Cette pull request intègre une solution complète de supervision pour notre application Quarkus, comprenant :

- Collecte de métriques via Prometheus exposées par Micrometer sur l'endpoint /q/metrics.
- Visualisation des métriques grâce à Grafana avec des dashboards personnalisés ou importés.
- Tracing distribué avec Jaeger pour analyser le parcours des requêtes dans l'application.
- Système d'alerting configuré avec PromQL et Alertmanager, incluant des notifications par email.

L'objectif est d'améliorer la visibilité sur les performances, la santé et le comportement de l'application en production comme en développement, afin de faciliter la détection rapide des anomalies et optimiser le diagnostic.
Le déploiement de ces outils est automatisé via Docker Compose pour simplifier l'intégration et la maintenance.

Ajout de prometheus pour collecter les métriques

Prometheus est un outil de monitoring open-source utilisé pour collecter et stocker des métriques sous forme de séries temporelles. Il permet de surveiller les performances de l'application Quarkus grâce aux données exposées via Micrometer sur l'endpoint /q/metrics. Ces métriques sont ensuite consultables via l'interface Prometheus. Nous avons également mis en place un système d'alerting basé sur PromQL et Alertmanager.

Objectifs :

- modifier la configuration de Quarkus pour obtenir les métriques sur l'adresse **"/q/metrics"**
- configurer Prometheus pour collecter les métriques de Quarkus
- déployer Prometheus pour qu'il tourne en continu via Docker
- Ajouter un système d'alerting basé sur PromQL avec notifications par mail via Alertmanager.

1. **Configuration Quarkus**

Dans “application.yaml” au niveau de quarkus

Ajout en clair du port 8080 pour l’écoute http

```
http:
    port: 8080
    host: 0.0.0.0
```

Ajout du chemin /q/metrics pour obtenir les metrics

```
micrometer:
  export:
    prometheus:
      enabled: true
      path: /q/metrics
```

Dans “pom.xml”

Ajout de la dépendance micrometer

Mise en commentaire de la dépendance Quarkus metric (elle entre en conflit avec celle de prometheus)

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
</dependency>
```

2. **Configuration Prometheus**

Dans “prometheus.yml”

Configuration de Prometheus avec le chemin d’accès et le chemin d’accès aux metrics

```
global:
  scrape_interval: 15s #interval de récupération des métrics

scrape_configs:
  - job_name: 'quarkus-app'
    metrics_path: '/q/metrics'
    static_configs:
      - targets: ['192.168.100.42:8080']
```

```
rule_files:
  - "alerts.yml"

alerting:
  alertmanagers:
    - static_configs:
        - targets:
            - 'host.docker.internal:9093'
```

Concernant la target, il faut que ce soit l'IP locale pour que Quarkus puisse exécuter Prometheus à partir de docker. Étant sous WSL, il a fallu taper la commande :
ip addr show eth0 | grep inet, puis récupérer l'adresse inet avant le masque.

3. Configurer AlertManager

Dans un fichier "alertmanager.yml" que l'on crée

```
global:
  resolve_timeout: 1m

route:
  receiver: 'default'

receivers:
  - name: 'default'
    email_configs:
      - to: '<destinataire@example.com>'
        from: '<expediteur@example.com>'
        smarthost: '<smtp_server>:587'
        auth_username: '<utilisateur@example.com>'
        auth_identity: '<utilisateur@example.com>'
        auth_password: '<mot_de_passe>'
```

4. Déployer prometheus dans docker

Dans "docker-compose.yaml"

Configuration Prometheus à partir de son image, son nom de conteneur, le port pour obtenir Prometheus

```
  prometheus:
```

```
    image: prom/prometheus
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus:/etc/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'

  alertmanager:
    image: prom/alertmanager
    container_name: alertmanager
    ports:
      - "9093:9093"
    volumes:
      - ./alertmanager.yml:/etc/alertmanager/alertmanager.yml
```

5. **Tests**

Pour observer les métriques, il faut :

- Lancer l'application Quarkus **./mvnw quarkus:dev**
- Démarrer Prometheus **docker-compose up prometheus**

Pour vérifier que la configuration est bonne, nous pouvons dans un premier temps vérifier que les métriques sont bien récupéré sur l'adresse :

- http://localhost:8080/q/metrics

Puis ensuite, obtenir les différentes données sur Prometheus à l'adresse :

- http://localhost:9090

De plus, en allant dans Target health sur Prometheus, la cible devrait être UP si la configuration est bonne.

Concernant les alertes nous pouvons les vérifier via PromQL avec l'exemple suivant :

```
groups:
  - name: exemple-alertes
    rules:
      - alert: HighMemoryUsage
        expr: process_resident_memory_bytes > 500000000
        for: 1m
        labels:
          severity: warning
```

```
        annotations:
          summary: "Consommation mémoire élevée"
          description: "L'application dépasse 500MB de mémoire depuis plus d'une
minute."
```

Voici un résumé du fonctionnement de l'architecture :

Quarkus (Micrometer Prometheus)

↓

Prometheus (scrape)

↓

PromQL + rules (alerts.yml)

↓

Alertmanager (notifications)

Ajout de Grafana pour la visualisation des métriques

Grafana est une plateforme de visualisation de données qui permet de créer des tableaux de bord interactifs à partir de sources comme Prometheus. Elle permet de visualiser graphiquement les métriques collectées par Prometheus à travers des dashboards personnalisables ou importés.

Objectifs :

- configurer docker-compose.yaml pour visualiser les métriques de Prometheus

1. **Configuration Quarkus pour Grafana**

Pour cela, nous avons modifié le fichier concerné pour ajouter Grafana

```
services :
  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - "3000:3000"
    volumes:
      - grafana-storage:/var/lib/grafana
    depends_on:
      - prometheus #Grafana ne se lancera que si prometheus l'est aussi

volumes:
  grafana-storage: #le volume permet de sauvegarder les données ici les
dashboards créé
```

2. **Tests**

Après avoir fait les deux commandes comme précédemment :

- Lancer l'application Quarkus **./mvnw quarkus:dev**
- Lancer les containers **docker-compose up -d**

Nous pouvons nous rendre sur l'adresse

- http://localhost:3000

Nous retrouvons ici Grafana qui va nous permettre par la suite de visualiser sur un dashboard différentes métriques que nous avons récupéré à partir de Prometheus en paramétrant comme entrée de Data Source : http://prometheus:9090.
Nous pouvons créer nous mêmes les dashboards pour suivre des métriques.
Mais nous pouvons aussi importer des dashboard déjà fait, dans notre cas de l'application Quarkus, nous pouvons importer le dashboard suivant : Apicurio Registry(JVM Quarkus - Micrometer Metrics), à partir de son ID (18253).

Ajout de Jaeger pour le tracing

Jaeger est un outil de tracing distribué utilisé pour analyser les requêtes dans les architectures microservices. Il permet de suivre le parcours complet d'une requête à travers les différents composants de l'application Quarkus, facilitant ainsi le diagnostic de latences ou d'erreurs grâce aux traces envoyées via OpenTracing.

Objectifs :

- ajouter Jaeger pour suivre les traces sur OpenTracing
- configurer Quarkus pour envoyer les traces sur Jaeger
- déployer Jaeger pour qu'il tourne sur docker également

1. **Configuration Quarkus pour OpenTracing / Jaeger**

Dans application.yaml,

configuration de Quarkus pour envoyer les traces à Jaeger

```yaml
jaeger:
    service-name: quarkus-app
    sampler-type: const
    sampler-param: 1
    endpoint: http://localhost:14268/api/traces
```

Dans pom.xml

ajout de la dépendance Jaeger

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jaeger</artifactId>
</dependency>
```

2. **Déployer Jaeger dans Docker**

Dans docker-compose.yaml

ajout du service Jaeger

```yaml
jaeger:
    image: jaegertracing/all-in-one:1.43
    ports:
      - "16686:16686" #port pour visualiser les traces
```

```
    - "14268:14268" #port pour collecter les traces
```

3. Tests

A nouveau nous lançons l'application avec les différentes commandes :

- Lancer l'application Quarkus **./mvnw quarkus:dev**
- Lancer les containers **docker-compose up -d**
- Lancer le front **npm start**

Par la suite après avoir effectué quelques requêtes sur le projet (ajout d'une réunion), nous pouvons aller sur l'adresse : http://localhost:16686, pour apercevoir les étapes des requêtes.

Nous aurions également pu utiliser Apache SkyWalking à la place de Jaeger, ce dernier offrant une solution tout-en-un (métriques, logs, traces) avec intégration possible à Grafana.

Points bloquants et solutions

Lors de l'intégration de Prometheus, un conflit de dépendances a empêché Quarkus de démarrer. Le problème venait d'une incompatibilité entre quarkus-smallrye-metrics et quarkus-micrometer-registry-prometheus. En supprimant la première, l'application a pu exposer correctement les métriques au format Prometheus.

Prometheus, exécuté dans Docker, ne parvenait pas à joindre l'application Quarkus en local. Ce souci était dû à l'utilisation de localhost dans la configuration. En remplaçant l'adresse par l'IP locale de WSL obtenue via ip addr show eth0, le scraping des métriques a fonctionné.

Grafana ne montrait initialement aucune métrique. Le problème venait d'une mauvaise configuration de la datasource. En la définissant manuellement sur http://prometheus:9090 et en important un dashboard adapté (ID 18253), les visualisations sont devenues disponibles.

Côté Jaeger, les traces n'étaient pas visibles au départ. Après vérification, l'endpoint n'était pas bien configuré. Une fois corrigées dans le fichier application.yaml, les traces ont été correctement envoyées et visualisées.

Enfin, certains tags Docker comme stable ne fonctionnaient pas correctement. Le passage à des versions spécifiques, comme jaegertracing/all-in-one:1.43, a permis de stabiliser les déploiements.