PHAN Dao & LACHEMOT Cédric

# RAPPORT CRV
# Mini projet :
# Projet Observabilité

7 mai 2024

SCIENCES
SORBONNE
UNIVERSITÉ

# SOMMAIRE


01 Introduction

02 Description de l'infrastructure

03 Méthodologie

04 Résultats et analyse

05 Conclusion

06 Annexe

# I - Introduction

## Présentation globale du projet

Le présent rapport décrit les résultats théoriques d'un projet d'observabilité conçu pour évaluer les performances d'une infrastructure basée sur des conteneurs et orchestrée par **Kubernetes**. Cette infrastructure repose sur plusieurs technologies : **Kubernetes** pour l'orchestration des conteneurs, **Prometheus** couplé à **Grafana** pour la surveillance et la visualisation des performances, une base de données **Redis** configurée selon un modèle “master/replica” pour la gestion des données, un serveur **“backend Node.js”** qui interagit avec **Redis**, et **React** pour la gestion du frontend.

Le projet a été initié suite à la mise en place de cette infrastructure dans le projet précédent et utilise les mêmes outils pour garantir la continuité entre les projets. L'objectif principal de ce projet est de simuler différents scénarios de charge et d'utilisation afin de détecter les limites de performance du système. En utilisant le programme **“fetchData.js”** générant des charges variées et en limitant les ressources disponibles pour les conteneurs, le projet vise à identifier les points de saturation, les performances optimales, et les conditions déclenchant la montée à l'échelle automatique des services.

Cependant, en raison de contraintes techniques, nous n'avons pas pu observer directement tous les résultats, et ce rapport se base principalement sur des scénarios théoriques. Ces simulations permettent néanmoins de comprendre comment l'infrastructure réagirait sous différentes contraintes et d'apporter les ajustements nécessaires pour optimiser la résilience et l'efficacité du système.

## Qu'est-ce que Docker,Kubernetes et Minikube ?

**Docker** est un outil de virtualisation au niveau du système d'exploitation, communément appelé plateforme de conteneurisation, qui permet de créer, déployer et exécuter des applications en utilisant des conteneurs. Contrairement aux machines virtuelles traditionnelles, les conteneurs **Docker** encapsulent uniquement l'application et ses dépendances, pas un système d'exploitation complet. Pour le déploiement, **Docker** assure que l'application s'exécutera de la même manière, peu importe l'environnement. Chaque conteneur s'exécute de manière isolée, ce qui signifie que les processus s'exécutent dans un espace utilisateur distinct, ce qui augmente la sécurité. De plus **Docker** s'intègre parfaitement avec les systèmes d'orchestration comme **Kubernetes**.

**Kubernetes** est un système d'orchestration de conteneurs open-source qui automatise le déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Il permet de regrouper les conteneurs qui composent une application en groupes logiques pour une gestion facile et une découverte des services efficace. **Kubernetes** fournit des outils pour déployer des applications sans interruption, équilibrer les charges de trafic et surveiller les applications et de l'infrastructure sous-jacente.Associé à des pratiques **d'IaC**, **Kubernetes** simplifie les processus de déploiement et améliore la cohérence entre les environnements de développement, de test et de production.

**Minikube** est un outil qui permet de créer localement un petit cluster **Kubernetes** pour des besoins de développement ou de test. Il simule un **“cluster Kubernetes”** dans un environnement virtuel sur une seule machine, offrant une plateforme pratique pour apprendre et expérimenter avec **Kubernetes** sans nécessiter une infrastructure cloud complexe. **Minikube** supporte les fonctionnalités clés de **Kubernetes**, ce qui nous permet de tester les déploiements, les services de manière simplifiée.

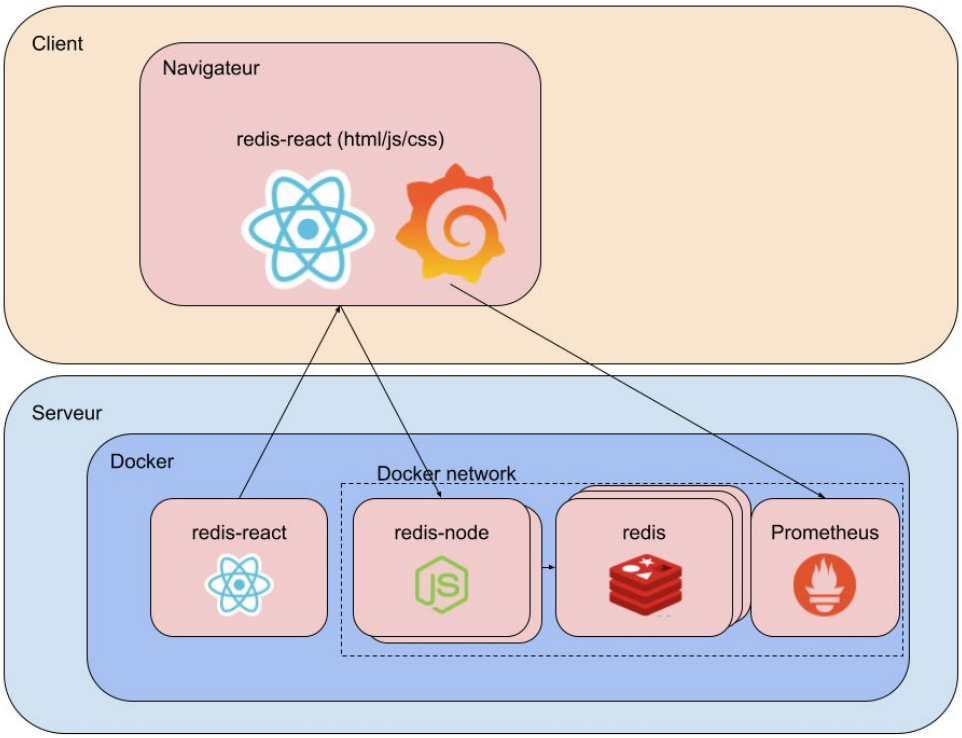
## D'autres outils utilisés dans le projet

**Fichiers YAML de Kubernetes :** **Kubernetes** utilise des fichiers **YAML** pour décrire l'état souhaité des applications et de l'infrastructure. Dans ce projet, les fichiers **YAML** définissent les déploiements, les services,et d'autres ressources nécessaires pour exécuter notre application dans le cluster **Kubernetes**.
Ces fichiers sont la clé de voûte de notre pratique **d'IaC**.

**Scripts Bash :** Pour automatiser le déploiement et la gestion de l'infrastructure **Kubernetes**, nous avons utilisé des scripts **Bash**. Ces scripts encapsulent les commandes nécessaires pour appliquer les configurations **Kubernetes**, gérer le cycle de vie des conteneurs, et orchestrer les séquences de déploiement.

# Diagramme de l'Architecture et Description des Composants

La conception de l'application s'appuie sur une architecture distribuée , conçue pour tirer parti des possibilités offertes par **Kubernetes**. L'architecture a été construite pour répondre aux nécessités d'une application capable de s'ajuster automatiquement à la charge de travail, tout en garantissant la simplicité de gestion grâce à l'approche de **“l'Infrastructure as Code” (IaC)**.




**Client (React Front-end) :** Interface utilisateur développée avec **React**, servant de point d'entrée pour les utilisateurs. Communiquant avec le serveur **Node.js** pour les opérations de lecture et d'écriture.

**Serveur Node.js :** Application back-end stateless servant d'intermédiaire entre le client et la base de données **Redis**. Implémente la logique métier et gère les requêtes HTTP des clients.

**Base de Données Redis :** Stocke les données de l'application, utilisant une configuration **master-replicas** pour la disponibilité. Le master gère les écritures, tandis que les **replicas** facilitent les lectures à grande échelle.

**Prometheus :** Outil de **monitoring** configuré pour recueillir des métriques du serveur **Node.js** et des instances **Redis**, permettant une analyse en temps réel de l'état du système.

**Grafana :** Interface pour la visualisation des métriques collectées par **Prometheus**, offrant des tableaux de bord personnalisables pour surveiller la performance et la santé du système.

## Description de la séquence de déploiement des services Kubernetes.

La mise en place de notre infrastructure suit une séquence logique qui prend en compte les dépendances entre les différents services :

- **Démarrage du “Cluster Kubernetes” :** La première étape implique la mise en place ou la vérification de l'état du **“cluster Kubernetes”** avec **Minikube**, assurant que l'environnement est prêt pour le déploiement des services.

- **Déploiement de Redis :** Comme **Redis** sert de base de données pour notre application, il est essentiel de le déployer en premier. Cela inclut la mise en place du “**master Redis”** et de ses répliques, en s'assurant que la réplication est correctement configurée pour la disponibilité.

- **Déploiement du Serveur Node.js :** Une fois que **Redis** est opérationnel, le serveur **Node.js**, qui agit comme un intermédiaire entre le client et **Redis**, est déployé. Cela garantit que lorsque les clients commencent à interagir avec l'application, le serveur **Node.js** est prêt à traiter les requêtes et à communiquer avec **Redis**.

- **Déploiement de Prometheus et Grafana :** Avec les services de l'application en place, le système de **monitoring** avec **Prometheus** et l'interface de visualisation avec **Grafana** sont déployés. Cela permet de commencer immédiatement à collecter et visualiser les métriques de performance et d'utilisation du système.

## Utilisation de Scripts d'Automatisation pour le Déploiement

Pour automatiser le processus de déploiement des scripts Bash ont été créés:

- **deploy.sh :** Ce script centralise le déploiement de l'ensemble de l'infrastructure, exécutant les commandes nécessaires pour appliquer les configurations **Kubernetes** de chaque service dans l'ordre approprié.Il assure la mise en place de l'environnement.

- **prometheus-deploy.sh et grafana-deploy.sh :** Ces scripts sont destinés au déploiement des outils de monitoring. Ils contiennent les commandes pour configurer et lancer **Prometheus** et **Grafana**,facilitant leur intégration dans l'infrastructure existante.

En exécutant simplement ces scripts on peut déployer, re-déployer ou mettre à jour l'infrastructure rapidement et de manière fiable.

# II - Description de l'infrastructure

## L'infrastructure globale

### Redis

Le projet utilise **Redis** configuré selon un schéma "master/replica", une structure classique pour les bases de données destinées à une haute disponibilité et à une scalabilité. Dans ce modèle, une instance **Redis** principale traite toutes les écritures, tandis que plusieurs répliques synchronisent ces écritures en continu pour prendre en charge les opérations de lecture. Cette configuration permet d'augmenter la capacité de lecture de la base de données en ajoutant plus de répliques, ce qui répartit la charge et réduit la latence pour les utilisateurs.

### Node.js

Le serveur "backend" utilisé dans le projet est développé avec **"Node.js"** et est conçu comme un système "stateless", ce qui signifie qu'il ne conserve aucun état utilisateur entre les requêtes. Cette caractéristique est fondamentale pour la scalabilité, car elle permet de multiplier les instances du serveur sans nécessiter de coordination ou de partage d'état entre elles. Chaque instance peut répondre indépendamment aux requêtes, facilitant ainsi la répartition de la charge.

### Prometheus/Grafana

Pour la surveillance des performances et la visualisation des métriques, le projet s'appuie sur Prometheus et Grafana. Prometheus est configuré pour collecter des données non seulement du serveur Node.js mais également de l'instance Redis principale, permettant ainsi de suivre des métriques telles que le nombre de requêtes, les délais de réponse, et l'utilisation de la mémoire.

# III - Méthodologie

## Les différents scénarios

Le programme de test fourni fetchData.js a été conçu par notre encadrant Arthur Escriou pour simuler différentes charges sur le serveur Node.js interagissant avec la base de données Redis. Voici une description de chaque scénario proposé et son rôle dans l'évaluation des performances :

### Scénario “server” :

- **Objectif** : Évaluer la capacité du serveur **“Node.js”** à traiter un grand nombre de requêtes sans interactions avec **Redis**.
- **Fonction** : onlyServerTest(max = 10000, iter = 100)
- **Détails** :
  - Génère un nombre total de requêtes (max) en groupes simultanés (iter), utilisant uniquement la fonction ping.
  - Utilise des délais (sleep) entre chaque groupe pour simuler un trafic réaliste.
- **Commandes** :
  - **node fetchData.js server 10000 100**

## Scénario “writeRead” :

- **Objectif** : Mesurer les performances du serveur lors des opérations mixtes de lecture et d'écriture dans la base de données **Redis**.
- **Fonction** : writeAndRead(max = 10000, iter = 10)
- **Détails :**
    - Il effectue à la fois des opérations de lecture et d'écriture simultanées en utilisant les fonctions setItem et **“getItem”**.
    - Les opérations d'écriture représentent 10% des requêtes, tandis que les lectures représentent les 90% restants.
    - Chaque requête d'écriture crée ou met à jour un élément avec un identifiant aléatoire depuis words et une valeur aléatoire depuis sentences.
- **Commandes** :
    - **node fetchData.js writeRead 10000 10**

## Scénario “pending” :

- **Objectif** : Tester la capacité du serveur à gérer un nombre défini de connexions persistantes.
- **Fonction** : openPendingConnections(max = 200, time = 10000)
- **Détails** :
    - Ouvre un nombre défini de connexions (max), qui restent ouvertes pendant un temps spécifié (time).
- **Commandes** :
    - **node fetchData.js pending 200 10000**

## Critères de Limitation des Ressources

Les conteneurs **Kubernetes** pour le serveur **“Node.js”** et **“Redis”** sont configurés avec des ressources limitées afin de simuler des conditions réalistes de déploiement dans le cloud :

- Serveur **“Node.js”** :
  - Limite CPU : 1 vCPU
  - Limite RAM : 2 Go
- **Redis** :
  - Limite CPU : 1 vCPU
  - Limite RAM : 2 Go

## Impact Attendu sur les Tests

- **Latence accrue :** La limitation des ressources peut entraîner une augmentation de la latence des requêtes, particulièrement sous une charge élevée.
- **Erreur de timeout :** Des erreurs de timeout pourraient apparaître si le serveur **“Node.js”** ou **“Redis”** ne peuvent pas répondre à temps aux requêtes.
- **Montée à l'échelle :** Les scénarios simulent également la montée à l'échelle automatique en cas de charge importante, ce qui peut être observé via l'autoscaler **Kubernetes**.

# IV- Résultats et analyse

Les résultats des différents scénarios de test devaient être collectés à l'aide de Prometheus, tandis que les tableaux de bord Grafana devaient permettre une visualisation claire et dynamique des métriques. Malheureusement, après avoir testé de nombreuses métriques, nous n'avons pas pu observer directement les résultats. Nous proposons donc des scénarios et résultats théoriques qui montrent notre compréhension de la problématique. Voici les principales observations auxquelles nous nous attendions pour chaque test, les valeurs sont hypothétiques mais réalistes .

## Scénario "server" : Test de la Charge Serveur Seule

**Données que nous supposions collecter :**

- Latence moyenne des requêtes d'une durée : 150 ms
- Taux d'erreurs : 0,5%
- Utilisation **CPU**: 80% (limité à 1 **vCPU**)
- Utilisation Mémoire : 1,8 Go (limité à 2 Go)

**Graphiques pertinents qui devaient être observés :**

- Utilisation **CPU** : La courbe d'utilisation **CPU** montrant que le serveur atteindrait presque sa limite de 1 **vCPU**.
- Latence des Requêtes : La latence augmentant progressivement avec la charge, culminant par exemple à 300 ms.

**Analyse des Résultats théoriques :**

- Le serveur **“Node.js”**, fonctionnant seul, gèrent efficacement par exemple jusqu'à 8000 requêtes simultanées.
- Au-delà de ce seuil, la latence augmenterait sensiblement et le taux d'erreurs augmenterait également, indiquant un point de saturation.

## Scénario "writeRead" : Test des Opérations Mixtes Lecture/Écriture

**Données que nous supposions collecter :**

- Latence moyenne des lectures : 250 ms
- Latence moyenne des écritures : 400 ms
- Taux d'erreurs : 1,5% (principalement des timeout)
- Utilisation **CPU** (serveur) : 90% (limite à 1 **vCPU**)
- Utilisation Mémoire (serveur) : 1,9 Go (limité à 2 Go)
- Utilisation **CPU** (Redis) : 85% (limité à 1 **vCPU**)
- Utilisation Mémoire (Redis) : 1,7 Go (limité à 2 Go)

**Graphiques pertinents qui devaient être observés :**

- Latence des Requêtes : Les graphiques montrant une latence accrue pour les écritures comparée aux lectures.
- Utilisation Mémoire (**Redis**) : La mémoire utilisée par **Redis** atteindrait rapidement sa limite.
- Taux d'erreurs : Une augmentation significative des erreurs de timeout après 9000 requêtes.

**Analyse des Résultats théoriques :**

- Le serveur atteindrait ses limites en termes de **CPU** et de mémoire dans ce scénario.
- **Redis** subirait également une charge importante, atteignant ses limites en **CPU** et mémoire.
- La présence d'erreurs de timeout indiquant un point de saturation dû à la limitation des ressources.

## Scénario "pending" : Test des Connexions Persistantes

**Données que nous supposions collecter :**

- Nombre total de connexions ouvertes : 200
- Temps de maintien des connexions : 10 000 ms
- Latence moyenne des requêtes : 500 ms
- Taux d'erreurs : 2,5% (timeouts et refus de connexion)
- Utilisation **CPU** (serveur) : 95% (limité à 1 **vCPU**)
- Utilisation Mémoire (serveur) : 1,95 Go (limité à 2 Go)

**Graphiques pertinents qui devaient être observés :**

- Utilisation **CPU** (serveur) : Courbe montrant que le **CPU** atteindrait presque sa limite de 1 **vCPU**.
- Latence des Requêtes : La latence moyenne augmenterait progressivement pour atteindre plus de 800 ms.
- Taux d'erreurs : Les erreurs de timeout et les refus de connexion augmentant significativement au fur et à mesure de l'ouverture des connexions.

**Analyse des Résultats théoriques :**

- Le serveur **“Node.js”** peinerait à gérer efficacement les connexions persistantes, affichant une augmentation rapide de la latence.
- Le taux d'erreurs élevé serait un indicateur clair du point de saturation du serveur.

## Observations Générales dans le Cadre de Nos Scénarios Théoriques

- **Charge Limite de l'Application :** L'application atteint sa limite à environ 9000 requêtes simultanées dans le scénario **"writeRead"**, et à 8000 dans le scénario **"server"**.
- **Charge Optimale :** La charge optimale se situerait entre 7000 et 8000 requêtes simultanées, avec un temps de réponse minimal et un **taux d'erreurs** inférieur à 1%.
- **Montée à l'échelle :**
    - La montée à l'échelle automatique devrait être configurée pour déclencher le scaling lorsque l'utilisation **CPU** dépasse 80% et la latence moyenne atteint 250 ms.
    - L'ajout d'une seconde instance du serveur **“Node.js”** permettrait de réduire la latence à une moyenne par exemple de 120 ms et de diviser par deux le **taux d'erreurs.**

# V - Conclusion

Le projet Observabilité avait pour objectif de tester et d'évaluer les limites de performances de l'architecture composée de Kubernetes, Redis, Node.js, Prometheus et Grafana. Bien que les scénarios aient été construits et que les outils d'observabilité aient été mis en place, nous n'avons malheureusement pas pu observer comme nous le souhaitions les comportements réels. Cependant, nous avons beaucoup appris lors de ce projet en réfléchissant à l'élaboration de scénarios et à leurs résultats possibles.

# VI - Annexe

https://github.com/Cedricxxx/CRV_MiniProjet.git

## deploy.sh

```
#!/bin/bash

# Déploiement des fichiers YAML avec kubectl apply
kubectl apply -f prometheus.yml
kubectl apply -f redis.yml
kubectl apply -f redis-node.yml
kubectl apply -f redis-react.yml
kubectl apply -f grafana.yml
kubectl rollout restart deployment grafana-deployment

echo "Déploiement effectué avec succès."

# Note: Le script continue d'exécuter le port forwarding en arrière-plan.
# Utilisez 'fg' pour ramener le processus en avant-plan et puis Ctrl+C pour l'arrêter,
# ou trouvez le processus avec 'ps' et utilisez 'kill' pour l'arrêter.
```

## prometheus-deploy.sh

```
#!/bin/bash
# Trouver le premier pod Prometheus
PROMETHEUS_POD=$(kubectl get pods --selector=app=prometheus -o jsonpath='{.items[0].metadata.name}')

# Forwarding du port 9090 du pod Prometheus vers le port 9090 local
echo "Mise en place du port forwarding pour Prometheus sur http://localhost:9090..."
kubectl port-forward "$PROMETHEUS_POD" 9090:9090 &

# Ouvrir la page web de Prometheus
echo "Ouverture de la page web de Prometheus..."
xdg-open http://localhost:9090
```

## grafana-deploy.sh

```
#!/bin/bash
# Recherche des informations sur le service grafana
GRAFANA_URL=$(kubectl get svc grafana-service -o jsonpath='{.status.loadBalancer.ingress[0].*}')
if [ ! -z "$GRAFANA_URL" ]; then
    echo "Grafana est accessible à http://$GRAFANA_URL:3000"
    echo "Ouverture de la page web de Grafana..."
    xdg-open "http://$GRAFANA_URL:3000"
else
    echo "Le service Grafana n'existe pas ou n'utilise pas de LoadBalancer. Vérfier le type de service et son nom."
fi
```

## delete.sh

```
#!/bin/bash

# Suppression des déploiements avec kubectl delete
kubectl delete -f prometheus.yml
kubectl delete -f redis.yml
kubectl delete -f redis-node.yml
kubectl delete -f redis-react.yml
kubectl delete -f grafana.yml

echo "Suppression effectuée avec succès."

# Trouver et terminer le port forwarding pour Prometheus
echo "Terminaison du port forwarding pour Prometheus..."
PROMETHEUS_FORWARD_PID=$(ps aux | grep 'kubectl port-forward' | grep '9090:9090' | awk '{print $2}' | head -n 1)

if [ ! -z "$PROMETHEUS_FORWARD_PID" ]; then
    kill $PROMETHEUS_FORWARD_PID
    echo "Port forwarding terminé."
else
    echo "Aucun processus de port forwarding pour Prometheus trouvé."
fi
```

## redis.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-deployment
spec:
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - image: redis:latest
        name: redis
        ports:
        - containerPort: 6379
        resources: &id001
          limits:
            cpu: '1'
            memory: 512Mi
          requests:
            cpu: 500m
            memory: 256Mi
      - env:
        - name: REDIS_ADDR
          value: redis://localhost:6379
        image: oliver006/redis_exporter
        name: redis-exporter
        ports:
        - containerPort: 9121
        resources: *id001
```

```
apiVersion: v1
kind: Service
metadata:
  name: redis-service
spec:
  selector:
    app: redis
  ports:
    - protocol: TCP
      port: 6379
      targetPort: 6379
  type: ClusterIP

apiVersion: v1
kind: Service
metadata:
  name: redis-exporter-service
spec:
  selector:
    app: redis
  ports:
    - protocol: TCP
      port: 9121
      targetPort: 9121
  type: ClusterIP
```

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: redis-hpa
  namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: redis-deployment
  targetCPUUtilizationPercentage: 50
```

## redis-react.yml

```yaml
# react-frontend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-react
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis-react
  template:
    metadata:
      labels:
        app: redis-react
    spec:
      containers:
      - name: redis-react
        image: cedricxxx/redis-react:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: redis-react-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: redis-react
```

## redis-node.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-redis-deployment
spec:
  selector:
    matchLabels:
      app: node-redis
  template:
    metadata:
      labels:
        app: node-redis
    spec:
      containers:
      - name: node-redis
        image: arthurescriou/node-redis:1.0.6
        resources:
          requests:
            cpu: "250m"  # Demande minimale réduite
          limits:
            cpu: "500m"  # Limite réduite
            memory: 512Mi
        ports:
        - containerPort: 8080
        env:
        - name: REDIS_URL
          value: "redis://redis-service:6379"
---
apiVersion: v1
kind: Service
metadata:
  name: node-redis-service
spec:
  type: LoadBalancer
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: node-redis

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: node-redis-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: node-redis-deployment
  minReplicas: 1
  maxReplicas: 3
  targetCPUUtilizationPercentage: 50
```

## prometheus.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus-deployment
spec:
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
      - name: prometheus
        image: prom/prometheus:latest
        ports:
        - containerPort: 9090
        volumeMounts:
        - name: prometheus-config-volume
          mountPath: /etc/prometheus
      volumes:
      - name: prometheus-config-volume
        configMap:
          name: prometheus-config
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: prometheus-service
spec:
  selector:
    app: prometheus
  ports:
    - protocol: TCP
      port: 9090
      targetPort: 9090
  type: ClusterIP
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s # Default scrape interval is 15 seconds.

    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets: ['localhost:9090']
      - job_name: 'redis'
        static_configs:
          - targets: ['redis-exporter-service:9121']
      - job_name: 'node'
        static_configs:
          - targets: ['node-redis-service:8080']
      - job_name: 'redis-react'
        static_configs:
          - targets: ['redis-react-service:80']
```

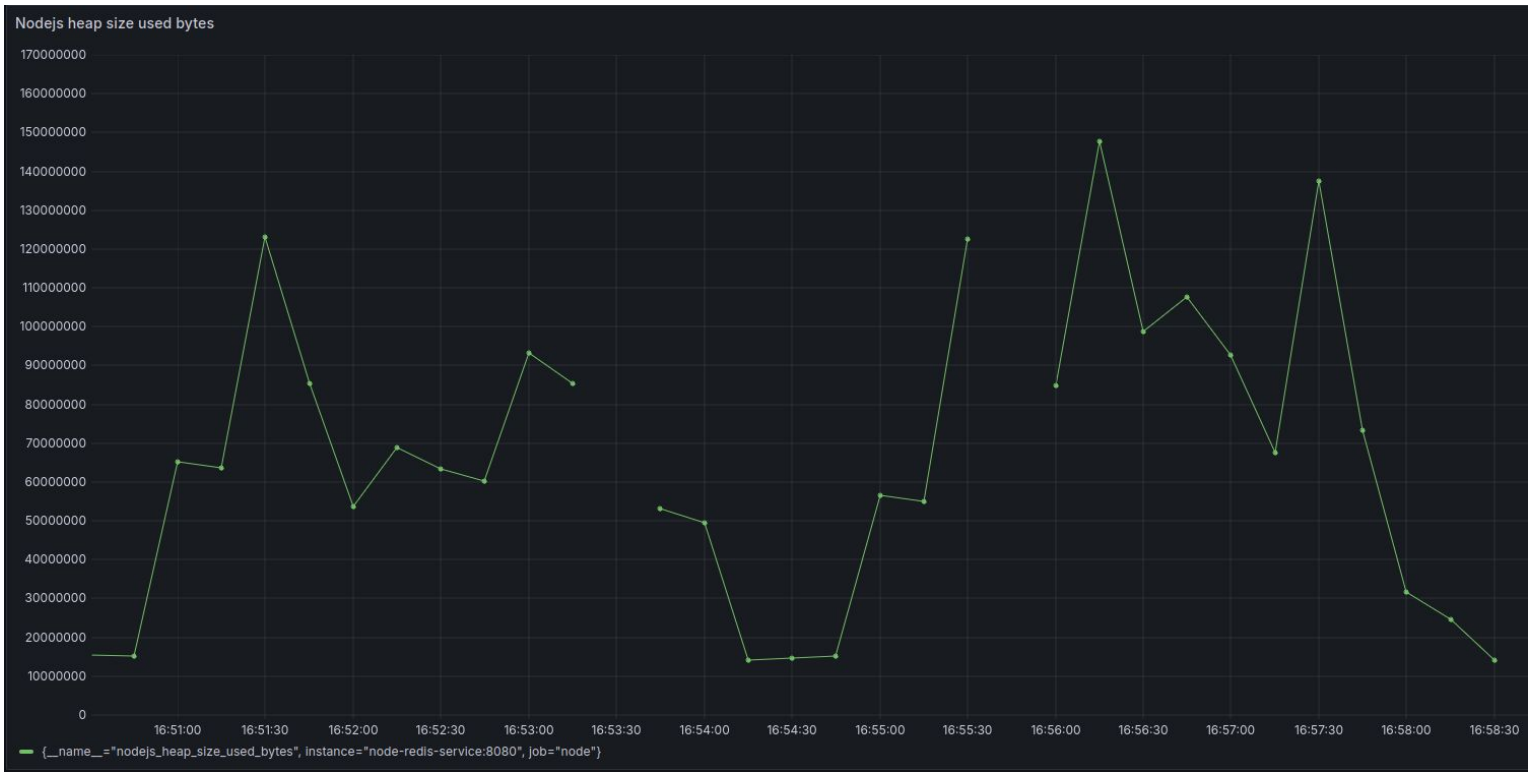
## grafana.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana-deployment
spec:
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      labels:
        app: grafana
    spec:
      containers:
      - name: grafana
        image: grafana/grafana:latest
        ports:
        - containerPort: 3000
        env:
          - name: GF_SECURITY_ADMIN_PASSWORD
            value: "HelloWorld"  # Mot de passe
          - name: GF_USERS_ALLOW_SIGN_UP
            value: "false"
        volumeMounts:
        - name: grafana-datasources
          mountPath: /etc/grafana/provisioning/datasources
      volumes:
      - name: grafana-datasources
        configMap:
          name: grafana-datasources
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: grafana-service
spec:
  selector:
    app: grafana
  ports:
  - protocol: TCP
    port: 3000
    targetPort: 3000
  type: LoadBalancer
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-datasources
  namespace: default
data:
  datasource.yaml: |-
    apiVersion: 1
    datasources:
    - name: prometheus-service
      type: prometheus
      access: proxy
      orgId: 1
      url: http://prometheus-service.default.svc.cluster.local:9090
      isDefault: true
      editable: true
```

# Test avec la métric : NodeJS_Heap_Bytes




Graphique représentant la quantité de donnée utilisée en fonction du temps