

TP part 02 - CI/CD



Checkpoint: call us to check your results
(don't stay blocked on a checkpoint if we are busy, we can check % checkpoints at the same time)



Point to document/report



Interesting information

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 gonna go through different useful tools to build your application, test it automatically, and check the code quality at the same time.

Useful links :

- <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>

Setup Github Actions

The first tool we are going to use is Github Action. Github Action 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 Pipeline](#) during your work life.

First steps into the CI world

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.



Ok, what is it supposed to do ?

This command will actually clear your previous builds inside your cache (otherwise you 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 test ?

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 to the `pom.xml` that is inside the `simple-api`, you will find some very helpful dependencies for your testing.

```

<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>testcontainers</artifactId>
  <version>1.13.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>jdbc</artifactId>
  <version>1.13.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>postgresql</artifactId>
  <version>1.13.0</version>
  <scope>test</scope>
</dependency>

```

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



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 run the command “mvn clean verify” you’ll be able to see the following:

```

2020-01-27 14:22:48.473 INFO 24545 --- [main] org.testcontainers.DockerClientFactory : Docker host IP address is localhost
2020-01-27 14:22:48.634 INFO 24545 --- [main] org.testcontainers.DockerClientFactory : Connected to docker:
  Server Version: 19.03.5
  API Version: 1.40
  Operating System: Ubuntu 18.04.3 LTS
  Total Memory: 15894 MB
2020-01-27 14:22:49.506 INFO 24545 --- [main] org.testcontainers.DockerClientFactory : Ryuk started - will monitor and terminate Testcontainers containers on JVM exit
  I Checking the system...
  ✓ Docker version should be at least 1.6.0
  ✓ Docker environment should have more than 2GB free disk space
2020-01-27 14:22:50.085 INFO 24545 --- [main] [postgres:12.0] : Creating container for image: postgres:12.0
2020-01-27 14:22:50.116 INFO 24545 --- [main] [postgres:12.0] : Starting container with ID: f43bc8dfdf4f0483489d19abe6032cb1069a6d27d72d6329749dedf701c7fc1f
2020-01-27 14:22:50.494 INFO 24545 --- [main] [postgres:12.0] : Container postgres:12.0 is starting: f43bc8dfdf4f0483489d19abe6032cb1069a6d27d72d6329749dedf701c7fc1f
2020-01-27 14:22:52.041 INFO 24545 --- [main] [postgres:12.0] : Container postgres:12.0 started

```

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

master	1 branch	0 tags	Go to file	Add file	Code
Mathilde-lorrain Update main.yml			9add37b 4 days ago 38 commits		
.github/workflows	Update main.yml	4 days ago			
database	init repository	12 months ago			
httpd	init repository	12 months ago			
simple-api	Update pom.xml	6 days ago			
.gitignore	init repository	12 months ago			
README.md	Update README.md	6 days ago			
docker-compose.yml	init repository	12 months ago			

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 2022 CPE
on:
  #to begin you want to launch this job in main and develop
  push:
    branches: #TODO
  pull_request:
jobs:
  test-backend:
    runs-on: ubuntu-18.04
    steps:
      #checkout your github code using actions/checkout@v2.3.3
      - uses: actions/checkout@v2.3.3

      #do the same with another action (actions/setup-java@v2) that
      enable to setup jdk 11
      - name: Set up JDK 11
        #TODO

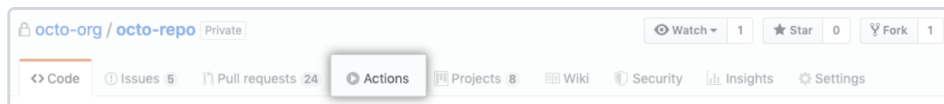
      #finally build your app with the latest command
      - name: Build and test with Maven
        run: #TODO
```

It's your turn, fill the TODOs !

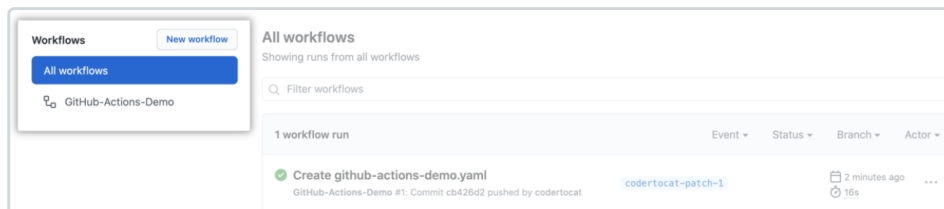
To see the result you must follow the next steps:

Viewing your workflow results

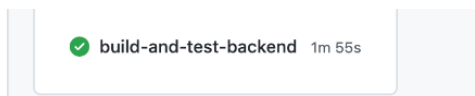
- 1 On GitHub.com, navigate to the main page of the repository.
- 2 Under your repository name, click **Actions**.



- 3 In the left sidebar, click the workflow you want to see.



And if it's GREEN you win !



2-2 Document your Github Actions configurations

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.

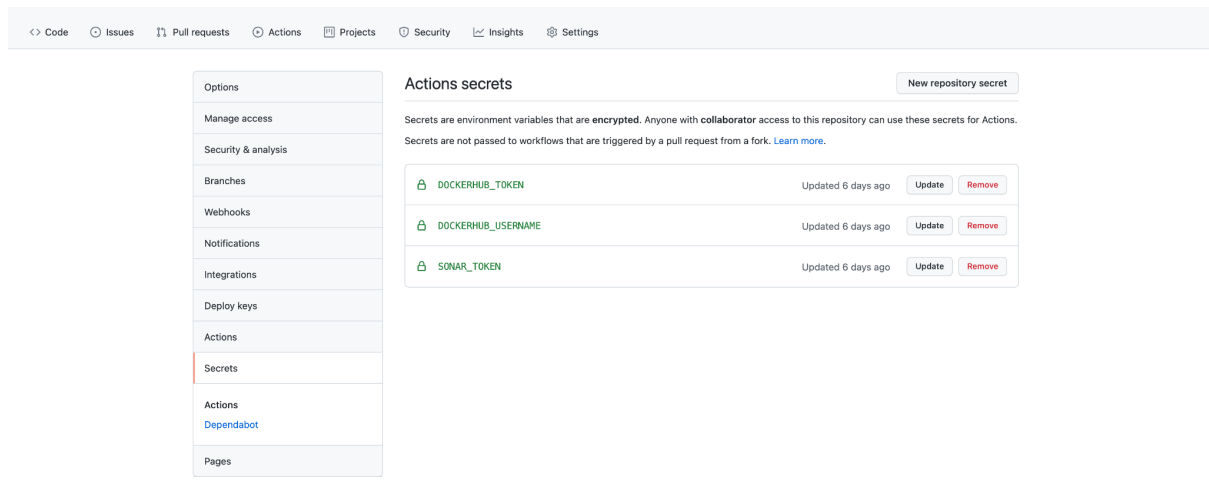
As you probably already noticed, you need to login to docker hub to perform any publication. However, you don't want to publish your credentials on a public repository (it is not even a good practise to do it on a private repository). Fortunately, Github allows you to create secured environment variables.

1 - Add your docker hub credentials to the environment variables in Github Action (and let them secured).



Secured variables, why ?

Now that you added them, you can freely declare them and use them inside your Github Action pipeline.



2 - Build your docker images inside your Github Action 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-latest

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

    - name: Build image and push backend
      uses: docker/build-push-action@v2
      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-cpe:simple-api

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

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

 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@v2
  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-epf:simple-api
    # build on feature branches, push only on main branch
    push: ${ github.ref == 'refs/heads/main' }
```

Do the same for other containers.

 For what purpose do we need to push docker images?

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

 Checkpoint : 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 insecure blocks. It helps you producing 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 practises.

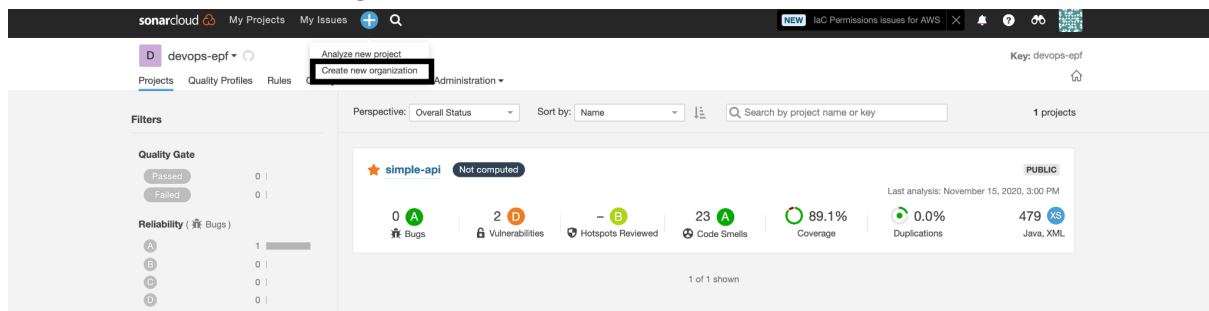
Register to SonarCloud

Create your free-tier account on <https://sonarcloud.io/>.

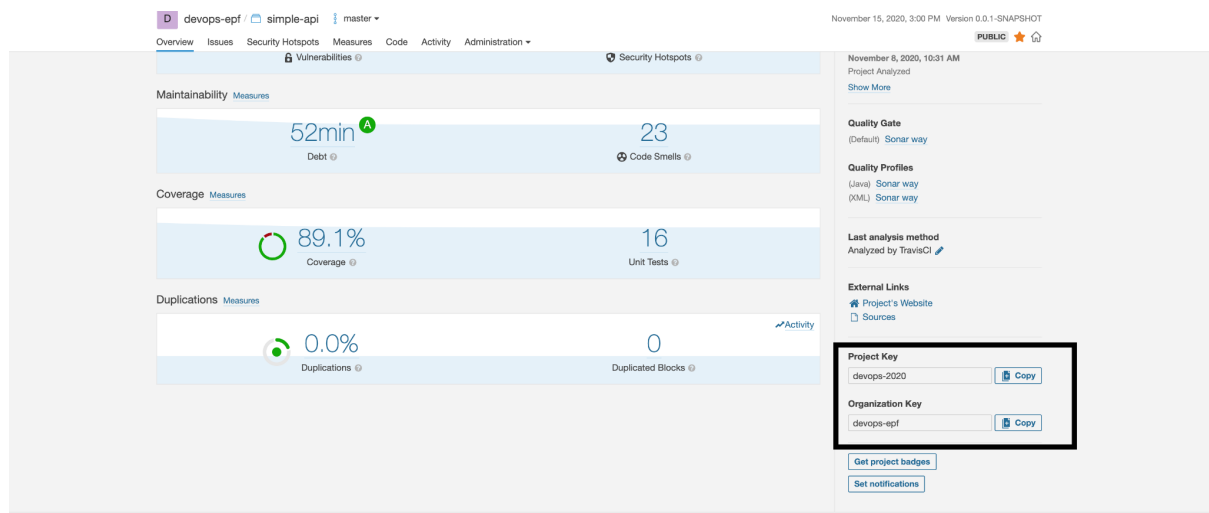
You'll see that you can directly link your GitHub account to your SonarCloud project. Go on and select the repository you want to analyse. Enter your collaboration information etc..

SonarCloud will propose you to setup your Github Action pipeline from the Github Action, 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



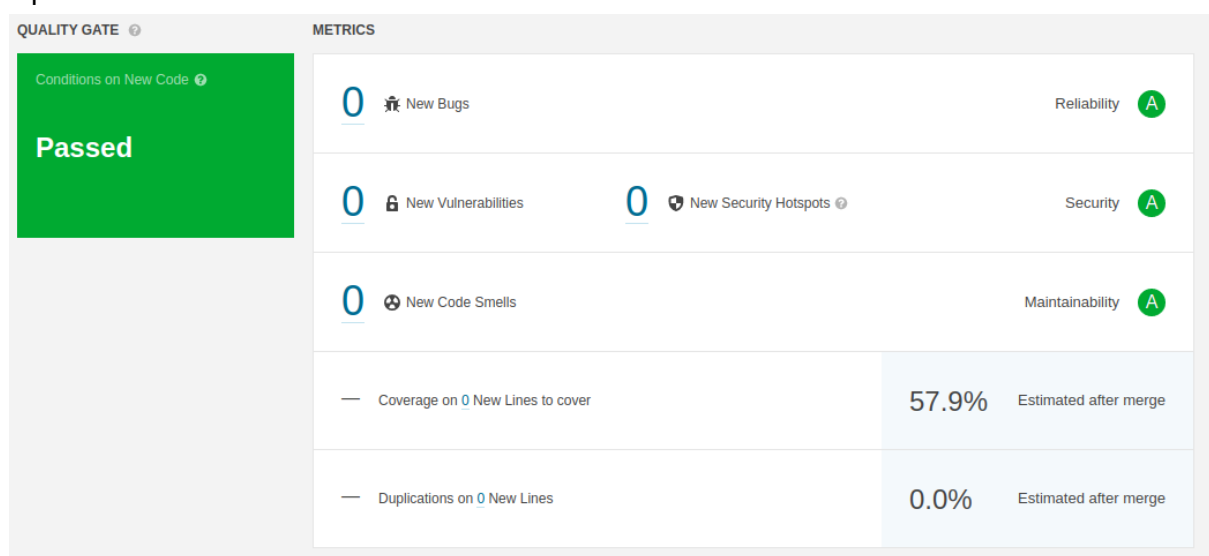
3 - You need to add this script to your main.yml for launch sonar at each commit

Setup 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-2022
-Dsonar.organization=devops-cpe
-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 :





Checkpoint : Working quality gate.



2-3 Document your quality gate configuration

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

Going further: Split pipeline (Optional)

1 - If you look closely you can see that the jobs are launched each time there is a change on main. To optimize your pipeline, you must just launch build-and-test-app job when there are changes on api directory.

```
jobs:
  test-backend:
    runs-on: ubuntu-latest
    env:
      working-directory: ./simple-api
```

2 - For the second job **build-and-push-docker-image** it's more complicated because we push the 3 images in the same job, the best would be to separate in 3 jobs and to push the images only if we have changes on the respective directory.

3 -

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

- **test-backend** must be launch 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

Help : you can use on : workflow_call to trigger a workflow when another workflow is passed