Programmation réactive

1 Découvrir la programmation réactive

Afin d'illustrer la programmation réactive, nous allons utiliser le framework reactor.

⏩ Travail à faire :

- Ajouter les dépendances MAVEN suivantes :

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
</dependency>
```

- Créer un package `myboot.app6`.
- Créer un package `myboot.app6.test`.

1.1 Traitement d'une donnée unique

Dans la programmation réactive, les actions sont toutes asynchrones et déclenchées par l'arrivée d'une donnée à traiter. Cette donnée peut être unique (ou absente) ou bien multiple. Dans Reactor, les données uniques sont codées par la classe `Mono`. Voici un exemple

```
package myboot.app6.test;

import org.junit.jupiter.api.Test;

import reactor.core.publisher.Mono;

public class testReactor {

    @Test
    public void testMono() {
        System.out.println("-- test Mono");
        Mono.just(10)//
                .subscribe(System.out::println);
    }

}
```

L'installation d'un consommateur de donnée (avec la méthode `subsribe`) active le traitement et la donnée `10` est transmise à `println`.

Travail à faire :

- Lire la documentation de `Mono` et tester la méthode `log`.

```
...
        Mono.just(10)//
                .log()//
                .subscribe(System.out::println);
...
```

- La méthode `block` termine la chaîne des traitements asynchrones et renvoie la valeur résultante. C'est donc une action bloquante. Tester cette possibilité.
- Nous pouvons maintenant appliquer un traitement (asynchrone) sur cette donnée avec la méthode `map` :

```
...
        Mono.just(10)//
                .map(i -> i + 10)//
                .subscribe(System.out::println);
...
```

- Nous pouvons aussi faire en sorte que le traitement change le type de la donnée transmise :

```
...
        Mono.just(10)//
                .map(v -> v + 10)//
                .map(v -> "Message␣" + v)//
                .map(v -> v.toUpperCase())//
                .subscribe(System.out::println);
...
```

- Nous pouvons nous abonner à des événements intéressants. Essayer les méthodes `doFirst`, `doOnNext` et `doAfterTerminate`.

1.2 Traitement de données multiples

Dans Reactor, les données multiples sont traitées par la classe `Flux`. Voici trois exemples :

```
Flux.just(1, 2, 3)//
        .subscribe(System.out::println);

Flux.range(1, 10)//
        .subscribe(System.out::println);

Mono.just(5)//
        .repeat(10)//
        .subscribe(System.out::println);
```

⏩ Travail à faire :

- Tester la méthode `filter`. Utiliser les méthodes bloquantes `blockFirst` et `blockLast` pour développer des tests unitaires. Utiliser ensuite `collectList` et `block`.
- Tester les méthodes `all` et `any`. Utiliser la méthode `block` pour développer des tests unitaires.
- Utiliser ensuite `sort` et `distinct` pour filtrer un flux.
- Agrandir ensuite un flux avec `concatWith` et `concatWithValues`.
- Ajouter un traitement (`map`) qui va freiner le flux pour observer les effets sur les étapes suivantes.
- Avec (`flatMap`) ajouter la production d'un flux.
- Tester la création d'un *buffer* avec `buffer(n)`, `bufferUntil` et `bufferUntilChanged`
- Avec `zipWith` créer un flux qui résulte du traitement de deux autres flux. Quel est le résultat si les flux n'ont pas la même taille ?
- Avec `zip` (méthode statique) agréger deux (ou trois) autres flux. Quel est le résultat ?
- Utiliser les méthodes `then` et `thenMany` pour enchaîner la fin des traitements d'un Flux avec d'autres traitements sur un autre flux.

1.3 Traitement des erreurs

⏩ Travail à faire :

- Générer un flux d'entiers entre 1 et 20. Ajouter un traitement qui va générer un erreur sur les entiers entre 10 et 15. Quel est le résultat ?
- Utiliser la méthode `onErrorContinue` pour capter l'erreur.

2 Utiliser Spring Data avec Redis

Travail à faire :

- Commencez, dans une autre fenêtre, à charger et à compiler la dernière version de la BD NoSql Redis (à partir des sources).
- Créez le package `myboot.app6.repo` .
- Ajoutez la dépendance ci-dessous :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

- Enrichissez la configuration avec l'activation de **Spring-Data-Redis** :

```
package myboot.app6.repo;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.repository.configuration.EnableRedisRepositories;

import myboot.Starter;

@Configuration
@EnableRedisRepositories(basePackageClasses = Starter.class)
public class ConfigRedis {

}
```

- Créez un POJO afin de modéliser la données à sauvegarder :

```
package myboot.app6.repo;

import org.springframework.data.annotation.Id;
import org.springframework.data.redis.core.RedisHash;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@RedisHash("Student")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Student {

    public enum Gender {
        MALE, FEMALE
    }

    @Id
    private String id;
    private String name;
    private Gender gender;
    private int grade;
}
```

- Définissez ensuite le dépôt Spring-Data :

```
package myboot.app6.repo;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StudentRepository extends CrudRepository<Student, String> {
}
```


- Testez ce composant avec le code ci-dssous :