From Zero to Docker

Training | 2019.05.16 | Mário Dagot, Jorge Dias

Docker is an open platform for developing, shipping, and running applications. Through the course of this training we will guide you to the most common feature and use cases of docker. Take this as an introduction and an opportunity to dive into the docker world.

AGENDA

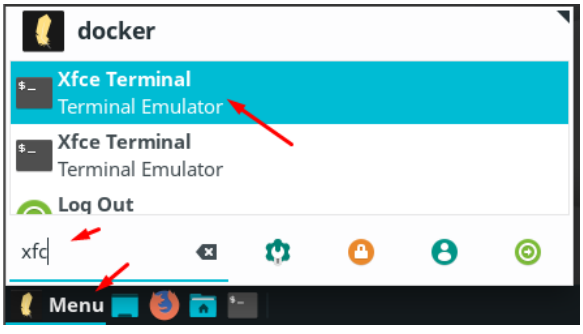
- 01 - Install Vim and Terminator and VSCode
- 02 - Install Docker CE for Ubuntu
- 03 - Hello from Busybox
- 04 - Webapp with Docker
- 05a - Webapp with Docker - My first Dockerfile – Nginx
- 05b.1 - Webapp with Docker - My first Dockerfile - Dotnet Core
- 05b.2 - Webapp with Docker - My first Dockerfile MultiStage - Dotnet Core
- 06 - Save and Restore and Push to Docker Hub
- 07a - Webapp with database integration - My first network – SpringBoot
- **07b - Webapp with database integration - My first docker-compose – SpringBoot**

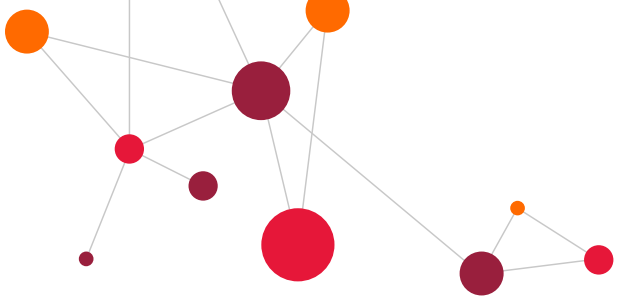
07B - WEBAPP WITH DATABASE INTEGRATION - MY FIRST DOCKER-COMPOSE – SPRINGBOOT

Objective

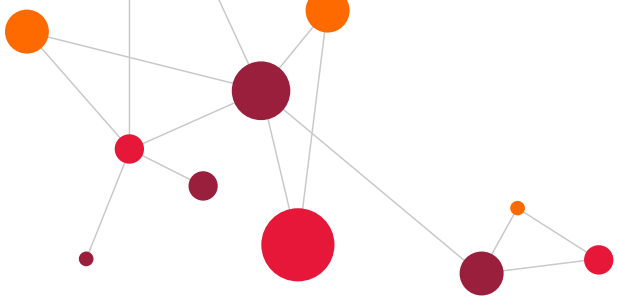
- Extend our spring java application and create our first docker-compose.

Step by Step

Open a terminal windows	
Let's create out multi-stage docker file to build our spring boot application.	`docker ~ cd ~/workspace/gs-accessing-data-mysql/complete/` `docker ~ > workspace > gs-accessing-data-mysql > complete vi Dockerfile`



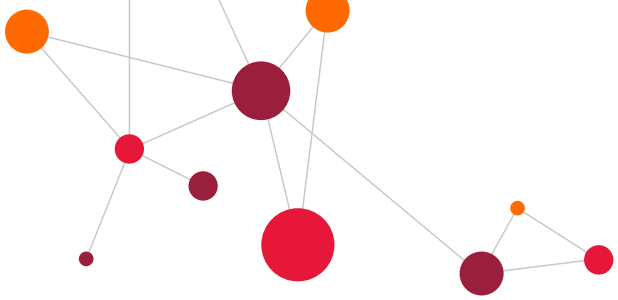
	docker ~ > workspace > gs-accessing-data-mysql > complete cat Dockerfile FROM maven:latest AS build WORKDIR /app COPY . /app RUN mvn clean package > /dev/null FROM openjdk:8-jre AS runtime WORKDIR /app COPY --from=build /app/target/*.jar /app ENTRYPOINT ["java", "-jar", "/app/gs-mysql-data-0.1.0.jar"]
Build the docker image.	docker ~ > workspace > gs-accessing-data-mysql > complete docker build --tag myjavaapp . Sending build context to Docker daemon 36.11MB Step 1/8 : FROM maven:latest AS build ---> cafa0008b735 Step 2/8 : WORKDIR /app ---> Using cache ---> fe36fac97745 Step 3/8 : COPY . /app ---> 44868e944ec4 Step 4/8 : RUN mvn clean package > /dev/null ---> Running in fee1cd6d39fb Removing intermediate container fee1cd6d39fb ---> d8db7d5ef911 Step 5/8 : FROM openjdk:8-jre AS runtime ---> b5ee13f1fc07 Step 6/8 : WORKDIR /app ---> Using cache ---> 6aa01624147c Step 7/8 : COPY --from=build /app/target/*.jar /app ---> d24076c9c0d2 Step 8/8 : ENTRYPOINT ["java", "-jar", "/app/gs-mysql-data-0.1.0.jar"] ---> Running in e4c2a76e8a09 Removing intermediate container e4c2a76e8a09 ---> bc4b5905ce16 Successfully built bc4b5905ce16



	`Successfully tagged myjavaapp:latest`

Create/update the docker-compose file.
 	
This is the basic structure.
 	
It uses a standard yaml file. Be careful, yaml files are picky with indentation.
 	
To notice:
An app is not just one single service. It's a group of services all working together.
On our example, we have a spring boot webapp and a database
We can define what services compose our application using docker-compose
Docker compose may contain many service definitions
Check out the docker-compose reference for all the options available
The compose file uses concepts we have seen when using docker standalone: ports, volumes, networks, etc
	`docker ~ > workspace > gs-accessing-data-mysql-complete > complete vi docker-compose.yml`
	`docker ~ > workspace > gs-accessing-data-mysql-complete > complete cat docker-compose.yml`
	`version: '3.2'`
	`services:`
	`  webapp:`
	`  image: myjavaapp`
	`  depends_on:`
	`  - mysql`
	`  ports:`
	`  - 8080:8080`
	`  mysql:`
	`  image: mysql`
	`  ports:`
	`  - "3306:3306"`
	`  environment:`
	`  - MYSQL_USER=springuser`
	`  - MYSQL_PASSWORD=ThePassword`
	`  - MYSQL_DATABASE=db_example`
	`  - MYSQL_ROOT_PASSWORD=root`
	`  volumes:`
	`  - mysql_data:/etc/mysql/conf.d:ro`
	`volumes:`
	`  mysql_data:`

Let's put our services up and wait 20 seconds to give time for all services to start.
 	
Something is wrong… myapp didn't start.
 	
If we look at the logs we can see that myapp failed because mysql service was not available.
	`docker ~ > workspace > gs-accessing-data-mysql-complete > complete docker-compose up -d && sleep 20 && docker-compose ps`
	`Creating network "complete_default" with the default driver`
	`Creating complete_mysql_1 ... done`
	`Creating complete_webapp_1 ... done`
	`  Name  Command  State  Ports`
	`----------`
	`complete_mysql_1  docker-entrypoint.sh mysqld  Up  0.0.0.0:3306->3306/tcp, 33060/tcp`



This is due to a naive approach on our docker-compose, using depends_on.

depends_on expresses dependency between services but it does not wait for mysql to be “ready” before starting myapp.

```
complete_webapp_1   java -jar /app/gs-mysql-da ...   Exit 1
```

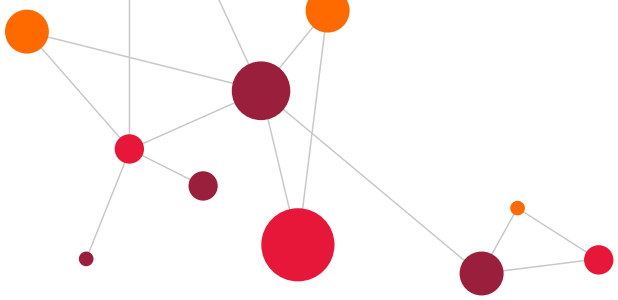
A better approach is use the container orchestration to our advantage.

Docker allows for many options to monitor and restart our containers when they are miss behaving.

One option is to use health checks.

Now, even if our database goes down (or we have a network connection issue), the docker daemon will try to restart the container using the policy we defined. If we are “luck enough” and connectivity is back the myapp service will be restarted and become up and running without any manual intervention.

```
docker ~ > workspace > gs-accessing-data-mysql-complete > complete cat
docker-compose.yml
version: '3.2'
services:
  webapp:
    image: myjavaapp
    depends_on:
      - mysql
    ports:
      - 8080:8080
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080"]
      interval: 30s
      timeout: 10s
      retries: 5
  mysql:
       image: mysql
       ports:
         - "3306:3306"
       environment:
         - MYSQL_USER=springuser
         - MYSQL_PASSWORD=ThePassword
         - MYSQL_DATABASE=db_example
         - MYSQL_ROOT_PASSWORD=root
       volumes:
         - mysql_data:/etc/mysql/conf.d:ro
volumes:
  mysql_data:
docker ~ > workspace > gs-accessing-data-mysql-complete > complete docker-
compose up -d && sleep 20 && docker-compose ps
complete_mysql_1 is up-to-date
```



	Recreating complete_webapp_1 ... done Name Command State Ports --- complete_mysql_1 docker-entrypoint.sh mysqld Up 0.0.0.0:3306->3306/tcp, 33060/tcp complete_webapp_1 java -jar /app/gs-mysql-da ... Up (health: starting) 0.0.0.0:8080->8080/tcp
Let's now try to access the webapp. All is fine.	docker ~ > workspace > myapp curl 'http://localhost:8080/demo/all' [] docker ~ > workspace > myapp curl 'http://localhost:8080/demo/add?user=Mario&email=mario@gmail.com' {"timestamp":"2019-05-09T14:41:42.859+0000","status":400,"error":"Bad Request","message":"Required String parameter 'name' is not present","path":"/demo/add"} docker ~ > workspace > myapp curl 'http://localhost:8080/demo/add?name=Mario&email=mario@gmail.com' Saved docker ~ > workspace > myapp curl 'http://localhost:8080/demo/add?name=Mario&email=mario@gmail.com' Saved docker ~ > workspace > myapp curl 'http://localhost:8080/demo/add?name=Mario&email=mario@gmail.com' Saved docker ~ > workspace > myapp curl 'http://localhost:8080/demo/all' [{"id":1,"name":"Mario","email":"mario@gmail.com"}, {"id":2,"name":"Mario","email":"mario@gmail.com"}, {"id":3,"name":"Mario","email":"mario@gmail.com"}]

Let's now look into volumes.
 	
As we have seen on previous examples, we can mount a local volume on the host filesystem on the container. We used the -v parameter of the docker cli. This is called a bind mount. It's quick and easy to use, but not something we would use in production. It brings also challenges due to permissions since the host and container each use their own accounts.
	docker ~ > workspace > gs-accessing-data-mysql-complete > complete docker volume ls
	DRIVER VOLUME NAME
	local complete_mysql_data
	docker ~ > workspace > gs-accessing-data-mysql-complete > complete docker inspect complete_mysql_data
	[
	{
	"CreatedAt": "2019-05-11T16:34:33+01:00",
	"Driver": "local",
	"Labels": {
	"com.docker.compose.project": "complete",

The recommended approach is to use volumes. They are completely managed by docker. Some advantages:
Easier to backup
Easy to manage using the docker cli
Safely shared across containers
No permission issus

The previous docker-compose file used docker volumes. We can inspect the existing volumes and even check the contents. Avoid directly changing its content.

```
            "com.docker.compose.version": "1.24.0",
            "com.docker.compose.volume": "mysql_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/complete_mysql_data/_data",
        "Name": "complete_mysql_data",
        "Options": null,
        "Scope": "local"
    }
]
docker ~ > workspace > gs-accessing-data-mysql-complete > complete ls -l /var/lib/docker/volumes/complete_mysql_data/_data
ls: cannot access '/var/lib/docker/volumes/complete_mysql_data/_data': Permission denied
docker ~ > workspace > gs-accessing-data-mysql-complete > complete sudo ls -l /var/lib/docker/volumes/complete_mysql_data/_data
[sudo] password for docker:
total 8
-rw-rw-r-- 1 root root   43 abr 25 01:22 docker.cnf
-rw-r--r-- 1 root root 1294 abr 13 13:08 mysql.cnf
```

Lessons learned

Using the very useful spring training and sources – how to create a web app with database integration – from here:

a. https://github.com/spring-guides/gs-accessing-data-mysql.git
b. https://spring.io/guides/gs/accessing-data-mysql/

We learned how containerize a spring boot application. And then how to orchestrate the different services (spring webapp and database) and handle them as one single unit. For this we created our first docker-compose. We used volumes to persist the application state, now even if the containers get destroyed or rebuilt state will not be lost.

The webapp depends on the database. We saw how to make the webapp more resilient to database outages and selfheal.

Revision History

Version	Date	Author	Description
1.0	2019.05.01	Mário Dagot, Jorge Dias	Initial Version