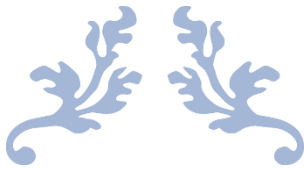


DEVOPS

TP

Table des matières

TP1 : Docker

Questions

Answers

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 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

2- Write dockerfile.

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

3- Now, to launch app you have to do the same thing that Basic step 1.

1. docker build -t my-java-app
2. 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.

```
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.

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
```

2. Put Database and simple-api in the same network
3. 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.

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

Detailed description of the specified object

- docker logs

standard output and standard error 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.

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 and have a better security accessing only the reverse proxy and not directly a backend app.

Link application

Docker-compose

1- Install docker-compose 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 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-network
    depends_on:
    - database

  database:
    container_name: tp1database
    build: ./Database
    networks:
    - my-network

  httpd:
    build: ./HTTPServer
    ports:
    - "8082:80"
    networks:
      - my-network
    depends_on:
    - backend

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 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.

```yaml
version: '3.7'

services:
  backend:
    container_name: simpleapi
    build: ./simple-api-student-main
    networks:
    - my-network
    depends_on:
    - database

  database:
    container_name: tp1database
    build: ./Database
    networks:
    - my-network

  httpd:
    build: ./HTTPServer
    ports:
    - "8082:80"
    networks:
      - my-network
    depends_on:
    - backend

networks:
  my-network: {}
```

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 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.

TP2 : Github Actions

Goals

Good Practice

Do not forget to document what you do along the steps.

Create an appropriate file structure, 1 folder per image.

Target Application

Complete pipeline workflow for testing and delivering your software application.

We are going to use different useful tools to build your application, test it automatically, and check the code quality at the same time.

Link

GitHub Actions

Setup GitHub Actions

The first tool we are going to use is **GitHub Actions**. **GitHub Actions** is an online service that allows you to build pipelines to test your application. Keep in mind that **GitHub Actions** is not the only one on the market to build integration pipelines.

Historically many companies were using Jenkins (and still a lot continue to do it), it is way less accessible than **GitHub Actions** but much more configurable. You will also hear about Gitlab CI and Bitbucket Pipelines during your work life.

First steps into the CI World

Note

Push your previous project on your personal GitHub repository.

Most of the CI services use a yaml file (except Jenkins that uses a… Groovy file…) to describe the expected steps to be done over the pipeline execution. Go on and create your first main.yml file into your project's root directory.

Build and test your Application

For those who are not familiar with Maven and Java project structures, here is the command for building and running your tests:

```
mvn clean verify
```

You need to launch this command from your `pom.xml` directory, or specify the path to it with `--file /path/to/pom.xml` argument.

Note

What is it supposed to do?

This command will actually clear your previous builds inside your cache (otherwise your can have unexpected behavior because maven did not build again each part of your application), then it will freshly build each module inside your application, and finally it will run both Unit Tests and Integration Tests (sometime called Component Tests as well).

Unit tests ? Component tests ?

Integration tests require a database to verify you correctly inserted or retrieved data from it. Fortunately for you, we've already taken care of this! But you still need to understand how it works under the hood. Take a look at your application file tree.

Let's take a look at the `pom.xml` that is inside the **simple-api**, you will find some very helpful dependencies for your testing.

```
<dependencies>
  <dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>jdbc</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

As you can see, there are a bunch of testcontainers dependencies inside the pom.

Question

2-1 What are testcontainers?

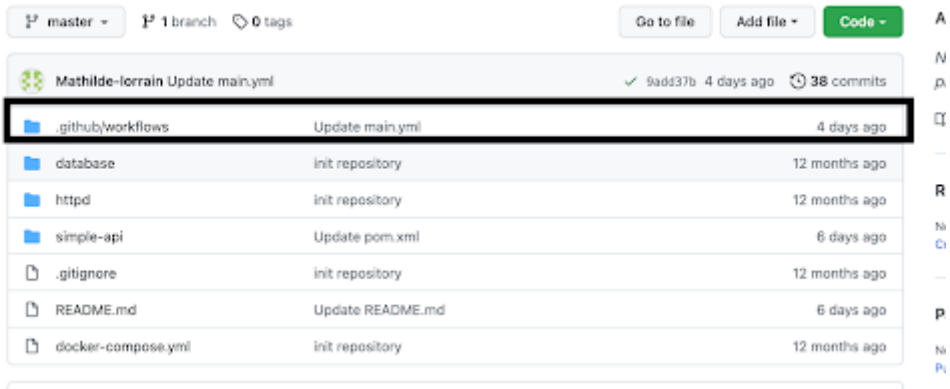
They simply are java libraries that allow you to run a bunch of docker containers while testing. Here we use the postgresql container to attach to our application while testing. If you run the command `mvn clean verify` you'll be able to see the following:

As you can see, a docker container has been launched while your tests were running, pretty convenient, isn't it?

Finally, you'll see your test results.

Now, it is up to you! Create your first CI, asking to build and test your application every time someone commits and pushes code on the repository.

First you create a `.github/workflows` directory in your repository on GitHub.



Put your `main.yml` inside workflows.

The `main.yml` holds the architecture of your pipeline. Each job will represent a step of what you want to do. Each job will be run in parallel unless a link is specified.

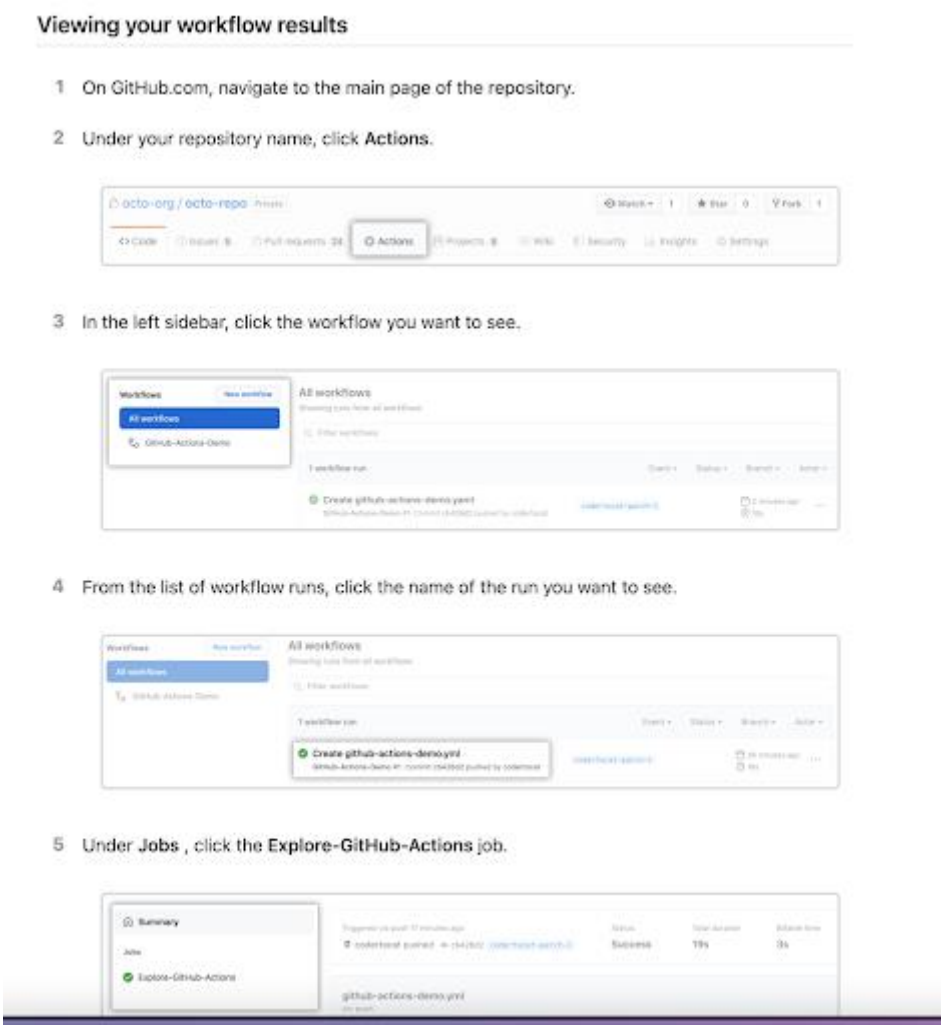
Here is what your `main.yml` should look like:

```
name: CI devops 2023
on:
```

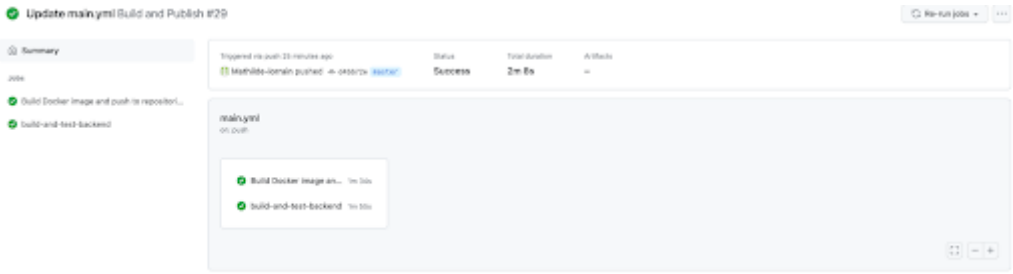
```
  #to begin you want to launch this job in main and develop
  push:
    branches: #TODO
  pull_request:

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

     #do the same with another action (actions/setup-java@v3) that enable to setup jdk 17
      - name: Set up JDK 17
        #TODO

     #finally build your app with the latest command
      - name: Build and test with Maven
        run: #TODO
```

It’s your turn, fill the #TODOs!

To see the result you must follow the next steps:



And if it’s GREEN you win!



Question

2-2 Document your **Github Actions** configurations. **+** Document your **quality gate** configuration.

on:

push:

branches:

- main

- develop

pull_request:

This section defines when the workflow should be triggered:

The workflow is triggered on "push" events to the branches "main" and "develop."

It's also triggered on "pull_request" events.

jobs:

test-backend:

runs-on: ubuntu-22.04

steps:

This section defines the first job named "test-backend" that runs on an Ubuntu 22.04 virtual machine.

- name: Checkout code

uses: actions/checkout@v3

This step checks out the code from your GitHub repository using the "actions/checkout@v3" action.

```
- name: Set up JDK 17
  uses: actions/setup-java@v3
  with:
    java-version: 17
    distribution: 'temurin'
```

This step sets up Java Development Kit (JDK) version 17 using the "actions/setup-java@v3" action.

```
- name: Build and test with Maven
  run: mvn -B clean verify sonar:sonar -Dsonar.projectKey=DehMatthieu_Devops -Dsonar.organization=dehmatthieu -Dsonar.host.url=https://sonarcloud.io -Dsonar.login=${{ secrets.SONAR_TOKEN }}  --file ./simple-api-student-main/pom.xml
```

This step builds and tests your application using Maven. It also includes SonarCloud integration for static code analysis.

```
build-and-push-docker-image:
  needs: test-backend
  runs-on: ubuntu-22.04
  steps:
```

This section defines the second job named "build-and-push-docker-image," which depends on the completion of the "test-backend" job.

```
- name: Checkout code
  uses: actions/checkout@v3
```

This step checks out the code again.

```
- name: Login to DockerHub
  run: docker login -u ${{ secrets.USER }} -p ${{ secrets.PWD }}
```

This step logs in to DockerHub using the provided Docker credentials stored in GitHub secrets.

```
- name: Build image and push backend
  uses: docker/build-push-action@v3
  with:
    context: ./simple-api-student-main
```

```
      tags:  ${{secrets.USER}}/tp1-backend
      push: ${{ github.ref == 'refs/heads/main' }}
```

This step builds a Docker image for the backend from the source code located in the "./simple-api-student-main" directory and pushes it to DockerHub. It's configured to push the image only when changes are pushed to the "main" branch.

```
    - name: Build image and push database
      uses: docker/build-push-action@v3
      with:
        context: ./Database
        tags:  ${{secrets.USER}}/tp1-database
        push: ${{ github.ref == 'refs/heads/main' }}
```

This step builds and pushes a Docker image for the database from the source code located in the "./Database" directory, and it's also configured to push the image only when changes are pushed to the "main" branch.

```
    - name: Build image and push httpd
      uses: docker/build-push-action@v3
      with:
        context: ./HTTPServer
        tags:  ${{secrets.USER}}/tp1-httpd
        push: ${{ github.ref == 'refs/heads/main' }}
```

This step builds and pushes a Docker image for the HTTP server from the source code located in the "./HTTPServer" directory, and it's also configured to push the image only when changes are pushed to the "main" branch.

This YAML code sets up a GitHub Actions workflow to automate the build, test, and deployment processes for your application, including Docker image creation and push to DockerHub. It's triggered on specific branch events and pull requests. Secrets are used to securely store sensitive information like DockerHub credentials.

First steps into the CD World

Here we are going to configure the Continuous Delivery of our project. Therefore, the main goal will be to create and save a docker image containing our application on the Docker Hub every time there is a commit on a main branch.

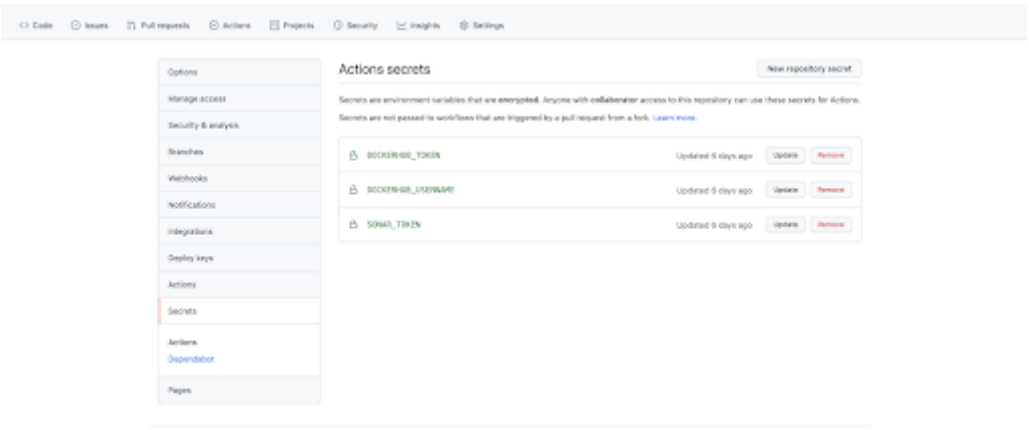
As you probably already noticed, you need to log in to docker hub to perform any publication. However, you don't want to publish your credentials on a public repository (it is not even a good practise to do it on a private repository). Fortunately, **GitHub** allows you to create secured environment variables.

1- Add your docker hub credentials to the environment variables in **GitHub Actions** (and let them secured).

Note

Secured Variables, why?

Now that you added them, you can freely declare them and use them inside your **GitHub Actions** pipeline.



2- Build your docker images inside your **GitHub Actions** pipeline.

Maybe the template Build a docker image can help you!

```
# define job to build and publish docker image
build-and-push-docker-image:
 needs: test-backend
 # run only when code is compiling and tests are passing
 runs-on: ubuntu-22.04

 # steps to perform in job
 steps:
   - name: Checkout code
     uses: actions/checkout@v2.5.0

   - name: Build image and push backend
     uses: docker/build-push-action@v3
     with:
       # relative path to the place where source code with Dockerfile is located
       context: ./simple-api
       # Note: tags has to be all lower-case
       tags:  ${{secrets.DOCKERHUB_USERNAME}}/tp-devops/simple-api

   - name: Build image and push database
       # DO the same for database

   - name: Build image and push httpd
     # DO the same for httpd
```

Note

Why did we put **needs: build-and-test-backend** on this job? Maybe try without this and you will see!

OK your images are built but not yet published on dockerhub.

3- Publish your docker images when there is a commit on the main branch.

Don't forget to do a docker login and to put your credentials on secrets!

```
- name: Login to DockerHub
  run: docker login -u ${{ secrets.DOCKERHUB_USERNAME }} -p ${{ secrets.DOCKERHUB_TOKEN }}
```

And after modify job `Build image and push backend` to add a `push` action:

```
- name: Build image and push backend
  uses: docker/build-push-action@v3
  with:
    # relative path to the place where source code with Dockerfile is located
    context: ./simple-api
    # Note: tags has to be all lower-case
    tags:  ${{secrets.DOCKERHUB_USERNAME}}/tp-devops:simple-api
    # build on feature branches, push only on main branch
    push: ${{ github.ref == 'refs/heads/main' }}
```

Do the same for other containers.

Note

For what purpose do we need to push docker images?

To be able to run the tests

Now you should be able to find your docker images on your docker repository.

Check ✓

Working CI & Docker images pushed to your repository.

Setup Quality Gate

What is quality about?

Quality is here to make sure your code will be maintainable and determine every unsecured block. It helps you produce better and tested features, and it will also prevent having dirty code pushed inside your main branch.

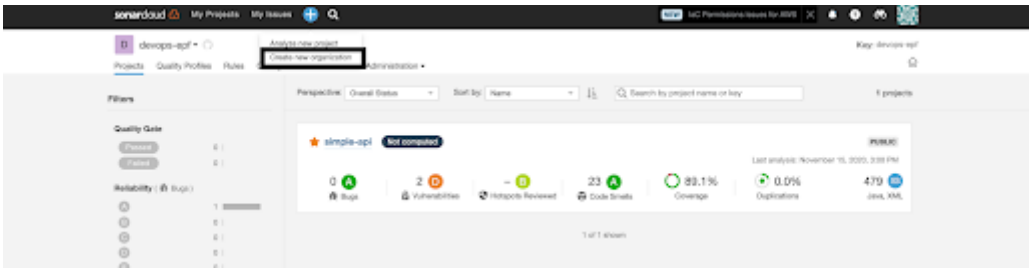
For this purpose, we are going to use **SonarCloud**, a cloud solution that makes analysis and reports of your code. This is a useful tool that everyone should use in order to learn java best practices.

Register to SonarCloud

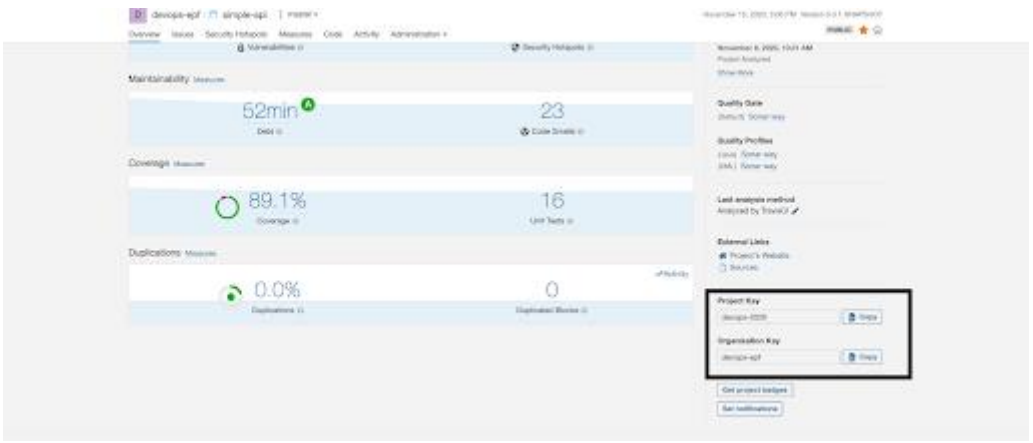
Create your free-tier account on SonarCloud.

SonarCloud will propose you to set up your **GitHub Actions** pipeline from the **GitHub Actions**, but forget about that, there is a much better way to save the **SonarCloud** provided and provide it into your `main.yml`.

1- You must create an organization.



2- And keep the project key and the organization key you will need it later.



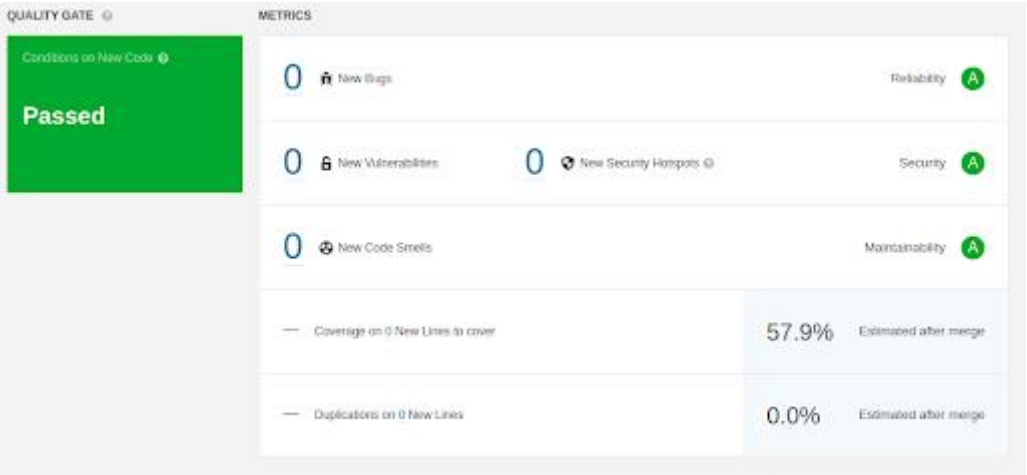
3- You need to add this script to your `main.yml` for launch sonar at each commit.

Set up your pipeline to use **SonarCloud** analysis while testing.

For that, you need to modify your first step named **Build and test** with Maven and change sonar organization and project key.

```
mvn -B verify sonar:sonar -Dsonar.projectKey=devops-2023 -Dsonar.organization=devops-school -
Dsonar.host.url=https://sonarcloud.io -Dsonar.login=${{ secrets.SONAR_TOKEN }}  --file ./simple-api/pom.xml
```

If you did your configuration correctly, you should be able to see the **SonarCloud analysis** report online:



Check ✓

Working quality gate.

Question

2-2 Document your **Github Actions** configurations. **+** Document your **quality gate** configuration.

Well done buddies, you've created your very first Quality Gate! Yay!

Bonus: split pipelines (Optional)

In this step you have to separate your jobs into different workflows so that they respect 2 things:

- **test-backend** must be launched on develop and master branch and **build-and-push-docker-image** on master only.
- The job that pushes the docker api image must be launched only if **test-backend** is passed.

Tip

You can use **on: workflow_run** to trigger a workflow when another workflow is passed.

Check ✓

Goals

Install and deploy your application automatically with ansible.

TP3 : Ansible

Introduction

Inventories

By default, Ansible's inventory is saved in the location `/etc/ansible/hosts` where you already defined your server.

The headings between brackets (eg: [webservers]) are used to group sets of hosts together, they are called, surprisingly, groups. You could regroup them by roles like database servers, front-ends, reverse proxies, build servers…

Let’s create a project specific inventory, in your project create an ansible directory, then create a new directory called inventories and in this folder a new file (`my-project/ansible/inventories/setup.yml`):

```
all:
 vars:
  ansible_user: centos
  ansible_ssh_private_key_file: /path/to/private/key
 children:
  prod:
   hosts: hostname or IP
```

Test your inventory with the ping command:

```
ansible all -i inventories/setup.yml -m ping
```

Facts

Let’s get information about hosts: these kinds of variables, not set by the user but discovered are called **facts**.

Facts are prefixed by ansible_ and represent information derived from speaking with your remote systems.

You will request your server to get your OS distribution, thanks to the setup module.

```
ansible all -i inventories/setup.yml -m setup -a "filter=ansible_distribution*"
```

Earlier you installed Apache httpd server on your machine, let’s remove it:

```
ansible all -i inventories/setup.yml -m yum -a "name=httpd state=absent" --become
```

With ansible, you just describe the state of your server and let ansible automatically update it for you.

If you run this command another time you won't have the same output as httpd would have been removed.

Question

3-1 Document your inventory and base commands

```yaml
all:
  vars:
    ansible_user: centos
    ansible_ssh_private_key_file: /home/mdehaynin/id_rsa
  children:
    prod:
      hosts: matthieu.dehaynin.takima.cloud
```

Playbooks

First playbook

Let's create a first very simple playbook in `my-project/ansible/playbook.yml`:

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

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

Just execute your playbook:

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

You can check your playbooks before playing them using the option: `--syntax-check`

Advanced Playbook

Let's create a playbook to install docker on your server, follow the documentation and create the corresponding tasks: https://docs.docker.com/install/linux/docker-ce/centos/.

```
- 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: Install python3
    yum:
      name: python3
      state: present

  - name: Install docker with Python 3
    pip:
      name: docker
      executable: pip3
    vars:
      ansible_python_interpreter: /usr/bin/python3

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

Good news, we now have docker installed on our server. One task was created to be sure docker was running, you could check this with an ad-hoc command or by connecting to the server until you really trust ansible.

Using roles

Our docker install playbook is nice and all but it will be cleaner to have in a specific place, in a role for example. Create a docker role and move the installation task there:

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

Call the docker role from your playbook to check your refactor and your installation.

Initialized role has a couple of directories, keep only the one you will need:

- tasks - contains the main list of tasks to be executed by the role.

- handlers - contains handlers, which may be used by this role or outside.

Question

3-2 Document your playbook

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

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

Deploy your App

Time has come to deploy your application to your Ansible managed server.

Create specific roles for each part of your application and use the Ansible module: docker_container to start your dockerized application. Here is what a docker_container task should look like:

```
- name: Run HTTPD
  docker_container:
    name: httpd
    image: jdoe/my-httpd:1.0
```

You must have at least this roles :

- install docker
- create network
- launch database
- launch app
- launch proxy

Note

- You will need to add env variables on app and database tasks. Ansible is able to modify the variables either in the .env for the db or in the application.yml for the app.
- Don’t forget to use existing module for example to create the network
- Don't forget to use the right python interpreter when creating the docker network (refer to `ansible_python_interpreter` variable usage)

Link

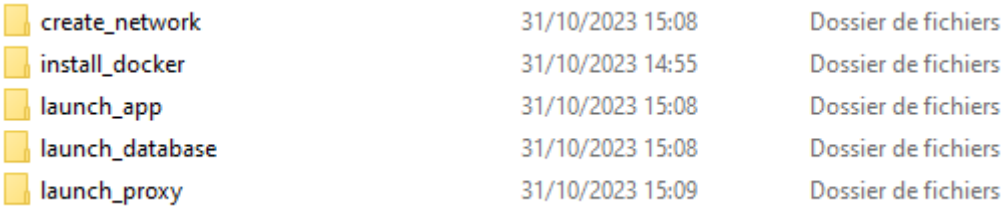
- docker_container module documentation
- docker_network module documentation

Check

You should be able to access your API on your server.

Question

Document your docker_container tasks configuration.




```
# tasks file for roles/create_network


- name: Create Docker Network
  docker_network:
    name: my-network
    state: present
```

```yaml
# tasks file for roles/docker

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

- name: Install lvm2
  yum:
    name: lvm2
    state: present

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

- name: Install python3
  yum:
    name: python3
    state: present

- name: Install docker with Python 3
  pip:
    name: docker
    executable: pip3
  vars:
    ansible_python_interpreter: /usr/bin/python3

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

```yaml
# tasks file for roles/launch_app
- name: Launch App Container
  docker_container:
    name: simpleapi
    image: mdehaynin/tp1-backend
    networks:
      - name: my-network
    state: started
```

```yaml
# tasks file for roles/launch_database
- name: Launch Database Container
  docker_container:
    name: tp1database
    image: mdehaynin/tp1-database
    networks:
      - name: my-network
    state: started
```

```yaml
# tasks file for roles/launch_proxy
- name: Launch Proxy Container
  docker_container:
    name: tp1-httpd
    image: mdehaynin/tp1-httpd
    ports:
      - "8082:80"
    networks:
      - name: my-network
```

Front

If you have reached the end of each TP, you are able to access your api through your server.

Your database, api and httpd must be up on your server and deployed with your Github action.

Everything under the hood of docker-compose.

Usually when we have an API we also have something called a front part to display our information.

That's your bonus part to do, you can find the code of the front ready.

You have to customize your httpd server to make the redirection correct between the API and the front. The httpd server is a proxy within your system.

Check

Front working

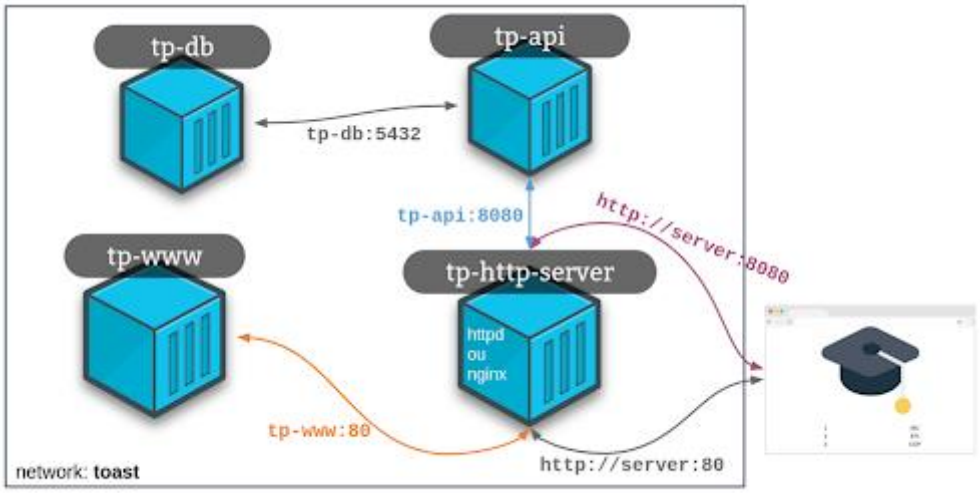
Continuous Deployment

Note

Do this part in a separate workflow.

Configure Github action to automatically deploy your application when you release it on the production branch of your github repository.

- It is a little bit overkilled to launch an Ansible job for deploying on one unique server. Therefore you ssh to your machine with your encrypted private key and only relaunch your http api backend application.
- You like challenges and overkilled solutions, you run your Ansible script through a Docker image (that provides Ansible, of course) and you use a VAULT to encrypt your private data.



Check

Full CI/CD pipeline in action.