Cahier de TPs

Quarkus

Pré-requis :
Poste développeur avec accès réseau Internet libre
Linux (Recommandé) ou Windows 10
Pré-installation de :

- Git
- JDK11
- Docker
- Distribution Kubernetes : kind recommandé
- IDE : Eclipse, VSCode ou IntelliJ
- lombok

Solutions des ateliers :

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

Atelier 1 : Premier projet

Objectifs

- Installation de Quarkus cli
- Création premier projet

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=resteasy-reactive
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

1.2 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 ressource présente dans ***src***/***main***/***java***
- le répertoire ***docker***
- le répertoire ***resources***
- la classe de test présente dans ***src***/***test***/***java***

Ensuite :

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

Atelier 2 : CDI

.

Objectifs

- Annotations CDI
- Exemple d'intercepteur

2.1 Récupération des sources

Mettre à jour votre ***pom.xml*** afin qu'il référence une bibliothèque de sérialisation JSON et lombok

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

Récupérer les sources fournis

2.2 Annotations

Annotez les sources fournis afin que le endpoint ***localhost:8080/livraisons*** renvoie une liste de Livraison en JSON

Atelier 3 : Configuration

Objectifs

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

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 :

- protocole (http ou https)
- Base URL
- port
- URI
- API Token

Ajouter des contraintes de validation et vérifier quelles sont effectives

S'injecter a configuration dans le bean ***LivraisonServiceImpl***

3.2 Profils

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

Construire un jar avec
`./mvnw package`

Démarrer l'application via :

```
java -jar target/delivery-service-1.0.0-SNAPSHOT-native-image-
source-jar/delivery-service-1.0.0-SNAPSHOT-runner.jar
```

3.3 Configuration des traces

Activer le mode DEBUG pour le logger root

Sauf pour le package *io.quarkus*

Utilisé les 2 techniques de log :

- simplifié
- org.jboss.logging.Logger

3.4 Profil Maven Native

Construire un artefact natif et l'exécuter

Construire une image Docker contenant l'artefact natif et l'exécuter

3.5 Tests

Se mettre dans le mode test continus

Mettre à jour la classe de test existante *GreetingsResourceTest* et faire échouer le test, puis le restaurer.

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

Écrire un test d'intégration et vérifier son exécution avec *./mvnw -Pnative verify*

Atelier 4 : API Rest

.

Objectifs

- Annotations JAX-RS
- Distinction mode réactif mode impératif
- Utilisation Jackson
- Problématiques classiques API Rest
- Invoquer un autre service REST

4.1 Mapping RestEasy-Reactive

Préfixer toutes les urls de l'application par "/api/v1"

Implémenter une API CRUD permettant

- Accéder à une livraison
- Créer/mettre à jour supprimer une livraison

Vérifier les endpoints par la DevUI

Implémenter un endpoint avec le modèle réactif.

Observer les threads utilisés pour les différents endpoints

4.2 Sérialisation avec Jackson

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

- ***Base*** : id, noCommande, status,, id du Livreur
- ***Complet*** : Avec en plus date de création , les informations complètes du Livreur incluant les Reviews

Appliquer les formats adéquats aux différents endpoints

4.3 Exceptions, Validation et OpenAPI

Faire en sorte de retourner une réponse 404 lorsqu'une livraison n'est pas retrouvée via son id

Mettre en place une documentation Swagger

Ajouter des contraintes de validation sur les objets du domaine et tester les cas de mise à jour d'une Livraison

4.4 Rest Client

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

Dans *delivery-service*, appeler le service de notification via un client REST lors de la création d'une Livraison

Atelier 5 : Persistance

5.1 Source de données

Démarrer une base PostgreSQL par le fichier docker-compose fourni

Configurer une base de développement h2 et une base Reactive PostgreSQL

5.2 Hibernate

Ajouter l'extension pour Hibernate
Remplacer les classes du domaine qui contiennent les annotations JPA

Configurer Hibernate afin qu'il affiche les traces SQL

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

Modifier la classe service afin qu'elle effectue des opérations de persistance

5.3 Panache

Ajouter l'extension pour Hibernate Panache

Modifier la classe Livraison afin qu'elle hérite de *PanacheEnittyBase* ou de *PanacheEntity* (vous devez alors enlever l'annotation @Id)

Réécrire la classe service afin d'utiliser l'API de Panache

5.4 Panache MongoDB

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

- resteasy-reactive
- mongo-panache
- openapi
- lombok

Récupérer les classes du modèle fourni ainsi que la ressource web.

Créer et implémenter une classe service afin que les endpoints REST soient opérationnels

Atelier 6 : Messaging

6.1 Émission de message vers topic Kafka

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

- resteasy-reactive
- openapi
- h2 et hibernate-panache
- smallrye-reactive-messaging
- smallrye-reactive-messaging-kafka
- lombok

Récupérer les sources fournis

Démarrer le projet et observer le démarrage d'une image docker

Configurer un canal de sortie *order-out* pointant vers le topic Kafka *order*

Emettre un message de type OrderEvent lors de la création d'un objet Order.

La méthode via les déclaration de channels est recommandée car elle permet de profiter des sérialiseurs JSON automatiquement

6.2 Réception de message

Du côté du projet ***product-service*** :

- Ajouter l'extension pour Kafka
- Définir un channel pointant le topic précédent
- S'abonner au channel et effectuer le traitement suivant :
  - Création d'un ticket
  - Publication sur le topic *order* d'un OrderEvent avec le statut PENDING

6.3 Réception de message et SSE

Revenir dans le projet order-service

- Dans le controller, ajouter un endpoint /status renvoyant un Multi<OrderEvent> correspondant au topic order au format MediaType.SERVER_SENT_EVENTS
- Récupérer le fichier orders.html et le placer dans src/main/resources/META-INF/resources
- Le comprendre

Créer via le swagger de order-service une commande et visualiser les événements dans la page html (http://<server>/orders.html

Vous pouvez également utiliser le script JMETER fourni

Atelier 7 : Sécurité

Objectifs

7.1 Authentification HTTP

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

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.scott=reader
%dev.quarkus.security.users.embedded.users.stuart=writer
%dev.quarkus.security.users.embedded.roles.scott=READER
%dev.quarkus.security.users.embedded.roles.stuart=READER,WRITER
```

Utiliser les annotations afin que les endpoints du controller GET soient accessibles par le rôle READER et les autres endpoints par le rôle WRITER

7.2 Protection via Bearer Token

Objectif : utiliser l'extension Quarkus OpenID Connect (OIDC) pour protéger les endpoints de ***delivery-service*** à l'aide de l'autorisation d'un jeton Bearer émis par OpenID Connect et des serveurs d'autorisation conformes à OAuth 2.0 tels que Keycloak.

Ajouter l'extension ***oidc***
Ajouter des contrôles d'accès :

- Role ***user*** pour les requêtes GET
- Rôle ***admin*** pour les autres requêtes

Démarrer l'application et accèder à la DevUI puis au lien OpenIDConnect
Se logger avec ***bob/bob***
Observer le token d'accès
Accéder à une URL GET (vous pouvez utiliser le lien swagger)
Essayer une URL POST
Observer les requêtes envoyées avec F12
Se logger avec ***alice/alice*** et essayer une URL POST

7.3 RestClient

Objectif : utiliser l'extension quarkus-oidc-client afin que l'appel REST de ***delivery-service*** vers notification-service fournisse un jeton Bearer. Le service notification peut alors être protégée

Ajouter OIDC à notification-service et protéger les endpoints via le rôle ***admin***

Ajouter l'extension ***quarkus-oidc-token-propagation-reactive*** au projet ***delivery-service***
Profiter de la configuration par défaut

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

Atelier 8: Déploiement

Objectifs

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

8.0 Installation Kubernetes

- minikube
- kind

+ Helm

8.1 Construction d'image

Créer un nouveau projet :

```
quarkus create app org.formation:greeting-service \
    --extension=resteasy-reactive
cd  greeting-service
```

Ajouter l'extension container-image-jib

Construire l'image en activant le profil de *dev* et en autorisant éventuellement contrôler le nom de l'image avec quarkus.container-image.name

Push vers DockerHub (Nécessite un compte DockerHub)

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

8.2 Kubernetes

Ajouter l'extension Kubernetes

Effectuer un build et visualiser les ressources Kubernetes créées

Assurer vous que votre cluster Kubernetes est démarré et que votre client kubectl est effectif

Déployer les ressources sur votre cluster avec *kubectl*

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/greeting-service 8080:80

Ajouter la propriété quarkus.kubernetes.deploy=true

Définir également le namespace defaul

8.2.2 Development mode

Ajouter 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
```

Effectuer un déploiement, vérifier le remplacement du container (kubectl get -w pod)
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 :
minkube : `minikube ip`
kind : `kubectl get po -o jsonpath='{.items[0].status.hostIP}'`
Récupérer également le NodePort de greeting-service

Accéder à l'URL http://<cluster-host>:<NodePort>
Vous devez voir la page de Welcome de Quarkus

indiquer cette URL dans la propriété :
`quarkus.live-reload.url`

Lancer ensuite quarkus remote-dev

Vous devez voir : Connected to remote server

Modifier le source et faire reload dans le navigateur

Essayer sur un autre projet

8.3 Observabilité

Visualisation des endpoints de health

Mise en place d'OpenTracing sur delivery-service et notification-service

Visualisation des métriques MicroMeter