version: "3.7"
services:
  ...
volumes:
  ...
networks:
  ...
```

Let's see what these elements actually are.

2.1. Services

First of all, ***services* refer to containers' configuration**.

For example, let's take a dockerized web application consisting of a front end, a back end, and a database: We'd likely split those components into three images and define them as three different services in the configuration:

```
services:
  frontend:
    image: my-vue-app
    ...
  backend:
    image: my-springboot-app
    ...
  db:
    image: postgres
    ...
```

There are multiple settings that we can apply to services, and we'll explore them deeply later on.

2.2. Volumes & Networks

Volumes, on the other hand, are physical areas of disk space shared between the host and a container, or even between containers. In other words, **a volume is a shared directory in the host**, visible from some or all containers.

Similarly, ***networks* define the communication rules between containers, and between a container and the host**. Common network zones will make containers' services discoverable by each other, while private zones will segregate them in virtual sandboxes.

Again, we'll learn more about them in the next section.

3. Dissecting a Service

Let's now begin to inspect the main settings of a service.

3.1. Pulling an Image

Sometimes, the image we need for our service has already been published (by us or by others) in Docker Hub, or another Docker Registry.

If that's the case, then we refer to it with the *image* attribute, by specifying the image name and tag:

```
services:
  my-service:
    image: ubuntu:latest
    ...
```

3.2. Building an Image

Instead, we might need to build an image from the source code by reading its *Dockerfile*.

This time, we'll use the *build* keyword, passing the path to the Dockerfile as the value:

```
services:
  my-custom-app:
    build: /path/to/dockerfile/
    ...
```

We can also use a URL instead of a path:

```
services:
  my-custom-app:
    build: https://github.com/my-company/my-project.git
    ...
```

Additionally, we can specify an *image* name in conjunction with the *build* attribute, which will name the image once created, making it available to be used by other services:

```
services:
  my-custom-app:
    build: https://github.com/my-company/my-project.git
    image: my-project-image
    ...
```

3.3. Configuring the Networking

Docker containers communicate between themselves in networks created, implicitly or through configuration, by Docker Compose. A service can communicate with another service on the same network by simply referencing it by container name and port (for example *network-example-service:80*), provided that we've made the port accessible through the *expose* keyword:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    expose:
      - "80"
```

To reach a container from the host, **the ports must be exposed declaratively through the *ports* keyword**, which also allows us to choose if exposing the port differently in the host:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    ports:
      - "80:80"
    ...
  my-custom-app:
    image: myapp:latest
    ports:
      - "8080:3000"
    ...
  my-custom-app-replica:
    image: myapp:latest
    ports:
      - "8081:3000"
    ...
```

Port 80 will now be visible from the host, while port 3000 of the other two containers will be available on ports 8080 and 8081 in the host. **This powerful mechanism allows us to run different containers exposing the same ports without collisions**.

Finally, we can define additional virtual networks to segregate our containers:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    networks:
      - my-shared-network
    ...
  another-service-in-the-same-network:
    image: alpine:latest
    networks:
      - my-shared-network
    ...
  another-service-in-its-own-network:
    image: alpine:latest
    networks:
      - my-private-network
    ...
networks:
  my-shared-network: {}
  my-private-network: {}
```

In this last example, we can see that *another-service-in-the-same-network* will be able to ping and to reach port 80 of *network-example-service*, while *another-service-in-its-own-network* won't.

3.4. Setting Up the Volumes

There are three types of volumes: *anonymous*, *named*, and *host* ones.

Docker manages both anonymous and named volumes, automatically mounting them in self-generated directories in the host. While anonymous volumes were useful with older versions of Docker (pre 1.9), named ones are the suggested way to go nowadays. **Host volumes also allow us to specify an existing folder in the host.**

We can configure host volumes at the service level and named volumes in the outer level of the configuration, in order to make the latter visible to other containers and not only to the one they belong:

```yaml
services:
  volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/my-volumes/named-global-volume
      - /tmp:/my-volumes/host-volume
      - /home:/my-volumes/readonly-host-volume:ro
    ...
  another-volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/another-path/the-same-named-global-volume
    ...
volumes:
  my-named-global-volume:
```

Here, both containers will have read/write access to the *my-named-global-volume* shared folder, no matter the different paths they've mapped it to. The two host volumes, instead, will be available only to *volumes-example-service*.

The */tmp* folder of the host's file system is mapped to the */my-volumes/host-volume* folder of the container. This portion of the file system is writeable, which means that the container can not only read but also write (and delete) files in the host machine.

We can mount a volume in read-only mode by appending *:ro* to the rule, like for the */home* folder (we don't want a Docker container erasing our users by mistake).

3.5. Declaring the Dependencies

Often, we need to create a dependency chain between our services, so that some services get loaded before (and unloaded after) other ones. We can achieve this result through the *depends_on* keyword:

```
services:
  kafka:
    image: wurstmeister/kafka:2.11-0.11.0.3
    depends_on:
      - zookeeper
    ...
  zookeeper:
    image: wurstmeister/zookeeper
    ...
```

We should be aware, however, that Compose will not wait for the *zookeeper* service to finish loading before starting the *kafka* service: it will simply wait for it to start. If we need a service to be fully loaded before starting another service, we need to get deeper control of startup and shutdown order in Compose.

4. Managing Environment Variables

Working with environment variables is easy in Compose. We can define static environment variables, and also define dynamic variables with the *${}* notation:

```
services:
  database:
    image: "postgres:${POSTGRES_VERSION}"
    environment:
      DB: mydb
      USER: "${USER}"
```

There are different methods to provide those values to Compose.

For example, one is setting them in a *.env* file in the same directory, structured like a *.properties* file, *key=value*:

```
POSTGRES_VERSION=alpine
USER=foo
```

Otherwise, we can set them in the OS before calling the command:

```
export POSTGRES_VERSION=alpine
export USER=foo
docker-compose up
```

Finally, we might find handy using a simple one-liner in the shell:

```
POSTGRES_VERSION=alpine USER=foo docker-compose up
```

We can mix the approaches, but let's keep in mind that Compose uses the following priority order, overwriting the less important with the higher ones:

1. Compose file
2. Shell environment variables
3. Environment file
4. Dockerfile
5. Variable not defined

5. Scaling & Replicas

In older Compose versions, we were allowed to scale the instances of a container through the *docker-compose scale* command. Newer versions deprecated it and replaced it with the *––scale* option.

On the other side, we can exploit Docker Swarm – a cluster of Docker Engines – and autoscale our containers declaratively through the *replicas* attribute of the *deploy* section:

```
services:
  worker:
    image: dockersamples/examplevotingapp_worker
    networks:
      - frontend
      - backend
    deploy:
      mode: replicated
      replicas: 6
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
      ...
```

Under *deploy*, we can also specify many other options, like the resources thresholds. Compose, however, **considers the whole *deploy* section only when deploying to Swarm**, and ignores it otherwise.

7. Lifecycle Management

Let's finally take a closer look at the syntax of Docker Compose:

```
docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
```

While there are many options and commands available, we need at least to know the ones to activate and deactivate the whole system correctly.

7.1. Startup

We've seen that we can create and start the containers, the networks, and the volumes defined in the configuration with *up*:

```
docker-compose up
```

After the first time, however, we can simply use *start* to start the services:

```
docker-compose start
```

In case our file has a different name than the default one (*docker-compose.yml*), we can exploit the *-f* and *--file* flags to specify an alternate file name:

```
docker-compose -f custom-compose-file.yml start
```

Compose can also run in the background as a daemon when launched with the *-d* option:

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

7.2. Shutdown

To safely stop the active services, we can use *stop*, which will preserve containers, volumes, and networks, along with every modification made to them:

```
docker-compose stop
```

To reset the status of our project, instead, we simply run *down*, **which will destroy everything with only the exception of external volumes**:

```
docker-compose down
```

8. Conclusion

In this tutorial, we've learned about Docker Compose and how it works.

As usual, we can find the source *docker-compose.yml* file on GitHub, along with a helpful battery of tests immediately available in the following image: