PROJET DEVOPS – Orchestration

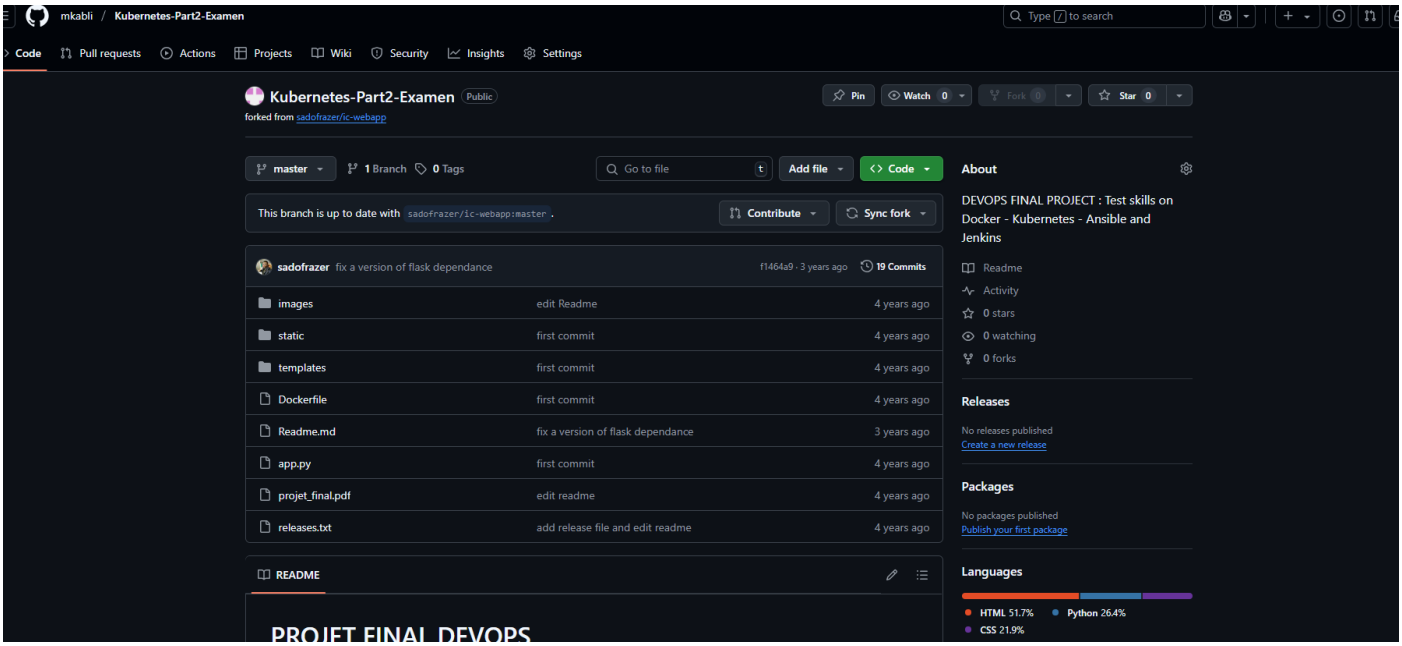
Kabli Mehdi

5SRC2

1. Introduction

Dans un premier temps nous commençons par préparer notre environnement

Nous faisons donc un fork du repo donner pour pouvoir utiliser notre propre repo



Ensuite nous clonons notre repo sur notre VM Ubuntu dans un répertoire préparer

```
mkabli@ubuntuserv:~/ProjetEXAM$ git clone https://github.com/mkabli/Kubernetes-P
art2-Examen.git
Cloning into 'Kubernetes-Part2-Examen'...
remote: Enumerating objects: 96, done.
remote: Counting objects: 100% (39/39), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 96 (delta 32), reused 32 (delta 32), pack-reused 57 (from 1)
Receiving objects: 100% (96/96), 1.14 MiB | 4.34 MiB/s, done.
Resolving deltas: 100% (37/37), done.
mkabli@ubuntuserv:~/ProjetEXAM$ ls
Kubernetes-Part2-Examen
```

Ensuite nous créons notre k8s manifests

```
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen$ mkdir k8s-manifests
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen$ ls
app.py      images         projet_final.pdf  releases.txt  templates
Dockerfile  k8s-manifests  Readme.md         static
```

Nous sommes maintenant prêt à commencer

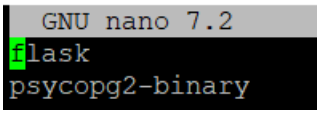
Dans un premier temps nous modifions notre fichier Dockerfile

Ce fichier Dockerfile permet de construire une image Docker qui contient une application Flask Python.
Il installe les dépendances système, copie le code source de l'application, installe les dépendances Python listées dans releases.txt, configure deux variables d'environnement (ODOO_URL et PGADMIN_URL), expose le port 5000 utilisé par Flask, et lance l'application avec app.py

```
  GNU nano 7.2
FROM python:3.6-alpine

# Dépendances système (pour psycopg2 et autres)
RUN apk add --no-cache gcc musl-dev libffi-dev openssl-dev python3-dev make

# Dossier de travail
WORKDIR /app

# Copie du code
COPY . /app

# Installation des dépendances Python
RUN pip install --no-cache-dir -r releases.txt

# Variables d'environnement (valeurs par défaut)
ENV ODOO_URL=http://odoo.local
ENV PGADMIN_URL=http://pgadmin.local

# Port exposé par Flask
EXPOSE 5000

# Commande de démarrage
CMD ["python", "app.py"]
```

Ensuite nous modifions notre fichier releases.txt qui contient la liste des bibliothèques Python nécessaires au bon fonctionnement de l'application, notamment :

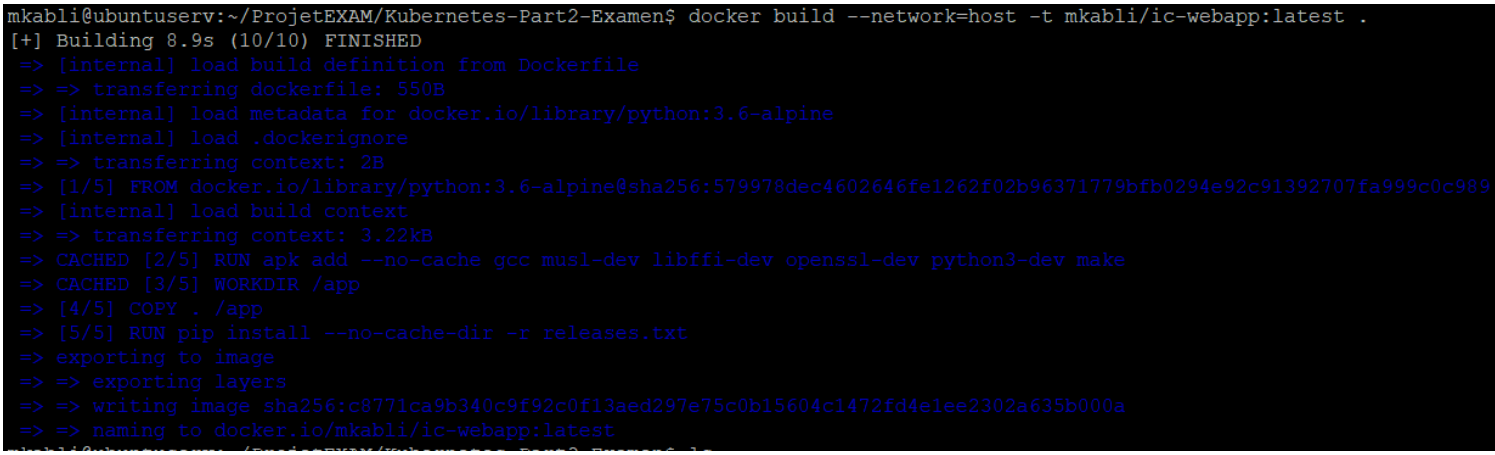
- flask : pour créer l'interface web
- psycopg2-binary : pour interagir avec une base de données PostgreSQL



Une fois les fichiers Dockerfile et releases.txt correctement configurés, j'ai procédé à la construction de l'image Docker de l'application.

Pour cela, j'ai utilisé la commande suivante :

docker build --network=host -t mkabli/ic-webapp:latest .



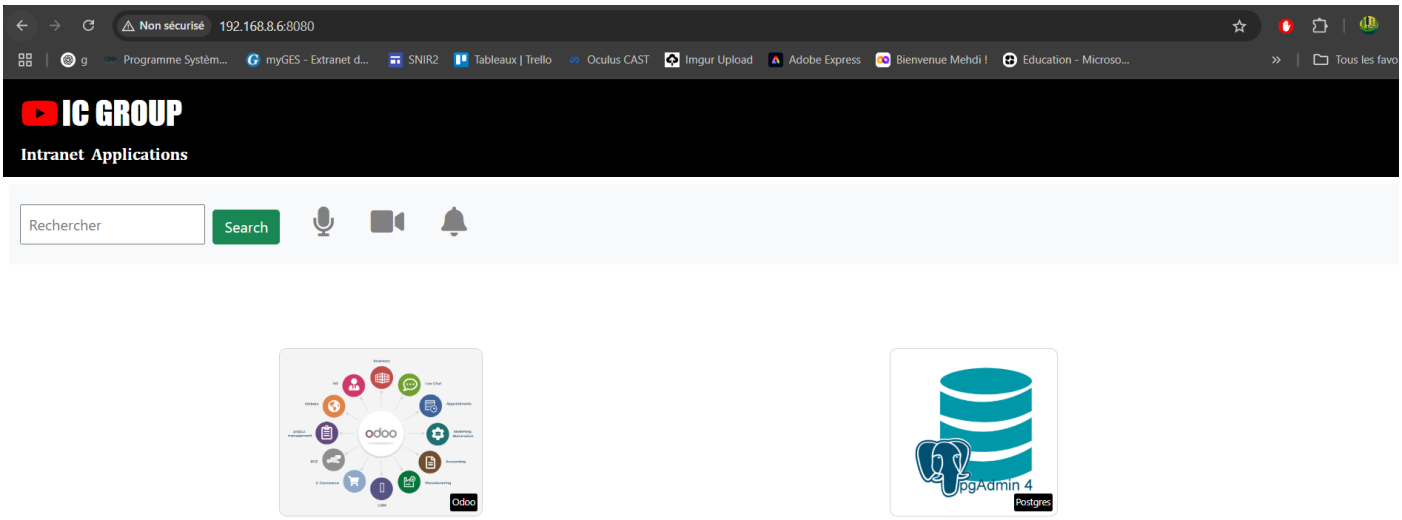
L'option --network=host a été utilisée car Docker rencontrait des problèmes DNS dans l'environnement Alpine.
Le build a permis de générer une image locale prête à être lancée.

Une fois l'image construite, j'ai démarré l'application en utilisant la commande :

docker run --network=host mkabli/ic-webapp:latest

```
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen$ docker run --network=host mkabli/ic-webapp:latest
 This is a sample web application for intranet applications display.


No Command line argument. Odoo url from environment variable =http://odoo.local
No Command line argument. Pgadmin url from environment variable =http://pgadmin.local
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://192.168.8.6:8080/ (Press CTRL+C to quit)
192.168.8.7 - - [17/May/2025 13:36:37] "GET / HTTP/1.1" 200 -
192.168.8.7 - - [17/May/2025 13:36:37] "GET /static/images/PgAdmin4.png HTTP/1.1" 200 -
192.168.8.7 - - [17/May/2025 13:36:37] "GET /static/CSS/bootstrap.min.css HTTP/1.1" 200 -
192.168.8.7 - - [17/May/2025 13:36:37] "GET /static/CSS/style.css HTTP/1.1" 200 -
192.168.8.7 - - [17/May/2025 13:36:37] "GET /static/images/odoo.jpg HTTP/1.1" 200 -
192.168.8.7 - - [17/May/2025 13:36:37] "GET /static/images/img-icon1.png HTTP/1.1" 200 -
```

Notre application est maintenant accessible !



2. Conteneurisation de l’application web.

Dans un premier temps nous créons les fichiers de déploiements en nous situant dans le dossier k8s

Ce fichier décrit comment **Kubernetes doit créer et gérer le pod** qui exécute l’application Flask.

```
  GNU nano 7.2                                                  ic-webapp-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ic-webapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ic-webapp
  template:
    metadata:
      labels:
        app: ic-webapp
    spec:
      containers:
      - name: ic-webapp
        image: mkabli/ic-webapp:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
```

Ce fichier crée un **Service Kubernetes** pour permettre l’accès au pod depuis l’extérieur.

```
  GNU nano 7.2                                                    ic-webapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: ic-webapp
spec:
  type: LoadBalancer
  selector:
    app: ic-webapp
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Nous appliquons ensuite nos manifeste dans minikube :

```
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl apply -f ic-webapp-deplo
yment.yaml
deployment.apps/ic-webapp created
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl apply -f ic-webapp-servi
ce.yaml
service/ic-webapp created
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$
```

Ceci affiche les services déployés dans le cluster. Ici, ic-webapp est bien exposé sur le port 8080 et attribué à une IP interne (10.106.7.31)

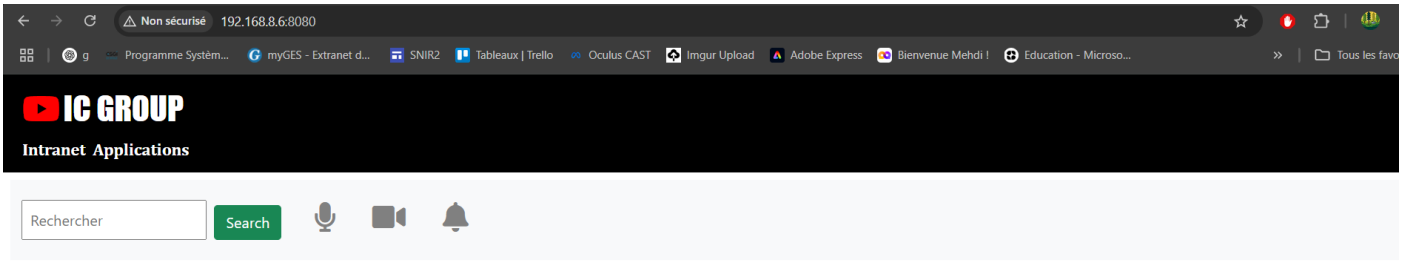
```
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl ge
t svc
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
ic-webapp    LoadBalancer   10.106.7.31    10.106.7.31   8080:30080/TCP   19h
kubernetes   ClusterIP      10.96.0.1      <none>        443/TCP          19h
```

Redirige le port 8080 sur **toutes les interfaces réseau** de la VM (y compris 192.168.8.6, ma carte bridge)

```
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl po
rt-forward service/ic-webapp 8080:8080
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
^Cmkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifestskubectl port-forward
 --address 0.0.0.0 service/ic-webapp 8080:8080
Forwarding from 0.0.0.0:8080 -> 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
```

L'application web Flask a bien été conteneurisée avec Docker, déployée dans un cluster Kubernetes local via Minikube, et exposée grâce à un service LoadBalancer.

L'accès à l'interface web a été testé et validé depuis la machine hôte à l'adresse http://192.168.8.6:8080, prouvant le bon fonctionnement du déploiement.



L'application a été **construite localement avec docker build** dans le contexte Docker de Minikube (eval $(minikube docker-env)).

Le pod a été déployé via un fichier YAML (Deployment), et le service via un second fichier (Service).

L'accès externe a été rendu possible avec un port-forward lié à l'interface bridge de la VM.

3. Déploiement des différentes applications dans un cluster Kubernetes

Pour commencer nous ajoutons le champs namespace: icgroup dans notre fichier deployment et service

```
 GNU nano 7.2                                                    ic-webapp-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ic-webapp
  namespace: icgroup
  labels:
    env: prod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ic-webapp
  template:
    metadata:
      labels:
        app: ic-webapp
        env: prod
    spec:
      containers:
      - name: ic-webapp
        image: mkabli/ic-webapp:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
```

```
 GNU nano 7.2                                                    ic-webapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: ic-webapp
  namespace: icgroup
  labels:
    env: prod
spec:
  type: LoadBalancer
  selector:
    app: ic-webapp
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Ensuite ici nous créons notre namespace puis déployons nos ressources deployment et service

```
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl create namespace icgroup
namespace/icgroup created
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl label namespace icgroup env=p
rod
namespace/icgroup labeled
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl apply -f ic-webapp-deployment
.yaml
kubectl apply -f ic-webapp-service.yaml
deployment.apps/ic-webapp created
service/ic-webapp created
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$
```

Déploiement d'Odoo

Toujours dans notre dossier k8s nous créons un nouveau fichier yaml odoo-db-pvc.yaml qui permettra de crée un **volume persistant** pour stocker les données de PostgreSQL (base de données Odoo) afin qu'elles ne soient **pas perdues si le pod redémarre**.

```
  GNU nano 7.2                                    odoo-db-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: odoo-db-pvc
  namespace: icgroup
  labels:
    env: prod
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Nous créons ensuite notre fichier deployment odoo-db-deployment.yaml qui permet de déployer le **pod PostgreSQL** qui va héberger la base de données de l'application Odoo.

```
  GNU nano 7.2                                    odoo-db-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: odoo-db
  namespace: icgroup
  labels:
    env: prod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: odoo-db
  template:
    metadata:
      labels:
        app: odoo-db
        env: prod
    spec:
      containers:
      - name: postgres
        image: postgres:13
        env:
        - name: POSTGRES_DB
          value: odoo
        - name: POSTGRES_USER
          value: odoo
        - name: POSTGRES_PASSWORD
          value: odoo
        ports:
        - containerPort: 5432
        volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: postgres-data
      volumes:
      - name: postgres-data
        persistentVolumeClaim:
          claimName: odoo-db-pvc
```

Et enfin nous créons le fichier odoo-db-service.yaml qui permettra de crée un **service interne (ClusterIP)** pour que le pod Odoo puisse **se connecter à la base PostgreSQL via le nom DNS odoo-db**.

```
  GNU nano 7.2                                  odoo-db-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: odoo-db
  namespace: icgroup
  labels:
    env: prod
spec:
  selector:
    app: odoo-db
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432
```

Nous appliquons ensuite ces 3 fichiers avec :

kubectl apply -f odoo-db-pvc.yaml

kubectl apply -f odoo-db-deployment.yaml

kubectl apply -f odoo-db-service.yaml

On peu donc voir que sa tourne correctement :

```
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$ kubectl get all -n icgroup
NAME                             READY   STATUS    RESTARTS   AGE
pod/ic-webapp-668b985fd9-zxkpp   1/1     Running   0          11m
pod/odoo-db-75d9b595fb-fdcjv     1/1     Running   0          6m27s

NAME                 TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
service/ic-webapp    LoadBalancer   10.107.206.41   10.107.206.41   8080:32143/TCP   11m
service/odoo-db      ClusterIP      10.107.18.87    <none>          5432/TCP         6m27s

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ic-webapp    1/1     1            1           11m
deployment.apps/odoo-db      1/1     1            1           6m27s

NAME                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/ic-webapp-668b985fd9   1         1         1       11m
replicaset.apps/odoo-db-75d9b595fb     1         1         1       6m27s
mkabli@ubuntuserv:~/ProjetEXAM/Kubernetes-Part2-Examen/k8s-manifests$
```