Docker Compose

<< Back to Docker main

Contents

Services

Introduction

In a distributed application, different pieces of the app are called ?services.? Services are really just ?containers in production.? A service only runs **one image**, but it codifies the way that image runs?what ports it should use, **how many replicas** of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily it?s very easy to define, run, and scale services with the Docker platform -- just write a docker-compose.yml file.

Source: https://docs.docker.com/get-started/part3/#prerequisites

YAMEL

YAML /'jæm.?/ is a human-readable data serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored (e.g. debugging output) or transmitted (e.g. document headers). YAML targets many of the same communications applications as XML but has a minimal syntax which intentionally breaks compatibility with SGML?[1]. It uses both Python-style indentation to indicate nesting, and a more compact format that uses [] for lists and {} for maps making YAML 1.2 a superset of JSON. Custom data types are allowed, but YAML natively encodes scalars (such as strings, integers, and floats), lists, and associative arrays (also known as hashes, maps, or dictionaries).

Docker Composition

Introduction

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 build, create containers and start all the services from your configuration with a single command.

Multiple isolated environments on a single host
Compose uses a project name to isolate environments from each other. You can make use of this project name in several different contexts:

- on a dev host, to create multiple copies of a single environment, such as when you want to run a stable copy for each feature branch of a project
- on a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number

Development environments
When you?re developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

- The Compose file provides a way to document and configure all of the application?s service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker-compose up).
- Together, these features provide a convenient way for developers to get started on a project. Compose can reduce a multi-page ?developer getting started guide? to a single machine readable Compose file and a few commands.

Automated testing environments
An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a Compose file, you can create and destroy these environments in just a few commands:

Source: https://docs.docker.com/compose/overview/

docker compose vs docker stack (swarm)

In recent releases, a few things have happened in the Docker world. Swarm mode got integrated into the Docker Engine in 1.12, and has brought with it several new tools. Among others, it?s possible to make use of docker-compose.yml files to bring up stacks of Docker containers, without having to install Docker Compose.

The command is called docker stack, and it looks exactly the same to docker-compose. Both docker-compose and the new docker stack commands can be used with docker-compose.yml files which are written according to the specification of version 3. For your version 2 reliant projects, you?ll have to continue using docker-compose. If you want to upgrade, it?s not a lot of work though.

As docker stack does everything docker compose does, it?s a safe bet that docker stack will prevail. This means that docker-compose will probably be deprecated and won?t be supported eventually.

However, switching your workflows to using docker stack is neither hard nor much overhead for most users. You can do it while upgrading your docker compose files from version 2 to 3 with comparably low effort.

If you?re new to the Docker world, or are choosing the technology to use for a new project - by all means, stick to using docker stack deploy.

Source: https://vsupalov.com/difference-docker-compose-and-docker-stack/

Install

docker-compose is not part of the standard docker installation. We have to install it form github.

```
sudo curl -L https://github.com/docker/compose/releases/download/1.21.2/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compo
sudo chmod +x /usr/local/bin/docker-compose

# docker-compose --version
docker-compose version 1.21.2, build a133471
```

How to use docker-compose

I will demonstrate the potential in docker-compose through a simple example. We are going to build a WordPress service that requires two containers. On for the mysql database and one for the WordPress itself. To make the example a little bit more complicated, we won't simple download the WordPress image from DockerHub, we are going to build it, using the wordPress image as the base image of our newly built image. So with a simple docker-compose.yml file we can build as many images as we want and we can construct containers from them in the given order. Isn't it huge?

```
$ mkdir wp-example
$ cd wp-example
$ mkdir wordpress
$ touch docker-compose.yml

[wp-example]# ll
total 8
-rw-r--r-- 1 root root  148 Jun 23 23:09 docker-compose.yml
drwxr-xr-x 2 root root 4096 Jun 23 22:58 wordpress

$ cd wordpress
$ touch Dockerfile
$ touch example.html

[wordpress]# ll
total 8
-rw-r--r-- 1 root root 159 Jun 23 22:58 Dockerfile
-rw-r--r-- 1 root root  18 Jun 23 22:44 example.html

FROM wordpress:latest
COPY ["./example.html","/var/www/html/example.html"]
VOLUME /var/www/html
ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["apache2-foreground"]
```

docker-compose.yml

```
version: '3'
services:
  wordpress:
    container_name: my-worldrpress-container
    image: myWordPress:6.0
    build: ./wordpress
    links:
      - db:mysql
    ports:
      - 8080:80

  db:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: example
```

Note
You can use only space to make indention. Tab is not supported

```
[wp-example]# docker-compose up -d
Building wordpress
Step 1/5 : FROM wordpress:latest
 ---> 7801d36d734c
Step 2/5 : COPY ./example.html /var/www/html/example.html
 ---> ab67aee3c270
Removing intermediate container a0894a2e834f
Step 3/5 : VOLUME /var/www/html
 ---> Running in 470025d9c877
 ---> 9890d3cd9f0a
Removing intermediate container 470025d9c877
Step 4/5 : ENTRYPOINT docker-entrypoint.sh
 ---> Running in 09548484b9b2
 ---> 555754d6a3a7
Removing intermediate container 09548484b9b2
Step 5/5 : CMD apache2-foreground
 ---> Running in 035fcfc0876d
 ---> 076e75c72b58
Removing intermediate container 035fcfc0876d
```

```
Successfully built 076e75c72b58
Successfully tagged wp2-example_wordpress:latest
WARNING: Image for service wordpress was built because it did not already exist. To rebuild this image you must use `docker-compose build` or
Creating wp2-example_db_1 ... done
Creating wp2-example_wordpress_1 ... done
Attaching to wp2-example_db_1, wp2-example_wordpress_1
```

```
[wp-example]# docker-compose ps
          Name                        Command               State          Ports
-------------------------------------------------------------------------------------
wp2-example_db_1          docker-entrypoint.sh mysqld      Up      3306/tcp
wp2-example_wordpress_1   docker-entrypoint.sh apach ...   Up      0.0.0.0:8080->80/tcp
```

```
# docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED              STATUS              PORTS                   NAMES
8ec2920234b6        wp2-example_wordpress    "docker-entrypoint..."   About a minute ago   Up About a minute   0.0.0.0:8080->80/tcp    wp2-exam
786fb7da1ca7        mariadb                  "docker-entrypoint..."   About a minute ago   Up About a minute   3306/tcp                wp2-exam
```

docker-compose.yml syntax

There are three major versions of compose file format.

Compose only

- **build**: Configuration options that are applied at build time
  - The object form is allowed in Version 2 and up.
  - In version 1, using build together with image is not allowed. Attempting to do so results in an error.
  - From version 3, if you specify image as well as build, then Compose names the built image with the webapp and optional tag specified in image
  - Note: This option is ignored when deploying a stack in swarm mode with a (version 3) Compose file. The docker stack command accepts only pre-built images.
    - **dockerfile**: Dockerfile-alternate
    - **CONTEXT**: Either a path to a directory containing a Dockerfile, or a url to a git repository.
    - **TARGET**: Build the specified stage as defined inside the Dockerfile (added in 3.4)

```
version: '3'
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-alternate
      target: prod
```

- **container_name**: define the name of the container that is created for the service.

Note

- If the build and the image is both provided, the image will be created with the name given in the image parameter.
- If the image tag is not provided, the default name of the new image is: </directory name>_<service_name>
- If the container_name is not provided, the container default name is: </directory name>_<service_name>

- **container_name**: Specify a custom container name, rather than a generated default name

- **external_links**: Link to containers started outside this docker-compose.yml or even outside of Compose, especially for containers that provide shared or common services. external_links follow semantics similar to links when specifying both the container name and the link alias (CONTAINER:ALIAS).

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

- **network_mode**: Network mode. Use the same values as the docker client --network parameter, plus the special form service:[service name].

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

'Swarm mode' only

See Docker Swarm mode for details

- **deploy**: This only takes effect when deploying to a swarm with docker stack deploy, and is ignored by docker-compose up and docker-compose run.
  - **ENDPOINT_MODE**:
    - ◊ vip: (default): Single Virtual IP for the service. Swarm is load balancing
    - ◊ dnsrr: (DNS round rubin): DNS query gives the list of the nodes for our own load balancing.
  - **MODE**: Either global (exactly one container per swarm node) or replicated (a specified number of containers)
  - **PLACEMENT**:
  - **REPLICAS**: If the service is replicated (which is the default), specify the number of containers that should be running at any given time.
  - **RESOURCES**
  - **RESTART_POLICY**:

```
version: '3'
services:
  redis:
    image: redis:alpine
    deploy:
      replicas: 6
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s
```

Compose && 'Swarm mode' common part

See Docker Swarm mode for details

- **command**: Override the default command.

```
command: ["bundle", "exec", "thin", "-p", "3000"]
```

- **ports**: specify the host:container port mapping like with the -p switch in the **run** command.

```
ports:
 - "3000"
 - "3000-3005"
 - "8000:8000"
 - "9090-9091:8080-8081"
 - "49100:22"
 - "127.0.0.1:8001:8001"
 - "127.0.0.1:5000-5010:5000-5010"
 - "6060:6060/udp"
```

- **dns**: Custom DNS servers. Can be a single value or a list.

```
dns: 8.8.8.8
dns:
  - 8.8.8.8
  - 9.9.9.9
```

- **dns_search**: Custom DNS search domains. Can be a single value or a list.

```
dns_search: example.com
dns_search:
  - dc1.example.com
  - dc2.example.com
```

- **entrypoint**: Override the default entrypoint.

```
entrypoint: /code/entrypoint.sh
or as a list:
entrypoint:
    - php
    - -d
```

Note: Setting entrypoint both overrides any default entrypoint set on the service?s image with the ENTRYPOINT Dockerfile instruction, and clears out any default command on the image - meaning that if there?s a CMD instruction in the Dockerfile, it is ignored.

- **env_file**: Add environment variables from a file. Can be a single value or a list.

```
env_file:
  - ./common.env
  - ./apps/web.env
```

- **environment**: Add environment variables

```
environment:
  - RACK_ENV=development
  - SHOW=true
```

- **expose**: Expose ports without publishing them to the host machine - they?ll only be accessible to linked services. Only the internal port can be specified.

```
expose:
 - "3000"
 - "8000"
```

- **extends**: Extend another service, in the current file or another, optionally overriding configuration.

```
extends:
  file: common.yml  <<from this file
  service: webapp  <<extend this service
```

- **extra_hosts**: Add hostname mappings. Use the same values as the docker client --add-host parameter.

```
extra_hosts:
 - "somehost:162.242.195.82"
 - "otherhost:50.31.209.229"
```

- **image**: Specify the image to start the container from. Can either be a repository/tag or a partial image ID.If the image does not exist, Compose attempts to pull it

Note: In the version 1 file format, using build together with image is not allowed. Attempting to do so results in an error.

- **labels**: Add metadata to containers using Docker labels. You can use either an array or a dictionary.

```
labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
```

- **links**: Link to containers in another service. Either specify both the service name and a link alias (SERVICE:ALIAS), or just the service name.

Links are a legacy option. We recommend using networks instead.

```
web:
  links:
   - db
   - db:database
   - redis
```

Containers for the linked service are reachable at a hostname identical to the alias, or the service name if no alias was specified. Links also express dependency between services in the same way as depends_on, so they determine the order of service startup. Note: If you define both links and networks, services with links between them must share at least one network in common in order to communicate.

- **networks**: Networks to join. The container created from this service will be connected to the given network(s). (web to new, worker to legacy)

```
services:
  web:
    build: ./web
    networks:
      - new

  worker:
    build: ./worker
    networks:
      - legacy
```

- **volumes**: specify the container volume:host dir mappaing like with the -v option in the run command. Long and short version:

```
version: "3.2"
services:
  web:
    image: nginx:alpine
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true
      - type: bind
        source: ./static
        target: /opt/app/static
```

```
db:
  image: postgres:latest
  volumes:
    - "/var/run/postgres/postgres.sock:/var/run/postgres/postgres.sock"
    - "dbdata:/var/lib/postgresql/data"
```

Good to know

You can control the order of service startup with the depends_on option. Compose always starts containers in dependency order, where dependencies are determined by depends_on, links, volumes_from, and network_mode: "service:...". However, Compose does not wait until a container is ?ready? (whatever that means for your particular application) - only until it?s running. There?s a good reason for this.