

Devops

Tp1



Database

Basics

Build this image and start a container properly, you should be able to access your database depending on the port binding you choose: localhost:PORT.

Your Postgres DB should be up and running. Connect to your database and check that everything is running smoothly.  
Don’t forget to name your docker image and container.

1. docker build -t mdehaynin/tp1devops .

2. docker run -d -p 8888:5432 --name tp1devops mdehaynin/tp1devops

Re-run your database and [adminer](https://hub.docker.com/_/adminer/" \t "_blank) with --network app-network to enable adminer/database communication. We use -–network instead of -–link because the latter is deprecated.

1. docker rm -f tp1devops

2. docker run -d -p 8888:5432 --name tp1devops mdehaynin/tp1devops

Also, does it seem right to have passwords written in plain text in a file? You may rather define those environment parameters when running the image using the flag -e.

Why should we run the container with a flag -e to give the environment variables?

To not have to write the password into the command prompt.

### Init database

docker network create app-network

It would be nice to have our database structure initialized with the docker image as well as some initial data. Any sql scripts found in /docker-entrypoint-initdb.d will be executed in alphabetical order, therefore let’s add a couple scripts to our image:

docker run -p "8090:8080" --net=app-network --name=adminer -d adminer

**01-CreateScheme.sql**

**02-InsertData.sql**

**When we talk about /docker-entrypoint-initdb.d it means inside the container, so you have to copy your directory's content and the container’s directory.**

**In the Dockerfile I add these 2 lines**

COPY CreateScheme.sql /docker-entrypoint-initdb.d  
COPY InsertData.sql /docker-entrypoint-initdb.d

Rebuild your image and check that your scripts have been executed at startup and that the data is present in your container.

1. docker build -t mdehaynin/tp1devops
2. docker rm -f tp1devops
3. docker run -d -p 8888:5432 --name tp1devops mdehaynin/tp1devops --network app-network

### Persist data

### You may have noticed that if your database container gets destroyed then all your data is reset, a database must persist data durably. Use volumes to persist data on the host disk.

## docker run -d -v C:\Windows\System32\cmd.exe:/var/lib/postgresql/data -p 8888:5432 --net=app-network --name tp1devops mdehaynin/tp1devops

## Backend API

### Basics

For starters, we will simply run a Java hello-world class in our containers, only after will we be running a jar. In both cases, choose the proper image keeping in mind that **we only need a Java runtime**.

Here is a complex **Java Hello World** implementation:

**Main.java**

public class Main {

public static void main(String[] args) {

System.out.println("Hello World!");

}

}

1. Compile with your target Java: javac Main.java.

docker run -it --rm --name my-running-app my-java-app

1. Write dockerfile.

FROM openjdk:11  
COPY . /usr/src/myapp  
WORKDIR /usr/src/myapp  
RUN javac Main.java  
CMD ["java", "Main"]

1. Now, to launch app you have to do the same thing that Basic step 1.
2. docker build -t my-java-app
3. docker run -it --rm --name my-running-app my-java-app

Hello World!

### Multistage build

In the previous section we were building Java code on our machine to have it running on a docker container. Wouldn’t it be great to have Docker handle the build as well? You probably noticed that the default openjdk docker images contain... Well... a JDK! Create a multistage build using the [Multistage](https://docs.docker.com/develop/develop-images/multistage-build/).

FROM maven:3.8.6-amazoncorretto-17 AS *myapp-build*ENV *MYAPP\_HOME* /opt/myapp  
WORKDIR $*MYAPP\_HOME*COPY pom.xml .  
COPY src ./src  
RUN mvn package -DskipTests  
  
# Run  
FROM amazoncorretto:17  
ENV *MYAPP\_HOME* /opt/myapp  
WORKDIR $*MYAPP\_HOME*COPY --from=*myapp-build* $*MYAPP\_HOME*/target/\*.jar $*MYAPP\_HOME*/myapp.jar  
  
ENTRYPOINT java -jar myapp.jar

1-2 Why do we need a multistage build? And explain each step of this dockerfile.

the second stage (Run) creates the minimal runtime image with only the compiled binary. This results in a much smaller and more secure final image.

**Check** ✓

#### Backend API

Let’s now build and run the backend API connected to the database. You can get the zipped source code here: [simple-api](https://github.com/takima-training/simple-api-student).

Adjust the configuration in simple-api/src/main/resources/application.yml (this is the application configuration). How to access the database container from your backend application? Use the deprecated --link or create a docker network.

1. Write in application.yml

url: jdbc:postgresql://tp1database:5432/db  
username: usr  
password: pwd

1. Put Database and simple-api in the same network
2. Connect to 8080 (the port of simpleapi)

Once everything is properly bound, you should be able to access your application API, for example on: [/departments/IRC/students](http://localhost:8080/departments/IRC/students).

**Check** ✓

## Http server

### Basics

Choose an appropriate base image.

Create a simple landing page: index.html and put it inside your container.

It should be enough for now, start your container and check that everything is working as expected.

Here are commands that you may want to try to do so:

* docker stats

Continous stats of containers

* docker inspect
* docker logs

docker exec tp1-server cat > config

### Reverse proxy

We will configure the http server as a simple reverse proxy server in front of our application, this server could be used to deliver a front-end application, to configure SSL or to handle load balancing.

So this can be quite useful even though in our case we will keep things simple.

Here is the documentation: [Reverse Proxy](https://httpd.apache.org/docs/2.4/en/howto/reverse_proxy.html).

Add the following to the configuration, and you should be all set:

ServerName localhost

<VirtualHost \*:80>  
ProxyPreserveHost On  
ProxyPass / http://simpleapi:8080/  
ProxyPassReverse / http://simpleapi:8080/  
</VirtualHost>  
LoadModule proxy\_module modules/mod\_proxy.so  
LoadModule proxy\_http\_module modules/mod\_proxy\_http.so

Why do we need a reverse proxy ?

To optimize the pages load.

## Link application

### Docker-compose

1- Install [docker-compose](https://docs.docker.com/compose/install/) if the docker compose command does not work .

You may have noticed that this can be quite painful to orchestrate manually the start, stop and rebuild of our containers. Thankfully, a useful tool called [docker-compose](https://docs.docker.com/compose/) comes in handy in those situations.

2- Let’s create a docker-compose.yml file with the following structure to define and drive our containers:

version: '3.7'  
  
services:  
 backend:  
 container\_name: simpleapi  
 build: ./simple-api-student-main  
 networks:  
 - my-networkdepends\_on:  
 - databasedatabase:  
 container\_name: tp1database  
 build: ./Database  
networks:  
 - my-network  
  
 httpd:  
 build: ./HTTPServerports:  
 - "8082:80"networks:  
 - my-network  
 depends\_on:  
 - backendnetworks:  
 my-network: {}

The docker-compose will handle the three containers and a network for us.

Once your containers are orchestrated as services by docker-compose you should have a perfectly running application, make sure you can access your API on [localhost](http://localhost/).

**Note**

The ports of both your backend and database should not be opened to your host machine.

**Tip**

Why is **docker-compose** so important?

To have a simplier way of running the application without initialize=ing everything manually

**Question**

1-3 Document docker-compose most important commands. 1-4 Document your docker-compose file.

**Check** ✓

A working 3-tier application running with docker-compose.

## Publish

Your docker images are stored locally, let’s publish them, so they can be used by other team members or on other machines.

You will need a [Docker Hub](https://hub.docker.com/) account.

1- Connect to your freshly created account with docker login.

2- Tag your image. For now, we have been only using the latest tag, now that we want to publish it, let’s add some meaningful version information to our images.

docker tag my-database USERNAME/my-database:1.0

3- Then push your image to dockerhub:

docker push USERNAME/my-database:1.0

Dockerhub is not the only docker image registry, and you can also self-host your images (this is obviously the choice of most companies).

docker tag tp1-database mdehaynin/tp1-database:1.0

docker push mdehaynin/tp1-database:1.0

docker tag tp1-backend mdehaynin/tp1-backend:1.0

docker push mdehaynin/tp1-backend:1.0

docker tag tp1-httpd mdehaynin/tp1-httpd:1.0

docker push mdehaynin/tp1-httpd:1.0

Once you publish your images to dockerhub, you will see them in your account: having some documentation for your image would be quite useful if you want to use those later.

Why do we put our images into an online repo ?

To share it and also be able to have separated backup.