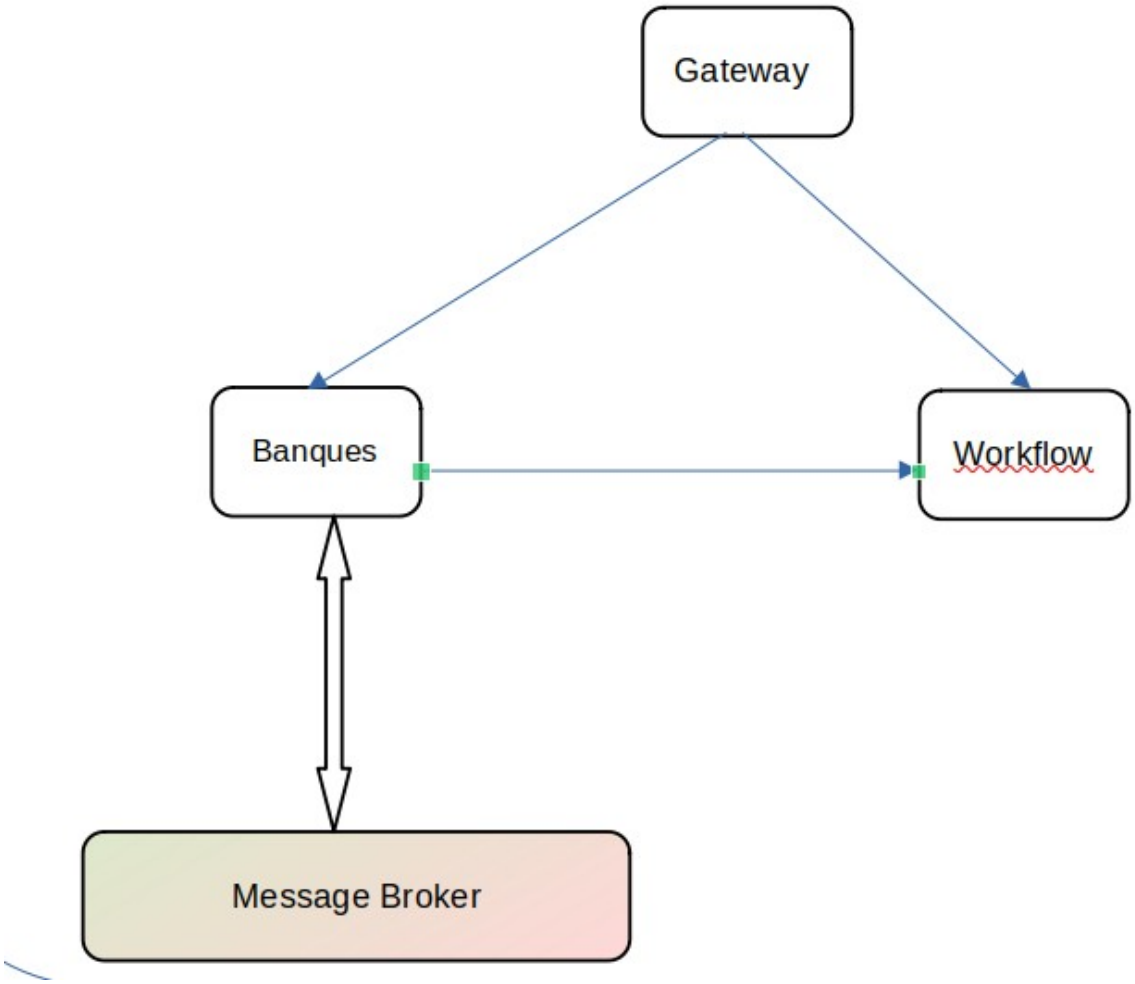
Exemple :services SpringBoot 3

Objectifs

- Expliciter certains starters de SpringBoot3
- problématique d’environnement de dev des micro-services
- Pratiques de test

Table des matières

Architecture



Workflow-service

Un micro-service « technique » offrant une API avec un moteur de workflow (machine à état simple) destiné à gérer les processus de traitement des dossiers de l'entreprise.

Répertoire : ***exemple/workflow-service***

Starters et dépendances

Développement :

- ***dev-tools*** : Dans l'environnement STS, redémarre automatiquement l'application à chaque modification, influe également sur les configurations des autres starters.
  Dans un environnement IntellijIdea, plus difficile à faire fonctionner
- ***configuration-processor*** : Dans l'environnement STS, permet la vérification et la complétion des propriétés applicatives encapsulées dans un bean @ConfigurationProperties
- ***lombok*** : Support pour annotations lombok
- ***docker-compose*** : Démarrage automatique de services de support via des containers.

Stack :

- ***data-mongodb-reactive*** : Spring Data pour Mongo en mode réactif
- ***starter-webflux*** : Couche Web reactive au dessus de Netty
- ***springdoc-openapi-starter-webflux-ui*** : Génération de la documentation OpenAPI V3 à partir du code source SpringBoot3

Test :

- ***starter-test*** : Annotations SpringBootTests et autres. Dépendances de base JUnit5, AssertJ, JsonAssert, Mockito, …
- ***reactor-test*** : Test des publisher reactif de Spring Reactor avec la classe StepVerifier
- ***testcontainers.junit-jupiter***, ***testcontainers.mongodb***, ***spring-boot-testcontainers*** : Dépendances permettant de démarrer un container Mongo durant les tests

Ops :

- ***actuator*** : Exposition des points d'observabilité du services

Revue de code

Domaine : L'agrégat du domaine est la classe ***Workflow*** qui embarque une liste de ***State*** et une liste de ***Transition***.
L'ID est géré par l'application.
Une interface ReactiveMongoRepository permet d'avoir à disposition toutes les opérations CRUD.
Des annotations Swagger ont été ajoutés pour parfaire l'interface swagger-ui.
Des annotations lombok génèrent les constructeurs, les getters/setters et le builder.

Service : La classe service encapsule toutes ses valeurs de retour dans des Publisher et s'appuie sur la classe Repository

La couche web expose les méthodes des services et propose les endpoints :

- ***GET /workflows*** : Liste tous les process de la base
- ***POST /workflows*** : Création d'un process
- ***DELETE /workflows*** : Suppression d'un process.
- ***GET /workflows/{id}/actions?state= ?*** : Les actions possibles en fonction d'un état

- ***POST workflows/{id}?action= ?*** : Effectue une transition de la machine à état et renvoie une classe *DomainEvent* encapsulant l'ancien statut et le nouveau statut du dossiers

Tests

Tests HTTP de bout en bout
Le test met en jeu tous les beans de l'application.
Il utilise *WebTestClient* qui encapsule *WebClient* mais qui expose une façade de test pour vérifier les réponses.
Durant le test un container MongoDb tout *neuf* est démarré grâce aux annotations du starter ***testcontainers***

Test d'intégration de la couche de persistance.
Seuls les beans nécessaires à la persistance sont chargés
Il utilise :

- l'annotation d'auto-configuration ***@DataMongoTest*** pour limiter le contexte Spring chargé.
- Les annotations de TestContainers pour démarrer un container Mongo pendant le test
- La classe utilitaire *StepVerifier* qui permet de faire des assertions sur des Publisher (*reactor-test*)

Démarrer les tests

- Dans votre IDE
- `via ./mvnw clean test`

Exécution dans l'IDE

Au démarrage un container Mongo persistant est démarré, il est configuré dans le fichier ***compose.yaml***

L'API est accessible à http://localhost:8081/swagger-ui.html

Vous pouvez également visualiser les liens *actuator*.

Packaging

2 formats de packaging. Pour ces 2 packages, la connexion à Mongo doit être précisé au moment du démarrage.

un docker-compose.yaml permet de démarrer au préalable un service Mongo :

```
docker-compose up -d
```

Jar

Obtenu via :

```
./mvnw clean package
```

Démarré localement par :

```
java -jar target/workflow-service-0.0.1-SNAPSHOT.jar \
--spring.profiles.active=prod
```

Docker

Obtenu via :

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

Démarré localement par

```
docker run --network host workflow-service:0.0.1-SNAPSHOT \
-e SPRING_PROFILES_ACTIVE=prod
```

Release

L’image docker a été poussé sur DockerHub :

dthibau/bceao-workflow-service:1.0

Banque-service

C’est un service métier dédier à la gestion d’établissement bancaire.

Il s’appuie sur le service précédent pour faire évoluer les statuts des établissements bancaires.

Il publie systématiquement ces changements de statut sur un topic Kafka

Répertoire : ***exemple/banques-service***

Starters et dépendances

Développement :

- Idem service précédent

docker-compose est utilisé pour démarrer une base PostgresSQL

Stack :

- ***postgresql-driver : Driver postgres***
- ***data-jpa*** : Spring Data pour JPA
- ***kafka*** : Gestion de topic, Production et réception de message
- ***starter-web*** : Couche Web servlet au dessus de Tomcat
- ***springdoc-openapi-starter-web-ui*** : Génération de la documentation OpenAPI V3 à partir du code source SpringBoot3

Test :

- ***starter-test*** : Annotations SpringBootTests et autres. Dépendances de base JUnit5, AssertJ, JsonAssert, Mockito, …
- ***kafka-test*** : Broker Kafka embarqué, utilitaire pour la vérification de la production/consommation de message
- ***testcontainers.junit-jupiter***, ***testcontainers.postgresql***, ***testcontainers.kafka***, ***spring-boot-testcontainers*** : Dépendances permettant de démarrer les containers Postgres et Kafka durant les tests
- ***contract-verifier*** : Pour la génération automatique des tests et l’approche DesignByContract

Ops :

- ***actuator*** : Exposition des points d’observabilité du services

Plugin de build :
Le plugin ***spring-cloud-contract-maven-plugin*** permet de générer des tests à partir de contrat et de publier le contrat dans un dépôt Maven
Le plugin ***native-maven-plugin*** permet de générer du code natif

Revue de code

Domaine : L’agrégat est la classe Banque qui embarque la valeur Adresse.
Un champ *@Version* permet une gestion optimiste des transactions concurrentes.
La classe Banque applique le pattern *DomainModel* : méthodes *createBanque* et *updateBanque*.
Normalement de logique métier pourrait être implémenté dans ces méthodes.

Service : La classe *BanqueService* s’appuie sur 3 dépendances :

- *BanqueRepository* : Opérations de persistance

- *WorkflowService* : Adaptateurs vers le service de workflow (Interaction REST). La structure DomainEvent est partagée entre les 2 services
- *EventService* : Adaptateurs vers le broker Kafka

Il applique le pattern *Domain Event* en publiant chaque changement d'état d'une Banque vers un topic.

Toutes ses méthodes s'exécutent dans une transaction BD

DTO : Classes d'échanges entre la couche web et la couche Services

La couche web expose les méthodes des services et propose les endpoints :
- ***GET /api/${api.version}/banques*** : Les banques en base
- ***GET /api/${api.version}/banques/{id}*** : Chargement d'une banque par son id
- ***POST /api/${api.version}/banques*** : Création de banque
- ***PUT /api/${api.version}/banques*** : Effectuer une action invoquant une mise à jour

La configuration applicative
- *API Config* : Pour la gestion de l'API en particulier sa version
- *KafkaConfig* : L'intégtation avec le broker
- *WorkflowConfig* : L'intégration avec le service de Workflow

Tests

Les tests sont générés à partir de contrats exprimés en Groovy, cette fonctionnalité est apporté par ***SpringCloudContract***

3 contrats sont présents permettant de tester différents cas de la création d'un établissement.

La commande suivante permet de générer et exécuter les tests :

```
./mvnw clean test
```

Les tests générés étendent la classe ***BaseTestClass*** qui démarre les containers Postgres et Kafka, mock le service *workflow-service* et configure un environnement MockMVC

Ce sont des tests composants qui teste l'intégralité du service en isolant ses dépendances

Une fois les tests passés, nous pouvons installer les contrats avec

```
./mvnw clean install
```
En plus de publier l'artefact vers le dépôt local Maven, les contrats sont également publiés, ils serviront aux tests de composants des consommateurs de ce service.

Exécution dans l'IDE

Pour exécuter dans l'ide, il est nécessaire de démarrer les dépendances.

Un fichier docker-compose permet de démarrer :

- Le service *workflow-service* avec une base mongo initialisée. On s'appuie sur la version docker généré précédemment
- Un cluster 1 nœud Kafka
- L'outil d'administration RedPanda permettant de visualiser les topics

Démarrer les dépendances avec

```
cd dependencies
docker-compose up -d
```

Vous pouvez vérifier la disponibilité de workflow-service à *http://localhost:8081/swagger-ui.html*

Le démarrage dans l'IDE démarre automatiquement un container Postgres défini par ***compose.yaml.***

L'API est alors disponible à *http://localhost:8080/swagger-ui.html*

Packaging

L'application supporte les types de packaging du service précédent + un packaging natif permettant des démarrages plus rapide.

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

Nous utilisons la 1ère possibilité (l'application a été downgradée en 3.2.1 car il existe un bug sur la 3.2.2)

```
./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.

```
docker-compose up -d
docker run --network host banques-service:0.0.1-SNAPSHOT
```

Gateway

Le service gateway est le point unique d'entrée d'une application front-end. Il effectue du routing direct vers le service métier banque-service et également fait de la composition d'API

Répertoire : ***exemple/gateway***

Starters et dépendances

Développement :

- Idem service précédent

Stack :

- ***spring-cloud-gateway :*** Suport pour acheminer les requêtes vers les services métier et d'implémenter les cross-cuting concernc comme la sécurité, la surveillance/les métriques et la résilience. Construit au dessus de SpringWebFlux

Test :

- ***starter-test*** : Annotations SpringBootTests et autres. Dépendances de base JUnit5, AssertJ, JsonAssert, Mockito, …
- ***contract-stub-runner*** : Pour la génération automatique de Mock Serveur à partir de contrat publié dans un dépôt Maven

Ops :

- ***actuator*** : Exposition des points d'observabilité du services

Revue de code

1 seul contrôleur effectuant de la composition d'API en mode réactif:

- 1 requête vers banques service pour récupérer tous les établissements bancaires
- Pour chaque établissement, récupérer les actions possibles définie dans *workflow-service*

Configuration des routes :

2 routes sont configurées dans *application.yml* permettant d'atteindre banque-service

Test

2 tests sont fournis :

Le premier mock les 2 services dépendant

Le second profite du contrat de banque-service précédemment publié dans Maven pour exécuter son test

Exécuter les tests :

```
./mvnw test
```

Packaging

Supporte jar et docker

Architecture complète

Vous pouvez démarrer l’ensemble des services via le fichier *exemple/docker-compose.yml*

```
docker-compose up -d
```

Essayer d’utiliser l’application via la gateway