DevOps practical work

November 2023

Émile Royer

Contents

1 Setting up Docker

Remark: adding the regular user to the `docker` group defeats all security isolation (it makes the user a quasi-root), so all Docker commands are run with `sudo`.

1.1 Creating the database image

We create a "database" directory to house the configuration for our database image.

There, we create a `Dockerfile` file instructing to create an image starting from the Alpine Linux version of the Postgres base image. Environment variables are provided for the configuration.

```
FROM postgres:14.1-alpine

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

The image is then build, and tagged "me/tp-database":

```
sudo docker build -t me/tp1-database database/
```

Once the image is build, a container can be started, binding it to the port 5000 on the host machine (Postgres' default port is 5432):

```
sudo docker run -p 5000:5432 --name database-layer me/tp-database
```

It runs without error.

Let's remove it since we'll want to reuse its name afterwards for a new instance:

```
sudo docker stop database-layer
sudo docker remove database-layer
```

We now create a network to isolate containers together:

```
sudo docker network create app-network
```

```
sudo docker run -p 5000:5432 --name database-layer --network app-network me/
tp-database
```

We use *Adminer* as a client to connect to the database and view its contents. It too is available as a Docker image:

```
sudo docker run -d \
     -p "8090:8080" \
```

```
    --net=app-network \
    --name=adminer \
    adminer
```

The database is currently empty — after all, it was just created.

To initialise the database, we create the files `01-CreateScheme.sql` and `02-InsertData.sql`. They are copied in the `/docker-entrypoint-initdb.d` directory of the container when it is started. The first file creates the tables "departments" and "students"; the second file inserts some data in them.

```
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
);
```

Listing 1: The `01-CreateScheme.sql` file.

The necessary instructions are added to the Dockerfile:

```
FROM postgres:14.1-alpine

COPY 01-CreateScheme.sql /docker-entrypoint-initdb.d/
COPY 02-InsertData.sql /docker-entrypoint-initdb.d/

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

And the image is rebuilt like before.

The data is indeed inserted in the database, as seen in Figure 1.

Lastly, we add a *bind volume* to the container for persistence:

```
sudo docker run -p 5000:5432 -v ./database/persistence:/var/lib/postgresql/data
--name database-layer --network app-network me/tp-database
```

This means the files from the container located in the directory `/var/lib/postgresql/data` are copied to the host, in the directory `database/persistence`.

With the data being kept on the host's disk, it stays between executions, even if the container is destroyed. The data created by Postgres looks like:

```
$ sudo ls database/persistence/
base           pg_ident.conf  pg_serial     pg_tblspc    postgresql.auto.conf
global         pg_logical     pg_snapshots  pg_twophase  postgresql.conf
pg_commit_ts   pg_multixact   pg_stat       PG_VERSION   postmaster.opts
pg_dynshmem    pg_notify      pg_stat_tmp   pg_wal
pg_hba.conf    pg_replslot    pg_subtrans   pg_xact
```

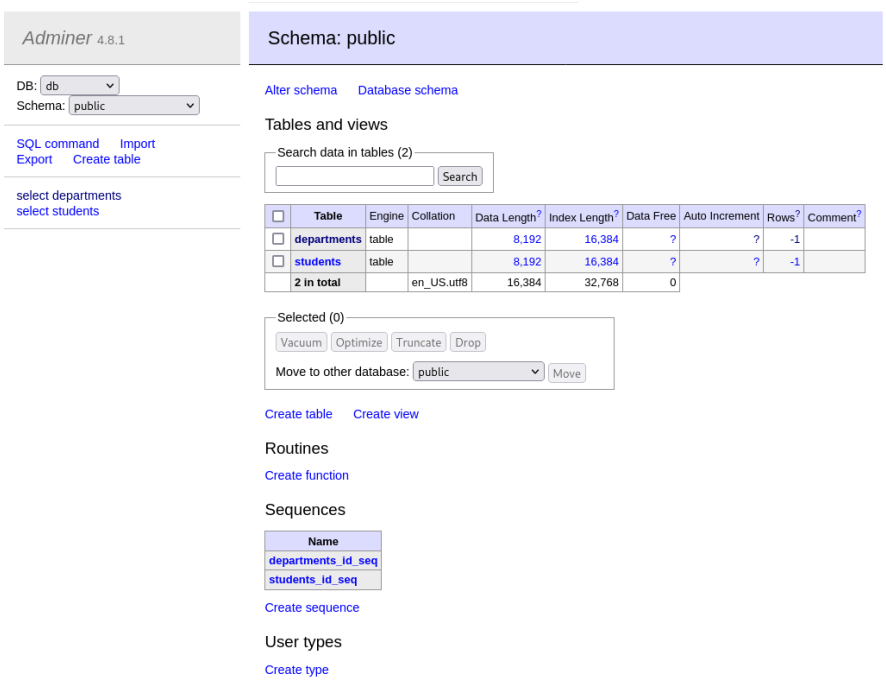

Figure 1: The database with the two tables added, in Adminer.

The database's password should not be written in a Dockerfile: this file is distributed to anyone, and should not contain any sensitive information. Instead, environment variables are better provided with command line options.

The cleaned-up Dockerfile is then:

```
FROM postgres:14.1-alpine

COPY 01-CreateScheme.sql /docker-entrypoint-initdb.d/
COPY 02-InsertData.sql /docker-entrypoint-initdb.d/
```

And the command to instantiate a container, including environment variables:

```
sudo docker run --rm -d \
     -p 5000:5432 \
     -v ./database/persistence:/var/lib/postgresql/data \
     -e POSTGRES_DB=db \
     -e POSTGRES_USER=usr \
     -e POSTGRES_PASSWORD=pwd \
     --network app-network \
     --name database-layer \
     me/tp-database
```

Question 1-1: *Document your database container essentials: commands and Dockerfile.*
To create an image:

- Create a file named `Dockerfile`,
- Use the keyword `FROM` to start from a base image: `FROM alpine:3.18`,
- You can copy files from the host to the container using the `COPY` keyword.
- Build an image from a Dockerfile with `sudo docker build -t <image-name> <path>`.

To create containers:

- The base command is `sudo docker run <image-name>`,
- Specify a port mapping to expose the container to with the `-p` option,
- A container can be named with the option `--name`,
- The `-e` option can be used to provide environment variables.

1.2 Create the back-end

We'll now create a Java app in the `backend/` directory.

1.2.1 Step 1, a dead simple program

Here is a classic "hello world" Java program.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Listing 2: Java's *Hello World*

Here is a Dockerfile that can run it (notice the `java Main` command to run the class):

```
FROM amazoncorretto:17

COPY Main.class /

CMD ["java", "Main"]
```

This image is built and run in a container:

```
sudo docker build -t me/tp1-backend backend/
```

```
sudo docker run -d \
     --name backend-layer \
     me/tp1-backend
```

It works, printing "Hello World!".

1.2.2 Simple API

Let's create a more complex app.

Spring initializr is used to create a Spring demo. This scaffold is extracted in `backend/`.

The directory now has (we kept the previous `Main.java` file):

```
$ ls backend/
Dockerfile HELP.md Main.class Main.java mvnw mvnw.cmd pom.xml src
```

We now use this multi-stage Dockerfile to compile and run the program:

```
# Build
FROM maven:3.8.6-amazoncorretto-17 AS myapp-build
ENV MYAPP_HOME /opt/myapp
WORKDIR $MYAPP_HOME
COPY pom.xml .
RUN mvn dependency:go-offline
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
```

With `RUN mvn dependency:go-offline` after the `pom.xml` file is copied to save the dependencies between builds.

Building the image with the new Dockerfile and instantiating a container is as was done before:

```
sudo docker build -t me/tp1-backend backend/
```

```
sudo docker run -d \
     -p 8000:8080 \
     --network app-network \
     --name backend-layer \
     me/tp1-backend
```

When visiting http://[::]:8080?name=Émile[1], the following response is obtained, as expected:

```
{
    "id":2,
    "content":"Hello, Émile!"
}
```

Question 1-2: *Why do we need a multistage build? And explain each step of this Dockerfile.*

A multistage build is a way to optimise Dockerfiles. They allow both a file to be more readable and to reduce the resulting image size.

To do this, the image is first primed with a first base image, a building step is run, then a new image is started (it can have a different base). The necessary files can be copied from one step to the other. Once all steps are finished, all the images are discarded but the final one.

This means the setup needed to compile and build an app is not kept in the final image, and only the executable is copied, allowing the remaining image to have a reduced size.

Because each step is conscripted to its own stage, its own image, the relationship between them can be more clearly laid down.

The steps:

1. Select `maven` as the base image (in order to build with Maven).
2. Create an environment variable `$MYAPP_HOME`, set to `/opt/myapp`.
3. Go to `/opt/myapp`.
4. Copy the `pom.xml` file there.
5. Download the dependencies offline (for Docker layer cache).
6. Copy full source tree to `./src`.
7. Compile the app.

[1] IPv6 is the present, but you can replace `[::]` with `localhost` if it doesn't work.

8. Create a new image based `amazoncorretto` (Maven is not needed anymore once it is built).
9. Re-create the environment variable `$MYAPP_HOME`, set to `/opt/myapp` (its a new image), go to it.
10. Copy the Jar produced during the build.
11. Set the image's entrypoint to run the obtained Jar.

1.2.3 Full API

We'll run use a more complete Java backend that provides a more complex API. Since we a right now focused on the operations more than the development, we'll use a ready-made Java application.

We download it from https://github.com/takima-training/simple-api-student.

The Dockerfile used previously for the Spring demo was not specific, and will function for most Maven-based Java project, including this one. This means we can reuse it..

Once built with the same name, it is run as before:

```
sudo docker run -d \
     -p 8000:8080 \
     --network app-network \
     --name backend-layer \
     me/tp1-backend
```

But it fails, on the ground of database access.

Because it is more complex, this program needs to use the database we created earlier. Reusing the same environment variables as previously for a clean setup, information in how to contact the database is added to `application.yml`:

```
spring:
  jpa:
    properties:
      hibernate:
        jdbc:
          lob:
            non_contextual_creation: true
    generate-ddl: false
    open-in-view: true
  datasource:
    url: jdbc:postgresql://database-layer:5432/${POSTGRES_DB}
    username: ${POSTGRES_USER}
    password: ${POSTGRES_PASSWORD}
    driver-class-name: org.postgresql.Driver
management:
 server:
   add-application-context-header: false
 endpoints:
   web:
     exposure:
       include: health,info,env,metrics,beans,configprops
```

In `jdbc:postgresql://database-layer:5432/${POSTGRES_DB}`, `database-layer` is the name of the container running the database. In a network, Docker automatically fills in the hostnames.

Running a container with the application and supplying the appropriate variables looks now like:

```
sudo docker run --rm -d \
     -p 8000:8080 \
     --network app-network \
     -e POSTGRES_DB=db \
     -e POSTGRES_USER=usr \
     -e POSTGRES_PASSWORD=pwd \
     --name backend-layer \
     me/tp1-backend
```

1.3 HTTP server

No modern web app is complete without at least one reverse proxy. So let's add one.

First, let's create a new directory for this part, `http/`.

1.3.1 A basic server to begin with

At its core, a reverse proxy is a classic HTTP server. So let's setup one that displays an HTML docuemnt to show it works. We'll use Apache's *httpd* for this.

A Dockerfile with a basic configuration for this task looks like :

```
FROM httpd:2.4
COPY ./index.html /usr/local/apache2/htdocs/
```

Let's build the image and run a container from it:

```
sudo docker build -t me/tp1-http http/
```

```
sudo docker run -d \
     -p 8888:80 \
     --name http-layer \
     me/tp1-http
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="color-scheme" content="light dark">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>A good HTML "Hello world"</title>
    <style>
      body{
        margin: auto;
        max-width: 80ch;
        font: sans-serif;
      }
    </style>
  </head>
  <body>
    <h1>Hello world!</h1>
```

```
  </body>
</html>
```

Listing 3: The `index.html` file used for the demo

Visiting http://[::]:8888 displays our HTML file.

1.3.1.1 A few Docker commmands

`docker stats` shows for each running container some information about its resources use, like CPU fraction used or memory comsumed.

`docker inspect <container>` gives detailed information about th etarget container, by default in a JSON format.

`docker logs <container>` shows the standard output of the target running container. In the HTTP server's case, this include the connexion logs.

1.3.2 Reverse proxy

A reverse proxy is commonly used to hide or abstract a service's architecture, in case there are many servers.

To set one up, we need to configure our HTTP server to behave like one. Let's retrieve the server's default configuration to base ours on it. We use `docker cp`, which can copy files between containers and the host.

```
sudo docker cp http-layer:/usr/local/apache2/conf/httpd.conf http/httpd.conf
```

The file is quite long, so not shown here.

Here are the changes that were made to the configuration to turn it into a reverse proxy:

```
--- http/httpd.conf
+++ http/httpd.conf
@@ -225,4 +225,14 @@
 #

+<VirtualHost *:80>
+   ProxyPreserveHost On
+   ProxyPass / http://backend-layer:8080/
+   ProxyPassReverse / http://backend-layer:8080/
+</VirtualHost>
+LoadModule proxy_module modules/mod_proxy.so
+LoadModule proxy_http_module modules/mod_proxy_http.so
+
 #
 # ServerAdmin: Your address, where problems with the server should be
@@ -230,4 +240,5 @@
 #
 #ServerName www.example.com:80
+ServerName localhost:80

 #
```

The new configuration file must now be copied in the image to replace the default one. Since the server will redirect all requests, we don't need the HTML file anymore.

```
FROM httpd:2.4
# COPY ./index.html /usr/local/apache2/htdocs/
COPY ./httpd.conf /usr/local/apache2/conf/httpd.conf
```

After building, running a container (in the network) is done with:

```
sudo docker run --rm -d \
     -p 80:80 \
     --network app-network \
     --name http-layer \
     me/tp1-http
```

The proxy runs on the port 80, the default port for HTTP, so we don't have to specify a port when doing requests.

And now, it behaves transparently, like if we were talking directly to the back-end.

1.4 Wrapping it all with Compose

Executing the Docker commands works well, but it is a bit tedious. Fortunately, Compose is here for us.

With Compose, we list all the properties we want our containers to have, and they are automagically provisioned. Once we don't need them, we ask Compose to stop them, and they are removed.

To have the same functionnality as we had spinning the containers up manually, we write the following `compose.yaml` file:

```
version: '3.7'

services:
    backend:
        build: ./backend
        container_name: backend-layer
        networks:
            - my-network
        depends_on:
            - database
        env_file: .env

    database:
        build: ./database
        container_name: database-layer
        networks:
            - my-network
        volumes:
            - db-persitence:/var/lib/postgresql/data
        env_file: .env

    httpd:
        build: ./http
        container_name: frontend-layer
        ports:
            - "80:80"
        networks:
            - my-network
```

```
        depends_on:
            - backend

networks:
    my-network: {}

volumes:
    db-persitence: {}
```

Listing 4: Our `compose.yaml` recipe file.

Question 1-3: *Document docker-compose most important commands.*

In symetry: `sudo docker compose up` to create the containers and `sudo docker compose down` to remove them.

Then, `sudo docker compose logs` is really useful to see what happens in the containers if they are in detached mode.

Question 1-4: *Document your docker-compose file.*

The containers need to live in the same network, so to communicate with each other but be isolated from potential other containers. This is done with the `networks` key. The actual name of the container does not really matter since we only have one and it is name only used in the file.

Similarly, it is better to declare a volume for the database storage, allowing to restore the state at startup and not have to recreate all entries. We create a named volume in the `volumes` key and bind it in the databse service definition.

Because we hard-coded the names of the containers in the URLs of the various configurations, we need to keep the same ones here. Hence the `container_name` keys. Set the container names

The database and back-end containers need some environment variables to work well. Since some are the same, an `.env` file is created to host the variables, and is provided to the containers with the `env_file` key.

Because the startup order of the containers can have an impact (the proxy must be able to reach the back-end, which must access the database), we declare a dependency tree with the `depends_on` key.

1.5 Publishing the images

I published the images as `e10e3/ops-database`, `e10e3/ops-backend`, and `e10e3/ops-frontend`[2].

Let's use the database as an example of how it's done.

First, building the image, with tag and version:

```
sudo docker build -t e10e3/ops-database:1.0 database/
```

[2]This last name will cause problems later and is changed in a following section.

Then, publishing it:

```
sudo docker push e10e3/ops-database:1.0
```

The result is: https://hub.docker.com/repository/docker/e10e3/ops-database/

Question 1-5: *Document your publication commands and published images in DockerHub.*

To push an image to Docker Hub, you must namespace it with you Docker Hub username first. In my case, this is *e10e3*. The tag of an image is set at build time (with the `--tag` option) or with the `docker image rename` command.

In order to push an image to your account, it is necessary to first log in. This is done with the command `docker login`.

Once an image is created and the account information entered, the image can be sent online with the `docker push` command.

2 Using Github Actions

The whole project — with the database, back-end and proxy — was uploaded in a Git repository at https://github.com/e10e3/devops.

Let's focus on the back-end, arguably the most brittle part of our system for now: the database and the HTTP server are more complex, but we only configured them and suppose they are both more mature and more tested. The Java app — while we didn't write it ourselves — is quite new and it was created from the ground up (albeit using frameworks that help a bit).

Thankfully, the back-end has some tests written to ensure its functionality. To run them all, we need to build the app. This is easily done in one command with Maven:

```
sudo mvn clean verify
```

This launches two Maven targets: *clean* will remove the temporary directories, like `target` (the build directory). *verify* will the the application's tests[3].

Some of the tests are more encompassing (they are called *integration tests*) and run in specialised Docker containers in order to have total control on their environment.

Question 2-1: *What are testcontainers?*

Testcontainers is a testing library (https://testcontainers.org) that run tests on the codebase in lightweight Docker containers. In the context of Java apps, they execute JUnit tests.

2.1 Yamllint is your friend

From now on, we'll use a tonne of YAML files. But the tools they configure will fail if the file are improperly formatted.

Yamllint is here to help! Running it on a file will give you a list of all malformed items, like wrong indentation.

[3] Maven has goal dependencies. This means that in order to excute the goal *verify* (the unit tests), it will first run the unit tests (*validate*) and build the program (*compile*), between others.

```yaml
# perhaps.yaml
key: on
enum:
  - name: Test
    kind: None
  - name: Other
```

```
$ yamllint perhaps.yaml
perhaps.yaml
  1:1       warning  missing document start "---"  (document-start)
  1:6       warning  truthy value should be one of [false, true]  (truthy)
  4:10      error    syntax error: mapping values are not allowed here (syntax)
```

YAML awaits you!

2.2 Creating a workflow

Part of the DevOps philosophy is CI — continuous integration. Because our Git repository is hosted on GitHub, let's use GitHub's own CI runner, GitHub Actions, to create a testing pipeline.

We created the file `.github/workflows/main.yml` to declare a workflow that runs the integration tests:

```yaml
name: CI devops 2023
on:
  #to begin you want to launch this job in main and develop
  push:
    branches:
      - main
      - develop
  pull_request:

jobs:
  test-backend:
    runs-on: ubuntu-22.04
    steps:
     #checkout your github code using actions/checkout@v2.5.0 (upgraded to v4)
      - uses: actions/checkout@v4

      #do the same with another action (actions/setup-java@v3) that enable to
setup jdk 17
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          distribution: 'temurin'
          java-version: '17'
          cache: 'maven'

    #finally build your app with the latest command (not forgetting the tests)
      - name: Build and test with Maven
        run: mvn -B clean verify --file backend/pom.xml
```

Note: *checkout* was updated to v4 because the previous version were deprecated. The JDK version installed is version 17 (the tried-and-true LTS version of the last few years), from the Eclipse *temurin* distribution.

The program is build with the command `mvn -B clean verify --file backend/pom.xml`. The option `-B` is for "batch" (no colours are needed for automation), and `--file` to indicate Maven it needs to use the `pom.xml` file at the root of the backend as the job runs at the root of the repository.

After a push to the remote repository, it runs with success.

Question 2-2: *Document your Github Actions configurations.*

The workflow reacts to two kinds of events: commit pushes on the *main* and *develop* branches, and all pull requests.

When the workflow starts, it creates an environment on a machine running Ubuntu 22.04, and checkouts the code that awoke it.

Then, it installs the Java JDK 17 and Maven. Once this is done, it runs the back-end's unit tests.

2.3 Automatically publish the images

In the previous part we created Docker images for the project's components. The other great keyword of DevOps is CD — continuous deployment. Using GitHub Actions, we can create a workflow that published the images to Docker Hub when code is pushed, no human action required.

2.3.1 Action secrets

Publishing to Docker Hub implies to log in, but we don't want to have our credentials visible to anyone and their dog in clear on GitHub. Fortunately, the forge has a solution, *secrets*.

We created the secrets `DOCKER_HUB_TOCKEN` and `DOCKER_HUB_USERNAME` (the names are case-insensitive). `DOCKER_HUB_TOCKEN` contains a dedicated access tocken from Docker Hub that only has read and write permission, in order to push images.

2.3.2 Pushing to Docker Hub

To push the newly built image to Docker Hub, a new job is created in the same workflow file, called `build-and-push-docker-images`. Because we only want to publish code that works at least a bit, it only runs it the previous one (`test-backend`) finishes successfully.

After checking out the code like before, the job logs in to Docker Hub using the secret credentials from Section 2.3.1 and the action `docker/login-action`.

Then, it uses `docker/build-and-push` to build each image (database, backend and frontend). If the current branch is `main`, it additionally pushes the result to the Hub.

2.4 Setup SonarCloud's Quality Gate

SonarCloud is a static analyser. Its role is to scan our app and tell what part of the code can be improved, if the tests are sufficient, or if there are potential security holes. It is an online service that can be called by Maven when the app is built.

Once we register our project, we are provided with a token. Because we want to call SonarCloud from our CI pipeline, we add the token to our action secrets. The `backend-tests` job can now be amended to use the following command for the tests:

```
mvn -B clean verify sonar:sonar \
--file backend/pom.xml \
-Dsonar.token=${{ secrets.SONAR_TOKEN }}
```

Additionally, we need to add some metadata to our `pom.xml` for SonarCloud to find its kitten:

```
<sonar.organization>e10e3</sonar.organization>
<sonar.host.url>https://sonarcloud.io</sonar.host.url>
<sonar.projectKey>e10e3_devops</sonar.projectKey>
```

And *voilà*! SonarCloud will now scan the code when the tests run, and can alert you when it goes south. It will even post comments to you pull requests!

Important note: because GitHub Actions are defined using YAML files, it is really important to use multi-line strings if you want to split your commands on multiple lines. To do this, add a pipe (|) or a "greater than" (>) symbol after the key and start a new line. For instance:

```
---
key: |
  multi-line
  string!
```

2.5 Fixing SonarCloud's alerts

SonarCloud raises two types of security warnings on the default back-end code:

- The first ones are for potential unauthorised cross-origin access, because the annotation `@CrossOrigin` was added to the controllers.
- The second one warns about using database-linked data types as the input from endpoints. This could be exploited to overload the database.

In addition, but not blocking at first, the code coverage was reported to be around 50%, where SonarCloud requires it to be above 80% for new code.

Fixing the first set of warnings was easy: removing the incriminated lines was enough. Since this is a simple case with only one route (coming from the proxy), nothing important goes missing with this deletion.

The second one is more involved. It is necessary to create a *data transfer object* (DTO) that will be the API's input, and will then be transformed into the regular object. For this kind of object that hold no logic, Java 14's records are perfect.

The DTO for the Student class is:

```
public record StudentDto(
        String firstname,
        String lastname,
        Long departmentId) {
}
```

Because a new object is introduced, the function signatures change, conversion needs to be handled, new service routines are added, the tests must be adapted, etc. A little bit everywhere is needed to secure the API.

And then, once the security is taken care of, SonarCloud complains about the tests. "Clean code" fails. Indeed: it detects only 20% of new code coverage. But the locally-run tests indicate 90%!

This is because the test coverage is not updated, so SonarCloud cannot see it. The tool used for this is called JaCoCo. The `jacoco:report` goal needs to be added when the tests are run, and the option `-Pcoverage` should be given to Sonar when called through Maven. After this all, the coverage is updated and the pipeline passes.

Because I was looking at the tests, I used the opportunity to add some missing tests for the original code, bringing the coverage to 97%[4].

2.6 Split pipelines

Our workflow work fine, but we want to refactor it in two separate workflows.

We remove the `build-and-push-docker-image` job from the previous workflow and add put it instead in its own workflow: "Push images".

The other one is renamed to "Run tests" for the occasion, and doesn't change further.

On the "Push images" side, we set it to be triggered by the successful termination of the tests workflow "Run tests", with the added condition that it only runs is the active branch is `main`.

In the end, the behaviour is as so:

If located on main, develop or a pull request ⟶ run the tests
If located on main and previous steps is successful ⟶ push the images

Figure 2: The split workflows' behaviour

2.7 Automatic image version

Pushing the latest version of an image is cool, but isn't it *cooler* for our images to have numbered versions?

What is needed is a system that makes so that the images that are build get automatically assigned a version number when a tag is pushed. Otherwise, they have the *main* and *latest* tag.

This is created by splitting the workflows in three:

- One workflow is a reusable one, it runs the tests on the back-end;
- One executes the preceding workflow (the reusable one) when a commit is pushed on *main* or *develop*,
- The last one build and pushes the Docker images when a commit is made on *main* or a tag is pushed. Before building, it runs the tests through the reusable workflow.

The calling jobs use the line `secrets: inherit` to transmit their action secrets to the reusable workflow.

[4]The coverage with these tools cannot be 100%, because some of the code cannot or has no point in begin tested individually.

Additionally, in order to factor the image push (it is three times the same thing), a matrix strategy is used for the various images we want:

```yaml
strategy:
  matrix:
    include:
      - image: e10e3/ops-database
        dockerfile: ./database
      - image: e10e3/ops-backend
        dockerfile: ./backend
      - image: e10e3/ops-frontend
        dockerfile: ./http
```

The image and Dockerfile are respectively accessed with `${{ matrix.image }}` and `${{ matrix.dockerfile }}`.

The images' tag are set to the correct version using the `docker/metadata-action` action.

And indeed, with the workflows complete, pushing a semantic versioning tag like `v1.1.0` pushes an image with the same tag on Docker Hub.

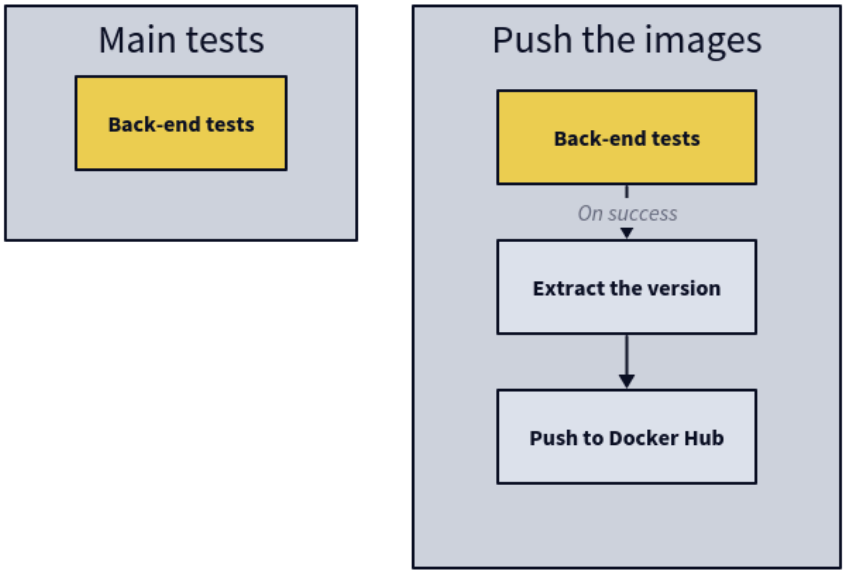

Figure 3: The two main workflows, and a reusable one (in yellow)

3 Ansible

Let's now deploy our app to a real server. I was assigned one, reachable with the domain name emile.royer.takima.cloud. Its only account is called *centos* (after the distribution installed on it, CentOS[5]).

Since this is a distant server, the preferred method to access it is with SSH. A key pair was generated, with the public one installed already on the server. The private one, `id_rsa`, is stored on our side, in the parent directory to our project.

[5] Buried by RedHat, RIP CentOS

Let's connect to our server:

```
ssh -i ../id_rsa centos@emile.royer.takima.cloud
```

Great, it works.

```
[centos@ip-10-0-1-232 ~]$ id
uid=1000(centos) gid=1000(centos) groups=1000(centos),4(adm),190(systemd-
journal) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

To deploy our apps on the server, we could use Docker Compose, as they are already packaged. But this means logging in the server with SSH, transferring files between hosts and running the commands manually.

We'll use *Ansible* to automate the deployment and discover how it could be done if we were operating at a larger scale. We only have one server, so it's a bit overkill, but will this stop us?

Ansible is configured using a tree of files, making it play nicely with POSIX systems. It happens to only run on them anyway.

3.1 Configuring the server with Ansible

In order to use Ansible with our server, we need to register it. Ansible uses its own file `/etc/ansible/hosts`. To add our server, we write its domain name in this file:

```
sudo echo "emile.royer.takima.cloud" > /etc/ansible/hosts
```

It can then be pinged with Ansible:

```
$ ansible all -m ping --private-key=../id_rsa -u centos
emile.royer.takima.cloud | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

Let's install an apache httpd server to how it works well:

```
ansible all -m yum -a "name=httpd state=present" --private-key=../id_rsa -u centos
--become
```

Now create and HTML file to display and start the service:

```
ansible all -m shell -a 'echo "<html><h1>Hello World</h1></html>" >> /var/www/
html/index.html' --private-key=../id_rsa -u centos --become
ansible all -m service -a "name=httpd state=started" --private-key=../id_rsa -u
centos --become
```

Visiting http://emile.royer.takima.cloud diplays "Hello World" as expected.

An inventory is created in order not to have to give the path to the private key at each command:

```
all:
 vars:
   ansible_user: centos
   ansible_ssh_private_key_file: ~/schol/devops/id_rsa
 children:
```

```
  prod:
    hosts: emile.royer.takima.cloud
```

Listing 5: The inventory file at `ansible/inventories/setup.yaml`

Using the inventory works just as well:

```
$ ansible all -i ansible/inventories/setup.yaml -m ping
emile.royer.takima.cloud | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

It is also possible to display information about the target systems:

```
$ ansible all -i ansible/inventories/setup.yaml -m setup -a
"filter=ansible_distribution*"
emile.royer.takima.cloud | SUCCESS => {
    "ansible_facts": {
        "ansible_distribution": "CentOS",
        "ansible_distribution_file_parsed": true,
        "ansible_distribution_file_path": "/etc/redhat-release",
        "ansible_distribution_file_variety": "RedHat",
        "ansible_distribution_major_version": "7",
        "ansible_distribution_release": "Core",
        "ansible_distribution_version": "7.9",
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false
}
```

Question 3-1: *Document your inventory and base commands*

An Ansible inventory is grouped in categories. Here, only one group of hosts is created: the default *all*.

For each group, variables can be created. Here, it is the user and the private key file path.

Groups can have children (subgroups): *prod* is implicitly created as a subgroup of *all*. It has one host: our server.

For now, we only used the base `ansible(1)` command. Its one required argument is <pattern>, indicating which groups to target. Some useful options are:

`-m` The Ansible module to run.

`-a` The arguments to give to the module.

`-i` The inventory to use.

`--private-key` Which private key should be used.

3.2 Creating a playbook

Just like with Docker, it is tedious to run so many commands, especially when there are many packages to set up on the hosts.

But just like with Docker, there is a solution: playbooks.

Let's create a basic one:

`ansible/playbook.yaml`

```
- hosts: all
  gather_facts: false
  become: true

  tasks:
   - name: Test connection
     ping:
```

Execute the playbook on our server:

```
ansible-playbook-i ansible/inventories/setup.yaml ansible/playbook.yaml
```

The ping is successful.

Because our apps are containerised, it'll be needed to install Docker on the server. A playbook to do this looks like:

`docker-playbook.yaml`

```
---
- hosts: all
  gather_facts: false
  become: true

# Install Docker
  tasks:

  - name: Install device-mapper-persistent-data
    yum:
      name: device-mapper-persistent-data
      state: latest

  - name: Install lvm2
    yum:
      name: lvm2
      state: latest

  - name: add repo docker
    command:
      cmd: sudo yum-config-manager --add-repo=https://download.docker.com/linux/
centos/docker-ce.repo

  - name: Install Docker
    yum:
      name: docker-ce
      state: present
```

```
  - name: Make sure Docker is running
    service: name=docker state=started
    tags: docker
```

Ansible has the notion of *roles*, where reusable tasks can be defined. A role can be added to any number of playbooks, and its tasks are then added to them, without having to all the tasks by hand.

`ansible-galaxy` helps creating roles:

```
ansible-galaxy init roles/docker
```

The *docker* role is outfitted with the tasks that were in the playbook to install Docker.

`roles/docker/tasks/main.yml`

```
- name: Install device-mapper-persistent-data
  yum:
    name: device-mapper-persistent-data
    state: latest

- name: Install lvm2
  yum:
    name: lvm2
    state: latest

- name: add repo docker
  command:
    cmd: sudo yum-config-manager --add-repo=https://download.docker.com/linux/
centos/docker-ce.repo

- name: Install Docker
  yum:
    name: docker-ce
    state: present

- name: Make sure Docker is running
  service: name=docker state=started
  tags: docker
```

And the *docker* role is added to the cleaned-up playbook:

```
---
- hosts: all
  gather_facts: false
  become: true

  tasks:
   - name: Test connection
     ping:

  roles:
    - docker
```

Question 3-2: *Document your playbook*

The playbook is now really simple — only a bit more complex that the basic one at the start.

It declares to apply to all the hosts, and to execute the commands as root (it says "become root").

The tasks from its roles are added to the list, meaning all the tasks there were before to install Docker. Finally, as the cherry on the cake, a *ping* is added, just to be sure.

3.3 Deploying our app

Let's now create a number of roles, one for each step need to have our API running online.

The roles we created are `create_network`, `create_volume`, `install_docker`, `launch_app`, `launch_database`, `launch_proxy`, and `setup-env-file`.

Install Docker This is the one called *docker* previously. It ensures Docker is installed in the environment.

Create network This role creates a named network for all our container to be in, so they are isolated and can communicate with each other.

Create volume This role creates a named volume in order to persist our database's entries, even between restarts.

Launch database This one starts the database, with its environment variables, the volume and the network.

Launch app This role starts the back-end with the environment variables necessary to connect to the database and is attached to the network.

Launch proxy This role starts the HTTP proxy server, includes it in the network and maps its port so that the hosts' port 80 redirects to it.

Setup env file This transient role is a dependency of the database and app roles. It is not called directly in our playbook. Its role is to copy our `.env` file to the server (in `/root/ansible-env`), in order for the other riles to access its content.

The task each contains is an equivalent of what the `compose.yaml` file what doing for Docker Compose.

Here is the `task/main.yml` file for *launch_database*:

```
---
- name: Create the database container
  community.docker.docker_container:
    name: database-layer
    image: e10e3/ops-database
    pull: true
    volumes:
      - db_storage:/var/lib/postgresql/data
    networks:
      - name: app_network
    env-files:
      - /root/ansible-env
```

All of the roles are called in the playbook.

playbook.yaml

```yaml
---
- hosts: all
  gather_facts: false
  become: true

  tasks:
    - name: Test connection
      ping:

  roles:
    - install_docker
    - create_network
    - create_volume
    - launch_database
    - launch_app
    - launch_proxy
```

In order to efficiently use Docker container and have a syntax that resembles Compose, a few Ansible modules are used. `community.docker.docker_container` allows to start containers from images and parameter them, `community.docker.docker_network` can create networks and `community.docker.docker_volume` will create storage volumes.

Question 3-3: *Document your `docker_container` tasks configuration.*

A handful of options of `docker_container` are used to configure the containers:

name Gives a name to the container. Useful to make a reference to it in an program's configuration.

image The image to use for the container. It is pulled from Docker Hub by default.

pull Whether to systematically fetch the image, even if it appears not to have changed. Because the images used are all with the *latest* tag, this option is useful to force Ansible to use the latest version of the *latest* tag.

networks A list of networks to connect the container to.

env A mapping of environment variables to create in the container. Useful for configuration.

volumes A list for volumes to map the container's directories to. Only the database has a use for this option in this case.

ports Port binding for the container. In this case, the proxy's port 80 is mapped to the server's port 80.

3.4 Ansible Vault

Environment variables to are listed in the roles' tasks, but some variable's value should stay a secret, whereas roles should be published. A secure way to handle secrets is needed.

Ansible itself supports variables, that are distinct from environment variables. These variables can be declared in the playbook, tasks files, and also in external files.

They allow to parameterise the configuration: a value is no longer hard-written, but references a variable declared in YAML.

The project's Ansible configuration is put in `ansible/env.yaml`.

```yaml
---
database:
  container_name: "database-layer"
  postgres_name: "db"
  postgres_username: "usr"
  postgres_password: "pwd"
backend:
  container_name: "backend-layer"
```

Ansible has another mechanism to protect secret: the vault.

The Ansible Vault encrypts file and variables in place, and decrypts them when e.g. a playbook is used. The files can only be decrypted if the correct pass is given (the same one that was used to encrypt it). Under the hood, it uses AES-256 to secure the data.

In this situation, the `en.yaml` file was encrypted with `ansible-vault encrypt env.yaml`, and a vault pass.

Encrypted files can still be referenced in playbooks:

```yaml
---
- hosts: all
  gather_facts: false
  become: true
  vars_files:
    - env.yaml

  tasks:
    - name: Test connection
      ping:

  roles:
    - install_docker
    - create_network
    - create_volume
    - launch_database
    - launch_app
    - launch_proxy
```

... and the decrypted value can be transferred to role (here with the database):

```yaml
---
- name: Create the database container
  community.docker.docker_container:
    name: "{{ database.container_name }}"
    image: e10e3/ops-database
    volumes:
      - db_storage:/var/lib/postgresql/data
    networks:
      - name: app_network
    env:
      POSTGRES_DB: "{{ database.postgres_name }}"
```

```
    POSTGRES_USER: "{{ database.postgres_username }}"
    POSTGRES_PASSWORD: "{{ database.postgres_password }}"
```

A playbook that uses an encrypted file can be launched with:

```
ansible-playbook -i inventories/setup.yaml playbook.yaml --ask-vault-pass
```

to ask for the secret pass.

4 A new front-end

As a further step, a front-end is introduced in the project.

In our case, since the reverse proxy was called “front-end”, this means we need to rename it everywhere. Here goes v2.

The front-end is found here: https://github.com/takima-training/devops-front. It is a Vue app, meaning a Node.js image is the base for the Dockerfile. But once Vue compiled the code, JavaScript is no longer needed on the server side, and the packaged `*.js` files can be distributed by a regular web server.

The reverse proxy now has a use: it will distribute the traffic between the back- and the front-end depending on the requested URL.

The proxy server needs to be reconfigured to handle the different redirections.

```
<VirtualHost *:80>
    ProxyPreserveHost On
    ProxyPass "/api/" "http://${BACKEND_CONTAINER_NAME}:8080/"
    ProxyPassReverse "/api/" "http://${BACKEND_CONTAINER_NAME}:8080/"
    ProxyPass "/" "http://${FRONTEND_CONTAINER_NAME}:80/"
    ProxyPassReverse "/" "http://${FRONTEND_CONTAINER_NAME}:80/"
</VirtualHost>
```

Listing 6: How the reverse proxy is configured in the HTTP server

The redirection is made as so:

- the traffic looking for `/api/*` will go to the back-end,
- the traffic looking for all other paths will go to the front-end.

The following architecture diagram shows the relation between the servers:

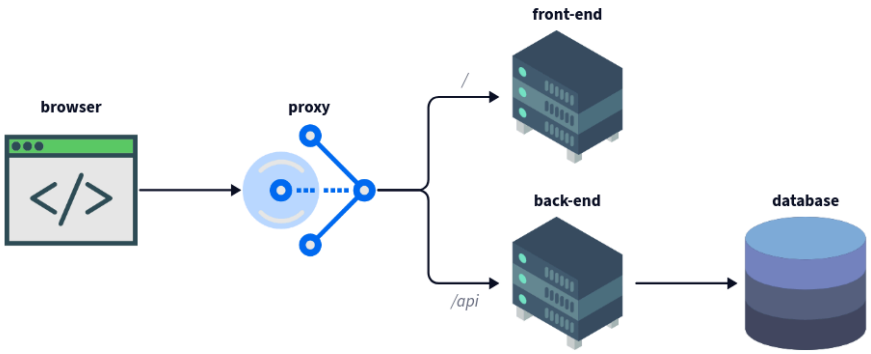

Figure 4: The app’s architecture

Since the front-end makes itself use of the API, it needs to make calls to `/api/`, routing though the reverse proxy.

In order to deploy the front-end with the rest of the app, a service was added to the Compose file, and a corresponding new role was created in Ansible. They both use the same principles as were seen before.

4.1 More parametrisation

Because the proxy needs to access the back-end ans front-end servers' container name, the occasion was used to further the parametrisation of the configuration. Now, in addition of the database name, user and password, the machine's hostname and the containers' name can be indicated directly in the configuration[6].

4.2 Changing environment variables at build time

Vue is configured at build time. It can be parametered with variables, but it only uses variables from a defined set of files, depending on what part of the lifecycle is wanted.

Thankfully these variables can be overridden with environment variables from the host. But remember: it needs to happen at build time.

In particular, it is needed to indicate what the base URL to access the API is. Without it, the front-end cannot function. We know it is `/api/`, but for *which* host name? (Local use and server use require different host names.)

To set this up, an environment variable is declared directly in the Dockerfile; but its value is not static:

```
ARG API_URL
ENV VUE_APP_API_URL ${API_URL}
```

This allows to change the value of `VUE_APP_API_URL` with the `-build-arg` option. Compose supports this option, so the local use-case is covered. For the general use-case, it is needed to go through GitHub Actions: this is where the images are built. A workflow variable was created, and the build argument uses this variable to change the API URL.

```
build-args: |
  API_URL=${{ vars.TARGET_HOSTNAME }}/api
```

5 Conclusion

A lot of things would have been faster if I had hard-written the configuration, and not had to use environment variables to set it. But at the end, I have the satisfaction of a well-tended project.

All in all we did created Docker containers, created a CI/CD workflow for them, fixed linting errors, and deployed the project on a server with Ansible. Success!

[6] The hostname is not decided by our app, so it cannot be changed on our side (the value is still used), but all other variables are commands on what to set.