Ateliers

Nouveautés Spring6

Pré-requis :
Poste développeur avec accès réseau Internet libre
Linux, Windows 10, MacOS
Pré-installation de :

- JDK17+
- Git
- Docker
- IDE : IntelliJ, VSCode, STS
- Apache JMeter

Solutions :

- `https://github.com/dthibau/spring6-solutions.git`

Table des matières

Atelier 1: Migration

L'objectif est de migrer un projet SB2.x vers SB3.x.
Le projet est une application Spring MVC classique s'appuyant sur une BD relationnelle et offrant une API REST documentée par swagger.
2 profils sont définis :

- dev : Utilisation d'une base H2
- prod : Utilisation de postgresql

Après avoir démarré l'application :

- Swagger est accessible à http://localhost:8080/swagger-ui.html
- Actuator est accessible à http://localhost:8080/actuator

1.1 Upgrade SB version

Reprendre le projet fourni

Upgrader vers Java 21 et la dernière version de Spring Boot 3

Faire en sorte que le projet compile, les tests passent, l'application se lance.

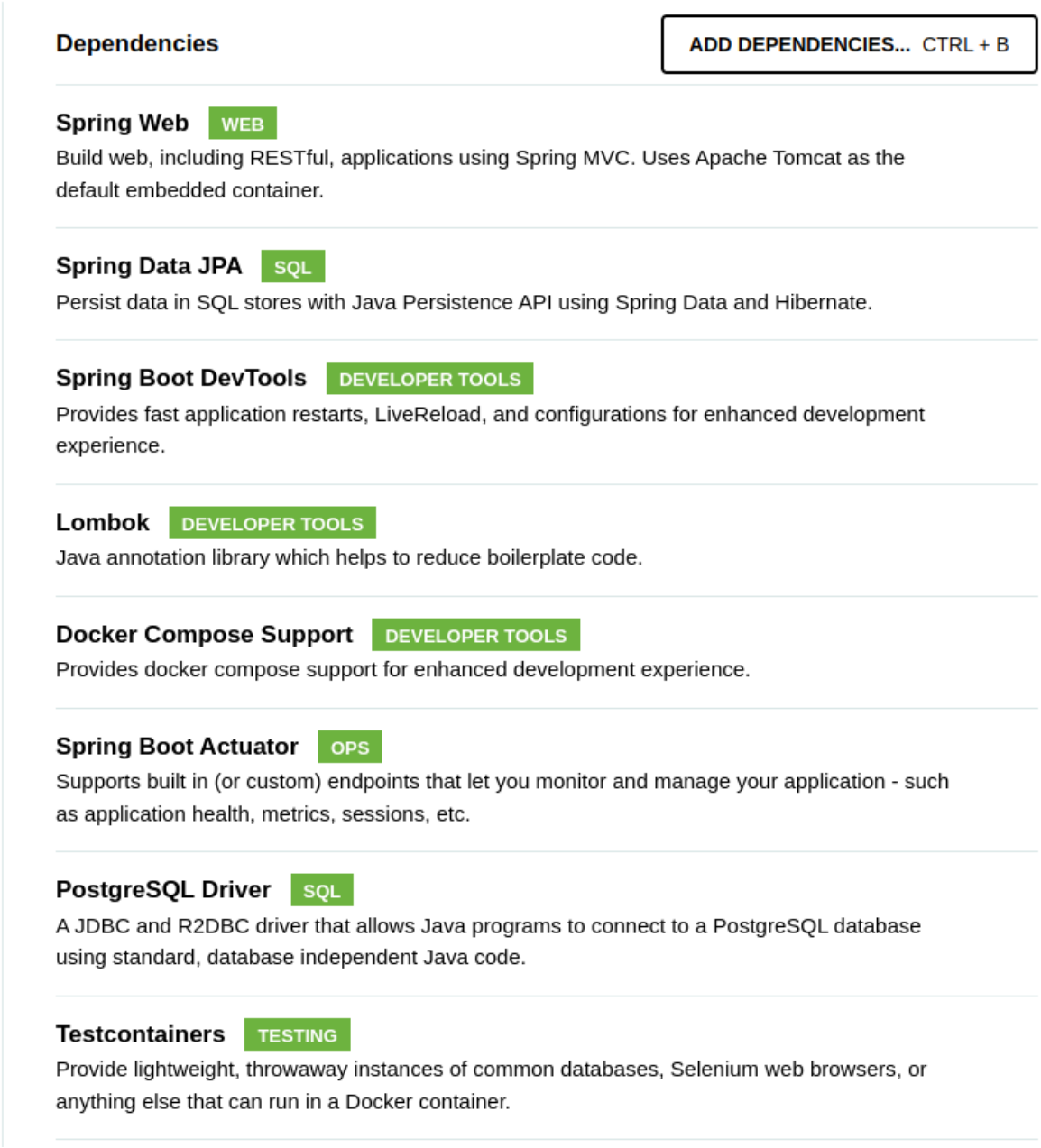
Tâches à faire :

- Migration vers jakarta
- Migration vers JUnit5
- Utilisation starter *spring-doc* et suppression *SwaggerConfiguration*

1.2 Migration vers Gradle

En s'aidant de starter.io, migrer le projet vers gradle.

Voici les starters à utiliser :



L’ajout du support docker compose suppose que l’on utilise la base postgres dans le profil par défaut.

Mettre à jour un fichier compose.yaml à la racine

```
services:
  delivery-postgresql:
    image: postgres
    volumes:
      - delivery_db:/var/lib/postgresql
      - delivery_data:/var/lib/postgresql/data
    networks:
      - back
    environment:
      - POSTGRES_USER=postgres
```

```
    - POSTGRES_PASSWORD=postgres
   ports:
    - 5434:5432

 pgadmin4:
   image: dpage/pgadmin4
   environment:
    PGADMIN_DEFAULT_EMAIL: "admin@admin.com"
    PGADMIN_DEFAULT_PASSWORD: "admin"
   ports:
    - "81:80"
   depends_on:
    - delivery-postgresql
   networks:
    - back
volumes:
 delivery_data:
 delivery_db:

networks:
 back:
```

Configurer spring.jpa afin que la base soit également cré à chaque démarrage.

Concernant les tests, un container postgres doit être démarré lors de l'excution impliquant la base de données :

- Test système complet DeliveryApplicationTest
- Test couche repository : org.formation.repository

En commande en ligne :

- Exécuter les tests
- Construire un jar et l'exécuter dans le profil prod.

Atelier 2 : Persistance

2.1 Migration du modèle

Revoir les classes du package modèle :

- Passer la classe Trace en mode immuable
- Sur la classe Livraison
  - Ajouter un champ ***commande*** au format JSON.
  - Une commande est représenté par le record suivant :
    ```
    public record Commande(String noCommande, String dateCommande, Long customerId) {}
    ```
- Sur la classe Livreur
  - Utiliser un UUID pour l'id
  - Ajouter une List d'entier reviews

Démarrer l'application et visualiser via pgAdmin :

- Les séquences
- Les modifications dans la base.

Utiliser l'interface Swagger dans son état pour interagir avec la BD

2.2 Validation API

Ajouter les contraintes suivantes sur le modèle

- Le n° de commande doit respecter le pattern suivant : ***\d{4}-\d{5}***
- La date de commande doit être dans le passé
- Les entiers de la liste ***reviews*** doivent être compris entre 1 et 5

Vérifier que les tests repository échouent puis les fixer

2.3 Package Repository

Ajouter dans la classe LivraisonRepository une méthode qui exécute la requête suivante en JPQL:

- La date la plus ancienne d'une Livraison donnée
- La moyenne de l'année des date d'une Livraison donnée

Ajouter des tests pour valider la syntaxe de votre requête

2.4 Mise en place du cache

2.4.1 Configuration par défaut

Ajouter le starter cache
Autoriser le caching avec ***@EnableCaching***

Ajouter les annotations de cache pour cacher la méthode ***findById***

Activer dans actuator le endpoint caches

Tester le cache en appelant plusieurs fois la méthode findById

Effectuer un PATCH en modifiant le statut d'une livraison

Tester le cache en ré-appelant la méthode findById

Visualiser le endpoint /*actuator/*caches

Nettoyer le cache via actuator :
curl -X DELETE http://localhost:8080/actuator/caches/livraisons

Tester le cache en ré-appelant la méthode findById

2.4.2 Mise en place de Caffeine

Ajouter les dépendances suivantes
implementation 'com.github.ben-manes.caffeine:caffeine:3.1.8'

Configurer Caffeine dans le application.yml

```
spring:
  cache:
    type: caffeine
    caffeine:
      spec: maximumSize=500,expireAfterAccess=10m
```

Redémarrer l'appli et vérifier son fonctionnement

2.4.2 Mise en place Hazelcast

Ajouter les dépendances suivantes :

implementation 'com.hazelcast:hazelcast-all:4.2.8'

Implémenter l'interface ***Serializable*** dans Livraison et Trace

Ajouter dans le ***compose.yaml*** le service Hazelcast :

```
hazelcast:
  image: hazelcast/hazelcast
  environment:
    HZ_NETWORK_PUBLICADDRESS : localhost:5701
    HZ_CLUSTERNAME: delivery
  ports:
    - 5701:5701
```

Créer un fichier de configuration Hazelcast pour spring ***hazelcast-client.yaml***

```
hazelcast-client:
  cluster-name: delivery  # Nom du cluster Hazelcast
  network:
    cluster-members:
      - localhost:5701  # Adresse et port des membres du cluster
    connection-timeout: 5000  # Temps d'attente (ms) avant l'échec de la connexion
  properties:
    hazelcast.client.invocation.timeout.seconds: 10  # Timeout pour les invocations
    hazelcast.client.statistics.enabled: true       # Activer les statistiques client
  load-balancer:
    type: round-robin  # Algorithme de répartition des connexions (par défaut : round-robin)
  metrics:
    enabled: true  # Activer la collecte des métriques client
  labels:
    - delivery-service  # Labels pour identifier le client (utile pour des tags spécifiques)
```

Démarrer l'application et susciter le cache

Vérifier avec actuator/caches

Démarrer une 2ème instance de delivery-service sur un autre port

Vérifier le bon comportement du cache

Atelier 3 : Couche web

3.1 Modèle de threads

L'objectif de cette partie est de comparer les performances et le scaling du modèle bloquant vis à vis du modèle non bloquant

Récupérer le projet Web fourni (modèle bloquant)

Créer un Spring Starter Project et choisir le starter ***web-reactive***

Implémenter un contrôleur équivalent à celui du modèle bloquant.

Utiliser le script JMeter fourni, effectuer des tirs avec les paramètres suivants :
NB_USERS=100, PAUSE=1000

A la fin du résultat, notez :

- Le temps d'exécution du test
- Le débit

Effectuez plusieurs tirs en augmentant le nombre d'utilisateurs.

Observer les threads

Reprendre le projet impératif et activer les VirtualThreads

Refaire des tirs

3.2 Spring Webflux

Reprendre le projet fourni ***3.3-reactive-stack***

Visualisez les starters réactifs utilisés

Visualiser les couches services, repository et controleur

Tester l'application via Swagger

Atelier 4 : RestClient / WebClient

4.1. RestClient

Créer une nouvelle application SpringBoot avec le starter *mvc*

Modifier la méthode main de la classe principale afin qu'elle ne démarre pas de serveur web :

```
SpringApplication app = new SpringApplication(RestclientApplication.class);
// prevent SpringBoot from starting a web server
app.setWebApplicationType(WebApplicationType.NONE);
app.run(args);
```

Écrire un classe service qui utilise le service REST de l'atelier delivery-service.

Le service exposera des méthodes permettant les requêtes suivantes :

- Création d'un nouvel Livraison
- Récupération de la livraison via son ID
- Patch de la livraison

Écrire une classe de test qui valide les interactions.

4.2. WebClient

Créer une nouvelle application SpringBoot avec le starter *webflux*

Modifier la méthode main de la classe principale afin qu'elle ne démarre pas de serveur web :

```
SpringApplication app = new SpringApplication(WebclientApplication.class);
// prevent SpringBoot from starting a web server
app.setWebApplicationType(WebApplicationType.NONE);
app.run(args);
```

Écrire une classe service qui utilise le service REST reactif r2dbc .

Le client effectuera les requêtes suivantes :

- Création d'un nouvel Account
- Récupération de l'account via son ID
- Récupération du premier élément retourné par la liste de tous les accounts

Écrire une classe de test qui valide les interactions.

Atelier 5 – SSE et Websockets

5.1 SSE

Reprendre le projet fourni.

Démarrer le broker Kafka via le docker-compose fourni :

- Soit directement en ligne de commande
- Soit via le starter docker support

Dans la classe Contrôleur implémenter un endpoint qui envoit un tick via SSE toutes les secondes

Démarrer l'application et accéder au endpoint

Définir un autre endpoint mappé sur le flux de message kafka

Accéder au navigateur

Produire des messages avec :

docker exec -it <container-kafka> bash

kafka-console-poducer --bootstrap-server localhost:9092 --topic sse

Ou en accédant à la console Redpanda :

http://localhost:9090

5.2 : WebSockets

5.2.1. Serveur

Créer une nouvelle application SpringBoot avec le starter ***webflux***

Fournir un bean implémentant *WebSocketHandler, le bean :*

- Écoutera le fluc d'entrée et l'affiche sur la console
- Envoie un *Flux<String>* de 5 événements

Dans une classe de configuration :

- Définir un mapping sur `"/event-emitter"` pour un *WebsocketHandler*,
- Créer un bean de type `WebSocketHandlerAdapter`

Démarrer le serveur et observer le démarrage de *Netty*

5.2.2 Client

Écrire une classe qui dans sa méthode main

- Instancie un `ReactorNettyWebSocketClient`
- Démarre une session WebSocket
  - Dans la méthode *send* envoie un simple message texte
  - Dans la méthode *receive*, reçoit tous les messages du serveur et les log.
- Bloquer le flux après un délai

Atelier 6 : Sécurité

6.1. Configuration MVC

6.1.1 Sécurité Web

Reprendre le projet delivery

Ajouter le starter *security*

Définir une configuration de sécurité :

- Une base utilisateur mémoire avec un utilisateur simple et un utilisateur avec le rôle *ADMIN*
- Tous les URLs nécessitent une authentification par formulaire
- Il faut avoir le rôle *ADMIN* pour créer ou patcher un*e Livraison*

6.1.2 Protection des méthodes

Activer la protection des méthodes

Protéger la méthodes permettant de patcher une livraison exigeant le rôle *SUPER_ADMIN*

6.2. Configuration reactive

6.2.1. Protection des URLs

Reprendre le projet réactif

Ajouter le starter *security*

Définir une configuration de sécurité Webflux :

- Une base utilisateur mémoire avec un utilisateur simple et un utilisateur avec le rôle *ADMIN*
- Tous les URLs nécessitent une authentification par formulaire
- Il faut avoir le rôle *ADMIN* pour créer un *Account*

6.2.2 Protection des méthodes

Activer la protection des méthodes

Protéger la méthode permettant de mettre à jour *Account* particulier en imposant que le champ owner soit égal à ***authentication.principal.username***

6.3 oAuth2/OpenID

6.3.1 Authentification via OpenId

Mise en place Keycloak

Démarrer un serveur KeyCloak via Docker :

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e
KEYCLOAK_ADMIN_PASSWORD=admin quay.io/keycloak/keycloak:20.0.2
start-dev
```

- Connecter vous à sa console d'administration avec admin/admin
- Créer un Realm ***StoreRealm***
- Sous ce realm, créer 2 clients
  - ***store-app***
  - ***monitor***

    Pour ces 2 clients, positionner ***access-type*** à ***confidential*** et Valid Redirect Uri à **** (pas recommandé en production)***

    Donner un scope ***MONITOR*** au client *monitor*
- Créer ensuite un utilisateur ***user/secret***

Obtention des jeton :

Grant type *password* pour le client *store-app*

```
curl -XPOST http://localhost:8089/realms/StoreRealm/protocol/openid-connect/token -
d grant_type=password -d client_id=store-app -d client_secret=<client_secret> -d
username=user -d password=secret
```

Obtenir un jeton de rafraîchissement en utilisant une requête POST avec :

```
client_id:store-app
client_secret:<client_secret>
'refresh_token': refresh_token_requete_précédente,
grant_type:refresh_token
```

Grant type *client_credentials* pour le client monitor

Dans KeyCloak, autoriser le grant type ***client_credentials***

Settings → Access Type = Confidential

Service Accounts Enabled

Tester avec un requête curl :
curl -XPOST http://localhost:8089/auth/realms/StoreRealm/protocol/openid-connect/token -d grant_type=client_credentials -d client_id=monitor -d client_secret=<client_secret>

OpenID login sur la Gateway

Ajouter les starters ***security*** ***et*** ***oauth2-client*** au projet gateway fourni

Configurer a sécurité de la gateway (Bean étendant *SecurityFilterChain*) comme suit :

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
            .pathMatchers("/actuator/**").permitAll()
            .pathMatchers("/auth/**").permitAll()
      .anyExchange().authenticated()
      .and().
      oauth2Login().csrf().disable().build();
}
```

Configurer le client oauth2 de Keycloak comme suit :

```
spring :
  security:
    oauth2:
      client:
        provider:
          keycloak:
            issuer-uri: http://localhost:8080/realms/StoreRealm
        registration:
          store-app:
            provider: keycloak
            client-id: store-app
            client-secret: <your_secret>
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
            scope: openid
```

Accéder ensuite à l'API via la gateway via un navigateur

6.3.2 Gateway as resource server

Ajouter les starters ***oauth2-resource-server*** ***et*** ***oauth2-jose*** au projet gateway.

Indiquer la propriété ***spring.security.oauth2.resourceserver.jwr.issuer-uri***

Modifier la configuration de la sécurité pour s'adapter au resource server

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
            .pathMatchers("/actuator/**").permitAll()
            .pathMatchers("/auth/**").permitAll()
      .anyExchange().authenticated()
      .and().
      oauth2ResourceServer(v -> v.jwt()).csrf().disable().build();
}
```

Configurer les routes de la gateway de telle façon que le serveur KeyCloak soit accessible via la gateway.

Utiliser le script *JMeter* ***oAuth2WithGateway*** pour valider votre configuration

6.3.3 : Relai de jeton

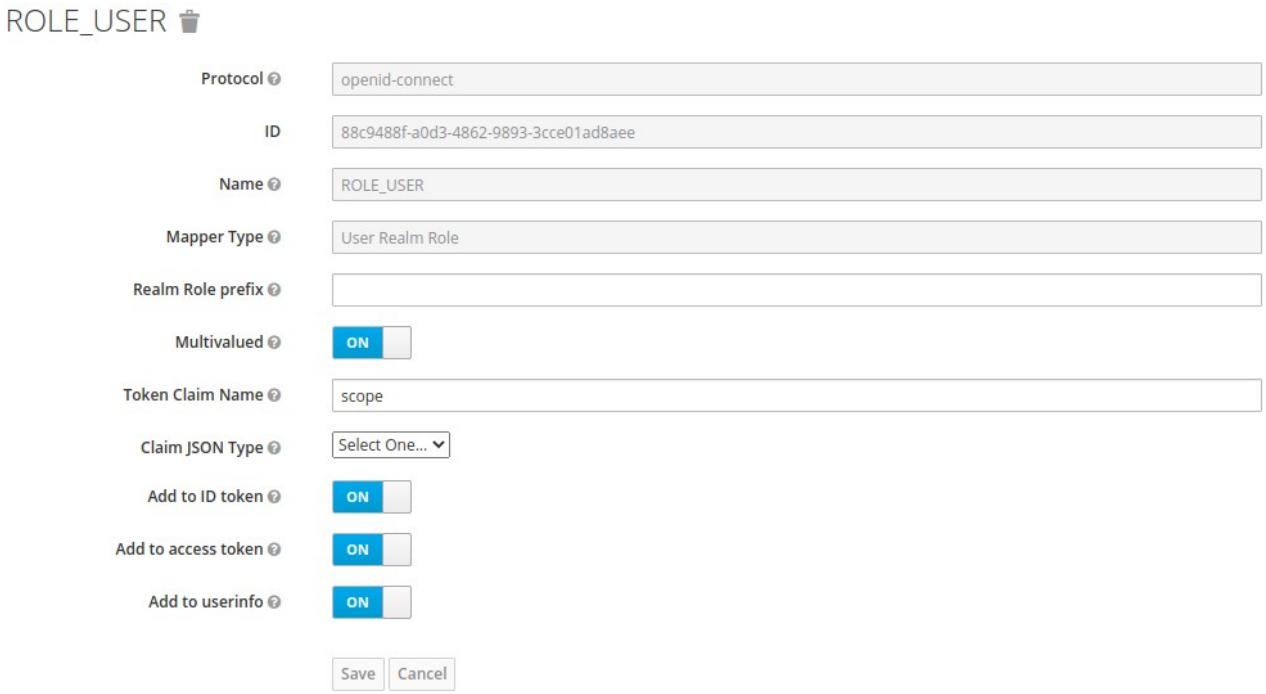
Rôle *USER* à l'utilisateur *user*

Dans KeyCloak, ajouter un rôle USER au client ***store-app***.

Affecter ce rôle à l'utilisateur user

Mapper pour inclure le rôle dans le jeton JWT sous l'attribut scope

Dans le client ***store-app***, ajouter un nouveau mapper, comme suit :



Sécurisation service order-service

Configurer *order-service* comme *ResourceServer* de la même manière que gateway

Définir les ACLs suivantes :

- Pour toutes le requête le client oAuth2 doit être dans le scope USER
- Pour accéder aux endpoints ***actuator***, le client doit être dans le scope MONITOR

Autoriser le relai des entêtes d'autorisation dans la configuration de la gateway

Utiliser le script jMeter ***oAuth2WithGateway*** pour valider votre configuration :

Le groupe d'utilisateur store-app Client valide les protections via les rôles USER

Le groupe d'utilisateur monitor client valide les protections via le scope associé à monitor

Atelier 7 : SpringBootTest et Test auto-configurés

Compléter les tests sur les projet delivery-service et réactif

7.1 Design By Contract et Test de composants

Côté producteur

Visualiser les contrats des projets ***OrderService*** et ***DeliveryService***

Générer les classes de test via ***./mvnw test-compile*** par exemple

Visualiser la classe de test puis exécuter les tests.
Les serveurs de *config,Eureka* et le broker Kafka sont démarrés lors de leur exécution

Une fois que les tests passés, installés l'artefact dans votre dépôt Maven :
./mvnw install

Effectuer le même processus pour les 2 projets producteur

Côté consommateur

Dans le projet ***order-query-service***

Visualiser la classe de test et les références aux 2 précédents contrats publiés

Effectuer le test en isolation.

Atelier 8 Packaging

8.1 Packaging jar

Comparer les temps de démarrage :

- jar pris tel quel
- jar explosé
- Utilisation de CDS

8.2 Image docker

Vérifier la présence de ***layers.idx***

Effectuer une construction d'image et identifier les LAYERS de l'image

docker history <image_name>

8.3 Format natif

Ajouter le support pour GraalVM sur delivery-service

2 possibilités pour construire un code natif

- Utiliser buildpack pour construire une image contenant GraalVM
- Disposer d'une distribution GralVM et construire un exécutable Linux

8.3.1 Avec Docker

./mvnw -Pnative spring-boot:build-image

Lors du démarrage de l'image, l'application cherche à se connecter à une base postgres.

Il faut donc démarrer un Postgres pour voir l'application fonctionner.

Mettre au point un docker compose permettant de démarrer la stack

8.3.2 Avec GraalVM

Installer GraalVM

export GRAALVM_HOME=/usr/lib/graalvm

export PATH=$GRAALVM_HOME/bin:$PATH

./gradlew nativeCompile

./build/native/nativeCompile/delivery-service