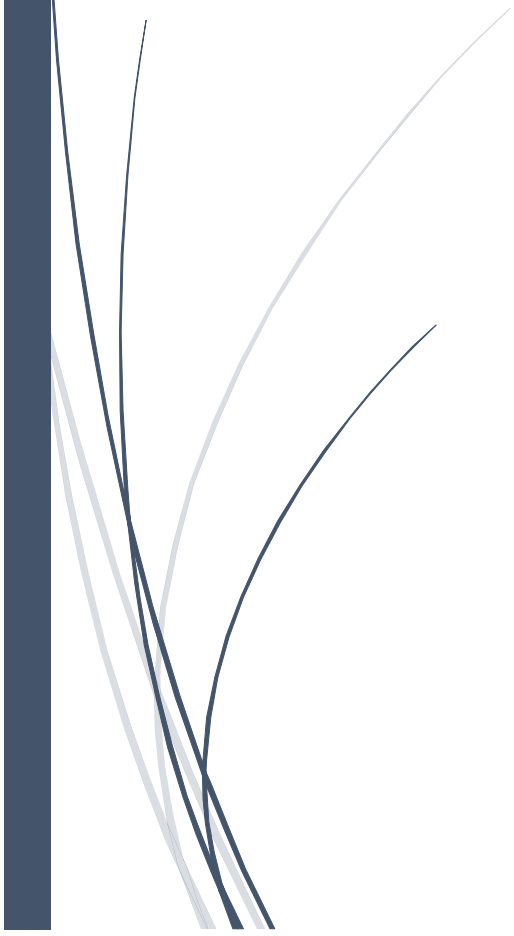
La programmation réactive

TP : Réaliser une application réactive avec Spring Boot, Spring WebFlux, Spring Data R2DBC , H2 et Thymeleaf



Architecture des composants d’entreprise

Table des matières

I. Objectif du TP

- Expliquer la programmation réactive.
- Expliquer les concepts de base de la programmation réactive.
- Expliquer les classes principales de Spring WebFlux.
- Développer une application réactive (couche Backend) avec Spring Boot, Spring WebFlux, Spring Data R2DBC et H2.
- Développer une application réactive (couche Frontend) avec Spring Boot, Spring WebFlux et Thymeleaf.

II. Prérequis

- IntelliJ IDEA ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.

NB : *Ce TP a été réalisé avec IntelliJ IDEA 2023.2.3 (Ultimate Edition).*

III. La programmation Réactive

a. C'est quoi la programmation Reactive ?

- La programmation réactive est un paradigme de programmation qui favorise une approche **asynchrone**, **non bloquante** et axée sur les événements du traitement des données.

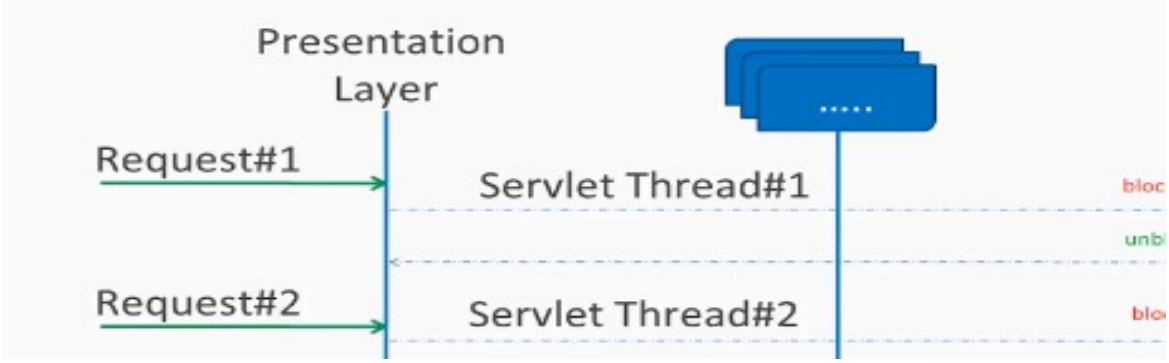
- La programmation réactive consiste à modéliser les données et les événements en tant que flux de données observables et à mettre en œuvre des routines de traitement des données pour réagir aux modifications de ces flux.

- La programmation réactive est basée sur le Design Pattern « Publisher-Subscriber ». Dans la programmation réactive, nous faisons une requête et commençons à effectuer d'autres choses. Lorsque les données sont disponibles, nous recevons la notification avec les données dans la ***callback function.*** La ***callback function*** gère la réponse en fonction des besoins de l'application/l'utilisateur.

- Pour construire une application véritablement non bloquante, il faut veiller à créer/utiliser **tous ses composants en tant que composants non bloquants**, c'est-à-dire le client, le contrôleur, les services intermédiaires et même la base de données. Si l'un d'entre eux bloque les requêtes, l'objectif ne sera pas atteint.

b. Traitement des requêtes bloquantes ou non bloquantes

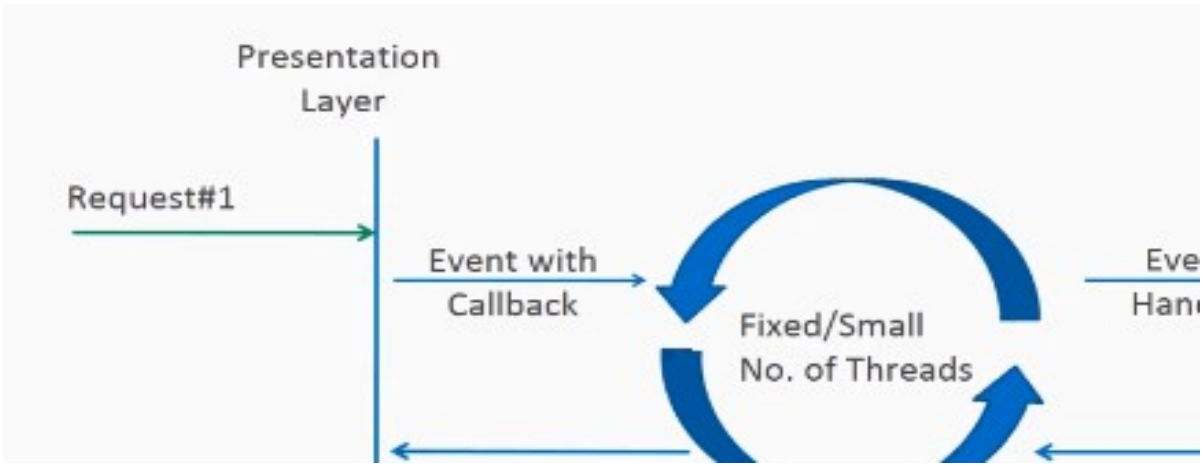
1. Requêtes bloquantes



- Dans les applications MVC traditionnelles, un nouveau thread de servlet est créé (ou obtenu à partir du pool de threads) lorsqu'une demande arrive au serveur. Il délègue la demande aux threads de travail pour les opérations d'E/S telles que l'accès à la base de données, etc. Pendant la période d'occupation des threads de travail, le thread de servlet (thread de requête) reste en attente et est donc bloqué. On l'appelle aussi traitement synchrone des requêtes.

- La multiplication des Thread peut entraver les performances et limiter l'utilisation complète de la capacité du serveur.

2. Requêtes non bloquantes



- Dans le traitement des requêtes non bloquantes ou asynchrones, aucun thread n'est en attente. Il n'y a généralement **qu'un seul thread** de requête reçevant la requête.

- Toutes les demandes entrantes sont accompagnées d'un ***event handler*** et d'une ***callback function***. Le thread de requête délègue les requêtes entrantes à un pool de threads (généralement un petit nombre de threads) qui délègue la requête à son ***handler function*** et commence immédiatement à traiter les autres requêtes entrantes à partir du thread de requête.

- Lorsque la ***handler function*** est terminée, un thread du pool collecte la réponse et la transmet à la ***callback function***.

c. L'api « Reactive Streams »

La nouvelle API « ***Reactive Streams*** » a été créée par des ingénieurs de Netflix, Pivotal, Lightbend, RedHat, Twitter et Oracle, entre autres, et fait désormais partie de Java 9. Il définit quatre interfaces : ***Publisher***, ***Subscriber***, ***Subscription*** et ***Processor***.

1. Publisher

- Un ***Publisher*** émet une séquence d'événements aux abonnés en fonction de la demande reçue de ses abonnés. Un ***Publisher*** peut servir plusieurs abonnés.

```
Publisher.java

public interface Publisher<T> {
  public void subscribe(Subscriber<? super T> s);
```

2. Subscriber

- Un ***Subscriber*** reçoit et traite les événements émis par un ***Publisher***.
- Aucune notification ne peut être reçue jusqu'à ce que Subscription#request(long) est appelée pour signaler la demande.

```
Subscriber.java

public interface Subscriber<T> {

  public void onSubscribe(Subscription s);
  public void onNext(T t);
  public void onError(Throwable t);
```

3. Subscription

- Défini une relation one-to-one entre le ***Publisher*** et le ***Subscriber***.
- Il est utilisé une seule fois par un ***Subscriber***.
- Il est utilisé pour exprimer une demande ou bien annuler une demande.

```
Subscription.java

public interface Subscription<T> {
  public void request(long n);
  public void cancel();
```

4. Processor

- Un **Processor** est un **Stage** dans un flux réactif qui à la fois **consomme** et **produit** des éléments de données. Il fonctionne comme un **intermédiaire** entre un **Publisher** (qui produit des éléments) et un **Subscriber** (qui consomme ces éléments).

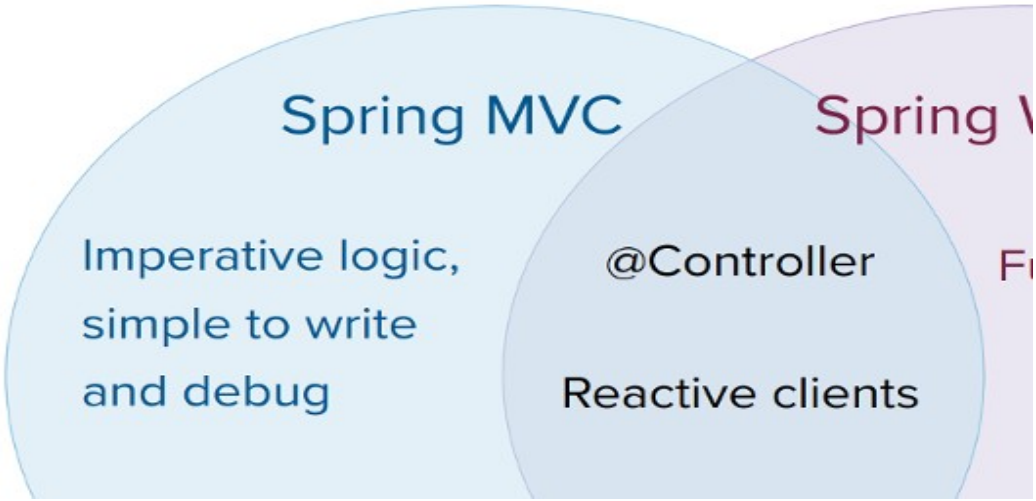
- Un **Processor** dans l'API Reactive Streams permet de gérer la transformation des données tout en maintenant le contrôle de flux sur ces données, en agissant comme un lien entre le ***Publisher*** qui produit des données et le ***Subscriber*** qui les consomme.

```
Processor.java

public interface Processor<T, R> extends Subscriber<T>, Publisher
}
```

IV. C'est quoi Spring WebFlux ?

a. Spring WebFlux



- ***Spring WebFlux*** est une version parallèle de Spring MVC et prend en charge les flux réactifs entièrement non bloquants. Il prend en charge le concept de ***backpressure*** et utilise ***Netty*** comme serveur intégré pour exécuter des applications réactives.

- Le ***backpressure*** dans Spring WebFlux permet d'assurer la fluidité et l'efficacité du flux de données entre producteurs et consommateurs. Il empêche les consommateurs d'être submergés par des données trop nombreuses ou trop rapides et il permet de contrôler la vitesse à laquelle les données sont traitées.

- *Spring WebFlux* utilise le projet *Reactor* comme librairie native. *Reactor* est une librairie Reactive Streams. Toutes les fonctionnalités de ***Spring WebFlux*** supportent le ***non-blocking back pressure***.

- ***Spring WebFlux*** propose deux classes *Publisher* : ***Mono*** et ***Flux***.

1. La classe Mono

- Retourne 0 ou 1 élément.

```
Mono<String> mono = Mono.just("Alex");
Mono<String> mono = Mono.empty();
```

Ci-après, les méthodes principales fournies par la classe Mono :

Méthodes de création :

- ***Mono.just(T value)*** : Crée un Mono contenant un élément donné (valeur non-null).
- ***Mono.empty()*** : Crée un Mono qui ne contient aucun élément (équivalent à Optional.empty()).
- ***Mono.error(Throwable error)*** : Crée un Mono qui émet une erreur.
- ***Mono.fromCallable(Callable<? extends T> callable)*** : Crée un Mono à partir d'une Callable.
- ***Mono.defer(Supplier<? extends Mono<? extends T>> supplier)*** : Crée un Mono en retard, uniquement lorsque l'abonnement se produit, ce qui permet de différer l'exécution de la logique.
- ***Mono.from(Mono<? extends T> publisher)*** : Crée un Mono à partir d'un autre publisher (par exemple, un autre Mono ou un Flux).
- ***Mono.delay(Duration duration)*** : Crée un Mono qui émet un élément après un délai donné.

Opérations de transformation :

- ***map(Function<? super T, ? extends V> mapper)*** : Applique une transformation à l'élément émis par le Mono si présent.
- ***flatMap(Function<? super T, ? extends Publisher<? extends V>> mapper)*** : Applique une transformation qui retourne un autre Publisher, permettant ainsi de chaîner plusieurs opérations asynchrones.
- ***flatMapMany(Function<? super T, ? extends Publisher<? extends R>> mapper)*** : Transformation similaire à flatMap mais pour retourner un Flux au lieu d'un Mono.
- ***filter(Predicate<? super T> predicate)*** : Filtre l'élément émis en fonction d'un prédicat.
- ***switchIfEmpty(Mono<? extends T> other)*** : Si le Mono est vide (ne contient pas de valeur), émet un autre Mono passé en argument.

Opérations de combinaison :

- ***zipWith(Mono<? extends V> other)*** : Combine les éléments de deux Mono en une paire (tous les Mono doivent émettre exactement un élément).
- ***mergeWith(Publisher<? extends T> other)*** : Combine les éléments de ce Mono avec un autre Publisher (émettant potentiellement plusieurs éléments).
- ***concatWith(Publisher<? extends T> other)*** : Enchaîne deux Publisher, en émettant les éléments du premier avant ceux du second.
- ***combineLatest(Mono<? extends V> other, BiFunction<? super T, ? super V, ? extends R> combinator)*** : Combine les éléments des deux Mono lorsque tous les deux ont émis une valeur.

Gestion des erreurs :

- ***onErrorResume(Function<? super Throwable, ? extends Mono<? extends T>> fallback)*** : Si une erreur est émise, permet de définir un Mono de remplacement.
- ***onErrorReturn(T fallback)*** : Si une erreur se produit, émet une valeur de remplacement.
- ***doOnError(Consumer<? super Throwable> onError)*** : Exécute une action lorsque le Mono rencontre une erreur.
- ***retry(long maxRetries)*** : Réessaie l'exécution du Mono en cas d'erreur, jusqu'à un nombre d'essais spécifié.

Exécution et souscription

- ***subscribe()*** : Permet de souscrire à un Mono sans effectuer d'opération supplémentaire.
- ***subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Runnable onComplete)*** : Permet de souscrire en spécifiant les actions pour chaque étape du cycle de vie (onNext, onError, onComplete).
- ***block()*** : Attends de manière synchrone le résultat du Mono. Cette méthode est bloquante et ne doit être utilisée que dans des scénarios où l'on veut explicitement bloquer le thread.
- ***blockOptional() :*** Renvoie un Optional<T>, soit avec la valeur émise, soit vide si aucun élément n'a été émis.

2. La classe Flux

Renvoie 0... N éléments. Un Flux peut être infini, ce qui signifie qu'il peut continuer à émettre des éléments pour toujours. En outre, il peut renvoyer une séquence d'éléments, puis envoyer une notification d'achèvement lorsqu'il a renvoyé tous ses éléments.

```
Flux<String> flux = Flux.just("A", "B", "C");
Flux<String> flux = Flux.fromArray(new String[]{"A", "B", "C"});
Flux<String> flux = Flux.fromIterable(Arrays.asList("A", "B", "C"
//To subscribe call method
```

Ci-après, les méthodes principales fournies par la classe Flux :

Méthodes pour créer un nouveau flux :

- just(T... data) : Crée un Flux à partir d'une ou plusieurs valeurs.

```
Flux<String> flux = Flux.just("
```

- fromIterable(Iterable<? extends T> source) : Crée un Flux à partir d'une collection.

```
Flux<Integer> flux = Flux.fromIterable(Arr
```

- fromArray(T[] array) : Crée un Flux à partir d'un tableau.

```
Flux<Integer> flux = Flux.fromStream(S
```

- fromStream(Stream<? extends T> stream) : Crée un Flux à partir d'un flux Java.

```
Flux<Integer> flux = Flux.fromStream(St
```

- empty() : Crée un Flux vide.

```
Flux<Object> flux = Flu
```

- error(Throwable e) : Crée un Flux qui émet une erreur.

```
Flux<Object> flux = Flux.error(new RuntimeException("Error occurred"));
```

- never() : Crée un Flux qui ne produit jamais de données ni ne se termine.

```
Flux<Object> flux = Flux.never();
```

- interval(Duration period) : Émet des éléments à des intervalles réguliers.

```
Flux<Long> flux = Flux.interval(Duration.ofSeconds(1));
```

Méthodes pour manipuler les éléments du flux :

- map(Function<? super T, ? extends R> mapper) : Transforme les éléments.

```
Flux<String> flux = Flux.just("A", "B", "C").map(String::toLowerCase);
```

- flatMap(Function<? super T, ? extends Publisher<? extends R>> mapper) : Aplatie les éléments en flux.

```
Flux<String> flux = Flux.just("A", "B", "C")
```

- filter(Predicate<? super T> predicate) : Filtre les éléments selon une condition.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4).fi
```

- distinct() : Élimine les doublons.

```
Flux<Integer> flux = Flux.just(1, 2, 2, 3).distinct();
```

- take(long n) : Prend les n premiers éléments.

```
Flux<Integer> flux = Flux.just(1,
```

- skip(long n) : Ignore les n premiers éléments.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4).skip(2);
```

Méthodes terminales et de souscription :

- subscribe(Consumer<? super T> consumer) : Souscrit et consomme les éléments.

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

- collectList() : Agrège les éléments dans une liste.

```
Mono<List<Integer>> mono = Flux.just(1,
```

- reduce(BinaryOperator<T> accumulator) : Réduit les éléments en une valeur unique.

```
Mono<Integer> mono = Flux.just(1, 2, 3).reduce(Integer::sum);
```

Méthodes pour gérer les erreurs :

- onErrorResume(Function<? super Throwable, ? extends Publisher<? extends T>> fallback) : Fournit un flux alternatif en cas d'erreur.

```
Flux<String> flux = Flux.error(new RuntimeExc
```

- onErrorReturn(T fallbackValue) : Renvoie une valeur par défaut en cas d'erreur.

```
Flux<String> flux = Flux.error(new Runtime
```

Méthodes pour combiner les flux :

- mergeWith(Publisher<? extends T> other) : Combine deux flux en parallèle.

```
Flux<Integer> flux = Flux.just(1, 2).mergeW
```

- concatWith(Publisher<? extends T> other) : Combine deux flux de manière séquentielle.

```
Flux<Integer> flux = Flux.just(1, 2).concatW
```

- zip(Publisher<? extends T> other, BiFunction<? super T1, ? super T2, ? extends V> combinator)
  - La méthode ***zip*** de la classe ***Flux*** permet de combiner plusieurs flux en un seul flux en regroupant les éléments émis correspondants.
  - La méthode zip applique une fonction pour produire un nouveau flux avec les valeurs combinées.

Quelques exemples :

```java
import reactor.core.publisher.Flux;

public class ZipExample {
    public static void main(String[] args) {
        Flux<String> flux1 = Flux.just("A", "B", "C");
        Flux<Integer> flux2 = Flux.just(1, 2, 3);

        Flux<String> combined = Flux.zip(flux1, flux2, (lett
```

Exemple de flux de durées différentes :

```java
import reactor.core.publisher.Flux;
import reactor.core.scheduler.Schedulers;

import java.time.Duration;

public class ZipExample {
    public static void main(String[] args) {
        Flux<String> flux1 = Flux.just("A", "B", "C").delayE
        Flux<Integer> flux2 = Flux.just(1, 2).delayElements(

        Flux<String> combined = Flux.zip(flux1, flux2, (lett

        combined.subscribe(System.out::println);

        // Permet à l'application d'attendre l'exécution
        try { Thread.sleep(3000); } catch (InterruptedExcept
    }
```

Explication :

- La méthode ***zip*** s'arrête dès que l'un des flux a terminé (le flux le plus court détermine la longueur du flux combiné).
- Les éléments restants du flux plus long sont ignorés.

Exemple de combinaison avec un tableau :

```java
import reactor.core.publisher.Flux;

public class ZipExample {
    public static void main(String[] args) {
        Flux<String> flux1 = Flux.just("Alpha", "Beta");
        Flux<Integer> flux2 = Flux.just(10, 20);

        Flux<String> combined = Flux.zip(flux1, flux2)
            .map(tuple -> tuple.getT1() + "-" + tuple.getT2()

        combined.subscribe(System.out::println);
    }
}
```

b. Reactive Web Client

- ***WebClient***, introduit au niveau de Spring 5, est un Framework non-blocking et supporte l'api ***Reactive Streams.***
- Vous pouvez utiliser *WebClient* pour créer un client pour récupérer les données à partir d'un service web Rest.

Ci-après les étapes à suivre pour utiliser WebClient :

- Au niveau du pom.xml, ajouter le starter suivant :

```xml
<dependency>
    <groupId>org.springframework.boot
    <artifactId>spring-boot-starter-w
    <version>3.3.2</version>
```

- Créer un contrôleur :

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    private final EmployeeRepository em
```

- Développer un Endpoint pour donner une seule ressource :

```
@GetMapping("/{id}")
public Mono<Employee> getEmployeeById(@Path
    return employeeRepository.findEmployeeB
```

- Développer un Endpoint pour donner plusieurs ressources :

```
@GetMapping
public Flux<Employee> getAllEmploy
    return employeeRepository.find
```

- Créer une instance de WebClient :

```
public class EmployeeWebClient {

    WebClient client = WebClient.create("ht
```

- Récupérer une seule ressource :

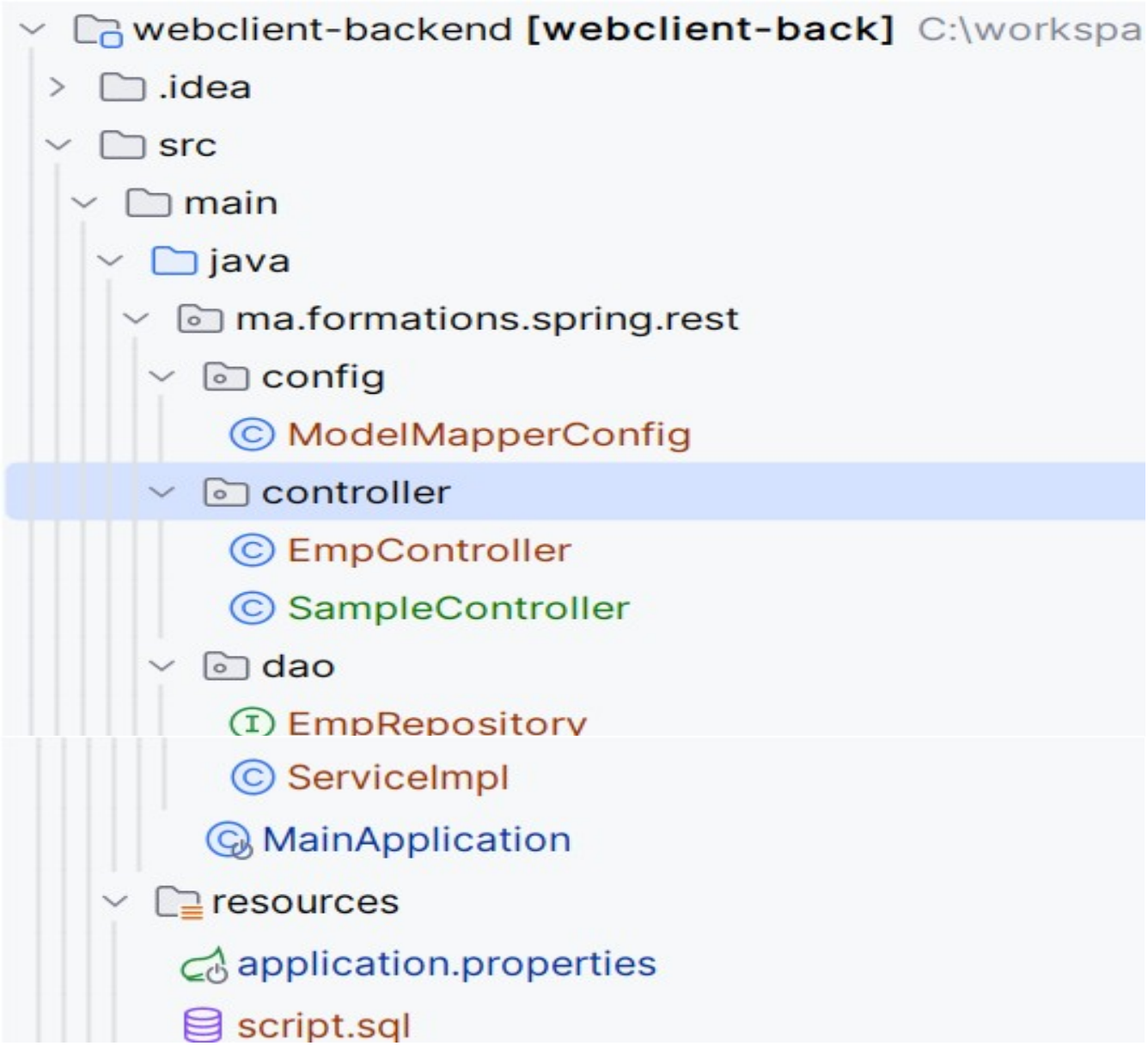
```
Mono<Employee> employeeMono = c
  .uri("/employees/{id}", "1")
  .retrieve()
  .bodyToMono(Employee.class);
```

- Récupérer un flux de données :

```
Flux<Employee> employeeFlux = cli
  .uri("/employees")
  .retrieve()
  .bodyToFlux(Employee.class);
```

V. Développement de la couche back-end avec Spring WebFlux

- Avec Spring Initializr ou Intellig Ultimate, créer un projet Spring Boot (par exemple webclient-backend).
- Créer l'arborescence suivante :



a. Le fichier pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://maven.apache.org/POM/4.0.0"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.5</version>
    <relativePath/>
  </parent>
  <groupId>ma.formations.spring</groupId>
  <artifactId>webclient-back</artifactId>
  <version>1.0.0</version>
  <!-- <packaging>war</packaging> -->
  <name>webclient-back</name>
  <description>WebClient tutorial</description>

  <properties>
    <java.version>17</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
    </dependency>


    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>

    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>io.r2dbc</groupId>
      <artifactId>r2dbc-h2</artifactId>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
```

```
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.modelmapper/modelmapper -->
    <dependency>
      <groupId>org.modelmapper</groupId>
      <artifactId>modelmapper</artifactId>
      <version>3.2.1</version>
    </dependency>

    <!--  Springboot data -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-r2dbc</artifactId>
    </dependency>


  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>


</project>
```

- Le starter « spring-boot-starter-webflux » est un module de Spring Boot qui facilite la création d'applications web réactives en utilisant **Spring WebFlux**. Spring WebFlux est une alternative non bloquante à Spring MVC pour la gestion des requêtes HTTP dans les applications modernes, en particulier celles qui nécessitent une grande scalabilité et des performances optimisées pour les applications asynchrones ou les architectures microservices.

- Spring webFlux utilise par défaut le serveur Netty. Toutefois, vous pouvez forcer Spring Boot à utiliser un conteneur web(Tomcat par exemple) même si vous utilisez Spring WebFlux en suivant les étapes suivantes :
  - Ajouter le starter spring-boot-starter-tomcat au niveau du pom.xml
  - Désactiver Netty dans application.properties en ajoutant la ligne suivante :

```
spring.main.web-application-type=servlet
```

- Pour utiliser H2 en mode réactive, il faut ajouter le starter spring-boot-starter-r2dbc, la librairie r2dbc-h2 et le driver H2.
- Le starter spring-boot-starter-r2dbc :
  - utilise un modèle non bloquant pour les requêtes de bases de données relationnelles, ce qui permet d'exploiter pleinement le potentiel des architectures réactives.
  - Basé sur ***Reactor*** : Comme Spring WebFlux, R2DBC repose sur Project Reactor, avec des types réactifs comme Mono et Flux pour gérer les résultats de requêtes de manière asynchrone.
  - R2DBC prend en charge les bases de données suivantes :
    - ***H2*** (io.r2dbc:r2dbc-h2)
    - ***MariaDB*** (org.mariadb:r2dbc-mariadb)
    - ***Microsoft SQL Server*** (io.r2dbc:r2dbc-mssql)
    - ***MySQL*** (io.asyncer:r2dbc-mysql)
    - ***jasync-sql MySQL*** (com.github.jasync-sql:jasync-r2dbc-mysql)
    - ***Postgres*** (io.r2dbc:r2dbc-postgresql)
    - ***Oracle*** (com.oracle.database.r2dbc:oracle-r2dbc)

- **r2dbc-h2** est une implémentation de R2DBC pour la base de données **H2**. Elle permet d'utiliser H2 de manière réactive dans les applications Spring Boot avec le starter spring-boot-starter-r2dbc.
- Le Framework ***Modelmapper*** permet de convertir les objets métiers (BO ou Business Object) vers les objets DTO (Data Transfert Object).

b. La classe ModelMapperConfig

```
package ma.formations.webclientback.rest.config;

import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ModelMapperConfig {

   @Bean
   public ModelMapper modelMapper() {
      return new ModelMapper();
   }

}
```

c. La classe Emp

```
package ma.formations. webclientback.rest.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Column;
```

```
import org.springframework.data.relational.core.mapping.Table;

@Data @Builder @NoArgsConstructor @AllArgsConstructor
@Table("EMP")
public class Emp {
   @Id
   @Column("ID")
   private Long id;
   @Column("NAME")
   private String name;
   @Column("FONCTION")
   private String fonction;
   @Column("SALAIRE")
   private Double salaire;
}
```

- L'annotation ***@Table*** dans le contexte de Spring Data R2DBC est utilisée pour indiquer qu'une classe représente une table dans la base de données relationnelle. Cette annotation est similaire à celle utilisée dans JPA (@Entity et @Table) pour le mappage objet-relationnel, mais elle est spécifiquement adaptée pour le contexte réactif avec R2DBC. Dans Spring Data R2DBC, l'annotation @Table permet de faire le lien entre une entité Java et une table dans la base de données relationnelle. Elle est utilisée pour définir le nom de la table à laquelle l'entité correspond.

- Idem pour @Id et @Column.

d. La classe EmpDto

```
package ma.formations.webclientback.rest.dtos;

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

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class EmpDto {
   private Long id;
   private String name;
   private String fonction;
   private Double salaire;
}
```

e. La classe EmpRepository

```
package ma.formations.webclientback.rest.dao;

import ma.formations.webclientback.rest.model.Emp;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import org.springframework.stereotype.Repository;
import reactor.core.publisher.Flux;

@Repository
public interface EmpRepository extends ReactiveCrudRepository<Emp, Long> {
   Flux<Emp> findByNameContaining(String name);

   Flux<Emp> findBySalaireBetween(Double min, Double max);
}
```

- L'interface **ReactiveCrudRepository** de **Spring Data R2DBC** (fournie par le starter spring-boot-starter-data-r2dbc) est une extension de l'interface ReactiveRepository qui permet de faciliter l'implémentation des opérations CRUD (Create, Read, Update, Delete) de manière réactive pour une base de données relationnelle.

- L'interface **ReactiveCrudRepository** repose sur **Project Reactor** et permet des opérations non-bloquantes, adaptées aux applications réactives (comme celles qui utilisent **Spring WebFlux** et **R2DBC** pour l'accès aux bases de données).

- Les méthodes de **ReactiveCrudRepository** renvoient des types réactifs comme **Mono** (pour un seul résultat) et **Flux** (pour plusieurs résultats), permettant un traitement asynchrone et non-bloquant, parfaitement adapté aux applications réactives utilisant **R2DBC** avec Spring WebFlux.

- Il faut rescpter les conventions de nommage de Spring Data R2DBC pour les méthodes de requêtes (par exemple, findBy<Attribut>, findBy<Attribut>And<Attribut>, etc.).
- Utiliser **@Query** pour des requêtes SQL personnalisées.

f. L'interface IService

```
package ma.formations.webclientback.rest.service;

import ma.formations.webclientback.rest.dtos.EmpDto;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface IService {
   public Flux<EmpDto> findAll();
   public Flux<EmpDto> findByNameContaining(String name);
   public Mono<EmpDto> findById(Long id);
```

```
    public Mono<EmpDto> save(EmpDto dto);
    public Mono<EmpDto> update(Long id, EmpDto dto);
    public Mono<Void> deleteById(Long id);
    public Mono<Void> deleteAll();
    public Flux<EmpDto> findBySalaireBetween(Double min, Double max);
}
```

g. La classe ServiceImpl

```
package ma.formations.webclientback.rest.service;

import lombok.AllArgsConstructor;
import ma.formations.webclientback.rest.dao.EmpRepository;
import ma.formations.webclientback.rest.dtos.EmpDto;
import ma.formations.webclientback.rest.model.Emp;
import org.modelmapper.ModelMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Optional;

@Service
@AllArgsConstructor
public class ServiceImpl implements IService {
    private EmpRepository empRepository;
    private ModelMapper modelMapper;

    public Flux<EmpDto> findAll() {
        return empRepository.findAll().map(bo -> modelMapper.map(bo, EmpDto.class));
    }

    public Flux<EmpDto> findByNameContaining(String name) {
        return empRepository.findByNameContaining(name).map(bo -> modelMapper.map(bo,
EmpDto.class));
    }

    public Mono<EmpDto> findById(Long id) {
        return empRepository.findById(id).map(bo -> modelMapper.map(bo, EmpDto.class));
    }

    @Transactional
    public Mono<EmpDto> save(EmpDto dto) {
        return empRepository.save(modelMapper.map(dto, Emp.class)).
                map(bo -> modelMapper.map(bo, EmpDto.class));
    }

    @Transactional
    public Mono<EmpDto> update(Long id, EmpDto dto) {
        return empRepository.findById(id).map(Optional::of).defaultIfEmpty(Optional.empty())
                .flatMap(optionalEmp -> {
                    if (optionalEmp.isPresent()) {
                        dto.setId(id);
                        return empRepository.save(modelMapper.map(dto, Emp.class)).
                                map(bo -> modelMapper.map(bo, EmpDto.class));
```

```
            }
            return Mono.empty();
        });
    }

    @Transactional
    public Mono<Void> deleteById(Long id) {
        return empRepository.deleteById(id);
    }

    @Transactional
    public Mono<Void> deleteAll() {
        return empRepository.deleteAll();
    }

    @Override
    public Flux<EmpDto> findBySalaireBetween(Double min, Double max) {
        return empRepository.findBySalaireBetween(min, max).map(bo -> modelMapper.map(bo,
EmpDto.class));
    }
}
```

h. La classe SampleController

```
package ma.formations.webclientback.rest.controller;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.time.Duration;

@RestController
public class SampleController {

    @GetMapping("/test1")
    Mono<String> test1() {
        return Mono.just("Hello world");
    }

    @GetMapping("/test2")
    Flux<Integer> test2() {
        return Flux.just(1, 22, 44, 55);
    }

    @GetMapping(value = "/test3", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    Flux<Integer> test3() {
        return Flux.just(1, 2, 3, 4, 5, 6, 7).delayElements(Duration.ofSeconds(1l)).
                log();
    }

    @GetMapping("/test")
    public String test() {
        return Thread.currentThread().getName();
```

```
    }

}
```

- **MediaType.TEXT_EVENT_STREAM_VALUE** est un type de média (MIME type) utilisé pour le streaming d'événements en temps réel via Server-Sent Events (SSE). Il est utilisé lorsque vous souhaitez que le serveur envoie un flux de données au client en temps réel.

- Les **Server-Sent Events** (**SSE**) sont un mécanisme dans le protocole HTTP où le serveur envoie de manière continue des mises à jour ou des événements au client via une seule connexion HTTP. Contrairement aux **WebSockets** (qui permettent une communication bidirectionnelle), les SSE sont unidirectionnels : seul le serveur envoie des événements vers le client.

- Les événements SSE sont envoyés sous forme de texte, et chaque événement dans le flux est généralement structuré sous forme de texte JSON ou texte brut.

i. La classe EmpController

```
package ma.formations.webclientback.rest.controller;

import lombok.AllArgsConstructor;
import ma.formations.webclientback.rest.dtos.EmpDto;
import ma.formations.webclientback.rest.service.IService;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.time.Duration;
import java.util.Map;

@CrossOrigin(origins = "http://localhost:8081")
@RestController
@AllArgsConstructor
public class EmpController {
   private IService service;

   @GetMapping(value = "/employees/{id}")
   Mono<EmpDto> getEmployees(@PathVariable Long id) {
      return service.findById(id);
   }

   @GetMapping(value = "/employees", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
   Flux<EmpDto> getEmployees() {
      return service.findAll().delayElements(Duration.ofSeconds(1L));
   }
```

```
    @PostMapping(value = "/employees")
    Mono<EmpDto> createEmployee(@RequestBody EmpDto dto) {
        return service.save(dto);
    }

    @PutMapping("/employees/{id}")
    @ResponseStatus(HttpStatus.OK)
    public Mono<EmpDto> updateEmployee(@PathVariable("id") Long id, @RequestBody EmpDto dto) {
        return service.update(id, dto);
    }

    @DeleteMapping("/employees/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> deleteEmployees(@PathVariable("id") Long id) {
        return service.deleteById(id);
    }

    @DeleteMapping("/employees")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> deleteAllEmployees() {
        return service.deleteAll();
    }

    @GetMapping("/employees/salaire")
    public Flux<EmpDto> getEmployeesBySalaireBetween(@RequestParam Map<String, String> parameters) {
        Double min = Double.parseDouble(parameters.get("min"));
        Double max = Double.parseDouble(parameters.get("max"));
        return service.findBySalaireBetween(min, max);
    }

}
```

j. Le fichier application.properties

```
spring.r2dbc.url=r2dbc:h2:mem:///test
```

k. Le fichier script.sql

```
CREATE TABLE IF NOT EXISTS emp
(
    id      INT NOT NULL AUTO_INCREMENT,
    name    VARCHAR(255),
    fonction VARCHAR(255),
    salaire  DOUBLE,
    PRIMARY KEY (id)
);
```

l. La classe MainApplication

```
package ma.formations.webclientback.rest;
```

```
import io.r2dbc.spi.ConnectionFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.r2dbc.connection.init.ConnectionFactoryInitializer;
import org.springframework.r2dbc.connection.init.ResourceDatabasePopulator;
import org.springframework.web.reactive.config.EnableWebFlux;

@SpringBootApplication
@EnableWebFlux
public class MainApplication {

   public static void main(String[] args) {
      SpringApplication.run(MainApplication.class, args);
      System.out.println("Application démarrée");
   }

   @Bean
   ConnectionFactoryInitializer initializer(ConnectionFactory connectionFactory) {

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

      return initializer;
   }

}
```

- Dans Spring R2DBC, ***ConnectionFactoryInitializer*** est une classe utilisée pour initialiser la base de données au démarrage de l'application, généralement pour exécuter des scripts SQL comme la création de tables, l'insertion de données ou la configuration de schémas

m. Tester la couche back-end

- Exécuter la méthode main de la classe MainApplication.
- Lancer l'URL http://localhost:8080/test1 et vérifier le résultat suivant :

Hello world

- Lancer l'URL http://localhost:8080/test2 et vérifier le résultat suivant :

1, 22, 44, 5

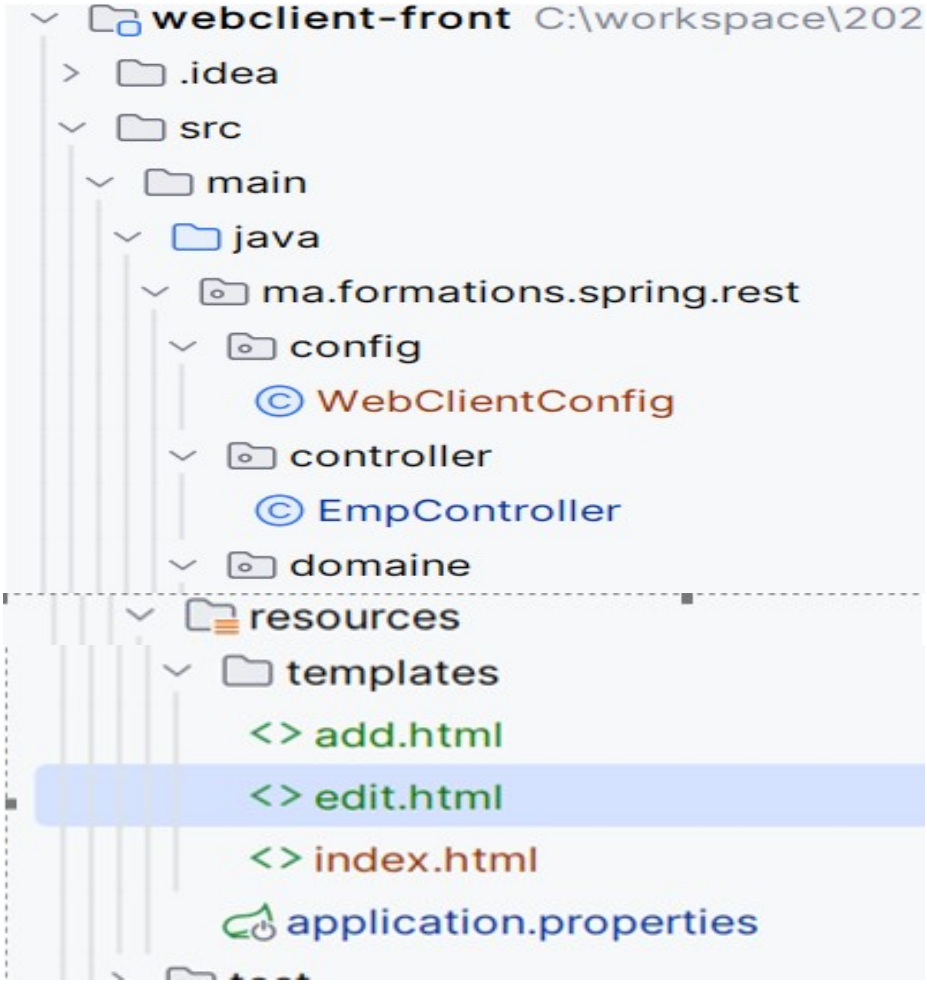
- Lancer l'URL http://localhost:8080/test3 et vérifier le résultat suivant :

- Chaque seconde le serveur envoi au une réponse (1,2, …7). Ici, c'est le principe du SSE (Server-Sent Events).
- Chaque réponse est tracé au niveau de la console et ceci chaque seconde.

- Lancer l'URL http://localhost:8080/test plusieurs fois et vérifier que c'est toujours le même Thread qui traite votre requête. Rappelez-vous que parmi les objectifs de la programmation est l'optimisation des ressources.
- Avec POSTMAN, tester la création d'un nouvel employé en exécutant l'URL http://localhost:8080/employees avec la méthode POST et en passant dans le corps de la requête le message JSON suivant (à titre d'exemple) :
  { "name " : "EMPLOYEE_1 ","fonction " : "FONCTION_1 ","salaire " : 10000}
- Lancer l'URL http://localhost:8080/employees et vérifier que le serveur envi chaque seconde un flux et qui sera affiché à temps réel par le navigateur.
- Avec POSTMAN, tester la modification d'un employé existant en exécutant l'URL http://localhost:8080/employees/1 avec la méthode PUT et en passant dans le corps de la requête les message JSON suivant (à titre d'exemple) :
  { "name " : "EMPLOYEE_1 ","fonction " : "NEW_FONCTION ","salaire " : 15000}
- Avec POSTMAN, tester la récupération d'un employé existant par son ID en exécutant l'URL http://localhost:8080/employees/1 avec la méthode GET.
- Tester la recherche des employées ayant les salaires entre MIN et MAX en exécutant l'URL http://localhost:8080/employees/salaire?min=8000&max=25000.
- Avec POSTMAN, tester la suppression d'un employé existant en exécutant l'URL http://localhost:8080/employees/1 avec la méthode DELETE.
- Avec POSTMAN, tester la suppression de tous les employés en exécutant l'URL http://localhost:8080/employees avec la méthode DELETE.

VI. Développement de la couche front-end avec Spring WebFlux

- Avec Spring Initializr ou Intellig Ultimate, créer un projet Spring Boot (par exemple webclient-front).
- Créer l'arborescence suivante :



a. Le fichier pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-parent</artifactId>
          <version>3.1.5</version>
          <relativePath /> <!-- lookup parent from repository -->
  </parent>
  <groupId>ma.formations.spring</groupId>
  <artifactId>webclient-front</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>webclient-front</name>
  <description>Using WebClient API</description>
  <properties>
          <java.version>17</java.version>
          <class>ma.formations.webclientback.rest.MainApplication</class>
  </properties>
  <dependencies>
          <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-webflux</artifactId>
          </dependency>
          <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-thymeleaf</artifactId>
          </dependency>

          <dependency>
                  <groupId>org.webjars</groupId>
                  <artifactId>bootstrap</artifactId>
                  <version>3.3.7</version>
          </dependency>

          <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-devtools</artifactId>
          </dependency>

          <dependency>
                  <groupId>org.projectlombok</groupId>
                  <artifactId>lombok</artifactId>
                  <optional>true</optional>
          </dependency>
```

```
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
        </dependency>

        </dependencies>
  <build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
        </plugins>
  </build>
</project>
```

- Le starter **spring-boot-starter-thymeleaf** permet de configurer le moteur de templates ***Thymeleaf*** dans votre projet.

- Bien que le starter spring-boot-starter-thymeleaf configure automatiquement Thymeleaf, vous pouvez personnaliser le dossier de vos vues, l'extension, l'activation de la cache :

```
#par défaut : .html
spring.thymeleaf.suffix=.html

#par défaut : /resources/templates
spring.thymeleaf.prefix=classpath:/templates/

# Active ou désactive le cache Thymeleaf (utile pour le développement)
spring.thymeleaf.cache=false
```

b. La classe WebClientConfig

```
package ma.formations.webclientback.rest.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class WebClientConfig {
   private WebClient webClient;
   @Value("${back-end.api}")
   private String url;
   @Bean
   public WebClient webClient() {
      return WebClient.builder().baseUrl(url).build();
   }
}
```

- @Value("${back-end.api}") permet d'injecter la valeur de la clé back-end.api définie au niveau du fichier application.properties.

- WebClient.builder().baseUrl(url).build() permet de configurer un ***WebClient*** avec une URL de base dans une application Spring. Elle permet de créer des requêtes HTTP réactives vers une API ou un service web tout en gardant la flexibilité de configurer le comportement du client HTTP via le builder.

c. La classe EmpVo

```
package ma.formations.webclientback.rest.domaine;

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

import java.io.Serializable;
import java.util.Date;
@NoArgsConstructor
@AllArgsConstructor
@Data
public class EmpVo implements Serializable {
        private Long id;
        private String name;
        private Double salaire;
        private String fonction;
}
```

- Les objets de la classe EmpVo jouent le rôle du DTO (Data Transfert Object) à échanger enter entre la couche Front et la couche Back.

d. L'interface IService

```
package ma.formations.webclientback.rest.service;

import ma.formations.webclientback.rest.domaine.EmpVo;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface IService {
   Flux<EmpVo> getAllEmployees();
   Mono<EmpVo> getEmployeeById(Long ID);
   Mono<EmpVo> createEmployee(EmpVo vo);
   Mono<EmpVo> updateEmployee(Long id,EmpVo vo);
   Mono<Void> deleteEmployee(Long id);
}
```

e. La classe ServiceImpl

```
package ma.formations.webclientback.rest.service;

import lombok.AllArgsConstructor;
import ma.formations.webclientback.rest.domaine.EmpVo;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
@AllArgsConstructor
public class ServiceImp implements IService{
   private WebClient webClient;
   @Override
   public Flux<EmpVo> getAllEmployees() {
      return webClient.get()
            .uri("/employees")
            .retrieve().bodyToFlux(EmpVo.class);
   }

   @Override
   public Mono<EmpVo> getEmployeeById(Long id) {
      return webClient.get()
            .uri("/employees/{id}",id)
            .retrieve().bodyToMono(EmpVo.class);
   }
```

```
    @Override
    public Mono<EmpVo> createEmployee(EmpVo vo) {
        return webClient.post()
                .uri("/employees")
                .bodyValue(vo)
                .retrieve().bodyToMono(EmpVo.class);
    }

    @Override
    public Mono<EmpVo> updateEmployee(Long id, EmpVo vo) {
        return webClient.put()
                .uri("/employees/{id}",id)
                .bodyValue(vo)
                .retrieve().bodyToMono(EmpVo.class);
    }

    @Override
    public Mono<Void> deleteEmployee(Long id) {
        return webClient.delete()
                .uri("/employees/{id}",id)
                .retrieve().bodyToMono(Void.class);
    }
}
```

f. La classe EmpController

```
package ma.formations.webclientback.rest.controller;

import lombok.AllArgsConstructor;
import ma.formations.webclientback.rest.domaine.EmpVo;
import ma.formations.webclientback.rest.service.IService;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.reactive.function.client.WebClient;
import org.thymeleaf.spring6.context.webflux.IReactiveDataDriverContextVariable;
import org.thymeleaf.spring6.context.webflux.ReactiveDataDriverContextVariable;
import reactor.core.publisher.Mono;

@Controller
@AllArgsConstructor
public class EmpController {
private IService service;

    @RequestMapping(value="/")
    public String showWelcomeFile(Model m) {
        // loads 1 and display 1, stream data, data driven mode.
        IReactiveDataDriverContextVariable reactiveDataDrivenMode =
                new ReactiveDataDriverContextVariable(service.getAllEmployees(), 1);

        m.addAttribute("employees", reactiveDataDrivenMode);
```

```
    return "index";
  }

  @GetMapping("/add")
  public String showAddForm( Model m) {
    m.addAttribute("emp", new EmpVo());
    return "add";
  }


  @PostMapping("/create")
  public String addEmployee(EmpVo emp,
                BindingResult result, Model m) {
    service.createEmployee(emp).subscribe();
    return "redirect:/";
  }

  @GetMapping(value = "/edit/{id}")
  public String showUpdateForm(@PathVariable("id") long id, Model m) {
    Mono<EmpVo> emp = service.getEmployeeById(id);
    m.addAttribute("emp",emp );
    return "edit";
  }

  @PostMapping("/update")
  public String updateEmp( EmpVo emp,
                Model m) {
    service.updateEmployee(emp.getId(), emp).subscribe();
    return "redirect:/";
  }

  @GetMapping("/delete/{id}")
  public String deleteEmp(@PathVariable("id") long id, Model model) {
    service.deleteEmployee(id).subscribe();
    return "redirect:/";
  }
}
```

- ***ReactiveDataDriverContextVariable*** est une classe spécifique à **Spring WebFlux** et à son moteur de templates **Thymeleaf**. Elle fait partie des extensions permettant une gestion réactive des variables de contexte dans **Thymeleaf** pour **un rendu dynamique** des vues.

- Dans un contexte réactif, lorsque vous travaillez avec **Thymeleaf** dans une application **WebFlux**, le moteur de template doit gérer les variables de contexte, qui peuvent être des données provenant de requêtes HTTP ou d'autres sources de données réactives.

- **ReactiveDataDriverContextVariable** permet à **Thymeleaf** de prendre en charge des objets réactifs comme des Mono ou Flux dans le modèle de contexte.

- La valeur « 1 » dans le constructeur :
  new ReactiveDataDriverContextVariable(service.getAllEmployees(), 1) défini le nombre de chunck (ou morceaux) qui seront rendus par Thymeleaf dans chaque stream. Ici, il s'agit d'un un seul chunck.

g. Le fichier application.properties

```
server.port=8081
back-end.api=http://localhost:8080
spring.thymeleaf.reactive.max-chunk-size=8192
```

- spring.thymeleaf.reactive.max-chunk-size=8192 : permet de configurer la taille du chunk (morceau) envoyé lors du rendu d'une page avec Spring WebFlux et Thymeleaf. Ici, c'est 8 KO.

- Cela permet d'optimiser la gestion des ressources et d'améliorer les performances dans des environnements hautement réactifs. Vous pouvez ajuster cette valeur en fonction de la taille des pages et des ressources disponibles pour mieux contrôler l'expérience utilisateur.

h. La classe MainApplication

```
package ma.formations.webclientback.rest;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class MainApplication {

	public static void main(String[] args) {
		SpringApplication.run(MainApplication.class, args);
		System.out.println("Application démarrée");
	}

}
```

i. La page index.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Test</title>
  <link rel="stylesheet" type="text/css" href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"/>
</head>
<body>
<p></p>
<div class="container">
  <table class="table">
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th>Fonction</th>
      <th>Salaire</th>
    </tr>
    <tr th:each="emp:${employees}">
      <td>[[${emp.id}]]</td>
      <td>[[${emp.name}]]</td>
      <td>[[${emp.fonction}]]</td>
      <td>[[${emp.salaire}]]</td>
      <td><a th:href="@{/edit/{id}(id=${emp.id})}">Edit</a></td>
      <td><a th:href="@{/delete/{id}(id=${emp.id})}">Delete</a></td>

    </tr>

  </table>
  <p><a href="/add" class="btn btn-primary">Ajouter un nouvel employé</a></p>
</div>
</body>
</html>
```

- La syntaxe [[${emp.id}]] dans Thymeleaf est utilisée pour afficher dynamiquement la valeur de emp.id dans le rendu HTML.

j. La page add.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Test</title>
  <link rel="stylesheet" type="text/css" href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"/>
</head>
```

```
<body>
<p></p>
<div class="container">
  <form action="#" th:action="@{/create}" th:object="${emp}" method="post">
    <label for="name">Name</label>
    <input type="text" th:field="*{name}" id="name" placeholder="Name">
    <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}"></span>

    <label for="fonction">Fonction</label>
    <input type="text" th:field="*{fonction}" id="fonction" placeholder="Fonction">
    <span th:if="${#fields.hasErrors('fonction')}" th:errors="*{fonction}"></span>

    <label for="salaire">Salaire</label>
    <input type="text" th:field="*{salaire}" id="salaire" placeholder="Salaire">
    <span th:if="${#fields.hasErrors('salaire')}" th:errors="*{salaire}"></span>

    <input type="submit" value="Ajouter" class="btn btn-primary">
  </form>
</div>
</body>
</html>
```

k. La page edit.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Test</title>
  <link rel="stylesheet" type="text/css" href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"/>
</head>
<body>
<div class="container">
  <form action="#" th:action="@{/update}" th:object="${emp}" method="post">
    <label for="name">ID</label>
    <input type="text" th:field="*{id}" id="id" placeholder="Name">
    <span th:if="${#fields.hasErrors('id')}" th:errors="*{id}"></span>

    <label for="name">Name</label>
    <input type="text" th:field="*{name}" id="name" placeholder="Name">
    <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}"></span>
    <label for="fonction">Fonction</label>
    <input type="text" th:field="*{fonction}" id="fonction" placeholder="Fonction">
    <span th:if="${#fields.hasErrors('fonction')}" th:errors="*{fonction}"></span>
    <label for="salaire">Salaire</label>
    <input type="text" th:field="*{salaire}" id="salaire" placeholder="Salaire">
    <span th:if="${#fields.hasErrors('salaire')}" th:errors="*{salaire}"></span>

    <input type="submit" value="Enregistrer" class="btn btn-primary">
  </form>
</div>
</body>
</html>
```

I. Tester la couche front

- Commencer par démarrer la couche back-end.
- Lancer la méthode main de la classe MainApplication.
- Aller au lien http://localhost:8081 et vérifier la liste des employés est affichée en mode réactive :
- Pour ajouter un nouvel employé cliquer sur le bouton « Ajouter un nouvel employé » :
  - Entrer le nom, la fonction et le salaire du nouvel employé.
  - Cliquer sur le bouton Ajouter et vérifier que l'employé a été bien ajouté.
- Pour modifier un employé existant, cliquer sur le lien ***edit*** :
  - Entrer les modifications.
  - Cliquer sur le bouton ***Enregistrer*** et vérifier que l'employé a été bien modifié.
- Pour supprimer un employé existant, cliquer sur le lien ***delete*** et vérifier que l'employé a été bien supprimé.