Ateliers

Spring5 / Programmation réactive

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

- JDK17
- Git
- Docker
- Spring Tools Suite, avec Lombok
- Apache JMeter

Solutions :

- `https://github.com/dthibau/spring5-solutions.git`

Table des matières

Atelier 1: Reactor

1.1 Reactor API

Créer un projet Maven (ou gradle)

Ajouter juste les dépendances suivantes :

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>{reactor-version}</version>
</dependency>

<dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>{latest}</version>
</dependency>
```

Créer une classe Main

1.1.1 Méthode1 : Lambda Expression

Dans une première méthode,

- créer un *Flux* qui publie 10 entiers,
- un abonné sous forme de lambda expression qui ajoute chaque événement dans une liste, utiliser l'opérateur *log()* sur le flux pour voir la séquence des événements.

1.1.2 Méthode 2 : Classe Subscriber

Obtenir le même résultat en utilisant une classe interne *Subscriber*

1.1.3 Méthode 3 : Back-pressure

Modifier ensuite le *Subscriber* afin qu'il demande les événements 2 par 2

1.1.4 Chaînage d'opérateurs

Utiliser la chaîne d'opérateur suivante :

- Multiplier chaque entier par 3
- Filtrer les nombres impairs
- Pour chaque entier, générer sa valeur et son opposé
- Logger
- Faire la somme

Récupérer le résultat et l'afficher.

1.1.5 Combinaison de flux

Combiner 2 flux d'entier avec *zipWith*

1.1.6 Temporisation

Générer un flux générant un compteur toutes les secondes, afficher les 5 premiers éléments

1.2 Threads

Reprendre un des exemples précédent et utiliser la méthode ***subscribeOn*** sur un Scheduler.
Observer les effets sur les logs.

Ajouter dans la chaîne (au milieu de chaîne) un appel à ***publishOn***
Observer les effets sur les logs.

1.3 Gestion des erreurs

Reprendre les exemples précédents.

Ajouter aux flux produit un événement erreur

1.3.1 Comportement par défaut
Observer le comportement par défaut.

1.3.2 Méthode onError
Implémenter ensuite la méthode *onError()* dans l'abonné

1.3.3 Méthode onResume
Enfin, utiliser une méthode *onErrorResume* pour générer un Flux de fallback en cas d'erreur

1.3.4 Retry
Donner une chance en donnant un autre essai !

1.4 Test et debug

Ajouter les dépendances ***JUnit5*** et ***reactor-test***
Écrire une classe de test vérifiant la séquence d'événements d'une des méthode précédente.

Mettre en place le debug et ré-exécuter les exemples précédents avec les événements d'erreur

1.5 Contexte

Écrire une pipeline :

- générant 10 entiers
- les transformant en 10 String, en préfixant chaque entier par une valeur lue dans le contexte

Utiliser un pool de threads pour exécuter 5 fois cette pipeline
Vérifier que la valeur lue dans le contexte et bien propagée

Atelier 2 : Spring Data Réactive

2.1 Reactive Mongo DB

Démarrer un Spring Starter Project et choisir les starters ***reactive-mongodb, embedded Mongo*** et ***lombok***

Récupérer la classe modèle fournie

2.1.1 Classe Repository

Créer une interface Repository héritant de `ReactiveMongoRepository<Account, String>`

Définir 2 méthodes réactives :

- Une méthode permettant de récupérer toutes les classes *Account* via leur attribut *amount*
- Une méthode permettant de récupérer la première classe *Account* via l'attribut *owner*

Implémenter une classe de test qui vérifie l'ajout d'instances *Account* dans la base, ainsi que les requêtes définies par la classe *Repository*

2.1.2 ReactiveMongoTemplate

Créer une classe de configuration créant un bean de type *ReactiveMongoTemplate* comme suit :

```
@Configuration
public class ReactiveMongoConfig {

    @Autowired
    MongoClient mongoClient;

    @Bean
    public ReactiveMongoTemplate reactiveMongoTemplate() {
        return new ReactiveMongoTemplate(mongoClient, "test");
    }
}
```

Créer ensuite une classe Service exposant une interface métier de gestion des classes *Account* et utilisant le template.

Écrire une classe de test de ce Service à partir de la classe principale.

2.2 R2DBC

Démarrer un Spring Starter Project et choisir les starters ***SpringData-r2dbc*** et ***H2***

Reprendre la classe modèle *Account* de l'atelier précédent et la modifier si besoin.

Reprendre également le fichier d'initialisation de la base ***src/main/resources/schema.sql***

2.2.1 Classe Repository

Ajouter un bean permettant de créer des Connexions vers la base :

```
    @Bean
    ConnectionFactoryInitializer initializer(ConnectionFactory connectionFactory) {

        ConnectionFactoryInitializer initializer = new ConnectionFactoryInitializer();
        initializer.setConnectionFactory(connectionFactory);
        initializer.setDatabasePopulator(new ResourceDatabasePopulator(new
ClassPathResource("schema.sql")));

        return initializer;
    }
```

Créer une interface Repository héritant de `ReactiveCrudRepository<Account, Long>`

Définir 2 méthodes réactives :

- Une méthode permettant de récupérer toutes les classes *Account* via leur attribut *amount*
- Une méthode permettant de récupérer la première classe *Account* via l'attribut *owner*

Implémenter une classe de test qui vérifie l'ajout d'instances *Account* dans la base, ainsi que les requêtes définies par la classe *Repository*

2.2.2 R2DBCEntityTemplate

Créer un bean Service exposant une interface métier de gestion des classes *Account* et utilisant un *R2DBCEntityTemplate*.

Les méthodes de service exposeront :

- `public Mono<Account> findById(String id) ;`
- `public Flux<Account> findAll() ;`
- `public Mono<Account> save(Account account) ;`
- `public Mono<Account> first() ;`

Implémenter une classe de test qui teste ces méthodes.

Atelier 3 : Spring Web Flux

3.1. Approche Contrôleur

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

3.2 Endpoint fonctionnels

L'objectif est d'offrir une API Rest pour la gestion de la base Mongo du TP précédent

Reprendre le TP précédent et y ajouter le starter WebReactif

Créer une classe *Handler* regroupant les méthodes permettant de définir les *HandlerFunctions* suivantes :

- « *GET /accounts* » : Récupérer tous les *accounts*
- « *GET /accounts/{id}* » : Récupérer un *account* par un id
- « *POST /accounts* » : Créer un *account*

Créer la classe de configuration *WebFlux* déclarant les endpoints de notre application.

Utiliser le script JMeter fourni pour tester votre implémentation.

Atelier 4 : WebClient

4.1. 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 un ***Webclient*** qui utilise le service REST du TP précédent.

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

Atelier 5a – Server-side events

https://www.baeldung.com/spring-server-sent-events

https://stackoverflow.com/questions/51370463/spring-webflux-flux-how-to-publish-dynamically

Atelier 5 : WebSockets

5.1. 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 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. Protection des URLs

Reprendre le projet du TP3 offrant l'API Rest

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 http-basique
- Il faut avoir le rôle *ADMIN* pour créer un *Account*

6.2 Protection des méthodes

Activer la protection des méthodes

Protéger la méthode permettant de récupérer un *Account* particulier en exigeant le rôle *ADMIN*