DevOps & Continouous Deployment - Project Report

Authors:

- Ayyoub ZEBDA
- Sofian YAHYAOUI
- Karam MANSOUR

GitHub link: https://github.com/ayyoub-zbd/devops-project-2024

Objectives:

1. Set up an entire automated CI-CD pipeline in order to deploy an application into different environments (DEV and PROD) to enhance development speed and reliability
2. Set up a complete monitoring stack to monitor your application and its infrastructure

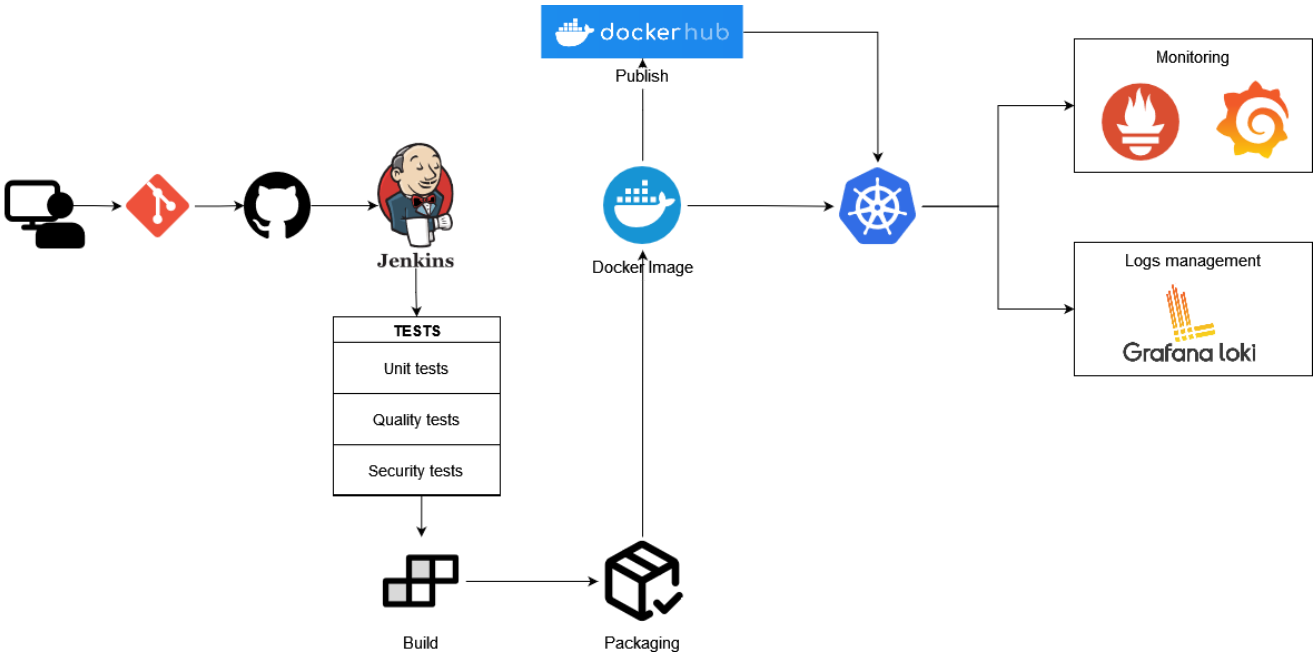
---

I. Build and deploy an application using Docker, Kubernetes and Jenkins pipeline

1. Architecture Diagram

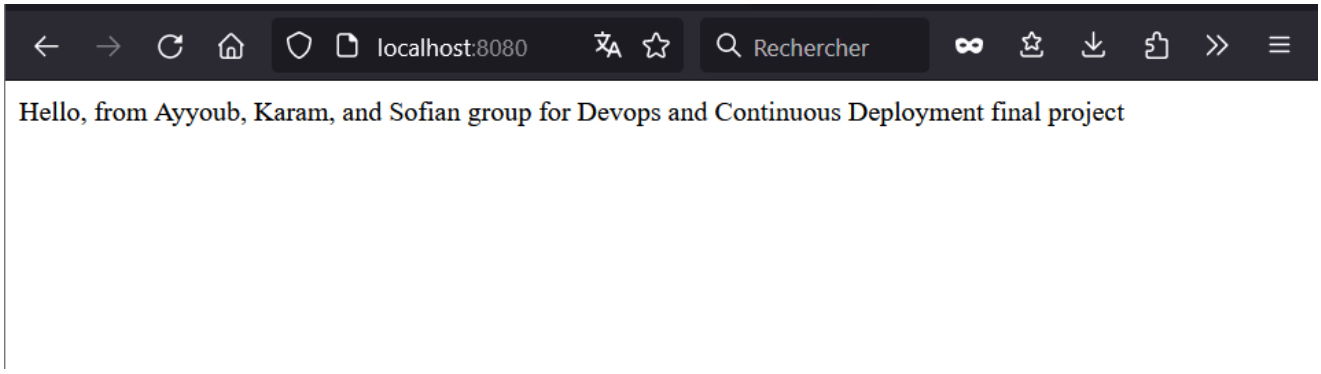
We designed the architecture taking into account all the steps required to deploy the application. Our pipeline includes the following tools, in order:

- **Git** and **GitHub**, available to developers to manage versioning and source code control.
- **Jenkins**, to automate the application build process and the various associated tests.
- **Docker** and **DockerHub**, used to encapsulate the application and its dependencies in containers, providing an isolated, portable environment.
- **Kubernetes**, which will enable us to orchestrate and manage Docker containers put into production.
- **Prometheus** and **Grafana**, tools we want to use to monitor application performance and infrastructure.
- **Grafana Loki**, which will enable us to easily manage the logs produced by our application.



2. App customization

We've modified the text displayed in the `getGreetings()` function in the `MyController.java` file to display the names of our group members:



3. Jenkins pipeline and Kubernetes deployment

Information

All the commands that will be described throughout this report, as well as some of the Jenkinsfiles lines, have been run on a Windows environment. They may differ if you run the project on a Linux environment.

Jenkins configuration

Before setting up our pipeline, we made several checks to ensure the correct configuration of our Jenkins environment and its slaves for our builds. These checks included :

- Validation of the presence of a Docker installation on the slave used.
- Checking that Maven had been correctly installed and that the path to its installation was correct.

- Ensure that the credentials for the Docker Hub repository were correctly configured for image uploads.
- Confirmation of the correct configuration of the credentials needed to access our project's GitHub repository (in our case, this step was optional, since our project repository is public.)
- Setting up Kubernetes credentials to connect to the virtual server and access kubectl commands

Kubernetes configuration

To be able to deploy our applications on Kubernetes via Jenkins, we first had to configure the connection between the two.

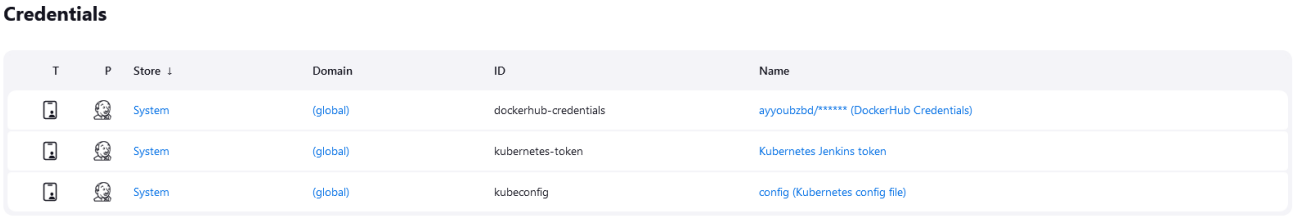
We first created and applied a service `serviceAccount.yaml` granting the necessary permissions to Jenkins (source: https://medium.com/@devayanthakur/minikube-configure-jenkins-kubernetes-plugin-25eb804d0dec).

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins
  namespace: default
---

kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: jenkins
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods","services"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
- apiGroups: [""]
```

```
  resources: ["persistentvolumeclaims"]
  verbs: ["create","delete","get","list","patch","update","watch"]

---
apiVersion: v1
kind: Secret
metadata:
  name: jenkins-token
  annotations:
    kubernetes.io/service-account.name: jenkins
type: kubernetes.io/service-account-token

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: jenkins
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins
---
# Allows jenkins to create persistent volumes
# This cluster role binding allows anyone in the "manager" group to read secrets
in any namespace.
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: jenkins-crb
subjects:
- kind: ServiceAccount
  namespace: default
  name: jenkins
roleRef:
  kind: ClusterRole
  name: jenkinsclusterrole
  apiGroup: rbac.authorization.k8s.io
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: jenkinsclusterrole
rules:
- apiGroups: [""]
```

```
    resources: ["persistentvolumes"]
    verbs: ["create","delete","get","list","patch","update","watch"]
```

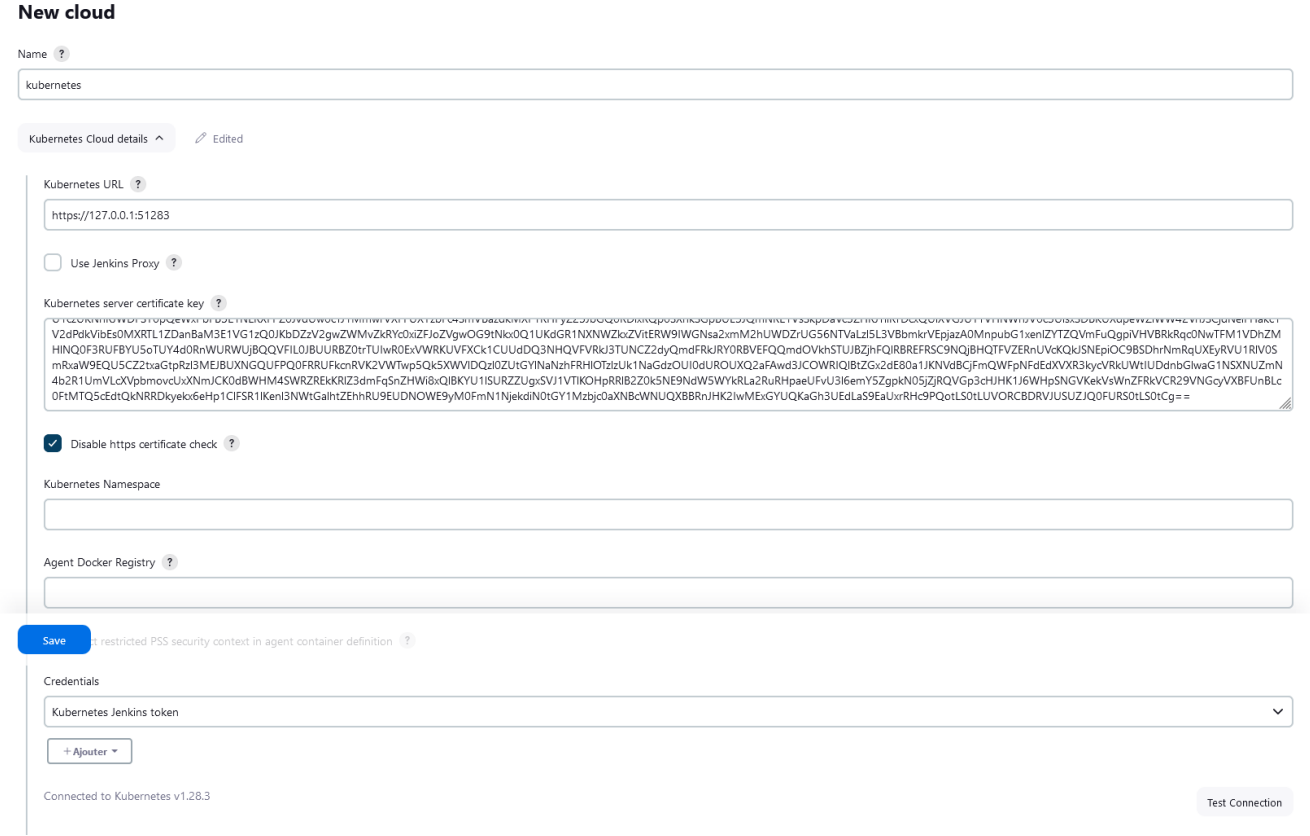
We then retrieved the token created for Jenkins using the command `kubectl describe secrets/jenkins-token`, for which we created a new credential in Jenkins, alongside a `kubeconfig` credential containing the path to the Kubernetes `config` file.



We then retrieved and decoded Minikube's authentication certificate using the following command:

```
certutil -encode "C:\Users\${User}\.minikube\ca.crt" "output.txt"
```

This enabled us to configure a Cloud so that Jenkins could access Kubernetes.



Pipeline creation

The deployment process we have set up includes the following steps:

1. Updating the project by retrieving the most recent version from the Github repository.
2. Match the Java project version number to the corresponding Jenkins build number, then update the Dockerfile accordingly.

3. Build a Docker image for our project.
4. Publish this image on the Docker Hub.
5. Deploy the application in a development environment using Kubernetes.
6. Run a validation test to ensure that the build works properly.
7. Deploy the application in a production environment once the validation test has been successfully completed.

To deploy our application via Kubernetes, we of course created the corresponding deployment files, which we then added to the project and called up in our Jenkins pipeline. Here is an example for our development build:
`./kubernetes/development.yml`

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: devops-project-development
  labels:
    app: devops-project
    env: development
spec:
  replicas: 2
  selector:
    matchLabels:
      app: devops-project
      env: development
  template:
    metadata:
      labels:
        app: devops-project
        env: development
    spec:
      containers:
      - name: devops-project-container
        image: ayyoubzbd/devops-project:latest
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: devops-project-development-service
  labels:
    app: devops-project
    env: development
spec:
  selector:
```

```
    app: devops-project
    env: development
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: NodePort
```

Here's the final pipeline as we implemented it. Please refer to the comments in the code for more details:

`./Jenkinsfile`

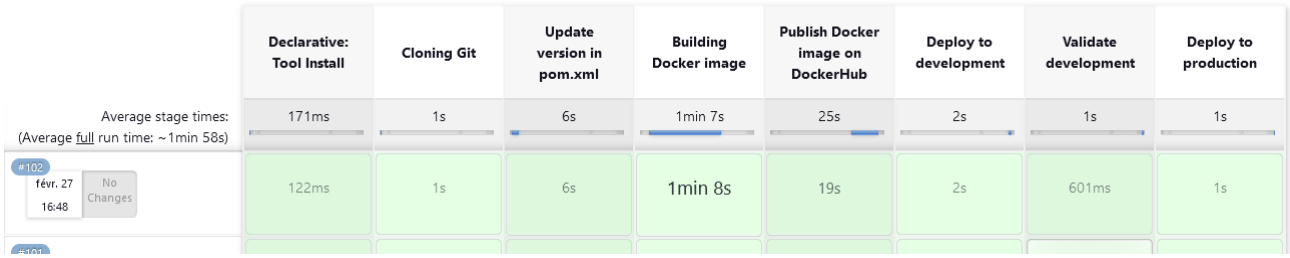
```
def buildId = env.BUILD_ID

pipeline {
    agent any

    tools {
            // Loading Maven, which then allows us to use the 'mvn' command to
update the version of our application according to the build number.
        maven 'Default'
    }

    stages {

                // STEP 1: Scan and retrieve the most recent version of the
project from Github
        stage('Cloning Git') {
            steps {
                checkout scmGit(branches: [[name: '*/main']], extensions: [],
userRemoteConfigs: [[url: 'https://github.com/ayyoub-zbd/devops-project-2024']])
            }
        }

                // STEP 2: Update application version by build number
        stage('Update version in pom.xml') {
            steps {
                bat "mvn versions:set -DnewVersion=${buildId}"
            }
        }

                // STEP 3: Create a Docker image for the version just built
        stage('Building Docker image') {
            steps {
                script {
                    projectImage = docker.build("ayyoubzbd/devops-
project:${buildId}", "--build-arg VARIABLE=${buildId} .")
                }
```

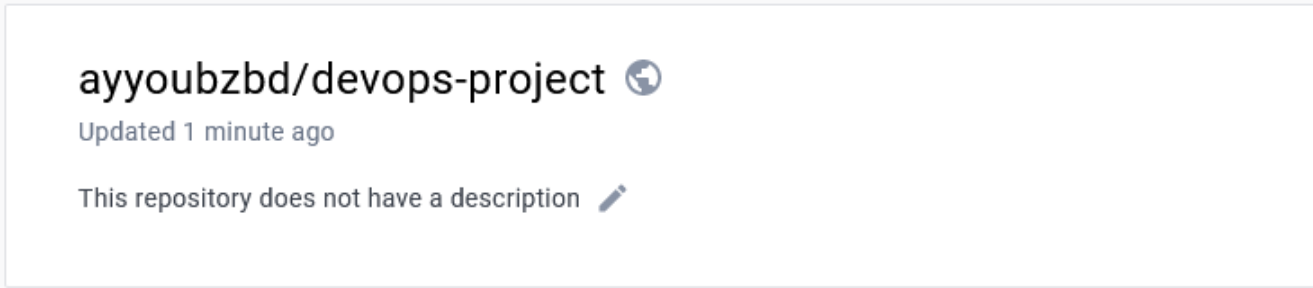
```
            }
        }

                // STEP 4: Publish Docker image on Docker Hub
        stage('Publish Docker image on DockerHub') {
            steps {
                script {
                    withDockerRegistry(credentialsId: 'dockerhub-credentials') {
                        projectImage.push()
                    }
                }
            }
        }

                // STEP 5: Deploying the application on Kubernetes, in a
development environment
        stage('Deploy to development') {
            steps {
                withKubeConfig([credentialsId: 'kubeconfig']) {
                    bat 'kubectl apply -f kubernetes/development.yaml'
                }
            }
        }

                // STEP 6: Test the response returned by our application's URL
once deployed. If it returns status 200 OK, the application has been deployed
correctly.
                // It's a very rudimentary test, but for the purposes of our
project we'll consider it sufficient.
        stage('Validate development') {
            steps {
                withKubeConfig([credentialsId: 'kubeconfig']) {
                    script {
                        def response = httpRequest
'http://127.0.0.1:8001/api/v1/namespaces/default/services/devops-project-
development-service' // Require "HTTP Request" plugin

                        if (response.status == 200) {
                            echo "Test passed!"
                        } else {
                            error "Test failed..."
                        }
                    }
                }
            }
        }

                // STEP 7: Deploying the application on Kubernetes, in a
production environment
```

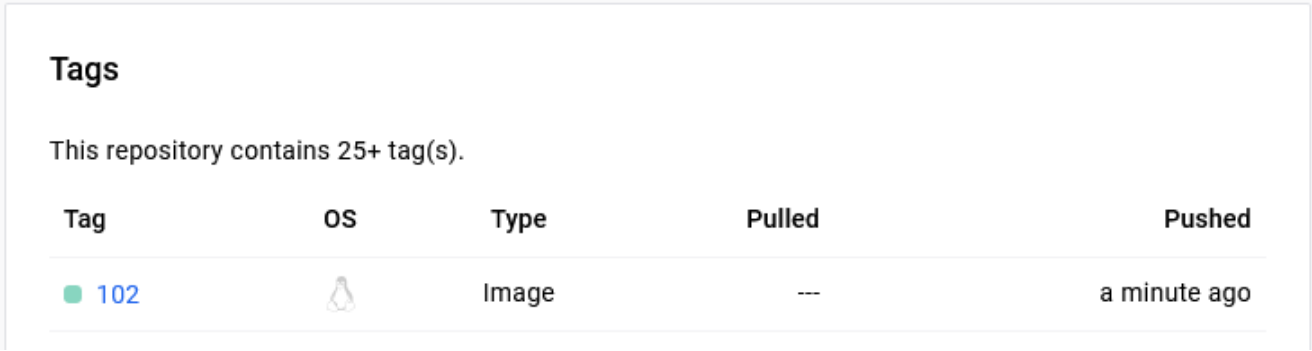
```
        stage('Deploy to production') {
            steps {
                withKubeConfig([credentialsId: 'kubeconfig']) {
                    bat 'kubectl apply -f kubernetes/production.yaml'
                }
            }
        }
    }
}
```

The pipeline is fully functional, the application and the Docker image are tagged with the correct version (here, for build n°102):

Stage View







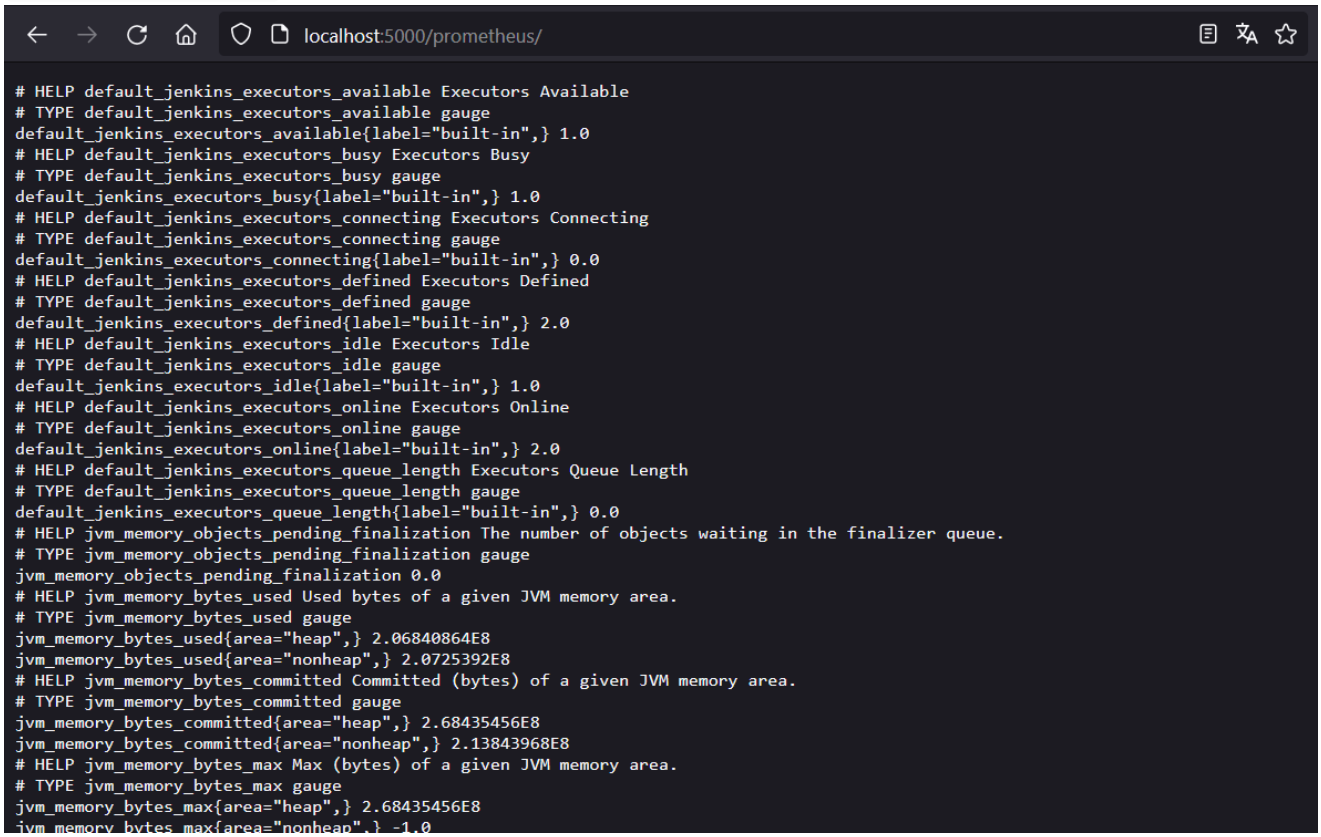
We have both development and production environments running:

```
PS C:\Users\Ayyoub> kubectl get services
NAME                                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
devops-project-development-service   NodePort    10.105.251.252   <none>        80:30476/TCP   45m
devops-project-production-service    NodePort    10.100.7.161     <none>        80:31559/TCP   2m39s
kubernetes                          ClusterIP   10.96.0.1        <none>        443/TCP        75m
```

II. Monitoring and Incident Management for containerized application

1. Jenkins configuration

To begin with, we installed the "Prometheus metrics" plugin, which we set up to update metrics every 10 seconds for our project, at the URL `http://{jenkins-url}:{port}/prometheus` :



2. Grafana installation

We started by getting the Grafana repository up to date:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
```

We then created a namespace dedicated to monitoring in our Kubernetes environment:

```
kubectl create namespace monitoring
```

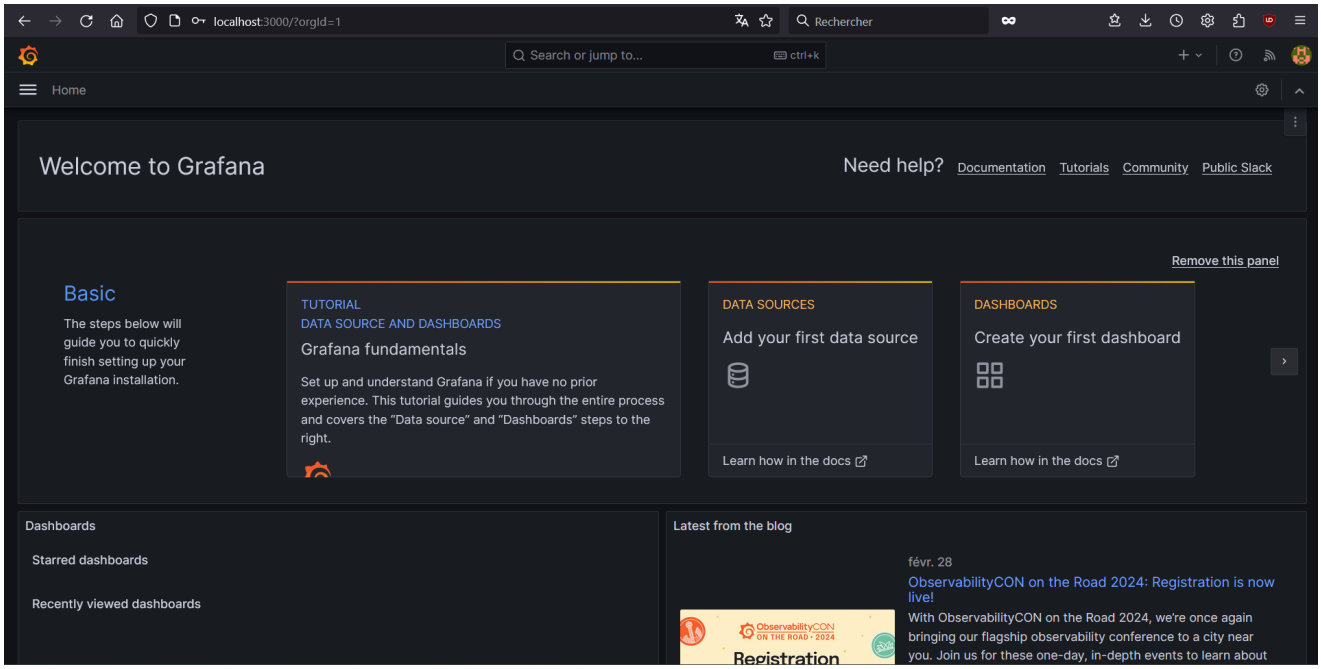
Then we installed Grafana with the following command:

```
helm install my-grafana grafana/grafana --namespace monitoring
```

```
PS C:\Users\Ayyoub> helm install my-grafana grafana/grafana --namespace monitoring
NAME: my-grafana
LAST DEPLOYED: Wed Feb 28 22:58:05 2024
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
NOTES:
1. Get your 'admin' user password by running:

   kubectl get secret --namespace monitoring my-grafana -o jsonpath="{.data.admin-password}" | base64
 --decode ; echo


2. The Grafana server can be accessed via port 80 on the following DNS name from within your cluster:


   my-grafana.monitoring.svc.cluster.local

   Get the Grafana URL to visit by running these commands in the same shell:     export POD_NAME=$(ku
bectl get pods --namespace monitoring -l "app.kubernetes.io/name=grafana,app.kubernetes.io/instance=m
y-grafana" -o jsonpath="{.items[0].metadata.name}")
     kubectl --namespace monitoring port-forward $POD_NAME 3000

3. Login with the password from step 1 and the username: admin
#################################################################################
######   WARNING: Persistence is disabled!!! You will lose your data when   #####
######            the Grafana pod is terminated.
#####
#################################################################################
```

We then realized that we couldn't continue to use the helm chart installation as requested in the instructions in a Windows environment, and rather than start the project from scratch in a Linux environment, we preferred to use the installer proposed on the official website for Windows.

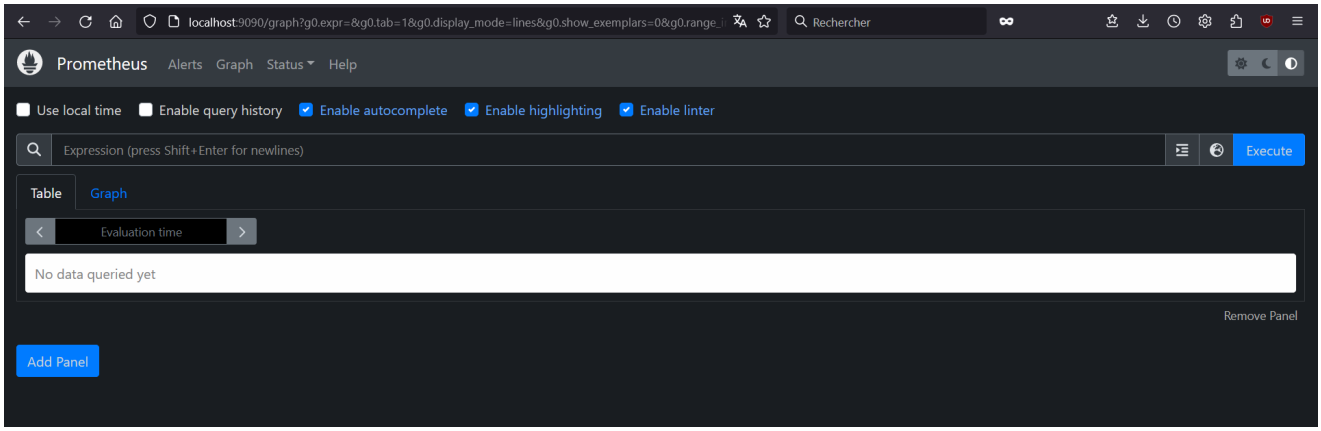
After installation, we have access to our Grafana dashboard on the default port:



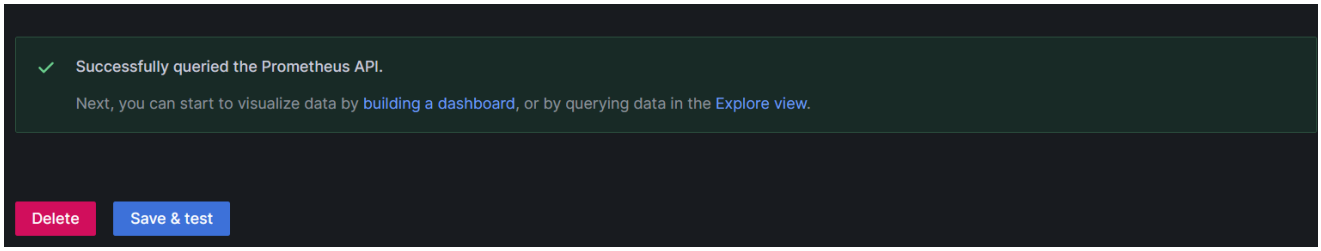
For Prometheus, we have retrieved the default configuration file `prometheus.yml` to which we have added a job for Jenkins:

```yaml
# ...

scrape_configs:
        # ...
```

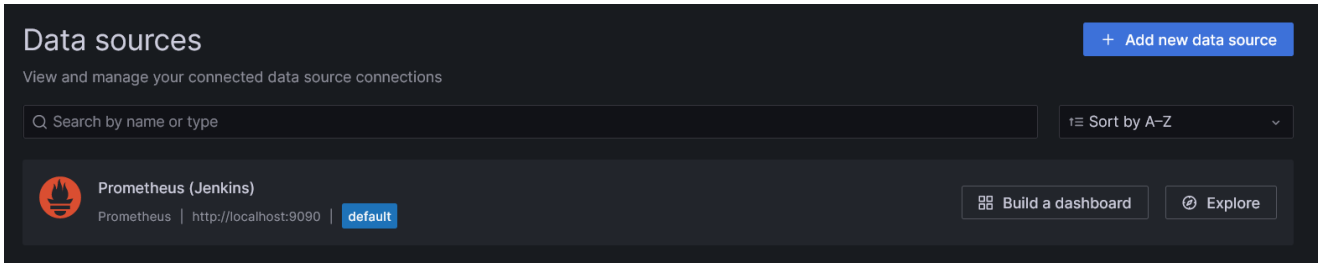
```
        - job_name: "jenkins"
                metrics_path: "/prometheus/"
                static_configs:
                - targets: ["localhost:5000"] # Our Jenkins is listening on port
5000
```



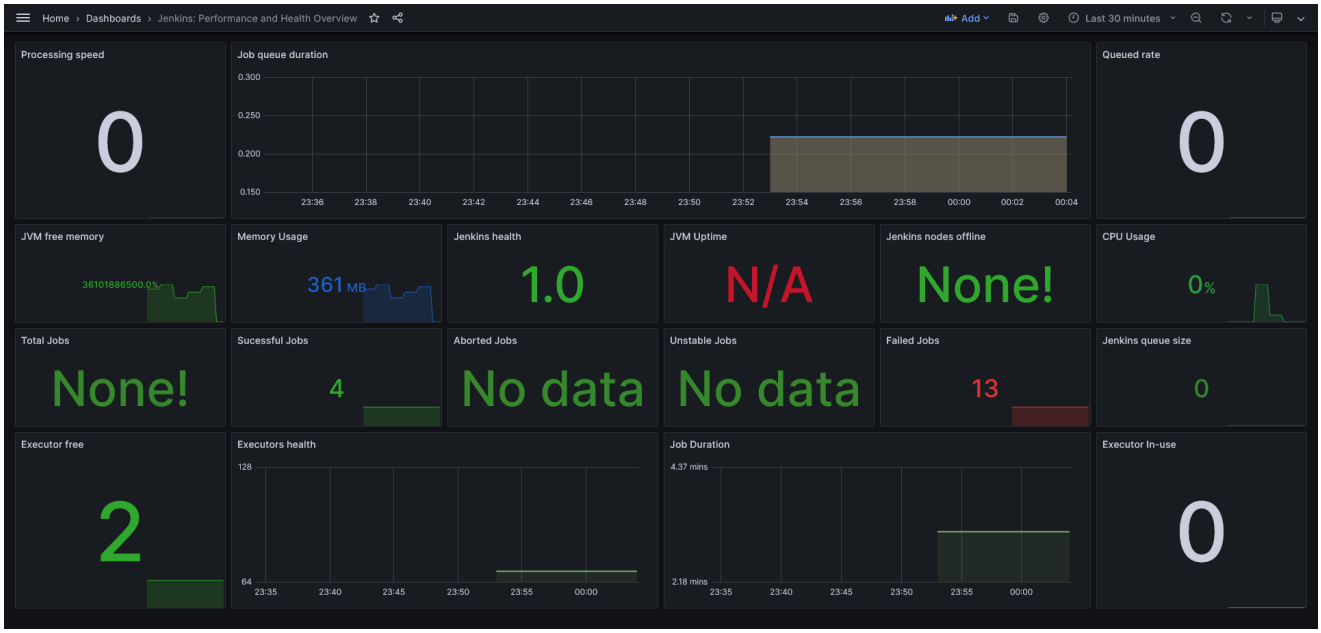
We then created a Datasource for Prometheus within Grafana:



Data sources list:



And here's our first dashboard !



3. Alert Manager

We decided to configure 2 alerts for our build as part of our project:

- An alert that is triggered when the number of replicas in our Kubernetes deployment is less than 2, as requested in the instructions.
- A message sent at the end of each build, announcing either its success or failure. The rules were defined in a `rules.yml` file in the Prometheus installation folder:

```
groups:
- name: Jenkins
         rules:
         - alert: "JenkinsLastBuildSucceded"
                expr: default_jenkins_builds_last_build_result_ordinal == 0
                for: 0m
                labels:
                        severity: info
                annotations:
                        summary: "The Jenkins build {{ $labels.jenkins_job }} was
successful."
                        description: "Hello from Ayyoub, Karam and Sofian
group!\n\nLast build succeded: {{ $labels.jenkins_job }} on {{ $labels.instance
}}"

        - alert: "JenkinsLastBuildFailed"
                expr: default_jenkins_builds_last_build_result_ordinal == 2
                for: 0m
                labels:
                        severity: critical
                annotations:
                        summary: "The Jenkins build {{ $labels.jenkins_job }}
```

```
failed."
                        description: "Hello from Ayyoub, Karam and Sofian
group!\n\nLast build failed: {{ $labels.jenkins_job }} on {{ $labels.instance }}"

- name: Kubernetes
        rules:
        - alert: KubernetesDeploymentReplicasLow
                expr:
kube_deployment_status_replicas_available{deployment="devops-project-
development"} < 2
                for: 0m
                labels:
                        severity: warning
                annotations:
                        summary: "Kubernetes deployment devops-project-
development has less than 2 replicas available"
                        description: "The Kubernetes deployment devops-project-
development has less than 2 replicas available."
```

The file is then referenced in the `prometheus.yml` configuration file:

```
# ...
rule_files:
  - rules.yml
# ...
```

In the Alert Manager configuration file, we then set up an e-mail address that will send the alert to the e-mail addresses of our choice:

```
global:
        resolve_timeout: 1m

route:
        receiver: 'gmail-notifications'

receivers:
        - name: 'gmail-notifications'
                email_configs:
                - to: "ayyoub.zebda@efrei.net, sofian.yahyaoui@efrei.net,
karam.mansour@efrei.net, mohamet.dia@intervenants.efrei.net"
                        from: ayyoub.zebda@gmail.com
                        headers:
                                subject: "[SE2] Hello from Ayyoub, Sofian and
Karam AlertManager!"
                        smarthost: smtp.gmail.com:587
                        auth_username: ayyoub.zebda@gmail.com
                        auth_identity: ayyoub.zebda@gmail.com
```

```
    auth_password: "**** **** **** ****"
    send_resolved: true
```

Normally you should have received an alert by email for a succesful build on 29/02/2024 at 12:12. Please check your spam as Efrei email addresses tend to treat it as such.