Introduction to container based development

Fabio Oliveira Costa

Contents

1 CONVENTIONS ON THIS GUIDE

Before we start let's stablish some conventions.

Text like the one below is to be executed on the terminal:

```
some command --on the terminal
```

Code like the one below is a file, it's contents should be on some branch on the project repository.

```
1 var thisIs = "an example";
2 console.log("It is " + thisIs); // It is an example
```

Listing 1: example.js

Highlighted code like the below is to indicate a change, we are probably reusing some file and the content changed and I want to make clear where it changed.

```
1 var thisIs = "an example";
2 console.log("It is " + thisIs); // It is an example
3 console.log(thisIs + " that changes");// an example that changes
```

Listing 2: example.js

2 INTRODUCTION

The main goal of developing with containers is to achieve speed, consistency and security. Containers allow us to have a predictable, lightweight and fast environment be it on production or development. Also since containers are created by files we can easily version control a whole installation instead of having to rely on documentations to reproduce the steps to install a piece of software.

Containers borrow the name from shipping containers a mean of transportation that revolutionized the way we ship goods[1]. Be it 100 tons of tuna, a whole house or thousands of unrelated Amazon packages they can be easily transported from anywhere to anywhere. Software containers are the same, the handlers of the containers don't care about what is inside they know how to handle containers and that is what make them flexible.

On the software world developing with containers allow us to eliminate the "it works on my machine", run multiple versions of the same software, create complex architectures on a local machine, make micros services with hundreds of different languages and many more nice things.

In this guide we are going to use Docker and Docker-compose to achieve our goals.

[1] Amazing podcast about shipping containers https://99percentinvisible.org/episode/containers-ships-tugs-port/

3 GETTING STARTED WITH DOCKER

To get started we are going to create a containerized system with docker[2] and for that *we will need to have docker and docker-compose installed*. Follow the official docker installation[3] and docker compose installation[4] for your machine.

3.1 THE CONCEPTS

3.1.1 IMAGE

Think of an image as a read only data, like a music "CD". You could play the PHP CD, perhaps you are feeling a little javazzy and play the java CD or you want to do your own thing so you remix a CD and put the data that you need. The CD *does not have a state* it is what it is and that particular CD will always have the same contents. A image is built with layers on top of a filesystem (fig. 1). Usually you create an image using another image and create it with a series of instructions, if docker already has these instructions on cache it will use that cache instead of building from scratch. Still on our CD metaphor the CD is done with each track separated, you have the drums, the guitar, the bass and the vocals each on a different layer, if you had a karaoke version of the CD the studio one would be the karaoke with the vocals track on top of it.

Images are hosted on repositories ,like docker hub they have a name and a tag, the tag is a version of the image.

Take the node image for instance, Installing *"node"* or *"node:latest"* or *"node:13"* or *"node:13-stretch"* currently all result on the same image.

If node 14 is released then *"node:13"* would still be the same image but *"node"* and *"node:latest"* would change, is important to know which image you are pointing to and usually is better to be as specific as possible like using "node:13.10" so you know which exactly version of node you would be getting.

Ok that is all well and nice if you want exactly an existing image or an off the shelf solution like a postgres database or a rabbitMQ broker but what if you want something custom? For these scenarios you need to create an image, and for that we use an image file (Usually called Dockerfile no extension) and a series of commands.

3.1.2 CONTAINER

A container is a running image and *it has state but does not have persistence.* Think about a CD and somebody manipulating it like a DJ, it is doing changes to the current state of the CD but

[2] https://www.docker.com/
[3] https://docs.docker.com/install/
[4] https://docs.docker.com/compose/install/

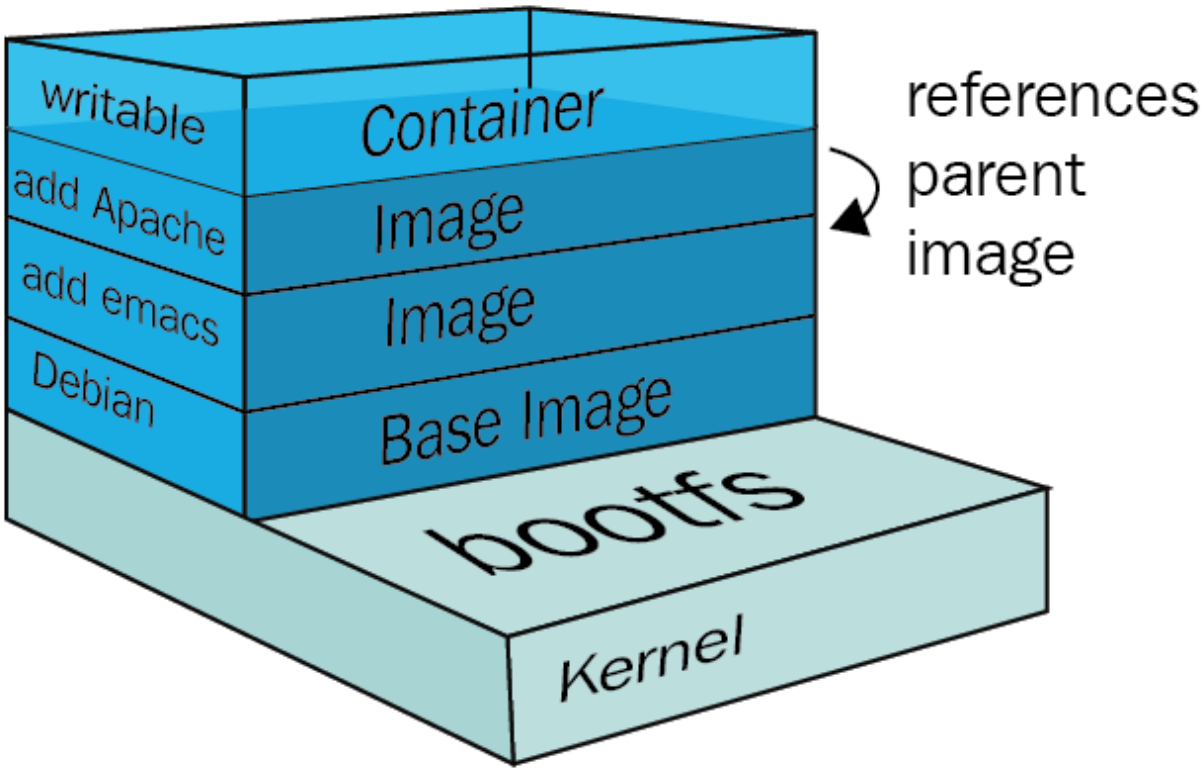

Figure 1: Container using an image with it's layers

by the time the execution is done the CD is back on the case and nothing changed on it. The execution of the CD was the container.

The container is the first really useful entity, is the first thing that can do something and it is what we want to build, manage, scale and destroy. Think of an application that is horizontally scalable, that is the term we use to say we could handle more load by adding more resources, we will want to be able to raise hundreds of containers when they are needed and then put then down when they are not needed anymore.

3.1.3 VOLUMES

A volume is where we have *persistent state*, this is where our changes are not lost, this is how containers read and write persistent data. For example think on an application that counts sales, it would be a pretty lousy application if we lost all sale data as soon as the application was shutdown, for that we persist data on volumes. Most database images will have a volume. Still on the musical analogy our volume is our live recording, we are working on it and it will change, even if we have some fixed tracks we are still changing the data on this one.

3.2 USING OUR FIRST IMAGE

First we need our image on our system so we pull it from the repository[5].

```
docker pull docker/whalesay:latest
```

You should see an output similar to the following:

```
latest: Pulling from docker/whalesay
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest: sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest
docker.io/docker/whalesay:latest
```

See the several pull complete? These are this image layers we were talking before, this image is done with 8 layers, if we had some image that used say the first 4 only the last 4 would be downloaded.

To run the image and thus create a container that would run until exist we do the following.

```
docker run docker/whalesay cowsay "Docker is awesome"
```

You should see the following output:

[5] https://hub.docker.com/

```
< Docker is awesome >
 ------------------
    \
     \
      \
                    ##        .
              ## ## ##       ==
           ## ## ## ##      ===
       /""""""""""""""""___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \______ o          __/
        \    \        __/
          \____\______/
```

That was a very simple use, on the next section we will see how to run something customized and more complex.

3.3 CREATING OUR FIRST IMAGE

The first step to create an image is to choose a image to base on. The most basic image is scratch[6] but usually you would choose some other image like ubuntu, debian, busybox[7], alpine[8] or even some image based on what you need like node.

We are going create a node program(listing 1) his behaviour is the following.

1. try to find a file and read from it;
2. output the content of the file;
3. save the content +1;
4. repeat every 1 second until 19.
5. if no content is found it will create the file and save with 0;

.

All the files for this step are on the projects repository on the firstConainer branch[9].

[6] https://hub.docker.com/_/scratch
[7] https://busybox.net/about.html
[8] https://alpinelinux.org/
[9] https://github.com/drFabio/containerDevelopment/tree/firstContainer

```
const fs = require('fs')
const path = require('path')

const counterFile = path.resolve(__dirname, './counter.txt')

const COUNTER_LIMIT = 20
const INTERVAL = 1000
let counterData = 0
if (fs.existsSync(counterFile)) {
  const fileData = fs.readFileSync(counterFile)
  counterData = parseInt(fileData, 10)
}

const recursiveCounter = () => {
  if (counterData === COUNTER_LIMIT) {
    process.exit(0)
  }
  console.log(counterData)
  counterData++
  fs.writeFileSync(counterFile, counterData)
  setTimeout(recursiveCounter, INTERVAL)
}

recursiveCounter()
```

Listing 1: nodeCounter.js

We will also need our Dockerfile(listing 2) it will create an image from a node based on alpine, create a folder for us to work with, copy a file on that folder, change the working directory to the newly created folder and run the program.

```
FROM node:13-alpine3.10
RUN mkdir -p /app/ourApp/
COPY ./nodeCounter.js  /app/ourApp/
WORKDIR  /app/ourApp/
CMD node ./nodeCounter.js
```

Listing 2: Dockerfile

Then we need to build our image, for convenience we will also tag it by using the -t argument.

```
docker build -t container_dev:first  .
```

You should see an output similar to the following:

```
Sending build context to Docker daemon  70.66kB
Step 1/5 : FROM node:13-alpine3.10
 ---> b74c51039fae
Step 2/5 : RUN mkdir -p /app/ourApp/
 ---> Using cache
 ---> b5464b5e4897
Step 3/5 : COPY ./nodeCounter.js  /app/ourApp/
 ---> Using cache
 ---> 27b39b7f43b6
Step 4/5 : WORKDIR  /app/ourApp/
 ---> Using cache
 ---> 6e1c78cb4629
Step 5/5 : CMD node ./nodeCounter.js
 ---> Using cache
 ---> 50c9fae3a235
Successfully built 50c9fae3a235
Successfully tagged container_dev:first
```

You can see your images list with the command:

```
docker images
```

The output should be a list of all your images with tags, ids and sizes.

```
REPOSITORY                          TAG                 IMAGE ID
container_dev                       first               50c9fae3a235
node                                13-alpine3.10       b74c51039fae
```

With our build image we can run it using the following command:

```
docker run container_dev:first
```

You should see a count from 0 to 19 on your console.

Run it again, if the containers kept state it should start and finish automatically since we are already at 19. What happens is that we see the same output again. Change the directory on which you run it no file will be created. The state of the container is not maintained on subsequent runs, no matter where or how many times you run the program the starting point will be the same as the one on the image. Open more than one terminal and run 2 or more in parallel they will all behave independently even though they are all using "the same file".

FRAME 1: FIRST RUN HIGHLIGHTS

- All changes done on a container even on the filesystem level do not persist.
- The container always start from the image data.
- Containers do not interfere with each other.
- Containers do not write to the host machine.

3.4 BUILDING OUR FIRST VOLUME

A program that can not persist is not so useful so let's modify our program a little so it will count the current file content and then count to that value +20. For that we need to create a volume on the container, this will make a specific folder on the container persist when mounted.Check the official docs[10] if anything is not clear. All the files for this step are on the projects repository on the secondContainer branch[11].

First we need to add a proper volume to our image (listing 3) and modifiy the javaScript code so it ends at the start + 20 (listing 4).

```
1 FROM node:13-alpine3.10
2 RUN mkdir -p /app/ourApp/data
3 COPY ./nodeCounter.js  /app/ourApp/
4 WORKDIR  /app/ourApp/
5 VOLUME /app/ourApp/data/
6 CMD node ./nodeCounter.js
```

Listing 3: Dockerfile

[10] https://docs.docker.com/storage/volumes/
[11] https://github.com/drFabio/containerDevelopment/tree/secondContainer

```
const fs = require('fs')
const path = require('path')

const counterFile = path.resolve(__dirname, './data/counter.txt')

const COUNTER_LIMIT = 20
const INTERVAL = 1000
let counterData = 0
if (fs.existsSync(counterFile)) {
  const fileData = fs.readFileSync(counterFile)
  counterData = parseInt(fileData, 10)
}

const stopCounter = counterData + COUNTER_LIMIT

const recursiveCounter = () => {
  if (counterData === stopCounter) {
    process.exit(0)
  }
  console.log(counterData)
  counterData++
  fs.writeFileSync(counterFile, counterData)
  setTimeout(recursiveCounter, INTERVAL)
}

recursiveCounter()
```

Listing 4: nodeCounter.js

Then build our image with the same name and a new tag.

```
docker build -t container_dev:second  .
```

And you should see some output similar to the one below

```
Sending build context to Docker daemon  80.38kB
Step 1/6 : FROM node:13-alpine3.10
 ---> b74c51039fae
Step 2/6 : RUN mkdir -p /app/ourApp/
 ---> Using cache
 ---> b5464b5e4897
Step 3/6 : VOLUME /app/ourApp
 ---> Running in 4e6cea96ab0d
Removing intermediate container 4e6cea96ab0d
 ---> 5b993cc09174
Step 4/6 : COPY ./nodeCounter.js  /app/ourApp/
 ---> a6c37aee91d2
Step 5/6 : WORKDIR  /app/ourApp/
 ---> Running in da56960902b7
Removing intermediate container da56960902b7
 ---> 85dadd87ad60
Step 6/6 : CMD node ./nodeCounter.js
 ---> Running in 474ecc39084e
Removing intermediate container 474ecc39084e
 ---> f67a63905897
Successfully built f67a63905897
Successfully tagged container_dev:second
```

Now we need to create a volume on the host system so our data actually persist somewhere.

```
docker volume create our-named-volume
```

To make sure out volume was created we need to list the volumes.

```
docker volume ls
```

Your volume should appear on the list

```
DRIVER              VOLUME NAME
local               our-named-volume
```

But where physically our volume is? For that we can inspect the volume and check it's exact path:

```
docker volume inspect our-named-volume
```

And that would output something like:

```
[
    {
        "CreatedAt": "2020-03-20T10:44:48+01:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/our-named-volume/_data",
        "Name": "our-named-volume",
        "Options": {},
        "Scope": "local"
    }
]
```

All things volume related done let's try to run it with a mounting command.

```
docker run -v our-named-volume:/app/ourApp container_dev:second
```

Run it twice and you will notice we start where we stopped. If you open 2 shells and do it a little bit after the first one start you should see they are sharing and modifying the same file (There is a race condition here but we don't mind for now).

We have a way to persist state, actually we even have a way to share the same set of files between multiple containers!

I would like to call attention about the importance of mounting, if we run without the "-v" command we will have the same behaviour as the first time, our state would not persist.

FRAME 2: FIRST VOLUME HIGHLIGHTS

- For a volume to be accessible in the container we need to create it at some location inside the container.
- We need to link a volume with the -v when running the docker run argument.
- Multiple containers can have access to the same volume.

4 MULTIPLE CONTAINERS WITH COMPOSE

Running containers manually is useful but it quickly gets out of hand when you need to manage complex architectures, to make things easy we can use docker-compose. It has a nice and centralized way to create multiple containers at the same time.

Let's port our old example with docker-compose and see how it looks.The official documentation[12] is full of more info and all the files for this step are on the projects repository on the firstDockerCompose branch[13].

Let's create on the same folder that our files are a folder to use as a volume. It is customary to keep all docker-compose related files under the same root, this folder will be called "ourLocalVolume". Then we need to create a compose file[14](listing 5). This compose file is defining a single service called runner, building it from a local dockerFile and creating a volume called ourLocalVolume mounted on the contianer /app/ourApp/data.

```
1 version: '3'
2 services:
3   runner:
4     build: .
5     volumes:
6       - ./ourLocalVolume:/app/ourApp/data
```

Listing 5: docker-compose.yml

[12] https://docs.docker.com/compose/gettingstarted/
[13] https://github.com/drFabio/containerDevelopment/tree/firstDockerCompose
[14] https://docs.docker.com/compose/compose-file/

To run our docker-compose based solution we do the following command.

```
docker-compose up
```

The output should look like:

```
Creating network "containerdevelopment_default" with the default driver
Building runner
Step 1/6 : FROM node:13-alpine3.10
 ---> b74c51039fae
Step 2/6 : RUN mkdir -p /app/ourApp/
 ---> Using cache
 ---> b5464b5e4897
Step 3/6 : COPY ./nodeCounter.js  /app/ourApp/
 ---> f13216878177
Step 4/6 : WORKDIR  /app/ourApp/
 ---> Running in beadc977fd0f
Removing intermediate container beadc977fd0f
 ---> bbe821c708da
Step 5/6 : VOLUME /app/ourApp
 ---> Running in 48e401d69e0f
Removing intermediate container 48e401d69e0f
 ---> d084b4ce684b
Step 6/6 : CMD node ./nodeCounter.js
 ---> Running in d9c0eaf9279e
Removing intermediate container d9c0eaf9279e
 ---> 4c917777860b
Successfully built 4c917777860b
Successfully tagged containerdevelopment_runner:latest
WARNING: Image for service runner was built because it did not already exist. To rebu
Creating containerdevelopment_runner_1 ... done
Attaching to containerdevelopment_runner_1
runner_1  | 0
.....
runner_1  | 18
runner_1  | 19
containerdevelopment_runner_1 exited with code 0
```

If we make any change to out code we would need to rebuild the container with the command below.

```
docker-compose build
```

Also we can run the container forcing it to rebuild with the command bellow:

```
docker-compose up --build
```

Rebuilding every time is not very productive. Our code is also very simple we are not doing anything special with the node image we could easily be more productive if our code changes on the host machine would be transferred to the container. The way to achieve that is by using a volume with our code and our data , actually after all this changes we don't even need to make our own custom Dockerfile we can do everything using docker-compose. We are going to tidy our space a little so see the changes the secondDockerCompose branch[15].

```
1 version: '3'
2 services:
3   runner:
4     image: node:13-alpine3.10
5     volumes:
6       - ./src:/app
7     working_dir: /app
8     command: node ./nodeCounter.js
```

Listing 6: docker-compose.yml

Since our code now lives on a volume along with our data we can update our code without needing to rebuild an image. Also since we are essentially only using a node image we do not need to create a custom image every time.

4.1 MULTIPLE SERVICES, MULTIPLE LANGUAGES

Now that we have some understanding about volumes,images and containers let's see how we could do some very fancy architecture with a few lines of code. We are going to maintain this counter but it will count to up to 20 and then reset and never stop.Also we are going to make a web frontend with php that is going to show the counter as it was at the moment somebody entered the page. Our file architecture will change again so check the thirdDockerCompose branch[16].

[15] https://github.com/drFabio/containerDevelopment/tree/secondDockerCompose
[16] https://github.com/drFabio/containerDevelopment/tree/thirdDockerCompose

One change to our node file will be that we are going to read from a fixed folder that we are going to link to a volume(listing 7).

```
1  const fs = require('fs')
2  const path = require('path')
3
4  const counterFile = path.resolve('/data/counter.txt')
5
6  const COUNTER_LIMIT = 20
7  const INTERVAL = 1000
8  let counterData = 0
9  if (fs.existsSync(counterFile)) {
10   const fileData = fs.readFileSync(counterFile)
11   counterData = parseInt(fileData, 10)
12 }
13
14 const stopCounter = counterData + COUNTER_LIMIT
15
16 const recursiveCounter = () => {
17   if (counterData === stopCounter) {
18     counterData = 0
19   }
20   console.log(counterData)
21   counterData++
22   fs.writeFileSync(counterFile, counterData)
23   setTimeout(recursiveCounter, INTERVAL)
24 }
25
26 recursiveCounter()
```

Listing 7: /src/counter/nodeCounter.js

Also the php will read from a volume and output the contents (listing 8)

```php
<?php
    $counter = file_get_contents('/data/counter.txt');
?>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Counter <?=$counter ?></title>
</head>
<body>
  <h1>Welcome to the counter view</h1>
  <p>At this moment the counter is <?=$counter ?></p>
  <p>Reload for even more counter fun!</p>
</body>
</html>
```

Listing 8: /src/web/index.php

Our docker-compose also need to change (listing 9).

```yaml
version: '3'
services:
  runner:
    image: node:13-alpine3.10
    volumes:
      - ./src/counter:/app
      - ./src/data:/data
    working_dir: /app
    command: node ./nodeCounter.js
  web:
    image: php:7.2-apache
    volumes:
      - ./src/web:/var/www/html/
      - ./src/data:/data
    ports:
      - 8081:80
```

Listing 9: docker-compose.yml

If we run docker-compose up we should be able to see our page on http://localhost:8081. That port is because of the port command[17] that is mapping the port 80 from php to 8081, we could easily make it map to our port 80[18] or any other number we want by just changing that value.

Note that our system now never stop to exit we need to send a SIGTERM[19] by typing "ctrl+c"

Let's take a moment to appreciate what is happening:

1. We created 2 services based on 2 different images.
2. We created 4 volumes being 2 of them shared between different containers.
3. We create a port forwarding from a containers to our host machine.
4. We "installed" a php with apache and node with only 16 lines of code!

Another useful tip is that we often do not want to see our services output and we would rather have them running on the background, for that we can run the up command in daemon mode.

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

If we run it in daemon mode we can no longer stop it with ctrl+C, for that we can run a stop command.

```
docker-compose stop
```

If you have a container running you can do some useful stuff with it. Start both services with "docker-compose up -d". You can do several commands like the ones bellow to enter on the container to change things inside or log data.

```
docker-compose exec runner sh # Enter on the runner container with sh shell
docker-compose exec web bash # Enter on the web container with sh shell
docker-compose logs -f runner # See the logs of the runner
```

[17] https://docs.docker.com/compose/compose-file/#short-syntax-1

[18] 80 is the default port for web, if we map it to port 80 going to localhost would show the page

[19] https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html

4.2 CONTAINER TO CONTAINER COMMUNICATION

We want to make an application that will use redis (A database that holds keys and values) our php web server and our node counter. We are going to change our structure once again because we want to make sure the node_modules[20] folder is built on a volume, the reason for that is that if we are running on different architectures a few packages might be different and our system won't work as predicted[21]. Our structure will change again so check the fourth-DockerCompose branch[22].

The node service now will need to be built as we want to be able to use third party packages. Also it will track the reset and the counter on the redis database (listing 10).

We want our node service to install dependencies when built, that also means that we will need to rebuild when we change dependencies and that our local dependencies are not the same as the container. For that we need to create a new Dockerfile (listing 11).

And finally we need to create our docker-compose(listing 12). Notice that we are passing a environment variable with the redis url to the runner service. All container within a docker-compose by default belong to the same internal network, that means that they can communicate with each other by their containers name (in this case redis_databse).

In these few lines we achieved the following:

1. We created 3 services based on 3 different images.
2. We created 6 volumes being 2 of them shared between different containers, 1 implicitly created (redis one) and one just to use the same file locally and inside the container.
3. We created 2 port forwarding from a containers to our host machine.
4. We linked 2 containers through their internal network.
5. We installed a redis service, node and a php with apache on 26 lines of code.

FRAME 3: DOCKER COMPOSE HIGHLIGHTS

- Docker-compose makes it easy to have your whole architecture on a single file.
- If we are using a base image and just running a file we don't need a dockerfile.
- It is possible for a container to communicate with another on through the network.
- If we want our code to be shared between the host and the container we can do that with volumes.
- We can pass environment data to a container on the docker-compose file-

[20] This is where node store 3rd party code, and we are going to use 3rd party code for the redis connection.
[21] Mainly compiled native code that could run on a MacOS would be wrongly compiled on a linux machine
[22] https://github.com/drFabio/containerDevelopment/tree/fourthDockerCompose

```
const fs = require('fs')
const path = require('path')
const redis = require("redis");

const client = redis.createClient(process.env.REDIS_URL);
const counterFile = path.resolve('/data/counter.txt')

const COUNTER_LIMIT = 20
const INTERVAL = 1000
let counterData = 0
let stopCounter = COUNTER_LIMIT
let resetCount = 0
const recursiveCounter = () => {
  if (counterData === stopCounter) {
    counterData = 0
    resetCount++
    client.set("resetCount", resetCount)
  }
  counterData++
  console.log(counterData)
  client.set("counterData", counterData)
  fs.writeFileSync(counterFile, `Counts: ${counterData} , Reset: ${resetCount}`)
  setTimeout(recursiveCounter, INTERVAL)
}
function startNodeCounter() {
  client.get("counterData", (err, redisCounterData) => {
    counterData = parseInt(redisCounterData || 0, 0)
    client.get("resetCount", (err, redisResetCount) => {
      resetCount = parseInt(redisResetCount || 0, 0)
      recursiveCounter()
    });
  });
}
startNodeCounter()
```

Listing 10: services/counter/src/nodeCounter.js

```
FROM  node:13-alpine3.10
RUN mkdir -p /app/
COPY ./package.json /app/package.json
WORKDIR /app
RUN npm install
VOLUME /app
```

Listing 11: services/counter/Dockerfile

```
version: '3'
services:
  runner:
    build:
      context: ./services/counter/
    volumes:
      - ./services/counter/src:/app/src
      - ./services/data:/data
      - ./services/counter/package.json:/app/package.json
    working_dir: /app
    command: "npm run start"
    depends_on:
      - redis_database
    environment:
      - REDIS_URL=redis://redis_database
  web:
    image: php:7.2-apache
    volumes:
      - ./services/web:/var/www/html/
      - ./services/data:/data
    ports:
      - 8081:80
  redis_database:
    image: redis:5-alpine
    ports:
      - 6379:6379
```

Listing 12: docker-compose.yml

5 FINAL REMARKS

With containers we can make complex systems without worrying about installation. Things that might seem overly complex like a Redis/PHP/Node architecture are easily built and shareable.

Besides the consistence on different environments we can host our projects infrastructure alongside our code without the need of much human intervention.

Knowing how to use networks and volumes are key tools on a container based development and I hope I gave you a glimpse of it's power and removed a little bit of the fear of coding with these tools.

Keep coding be gentle to yourselves and to others and give back what you can when your can.

6 ABOUT THE AUTHOR

Fabio Oliveira Costa is a brazillian developer living in Germany. He is trying to become a better teacher by sharing on and offline. He loves a lot of technologies being javascript his most during relationship, Java the first love that never stops calling and PHP a hookup that paid a lot of bills.

E-mail: fabiocostadev@gmail.com

Glossary

- **container** A standard to ship software.A container is a run-time instance of a docker image. plural. 4–6, 11, 15–17, 20, 21, 24

- **docker** Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Docker is the leading solution on container applications. 4

- **docker-compose** Compose is a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running.. 4, 15

- **dockerfile** A Dockerfile is a text document that contains all the commands you would normally execute manually in order to build a Docker image. Docker can build images automatically by reading the instructions from a Dockerfile.. 5, 9

- **image** Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes.. 5–8, 10, 11, 17, 20, 21

- **layer** a layer is modification to the image, represented by an instruction in the Dockerfile. Layers are applied in sequence to the base image to create the final image. When an image is updated or rebuilt, only layers that change need to be updated, and unchanged layers are cached locally. This is part of why Docker images are so fast and lightweight. The sizes of each layer add up to equal the size of the final image.. 5, 7

- **repository** A repository is a set of Docker images. A repository can be shared by pushing it to a registry server. The different images in the repository can be labeled using tags... 5, 7

- **tag** A tag is a label applied to a Docker image in a repository. Tags are how various images in a repository are distinguished from each other.. 5, 10

- **volume** There are three types of volumes: host(lives on the Docker host's filesystem and can be accessed from within the container), named (a volume which Docker manages where on disk the volume is created, but it is given a name.) and anonymous (similar to a named volume, however, it can be difficult, to refer to the same volume over time when it is an anonymous volumes. Docker handle where the files are stored.). 6, 11, 13, 17, 18, 20, 21