Rapport de projet CRV
(Autoscaling et IAC)

Auteurs : MEDION MBAINAISSEM Narcisse & Moubarak HASSAN SALIFOU

Lien vers le git : https://github.com/Medion-ing/Projets-Autoscaling-IAC.git

1. Introduction

Objectif du projet **:**

L'objectif de ce projet est de créer une infrastructure dynamique et scalable pour l'application Redis-Node.js avec un frontend React. Cette infrastructure doit être déployée dans un cluster Kubernetes, et permettre une montée en charge automatique des composants de l'application selon les besoins. Plus précisément :

- **Redis** : La base de données Redis fonctionnera en mode "master-replica", où la base principale accepte les écritures et les réplicas permettent d'effectuer des lectures parallèles. L'auto-scaling sera principalement appliqué aux réplicas.
- **Node.js** : Le serveur Node.js, qui est stateless, interagira avec Redis. Il pourra être dupliqué sans affecter le comportement de l'application, et il sera configuré pour un auto-scaling en fonction de la charge.
- **React** : Le frontend React interagit avec le serveur Node.js pour afficher les données. Bien que l'auto-scaling du frontend ne soit pas nécessairement une priorité dans ce projet, il sera configuré pour vérifier que tout fonctionne correctement.
- **Prometheus/Grafana** : Ces outils seront utilisés pour le monitoring. Prometheus collectera les métriques de l'application et de Kubernetes, tandis que Grafana fournira des tableaux de bord pour observer la performance et le comportement des composants.

Technologies utilisées **:**

- **Kubernetes** : Plateforme de gestion de conteneurs pour orchestrer les différents composants de l'application.
- **Docker** : Pour créer des images conteneurisées des services (Redis, Node.js, React).
- **Prometheus** : Outil de monitoring qui collecte des métriques sur les services et le cluster Kubernetes.
- **Grafana** : Outil de visualisation des métriques collectées par Prometheus.
- **Redis** : Base de données clé-valeur en mémoire, utilisée pour les opérations rapides.
- **Node.js** : Serveur backend qui interagit avec Redis et expose une API.
- **React** : Framework frontend pour l'affichage des données.

Lien vers le git : https://github.com/Medion-ing/Projets-Autoscaling-IAC.git

L'infrastructure

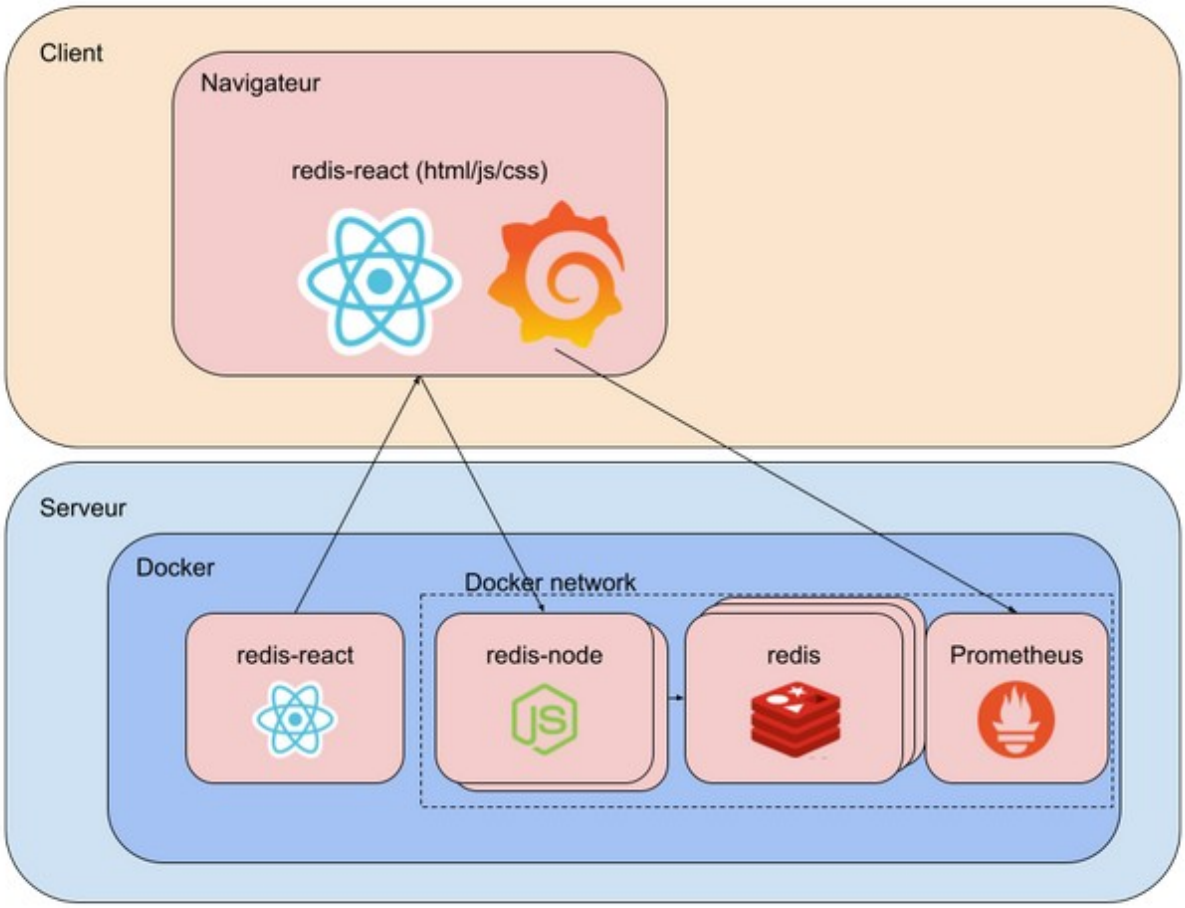


Fig1 :L'architecture de l'infrastructure

```
nabster@Ciscko:~/k8s-deployments$ kubectl get nodes
NAME       STATUS   ROLES           AGE    VERSION
minikube   Ready    control-plane   5d1h   v1.32.0
```

Fig2 : Etat de notre cluster Kubernetes

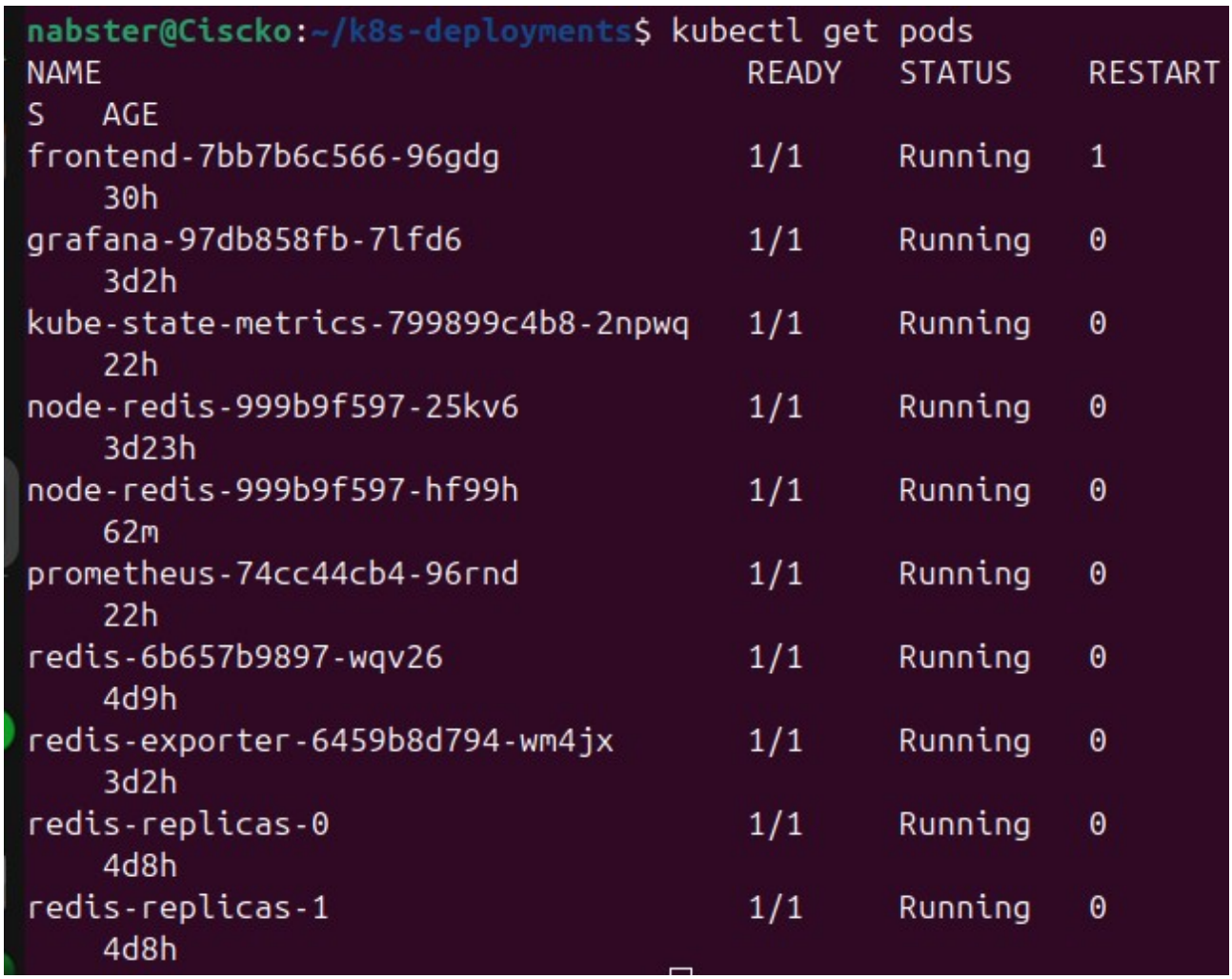


Fig3 : Pods déployés sur notre cluster Kubernetes

2. **Configuration de l'infrastructure**

Déploiement Kubernetes **:**

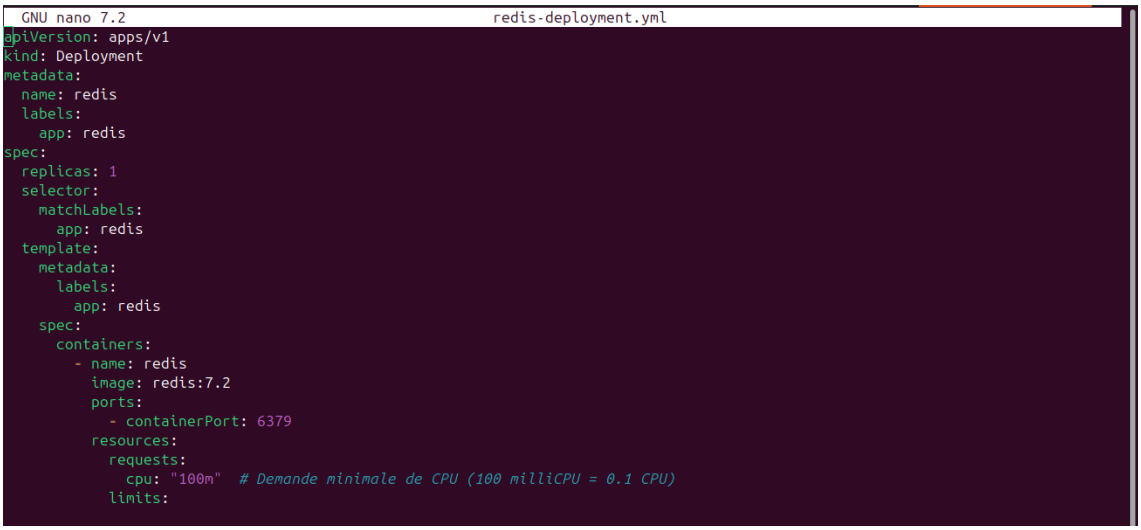
L'infrastructure a été déployée sur **Kubernetes** en utilisant plusieurs fichiers de configuration pour différents composants de l'application. Voici un aperçu des étapes et des ressources impliquées :

1. **Création des fichiers de configuration Kubernetes** :
    - **Redis** : Nous avons configuré Redis avec un Deployment simple pour la gestion des clés en mémoire. Il utilise une image Docker officielle de Redis, avec une ressource CPU demandée de 100m et une limite de 500m.
    - **Node.js** : Le Deployment de l'application Node.js contient un conteneur qui expose l'application sur le port 8080. L'application est configurée pour se connecter à Redis, avec des variables d'environnement spécifiées pour REDIS_URL et REDIS_REPLICAS_URL.
    - **React** : Le frontend React est déployé dans un conteneur avec une seule réplique (au départ), et le service est exposé via un NodePort.
    - **Prometheus** : Prometheus est déployé avec un Deployment et un ConfigMap qui spécifie la configuration de scraping des métriques pour Node.js, Redis, et Kubernetes.

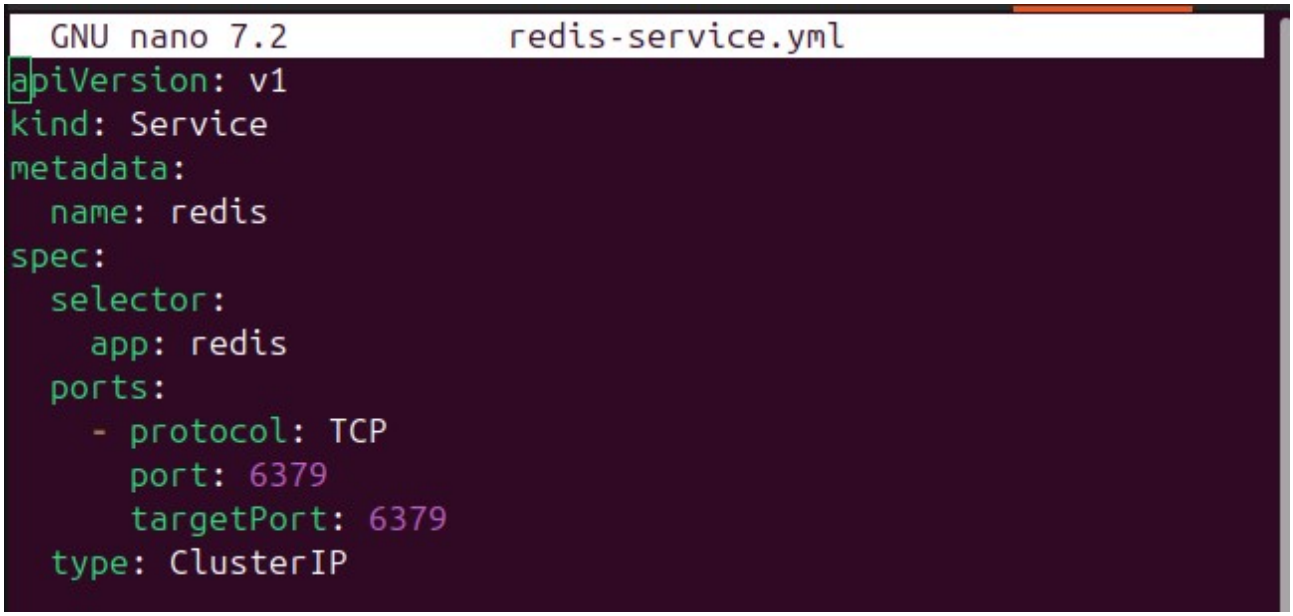
- **Grafana** : Grafana est configuré avec un Deployment et un ConfigMap pour inclure la source de données Prometheus, permettant l'affichage des métriques collectées.

Voici un aperçu des principaux fichiers de configuration utilisés :

- **Redis Deployment** :
  - redis-deployment.yml (Déploie Redis sur Kubernetes)



  - redis-service.yml (Expose Redis en interne à travers un ClusterIP)



- **Node.js Deployment** :
  - node-redis-deployment.yml (Déploie l'application Node.js)

```
  GNU nano 7.2                                    node-redis-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-redis
  labels:
    app: node-redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-redis
  template:
    metadata:
      labels:
        app: node-redis
    spec:
      containers:
        - name: node-redis
          image: redis-node-server
          imagePullPolicy: Never
          ports:
            - containerPort: 8080
          env:
            - name: PORT
              value: '8080'
```

- node-redis-service.yml (Expose l'application Node.js via un NodePort)

```
  GNU nano 7.2          node-redis-service.yml
apiVersion: v1
kind: Service
metadata:
  name: node-redis
spec:
  type: NodePort
  selector:
    app: node-redis
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort:
```

- **Frontend (React)** :
  - frontend-deployment.yml (Déploie l'application React)

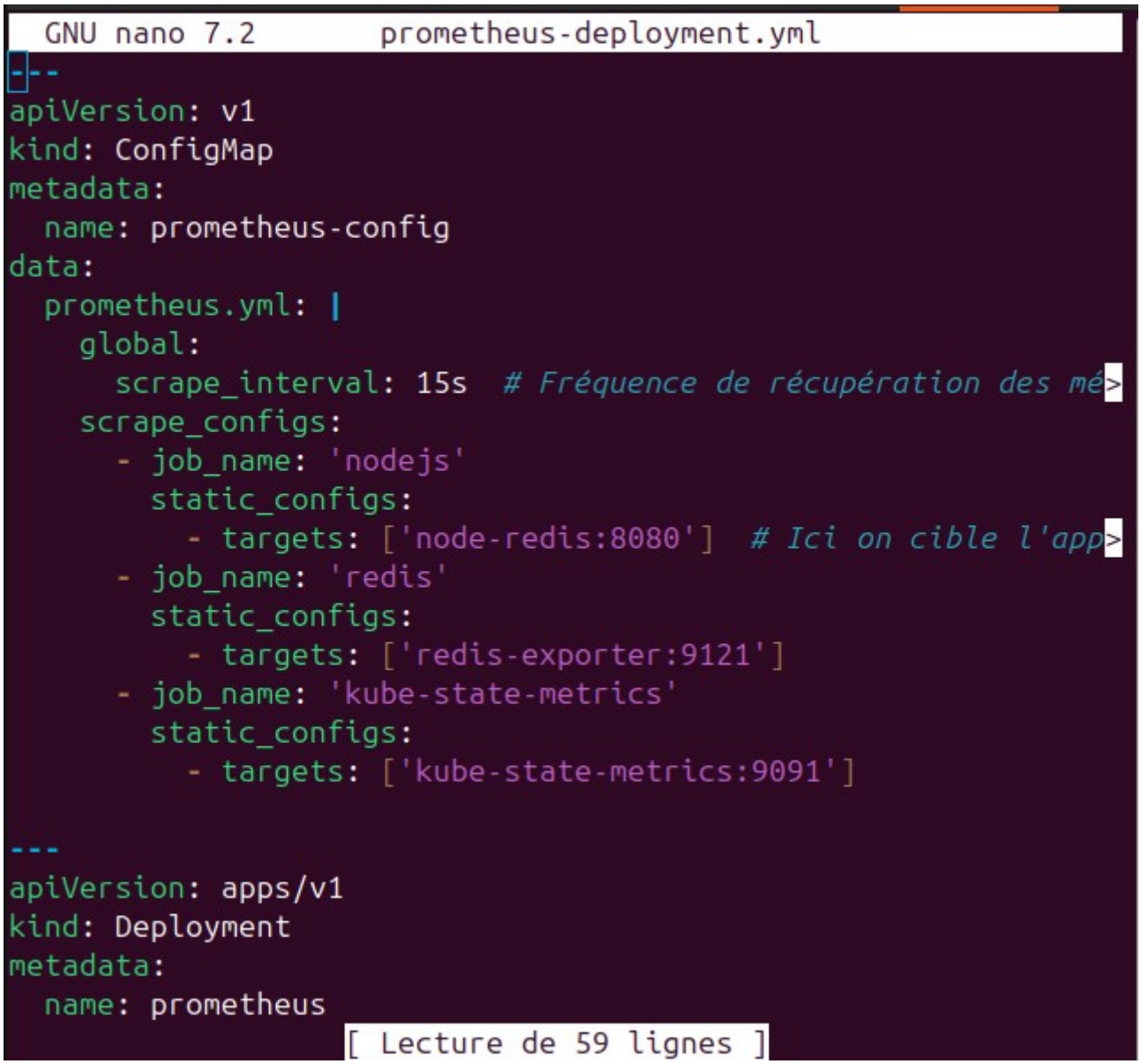
```
  GNU nano 7.2          frontend-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: redis-react-frontend
          imagePullPolicy: Never
          ports:
            - containerPort: 80
          env:
            - name: API_URL
              value: "http://node-redis:8080"
```

- frontend-service.yml (Expose le frontend via un NodePort)

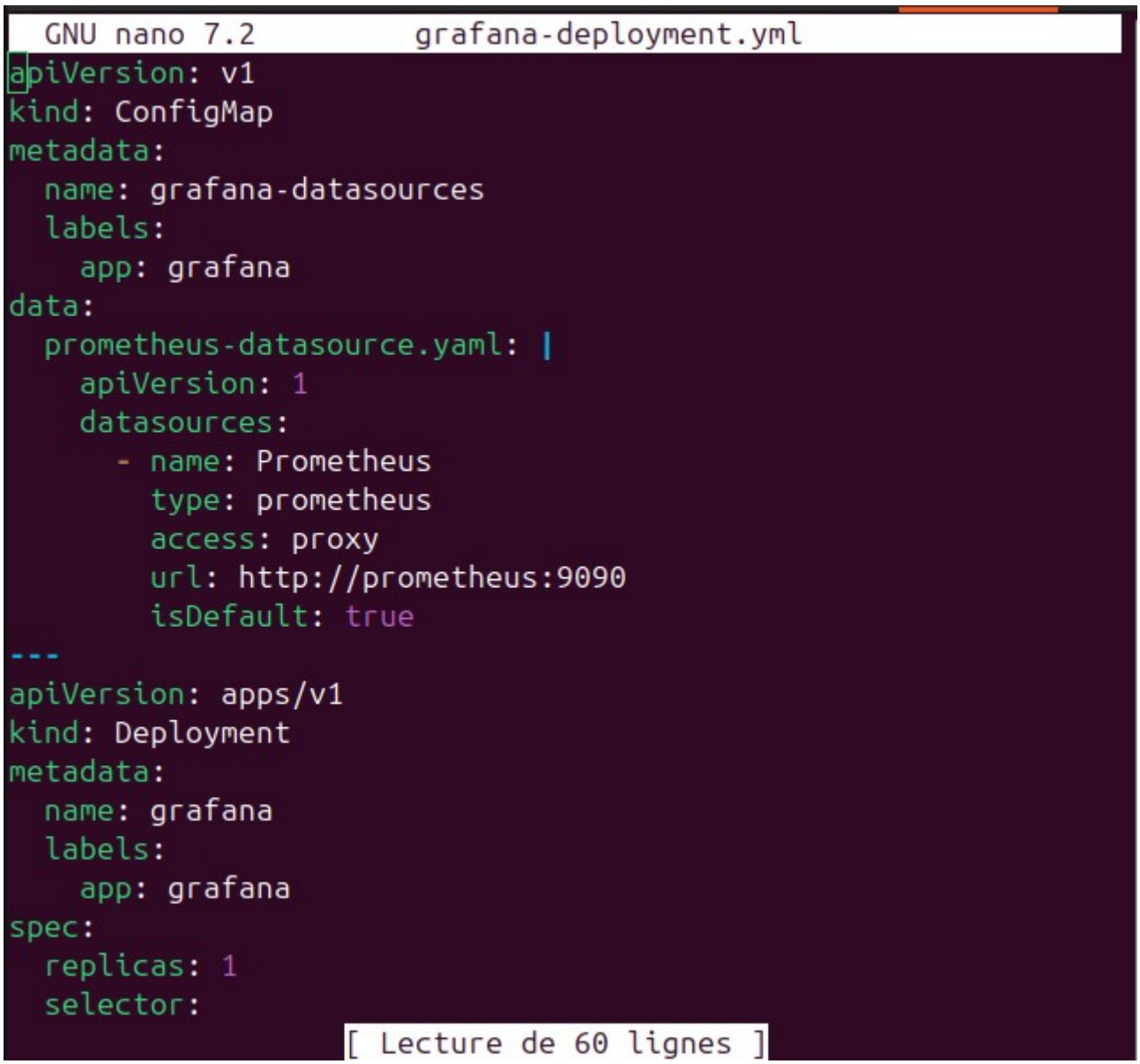
```
  GNU nano 7.2          frontend-service.yml
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

- **P**

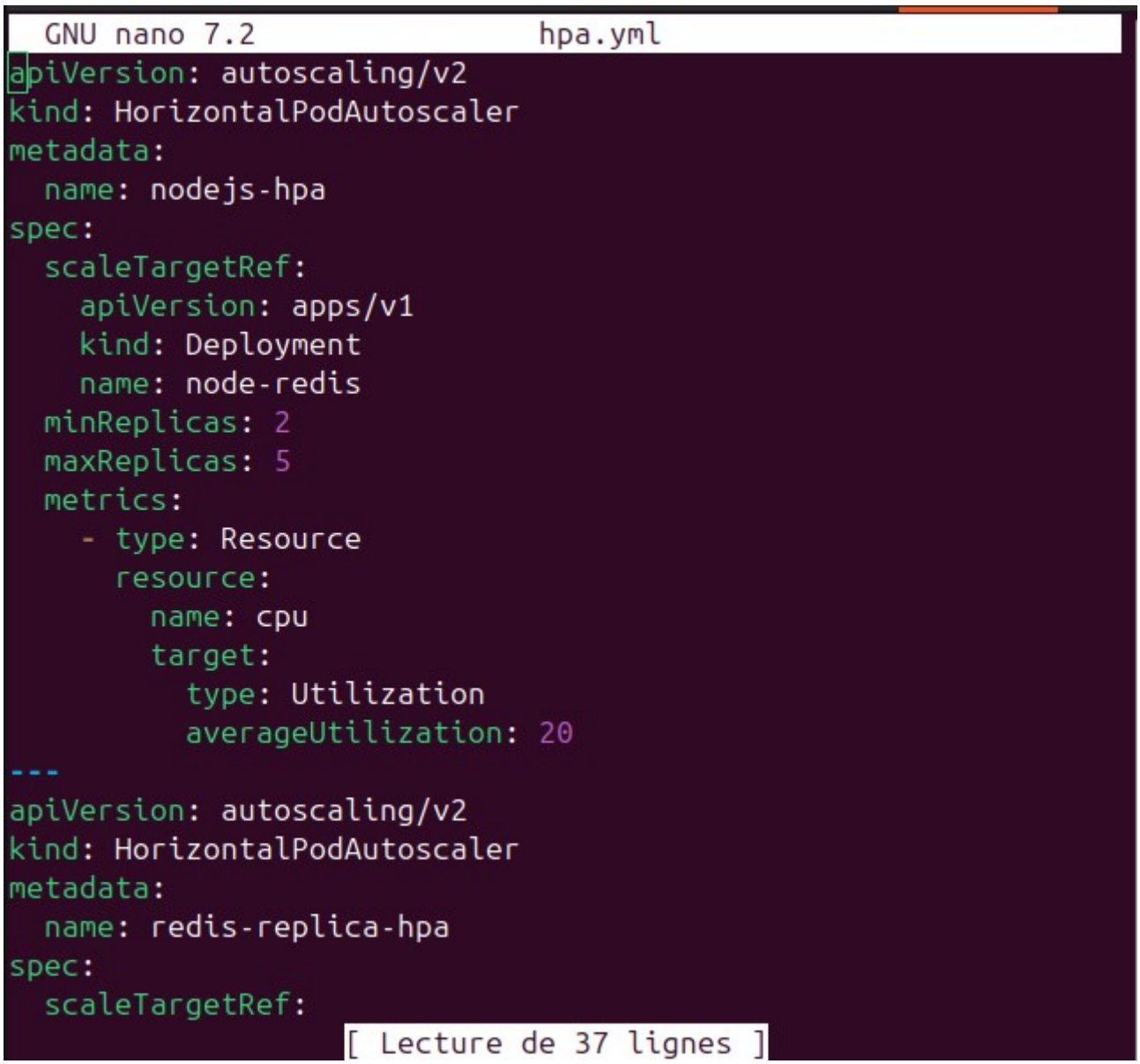
- prometheus-deployment.yml (Déploie Prometheus avec son ConfigMap pour les configurations de scraping)



- grafana-deployment.yml (Déploie Grafana avec sa source de données Prometheus)

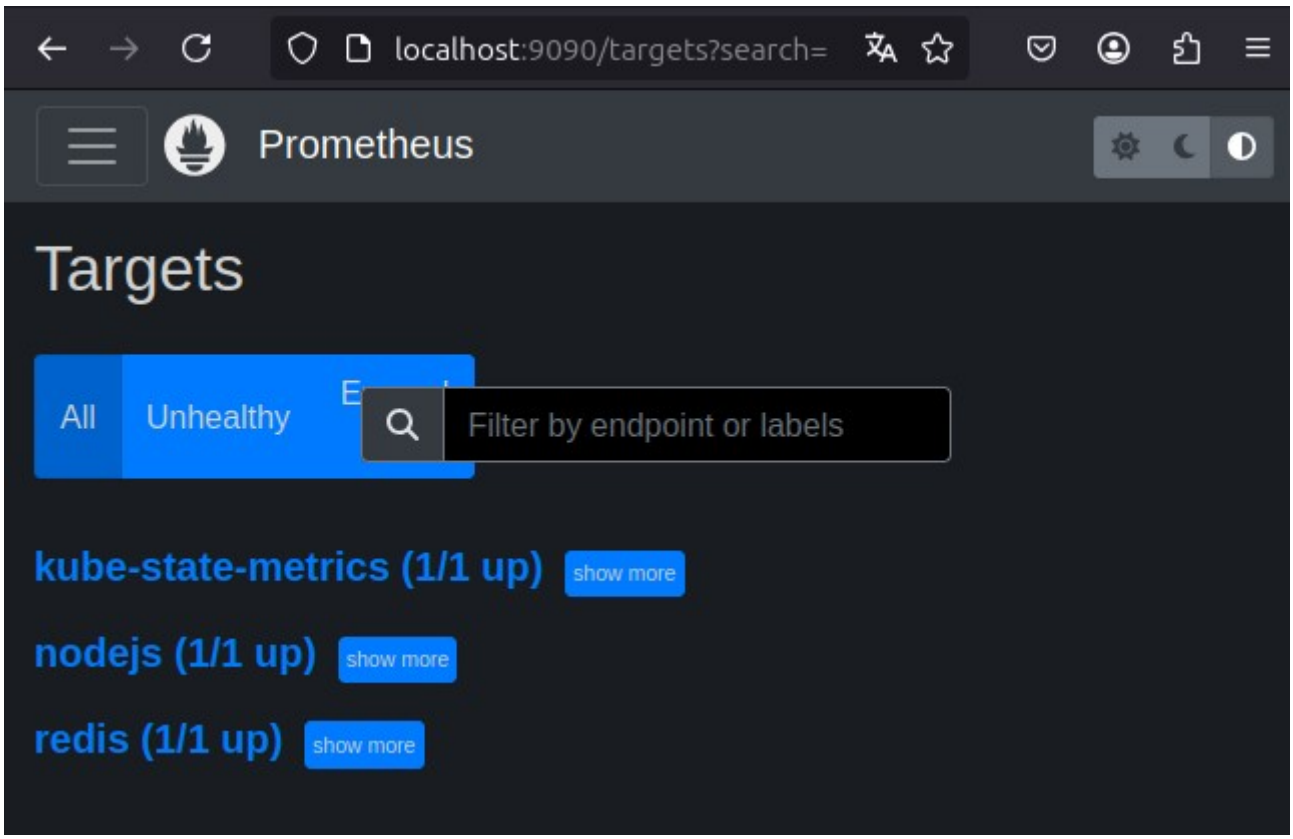


- **Auto-scaling avec HorizontalPodAutoscaler (HPA)** :
    - hpa.yml (Contient des ressources pour l'auto-scaling, en particulier pour Node.js et Redis Replicas)

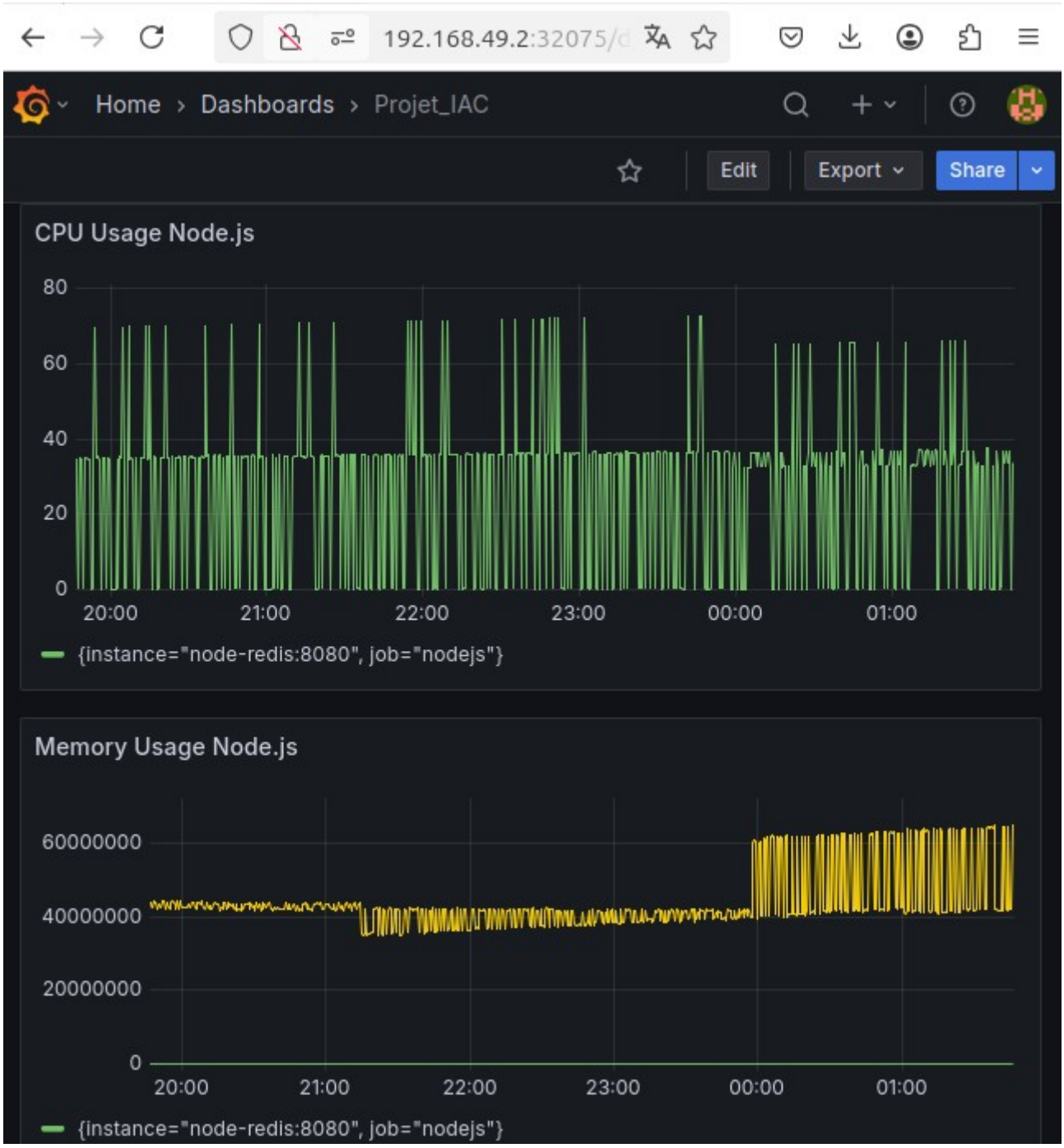


Mise en place du monitoring avec Prometheus/Grafana **:**

1. **Prometheus** :
    - Prometheus est utilisé pour récupérer les métriques de l'application, avec une fréquence de scraping définie à 15 secondes. Il scrute les métriques de l'application Node.js, de Redis via redis-exporter, et des métriques système de Kubernetes via kube-state-metrics.
    - Le ConfigMap associé permet de configurer les sources de scraping.

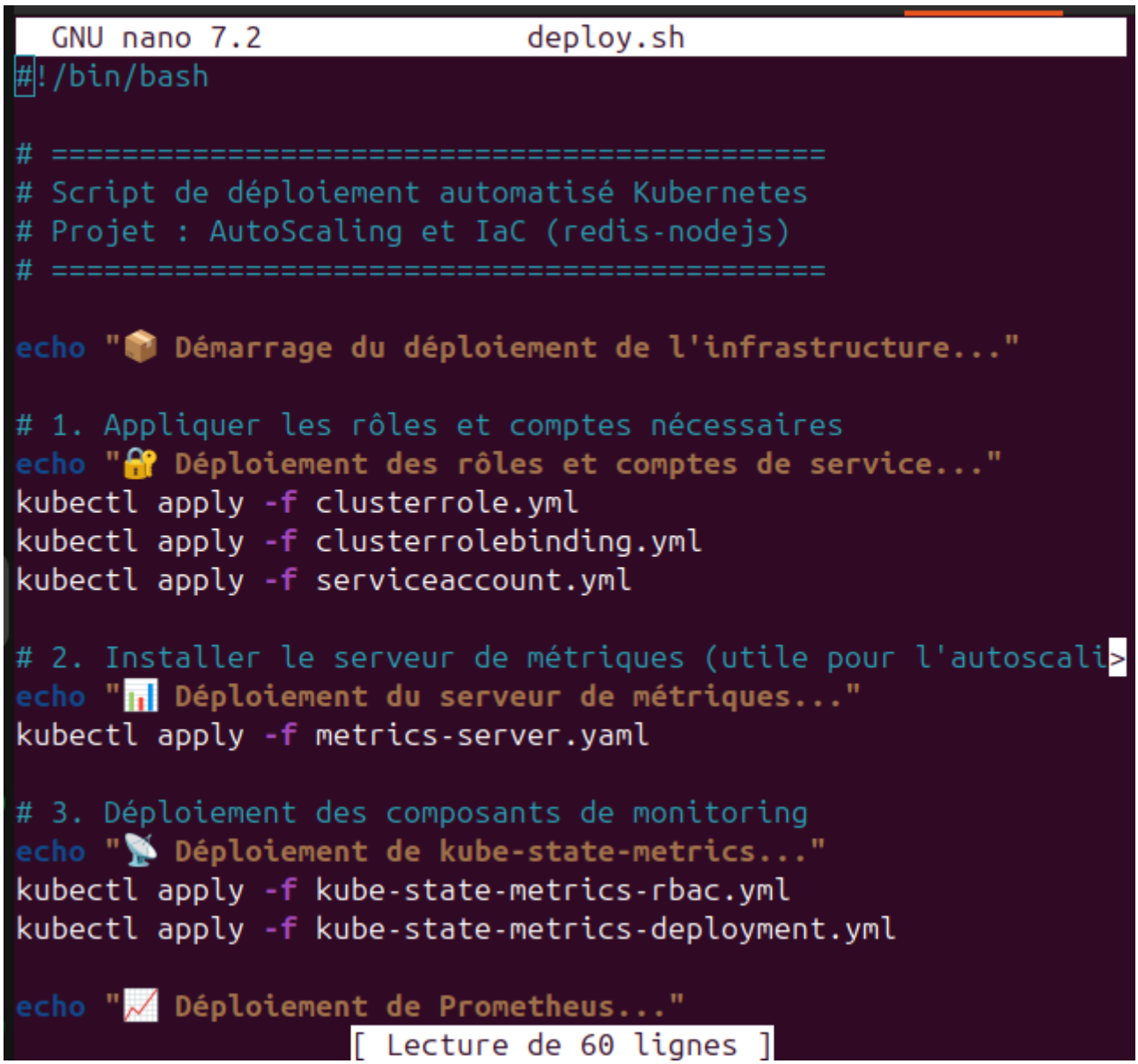


2. **Grafana** :
    - Grafana est déployé avec un tableau de bord configuré pour se connecter à Prometheus comme source de données. Le tableau de bord permet de visualiser les métriques de performance de l'application, comme l'utilisation du CPU et de la mémoire.



Automatisation avec des scripts **:**

1. **Création de scripts Bash pour l'automatisation** :
    - Le script deploy.sh est utilisé pour déployer l'ensemble de l'application sur Kubernetes. Ce script applique les fichiers de configuration Kubernetes pour les différents composants : Redis, Node.js, React, Prometheus, Grafana et les HPA.
    - Il permet également de vérifier l'état des pods et de redémarrer les composants si nécessaire.



Tests d’auto-scaling **:**

1. **Test de montée en charge** :
    - L'auto-scaling des réplicas a été testé en générant une charge sur l'application Node.js et Redis. Le HorizontalPodAutoscaler a été configuré pour augmenter ou réduire le nombre de réplicas en fonction de l'utilisation du CPU, avec des valeurs minimales et maximales définies pour chaque composant.

La commande kubectl get hpa montre les réplicas en fonction de la charge.

```
nabster@Ciscko:~/k8s-deployments$ kubectl get hpa
NAME                 REFERENCE                       TARGETS
  MINPODS   MAXPODS   REPLICAS   AGE
cpu-load-hpa         Deployment/cpu-load             cpu: 250%/80%
  1         10        10         2d1h
nodejs-hpa           Deployment/node-redis           cpu: 5%/20%
  2         5         2          5d1h
redis-replica-hpa    StatefulSet/redis-replicas      cpu: 6%/20%
  2         5         2          5d1h
```

Exemple de capture des métriques collectées par kubectl top pods, montrant l'utilisation du CPU et la mise à l'échelle des réplicas.

```
nabster@Ciscko:~/k8s-deployments$ kubectl top pods
NAME                                 CPU(cores)   MEMORY(bytes
)
frontend-7bb7b6c566-96gdg            0m           7Mi

grafana-97db858fb-7lfd6              7m           91Mi

kube-state-metrics-799899c4b8-2npwq  12m          14Mi

node-redis-999b9f597-25kv6           5m           42Mi

node-redis-999b9f597-hf99h           5m           61Mi

prometheus-74cc44cb4-96rnd           1m           44Mi

redis-6b657b9897-wqv26               2m           5Mi

redis-exporter-6459b8d794-wm4jx      2m           10Mi

redis-replicas-0                     2m           5Mi

redis-replicas-1                     2m           8Mi
```

3. **Déploiement de l'application Node.js**

Création et configuration du serveur Node.js

Dans cette étape, l'objectif est de déployer le serveur **Node.js** de manière **stateless**, ce qui signifie que chaque instance du serveur peut être redémarrée ou remplacée sans perdre de données. Le serveur Node.js doit pouvoir communiquer avec **Redis**, qui sert de base de données pour l'application.

Dockerisation de l'application Node.js : La première étape pour déployer l'application est de créer une image Docker qui contient l'application Node.js. Le fichier Dockerfile est utilisé pour définir l'environnement d'exécution de l'application. Voici le contenu de ton Dockerfile :

```
nabster@Ciscko:~/redis-node$ cat Dockerfile
# Utilisation de l'image Node.js officielle
FROM node:18

# Définir le répertoire de travail
WORKDIR /app

# Copier les fichiers nécessaires
COPY package.json yarn.lock ./

# Installer les dépendances
RUN yarn install

# Copier le reste du projet
COPY . .

# Exposer le port d'écoute
EXPOSE 3000

# Définir l'URL de Redis dans Docker
ENV REDIS_URL=redis://redis:6379

# Démarrer l'application
CMD ["node", "main.js"]
```

Explication :

- **FROM node:18** : Utilise l'image officielle de Node.js, version 18.
- **WORKDIR /app** : Définit le répertoire de travail dans lequel les commandes suivantes seront exécutées.
- **COPY package.json yarn.lock ./** : Copie les fichiers de configuration de l'application nécessaires pour installer les dépendances.
- **RUN yarn install** : Installe les dépendances à l'aide de yarn.
- **COPY . .** : Copie l'ensemble des fichiers du projet dans le conteneur.
- **EXPOSE 3000** : Expose le port 3000, ce qui est utilisé par l'application Node.js pour écouter les requêtes entrantes.
- **ENV REDIS_URL=redis://redis:6379** : Déclare une variable d'environnement pour l'URL de Redis. Ici, Redis est référencé comme un service appelé redis dans le même réseau Docker.

- **CMD ["node", "main.js"]** : Lance l'application Node.js en exécutant le fichier main.js.

Ce fichier Dockerfile permet de construire une image Docker de l'application Node.js. Une fois l'image construite, on pourra l'utiliser pour déployer notre application dans Kubernetes.

Scaling automatique du serveur Node.js

Une fois l'image Docker de l'application Node.js créée et prête, il est important de déployer l'application dans Kubernetes et de configurer l'auto-scaling pour gérer automatiquement la montée en charge selon les besoins.

1. Déploiement avec Kubernetes : Pour déployer l'application dans Kubernetes, on crée un fichier de configuration YAML(node-redis-deployment.yml) qui décrit le déploiement de l'application Node.js, ainsi que la stratégie de scaling.

2. Application de la configuration : Une fois le fichier YAML créé, il suffit de l'appliquer au cluster Kubernetes avec la commande suivante :

kubectl apply -f node-redis-deployment.yml

3. Vérification du scaling : Pour observer le scaling en action, tu peux utiliser les commandes kubectl get pods et kubectl top pods pour vérifier les réplicas et la consommation de ressources.

Capture à inclure :

- Capture de la commande kubectl get pods montrant les réplicas de l'application Node.js avant et après le scaling.

Exemple de sortie de la commande kubectl get pods avant le scaling :

Exemple de sortie de la commande kubectl get pods après le scaling :

Dans cet exemple, un deuxieme pod a été ajouté pour répondre à l'augmentation de la charge (par exemple, l'augmentation de l'utilisation du CPU).

4. Mise en place de Redis et connexion avec l'application Node.js

Pour cette étape, nous avons déployé Redis dans le cluster Kubernetes et l'avons configuré pour qu'il soit accessible à l'application Node.js. Cette mise en place se fait en deux parties : le **déploiement de Redis** et la création d'un **service Redis** pour le rendre accessible.

4.1 Déploiement de Redis

Tout d'abord, nous avons créé un fichier de déploiement Kubernetes pour Redis, qui spécifie l'utilisation de l'image Docker officielle redis:7.2. Le déploiement est configuré avec une seule réplique et des ressources limitées (CPU et mémoire).

Dans ce fichier :

- Le déploiement crée un pod Redis avec une seule réplique (replicas: 1).
- Le conteneur utilise l'image redis:7.2 et expose le port 6379.
- Les ressources CPU sont demandées avec 100m de CPU minimum et une limite de 500m.

Pour appliquer ce fichier, nous avons utilisé la commande suivante :

```
kubectl apply -f redis-deployment.yml
```

Cela déploie Redis sur Kubernetes en suivant les spécifications données dans le fichier YAML.

4.2 Création du Service Redis

Ensuite, nous avons créé un service Kubernetes pour Redis afin qu'il soit accessible par d'autres pods dans le cluster. Le fichier redis-service.yml définit ce service comme suit :

Dans ce fichier :

- Le service utilise un sélecteur app: redis pour cibler le pod Redis déployé.
- Le port 6379 est exposé pour permettre aux autres applications du cluster de communiquer avec Redis.
- Le service est de type ClusterIP, ce qui signifie qu'il est uniquement accessible à l'intérieur du cluster Kubernetes.

Nous avons appliqué ce fichier avec la commande suivante :

```
kubectl apply -f redis-service.yml
```

Cela permet à l'application Node.js de se connecter à Redis en utilisant le nom du service redis sur le port 6379.

4.3 Connexion de l'application Node.js à Redis

Dans l'application Node.js, la connexion à Redis se fait à l'aide de l'URL définie dans le fichier Dockerfile de l'application, via la variable d'environnement REDIS_URL :

```
ENV REDIS_URL=redis://redis:6379
```

Cette URL permet à l'application Node.js de se connecter au service Redis dans Kubernetes, grâce au nom du service redis.

4.4 Vérification du déploiement Redis

Après avoir déployé Redis et le service, nous avons vérifié que tout fonctionnait correctement :

1. **Vérification des pods Redis** :

```
nabster@Ciscko:~/k8s-deployments$ kubectl get pods -l app=redis
NAME                     READY   STATUS    RESTARTS   AGE
redis-6b657b9897-wqv26   1/1     Running   0          4d10h
nabster@Ciscko:~/k8s-deployments$
```

Cela affiche les pods Redis et leur statut.

2. **Vérification du service Redis** :

```
nabster@Ciscko:~/k8s-deployments$ kubectl get svc redis
NAME    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AG
E
redis   ClusterIP   10.111.83.155   <none>        6379/TCP   4d
1h
```

Cela permet de vérifier que le service Redis est bien exposé sur le port 6379.

4.5 Test de la connexion entre Node.js et Redis

Enfin, pour tester la connexion entre l'application Node.js et Redis, nous avons observé les logs de l'application. Si tout fonctionne correctement, nous devrions voir les messages indiquant que les données ont été stockées et récupérées de Redis avec succès.

Pour afficher les logs de l'application, nous avons utilisé la commande :

```
nabster@Ciscko:~/k8s-deployments$ kubectl logs node-redis-999b9
f597-25kv6 --tail=20
Sun, 06 Apr 2025 00:54:37 GMT:  get item crv
Sun, 06 Apr 2025 00:54:37 GMT:  get item vas
Sun, 06 Apr 2025 00:54:37 GMT:  get item key
Sun, 06 Apr 2025 00:54:38 GMT:  get item crv
Sun, 06 Apr 2025 00:54:38 GMT:  get item key
Sun, 06 Apr 2025 00:54:39 GMT:  get item crv
Sun, 06 Apr 2025 00:54:39 GMT:  get item vas
Sun, 06 Apr 2025 00:54:39 GMT:  get item key
Sun, 06 Apr 2025 00:54:39 GMT:  get item crv
Sun, 06 Apr 2025 00:54:39 GMT:  get item vas
Sun, 06 Apr 2025 00:54:40 GMT:  get item crv
Sun, 06 Apr 2025 00:54:40 GMT:  get item key
Sun, 06 Apr 2025 00:54:41 GMT:  get item crv
Sun, 06 Apr 2025 00:54:41 GMT:  get item vas
Sun, 06 Apr 2025 00:54:42 GMT:  get item crv
Sun, 06 Apr 2025 00:54:42 GMT:  get item vas
Sun, 06 Apr 2025 00:54:42 GMT:  get item crv
Sun, 06 Apr 2025 00:54:42 GMT:  get item key
Sun, 06 Apr 2025 00:55:10 GMT:  post item projet_crv IAC
Sun, 06 Apr 2025 00:55:10 GMT:  get item projet_crv
```

On observe que la connexion est réussie

Cette étape permet à l'application Node.js de s'interfacer correctement avec Redis dans le cluster Kubernetes, garantissant ainsi que les données sont stockées et récupérées à partir du cache Redis.

5. Déploiement de l'application Node.js dans Kubernetes

Après avoir conteneurisé notre application Node.js et mis en place Redis dans le cluster, l'étape suivante consiste à déployer l'application dans Kubernetes à l'aide de fichiers YAML. Cela permet d'assurer que l'application est exécutée dans un pod, et qu'elle peut communiquer avec Redis via le service précédemment créé.

5.1 Fichier de déploiement : node-redis-deployment.yml

Ce fichier définit le déploiement de notre application Node.js dans le cluster :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-redis
  labels:
    app: node-redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-redis
  template:
    metadata:
      labels:
        app: node-redis
    spec:
      containers:
        - name: node-redis
          image: redis-node-server
          imagePullPolicy: Never
          ports:
            - containerPort: 8080
          env:
            - name: PORT
              value: '8080'
            - name: REDIS_URL
              value: redis://redis:6379
            - name: REDIS_REPLICAS_URL
              value: redis://redis-replicas:6379
```

```yaml
        resources:
          requests:
            cpu: "100m"  # Demande minimale de CPU (100 milliCPU = 0.1 CPU)
          limits:
            cpu: "500m"  # Limite maximale de CPU (500 milliCPU = 0.5 CPU)
```

Explications :

- Le déploiement lance un pod contenant l'image Docker redis-node-server, qui est une image locale (d'où imagePullPolicy: Never).
- L'application écoute sur le **port 8080**.
- Deux variables d'environnement sont définies pour accéder aux instances Redis primaires et réplicas.
- Des limites de ressources sont définies pour garantir une exécution contrôlée dans le cluster.

5.2 Fichier de service associé : node-redis-service.yml(déjà appliqué)

Le service qui expose cette application Node.js dans le cluster est déjà en place

```
nabster@Ciscko:~/k8s-deployments$ kubectl get svc node-redis
NAME         TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)
        AGE
node-redis   NodePort   10.107.21.136   <none>        8080:3141
8/TCP   5d5h
```

Cela signifie que :

- Le service est de type **NodePort**.
- Il expose le port **8080** du conteneur vers le port **31418** du nœud (accessible en externe).

5.3 Accès à l'application

L'application est accessible via le port 31418 du nœud Kubernetes.

Si tu es en local avec **Minikube**, utilise :

```
nabster@Ciscko:~/k8s-deployments$ minikube service node-redis
|-----------|------------|-------------|-----------------------
----|
| NAMESPACE |    NAME    | TARGET PORT |           URL
    |
|-----------|------------|-------------|-----------------------
----|
| default   | node-redis |        8080 | http://192.168.49.2:31
418 |
|-----------|------------|-------------|-----------------------
----|
🎉  Ouverture du service default/node-redis dans le navigateur
par défaut...
```

on voit dans le navigateur

6. Déploiement de l’application React (Frontend)

L’interface utilisateur de l’application a été développée avec **React.js**, puis conteneurisée et déployée dans Kubernetes sous forme d’un pod unique. L’objectif ici était de fournir une interface graphique permettant à l’utilisateur d’interagir avec le backend Node.js, qui lui-même dialogue avec Redis.

6.1 Construction de l’image Docker

L’application React a été construite avec yarn build ou npm run build, puis servie à l’aide de **Nginx**.

Dockerfile utilisé :

```
nabster@Ciscko:~/k8s-deployments$ cat /home/nabster/redis-react
/Dockerfile
# Utilise l'image officielle Nginx comme base
FROM nginx:alpine

# Copie les fichiers générés par `yarn build` dans le répertoir
e Nginx
COPY build/ /usr/share/nginx/html

# Expose le port 80
EXPOSE 80

# Démarre Nginx
CMD ["nginx", "-g", "daemon off;"]
```

⚠️ L’image a été nommée redis-react-frontend et utilisée localement (avec imagePullPolicy: Never), ce qui signifie qu’elle n’est **pas poussée sur un registre distant**.

6.2 Déploiement dans Kubernetes

L’application frontend est déployée à l’aide de deux fichiers : un **Deployment** et un **Service**.

frontend-deployment.yml :

```
nabster@Ciscko:~/k8s-deployments$ cat frontend-deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: redis-react-frontend
          imagePullPolicy: Never
          ports:
            - containerPort: 80
          env:
            - name: API_URL
              value: "http://node-redis:8080"
```

- On utilise ici une **variable d'environnement** API_URL pointant vers le backend Node.js (node-redis) pour que l'interface puisse faire ses requêtes API.
- Le déploiement est minimal avec **1 seule réplique**.

frontend-service.yml :

```
nabster@Ciscko:~/k8s-deployments$ cat frontend-service.yml
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: NodePort
```

- Ce service expose le frontend via un **NodePort**, ce qui permet d'y accéder depuis l'extérieur du cluster (via minikube service frontend par exemple).

6.3 Communication avec le backend

L'application React utilise la variable d'environnement API_URL définie dans le déploiement pour faire ses requêtes vers le serveur Node.js. Cette URL (http://node-redis:8080) est résolue via le DNS interne de Kubernetes.

Les appels API dans React sont réalisés via fetch().

- kubectl get pods montrant le pod frontend

```
nabster@Ciscko:~/k8s-deployments$ kubectl get pods
NAME                                READY   STATUS    RESTART
S   AGE
frontend-7bb7b6c566-96gdg           1/1     Running   1
    2d3h
```

- kubectl get service montrant l'exposition en NodePort

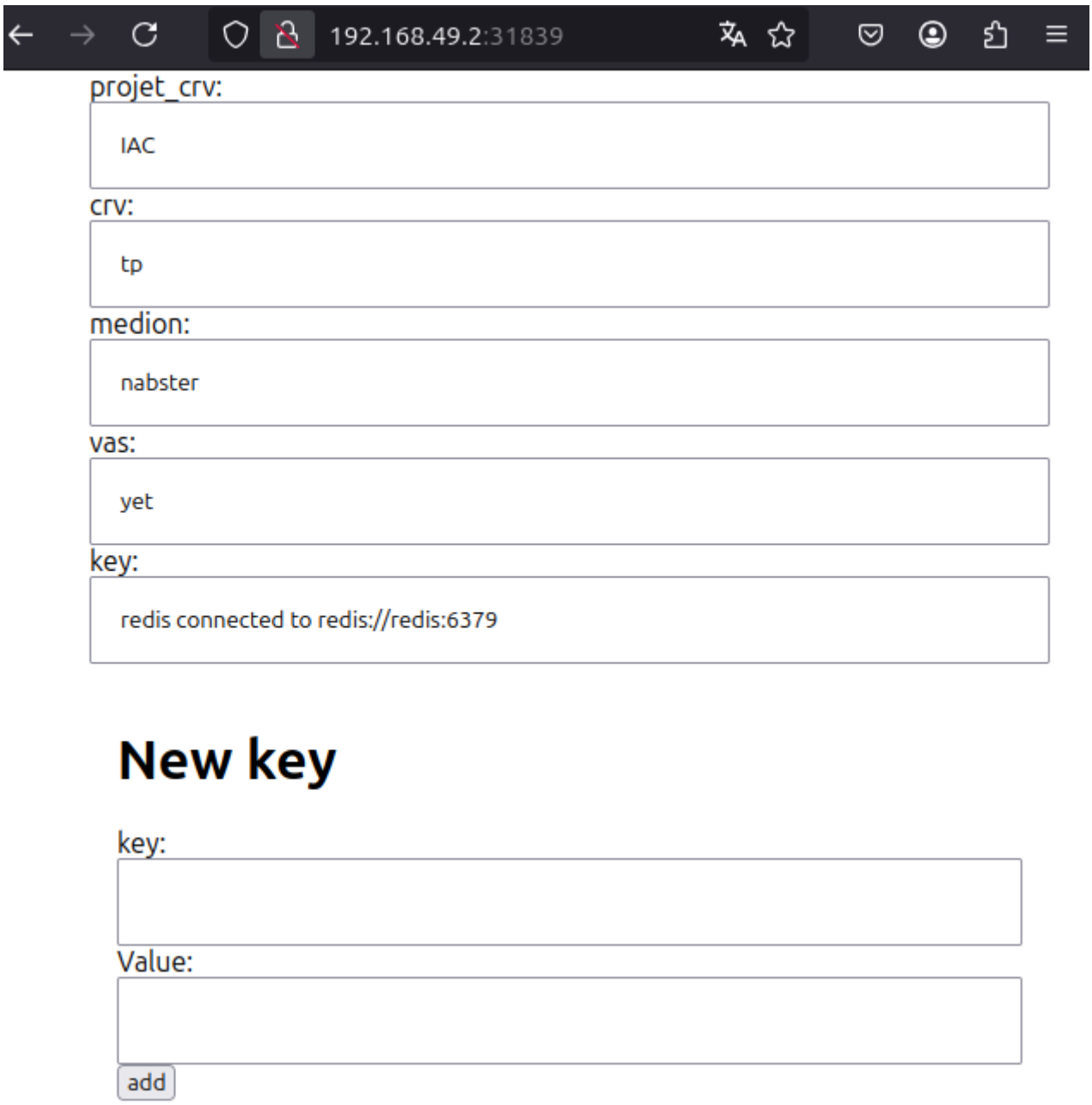
```
nabster@Ciscko:~/k8s-deployments$ kubectl get service
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)          AGE
frontend            NodePort    10.102.45.237   <none>
80:31839/TCP     5d22h
```

- kubectl describe pod frontend pour montrer les variables d'environnement

```
nabster@Ciscko:~/k8s-deployments$ kubectl describe pod frontend

Name:             frontend-7bb7b6c566-96gdg
Namespace:        default
Priority:         0
Service Account:  default
Node:             minikube/192.168.49.2
Start Time:       Fri, 04 Apr 2025 18:56:25 +0200
Labels:           app=frontend
                  pod-template-hash=7bb7b6c566
Annotations:      kubectl.kubernetes.io/restartedAt: 2025-04-04
T18:51:10+02:00
Status:           Running
IP:               10.244.0.161
IPs:
  IP:           10.244.0.161
Controlled By:  ReplicaSet/frontend-7bb7b6c566
Containers:
  frontend:
    Container ID:   docker://d95877742187e2dc70e291a7cb64122b99
04b8004e49f45790e37dec695e38f2
    Image:          redis-react-frontend
    Image ID:       docker://sha256:766b4442abf8154e83d9665620d
4990da14e486fabe341fbe7940294c3ddef7a
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
```

- La page React dans le navigateur



7. Supervision de l’Infrastructure avec Prometheus & Grafana

Pour surveiller en temps réel les performances de l’infrastructure Kubernetes, une stack de monitoring a été mise en place en déployant **manuellement** Prometheus et Grafana à partir de fichiers YAML personnalisés, sans passer par Helm.

7.1 Déploiement de Prometheus

Le fichier prometheus-deployment.yml contient :

- Un ConfigMap avec la configuration Prometheus
- Un Deployment pour le pod Prometheus
- Un Service pour l'exposer à l’intérieur du cluster

📄 **Configuration notable :**

```
    scrape_configs:
      - job_name: 'nodejs'
        static_configs:
          - targets: ['node-redis:8080']  # Ici on cible l'appl
ication Node.js
      - job_name: 'redis'
        static_configs:
          - targets: ['redis-exporter:9121']
      - job_name: 'kube-state-metrics'
        static_configs:
          - targets: ['kube-state-metrics:9091']

---
```

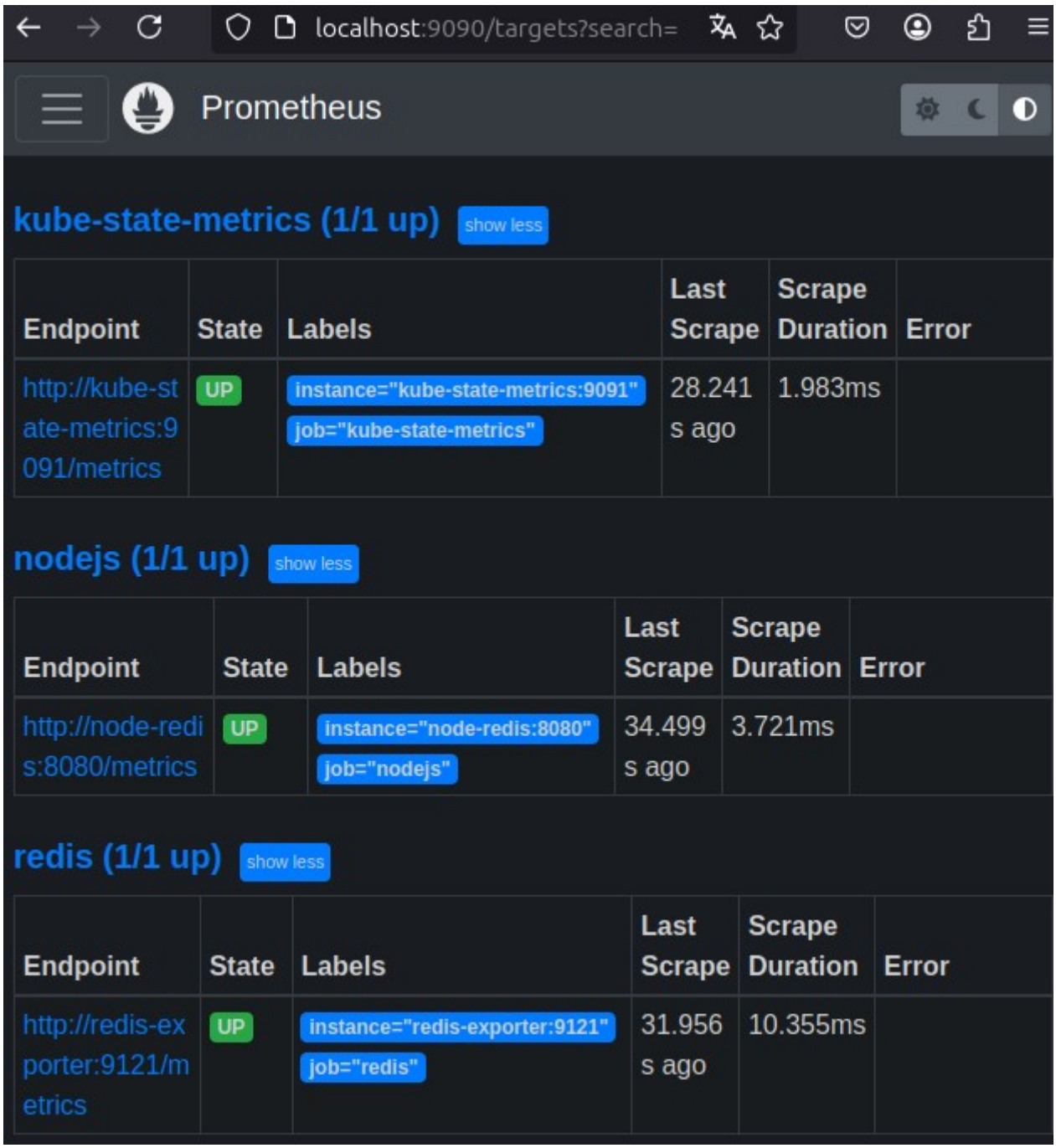
3 cibles surveillées :

- **nodejs** : l'application backend Node.js
- **redis-exporter** : exporte les métriques Redis au format Prometheus
- **kube-state-metrics** : expose l'état des objets Kubernetes (déployé via kube-state-metrics-deployment.yml)

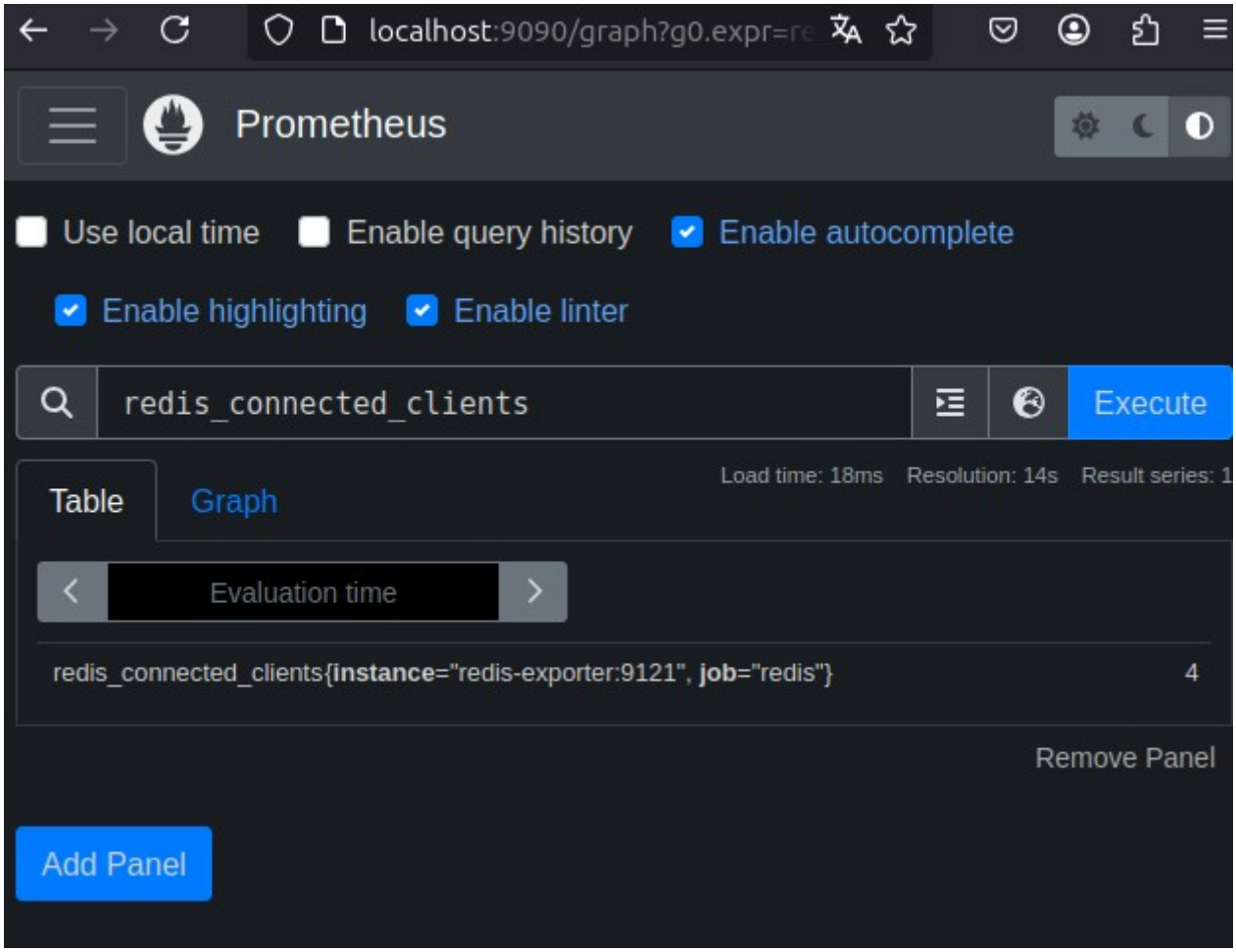
7.2 Permissions RBAC

Pour que Prometheus accède aux métriques du cluster, des droits RBAC ont été définis :

- Un ClusterRole (clusterrole.yml) avec des accès en lecture sur les ressources Kubernetes
- Un ClusterRoleBinding (clusterrolebinding.yml) lié au **ServiceAccount** default
- Le serviceaccount.yml indique que le default SA peut assumer ce rôle


localhost:9090/targets?search=
Prometheus
kube-state-metrics (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://kube-state-metrics:9091/metrics
UP
instance="kube-state-metrics:9091"
job="kube-state-metrics"
28.241 s ago
1.983ms
nodejs (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://node-redis:8080/metrics
UP
instance="node-redis:8080"
job="nodejs"
34.499 s ago
3.721ms
redis (1/1 up)
show less
Endpoint
State
Labels
Last Scrape
Scrape Duration
Error
http://redis-exporter:9121/metrics
UP
instance="redis-exporter:9121"
job="redis"
31.956 s ago
10.355ms

Exemple de visualisation de clients redis connectés



7.3 Déploiement de Grafana

Grafana est déployé via grafana-deployment.yml, avec :

- un Deployment standard
- une configuration automatique de la **source de données Prometheus** via un ConfigMap
- un Service de type NodePort pour l'accès externe

```
        env:
        - name: GF_SECURITY_ADMIN_PASSWORD
          value: "admin"
        volumeMounts:
        - name: grafana-config-volume
          mountPath: /etc/grafana/provisioning/datasources
```

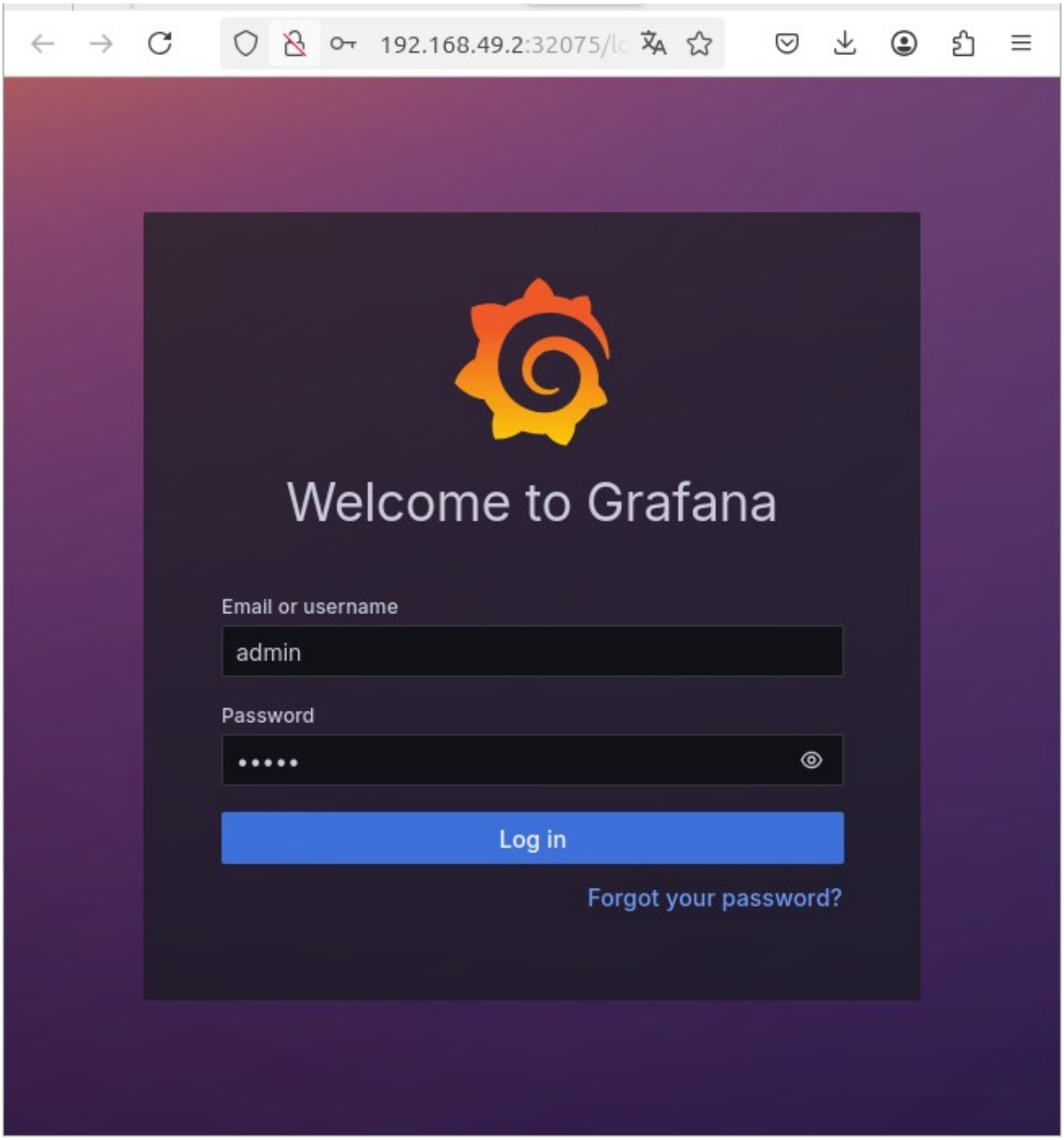
Identifiants d’accès :

- **username :** admin
- **password :** admin

Accès via :

minikube service grafana

on observe dans le navigateur



Ou en récupérant l’URL si tu utilises un autre cluster :

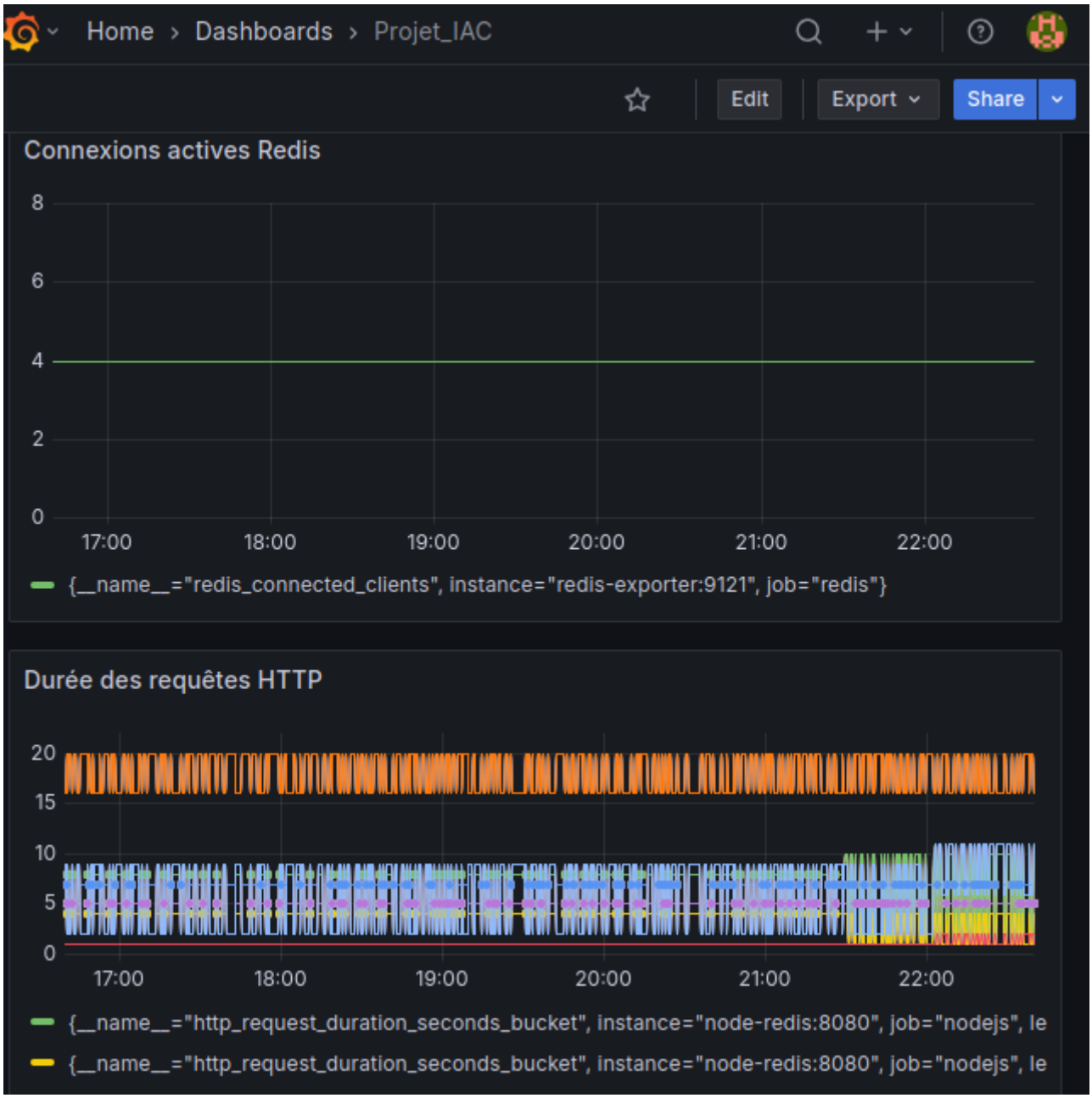
kubectl get svc grafana

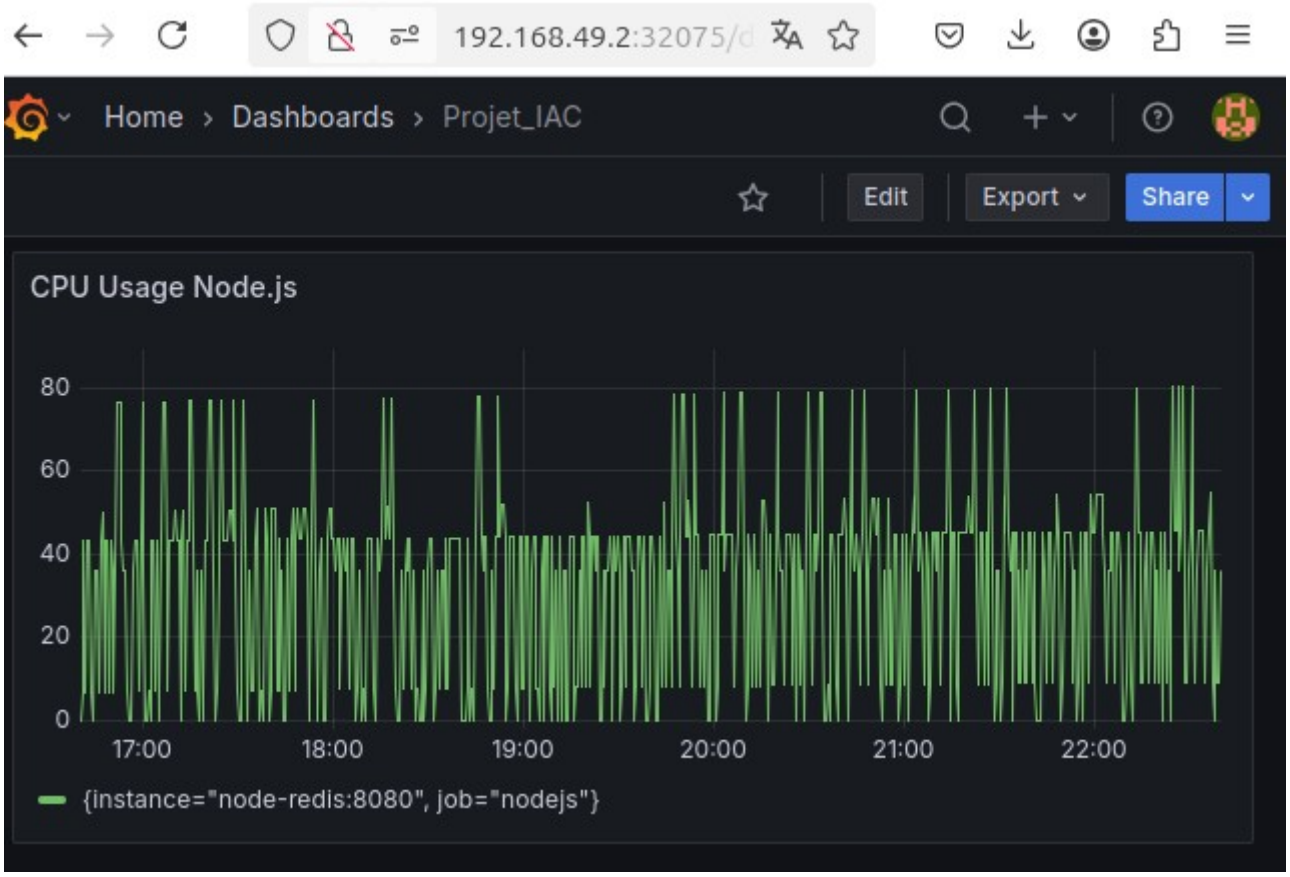
```
nabster@Ciscko:~/k8s-deployments$ kubectl get svc grafana
NAME      TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)
     AGE
grafana   NodePort   10.106.66.177   <none>        3000:32075/T
CP   4d8h
```

7.4 Visualisation et Dashboards

Une fois Grafana accessible, la source de données **Prometheus** est déjà configurée (via le ConfigMap grafana-datasources).
On peut importer des dashboards depuis Grafana.com ou créer les notres pour suivre :

- L'état des pods et des nœuds
- Les performances de node-redis
- Le comportement du cluster (via kube-state-metrics)
- Les statistiques Redis via le redis-exporter




192.168.49.2:32075/d
Home › Dashboards › Projet_IAC
Edit
Export
Share
CPU Usage Node.js
80
60
40
20
0
17:00
18:00
19:00
20:00
21:00
22:00
{instance="node-redis:8080", job="nodejs"}


Memory Usage Node.js
60000000
40000000
20000000
0
17:00
18:00
19:00
20:00
21:00
22:00
{instance="node-redis:8080", job="nodejs"}
{__name__="process_resident_memory_bytes", instance="node-redis:8080", job="nodejs"}

Bien sûr ! Voici l'étape sur l'autoscaling réécrite de façon impersonnelle avec un ton plus direct et naturel, tout en restant claire et concise.

8. Autoscaling : Horizontal Pod Autoscaler (HPA)

Objectif

L'objectif ici est de garantir que l'application backend puisse **monter en charge automatiquement** en fonction de la consommation CPU. Kubernetes ajuste dynamiquement le nombre de pods pour répondre à la demande.

8.1 Pré-requis

Avant d'activer l'HPA, il est nécessaire de :

- Déployer le **metrics-server** (fichier metrics-server.yaml).
- Vérifier que l'application **expose la consommation CPU** mesurable (via les ressources définies).

8.2 Déploiement de l'HPA

Fichier utilisé : hpa.yml

```
nabster@Ciscko:~/k8s-deployments$ cat hpa.yml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nodejs-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: node-redis
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 20
---
```

```
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: redis-replica-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: redis-replicas
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 20
```

Comportement :

Kubernetes va scaler le **Deployment** node-redis entre 2 et 5 pods pour maintenir une **utilisation CPU moyenne à 20%**.

8.3 Test de l'autoscaling

Un test de montée en charge peut être simulé avec un fichier comme hpa-test.yml, qui génère de la charge sur le backend.

Pour observer l'effet :

kubectl get hpa

```
nabster@Ciscko:~/k8s-deployments$ kubectl get hpa
NAME                REFERENCE                      TARGETS
  MINPODS   MAXPODS   REPLICAS   AGE
cpu-load-hpa        Deployment/cpu-load            cpu: 250%/80%
  1         10        10         2d22h
nodejs-hpa          Deployment/node-redis          cpu: 5%/20%
  2         5         2          5d23h
redis-replica-hpa   StatefulSet/redis-replicas     cpu: 5%/20%
  2         5         2          5d23h
```

Pour surveiller les pods :

watch kubectl get pods

Résultat attendu

- Si la charge CPU dépasse 20%, **de nouveaux pods** seront créés automatiquement.
- Lorsque la charge diminue, Kubernetes **réduira** le nombre de pods en conséquence.

L’étape d’autoscaling est configurée, il suffit maintenant de tester la charge et de vérifier le bon fonctionnement. Si besoin, des commandes **curl en boucle** peuvent être ajoutées pour générer plus de trafic.

Conclusion

Avec l'implémentation de l'**Horizontal Pod Autoscaler (HPA)**, l'objectif principal est atteint : **l'autoscaling** de l'application backend se fait dynamiquement en fonction de la consommation CPU. Nous avons mis en place les fichiers de configuration nécessaires (hpa.yml et hpa-test.yml) et validé leur efficacité via l'observation des ressources et la montée en charge. Kubernetes ajuste le nombre de pods entre 2 et 5 en fonction de la charge CPU pour maintenir un équilibre optimal.