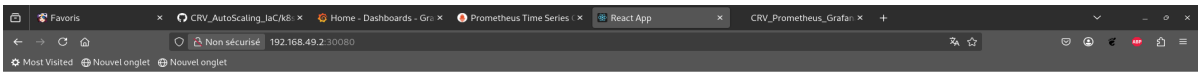
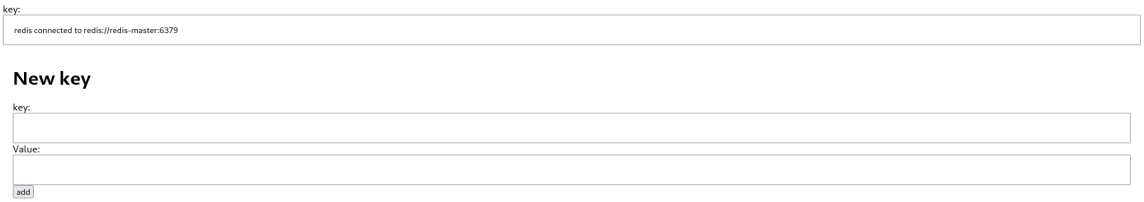
Emilie Lin 21105099
Maximilien Piron-Palliser 21107603

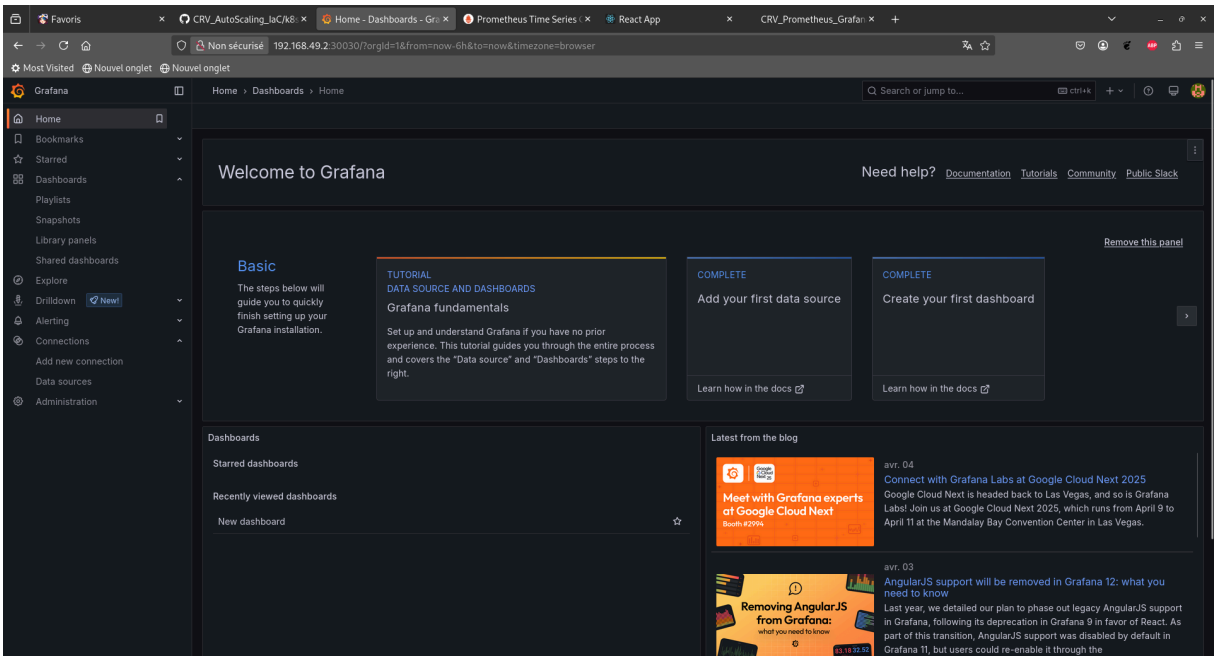
Rapport : AutoScaling et IaC

Pour réaliser ce mini projet nous avons utilisé plusieurs technologies différentes. Nous avons d'abord posé les bases du projet avec *Docker*, une plateforme qui permet de créer, empaqueter et exécuter des applications dans des conteneurs. Ensuite, pour gérer plusieurs services et assurer leur coordination, nous avons utilisé *Kubernetes,* une plateforme d'orchestration qui permet de déployer et de gérer des conteneurs sur un cluster de machines. Elle garantit notamment la montée en charge, la résilience et une gestion simplifiée de l'infrastructure. Enfin nous avons découvert les services de *Prometheus* et *Grafana* qui sont respectivement un outil de monitoring qui collecte et stocke des métriques sous forme de séries temporelles (Requêtes en *PromQL*) et une interface de visualisation qui permet d'afficher ces données sous forme de tableaux de bord interactifs.

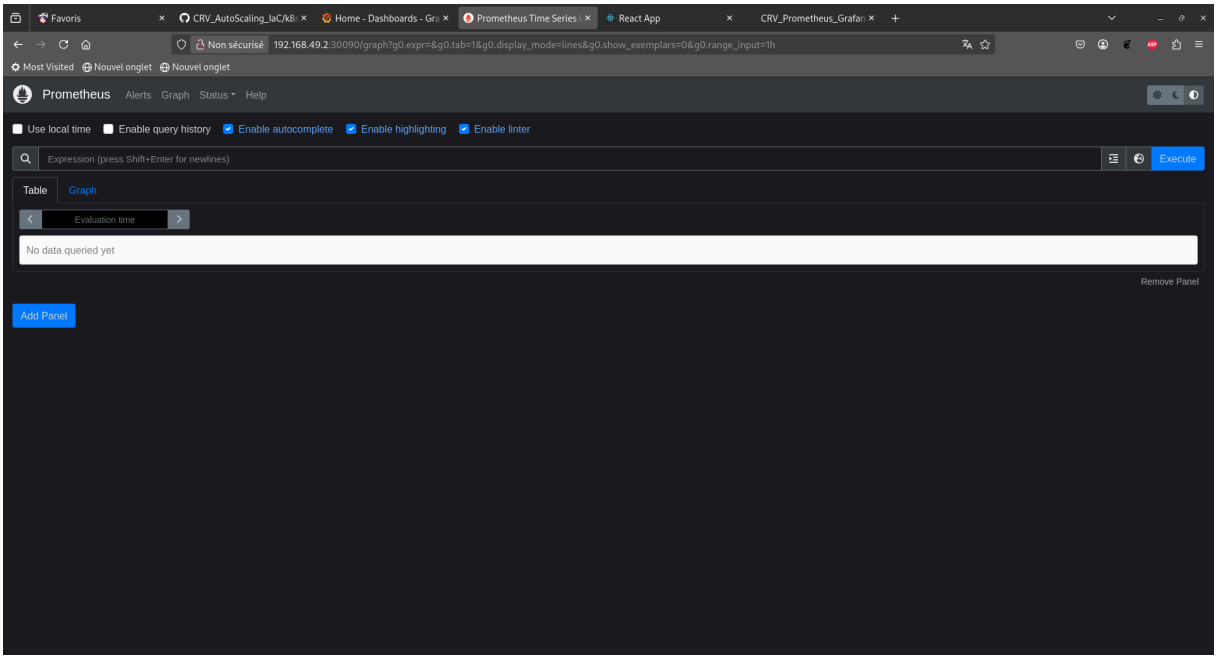
L'infrastructure du projet est découpée en deux parties, la partie *Client* et la partie *Serveur* : La partie client regroupe le frontend *React* (redis-react) ainsi que *Grafana*.





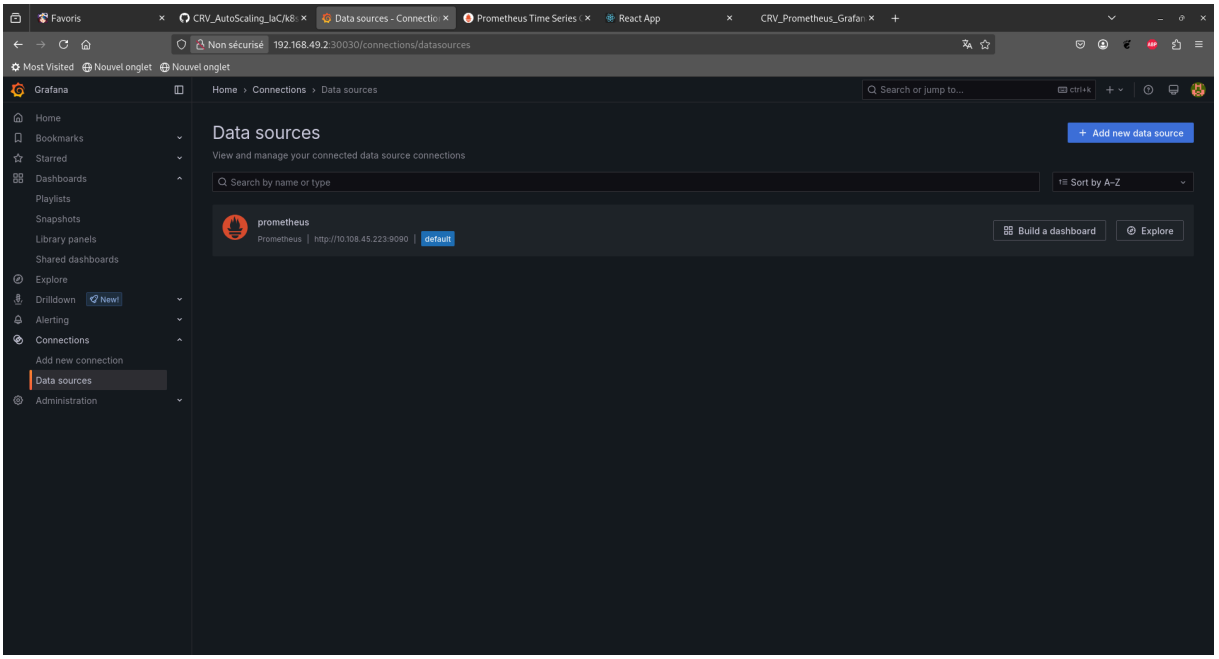


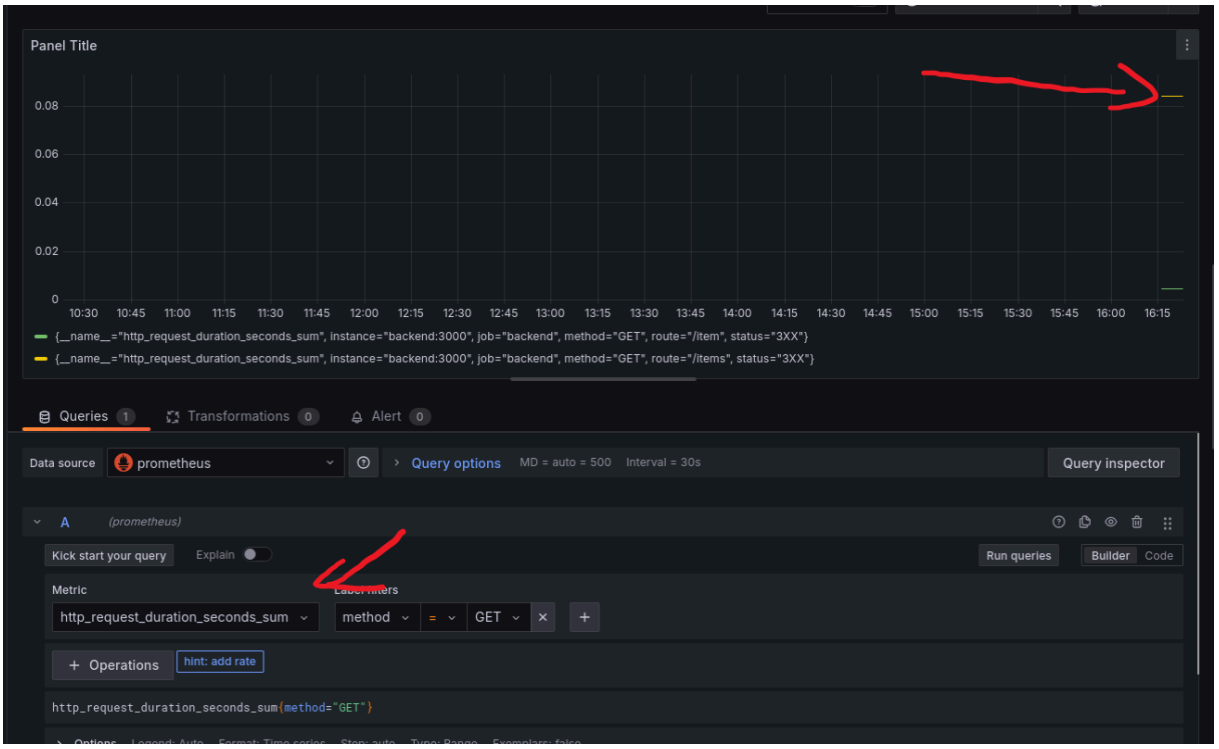
La partie serveur comprend le backend *Node.js* (redis-node), la base de données *Redis* (et de potentiels réplicas), et *Prometheus*.

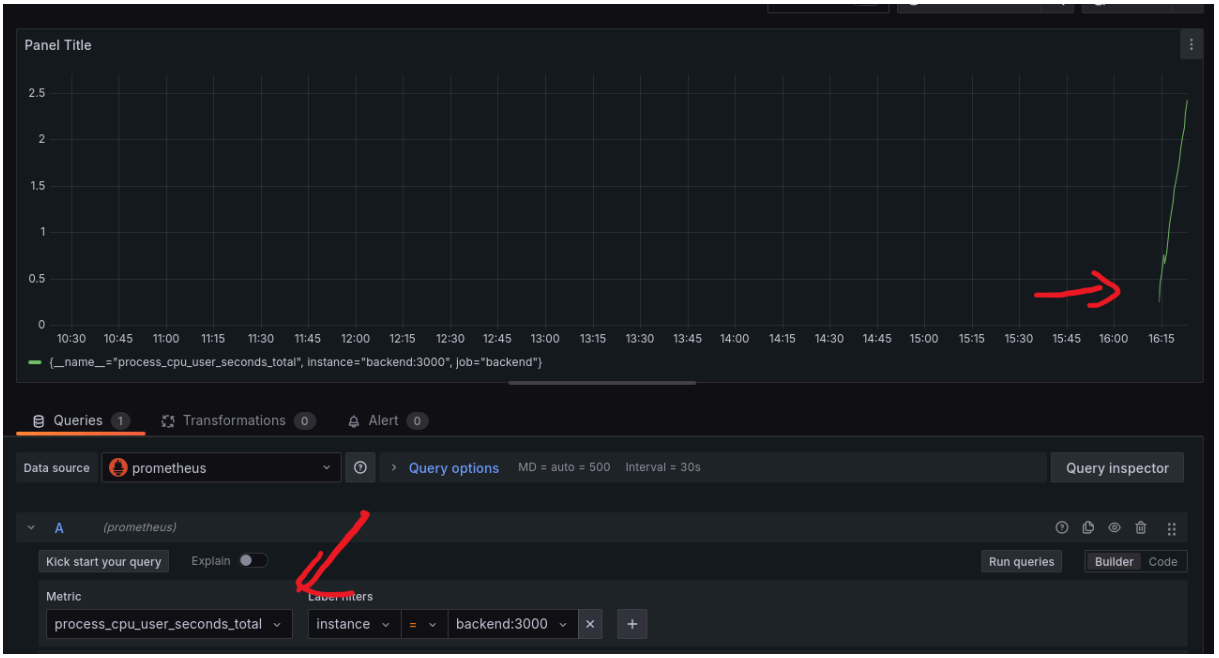


Le frontend *React* est exécuté dans un conteneur qui affiche l'interface sur le navigateur via l'adresse `http://$(minikube ip):30080`. Cette interface appelle le serveur backend *Node.js*, qui communique avec la base de données *Redis*, éventuellement en répartissant la charge entre le "maître" et ses réplicas.

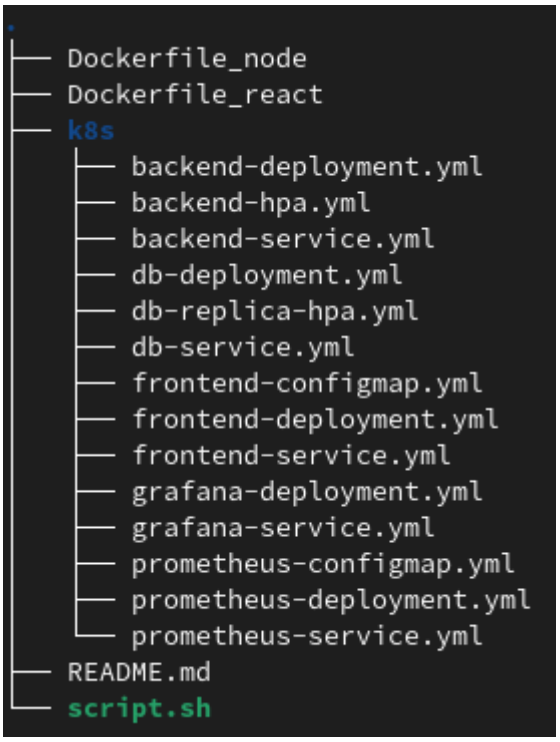
Du côté monitoring, *Grafana* interroge *Prometheus*, qui est connecté au backend (via l'endpoint */metrics*) pour collecter et afficher les données telles que l'utilisation CPU, la mémoire ou encore le nombre de requêtes.




Panel Title
0.08
0.06
0.04
0.02
0
10:30 10:45 11:00 11:15 11:30 11:45 12:00 12:15 12:30 12:45 13:00 13:15 13:30 13:45 14:00 14:15 14:30 14:45 15:00 15:15 15:30 15:45 16:00 16:15
{__name__="http_request_duration_seconds_sum", instance="backend:3000", job="backend", method="GET", route="/item", status="3XX"}
{__name__="http_request_duration_seconds_sum", instance="backend:3000", job="backend", method="GET", route="/items", status="3XX"}
Queries 1
Transformations 0
Alert 0
Data source
prometheus
Query options
MD = auto = 500
Interval = 30s
Query inspector
A
(prometheus)
Kick start your query
Explain
Run queries
Builder
Code
Metric
http_request_duration_seconds_sum
method
=
GET
Operations
hint: add rate
http_request_duration_seconds_sum{method="GET"}


Panel Title
2.5
2
1.5
1
0.5
0
10:30 10:45 11:00 11:15 11:30 11:45 12:00 12:15 12:30 12:45 13:00 13:15 13:30 13:45 14:00 14:15 14:30 14:45 15:00 15:15 15:30 15:45 16:00 16:15
{__name__="process_cpu_user_seconds_total", instance="backend:3000", job="backend"}
Queries 1
Transformations 0
Alert 0
Data source
prometheus
Query options
MD = auto = 500
Interval = 30s
Query inspector
A
(prometheus)
Kick start your query
Explain
Run queries
Builder
Code
Metric
process_cpu_user_seconds_total
instance
=
backend:3000

Pour détailler un peu plus, chaque composant est déployé dans le cluster *Kubernetes* via des fichiers *.yml* présentes dans le répertoire main/k8s. Les services *Kubernetes* permettent aux pods de communiquer entre eux via des noms DNS internes (db-service, backend-service…). Pour y accéder depuis l'extérieur du cluster, nous avons utilisé NodePort.



Ceci dit, l'application *React* a besoin de connaître dynamiquement l'adresse du backend (car cette dernière peut changer selon l'environnement. Pour cela on utilise un *ConfigMap* (frontend-configmap.yml) qui injecte l'URL du backend dans une variable d'environnement *REACT_APP_API_URL*. Cette variable est générée dynamiquement via le script d'automatisation qui récupère l'adresse IP ou le nom de service à l'aide de kubectl et minikube, puis écrit la variable dans le ConfigMap du frontend pour avoir une URL de serveur dynamiquement. Ainsi on peut déployer le frontend sans avoir à reconstruire l'image Docker manuellement à chaque changement d'IP ou de port.

A noter que nous avons mis cette variable d'environnement dans le fichier `redis-react/src/conf.js`, qui contient :

```
export const URL = process.env.REACT_APP_API_URL;
```

Nous avons aussi mis en place un système d'autoscaling grâce aux *Horizontal Pod Autoscalers* (HPA). Nous avons définis deux HPA : un premier pour le serveur *Node.js* (backend-hpa.yml) et l'autre pour les réplicas *Redis* (db-replica-hpa.yml). L'autoscalling est déclenché selon l'utilisation CPU. Cela permet d'adapter automatiquement le nombre de pods/réplicas en fonction de la charge, ce qui rend le système plus élastique et résilient.

Nous avons pu le tester grâce à la commande `watch kubectl get hpa` (à exécuter dans un autre terminal) qui nous décrit le nombre de réplicas créer ainsi que le rapport d'utilisation des ressources.

Enfin, *Prometheus* est aussi configuré via un *ConfigMap* (prometheus-configmap.yml) pour définir les cibles à surveiller (comme le backend par exemple). Les métriques sont exposées, comme dit précédemment via l'endpoint */metrics* sur le serveur *Node.js*. *Grafana* lui est déployé avec son propre Service, il récupère ces données depuis *Prometheus* et les affiche dans des tableaux de bord. Cela permet de suivre en temps réel les indicateurs importants du système.

Avec notre mise en place, nous pouvons faire plusieurs observations. L'infrastructure est bien modulaire, chaque composant est isolé et orchestré proprement. L'utilisation des fichiers *ConfigMap* et du script rend la configuration du frontend dynamique et flexible. L'autoscaling est efficace pour gérer les montées en charge automatiquement. Le monitoring avec *Prometheus/Grafana* permet une visibilité complète sur l'état du système. Cependant, nous avons aussi pu observer des limites et chercher quelques améliorations possibles, par exemple, l'utilisation d'un *NodePort* n'est pas idéale. Un *Ingress Controller* permettrait une exposition plus propre, sécurisée et gérable. Le monitoring pourrait être enrichi avec des alertes automatiques, via *AlertManager.* La gestion des données sensibles pourrait être améliorée avec les objets *Secret* de *Kubernetes* ou encore, les images Docker pourraient être optimisées avec un build multi-étapes pour réduire leur taille.