



Développement d'applications modulaires en Java

.....

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr



Plan de l'ECUE

1. (Rappels sur le) Développement d'applications Web avec Java
2. Modulariser les applications Java avec Spring
3. Bien structurer une application Web avec Spring MVC
4. Auto-configurer une application Web avec Spring Boot
5. Sécuriser une application Web avec Spring Security
6. Gérer des données massives avec Apache Kafka et Spring
7. Tester une application Web Spring
8. Écrire des applications Web (API) réactives avec Spring WebFlux

Plan du cours

1. API *Reactive Streams* et son implem *Reactor*
2. Développer une API REST réactive avec des contrôleurs annotés
3. Développer une API REST réactive avec des *endpoints* fonctionnels

Pourquoi une autre API pour les app Web ?

Besoins en appels non-bloquants et asynchrones

- L'API des servlets permet de créer des servlets synchrones
- Possibilité d'écrire des servlets asynchrones depuis l'API 3.1+ (vus dans le premier cours)
- Malgré cela, ce qui entoure ces servlets reste synchrone, comme les appels dans les filtres, ou bloquant comme les appels à `getParameter()`
- Besoin de gérer la concurrence (et la scalabilité) avec peu de ressources matérielles en utilisant une API asynchrone et non-bloquante
- Cela a été rendu possible grâce à des serveurs comme Netty

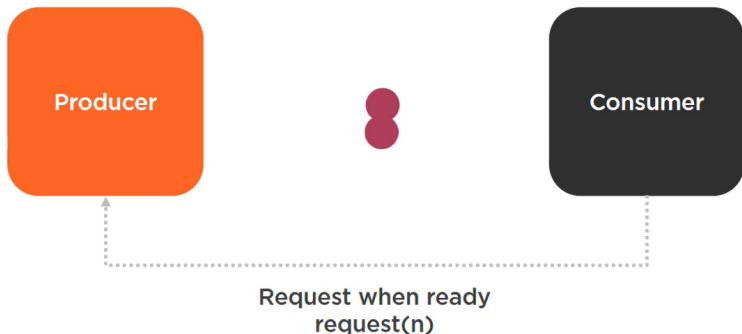
Besoins d'utiliser la programmation déclarative & fonctionnelle

- Arrivée des annotations dans Java 5
- Arrivée des lambdas et de l'API Stream dans Java 8

Que veut dire “programmation réactive” ?

- Un modèle de programmation basé sur la réaction au changement :
 - Des composants en réseau réagissant aux entrées/sorties
 - Des composants UI réagissant aux événements de la souris
- Un programme est réactif \Rightarrow il n'est pas bloqué
- Au lieu de rester bloqué (en attente d'une opération qu'elle se termine), il est prêt à réagir aux notifications et à la disponibilité des données
- Rien de nouveau : pattern *publish-subscribe* (1 producteur de données et 1 souscripteur, qui est réveillé quand des données sont produites)

Back Pressure



- Concept étroitement lié à la programmation réactive
- Il signifie le contrôle de la production de données pour que le souscripteur ne soit pas dépassé par le traitement des données qu'il reçoit du producteur

Back Pressure -suite-

Et si le producteur ne peut pas contrôler la production de données ?

- Il doit être capable de bufferiser les données, les ignorer ou s'arrêter

Ceci relève de l'applicatif et non du mécanisme des *Reactive Streams*

Reactive Streams

- Une API, adoptée dans Java 9, permettant d'écrire des composants asynchrones avec *back pressure*
- Des composants Publishers et des composants Subscribers
- Exemple dans une app Web : Un (data) repository agit comme Publisher et un serveur Web (un contrôleur) agit comme Subscriber pour écrire les données dans des réponses HTTP
- Au lieu d'avoir un contrôleur qui appelle de manière synchrone le (data) repository (tel qu'on l'a vu dans Spring MVC)
- C'est une spec : <https://www.reactive-streams.org/>
La classe Flow dans Java : <https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/Flow.html>
- Elle possède plusieurs implem, comme *Reactor*

API Reactive Streams

4 interfaces :

- Publisher
- Subscriber
- Subscription
- Processor (Publisher & Subscriber at the same time)


Publisher & Subscriber

```
org.reactivestreams  
public interface Publisher<T>
```

A **Publisher** is a provider of a potentially unbounded number of sequenced elements, publishing them according to the demand received from its **Subscriber(s)**.

A **Publisher** can serve multiple **Subscribers** subscribed `subscribe(Subscriber)` dynamically at various points in time.

Type parameters: `<T>` – the type of element signaled.

 Gradle: `org.reactivestreams:reactive-streams:1.0.3`

```
org.reactivestreams  
public interface Subscriber<T>
```


Will receive call to `onSubscribe(Subscription)` once after passing an instance of **Subscriber** to `Publisher.subscribe(Subscriber)`. No further notifications will be received until `Subscription.request(long)` is called.

After signaling demand:

- One or more invocations of `onNext(Object)` up to the maximum number defined by `Subscription.request(long)`
- Single invocation of `onError(Throwable)` or `onComplete()` which signals a terminal state after which no further events will be sent.

Demand can be signaled via `Subscription.request(long)` whenever the **Subscriber** instance is capable of handling more.

Type parameters: `<T>` – the type of element signaled.

 Gradle: `org.reactivestreams:reactive-streams:1.0.3`

Publisher & Subscriber -suite-


```
public interface Publisher<T> {  
    public void subscribe(Subscriber<? super T> s);  
}  
  
public interface Subscriber<T> {  
    public void onSubscribe(Subscription s);  
    public void onNext(T t);  
    public void onError(Throwable t);  
    public void onComplete();  
}
```

Subscription

```
org.reactivestreams  
public interface Subscription
```

A `Subscription` represents a one-to-one lifecycle of a `Subscriber` subscribing to a `Publisher`. It can only be used once by a single `Subscriber`.

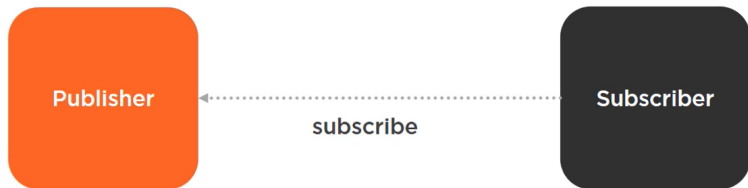
It is used to both signal desire for data and cancel demand (and allow resource cleanup).

 Gradle: org.reactivestreams:reactive-streams:1.0.3

```
public interface Subscription {  
    public void request(long n);  
    public void cancel();  
}
```

`request()` permet de mettre en place la *backpressure*
(demander `n` éléments que le *subscriber* est capable de traiter)

Flux de données



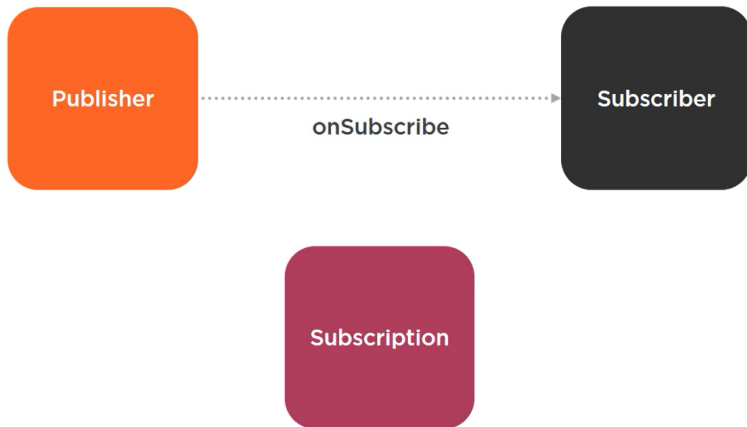
- Le *subscriber* souscrit à un *publisher*

Flux de données



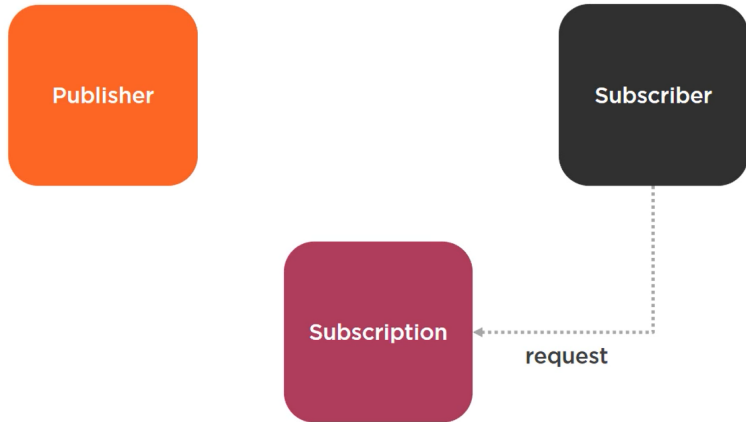
- Un objet *subscription* est créé

Flux de données



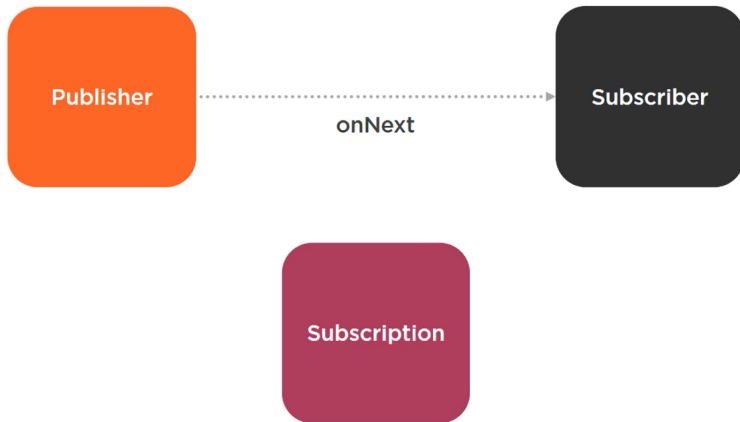
- La méthode `onSubscribe()` du *subscriber* est invoquée et l'objet *subscription* est passé en argument

Flux de données



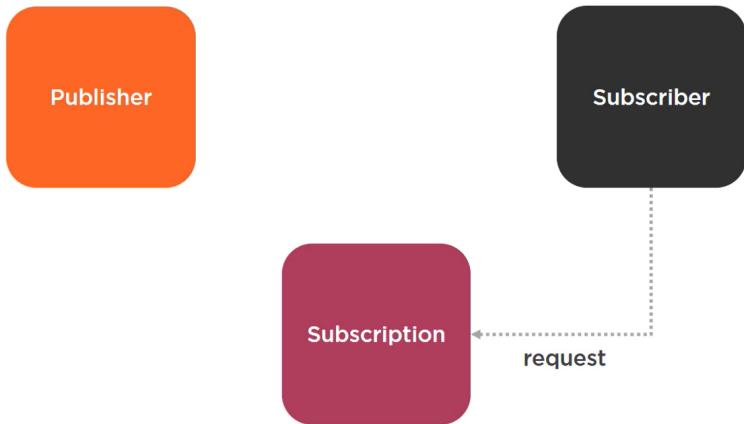
- Le *subscriber* utilise la *subscription* pour indiquer combien de données il est capable de traiter (avec request)

Flux de données



- `onNext()` est invoquée et une donnée est passée en argument
- C'est fait plusieurs fois (n fois – n indiqué dans *request*)

Flux de données



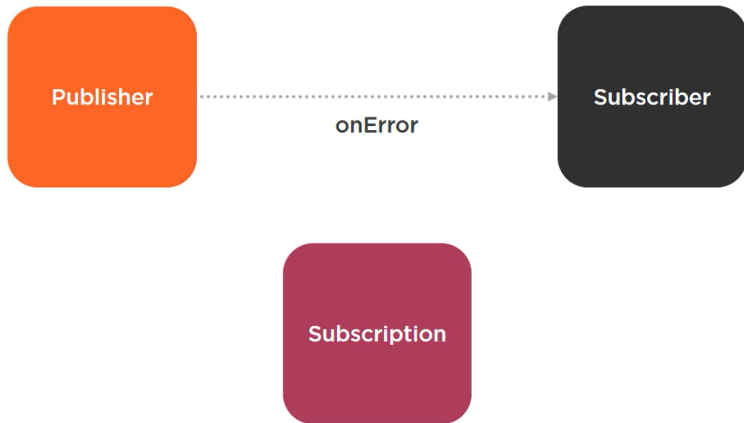
- Soit le *subscriber* demande de nouvelles données

Flux de données



- Ou bien, c'est fini, `onComplete()` de *subscriber* est invoquée et la *subscription* est annulée

Flux de données



- Ou encore, il y a une erreur, `onError ()` de *subscriber* est invoquée et la *subscription* est annulée

Plusieurs impls de l'API Reactive Streams

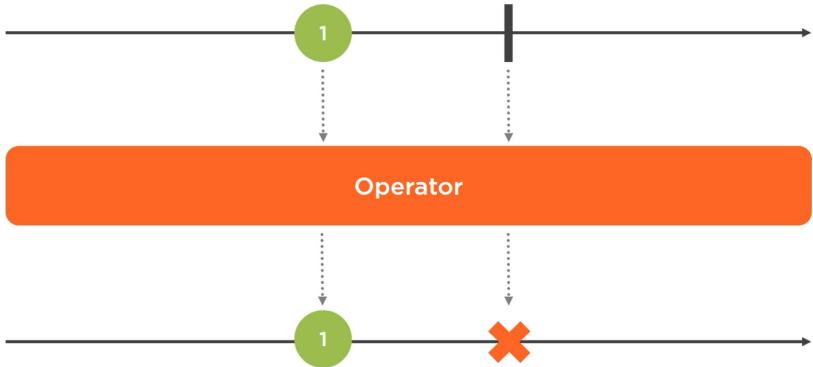
- Reactor
- RxJava et RxJava2
- CompletableFeature
- Java 9+ Flow API

(*Project*) Reactor est l'impl par défaut (et recommandée) dans Spring Web Flux : <https://projectreactor.io/>

Publishers de Reactor

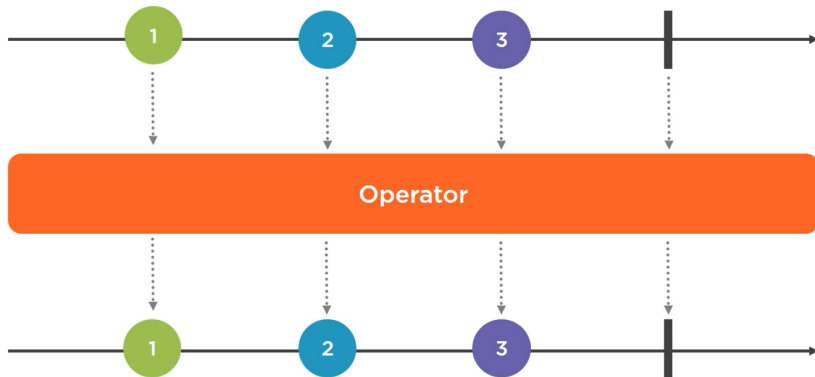
- **Mono** permet de produire 0 ou une donnée
 - Mono est l'équivalent *réactif* du type `Optional` de Java 8+
 - A utiliser quand on veut retourner dans une méthode 1 objet ou bien void
- **Flux** permet de produire plus d'une donnée
 - A utiliser quand on veut retourner une liste

Marble Diagram de Mono



- Operator permet de transformer ou vérifier la donnée produite

Marble Diagram de Flux



- La ligne verticale indique le onComplete et la croix le onError

Tester Reactor

- Créer un projet Gradle & Java sur IntelliJ
- Ajouter la dépendance suivante :
implementation 'io.projectreactor:reactor-core:3.4.0'
- Dans la classe de test, ajouter une méthode pour tester Mono :

```
@Test
public void firstMono() {
    Mono.just("A")// Create a Publisher that produces "A" once
        .log()// Log the events
        .subscribe(s->System.out.println(s));// Create a Subscriber
}
```

- On a créé un *publisher* de type Mono qui produit la chaîne "A"
- On a créé un *subscriber*, ici un objet
java.util.function.Consumer qui correspond à une
lambda qui fait un println

Tester Reactor -Mono-

- Le même exemple avec des écouteurs d'événements :

```
@Test
public void monoWithDoOn() {
    Mono.just("A") // Publish "A"
        .log() // Log the events
        .doOnSubscribe(subscription
            -> System.out.println("Subscription : "+subscription))
        .doOnRequest(request
            -> System.out.println("Request: "+request))
        .doOnSuccess(complete
            -> System.out.println("Complete: "+complete))
        .subscribe(System.out::println);
}
```

Noter le remplacement de la lambda par une référence de méthode à la dernière ligne

Tester Reactor -Mono-

- Mono vide :

```
@Test
public void emptyMono() {
    Mono.empty()
        .log()
        .subscribe(System.out::println);
}
```

- Ici, onNext n'est pas invoquée parce que le Mono ne produit pas une vraie donnée
- Seule onComplete sera invoquée
- Mais à quoi sert donc un Mono.empty() ?
A simuler un return dans une méthode void et dire que le process de production du Publisher s'est terminé (mais sans envoi de données)

Tester Reactor -Mono-

- Mono qui produit une erreur :

```
@Test
public void monoWithError () {
    Mono.error(new RuntimeException ())
        .log ()
        .subscribe ();
}
```

- Ici, on fait une souscription mais sans indiquer ce qui doit se passer après onNext
- A l'exécution, vous verrez onError qui affiche la *stack trace* de l'exception
- Pas de try-catch dans la programmation réactive avec Reactor
- Possibilité de capture avec onErrorResume ou onErrorReturn

Tester Reactor -Flux-

- Ajouter une méthode pour tester Flux :

```
@Test
public void firstFlux () {
    //Flux.just ("A","B","C")
    Flux.range (10 ,5)
        .log ()
        .subscribe (System.out::println );
}
```

- Possibilité de faire Flux.fromIterable, .fromArray, .fromStream, fromPublisher, ...

Tester Reactor -Flux-

- Produire des valeurs avec un intervalle de temps :

```
Flux.interval(Duration.ofSeconds(1))
```

Duration est issue de `java.time`

```
@Test
public void firstFlux () {
    Flux.interval(Duration.ofSeconds(1))
        .log()
        .subscribe(System.out::println);
    Thread.sleep(5000);
}
```

- `interval` démarre dans un nouveau thread la production d'entier à partir de 0 toutes les 1 seconde
- On met le `Thread.sleep` pour ne pas avoir le test qui se termine (et donc le thread créé par `interval` tué) avant la production des données

Tester Reactor -Flux-

- Dans l'exemple, onComplete n'est pas invoquée parce que le thread a été tué à la fin de l'exécution du test
- Dans une application Web, on a des processus/threads qui s'exécutent en continu
- On peut donc avoir une production de valeurs en continu
- Pour annuler la production de données au delà d'un certain nombre (de données), on peut utiliser l'opérateur take :

```
Flux.interval(Duration.ofSeconds(1))  
    .log()  
    .take(5) // Annule la souscription apres 5 donnees  
    .subscribe(System.out::println); Thread.sleep(8000);
```

onComplete n'est pas invoquée (ce n'est pas un mécanisme de *backpressure*). C'est plutôt un cancel (lire la doc de take sur IntelliJ).

Tester Reactor -Flux-

- Pour appliquer une *backpressure*, utiliser l'opérateur `limitRate` :

```
Flux.range(1,8)
    .log()
    .limitRate(3)
    .subscribe();
```

- Plus de détails :

<https://projectreactor.io/docs/core/release/reference/>

Backpressure avec un subscriber personnalisé

```
Flux.range(1, 10)
    .log().subscribe(new BaseSubscriber<Integer>() {
        int elementsToProcess = 3; int counter = 0;
        @Override
        protected void hookOnSubscribe(Subscription subs) {
            System.out.println("Subscribed");
            subs.request(elementsToProcess);
        }
        @Override
        protected void hookOnNext(Integer value) {
            counter++;
            if(counter == elementsToProcess) {
                counter = 0;
                Random r = new Random();
                elementsToProcess = r.ints(3,5).findFirst().getAsInt();
                request(elementsToProcess);
            }
        }
    });
```

Quelques opérateurs

1. map
2. flatMap
3. concat et merge
4. zip

Opérateur map

- Similaire à map vue dans l'API Stream (en IG4). Applique une fonction synchrone à chaque donnée du Flux

```
@Test
public void fluxWithMap () {
    Flux.range(1,5)
        .map(i -> i * 5)
        .subscribe(System.out::println);
}
```

- Survoler le nom de la méthode map sur IntelliJ pour voir sa documentation

Opérateur flatMap

- Version réactive de map. Doit donc retourner un Mono ou un Flux :

```
@Test
public void fluxWithFlatMap() {
    Flux.range(1,5)
        .flatMap(i->Flux.range(i*5,10))
        .subscribe(System.out::println);
}
```

- Chaque donnée est transformée en un publisher qui exploite la donnée
- Survoler le nom de la méthode flatMap sur IntelliJ pour voir son fonctionnement avec le Marble Diagram

Opérateur flatMapMany

- Transforme un Mono en un Flux

```
@Test
public void fluxWithFlatMapMany() {
    Mono.just(3)
        .flatMapMany(i -> Flux.range(1, i))
        .subscribe(System.out::println);
}
```

- Crée un Flux à partir d'un Mono en exploitant la donnée produite par le Mono

Opérateurs concat et merge

- concat concatène les données produites par des publishers

```
@Test
public void fluxWithConcat() throws
    InterruptedException {
    Flux<Integer> oneToFive = Flux.range(1,5)
        .delayElements(Duration.ofMillis(200));
    Flux<Integer> sixToTen = Flux.range(6,5)
        .delayElements(Duration.ofMillis(400));
    Flux.concat(oneToFive, sixToTen)
        .subscribe(System.out::println);
    Thread.sleep(5000);
}
```

- Noter l'utilisation de l'opérateur `delayElements` pour retarder la production de chaque donnée (élément)
- Un autre opérateur `merge` mélange les données des deux publishers (le tester pour voir la différence)

Opérateur zip

- Transforme les données des publishers et produit un nouveau publisher qui produit les données transformées

```
@Test
public void fluxWithZip () {
    Flux<Integer> oneToFive = Flux.range(1,5);
    Flux<Integer> sixToTen = Flux.range(6,5);
    Flux.zip(oneToFive, sixToTen,
        (item1, item2) -> item1 + "-" + item2)
        .subscribe(System.out::println);
}
```

Tester pour voir

Sur le site de *Project Reactor*, vous avez une bonne documentation, et notamment dans l'annexe A, on vous indique quel opérateur utiliser dans vos programmes

Plan du cours

1. *API Reactive Streams* et son implem *Reactor*
2. Développer une API REST réactive avec des contrôleurs annotés
3. Développer une API REST réactive avec des *endpoints* fonctionnels

Spring Web Flux vs Spring MVC

- Spring Web Flux est une alternative à Spring MVC, qui permet d'écrire des app Web réactives

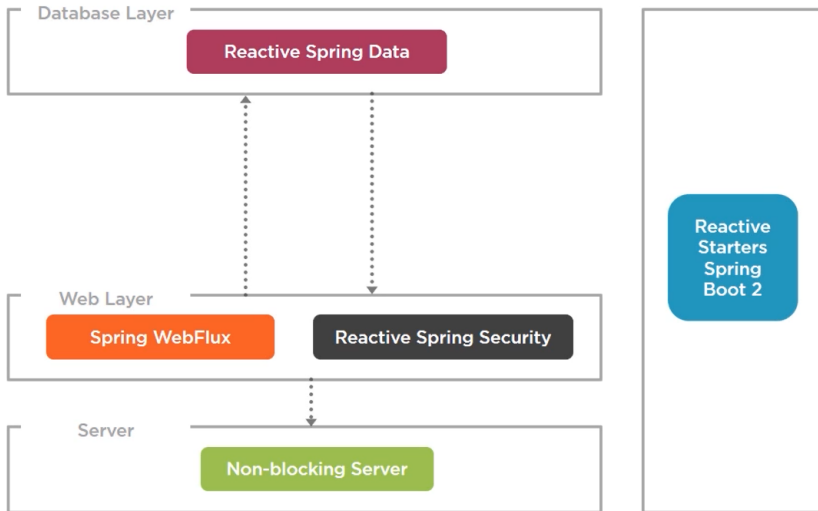
Spring MVC	Spring WebFlux
Servlet API	Reactive Streams
Blocking API	Non-blocking API (Servlet 3.1+)
Synchronous	Asynchronous
One request per thread	Concurrent connections with few threads

Spring Web Flux vs Spring MVC

Annotations	Functional Endpoints
spring-web-mvc	spring-web-reactive
Servlet API	HTTP / Reactive Streams
Servlet Container	Netty, Tomcat, Jetty, Undertow

Dans cette section, on voit comment utiliser les annotations pour écrire une API REST réactive

Reactive partout



Exemple que l'on va écrire dans cette partie

- Un contrôleur annoté WebFlux pour envoyer des informations (des nouvelles) Covid aux utilisateurs
 - Les annotations WebFlux et Web MVC sont les mêmes (@Controller (contrôleur de vues), @RestController (contrôleur d'API Rest – données JSON par ex), @RequestMapping, @GetMapping, ...)
- Nous allons utiliser une base de données NoSQL (à documents) MongoDB pour stocker les infos COVID
- Nous allons utiliser *Server Sent Events* (cours IG3-FAR sur les websockets) comme moyen de faire du Server Push

Requêtes et réponses HTTP réactives

- Spring Web Flux introduit des classes particulières pour représenter les requêtes et réponses HTTP réactives : `ServerHttpRequest` et `ServerHttpResponse`
- Les méthodes des contrôleurs peuvent utiliser les types habituels dans Spring MVC (`String`, `ResponseEntity`, ...) ou bien `Mono` et `Flux` comme type de retour pour retourner des données (voir les exemples qui vont suivre)

Initialiser un nouveau projet

- Aller sur le site Spring Initializr : <https://start.spring.io/>
- Configurer un nouveau projet (Gradle & Java) en ajoutant les dépendances suivantes :
 - Reactive Web
 - Reactive MongoDB (qui vient avec le driver *reactive* de MongoDB)
 - Embedded MongoDB (serveur de Bdd)

Initialiser un nouveau projet -suite-

- Embedded MongoDB est considéré comme un serveur à utiliser pour les tests. Si vous avez dans votre script de build `testImplementation` devant cette dépendance, changez la en `implementation`
- Il faudrait avoir les dépendances suivantes :

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-  
        starter-webflux'  
    implementation 'org.springframework.boot:spring-boot-  
        starter-data-mongodb-reactive'  
    implementation 'de.flapdoodle.embed:de.flapdoodle.embed.  
        mongo'  
    testImplementation 'org.springframework.boot:spring-boot-  
        starter-test'  
    testImplementation 'io.projectreactor:reactor-test'  
}
```

Le modèle dans l'application

- Créer un package model et une classe CovidInfo dans ce package
- Dans cette classe, ajouter :
 - une annotation @Document (pour que nos entités correspondent à des documents dans MongoDB)
 - un attribut annoté @Id de type String et nommé id
 - les attributs suivants :

```
private String title; // titre de l'info Covid
private String content; // contenu de l'info
private LocalDateTime publicationDate;
```

- Un constructeur sans paramètre et 1, avec tous les attributs
- Générer avec IntelliJ les getters, setters, equals, hashCode et toString

Le repository

- Créer un package repository et une interface

CovidInfoRepository :

```
import org.springframework.data.mongodb.repository.  
    ReactiveMongoRepository;  
import poly.mtp.infoapi.model.CovidInfo;  
import reactor.core.publisher.Flux;  
import java.time.LocalDateTime;  
public interface CovidInfoRepository  
    extends ReactiveMongoRepository<CovidInfo, String> {  
    Flux<CovidInfo> findByTitleOrderByPublicationDate(String  
        title);  
    Flux<CovidInfo> findByPublicationDateOrderByTitle(Date  
        publicationDate);  
}
```

Notez les méthodes ajoutées à l'interface pour faire des find avec autre chose que l'id (ces méthodes doivent respecter des conventions de nommage Spring Data)

Mettre en place la base de données

- Dans la classe annotée `@SpringBootApplication`, ajouter l'annotation `@EnableMongoRepositories`
- Y ajouter la méthode suivante pour initialiser la BdD :

```
@Bean
CommandLineRunner init(CovidInfoRepository repository) {
    return args -> {
        Flux<CovidInfo> covidInfoFlux = Flux.just(
            new CovidInfo(null, "Point de situation",
                "Nouveaux cas confirmés ...", LocalDateTime.now()),
            new CovidInfo(null, "Informations sur les mesures
                nationales", "Le 22.11.2021 ..", LocalDateTime.now()))
        .flatMap(repository::save);
        // Just to check if data has been inserted
        covidInfoFlux.thenMany(repository.findAll())
            .subscribe(System.out::println);
    };
}
```

Tester votre application

- Faire un `bootRun` pour démarrer l'application
- Si jamais vous avez une erreur lors du téléchargement de MongoDB (téléchargement trop lent, connexion instable, ...), télécharger à la main le ZIP du site <http://downloads.mongodb.org> et l'installer dans le dossier `.embedmongo` de votre `user.home`
- Ceci permet à Gradle d'aller le chercher dans ce dossier plutôt que de le télécharger
- Il faut que vous soyez capables de voir les données, insérées dans la base, affichées dans la console par le `println` du `subscriber` ci-dessus

Le contrôleur

- Ajouter un package controller et une classe CovidInfoController annotée de la façon suivante :

```
@RestController  
@RequestMapping("/infos")
```

- Ajouter un attribut auto-injecté pour avoir accès au repository :

```
@Autowired  
CovidInfoRepository repository;
```

- Ajouter une méthode Get (pour obtenir toutes les infos) :

```
@GetMapping  
public Flux<CovidInfo> getAllInfos() {  
    return repository.findAll();  
}
```

Noter le type de retour Flux. Pas de *subscriber* ici, Spring le fera automatiquement

Autres méthodes du contrôleur

- Obtenir une info avec son ID :

```
@GetMapping("/{id}")  
public Mono<ResponseEntity<CovidInfo>> getInfo(  
    @PathVariable String id) {  
    return repository.findById(id).map(covidInfo ->  
        ResponseEntity.ok(covidInfo))  
        .defaultIfEmpty(ResponseEntity.notFound().build());  
}
```

- Noter le type de retour Mono (une réponse unique)
- Retourner le résultat de findById provoque l'envoi d'un empty (réponse 200) si jamais l'info avec l'Id reçu n'existe pas
- C'est pour cela que cette méthode utilise map pour retourner une réponse 200 avec l'info Covid ou bien retourne une réponse 404 (type de retour : Mono<ResponseEntity<CovidInfo> >)

Autres méthodes du contrôleur -suite-

- Méthode POST simplifiée :

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Mono<CovidInfo> saveCovidInfo(@RequestBody
    CovidInfo covidInfo) {
    return repository.save(covidInfo);
}
```

- Méthode deleteAll :

```
@DeleteMapping
public Mono<Void> deleteAllProducts() {
    return repository.deleteAll();
}
```

Autres méthodes du contrôleur -suite-

- Méthode delete une info :

```
@DeleteMapping("{id}")
public Mono<ResponseEntity<Void>> deleteInfo(
    @PathVariable("id") String id) {
    return repository.findById(id)
        .flatMap(existingCovidInfo ->
            repository.delete(existingCovidInfo)
                .then(Mono.just(ResponseEntity.ok().<Void>build())))
        .defaultIfEmpty(ResponseEntity.notFound().build());
}
```

Noter l'utilisation de flatMap pour produire, à partir du premier Mono qui a fait une recherche de l'info, un deuxième Mono qui, après avoir supprimé l'info, construit une réponse Void

Autres méthodes du contrôleur -suite-

- Méthode de mise à jour d'une info Covid :

```
@PutMapping("{id}")
public Mono<ResponseEntity<CovidInfo>> updateInfo(
    @PathVariable("id") String id,
    CovidInfo covidInfo) {
    return repository.findById(id)
        .flatMap(existingCovidInfo -> {
            existingCovidInfo.setContent(covidInfo.getContent());
            existingCovidInfo.setTitle(covidInfo.getTitle());
            existingCovidInfo.setPublicationDate(covidInfo.
                getPublicationDate());
            return repository.save(existingCovidInfo);
        })
        .map(updatedInfo -> ResponseEntity.ok(updatedInfo))
        .defaultIfEmpty(ResponseEntity.notFound().build());
}
```

Noter les flatMap puis map pour construire la réponse

Tester l'application

- Démarrer votre application
- Tester le contrôleur avec des requêtes vers les différentes routes sur votre navigateur et sur Postman ou Curl

Méthode dans le contrôleur pour *SSE : Server Sent Events*

- Nous allons ajouter une méthode dans le contrôleur pour retourner les infos, mais dans une réponse HTTP avec les *SSE : Server Sent Events* pour que le serveur effectue des push (quand il y a des nouvelles infos COVID) sans que le client n'envoie une requête
- Nous allons d'abord ajouter dans le modèle une classe CovidInfoEvent avec les deux attributs suivants :

```
private long eventId;  
private String eventType;
```

Y ajouter deux constructeurs (sans param et avec des param pour initialiser tous les attributs), des getters/setters, equals, hashCode et toString

Méthode dans le contrôleur pour SSE : *Server Sent Events*

- Ajouter la méthode suivante au contrôleur :

```
@GetMapping(value = "/events", produces = MediaType.  
    TEXT_EVENT_STREAM_VALUE)  
public Flux<CovidInfoEvent> getInfoEvents() {  
    return Flux.interval(Duration.ofSeconds(1))  
        .map(val ->  
            new CovidInfoEvent(val, "Covid Info Event"));  
}
```

Tester cette route depuis votre navigateur (Chrome supporte les SSE)

Client HTML à cette route

- Créer un dossier public sous le dossier resources
- Ajouter un fichier index.html dans le dossier resources/public
- Y ajouter le code suivant : à tester

```
<h3>Here are the infos received from Covid Server</h3>
<ul></ul>
<script>
if (!!window.EventSource) {
  let eventSource = new EventSource("infos/events/");
  let eventList = document.querySelector("ul");
  eventSource.onmessage = function(e) {
    let newElement = document.createElement("li");
    newElement.textContent = "Event: " + e.data;
    eventList.append(newElement);
  } }
else { alert("Your browser does not support SSE"); }
</script>
```

Plan du cours

1. API *Reactive Streams* et son implem *Reactor*
2. Développer une API REST réactive avec des contrôleurs annotés
3. Développer une API REST réactive avec des *endpoints* fonctionnels

Endpoints fonctionnels

- C'est une alternative aux contrôleurs annotés
- S'appuyer sur le style de programmation fonctionnelle
- Il s'agit d'écrire des fonctions qui ne sont pas annotées comme ce que l'on a vu dans la section précédente, mais plutôt retourner des lambdas
- Il s'agit d'écrire : 1) des fonctions *Handler* et 2) une fonction *Router*

Fonctions *Handler*

- N'importe quelle méthode qui reçoit un objet `ServerRequest` (`org.springframework.web.reactive.function.server.ServerRequest`) comme paramètre et qui retourne un objet `Mono<ServerResponse>`
- Les deux sont des objets gérés de façon asynchrone
- Exemple : fonction handler pour des requêtes Get (toutes les infos Covid) :

```
public Mono<ServerResponse> getAllInfos (ServerRequest
    request) {
    Flux<CovidInfo> infos = repository.findAll();
    return ServerResponse.ok()
        .contentType(MediaType.APPLICATION_JSON)
        .body(infos, CovidInfo.class);
}
```

Fonction *Router*

- Cette fonction traite les requêtes Http en premier
- C'est une méthode qui prend en paramètre un objet de type `ServerRequest` et qui retourne un objet

`RouterFunction<ServerResponse>` :

```
RouterFunction <ServerResponse> routes ( CovidInfoHandler  
    handler ) {  
    // ...  
}
```

`CovidInfoHandler` est la classe qui contient les fonction handlers

Fonction Router -suite-

On définit dans la fonction précédente les routes :

```
RouterFunctions.route(RequestPredicate, HandlerFunction)
```

- RequestPredicate est une interface fonctionnelle qui représente une fonction qui permet de tester si la requête match avec un certain chemin (*path*) ou une méthode HTTP
- HandlerFunction est une référence vers une fonction handler ou une lambda dans laquelle on invoque une méthode handler
- Il existe plusieurs méthodes statiques qu'on peut utiliser comme RequestPredicate, comme accept(MediaType...), method(HttpMethod), path(String), ... de la classe :

org.springframework.web.reactive.function.server.RequestPredicates

```
RouterFunctions.route(GET("/infos").and(accept(
    APPLICATION_JSON)), handler::getAllInfos)
```

Projet à mettre en place

- Générer un projet sur Spring Initializr équivalent au projet précédent (mêmes dépendances)
- Remplacer testImplementation par implementation pour Embedded MongoDB
- Créer un package model et y mettre la classe CovidInfo (la même qu'avant)
- Ajouter dans le même package la classe CovidInfoEvent (la même qu'avant)
- Créer un package repository et y mettre l'interface CovidInfoRepository (la même qu'avant)
- Ajouter dans la classe qui contient main la même méthode d'initialisation de la Bdd (la méthode init)

Projet à mettre en place

- Créer un package handler
- Dans ce package, créer la classe CovidInfoHandler et l'annoter @Component
- Ajouter dans cette classe un attribut de type CovidInfoRespository
- Ajouter un constructeur avec un paramètre de type CovidInfoRespository qu'il faudra utiliser pour initialiser l'attribut. Spring injectera automatiquement le paramètre du constructeur quand il créera le bean CovidInfoHandler
- On va maintenant définir les fonctions handler

Fonctions Handler Get

```
public Mono<ServerResponse> getAllInfos(  
    ServerRequest request) {  
    Flux<CovidInfo> infos = repository.findAll();  
    return ServerResponse.ok()  
        .contentType(MediaType.APPLICATION_JSON)  
        .body(infos, CovidInfo.class);  
}
```

Méthode qui invoque findAll() du repository et construit un Mono de ServerResponse avec dans son body les objets CovidInfo

Fonctions Handler Get -suite-

```
public Mono<ServerResponse> getCovidInfo(  
    ServerRequest request) {  
    String id = request.pathVariable("id");  
    Mono<CovidInfo> infoMono = repository.findById(id);  
    Mono<ServerResponse> notFound = ServerResponse.notFound().  
        build();  
    return infoMono.flatMap(  
        covidInfo -> ServerResponse.ok()  
            .contentType(MediaType.APPLICATION_JSON)  
            .body(covidInfo, CovidInfo.class))  
        .switchIfEmpty(notFound); // un if-else reactive  
    }
```

- Cette méthode récupère l'id de l'objet request reçu en paramètre, fait un findById, prévoit une réponse NotFound et enfin construit le Mono de ServerResponse avec soit la réponse avec dans son body l'info covid, soit la réponse 404

Fonction Handler Post

```
public Mono<ServerResponse> saveCovidInfo (ServerRequest  
    request) {  
    Mono<CovidInfo> infoMono = request.bodyToMono(CovidInfo.  
        class);  
    return infoMono.flatMap(  
        covidInfo -> ServerResponse.status(HttpStatus.CREATED)  
            .contentType(MediaType.APPLICATION_JSON)  
            .body(repository.save(covidInfo), CovidInfo.class));  
}
```

- Même schéma, avec une réponse avec le code 201 (CREATED)

Fonction Handler Put

```
public Mono<ServerResponse> updateCovidInfo(ServerRequest request) {  
    String id = request.pathVariable("id");  
    Mono<CovidInfo> existingInfoMono=repository.findById(id);  
    Mono<CovidInfo> infoMono=request.bodyToMono(CovidInfo.class);  
    Mono<ServerResponse> notFound=ServerResponse.notFound().build();  
  
    return infoMono.zipWith(existingInfoMono,  
        (CovidInfo covidInfo, CovidInfo existingCovidInfo) ->  
        new CovidInfo(existingCovidInfo.getId(),  
            covidInfo.getTitle(), covidInfo.getContent(),  
            covidInfo.getPublicationDate())).flatMap(covidInfo  
        -> ServerResponse.ok().contentType(MediaType.APPLICATION_JSON)  
            .body(covidInfo, CovidInfo.class))  
        .switchIfEmpty(notFound);  
}
```

- Noter l'utilisation de l'opérateur zip
- Noter ici le new CovidInfo() pour respecter le style de programmation fonctionnelle (objets immuables uniquement), au lieu de transformer l'objet existingCovidInfo (en appelant ses setters)

Fonctions Handlers Delete

```
public Mono<ServerResponse> deleteCovidInfo(  
    ServerRequest request) {  
    String id = request.pathVariable("id");  
    Mono<CovidInfo> infoMono = repository.findById(id);  
    Mono<ServerResponse> notFound = ServerResponse.notFound().  
        build();  
  
    return infoMono.flatMap(  
        existingCovidInfo -> ServerResponse.ok()  
        .build(repository.delete(existingCovidInfo))  
        .switchIfEmpty(notFound);  
    }  
public Mono<ServerResponse> deleteAllInfos(  
    ServerRequest request) {  
    return ServerResponse.ok()  
        .build(repository.deleteAll());  
    }  
}
```

- Même schéma que précédemment

Fonction Handler Get SSE

```
public Mono<ServerResponse> getInfoEvents(  
    ServerRequest request) {  
    Flux<CovidInfoEvent> eventsFlux =  
        Flux.interval(Duration.ofSeconds(1))  
            .map(val -> new CovidInfoEvent(val, "Covid Info Event"));  
    return ServerResponse.ok()  
        .contentType(MediaType.TEXT_EVENT_STREAM)  
        .body(eventsFlux, CovidInfoEvent.class);  
}
```

- Le même principe que pour le contrôleur annoté mais en utilisant la prog. fonctionnelle

Fonction Router

```
@Bean
RouterFunction<ServerResponse> routes(CovidInfoHandler handler) {
    return route(GET("/infos").and(accept(APPLICATION_JSON)),
        handler::getAllInfos)
        .andRoute(POST("/infos").and(contentType(APPLICATION_JSON)),
            handler::saveCovidInfo)
        .andRoute(DELETE("/infos").and(accept(APPLICATION_JSON)),
            handler::deleteAllInfos)
        .andRoute(GET("/infos/events").and(accept(TEXT_EVENT_STREAM)),
            handler::getInfoEvents)
        .andRoute(GET("/infos/{id}").and(accept(APPLICATION_JSON)),
            handler::getCovidInfo)
        .andRoute(PUT("/infos/{id}").and(contentType(APPLICATION_JSON)),
            handler::updateCovidInfo)
        .andRoute(DELETE("/infos/{id}").and(accept(APPLICATION_JSON)),
            handler::deleteCovidInfo);
}
```

- Important d'écrire les routes dans le bon ordre
- Ex : /infos/events avant /infos/id sinon le mot events sera considéré comme id pour la route Get une info Covid (pas SSE)

Conclusion & Références

Conclusion :

- Une autre façon d'écrire une API Web avec Spring (Web Flux)
- Une API Web asynchrone et non-bloquante grâce à la prog. réactive
- Une API écrite avec le style de prog. fonctionnelle
- Objectif : scalabilité et faire du server push
- On peut écrire des programmes clients avec WebClient et des tests avec WebClient : <https://www.baeldung.com/spring-5-webclient>

Références biblio :

- Site Web de Spring Web Flux : <https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html>
- Tutoriels sur Pluralsight et Baeldung

