1. docker-compose.md

**docker-compose.md**

**Docker compose**

**References**

* [Docker compose](https://docs.docker.com/compose/)
* [Docker references](https://docs.docker.com/reference/)
* [Differences between versions](https://docs.docker.com/compose/compose-file/compose-versioning/#upgrading)
* [Docker compose networking](https://docs.docker.com/compose/networking/)
* [Docker compose volumes](https://docs.docker.com/compose/compose-file/#volumes)

**Overview**

Docker compose is the tool that makes creating and managing multi-container applications easier.

It's fundamental use is based around the docker-compose file.

The docker-compose file allows you to run multiple containers in a way that eases otherwise cumbersome configuration. Moreover it makes it easy to set up those containers to talk to one another. It's a YAML file(.yaml or .yml) that includes key information including environment variables, ports, volumes etc.

The file is typically named docker-compose.yml or docker-compose.yaml.

The file can be used simply with the docker-compose up command. This searches the current directory for the appropriate file and spins up the containers outlined.

**Docker-compose file components**

Docker compose files have an overarching format that has evolved from version to version. There are three overarching versions of the docker-compose file, 1.x, 2.x and 3.x, with smaller upgrades constituting the second part of the version number. This document discusses version 3, but both version 2 and 3 follow the same general format of services, networks, and volumes. There are just some differences in the options/parameters available between the two versions.

Each service definition is analogous to the docker run command. Meanwhile the networks and volumes definitions are analogous to docker network create and docker volume create.

**Compose file options**:

* [version](https://app.revature.com/admin-v2/)
* [services](https://app.revature.com/admin-v2/)
  + [Options for a service](https://app.revature.com/admin-v2/)
    - [image](https://app.revature.com/admin-v2/)
    - [build](https://app.revature.com/admin-v2/)
    - [ports](https://app.revature.com/admin-v2/)
    - [environment](https://app.revature.com/admin-v2/)
    - [env\_file](https://app.revature.com/admin-v2/)
* [volumes](https://app.revature.com/admin-v2/)
* [networks](https://app.revature.com/admin-v2/)

**Version declaration:**

The docker-compose file begins with the version declaration. It's the first line of the file and helps the docker compose tool understand/process the file correctly.

For example if we were to use version 3.8. Then we would have:

version: '3.8'

**services**

The various docker containers that you spin up via a docker compose file each constitute a particular service.

At the top level of the docker compose file, you specify a services option and then all the information for each service definition is specified under that service's name (under the services option). For instance:

version: '3.8'

services:

db:

#all the information for the database set up

webapp:

#all the information for the webapp setup

...

**Options for a particular service**

The following constitute a foundational list of the options for configuring a particular service.

**IMAGE**

The image from which the service ought to be built. For example, it might look like this

version: '3.8'

services:

db:

image: postgres:latest

**BUILD**

The build option allows you to outline how to build a service from a dockerfile. It allows you to specify the configuration needed at build time. This includes information like context, arguments and dockerfile.  
For example, in the simplest case, where the dockerfile and context are the same as the directory of the compose file, you might have:

version: '3.8'

services:

db:

build: .

This indicates to build the database from the context of the current directory and a dockerfile within it.

**PORTS**

Allows you to expose ports in the container for the particular service and to map them to ports on the host. The short version of the option syntax is ports: host:container. So for example, the declaration might look like this, if postgres was listening in the container on port 8080:

version: '3.8'

services:

db:

image: 'postgres:11'

ports:

- "80:8080"

...

**ENVIRONMENT**

Allows you to specify environment variables for a particular service. For example, you might have

version: '3.8'

services:

db:

image: 'postgres:11'

ports:

- "80:8080"

environment:

- POSTGRESQL\_HOST=postgresql

- POSTGRESQL\_ROOT\_USER=postgres

**ENV\_FILE**

Allows you to specify environment variables for a particular service as the contents of a file. Each line of the file will be an environment variable. (Variables defined under the environment option override environment variables from the env\_file) For example, you might have a file called *db.env* that contains:

POSTGRESQL\_HOST=postgresql

Then your compose file might look like this:

version: '3.8'

services:

db:

image: postgres:latest

ports:

- "80:8080"

env\_file:

- ./db.env

**RESTART**

The restart option indicates the conditions upon which to restart a container.

The possibilities are:

* no
  + the default and indicates not to restart the container under any circumstances
* always
  + always tries to restart the container
* on-failure
  + when there is an on-failure error
* unless-stopped
  + restart unless the container has been stopped (regardless of whether it's manual or not)

For example you might want you web application to always try to restart:

version: '3.8'

services:

web:

build: .

ports:

- "80:8080"

env\_file:

- ./web.env

restart: always

**volumes**

*Volumes* is both a top level option and part of the service definition.

It allows you to specify how the container might interact with some shared and persistent portion of memory.

For a particular service definition the *volumes* option outlines that service's named volumes or mount host paths.

Here is a set of examples from the docker docs that outline the different short syntax for the various versions of volumes/mounts.

myservice:

...

volumes:

# Just specify a path and let the Engine create a volume

- /var/lib/mysql

# Specify an absolute path mapping

- /opt/data:/var/lib/mysql

# Path on the host, relative to the Compose file

- ./cache:/tmp/cache

# User-relative path

- ~/configs:/etc/configs/:ro

# Named volume

- datavolume:/var/lib/mysql

See [this for reference](https://docs.docker.com/compose/compose-file/#volumes).

If a named volume is meant to be shared among multiple services then you will need the top level *volumes* element as well.

In the top level *volumes* you will list the named volumes and for each you can specify the driver or you can leave it empty. If its empty then it will use the default driver configured for your docker engine.

The driver of a volume provides a layer of abstraction allowing you to have your volume locally or on some remote host.

For example, you might have something similar to this, to use the default volume driver which is typically local for docker engines that have not been reconfigured:

version: '3.8'

services:

db:

...

volumes:

- mydata:/var/postgres

...

backup-db:

...

volumes:

- mydata:/example

volumes:

mydata:

The ... signify omitted portions of the file.

**networks**

Networks were introduced with version 2 of docker compose and they enable containers to easily communicate with one another in a controlled way. (In docker compose version 1, links facilitated the communication between services and they are no longer recommended.)

By default all services specified in a docker compose file are considered part of the same network. The network is by default named after the directory in which the docker-compose file can be found. Containers on the same network are by default reachable and discoverable by one another using the name of the service and the container.

Let's take for example this file out lined in the [docker documentation.](https://docs.docker.com/compose/networking/)

version: "3"

services:

web:

build: .

ports:

- "8000:8000"

db:

image: postgres

ports:

- "8001:5432"

The web service can connect to the db service through the url- postgres://db:5432. Note the url begins with the protocol in this case *postgres* and then specifies the host and the port. In this case the host is db and the port is 5432 (which is the container port). For inter container communication the container's use the container ports.

If we wanted to connect to the db via the host. Our url would be postgres://{DOCKER\_IP}:8001. Notice that the port would be the host port.

The docker ip is assigned when the docker container is spun up. By default it's assigned one for each Docker network it's a part of. We can find this information through the command docker inspect --format='{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' containerID.

So that's all great, but what if you want the containers to be part of a smaller network. Maybe you only want two out of the three containers you define in your docker compose file to be able to talk to one another. Well that's where the user defined networks come into play.

You can name one of the networks or multiple networks in a particular service definition, but then you also need to name these networks using the top level option.

For example, here there is a db that connects to a backend network. An app that connects to both networks and a service called web that connects to the frontend network.

version: "3"

services:

app:

build: ./app

networks:

- backend

- frontend

web:

build: ./web

ports:

- "8000:8000"

networks:

- frontend

db:

image: postgres

ports:

- "8001:5432"

networks:

- backend

networks:

frontend:

backend:

In the top level definition, where we define the custom networks, we can specify to use specific network drivers and some other configuration options. In this case we use all the defaults.

As with volume drivers, network drivers provide core networking functionality in a way that abstracts it from the hardware itself. The default type is *bridge*. More details on the specifics of drivers and networking can be found [in the article on networking standalone containers.](https://docs.docker.com/network/network-tutorial-standalone/)

Now that the key components of the docker-compose file have been outlined. Let's put them all together.

**Docker-compose format**

The following is the format all put together with comments detailing the use of the options.

# This is a comment and has nothing to do with the file,

# but it is helpful for me to describe the different components.

version: "version number"

# ^ The version of docker compose. For the differences between the different versions see the reference.

services:

nameofservice:

# First you name the service then you define how to build it

# Either you specify a build including context and dockerfile

# path and arguments

build:

context:

dockerfile: Dockerfilepath

# any build arguments

args:

nameofarg: value

# Then you specify additional information like the ports, network, volumes

ports:

- portnum:portnum

# networks allows you to name a network so that services can only talk to other services on the same network

networks:

- nameofnetwork

# Then you specify a name of another service to be created

volumes:

- nameofvolumeorpath:/pathincontainer

#environment variables for the particular container

env\_file:

- nameoffile.env

environment:

VARIABLE\_NAME: value

# The other format has - Variablename=value instead of VARIABLE\_NAME: value

restart: restartoption

# indicates the conditions upon which to restart a container

# either always, on-failure, unless-stopped, and "no"

nameofanotherservice:

#if you are using a service with a predefined image you can

#omit using the build instruction

image: nameofimage

# any additional information needed just like the format above

# If there is a named network earlier in the file then it will need to be specified as a network with it's driver indicated

networks:

nameofnetwork:

# If there is a named volume earlier in the file then it will need to be specified as a volume

volumes:

nameofvolume:

**Examples**

The following is a simple example of a web application. While we wouldn't typically use a docker compose file to spin up such a simple application, it illustrates a basic version of using docker compose.

Save the following file as *docker-compose.yaml*.

version: '3'

services:

webapp:

image: 'mcr.microsoft.com/dotnet/core/samples:aspnetapp'

ports:

- '80:80'

Then from the commandline in the directory in which the file is saved.

Run the following command.

docker-compose up -d

(The -d flag indicates to do it in detached mode, so that all the standard output and error information isn't printing to the shell from which you executed the command.)

You should see the application if you go to localhost:80 in your web browser.

Then to stop the containers you can run the following command in the command line from the same directory.

docker-compose down

For an even simpler example that uses a dockerfile. Save the following docker compose file (docker-compose.yml) to the same folder as the docker file after it.

version: '3'

services:

javahelloworld:

build: .

# Define the parent image

FROM ubuntu

# Install needed programs

RUN apt-get update

RUN apt-get -y dist-upgrade

RUN apt-get -y install default-jdk

# Within the image and thus container, set the working directory to the new directory example

WORKDIR /example

# Create Hello World Java program and save it in the appropriate file

RUN echo 'public class HiWorld{ public static void main(String[] args){System.out.println("Hi world");}}'> HiWorld.java

# Compile the Java program, creating the file that the JVM can actually run

RUN javac HiWorld.java

# Run the HelloWorld program in the container

CMD ["java", "HiWorld"]

From the directory with both files run the command docker-compose up and you'll see "Hello World" output.

Finally, let take a look at a more complex example from [docker docs](https://docs.docker.com/compose/wordpress/).

version: '3.3'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: somewordpress

MYSQL\_DATABASE: wordpress

MYSQL\_USER: wordpress

MYSQL\_PASSWORD: wordpress

wordpress:

depends\_on:

- db

image: wordpress:latest

ports:

- "8000:80"

restart: always

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: wordpress

WORDPRESS\_DB\_PASSWORD: wordpress

WORDPRESS\_DB\_NAME: wordpress

volumes:

db\_data: {}

The file outlines using a volume to persist data for the database and application that connects to it. The environment variables allow you to spin up the container in such a way that you have the default credentials set up. You'll also notice that the wordpress web application depends on the database. This ensures the order that the services spin up.

Save the file into a new directory and from that directory run the command docker-compose up -d.

Then after a giving it a few seconds check out your site at <http://localhost:8000/>.

Great work!

**Helpful Examples**

* [Docker compose example](https://docs.docker.com/compose/gettingstarted)