

Cahier de TPs

Quarkus

Pré-requis :

Poste développeur avec accès réseau Internet libre

Linux (Recommandé) ou Windows 10

Pré-installation de :

- Git
- JDK17+
- Docker
- Distribution Kubernetes : *kind* recommandée
- IDE : Eclipse, VSCode ou IntelliJ
- lombok

Solutions des ateliers :

<https://github.com/dthibau/quarkus-solutions>

Table des matières

Ateliers 1 : Développer avec Quarkus.....	3
1.1 Démarrage projet.....	3
1.1.1 Création de projet.....	3
1.1.2 Mise en place IDE.....	3
1.2 : Annotations CDI.....	4
1.2.1 Mise à jour pom.xml.....	4
1.2.2 Beans et injection.....	4
1.2.3 Intercepteurs.....	5
1.3 : Configuration, Profils, Trace, Natifs et tests.....	5
1.3.1 Configuration.....	5
1.3.2 Profils.....	5
1.3.3 Configuration des traces.....	6
1.3.4 Construction Native.....	6
1.3.5 Tests.....	7
Ateliers 2 : Persistance.....	8
2.1 Hibernate, JPA.....	8
2.2 Panache.....	8
2.3 Panache réactif.....	8
2.3.1 Mise en place du projet.....	8

2.3.2 Implémentation Repository Pattern.....	9
2.4 Panache MongoDB.....	10
2.4.1 Mise en place.....	10
2.4.2 Implémentation.....	10
Ateliers 3 : API Rest.....	11
3.1 Mapping JAX-RS.....	11
3.1.1 Mode impératif.....	11
3.1.2 Mode réactif.....	11
3.2 Problématiques RestFul.....	12
3.2.1 Sérialisation avec Jackson.....	12
3.2.2 Exceptions, Validation et OpenAPI.....	12
Ateliers 4 : Interactions RPC.....	13
4.1 Rest Client.....	13
4.1.1 Appel synchrone.....	13
4.1.2 Appel réactif.....	13
4.2 SOAP Client.....	13
Ateliers 5 : Messaging.....	16
5.1 Émission de message vers topic Kafka.....	16
5.2 Consommation de message.....	16
5.3 SSE.....	17
Ateliers 6 : Sécurité.....	18
6.1 Authentification HTTP.....	18
6.2 OpenID Connect.....	18
6.2.1 Mise en place.....	18
6.2.2 OpenID Connect.....	18
6.3 Propagation de jeton.....	19
6.3 Client credentials.....	19
Atelier 7: Containers.....	21
7.1 Construction d'image.....	21
7.2 Ressources Kubernetes.....	22
7.3 Development mode.....	22
Ateliers 8. Observabilité.....	25
8.1 Health check.....	25
8.2 Tracing distribué.....	26
8.3 Métriques.....	27

Ateliers 1 : Développer avec Quarkus

1.1 Démarrage projet

Objectifs :

- Installation de Quarkus CLI
- Découverte DevUI
- Création premier projet

1.1.1 Création de projet

Installer Quarkus CLI :

```
https://quarkus.io/guides/cli-tooling#installing-the-cli
```

Créer un projet *delivery-service* avec l'extension *resteasy-reactive*

```
$ quarkus create app org.formation:delivery-service \
  --extension=quarkus-rest
$ cd delivery-service
$ quarkus dev
```

Accéder à la page d'accueil et :

- Visiter la *DevUI*
- La ressource */hello*

Lister les extensions disponibles pour le projet :

```
$ quarkus ext ls
```

1.1.2 Mise en place IDE

Installer les plugins pour l'IDE de votre choix

Puis importer le projet Maven dans votre IDE

Visualiser :

- le fichier *pom.xml*
- la classe présente dans *src/main/java*
- le répertoire *docker*
- le répertoire *src/main/resources*

- la classe de test présente dans `src/test/java`

Ensuite :

1. Visualisez la ressource */hello* dans un navigateur
2. Modifier le fichier *GreetingResource*
3. Faire un reload dans le navigateur
4. Provoquer une erreur
5. Faire un reload

1.2 : Annotations CDI

Objectifs

- Annotations CDI
- Exemple d'intercepteur

1.2.1 Mise à jour pom.xml

Ajouter une dépendance sur *lombok*¹ en éditant votre *pom.xml* :

```
<lombok.version>1.18.30</lombok.version>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>${lombok.version}</version>
  <scope>provided</scope>
</dependency>
```

Ajouter également une bibliothèque de sérialisation JSON, par exemple :

```
$ quarkus ext add resteasy-reactive-jackson
```

1.2.2 Beans et injection

Récupérer les sources fournis

Annotez les sources fournis afin que le endpoint *localhost:8080/livraisons* renvoie un tableau d'objets Livraison en JSON

1 Si vous êtes avec Eclipse, vous devez configurer le démarrage d'Eclipse afin qu'il prenne en compte lombok

1.2.3 Intercepteurs

Définissez un intercepteur **@Logged** tel que présenter dans les slides.

L'appliquer :

- sur la méthode **findAll()** de **LivraisonController**
- sur toutes les méthodes de **LivraisonServiceImpl**

1.3 : Configuration, Profils, Trace, Natifs et tests

Objectifs

- Configuration propriétés Quarkus
- Objets de configuration
- Injection de propriété
- Profils de configuration

1.3.1 Configuration

Configurer Quarkus afin que le serveur écoute sur le port 8000

Définir une interface qui définit les propriétés d'un service de notification offrant une API Restful :

- **protocol** : http ou https, valeur par défaut : « http »
- **host** : Non vide
- **port** : Optionnel
- **rootUrl** : Optionnel
- **token** : Longueur minimale 10 caractères

Ajouter des contraintes de validation et vérifier quelles sont effectives

S'injecter la configuration dans le bean **LivraisonServiceImpl** et afficher les valeurs de config dans la méthode d'initialisation

1.3.2 Profils

Définir un autre port pour le profil de **prod**

Construire un jar avec

```
./mvnw package
```

Démarrer ensuite l'application via :

```
java -jar target/quarkus-app/quarkus-run.jar
```

Vérifier que le profil prod est activé

Définir un autre port pour le profil **staging**

Fixer la propriété **quarkus.application.name** pour ce profil

Construire un jar avec

```
./mvnw package -Dquarkus.profile=staging
```

Démarrer l'application via :

```
java -jar target/quarkus-app/quarkus-run.jar
```

Et vérifier les profils activés

1.3.3 Configuration des traces

Activer le mode DEBUG pour le logger root, sauf pour le package *io.quarkus*

Générer un message de DEBUG dans la classe Controller ou Service

1.3.4 Construction Native

Si Linux :

Construire un artefact natif en utilisant docker comme machine de build, l'exécuter
quarkus build --native --no-tests -
Dquarkus.native.container-build=true

Exécuter l'exécutable natif :

```
target/delivery-service-1.0.0-SNAPSHOT-runner
```

Image Docker

Ajouter l'extension **container-image-docker**

Construire une image Docker contenant l'artefact natif

```
./mvnw package -Pnative \  
-Dquarkus.native.container-build=true \  
-Dquarkus.container-image.build=true
```

L'exécuter

```
docker run -p 9000:9000 \  

```

```
<userid>/delivery-service:1.0.0-SNAPSHOT
```

Observer les temps de démarrage

1.3.5 Tests

Se mettre dans le mode « tests continus »

Écrire une classe de test testant une URL de la ressource *LivraisonController*

Mettre à jour la classe de test : faire échouer le test, puis le restaurer.

Écrire un test d'intégration et vérifier son exécution avec

```
./mvnw -Pnative -Dquarkus.native.container-build=true \
verify
```

Ateliers 2 : Persistance

2.1 Hibernate, JPA

Ajouter l'extension *quarkus-jdbc-h2* et *quarkus-hibernate-orm*

Récupérer les classes entités fournies (package *org.formation.domain*) fournies

Placer le fichier *import.sql* fourni dans *src/main/resources*

Configurer Hibernate afin qu'il affiche les traces SQL

Recharger l'application et visualiser les traces SQL de génération de la base

Implémenter ensuite les méthodes de *LivraisonServiceImpl* en s'appuyant sur un objet *EntityManager*.

Assurer vous que les tests de la classe de test fournie passent. (Remarquer bien *@TestTransaction*)

2.2 Panache

Ajouter l'extension pour Hibernate Panache

Récupérer la nouvelle classe *Livreur*

Implémenter le pattern Entity sur la classe classe Livreur

Fournir les requêtes suivante :

- Retrouver tous les livreurs actifs
- Ajouter une revue à un Livreur
- Effacer toutes les revues d'un Livreur

Écrire une classe de test validant votre travail.

2.3 Panache réactif

2.3.1 Mise en place du projet

Créer un nouveau projet *order-service*


```
quarkus create app order-service --no-code
```

Ajouter les extensions :

- quarkus-hibernate-reactive-panache
- quarkus-reactive-pg-client

Ajouter également la dépendance lombok

Démarrer une base de données postgres en utilisant le fichier docker-compose fourni :

```
docker-compose -f postgres-docker-compose.yml up -d
```

Créer une base de données *order-service* via pgAdmin

Récupérer les classes de domaine fournies

Configurer la datasource réactive comme suit :

```
# configure your datasource
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = postgres
quarkus.datasource.password = postgres
quarkus.datasource.reactive.url =
vertex-reactive:postgresql://localhost:5432/order-service

# drop and create the database at startup
quarkus.hibernate-orm.database.generation = drop-and-create
quarkus.hibernate-orm.log.sql=true
```

Effectuer un premier démarrage et observer la création des tables

2.3.2 Implémentation Repository Pattern

Définir un bean *OrderRepository* implémentant *PanacheRepository<Order>* y définir de requêtes spécifiques :

- Les commandes où une remise (champ discount) a été appliquée
- Les commandes dont la date est supérieur à un paramètre

Exécuter les tests fournis de *OrderRepositoryTest*

Implémenter un autre bean *org.formation.service.OrderService* utilisant le repository précédent et coder la méthode :

```
public Uni<Order> createOrder(List<OrderItem> lineItems,
```

`Address deliveryAddress,`
`PaymentInformation paymentInformation)`

2.4 Panache MongoDB

2.4.1 Mise en place

Créer un nouveau projet *ticket-service* avec les extensions :

- quarkus-mongodb-panache
- lombok

Fournir un *application.properties* donnant un nom à la base mongo
`quarkus.mongodb.database=tickets`

Faire un premier démarrage et observer le démarrage du container MongoDB

2.4.2 Implémentation

Récupérer les sources fournis : classe du modèle + service

Définir *org.formation.domain.Ticket* comme une *PanacheMongoEntity*

Compléter les méthodes de la classe *org.formation.service.TicketService*

Vérifier que les tests passent

Ateliers 3 : API Rest

Objectifs

- Annotations JAX-RS
- Distinction mode réactif mode impératif
- Utilisation Jackson
- Problématiques classiques API Rest

3.1 Mapping JAX-RS

3.1.1 Mode impératif

Sur le projet *delivery-service*

Dans la classe *org.formation.web.LivraisonController*, implémenter une API CRUD permettant

- Accéder à une livraison par son id
- Créer/mettre à jour, supprimer une livraison

Vous pouvez modifier l'interface *LivraisonService* pour implémenter cette API

Implémenter un test de l'API de création

3.1.2 Mode réactif

Sur le projet *order-service*

Ajouter les extensions :

- *resteasy-reactive*
- *resteasy-reactive-jackson*

Écrire un contrôleur implémentant les endpoints:

- Retournant toutes les commandes
- Permettant de créer une commande

Écrire des tests *RestAssured* pour ces 2 endpoints

3.2 Problématiques RestFul

3.2.1 Sérialisation avec Jackson

Sur le projet *delivery-service*

A l'aide de `@JsonView`, définir 2 formats de sérialisation pour la classe *Livraison* :

- **Base** : *id, noCommande, status, dateCreation, id* du Livreur
- **Comple**t : Idem avec en plus les informations complètes du Livreur incluant les *Reviews*

Appliquer les formats adéquats aux différents endpoints

3.2.2 Exceptions, Validation et OpenAPI

Sur le projet *delivery-service*

En utilisant `@ServerExceptionMapper`, faire en sorte de retourner une réponse 404 lorsqu'une entité n'est pas retrouvée via son id

Mettre en place une documentation Swagger. Tester les 2 URLS :

- *http://localhost:8000/q/openapi/*
- *http://localhost:8000/q/swagger-ui/*

Sur le projet *order-service*

Ajouter des contraintes de validation pour la création d'une commande

Tester avec Swagger

Ateliers 4 : Interactions RPC

4.1 Rest Client

Récupérer les sources du projet *notification-service*, le démarrer et visualiser la documentation Swagger

4.1.1 Appel synchrone

Dans le projet *delivery-service*, .

On utilisera le endpoint impératif.

Ajouter l'extension :

quarkus-rest-client-reactive-jackson

Définir une interface *NotificationService*, y définir une méthode :

```
NotificationDto sendMail(NotificationDto notification);
```

L'annoter en conséquence

Modifier *application.properties* pour définir l'URL racine du service

Dans la classe *org.formation.service.impl.LivraisonServiceImpl*, se faire injecter le client REST avec *@RestClient*

Ajouter le code permettant d'invoquer le endpoint. lors de la création d'une Livraison

Tester par l'interface swagger et vérifier la console de *notification-service*

4.1.2 Appel réactif

Dans le projet *order-service*

Faire la même chose que précédemment mais en utilisant le endpoint reactif

Tester via swagger

4.2 SOAP Client

Démarrer le service web Calculator via

```
docker run -p 8082:8080 quay.io/l2x6/calculator-ws:1.0
```

Accéder au WSDL

curl -s <http://localhost:8082/calculator-ws/CalculatorService?wsdl>

Créer un projet quarkus *soap-client*

```
quarkus create app soap-client -no-code
```

Ajouter l'extension CXF

```
quarkus ext add io.quarkiverse.cxf:quarkus-cxf
```

Et Resteasy

```
quarkus ext add quarkus-resteasy-jackson
```

Vérifier la présence de l'objectif *generate-code* dans le pom.xml

Placer le wsdl dans *src/main/resources*

```
curl -s  
http://localhost:8082/calculator-ws/CalculatorService?  
wsdl > src/main/resources/calculator.wsdl
```

Indiquer l'emplacement du wsdl dans application.properties :

```
quarkus.cxf.codegen.wsdl2java.includes=*.wsdl
```

Effectuer un démarrage avec *quarkus dev* et vérifier la génération de code

Ajouter les configurations du service web :

```
cxf.it.calculator.baseUrl=http://localhost:8082  
quarkus.cxf.client.myCalculator.wsdl = ${cxf.it.calculator.baseUrl}/calculator-  
ws/CalculatorService?wsdl  
quarkus.cxf.client.myCalculator.client-endpoint-url = $  
{cxf.it.calculator.baseUrl}/calculator-ws/CalculatorService  
quarkus.cxf.client.myCalculator.service-interface =  
org.jboss.eap.quickstarts.wscalculator.calculator.CalculatorService
```

Développer une ressource REST :

```
@Path("/cxf/calculator-client")  
public class CxfClientResource {  
    @CXFClient("myCalculator")  
    CalculatorService myCalculator;  
  
    @GET  
    @Path("/add")  
    @Produces(MediaType.TEXT_PLAIN)  
    public int add(@QueryParam("a") int a, @QueryParam("b") int b) {  
        return myCalculator.add(a, b);  
    }  
}
```

```
}  
  
}
```

Tester avec l'URL

</cxf/calculator-client/add?a=1&b=1>

Ateliers 5 : Messaging

5.1 Émission de message vers topic Kafka

Sur le projet *order-service* ajouter les extensions :

- smallrye-reactive-messaging
- smallrye-reactive-messaging-kafka

Démarrer le projet et observer le démarrage d'un container Kafka

Accéder à la DevUI et visualiser les liens relatifs à Kafka

Récupérer les sources fournies et en particulier la classe du domaine *OrderEvent*

Écrire un service *org.formation.service.PublishService* permettant d'envoyer des *OrderEvent* vers le topic *orders*

La méthode via les déclaration de channels est recommandée car elle permet de profiter des configuration par défaut et en particulier les sérialiseurs

Utiliser ce service dans *OrderService* lors de la création d'une commande.

Tester l'envoi de message via swagger

5.2 Consommation de message

Du côté du projet *ticket-service* :

- Ajouter les extensions pour Kafka

Récupérer les sources fournis, visualiser les différences avec les classes de données de *order-service*

Créer un service *org.formation.service.ConsumeOrderService*

- S'injecter un *Channel* pointant vers le topic *orders* précédent sous la forme de `Multi<OrderEvent>` et déclencher la souscription au démarrage de Quarkus.
- Le traitement des messages consiste à tester le type d'évènement.
Si `CREATED` alors création de ticket

Envoyer un message via le projet *order-service* et vérifier la réception côté *ticket-*

service

5.3 SSE

Dans le projet ***ticket-service***

- Ajouter si besoin l'extension ***quarkus-resteasy-reactive-jackson***
- Créer une ressource *GET* sur le endpoint ***/incoming-orders*** qui renvoie un ***Multi<OrderEvent>*** correspondant au topic ***orders*** au format ***MediaType.SERVER_SENT_EVENTS***
- Récupérer le fichier `orders.html` et le placer dans ***src/main/resources/META-INF/resources***
- Le comprendre

Utiliser le script JMeter pour créer un certain nombre de commandes et visualiser les événements dans la page html (`http://<server>/orders.html`)

Vous pouvez également utiliser le script JMeter fourni

Ateliers 6 : Sécurité

6.1 Authentification HTTP

Sur le projet *order-service*, ajouter l'extension : *elytron-security-properties-file*

Activer l'authentification http basique

```
quarkus.http.auth.basic=true
```

Mettre en place un *IdentityProvider* basé sur un fichier *.properties* comme suit :

```
%dev.quarkus.security.users.embedded.enabled=true
%dev.quarkus.security.users.embedded.plain-text=true
%dev.quarkus.security.users.embedded.users.client=client
%dev.quarkus.security.users.embedded.users.manager=manager
%dev.quarkus.security.users.embedded.roles.client=user
%dev.quarkus.security.users.embedded.roles.manager=admin,user
```

Utiliser les annotations afin que les endpoints

- GET (findAll) ne soit accessible que par le rôle *admin*
- POST createOrder par le rôle *user*

6.2 OpenID Connect

6.2.1 Mise en place

Récupérer le projet *gateway* fourni.

Visualiser le code source

Enlever les contrôles sur les rôles de *order-service*

Démarrer les 2 projets et créer une commande en passant par la gateway

6.2.2 OpenID Connect

Sur le projet *gateway*, ajouter l'extension *quarkus-oidc*

Protéger la ressource Gateway via *@Authenticated*

Démarrer Gateway et accéder à la DevUI puis au lien OpenIDConnect

Se logger avec *bob/bob*

Observer le token d'accès et d'identification

Créer une commande via Swagger

Se connecter avec *alice/alice* et visualiser les jetons

6.3 Propagation de jeton

Ajouter l'extension *quarkus-oidc-token-propagation-reactive* au projet *gateway*

Application du filtre *AccessTokenRequestReactiveFilter* sur REST Client accédant à *orders-service*

Sur le projet *order-service* :

- Enlever l'extension *elytron-security-properties-file*
- Ajouter *quarkus-oidc*

Remettre les contrôles d'accès sur les ressources de *order-service*

Tester via Swagger

6.3 Client credentials

Vérifier que le serveur Keycloak est démarré et que vous pouvez obtenir un jeton via :

```
curl -X POST \  
  -u "quarkus-app:secret" \  
  -d "grant_type=client_credentials" \  
  "http://localhost:<port-keycloak>/realms/quarkus/protocol/openid-connect/  
token"
```

Démarrer *notification-service*

Sur le projet *order-service* ajouter les extensions *quarkus-oidc-client* et *quarkus-oidc-client-reactive-filter*

Définir le serveur Keycloak et le client dans *application.properties*

quarkus.oidc-client.auth-server-url=http://localhost:32783/realms/quarkus

quarkus.oidc-client.client-id=quarkus-app

quarkus.oidc-client.credentials.secret=secret

Modifier l'interface *NotificationService* afin que l'appel vers *notification-service*

fournisse le jeton en appliquant le filtre ***OidcClientRequestReactiveFilter***

Activer les traces pour les requêtes de RestClient

quarkus.rest-client.logging.scope=request-response

quarkus.rest-client.logging.body-limit=50

quarkus.log.category."org.jboss.resteasy.reactive.client.logging".level=DEBUG

Effectuer ensuite un création de commande via la gateway et visualiser les traces dans *order-service* lors de l'appel au service de notification.

Vous devriez voir le jeton obtenu par *order-service*

Visualiser le jeton sur jwt.io

Le service notification peut alors être protégé.

Y ajouter l'extension ***quarkus-oidc*** et ajouter l'acl *@Roles("default-roles-quarkus")*

Tester à nouveau

Atelier 7: Containers

Objectifs

- Construction d'image
- Déploiement vers un cluster
- Remote development / Debug development
- Variables d'environnement et ConfigMap

7.1 Construction d'image

Nécessite un compte DockerHub

Récupérer le projet *delivery-service*, il apporte les modifications suivantes :*

- Mock du service notification-service durant les tests
- Définition d'une base H2 en profil prod

Ajouter l'extension *container-image-jib* et supprimer *container-image-docker*

Indiquer le nom de l'image dans *application.properties*, DockerHub ne vous laissera pousser que des images préfixées par votre compte

```
quarkus.container-image.image=dthibau/quarkus-delivery-service:1.0.0-SNAPSHOT
```

Construire l'image en activant le profil de *prod*:

```
quarkus build -Dquarkus.container-image.build=true -Dquarkus.profile=prod
```

Ensuite push vers DockerHub

```
docker login
docker push dthibau/quarkus-delivery-service:1.0.0-SNAPSHOT
```

Exécuter ensuite l'image dans l'environnement local et essayer d'accéder à l'application.

Par exemple :

```
docker run --network host dthibau/quarkus-delivery-service:1.0.0-SNAPSHOT
```

7.2 Ressources Kubernetes

Démarrer votre cluster Kubernetes

Sur le projet *delivery-service*

Ajouter l'extension kind

```
quarkus ext add quarkus-kind
```

Effectuer un build et visualiser les ressources Kubernetes créées

```
quarkus build
```

Assurer vous que votre cluster Kubernetes est démarré et que votre client *kubectl* est effectif. Déployer les ressources sur votre cluster avec *kubectl* :

```
kubectl apply -f target/kubernetes/kind.yml
```

Vérifier le démarrage du pod et la présence du service

Effectuer un port-forward afin de pouvoir accéder au service via un port local :

```
kubectl port-forward service/delivery-service 8001:80
```

7.3 Remote development mode

Supprimer les ressources *kubernetes* déployées manuellement :

```
kubectl delete service delivery-service
```

```
kubectl delete deployment delivery-service
```

Ajouter les propriétés

```
quarkus.kubernetes.deploy=true
```

```
quarkus.kubernetes.namespace=default
```

Effectuer un build et observer le déploiement automatique sur votre cluster

Ajouter ensuite la configuration suivante :

```
quarkus.kubernetes.service-type=node-port
quarkus.package.type=mutable-jar
quarkus.kubernetes.env.vars.quarkus-launch-devmode=true
quarkus.live-reload.password=secret
```

Supprimer les configurations spécifiques sur les ports http (sinon soucis sur la génération de la ressource service kind)

Effectuer un déploiement, vérifier le remplacement du container

```
kubectl get po -w
```

Lors la nouvelle version est déployée, vérifier les logs :

```
kubectl logs <pod-id>
```

afin d'y voir :

```
(Quarkus Main Thread) Profile dev activated. Live Coding
activated.
```

Récupérer le host de votre cluster *kind*

```
export CLUSTER_HOST=$(kubectl get po -l
app.kubernetes.io/name=delivery-service -n default -o
jsonpath='{.items[0].status.hostIP}')
```

```
export NODE_PORT=$(kubectl -n default get service
delivery-service -o
jsonpath='{.spec.ports[0].nodePort}')
```

Accéder à l'URL ***http://\$CLUSTER_HOST:\$NODE_PORT***

Vous devez voir la page de Bienvenue de Quarkus

Indiquer cette URL dans la propriété :

```
quarkus.live-reload.url
```

Lancer ensuite :

```
mvn quarkus:remote-dev
```

Vous devez voir : ***Connected to remote server***

Modifier le source et faire reload dans le navigateur. Le container devrait être déployé

avec la modification.

Ateliers 8. Observabilité

8.1 Health check

Sur le projet *delivery-service*

Ajouter l'extension *quarkus-smallrye-health*

Démarre le projet via

```
mvn quarkus:remote-dev
```

Visualisation les endpoints de health

Créer une nouvelle classe *org.formation.DiskSpaceHealthCheck* avec le code suivant :

```
@Startup
@ApplicationScoped
public class DiskSpaceHealthCheck implements HealthCheck{

    private static final long MINIMUM_DISK_SPACE_REQUIRED = 1_000_000_000; // 1 GB

    @Override
    public HealthCheckResponse call() {
        long freeDiskSpace = getFreeDiskSpace();

        if (freeDiskSpace >= MINIMUM_DISK_SPACE_REQUIRED) {
            return HealthCheckResponse
                .named("disk-space")
                .up()
                .withData("freeDiskSpace", freeDiskSpace)
                .build();
        } else {
            return HealthCheckResponse
                .named("disk-space")
                .down()
                .withData("freeDiskSpace", freeDiskSpace)
                .withData("message", "Insufficient disk space.")
                .build();
        }
    }

    private long getFreeDiskSpace() {
        File root = new File("/");
        return root.getUsableSpace();
    }
}
```

Recharger les URLs

- [http://\\$CLUSTER_HOST:\\$NODE_PORT/q/health/](http://$CLUSTER_HOST:$NODE_PORT/q/health/)
- [http://\\$CLUSTER_HOST:\\$NODE_PORT/q/health/started](http://$CLUSTER_HOST:$NODE_PORT/q/health/started)

8.2 Tracing distribué

Démarrer un serveur Jaeger

```
docker run -p 5775:5775/udp -p 6831:6831/udp -p 6832:6832/udp -p 5778:5778 -p 16686:16686 -p 14268:14268 jaegertracing/all-in-one:latest
```

Enlever les tests de sécurité sur notification-service

Supprimer l'extension *quarkus-oidc*

```
quarkus ext remove quarkus-oidc
```

Ajouter l'extension *quarkus-smallrye-opentracing*

Mettre à jour les propriétés

```
quarkus.jaeger.service-name=notification-service
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, parentId=%X{parentId}, spanId=%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%n
```

Démarrer le service

Faire la même chose pour *delivery-service*

Accéder à *delivery-service* et faire quelques requêtes, en particulier création d'une livraison.

Observer les traces des services, vous devez y voir les traces d'OpenTracing indiquant des `traceId` et des `spanIds`

Accéder à Jaeger : <http://localhost:16686/>

Retrouver les traces de vos appels.

8.3 Métriques

Sur le projet *delivery-service*

Ajouter les extensions :

- *quarkus-micrometer*
- *micrometer-registry-prometheus*

Démarrer et accéder à la page Prometheus (lien accessible dans la DevUI)

<http://localhost:8080/q/dev-ui/io.quarkus.quarkus-micrometer/prometheus>

Effectuer quelques appels via l'API

Ajouter ensuite les métriques pour la base de données :

```
quarkus.datasource.metrics.enabled=true
```