Microservice Architecture

14 April 2023 15:06

- It is an approach to develop a single application as a suite of small services , each running its own process and communicating with lightweight mechanisms.
- Each component is continuously developed and separately maintained , and the application is then simply the sum of constituent components.
- It solely means , having an application in which we separate the functional components of the application , and maintaining each one of them separate and unifying them at last in production , all the development for these individual components are separate.

Benefits:

- **Developer Independence:** Each and every developer have their own component to work on , there will be less functional dependency from others , Integrational dependency will be occurred later.
- **Isolation and resilience:** As said each and every developer('s) will have their own functional component isolation.
- **Scalability:** As the application is into many individual components , developers have more control on their part of application , so the capacity to progress on the their part application will be more. So the whole application development will be robust and dynamic.
- **Lifecycle automation:** Due to the modularity property of microservice architecture it will be easy to automate the process.

How can we perform the microservice solution using Docker:

- Compose the application using Docker
- Break the application components into individual containers. In here make those application components as docker images , so they will have their associated container.
- We can store the data of the container's into volumes , and we can also split the data that's shared between services into volumes.
- Separate responsibilities , so that each containers runs only one component/executable
- If there is something unchangeable data , we can store them in the image itself, but if there is something like changeable data , for example: configurations , logs as volumes , so that they are mounted on various containers.

Docker-compose Introduction

17 April 2023 01:16

In a machine , when we install docker , we get to install docker daemon and docker client.
If we want to run a container , we need to run commands with help of docker client and able to run the container.
If we want to run a multiple container's in a given machine at a given point of time , we use docker-compose , as basic docker isn't sufficient.

Docker Compose:

Main usage : deploying all the containers based on different images on the same machine , which generate an application as a whole.

- Compose is a tool for defining and running multi-container Docker applications
- With compose , you use a compose file to configure your application's services. Then , using a single command , you create and start all the services from your configuration.

Compose has commands for managing the whole lifecycle of our application:

- Start , stop and rebuild services.
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service.

Docker compose uses a configuration file , where you define all the different stuffs we perform on a command line into a file. It will be basically an yaml file (.yml) , default file is docker-compose.yml

Compose file Template:

version: "compose version"
services:
    <ServiceName>:
        <attribute1>:value
        <attribute2>:value

- DockerCompose launches all the services that are mentioned in the docker-compose file , which are required for the application.

Tabs are not allowed in docker-compose , only spaces are allowed in docker-compose , we can use n number of spaces to indent the child properties under parent. But if we indented the child property with 3 spaces , we must also use all other properties under that parent with 3 spaces , so that docker-compose comes to know that , all these services are unique as they declared.

Example:

```
Name: adam
Month: april
Session:
  day: sat
  topic: compose
  time: 6PM
```

Grouping of parent key is called as section , section name for a parent group is defined using "-"

Example:

```
- key:
    key1: value
    key2: value
  keys: value
- keyss: value2
  keysss: value

- name: adam
  month: april
  session:
    day: sat
    topic: compose
    time: 6 PM
- name: student
  session: 2nd
```

Extension for docker-compose is yml or yaml

Keywords of docker-compose file

Version: what API version that we would use. Typically there would be two or three API versions that we use.

Services: Service is a collection of everything that we want to create as an instance. For example, if we want to launch two containers of same image , instead of executing two docker run's , we can mention that in docker-compose file , so , while we execute the compose file , the service is up. We also mention what and home many containers that we would launch and even , we can map the port numbers for the container , what volumes that we are going to give for the container.

"Docker-compose handles everything as a service , the service launches the containers."

Compose File Template:

```
version: "compose version"
services:
  <ServiceName>:
      <attribute1>: value
      <attribute1>: value
```

If we want to execute multiple containers on the same machine , then we would use the docker-compose file.

How to install docker-compose & docker

Install Docker (Ubuntu):

```
$ apt update
$ apt install –y docker.io
```

Install Docker Compose (Ubuntu):

```
$ apt install –y docker-compose
```

Docker-compose Example-1

17 April 2023 01:22

```
[jeevan@localhost example1]$ cat docker-compose.yml
version: '3'
services:
 one:
   build:
     context: ./one
   image: imgone
 two:
   build:
     context: ./two
   image: imgtwo

[jeevan@localhost example1]$
```

Here , we are having this docker-compose file , where we initially giving version number , and then we are declaring the services. The service names are named after 'one' & 'two' , and in these services , we are typically doing the build of the images 'imgone' & 'imgtwo' , here we are also giving the context of the build for imgone as the folder 'one' where it fetches the 'Dockerfile' for the build , if we didn't give any external name for the Dockerfile , it refers the dockerfile , else it refers the file that we mentioned.

```
[jeevan@localhost one]$ cat Dockerfile
FROM ubuntu
COPY run.sh /tmp/run.sh
CMD ["/tmp/run.sh", "one"]
[jeevan@localhost one]$
```

Dockerfile under folder 'one'

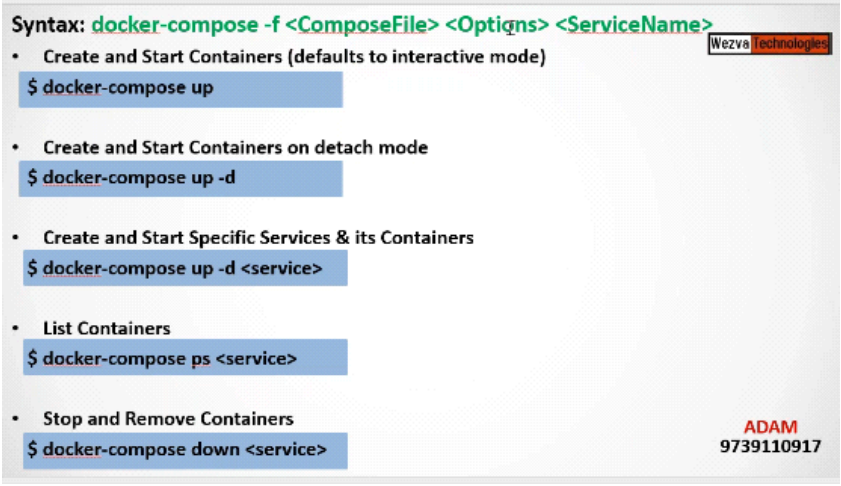
```
[jeevan@localhost two]$ cat Dockerfile
FROM ubuntu
COPY run.sh /tmp/run.sh
CMD ["/tmp/run.sh", "two"]
```

Dockerfile under folder 'two'

```
[jeevan@localhost two]$ cat run.sh
#!/bin/sh
while true;
do
  echo $1
  sleep 8
done
```

Run script is a script basically , we execute it through the image as an triggering point for the container.

Basic commands of docker-compose:



docker-compose commands basically expects the docker-compose.yml file in the same directory as

the docker-compose is been executed , else , if we want to mention the context location , where docker-compose is located , we need to mention that through "-f" tag.

docker-compose -f <composeFile> <Options>

If we want to up a particular service , which is declared in docker-compose file , then we must use

docker-compose <options> <service_name>

docker-compose typically calls the docker daemon , so that the queries of docker-compose file , will be executed through docker daemon.

Docker-compose default creates and runs the container in interactive mode , if we want to make them run in detach mode , we does it through the "-d" flag.

```
[jeevan@localhost two]$ ls
Dockerfile  run.sh
[jeevan@localhost two]$ cd ..
[jeevan@localhost example1]$ ls
docker-compose.yml  one  two
[jeevan@localhost example1]$ pwd
/home/jeevan/docker-compose-practice/docker/dockercompose/example1
[jeevan@localhost example1]$ docker-compose up -d
[+] Running 0/2
 # two Warning                                                                    3.4s
 # one Warning                                                                    3.5s
[+] Building 7.1s (10/11)
 => [imgone internal] load .dockerignore                                         1.2s
 => => transferring context: 2B                                                  0.1s
 => [imgone internal] load build definition from Dockerfile                      1.4s
 => => transferring dockerfile: 159B                                             0.1s
 => [imgtwo internal] load build definition from Dockerfile                      1.3s
 => => transferring dockerfile: 160B                                             0.0s
 => [imgtwo internal] load .dockerignore                                         1.2s
 => => transferring context: 2B                                                  0.1s
 => [imgtwo internal] load metadata for docker.io/library/ubuntu:latest          3.7s
 => [imgone internal] load build context                                         0.9s
 => => transferring context: 143B                                                0.1s
 => [imgone 1/2] FROM docker.io/library/ubuntu@sha256:67211c14fa74f070d27cc59d69a7fa9a  1.9s
 => => resolve docker.io/library/ubuntu@sha256:67211c14fa74f070d27cc59d69a7fa9aeff8e28  0.5s
 => => sha256:67211c14fa74f070d27cc59d69a7fa9aeff8e28ea118ef3babc295a0 1.13kB / 1.13kB  0.0s
 => => sha256:7a57c69fe1e9d5b97c5fe649849e79f2cfc3bf11d10bbd5218b4eb61716a 424B / 424B  0.0s
 => => sha256:08d22c0ceb150ddeb2237c5fa3129c0183f3cc6f5eeb2e7aa4016da3 2.30kB / 2.30kB  0.0s
 => [imgtwo internal] load build context                                         1.0s
 => => transferring context: 143B                                                0.1s
 => [imgone 2/2] COPY run.sh /tmp/run.sh                                         0.3s
 => [imgone] exporting to image                                                  1.0s
 => => exporting layers                                                          0.4s
 => => writing image sha256:9154c032a7a784f1cd2a354c8190f3651a5ece56806ff70fb6b39b2682  0.1s
 => => writing image sha256:7016b503a1cb5d8e1422f91ec7229e3ed88694fd768c3cb57e0bcbbbb4  0.0s
 => => naming to docker.io/library/imgtwo                                        0.0s
 => => naming to docker.io/library/imgone                                        0.1s
[+] Running 3/3
 # Network example1_default  Created                                             2.4s
 # Container example1-one-1  Started                                            10.9s
 # Container example1-two-1  Started                                            10.9s
[jeevan@localhost example1]$
```

As seen , two images are created and containers are also created for them , as you seen below , we even extracted the running container logs.

```
[jeevan@localhost example1]$ docker ps
CONTAINER ID   IMAGE     COMMAND             CREATED              STATUS              PORTS
   NAMES
a79883a82726   imgtwo    "/tmp/run.sh two"   About a minute ago   Up About a minute
   example1-two-1
5404bd7ea241   imgone    "/tmp/run.sh one"   About a minute ago   Up About a minute
   example1-one-1
[jeevan@localhost example1]$ docker logs a79883a82726
two
two
two
two
two
two
two
two
two
two
two
two
two
```

Difference between **docker-compose ps & docker ps**
docker-compose ps => List all the containers , generated by the docker-compose service. Interacts with docker client which in turn interact with the docker daemon to get out the result out.
docker ps => List all the containers , including containers generated by docker-compose. Interacts with docker daemon to get out the result.
We can also mention the service name , to list out all the containers generated by the service => **docker-compose ps <service-name>**
To check logs of the containers based on the service which generated those containers => **docker-compose logs -f <service-name>**

If we want to display logs of both the containers generated by a service , we must use => **docker-compose logs -f**

docker-compose stop <service-name> => stops the containers
docker-compose down <service-name> => stops and removes the containers
docker-compose rm <service-name> => removes the containers
docker-compose build <service-name> => build the images

Run multiple containers together into service , we use 'docker-compose exec'
To run interactively => **docker-compose exec <service> <command>**
To run a command in detached mode => **docker-compose exec -d <service> <command>**
To get into the container , generated by docker-compose => **docker-compose exec <service-name> /bin/bash**

Command that is executed in interactive mode , generates us an output , but a command that is executed in detached mode , doesn't generate an output.

Scale containers for a service: => **docker-compose up -d --scale <service>=<num>**

When coming to scaling the containers , it says us to generate more container's for a service , or decreasing the container's for a service.

We can specify the name of docker-compose file , while we are pushing up the service. => **docker-compose -f <compose-file> up -d**

Docker-compose Example-2

17 April 2023 11:53

13 lines (13 sloc) | 229 Bytes

```
version: '3'
services:
 testservice:
  build:
    context: ./myapache
    dockerfile: dockerfile.apache
  container_name: myapachecont
  image: myapache
  ports:
    - 80:80
  restart: always
  volumes:
    - /var/www:/var/www
```

This is example 2 , where we can see , we are building an image 'myapache' based on the Dockerfile 'dockerfile.apache' , giving the context for the build as 'myapache' folder , and as container generates it must be named as 'myapachecont' , the following service is named after 'testservice' , which is hosted at 80:80. volume mount is also declared for the service.

The equivalent for above docker-compose file in docker is:
docker run -d --name myapachecont -p 80:80 -v /var/www:/var/www --restart=always myapache

To get the details of all the services that are executing on port 80 , to check , we must do the following
netstat -an | grep 80

If we want to run multiple containers on same machine at a given instance on one machine , we use docker-compose.