TP part 01 - Docker

 Checkpoint: call us to check your results

 Ask yourself: how? why?

 Point to document/report

Goals

Good practice

Do not forget to document what you do along the steps, the documentation you provide will be evaluated as your report.
Create an appropriate file structure, 1 folder per image.

Target application

3-tiers application:

- HTTP server
- Backend API
- Database

For each of those applications, we will follow the same process: choose the appropriate docker base image, create and configure this image, put our application specifics inside and at some point have it running. Our final goal is to have a 3-tier web API running.

Base images

HTTP server:

https://hub.docker.com/_/httpd

Backend API:

https://hub.docker.com/_/openjdk

Database:

https://hub.docker.com/_/postgres

Database

Basics

We will use the image: postgres:11.6-alpine.
Let's have a simple postgres server running, here would be a minimal Dockerfile:

```
FROM postgres:11.6-alpine

ENV POSTGRES_DB=db \
   POSTGRES_USER=usr \
   POSTGRES_PASSWORD=pwd
```

Build this image and start a container properly, you should be able to access your database depending on the port binding you chose: localhost:PORT.
Your Posgres DB should be up and running. Connect to your database and check that everything is running smoothly.
You don't have a SQL client? It's okay, just run adminer (https://hub.docker.com/_/adminer/).
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".

Init database

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:
01-CreateScheme.sql

```
CREATE TABLE public.departments
(
 id       SERIAL      PRIMARY KEY,
 name     VARCHAR(20) NOT NULL
);

CREATE TABLE public.students
(
 id              SERIAL      PRIMARY KEY,
 department_id   INT         NOT NULL REFERENCES departments (id),
 first_name      VARCHAR(20) NOT NULL,
 last_name       VARCHAR(20) NOT NULL
);
```

02-InsertData.sql

```
INSERT INTO departments (name) VALUES ('IRC');
INSERT INTO departments (name) VALUES ('ETI');
INSERT INTO departments (name) VALUES ('CGP');


INSERT INTO students (department_id, first_name, last_name) VALUES (1,
'Eli', 'Copter');
INSERT INTO students (department_id, first_name, last_name) VALUES (2,
'Emma', 'Carena');
INSERT INTO students (department_id, first_name, last_name) VALUES (2,
'Jack', 'Uzzi');
INSERT INTO students (department_id, first_name, last_name) VALUES (3,
'Aude', 'Javel');
```

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

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.
`-v /my/own/datadir:/var/lib/postgresql/data`
Check that data survives when your container gets destroyed.

 Checkpoint: a fully working containerized database

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!");
  }
}
```

Compile with your target Java: “javac Main.java”
Build your image from the proper jre image, add the compiled java (aka bytecode, aka .class) and run the Java with: “java Main” command.

Here you have it the first glimpse of your backend application. In the next step we will simply complexify the build (using maven instead of a minimalistic javac) and execute a jar instead of a simple .class.

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 are containing… Well.. a JDK ! Create a multistage build using the https://docs.docker.com/develop/develop-images/multistage-build/.
Your Dockerfile should look like this:

```
FROM openjdk:11
# Build Main.java
# ...

FROM openjdk:11-jre
# Copy resource from previous stage
COPY --from=0 /usr/src/Main.class .
# Run java code with the JRE
# ...
```

Backend simple app

We will deploy a Springboot application providing a simple API with a single greeting endpoint.
Create your Springboot application on: https://start.spring.io/
Use the following config:

- Project: Maven
- Language: Java 11
- Spring Boot: 2.2.4
- Packaging: Jar
- Dependencies: Spring Web (for now)

Generate the project and give it a simple GreetingController class:

```
package fr.takima.training.simpleapi.controller;

import org.springframework.web.bind.annotation.*;

import java.util.concurrent.atomic.AtomicLong;

@RestController
```

```java
public class GreetingController {

  private static final String template = "Hello, %s!";
  private final AtomicLong counter = new AtomicLong();

  @GetMapping("/")
  public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {
    return new Greeting(counter.incrementAndGet(), String.format(template, name));
  }

  class Greeting {

    private final long id;
    private final String content;

    public Greeting(long id, String content) {
      this.id = id;
      this.content = content;
    }

    public long getId() {
      return id;
    }

    public String getContent() {
      return content;
    }
  }
}
```

You can now build and start your application, of course you will need maven and a jdk-11. How convenient would it be to have a virtual container to build and run our simplistic API? Oh wait, we have docker, here is how you could build and run your application with Docker:

```
# Build
FROM maven:3.6.3-jdk-11 AS myapp-build
ENV MYAPP_HOME /opt/myapp
WORKDIR $MYAPP_HOME
COPY pom.xml .
COPY src ./src
RUN mvn package -DskipTests

# Run
FROM openjdk:11-jre
ENV MYAPP_HOME /opt/myapp
WORKDIR $MYAPP_HOME
COPY --from=myapp-build $MYAPP_HOME/target/*.jar $MYAPP_HOMME/myapp.jar

ENTRYPOINT java -jar myapp.jar
```

 Checkpoint: a working Springboot application with a simple HelloWorld endpoint

Did you notice that maven downloads all libraries on every image build? You can contribute to saving the planet caching libraries when maven pom file has not been changed by running the goal: “mvn dependency:go-offline”.

Backend API

Let's now build and run the backend API connected to the database. You can get the zipped source code here:
https://github.com/takima-training/sample-application-students/releases/download/simple-api/simple-api.zip
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.
Once everything is properly bound you should be able to access your application API, for example on: http://localhost:8080/departments/IRC/students

```
[

  "id": 1,
  "firstname": "Eli",
  "lastname": "Copter",
  "department": {
    "id": 1,
    "name": "IRC"
  }
]
```

Ask yourself: explore your API other endpoints, have a look at the controllers in the source code

Checkpoint: a simple web API on top of your database

Http server

Useful links:

- https://hub.docker.com/_/httpd
- http://httpd.apache.org/docs/2.4/getting-started.html

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 a couple command you may want to try to do so:

- docker stats
- docker inspect
- docker logs

Configuration

You are using the default apache configuration and it will be enough for now, you use yours by copying it in your image.
Use “docker exec” to retrieve this default config from your running container “/usr/local/apache2/conf/httpd.conf”.

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: https://httpd.apache.org/docs/2.4/en/howto/reverse_proxy.html

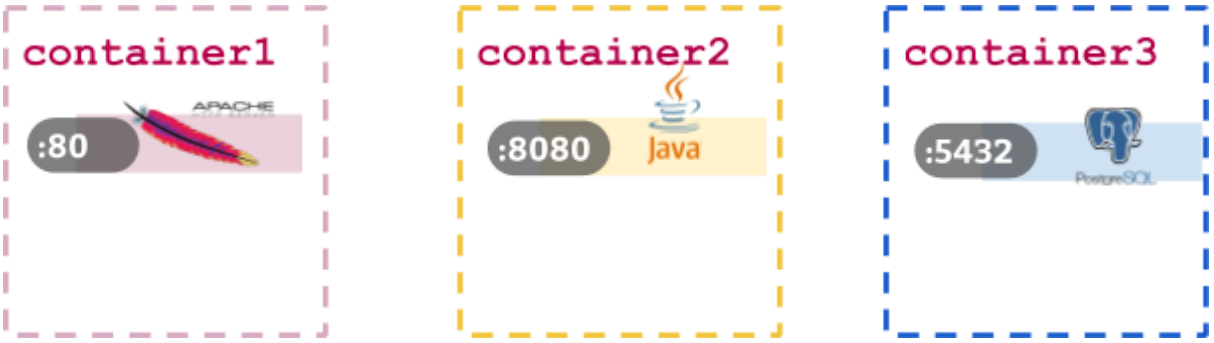
Add the following to the configuration and you should be all set:

```
ServerName localhost

<VirtualHost *:80>
  ProxyPreserveHost On
  ProxyPass / http://YOUR_BACKEND_LINK:8080/
  ProxyPassReverse / http://YOUR_BACKEND_LINK:8080/
</VirtualHost>
```

Checkpoint: a working application through a reverse proxy

Link application



Docker-compose

Install docker-compose: https://docs.docker.com/compose/install/

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 comes in handy in those situations (https://docs.docker.com/compose/).
Let's create a docker-compose.yml file with the following structure to define and drive our containers:

```
version: '3.7'
services:
 backend:
   build:
     #TODO
   networks:
     #TODO
   depends_on:
     #TODO

 database:
   build:
     #TODO
   networks:
     #TODO

 httpd:
   build:
     #TODO
   ports:
     #TODO
   networks:
     #TODO
   depends_on:
     #TODO

networks:
 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 http://localhost/.
Note that the ports of both your backend and database should not be opened to your host machine.

 Ask yourself: what are the useful docker-compose sub-commands?

 Checkpoint: a working 3-tier application runned by 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 an account on: https://hub.docker.com/

Connect to your freshly created account with: *docker login*
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
Then push your image to dockerhub:
docker push USERNAME/my-database
Dockerhub is not the only docker image registry and you can also self host your images (this is obviously the choice of most companies).

Once you published 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.