Cahier de TP et démonstrations
«Architecture des SI »

Outils utilisés :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10
- JDK11+
- Spring Tools Suite 4.x avec Lombok
- Git
- Docker
- Apache JMeter
- npm

Dépôts Git des solutions :
git clone https://github.com/dthibau/architecure-si-solutions.git

Certains ateliers utilisent des services d'appui, pour les démarrer nous nous appuyons sur docker-compose

- Postgres :
  `docker-compose -f postgres-docker-compose.yml up -d`
  Se connecter à l'interface d'administration à http://localhost:81 avec
  admin@admin.com/admin
- Kafka
  `docker-compose -f kafka-docker-compose.yml up -d`
  Déclarer dans votre fichier ***hosts*** (*/etc/hosts* Linux, c:*\Windows\System32\drivers\etc\hosts* sous Windows)
  `127.0.0.1 kafka`

Atelier 1: BD, ORM et Transaction

Importer le projet Maven ***1-JPA-Transaction*** dans votre IDE

Créer une base de données postgres en accord avec le fichier ***META-INF/persistence.xml***

Le projet est constitué de 2 packages :

- ***org.formation.dao*** : Utilitaires pour intégration Hibernate
- ***org.formation.domain*** : Modèle métier

Regarder les annotations des entités ***Departement*** et ***Employe.***

Quel est le champ/annotation utilisé pour la gestion de la concurrence des transactions ?

Le projet contient également une classe de test. Regarder son code et bien comprendre les intentions des 2 méthodes de tests.

S'assurer de la bonne exécution des tests

Atelier 2: Communication asynchrone avec Kafka

Démarrer le cluster Kafka

2.1 Production de messages dans un topic Kafka

Importer le projet ***2-position-service*** dans votre IDE

Le projet offre une ressource HTTP (/position) permettant à une flotte de coursier d'envoyer régulièrement leur position (***org.formation.rest.PositionController***)

Le contrôleur s'appuie sur une classe service (org.formation.service.PositionService) qui stocke les informations dans un topic Kafka.

La configuration du topic se trouve dans ***src/main/ressources/application.yml***

Démarrer l'application (*Run As → Spring Boot App*) et vérifier sa bonne connexion à Kafka

Lancer JMeter et ouvrir le scénario de test ***Position.jmx*** permettant de simuler une flotte de 500 coursiers

Lancer le test et vérifier la console de l'application *position-service*
Si tout se passe bien laisser tourner le test pendant 5 minutes afin de générer suffisamment de messages

2.2 Réception des messages à posteriori

Importer le projet ***2-average-service*** dans votre IDE

Le projet :

1. consomme un topic Kafka, (Voir ***org.formation.service.PositionsListener***)
2. calcule la position moyenne pour chaque coursier minute par minute
3. et écrit son résultat dans un autre topic Kafka (Voir ***org.formation.service.AverageService***)

La configuration des topics se trouve dans ***src/main/ressources/application.yml***

Nous avons configuré le consommateur afin qu'il consomme les messages depuis le début du topic au démarrage

Arrêter le projet précédent

Démarrer l'application (*Run As → Spring Boot App*) et vérifier sa bonne connexion à Kafka et la consommation à posteriori des messages envoyés précédemments

Relancer le projet ***position-service*** et observer les traitements d'événements en « temps-réel »

Atelier 3: Monolithique Spring MVC + Data

Importer le projet ***3-mvc*** dans votre IDE

Le démarrer, accéder et logger vous en utilisant les utilisateurs définis dans ***src/main/resources/users.csv***

Effectuer le cas d'utilisation d'édition d'un produit.

Quel est la séquence des appels (contrôleurs et vue) lors de ce cas d'utilisation ?

Atelier 4: Services RestFul

Importer le projet ***4-rest*** dans votre IDE

Le démarrer et trouver les points d'accès de l'API

Effectuer des exemples de requêtes

Trouver l'adresse de la documentation et exécuter des requêtes via la documentation générée par l'outil Swagger

Atelier 5: OpenAPI et Design By Contract

Nécessite *npm*

Lancer Swagger-editor via :

docker run -d -p 80:8080 swaggerapi/swagger-editor

Importer le projet Maven ***5-contract***

Visualiser le fichier open-api ***contract.yml*** via swagger-editor (accessible en http://localhost)

Installer le générateur :
Dans le répertoire du projet :
npm install @openapitools/openapi-generator-cli -D

Générer le stub :
npx @openapitools/openapi-generator-cli generate -i contract.yml -g spring -o . --additional-properties=sourceFolder=src/open-api/java --additional-properties=delegatePattern=true --additional-properties=basePackage=org.formation

Les classes générés sont dans ***src/open-api/java***

Reprendre ***pom.bak.xml*** et le copier dans ***pom.xml*** qui a été écrasé par la génération

Démarrer l'application, accéder à la documentation et afficher la liste des produits.

Atelier 6: Load-balancing et circuit-breaker pour les micro-services

2.1 Mise en place serveurs de discovery et de config

Déclarer dans votre fichier hosts, les lignes suivantes

```
127.0.0.1 annuaire
127.0.0.1 config
```

Démarrer le serveur de configuration en ligne de commande :

```
cd 6-config
./mvnw clean package
java -jar target/config-0.0.1-SNAPSHOT.jar
```

Vérifier le bon démarrage avec les URLS suivantes :

- *http://config:8888/application/default*
- *http://config:8888/ProductService/replica*

Démarrer le serveur de discovery :

```
cd 6-eureka
./mvnw clean package
java -jar target/eureka-0.0.1-SNAPSHOT.jar
```

Vérifier le serveur eureka :

- http://localhost:1111

2.2 Micro-services applicatifs

Importer les 2 projets ***6-OrderService*** et ***6-ProductService*** dans l'IDE

Démarrer une instance de ***Order-service***

Démarrer *product-service* 2 fois dont une fois en activant le profile ***replica***

Vérifier les bonnes inscriptions des 2 services dans le serveur de discovery Eureka

2.3 Répartition de charge et Circuit Breaker Pattern

Utiliser le script JMeter ***CreateOrder.jmx*** présent dans le projet *6-OrderService* pour simuler une charge

Visualiser la répartition de charge, les 2 instances de *ProductService* doivent produire des traces

Arrêter 1 puis l'autre instance de *ProductService* et observer les traces de *OrderService*.

Redémarrer ensuite un instance

Atelier 7: Interface utilisateur Angular

Ouvrir le projet ***7-angular*** avec vsCode.

Ce projet a été réalisé suivant l'excellent tutorial :
https://www.kevinboosten.dev/how-i-use-an-openapi-spec-in-my-angular-projects

Il permet à partir du contrat ***contract.yml*** (atelier 5) de générer les classes services d'Angular.

Commencer par installer angular-cli version 8.2.2
npm install -g @angular/cli@8.2.2

Installer ensuite le générateur de code :
npm i @openapitools/openapi-generator-cli -D

7.1 Génération de code

- Visualiser la commande de génération dans ***package.json***
- Supprimer le répertoire ***product-openapi/src/app/core/api***
- Générer le code service
- Visualiser son utilisation dans le composant Angular ***product-list***

7.2 Simulateur d'API

Dans le répertoire ***simulator***, visualiser la référence à la spécification dans index.ts

Démarrer le serveur
npm start

Accéder à ***http://localhost:9000/produits***

Utiliser l'application angular

Atelier 8: Démarrage de la stack via docker-compose

8.1 Construction de l'image du back-end

Pour construire l'image du back-end, nous nous appuyons sur le plugin Maven *spring-boot*.
Dans le répertoire ***5-contract*** :
./mvnw spring-boot:build-image

L'image créée doit être taggée ***openapi-spring:v0***

Pour la tester :
docker run -p 8082:8080 openapi-spring:v0

8.2 Construction de l'image front-end

Pour construire l'image *front-end*, nous nous appuyons sur un *Dockerfile*

Visualiser celui qui est dans le projet *7-client-angular*
Vérifier la base URL du projet Angular et faites la pointer vers http://localhost:8082/api
Construire en version *prod* l'application Angular :
ng build --prod

Construire le conteneur :
docker build . -t product-openapi

Tester le démarrage du conteneur
docker run -p 82:80 product-openapi

8.3 Démarrage de la stack via docker-compose

Visualiser le fichier *docker-compose* à la racine du workspace

Reconstruire le front-end :
docker-compose build

Démarrer la stack via :
docker-compose up -d

Accéder à l'application