





# Spring réactif

---

David THIBAU – 2024

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- Introduction

- La programmation réactive

- Reactor

- Cœur de l'API : Mono et Flux
- Threads, Scheduler
- Gestion des erreurs, debugging, Tracing
- Propagation de contexte

- Pile réactive

- SpringData
- Spring WebFlux
- Client réactif
- Server-side Event
- Reactive websockets
- Sécurité

- Tests et Spring Cœur

- Spring cœur et nouvelles annotations
- Annotations JUnit
- Test des clients web



# Introduction

---

## **Programmation Réactive**



# IO asynchrones

---

La programmation réactive s'appuie sur les entrées/sorties asynchrones.

L'idée est simple :

Récupérer des ressources qui seraient en attente d'E/S.

=> Un client d'un flux de données est informé lors les nouvelles données sont prêtes au lieu de les demander



# Apports / Complexité

---

L'implémentation à partir d'I/O asynchrones est nettement plus complexe.

Alors en a t on vraiment besoin ?

- Si votre appli Web est capable de produire des réponses aux requêtes entrantes plus rapidement que la vitesse à laquelle les requêtes arrivent : NON
- Si votre appli est limitée par les I/O : OUI



# Abstraction

---

Un client d'un flux de données est désormais informé lorsqu'une donnée est disponible.

Le code est alors organisé en

- Des **Publishers/Observable** qui émettent des données
  - Des **Subscribers** qui réagissent à l'arrivée de données
- 
- ✓ Les Publishers peuvent envoyer un nombre illimité de données.
  - ✓ Ils peuvent avoir un nombre illimité d'abonnés.
  - ✓ Ils sont actifs tant qu'il a au moins un abonné.
  - ✓ Le flux est temps réel, dès qu'une donnée est publiée, elle est traitée par les abonnés



# Traitement des évènements

Les évènements reçus au fur et à mesure doivent être traité : La programmation fonctionnelle est idéale pour définir les traitements.

L'API **ReactiveX<sup>1</sup>** formalise un ensemble de types d'opérations pouvant être effectuées sur ces flux d'évènements : les opérateurs

Les opérateurs ou processeurs produisent eux même des flux d'évènements et peuvent être combinés



1. De nombreuses implémentations existent (RxJS, RxJava, Rx.NET)

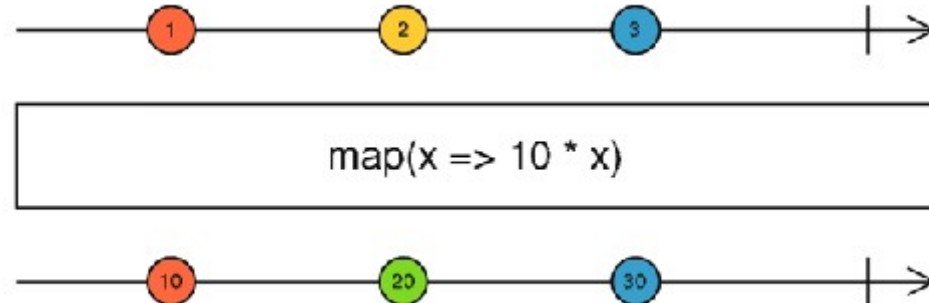




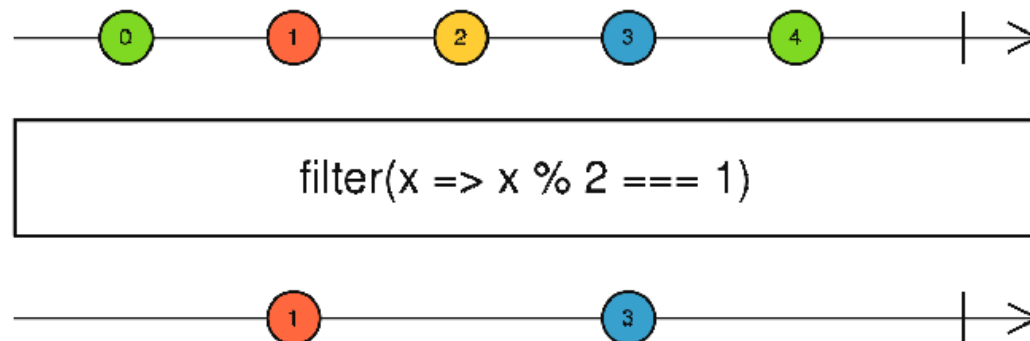
# Marble diagrams

Les opérateurs sont documentés via des ***marble diagrams***

Exemple *map* :



Exemple *filter*





# Back-pressure

---

Les consommateurs ne connaissent pas la taille du flux et ne savent pas quel volume de données sera poussé par le producteur.

Cela peut conduire à un débordement du consommateur

Il faut donc donner la possibilité au consommateur de communiquer au producteur la quantité de données qu'il est prêt à gérer.

=> ***Back-pressure***



# Reactive Streams

---

***Reactive Streams*** a pour but de définir un standard pour le traitement asynchrone de flux d'événements offrant une fonctionnalité de ***non-blocking back pressure***

Il concerne les environnements Java et Javascript ainsi que les protocoles réseau

Le standard permet l'inter-opérabilité mais reste très bas-niveau

Il fournit un ensemble de processeurs/ opérateurs inspiré de *ReactiveX*



# Interfaces Reactive Streams

---

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

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

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

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



# Back pressure

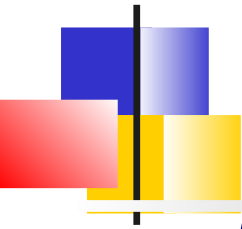
---

La notion de ***back pressure*** décrit la possibilité des abonnés de contrôler la cadence d'émission des événements du service qui publie.

*Reactive Stream* permet d'établir le mécanisme et de fixer les limites de cadence via la méthode :

***void request(long n) de Subscription***

Si l'*Observable* ne peut pas ralentir, il doit prendre la décision de bufferiser, supprimer ou tomber en erreur.



# *Reactor*

---

***Reactor*** se concentre sur la programmation réactive côté serveur.

Il est développé conjointement avec Spring.

- Il fournit principalement les types de plus haut niveau ***Mono*** et ***Flux*** représentant des séquences d'événements
- Il offre un ensemble d'opérateurs alignés sur *ReactiveX*.
- C'est une implémentation de *Reactive Streams*



# Eco-système Reactive Spring

---

**Reactor** est la brique de base de la pile réactive de Spring.

Il est utilisé dans tous les projets réactifs :

- **Spring WebFlux** des services web back-end réactifs. Alternative à *Spring MVC*
- **Spring Data** : *MongoDB, Cassandra, R2DBC*
- **Spring Reactive Security** : Sécurité réactive
- **Spring Test** : Support pour le réactif
- **Spring Cloud** : Projets micro-services. En particulier *Spring Cloud Data Stream, Spring Gateway, ...*



# Reactor

---

## **Cœur de l'API**

Threads et Scheduler

Traitement des Erreurs

Test, Debug

Propagation de contexte





# Dépendance Maven

---

```
<dependency>
```

```
  <groupId>io.projectreactor</groupId>
```

```
  <artifactId>reactor-core</artifactId>
```

```
  <version>${version}</version>
```

```
</dependency>
```



# 2 Types

---

*Reactor* offre principalement 2 types Java :

- ***Mono*** : Flux de 0..1 éléments
- ***Flux*** : Flux de 0..N éléments

Tous les 2 sont des implémentations de  
l'interface ***Publisher*** de *Reactive Stream*

*Mono* offre un sous-ensemble des  
opérateurs de Flux



# Méthodes des abonnés

---

Un ***Flux<T>*** représente une séquence asynchrone de 0 à N événements, optionnellement terminée par un signal de fin ou une erreur.

***Mono<T>*** représente une séquence de 0 à 1 événement, optionnellement terminée par un signal de fin ou une erreur

Les événements sont traduits par des appels de méthode sur les abonnés :

- Nouvelle valeur : *onNext()*
- Signal de fin : *onComplete()*
- Erreur : *onError()*



# Production d'un flux de données

---

La façon la plus simple de créer un *Mono* ou un *Flux* est d'utiliser les méthodes *Factory* à disposition.

```
Mono<Void> m1 = Mono.empty()
Mono<String> m2 = Mono.just("a");
Mono<Book> m3 = Mono.fromCallable(() -> new Book());
Mono<Book> m4 = mono.fromFuture(myCompletableFuture);
```

```
Flux<String> f1 = Flux.just("a","b","c");
Flux<Integer> f2 = Flux.range(0, 10);
Flux<Long> f3 = Flux.interval(Duration.ofMillis(1000).take(10);
Flux<String> f4 = Flux.fromIterable(bookCollection);
Flux<Book> f5 = Flux.fromStream(bookCollection.stream());
```



# Encapsulation méthodes asynchrones

---

Mono peut être très utile pour encapsuler des opérations asynchrones telles que des requêtes HTTP ou de base de données via ses méthodes

```
fromCallable(Callable) ,  
fromRunnable(Runnable) ,  
fromSupplier(Supplier) ,  
fromFuture(CompletableFuture) ,  
fromCompletionStage(CompletionStage)
```

Ex :

```
Mono<String> stream8 = Mono.fromCallable(() ->  
    httpRequest());
```



# Abonnement

---

L'abonnement au flux s'effectue via la méthode ***subscribe()***

Généralement, des lambda-expressions sont utilisées

**// Déclenche le flux**

```
subscribe();  
subscribe(Consumer<? super T> consumer);  
subscribe(Consumer<? super T> consumer,  
          Consumer<? super Throwable> errorConsumer);  
subscribe(Consumer<? super T> consumer,  
          Consumer<? super Throwable> errorConsumer,  
          Runnable completeConsumer);
```



# Interface *Subscriber*

---

Sans utiliser les lambda-expression, on peut fournir une implémentation de l'interface ***Subscriber*** qui définit 4 méthodes :

```
void onComplete()  
void onError(java.lang.Throwable t)  
void onNext(T t)  
void onSubscribe(Subscription s)
```

Invoquées après

*Publisher.subscribe(Subscriber)*



# *Subscription*

---

***Subscription*** représente un abonnement d'un (seul) abonné à un *Publisher*.

Il est utilisé

- pour demander l'émission d'événement  
**`void request(long n)`**
- Pour annuler la demande et permettre la libération de ressource  
**`void cancel()`**





# Exemple

---

```
Flux.just(1, 2, 3, 4)
    .log()
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onSubscribe(Subscription s) {
            s.request(Long.MAX_VALUE); // Provoque l'émission de tous les évts
        }

        @Override
        public void onNext(Integer integer) {
            elements.add(integer);
        }

        @Override
        public void onError(Throwable t) {}

        @Override
        public void onComplete() {}
    });
```



# Sinks

---

Les Sinks permettent de pousser programmatiquement des événements vers un flux réactif : Flux ou Mono.

Spring Reactor fournit 2 types de Sinks :

- Sink.one() met un seul élément ou une erreur. => Mono.
- Sink.many() : Émet plusieurs éléments. => Flux.

Ils exposent des méthodes *tryEmit\** qui permettent d'envoyer une donnée, une erreur ou un signal de terminaison.

Principalement utilisé par les bibliothèques réactives



# Variantes Sinks.many()

---

Plusieurs variantes de *Sinks.many()*

- ***Sink.many().unicast()*** : Permet à un seul abonné de recevoir des éléments émis séquentiellement.
- ***Sink.many().multicast()*** : Permet à plusieurs abonnés de recevoir des éléments émis.
- ***Sink.many().replay()*** : Permet de rejouer les éléments émis à de nouveaux abonnés.



# Exemple

---

```
// Création d'un Sink pouvant émettre plusieurs évènements
// vers un seul abonné avec bufferisation éventuelle des événements
Sinks.Many<String> sink = Sinks.many().unicast().onBackPressureBuffer();

// Conversion en flux et abonnement
sink.asFlux().subscribe(System.out::println);

// Émission d'éléments dans le flux
sink.tryEmitNext("Element 1");
sink.tryEmitNext("Element 2");

// Émission d'une erreur dans le flux
sink.tryEmitError(new RuntimeException("Something went wrong"));

// Complétion du flux
sink.tryEmitComplete();
```



# FluxSink et MonoSink

**FluxSink** et **MonoSink** permettent également d'émettre des événements programmatiquement dans un Flux ou un Mono. Exemples :

```
Flux<Integer> flux = Flux.create(sink -> {  
    for (int i = 1; i <= 5; i++) {  
        sink.next(i);  
    }  
    sink.complete();  
});
```

```
Mono<String> mono = Mono.create(sink -> {  
    try {  
        // Simulate some asynchronous processing  
        String result = someAsyncOperation();  
        sink.success(result);  
    } catch (Exception e) {  
        sink.error(e);  
    }  
});
```



# Opérateurs

---

Les opérateurs permettent différents types d'opérations sur les éléments de la séquence :

- Transformer
- Choisir des événements
- Filtrer
- Gérer des erreurs
- Opérateurs temporels
- Séparer un flux
- Revenir au mode synchrone

Ils n'affectent pas le *Publisher* sur lequel ils opèrent.

Ils créent de nouveaux *Publisher*, chaque *Publisher* étant immuable.



# Transformation

---

1 vers 1 :

**map** (nouvel objet), **cast** (chgt de type), **index** (Tuple avec ajout d'un indice)

1 vers N :

**flatMap**, **concatMap** (*garantie l'ordre*), **switchMap**

Ajouter des éléments à une séquence :

**startsWith**, **concatWith**, **concatWithValues**

Agréger => transformer en Mono :

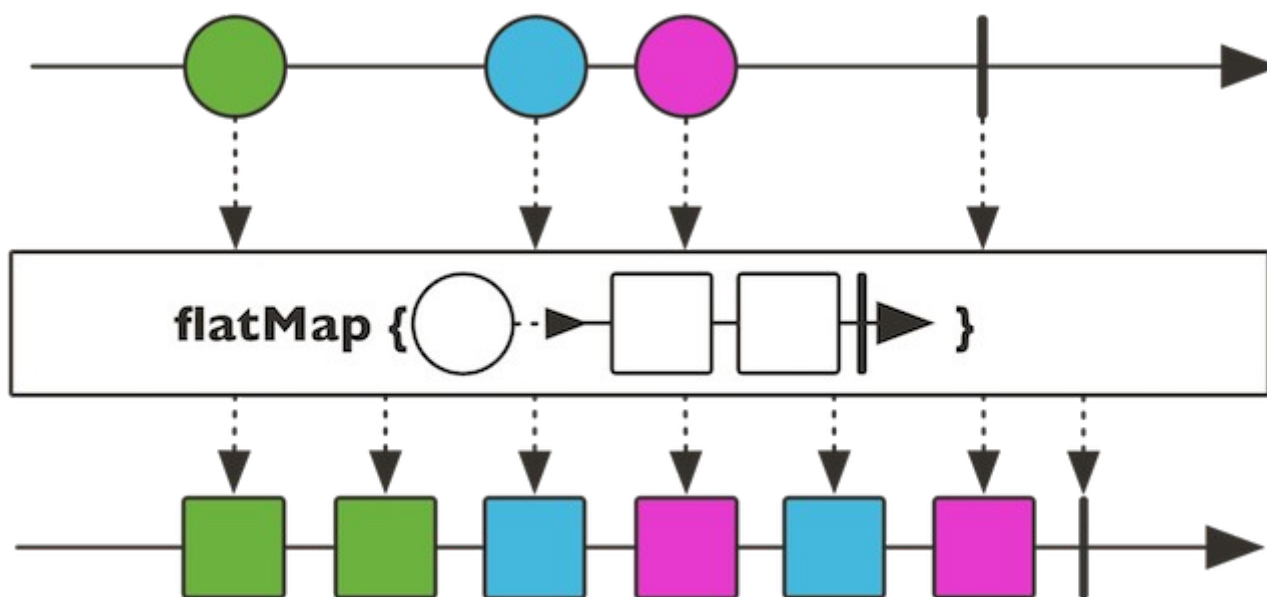
**collectList**, **collectMap**, **count**, **reduce**, **scan**,

Agréger en booléen :

**all**, **any**, **hasElements**, **hasElement**



# *flatMap*







# Filtres

---

Filtre sur fonction arbitraire :

***filter***

Sur le type :

***ofType***

Ignorer toutes les valeurs :

***ignoreElements***

Ignorer les doublons :

***distinct***

Seulement un sous-ensemble :

***take, takeLast, elementAt***

Skipper des éléments :

***skip(Long | Duration), skipWhile***



# Opérateurs temporels

---

Associé l'événement à un timestamp :

***elapsed, timestamp***

Séquence interrompue si délai trop important  
entre 2 événements :

***timeout***

Séquence à intervalle régulier :

***interval***

Ajouter des délais

***Mono.delay, delayElements,  
delaySubscription***



# Combinaison de Flux

---

Les opérateurs de combinaison de Flux permettent de combiner les éléments émis par plusieurs flux en un seul flux

***zipWith*** : Combiner les évènements via une lambda  
`flux1.zipWith(flux2, (x, y) -> x + y)`

***merge*** : Fusionne les flux en 1 seul flux, les éléments sont produit dans leur ordre d'apparition

***concat*** : Mettre bout à bout les flux

***combineLatest*** : Combine les dernier éléments via une lambda



# Enchaînement

Les méthodes ***then()*** et ***thenMany()*** permettent de spécifier un flux après la complétion d'un autre flux.

```
Flux<Integer> flux1 = Flux.just(1, 2, 3);  
Flux<Integer> flux2 = Flux.just(4, 5, 6);
```

```
flux1  
    .then(Mono.just(100)) // Exécute Mono.just(100) après la fin de flux1  
    .subscribe(System.out::println); // Affiche 100
```

```
flux1  
    .thenMany(flux2) // Exécute flux2 après la fin de flux1  
    .subscribe(System.out::println); // Affiche 4, 5, 6
```



# Opérateurs *doOn\**

Les opérateurs ***doOn\**** ne travaillent pas sur les événements mais effectue une action //<sup>1</sup> lors de la réception de l'événement.

```
Flux<Integer> on = Flux.<Integer>create(sink -> {  
    sink.next(1);  
    sink.next(2);  
    sink.next(3);  
    sink.error(new IllegalArgumentException("oops!"));  
    sink.complete();  
})//  
    .doOnNext(nextValues::add) //  
    .doOnEach(signals::add)//  
    .doOnSubscribe(subscriptions::add)//  
    .doOnError(IllegalArgumentException.class, exceptions::add)//  
    .doFinally(finallySignals::add);
```



# *handle*

La méthode ***handle*** disponible permet un contrôle total sur les événements entrants et sortants.

Flux//

```
.range(0, MAX_VALUE) ①
.handle((value, sink) -> {
    var upTo = Stream.iterate(0, i -> i < NB_ERRORS, i -> i + 1)
        .collect(Collectors.toList());
    if (upTo.contains(value)) {
        sink.next(value);
        return;
    }
    if (value == numberToError) {
        sink.error(new IllegalArgumentException("No 4 for you!"));
        return;
    }
    sink.complete();
});
```



# Trace

---

L'opérateur ***log()*** permet de tracer les événements de la séquence

- Chaîné dans une séquence, il récupère tous les événements du stream amont (*onNext, onError, onComplete + subscriptions, cancellations et requests*).
- Il utilise la classe utilitaire *Loggers* qui retrouve le framework de logging (log4j, logback)



# Exemple

---

```
Flux<Integer> flux = Flux.range(1, 10)
                        .log()
                        .take(3);

flux.subscribe()
```

```
-----
10:45:20.200 [main] INFO reactor.Flux.Range.1 - |
           onSubscribe([Synchronous Fuseable] FluxRange.RangeSubscription)
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | request(unbounded)
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | onNext(1)
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | onNext(2)
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | onNext(3)
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | cancel()
```





# Reactor

---

Cœur de l'API

**Threads et Scheduler**

Traitement des Erreurs

Test, Debug

Propagation de contexte



# Event-loop

---

*Reactor* est une boucle d'événements<sup>1</sup> : il utilise un *Scheduler* (pool de threads) pour affecter le travail aussi rapidement que possible.

- Le runtime déplace sans arrêt le flot d'exécution des flux d'une thread à une autre.
- Par défaut, tout le code s'exécute sur les threads de ce *Scheduler*.
- Ce *Scheduler* global crée un thread par cœur sur la machine cible

=> Ne jamais bloquer une thread de ce Scheduler !!



# Contrôle de la concurrence

---

*Spring Reactor* offre du support pour contrôler la concurrence et l'utilisation des Threads

L'interface ***Scheduler*** permet de déléguer le traitement des événements à des nouvelles threads séparés

Cela peut être utile lorsque l'on a des traitements bloquants

Les objets *Flux*, *Mono* ou *Subscriber* n'ont pas de dépendance sur le modèle de concurrence. Ils utilisent des instances de *Scheduler* via les méthodes :

- ***publishOn***
- ***subscribeOn***



# Scheduler

---

De nombreuses implémentations de **Scheduler** sont fournies par le biais de factory :

- ***Schedulers.fromExecutorService(ExecutorService)*** : Pool de threads à partir de *ExecutorService*.
- ***Schedulers.newParallel(java.lang.String)*** : Optimisé pour des exécutions rapides de *Runnable*
- ***Schedulers.single()*** : Un seul thread
- ***Schedulers.immediate()*** : Exécution immédiate des tâches sur la thread de l'appelant.
- ***Schedulers.boundedElastic()*** : Pool de threads avec un nombre dynamique



# Opérateurs et Scheduler

---

Certains opérateurs utilisent par défaut une implémentation de *Scheduler*

Ils fournissent généralement la possibilité de spécifier une autre implémentation

Par exemple :

```
// Par défaut Schedulers.parallel()
```

```
Flux.interval(Duration.ofMillis(300))
```

```
// Positionnement de Schedulers.single()
```

```
Flux.interval(Duration.ofMillis(300),  
    Schedulers.newSingle("test"))
```



# Changer la thread d'exécution

*Reactor* offre 2 moyens de changer la thread d'exécution dans une chaîne réactive : ***publishOn*** et ***subscribeOn***.

La position de *publishOn* dans la chaîne a une importance pas celle de *subscribeOn*

- ***publishOn*** s'applique de la même façon que les autres opérateurs.  
Prend le signal amont et le rejoue vers l'aval en exécutant le callback sur le worker du *Scheduler* associé.
- ***subscribeOn*** s'applique à l'abonnement lorsque la chaîne inverse est construite. Quelque soit sa position, il affecte le contexte de thread de la source de l'émission.  
Seul le premier *subscribeOn* dans la chaîne est pris en compte.



# Clean up

---

Les méthodes factory peuvent créer des threads classiques ou des daemons (JVM existe si il n'y a plus que des daemons).

Il faut quelquefois nettoyer les threads qui ne servent plus.

Cela peut être implémenté par la méthode ***doFinally*** qui traite les événements *onError*, *onComplete* et *cancel*

```
Flux.range(1, 10000)
    .publishOn(s)
    .doFinally(sig -> {
        s.dispose();
        logger.info("Shut down all Scheduler worker threads");
    })
```



# Reactor

---

Cœur de l'API

Threads et Scheduler

**Traitement des Erreurs**

Test, Debug

Propagation de contexte





# Introduction

---

Les erreurs sont des événements terminaux stoppant la séquence, arrêtant la chaîne d'opérateurs et provoquant l'appel de la méthode ***onError()*** de l'abonné.

Si *onError()* n'est pas définie, l'exception ***UnsupportedOperationException*** est lancée.



# Opérateurs de traitement d'erreurs

---

*Reactor* permet de définir des opérateurs de traitement d'erreur en milieu de chaîne.

- Cela n'évite pas l'arrêt de la séquence originale
- Mais cela permet de démarrer une nouvelle séquence : la séquence de fallback



# Cas d'usage

---

Ces opérateurs permettent de retrouver les alternatives que l'on a avec les blocs *try*, *catch*, *finally* des *Exceptions*

- 1) Attraper et retourner une valeur statique par défaut
- 2) Attraper et exécuter une méthode de fallback.
- 3) Attraper et calculer une valeur de fallback.
- 4) Attraper, encapsuler dans une *BusinessException*, et relancer.
- 5) Attraper, log un message d'erreur et relancer.
- 6) Utiliser le bloc *finally* pour libérer les ressources ou "try-with-resource" de Java 7 .

Les opérateurs sont ***onErrorReturn***, ***onErrorResume***, ***doOnError***, ...



# Exemples

---

**// Valeur statique de fallback**

```
Flux.just(10)
    .map(this::doSomethingDangerous)
    .onErrorReturn("RECOVERED");
```

**// Méthode de fallback**

```
Flux.just("key1", "key2")
    .flatMap(k -> callExternalService(k))
    .onErrorResume(k -> getFromCache(k)) );
// (getFromCache retourne un Flux)
```



# Examples (2)

---

```
// Catch and rethrow
Flux.just("timeout1")
    .flatMap(k -> callExternalService(k))
    .onErrorResume(original -> Flux.error(
        new BusinessException("oops, SLA exceeded", original));

// Log (operator side-effect)
LongAdder failureStat = new LongAdder();
Flux<String> flux =
    Flux.just("unknown")
        .flatMap(k -> callExternalService(k))
        .doOnError(e -> {
            failureStat.increment();
            log("uh oh, falling back, service failed for key " + k);
        })
        .onErrorResume(e -> getFromCache(k))
    );
```



# Exemple (3)

---

```
// Finally
AtomicBoolean isDisposed = new AtomicBoolean();
Disposable disposableInstance = new Disposable() {
    @Override
    public void dispose() { isDisposed.set(true); }

    @Override
    public String toString() { return "DISPOSABLE"; }
};

Flux<String> flux =
Flux.using(
    () -> disposableInstance, // Génère la ressource
    disposable -> Flux.just(disposable.toString()),
    Disposable::dispose // <=> bloc finally
);
```



# *retry()*

---

***retry*** permet de se réabonner au *Publisher* pour lequel l'erreur s'est produite.

On obtient alors une nouvelle séquence, la séquence originale étant terminée

```
Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .retry(1)
    .subscribe(System.out::println, System.err::println);
```

Quelle sortie ??



# Exceptions dans les fonctions des opérateurs

---

Les exceptions lancées dans les fonctions des opérateurs peuvent être de 3 types :

- ***Fatal*** (*OutOfMemoryError*) : *Reactor* estime que l'on ne peut rien faire et lance l'exception
- ***Unchecked*** : Propagation vers la méthode *onError()*
- ***Checked*** : Construction *try/catch* dans la fonction, avec possibilité d'utiliser une classe utilitaire *Exceptions*





# Exemple

---

**// Unchecked**

```
Flux.just("foo")
    .map(s -> { throw new IllegalArgumentException(s); })
    .subscribe(v -> System.out.println("GOT VALUE"),
               e -> System.out.println("ERROR: " + e));
```

**// Checked**

```
Flux<String> converted = Flux
    .range(1, 10)
    .map(i -> {
        try { return convert(i); }
        catch (IOException e) { throw Exceptions.propagate(e); }
    });
```



# Reactor

---

Cœur de l'API  
Threads et Scheduler  
Traitement des Erreurs  
**Test, Debug**  
Propagation de contexte



# Test

---

*Reactor* propose ***reactor-test***, un module pour les tests qui permet principalement 2 choses :

- Tester qu'une séquence suit un scénario donné avec ***StepVerifier***.
- Produire des données afin de tester le comportement d'opérateurs : ***TestPublisher***

```
// Gradle
dependencies {
    testcompile 'io.projectreactor:reactor-test'
}
```



# *StepVerifier*

---

```
public <T> Flux<T> appendBoomError(Flux<T> source) {  
    return source.concatWith(Mono.error(new IllegalArgumentException("boom")));  
}
```

-----

@Test

```
public void testAppendBoomError() {  
    Flux<String> source = Flux.just("foo", "bar");
```

```
    StepVerifier.create(  
        appendBoomError(source))  
        .expectNext("foo")  
        .expectNext("bar")  
        .expectErrorMessage("boom")  
        .verify();
```

```
}
```



# *TestPublisher*

---

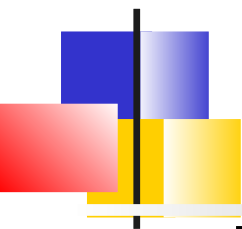
```
TestPublisher<String> publisher = TestPublisher.create();
AtomicLong count = new AtomicLong();

Subscriber<String> subscriber = new CoreSubscriber<String>() {

    public void onError(Throwable t) { count.incrementAndGet(); }
    public void onComplete() { count.incrementAndGet(); }
};

publisher.subscribe(subscriber);
publisher.complete()
    .emit("A", "B", "C")
    .error(new IllegalStateException("boom"));

assertThat(count.get()).isEqualTo(1);
```



# Debug

---

Dans le modèle impératif, le debug consiste principalement à lire la stack-trace

Cela devient un peu plus compliqué dans le modèle réactif, la stack-trace ne comporte souvent qu'une succession d'appels à *subscribe* et *request*

Reactor permet de fournir des stack-traces plus lisibles en activant le mode **debug** (impact sur les performances)

- Le mode debug peut être appliqué globalement ou à des points précis du code



# Activation globale

---

L'activation globale s'effectue par :

**`Hooks.onOperatorDebug()` ;**

Cet appel permet d'instrumenter les appels aux opérateurs de *Flux* et *Mono*.

Le mode débogage peut être activé/désactivé à tout moment, mais cela n'affecte pas les Flux/Mono déjà instanciés.

Pour désactiver :

`Hooks.resetOnOperatorDebug()`



# StackTrace

---

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
at reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:120)
at reactor.core.publisher.FluxOnAssembly$OnAssemblySubscriber.onNext(FluxOnAssembly.java:314)
...
...
at reactor.core.publisher.Mono.subscribeWith(Mono.java:2668)
at reactor.core.publisher.Mono.subscribe(Mono.java:2629)
at reactor.core.publisher.Mono.subscribe(Mono.java:2604)
at reactor.core.publisher.Mono.subscribe(Mono.java:2582)
at reactor.guide.GuideTests.debuggingActivated(GuideTests.java:727)
Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Assembly trace from producer [reactor.core.publisher.MonoSingle] :
reactor.core.publisher.Flux.single(Flux.java:5335)
reactor.guide.GuideTests.scatterAndGather(GuideTests.java:689)
reactor.guide.GuideTests.populateDebug(GuideTests.java:702)
org.junit.rules.TestWatcher$1.evaluate(TestWatcher.java:55)
org.junit.rules.RunRules.evaluate(RunRules.java:20)
Error has been observed by the following operator(s):
|_ Flux.single(TestWatcher.java:55)
```





# Reactor

---

Cœur de l'API  
Threads et Scheduler  
Traitement des Erreurs  
Test, Debug  
**Propagation de contexte**



# Introduction

---

La propagation de contexte permet de conserver un état sur des pipelines d'exécution asynchrones.

Cela a le même objectif que l'utilisation de ***ThreadLocal*** dans un modèle impératif.

Les cas d'usages :

- Conserver l'identification du client entre les différentes étapes d'exécution
- Conserver un id de transaction
- Etc.

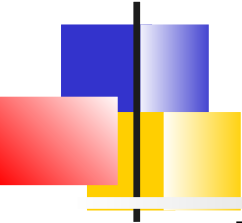


# Interface *Context*

---

Depuis la version 3.1.0, Reactor propose l'interface ***Context*** (~Map) permettant de stocker des paires clés-valeurs. Il peut exposer une version en read-only : ***ContextView***

- Un *Context* est immuable, les méthodes d'écritures comme *put()* et *putAll()* produisent une nouvelle instance.
- Il est possible de tester la présence d'une clé via *hasKey()*
- *getOrDefault()* ou *getOrEmpty()* permettent de récupérer une valeur
- *delete()* permet de supprimer une clé



# Association à un Flux et écriture d'un Contexte

---

Un *Context* est lié à une séquence spécifique afin d'être accessible par les opérateurs en amont d'une chaîne.

Le remplissage du contexte ne peut être fait qu'au moment de l'abonnement via l'opérateur ***contextWrite(ContextView)***

- L'opérateur fusionne le *ContextView* fourni et le *Context* en aval (appel à *putAll()*).



# Lecture du contexte

---

Les opérateurs de lecture permettent d'obtenir des données du *Context* dans une chaîne d'opérateurs en exposant son *ContextView* :

- A partir d'une source, on peut utiliser la méthode ***deferContextual(Function<ContextView, ? extends Mono<? extends T>> contextualMonoFactory)***
- A partir du milieu d'une chaîne d'opérateurs, on peut utiliser ***transformDeferredContextual(BiFunction)***

```
Mono.just("message")
...
.transformDeferredContextual((originalMono, context) ->
originalMono.doOnNext(e -> {
    log.info("correlation-id: " + context.get(CORRELATION_ID));
}))
```



# Exemples

---

```
// Le contexte est immuable et son contenu ne peut être vu  
// que par les opérateurs au-dessus de lui.
```

```
String key = "message";  
Mono<String> r = Mono.just("Hello")  
    .flatMap(s -> Mono.deferContextual(ctx ->  
        Mono.just(s + " " + ctx.get(key))))  
    .contextWrite(ctx -> ctx.put(key, "World"));
```

```
StepVerifier.create(r)  
    .expectNext("Hello World")  
    .verifyComplete();
```

```
//  
String key = "message";  
Mono<String> r = Mono.just("Hello")  
    .contextWrite(ctx -> ctx.put(key, "World"))  
    .flatMap( s -> Mono.deferContextual(ctx ->  
        Mono.just(s + " " + ctx.getOrDefault(key, "Stranger"))));
```

```
StepVerifier.create(r)  
    .expectNext("Hello Stranger")  
    .verifyComplete();
```



# Pile réactive

---

***Spring Data***

*Spring Webflux*

*WebClient*

*Server-side events*

*Reactive Web Sockets*

*Sécurité*



# Introduction

---

La programmation réactive s'invite également dans *SpringData*

Attention, cela ne concerne pas JPA qui reste une APIs bloquante

Sont supportés :

- MongoDB
- Cassandra
- Redis
- JDBC (relativement récemment)





# Accès réactifs aux données persistante

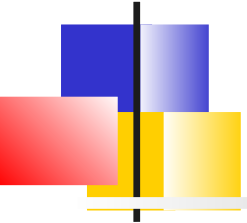
---

Les appels sont asynchrones, non bloquants, pilotés par les événements

Les données sont traitées comme des flux

Cela nécessite :

- Spring Reactor
- Spring Framework 5
- Spring Data 2.0
- Un pilote réactif (Implémentations NoSQL, JDBC)
- Éventuellement Spring Boot (2.0)



# Mélange bloquant non-bloquant

---

S'il faut mélanger du code bloquant et non bloquant, il ne faut pas bloquer la thread principale exécutant la boucle d'événements.

On peut alors utiliser les *Scheduler* de Spring Reactor.



# Apports

---

La fonctionnalité réactive reste proche des concepts *SpringData* :

- API de gabarits réactifs (Reactive Templates)
- Repository réactifs
- Les objets retournées sont encapsulés dans des Flux ou Mono



# *Reactive Template*

---

L'API des classes *Template* devient :

```
<T> Mono<T> insert(T objectToSave)
<T> Mono<T> insert(Mono<T> object)
<T> Flux<T> insertAll(Collection<? extends T>
    objectsToSave)
<T> Flux<T> find(Query query, Class<T> type
...

```

Exemple :

```
Flux<Person> insertAll = template
.insertAll(Flux.just(new Person("Walter", "White", 50), //
new Person("Skyler", "White", 45), //
new Person("Saul", "Goodman", 42), //
new Person("Jesse", "Pinkman", 27)).collectList());

```



# Reactive Repository

---

L'interface ***ReactiveCrudRepository<T,ID>*** permet de profiter d'implémentations de fonction CRUD réactives.

Par exemple :

```
Mono<Long> count()  
Mono<Void> delete(T entity)  
Flux<T> findAll()  
Mono<S> save(S entity)  
..
```



# Requêtes

---

Les requêtes peuvent être déduites du nom des fonctions ou de l'annotation `@Query`:

```
public interface ReactivePersonRepository extends
    ReactiveCrudRepository<Person, String> {

    Flux<Person> findByLastname(String lastname);

    @Query("{ 'firstname': ?0, 'lastname': ?1}")
    Mono<Person> findByFirstnameAndLastname(String firstname, String
    lastname);

    // Paramètre avec type réactif pour une exécution différée
    Flux<Person> findByLastname(Mono<String> lastname);

    Mono<Person> findByFirstnameAndLastname(Mono<String> firstname, String
    lastname);
}
```



# Exemple dépendance *MongoDB* avec *SpringBoot*

---

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.BUILD-SNAPSHOT</version>
</parent>
<!-- Ramène en particulier : spring-data-mongodb et reactor-core -->
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
  </dependency>
</dependencies>
```

=> Utilisation des classes *ReactiveMongoRepository*  
et *ReactiveMongoTemplate*



# Spring Data R2DBC

---

## ***Spring Data R2DBC (Reactive Relational Database Connectivity)***

permet une approche fonctionnelle pour interagir avec la base en mode réactif

Les principales BD offrent des drivers  
(*h2, MariaDB, Mysql, MSQl, Postgres, Oracle*)





# Support

---

R2DBC offre les fonctionnalités suivantes :

- *R2dbcEntityTemplate* en tant que classe centrale pour les opérations liées à l'entité avec un mapping d'objets.
- Mapping évolué intégré au service de conversion de Spring.
- Métadonnées de mapping basées sur des annotations extensibles.
- Implémentation automatique des interfaces *Repository* avec déduction requête via le nom de méthode ou annotation `@Query`



# API

---

Les classes principales d'un projet R2DBC sont :

- Des classes modèles annotées avec ***org.springframework.data.annotation.Id***;
- Des interfaces filles de ***ReactiveCrudRepository*** ou des ***R2dbcEntityTemplate***
- Des ***ConnectionFactory*** ou ***ConnectionFactoryInitializer*** qui permettent d'accéder à la base



# Annotations de mapping

---

**@Id**: La clé primaire (généralement générée automatiquement)

**@Table**: Table primaire de la classe entité

**@Transient**: Marque un champ non-persistant

**@PersistenceConstructor**: Marque le constructeur a utilisé par R2DBC pour convertir les lignes en objets

**@Value**: (Spring Framework). Permet l'utilisation de Spel

**@Column**: Mapper un attribut vers une colonne.



# Example

---

```
PostgresqlConnectionFactory connectionFactory = new
    PostgresqlConnectionFactory(PostgresqlConnectionFactory.builder()
        .host(...)
        .database(...)
        .username(...)
        .password(...).build());

R2dbcEntityTemplate template = new R2dbcEntityTemplate(connectionFactory);

Mono<Integer> update = template.update(Person.class)
    .inTable("person_table")
    .matching(Query.query(where("firstname").is("John")))
    .apply(update("age", 42));

Flux<Person> all = template.select(Person.class)
    .matching(Query.query(where("firstname").is("John")
        .and("lastname").in("Doe", "White")))
    .sort(by(desc("id"))))
    .all();
```



# Exemple Repository

```
interface PersonRepository extends ReactiveCrudRepository<Person, String> {  
  
    Flux<Person> findByFirstname(String firstname);  
  
    @Modifying  
    @Query("UPDATE person SET firstname = :firstname where lastname  
        = :lastname")  
    Mono<Integer> setFixedFirstnameFor(String firstname, String lastname);  
  
    @Query("SELECT * FROM person WHERE lastname = :#[0]")  
    Flux<Person> findByQueryWithExpression(String lastname);  
  
}
```



# Pile réactive

---

*Spring Data*

***Spring Webflux***

*WebClient*

*Server-side events*

*Reactive Web Sockets*

*Sécurité*



# Motivation

---

2 principales motivations pour Spring Webflux :

- Le besoin d'un stack non-bloquante permettant de gérer la concurrence avec peu de threads et de scaler avec moins de ressources CPU/mémoire
- La programmation fonctionnelle



# Introduction

---

Starter ***spring-reactive-web***

Spring Webflux offre en plus 2 modèles de programmation :

- **Contrôleurs annotés** : Idem à Spring MVC avec les mêmes annotations.  
Les méthodes des contrôleurs peuvent retourner des types réactifs, des arguments réactifs sont associés à *@RequestBody*.
- **Endpoints fonctionnels** : Programmation fonctionnelle basée sur les lambdas.  
Idéal pour de petites librairies permettant de router et traiter des requêtes.  
Dans ce cas, l'application est en charge du traitement de la requête du début à la fin.

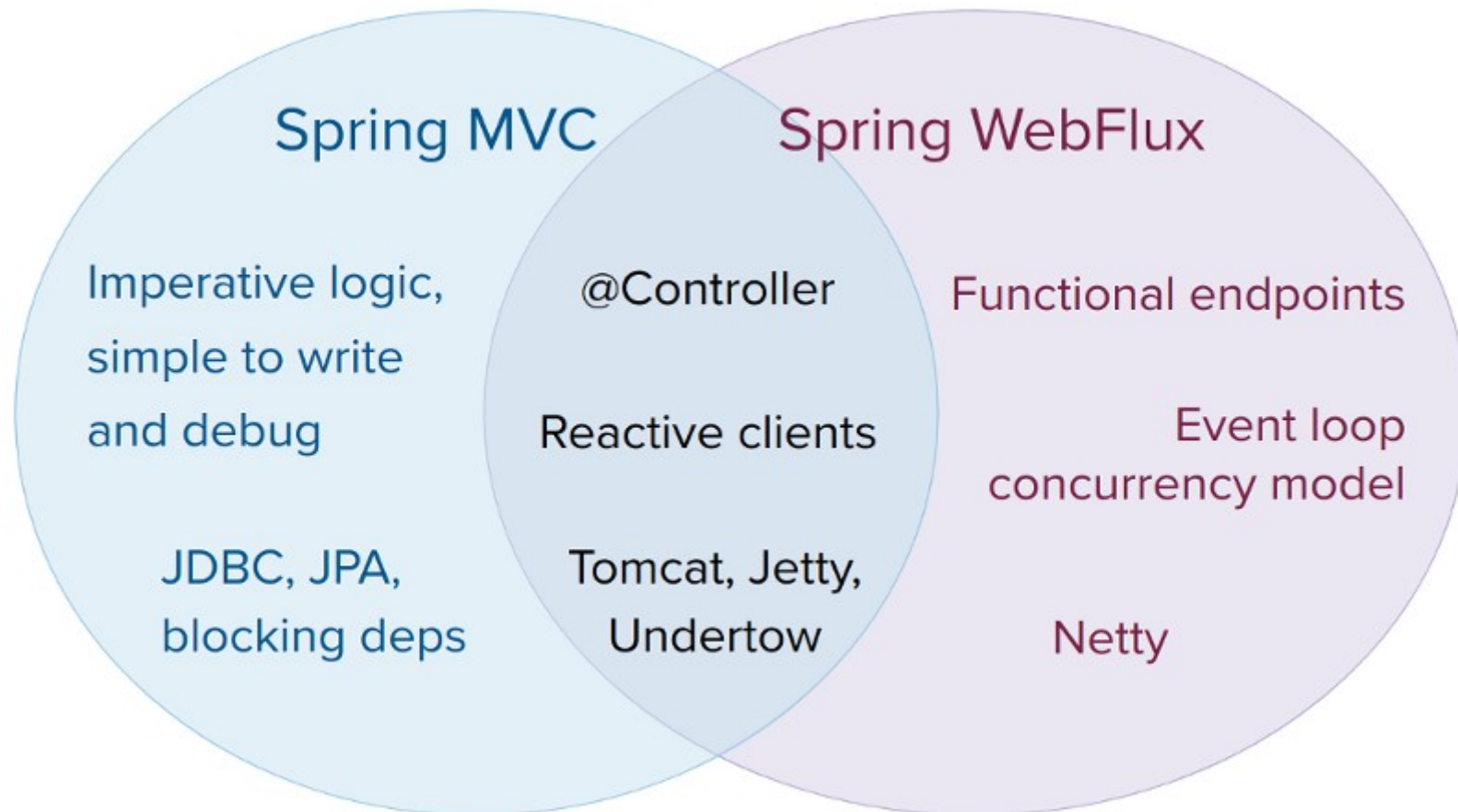
Dans la version récente de SpringBoot on peut mixer du code réactif et du code impératif.





# MVC et WebFlux

---





# Serveurs

---

Spring WebFlux est supporté sur

- Tomcat, Jetty, et les conteneurs de Servlet 3.1+,
- Ainsi que les environnements non-Servlet comme *Netty* ou *Undertow*

Le même modèle de programmation est supporté sur tous ces serveurs

Avec *SpringBoot*, la configuration par défaut démarre un serveur embarqué Netty



# Performance et Scaling

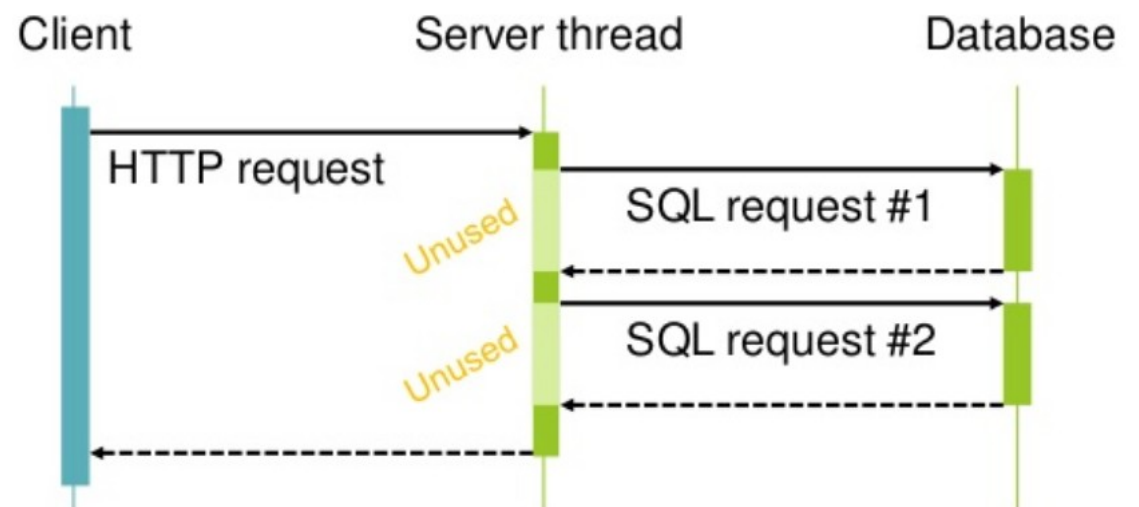
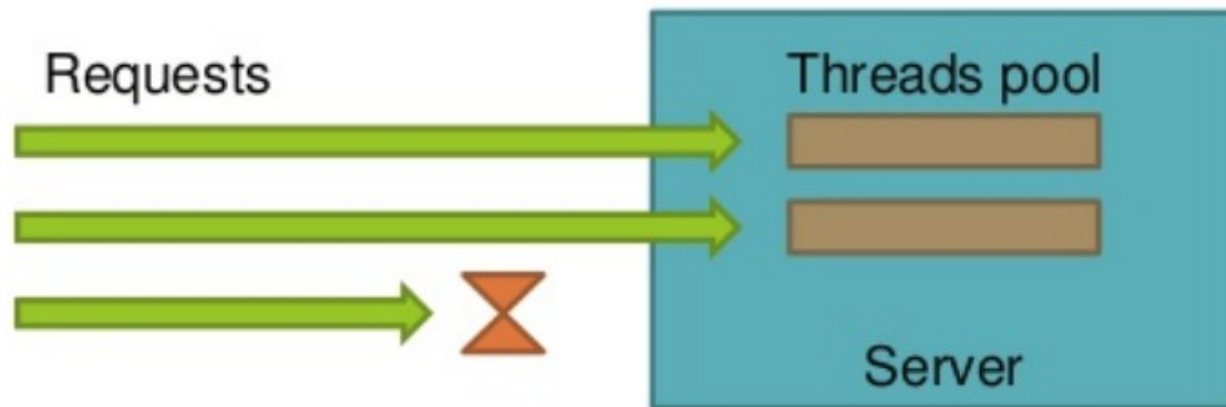
---

Le modèle réactif et non bloquant n'apporte pas spécialement de gain en terme de temps de réponse. (il y a plus de chose à faire et cela peut même augmenter le temps de traitement)

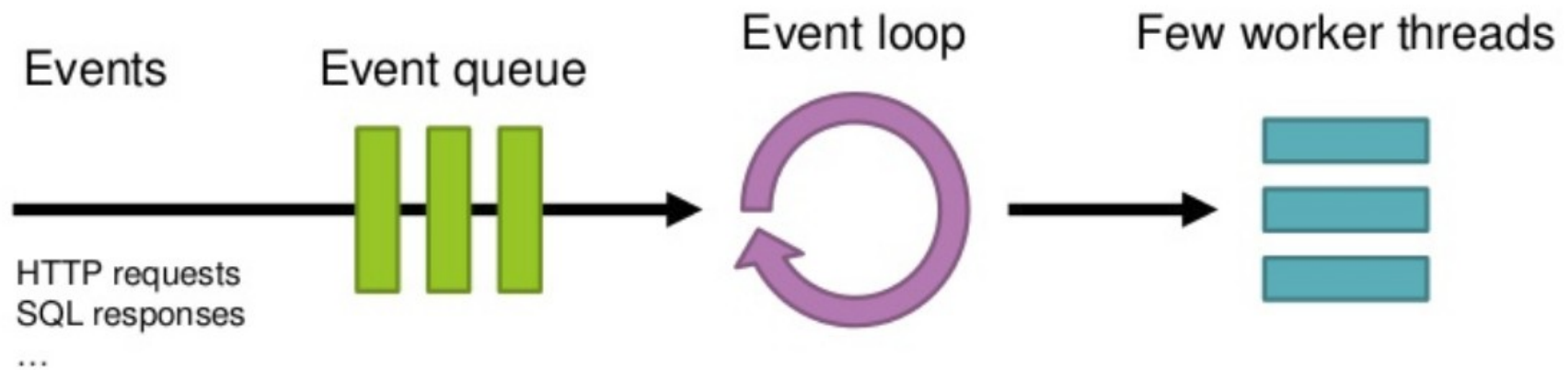
Le bénéfice attendu est la possibilité de **scaler** avec un petit nombre de threads fixes et moins de mémoire. Cela rend les applications plus résistantes à la charge.

Pour pouvoir voir ces bénéfices, il est nécessaire d'introduire de la latence, par exemple en introduisant des IO réseaux lents ou non prédictibles.

# Modèle bloquant



# Modèle non bloquant





# Modèle de threads

---

Pour un serveur *Spring WebFlux* entièrement réactif, on peut s'attendre à 1 thread pour le serveur et autant de threads que de CPU pour le traitement des requêtes.

Si on doit accéder à des données de manière bloquante (JPA ou JDBC par exemple), il est nécessaire d'utiliser des threads de type worker via *Schedulers*

Pour configurer le modèle de threads du serveur, il faut utiliser leur API de configuration spécifique ou voir si *Spring Boot* propose un support.



# Généralités API

---

En général l'API WebFlux

- accepte en entrée un *Publisher*,
- l'adapte en interne aux types *Reactor*,
- l'utilise et retourne soit un *Flux*, soit un *Mono*.

En terme d'intégration :

- On peut fournir n'importe quel *Publisher* comme entrée
- Il faut adapter la sortie si l'on veut quelle soit compatible avec une autre librairie que *Reactor*



# API Web

---

Le package *org.springframework.web.server* fournit une API pour traiter les requêtes HTTP.

La requête est alors traitée par :

- Une chaîne de **WebExceptionHandler**
- Une chaîne de **WebFilter**
- Et un **WebHandler** (*DispatcherHandler*)

Les chaînes de filtres et de gestionnaires d'exception sont configurées par **WebFluxConfigurer**

*DispatcherHandler* délègue le traitement de la requête via les annotations *@Controller* ou les endpoints fonctionnels

D'autre part, SpringWebFlux ajoute un id de requête dans les traces





# Contrôleurs annotés

---

Les annotations **@Controller** et **@RestController** de Spring MVC sont donc supportés par *WebFlux*.

Ils utilisent les même annotations que celles de Spring MVC :

- *@GetMapping, @PostMapping, @PutMapping, ...*
- *@PathVariable, @RequestParam, @RequestBody, @RequestHeader, @CookieValue*
- *@ResponseStatus*



# Arguments des méthodes

---

Certains types d'argument sont automatiquement renseignés par le `DispatcherHandler` :

- *ServerWebExchange* : Fournit un accès à la requête et à la réponse HTTP + propriétés et fonctionnalités supplémentaires liées au traitement côté serveur, telles que les attributs de requête.
- *ServerHttpRequest* et *ServerHttpResponse*
- *WebSession*
- *java.security.Principal*
- *java.util.Locale*
- ...



# Valeurs de retour

---

Les valeurs de retour des méthodes annotées peuvent être :

- ***Flux<T>*** ou ***Mono<T>***. L'objet est sérialisé par un `HttpMessageWriter`
- ***HttpEntity<B>***, ***ResponseEntity<B>*** : La réponse complète
- Des ***View***, ***String***, ***Map*** permettant de rediriger vers une vue (Legacy MVC)
- ***Flux<ServerSentEvent>*** : Emission d'événements serveur (text/event-stream)



# Example

---

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }

    @PostMapping("/person")
    Mono<Void> create(@RequestBody Publisher<Person> personStream) {
        return this.repository.save(personStream).then();
    }

    @GetMapping("/person")
    Flux<Person> list() {
        return this.repository.findAll();
    }

    @GetMapping("/person/{id}")
    Mono<Person> findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```



# Filtre

Les filtres avec *SpringWebFlux* manipulent *ServerWebExchange* et la chaîne de filtre.

## Exemple :

```
@Component
public class SecurityWebFilter implements WebFilter{
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain)
    {
        if(!exchange.getRequest().getQueryParams().containsKey("user")){
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
        }
        return chain.filter(exchange);
    }
}
```



# Exceptions

---

Les classes *@Controller* et *@ControllerAdvice* peuvent avoir des méthodes annotés par ***@ExceptionHandler***

Ces méthodes sont responsable de générer la réponse en cas de déclenchement de l'exception.

```
@ExceptionHandler
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
```

SpringFlux fournit une implémentation : *WebFluxResponseStatusExceptionHandler* qui traite les exceptions de type *ResponseStatusException*.



# Configuration

---

Dans la configuration Java, on peut implémenter un ***WebFluxConfigurer*** afin de surcharger la configuration par défaut.

Différentes méthodes peuvent alors être surchargées.

Citons :

*addCorsMappings(CorsRegistry registry)* qui permet de configurer globalement le CORS.



# Exemple

---

**@Configuration**

**@EnableWebFlux**

```
public class WebConfig implements WebFluxConfigurer {  
    @Bean  
    public RouterFunction<?> routerFunctionA() { // ... }  
    @Bean  
    public RouterFunction<?> routerFunctionB() { // ... }  
    @Override  
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {  
        // configure message conversion...  
    }  
    @Override  
    public void addCorsMappings(CorsRegistry registry) {  
        // configure CORS...  
    }  
    @Override  
    public void configureViewResolvers(ViewResolverRegistry registry) {  
        // configure view resolution for HTML rendering...  
    }  
}
```





# *Endpoints* fonctionnels

---

Dans ce modèle de programmation fonctionnelle, les fonctions (lambda-expression) sont utilisées pour mapper et traiter les requêtes.

Les interfaces représentant l'interaction HTTP (requête/réponse) sont immuables

=> Thread-safe nécessaire pour le modèle réactif



# *ServerRequest et ServerResponse*

---

***ServerRequest*** et ***ServerResponse*** sont donc des interfaces qui offrent des accès via des lambda-expression aux messages HTTP.

- ***ServerRequest*** expose le corps de la requête comme Flux ou Mono.  
Flux<Person> people = request.bodyToFlux(Person.class);  
Elle donne accès aux éléments HTTP (Méthode, URI, ..) à travers une interface séparée *ServerRequest.Headers*.
- ***ServerResponse*** accepte tout *Publisher* comme corps.  
Elle est créée via un builder permettant de positionner le statut, les entêtes et le corps de réponse  
ServerResponse.ok()  
.contentType(MediaType.APPLICATION\_JSON).body(person);



# Traitement des requêtes via *HandlerFunction*

---

Les requêtes HTTP sont traitées par une ***HandlerFunction*** : une fonction qui prend en entrée un *ServerRequest* et fournit un *Mono<ServerResponse>*

Exemple :

```
HandlerFunction<ServerResponse> helloWorld =  
    request ->  
    ServerResponse.ok().body(fromObject("Hello World"));
```

Généralement, les fonctions similaires sont regroupées dans une classe *contrôleur*.



# Example

---

```
public class PersonHandler {  
    private final PersonRepository repository;  
  
    public PersonHandler(PersonRepository repository) { this.repository = repository;}  
  
    public Mono<ServerResponse> listPeople(ServerRequest request) {  
        Flux<Person> people = repository.allPeople();  
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person.class);  
    }  
  
    public Mono<ServerResponse> createPerson(ServerRequest request) {  
        Mono<Person> person = request.bodyToMono(Person.class);  
        return ServerResponse.ok().build(repository.savePerson(person));  
    }  
  
    public Mono<ServerResponse> getPerson(ServerRequest request) {  
        int personId = Integer.valueOf(request.pathVariable("id"));  
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();  
        Mono<Person> personMono = this.repository.getPerson(personId);  
        return personMono  
            .then(ServerResponse.ok().contentType(APPLICATION_JSON).body(personMono, Person.class))  
            .switchIfEmpty(notFound);  
    }  
}
```



# Mapping via *RouterFunction*

---

Les requêtes sont routées vers les *HandlerFunction* avec une ***RouterFunction*** :

Prend en entrée un *ServerRequest* et retourne un *Mono<HandlerFunction>*

- Les fonctions ne sont en général pas écrites directement. On utilise :  
***RouterFunctions.route(RequestPredicate, HandlerFunction)***  
permettant de spécifier les règles de matching

Exemple :

```
RouterFunction<ServerResponse> helloWorldRoute =  
RouterFunctions.route(RequestPredicates.path("/hello-world"),  
request -> Response.ok().body(fromObject("Hello World")));
```



# Combinaison

---

2 fonctions de routage peuvent être composées en une nouvelle fonction via les méthodes

```
RouterFunction.and(RouterFunction)
```

```
RouterFunction.andRoute(RequestPredicate,  
    HandlerFunction)
```

Si la première règle ne matche pas, la seconde est évaluée ... et ainsi de suite



# Example

---

```
PersonRepository repository = ...
```

```
PersonHandler handler = new PersonHandler(repository);
```

```
RouterFunction<ServerResponse> personRoute = RouterFunctions.
```

```
  route(RequestPredicates.GET("/person/{id}"))
```

```
    .and(accept(APPLICATION_JSON)), handler::getPerson)
```

```
  .andRoute(RequestPredicates.GET("/person"))
```

```
    .and(accept(APPLICATION_JSON)), handler::listPeople)
```

```
  .andRoute(RequestPredicates.POST("/person"))
```

```
    .and(contentType(APPLICATION_JSON)), handler::createPerson);
```



# Filtres via *HandlerFilterFunction*

---

Les routes contrôlées par un fonction de routage peuvent être filtrées :

```
RouterFunction.filter(HandlerFilterFunction)
```

***HandlerFilterFunction*** est une fonction prenant une *ServerRequest* et une *HandlerFunction* et retourne une *ServerResponse*.

Le paramètre *HandlerFunction* représente le prochain élément de la chaîne : la fonction de traitement ou la fonction de filtre.





# Exemple :

## *Basic Security Filter*

```
import static org.springframework.http.HttpStatus.UNAUTHORIZED;

SecurityManager securityManager = ...
RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =
    route.filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    });
```



# Pile réactive

---

*Spring Data*  
*Spring Webflux*  
***WebClient***  
*Server-side events*  
*Reactive Web Sockets*  
*Sécurité*



# Introduction

---

*WebFlux* inclut ***WebClient*** facilitant les interactions REST<sup>1</sup>

- Expose les I/O réseau via *ClientHttpRequest* et *ClientHttpResponse*.
  - Les corps de la requête et de la réponse sont des *Flux<DataBuffer>* plutôt que des *InputStream* et *OutputStream*.
- Même mécanismes de sérialisation (JSON, XML) permettant de travailler avec des objets typés.
- En interne, *WebClient* délègue à une librairie client HTTP (Par défaut, *Reactor Netty*)

1. *Alternative non-bloquante à RestTemplate.*



# Création

---

La façon la + simple de créer un *WebClient* est d'utiliser les méthodes statiques :

```
WebClient.create()
```

```
WebClient.create(String baseUrl)
```

On obtient alors un *HttpClient* de *Reactor Netty* avec les configurations par défaut

Il existe également un *WebClient.Builder* qui permet de préciser toutes les options de configuration

Une fois construit, une instance de *WebClient* est immuable.



# Réponse

---

La méthode ***retrieve()*** permet de récupérer une réponse et de la décoder :

```
WebClient client = WebClient.create("http://example.org");

Mono<Person> result = client.get()
    .uri("/persons/{id}", id)
    .accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .bodyToMono(Person.class);
```



# Exceptions

---

Les réponses avec des statuts 4xx ou 5xx provoquent une ***WebClientResponseException***.

Il est possible d'utiliser la méthode ***onStatus*** pour personnaliser le traitement de l'exception:

```
Mono<Person> result = client.get()
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .onStatus(HttpStatus::is4xxServerError, response -> ...)
    .onStatus(HttpStatus::is5xxServerError, response -> ...)
    .bodyToMono(Person.class);
```



# Contrôle de la réponse

---

La méthode ***exchange()*** permet un meilleur contrôle de la réponse en permettant d'avoir un accès à *ClientResponse*

```
Mono<Person> result = client.get()  
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)  
    .exchange()  
    .flatMap(response -> response.bodyToMono(Person.class));
```



# Corps de requête

---

Le corps de la requête peut être encodé à partir d'un *Mono*, d'un *Flux* ou d'une type simple:

```
Mono<Person> personMono = ... ;
```

```
Mono<Void> result = client.post()  
    .uri("/persons/{id}", id)  
    .contentType(MediaType.APPLICATION_JSON)  
    .body(personMono, Person.class)  
    .retrieve()  
    .bodyToMono(Void.class);
```





# Corps de la requête (2)

---

```
Flux<Person> personFlux = ... ;
```

```
Mono<Void> result = client.post()  
    .uri("/persons/{id}", id)  
    .contentType(MediaType.APPLICATION_STREAM_JSON)  
    .body(personFlux, Person.class)  
    .retrieve()  
    .bodyToMono(Void.class);
```

```
Person person = ... ;
```

```
Mono<Void> result = client.post()  
    .uri("/persons/{id}", id)  
    .contentType(MediaType.APPLICATION_JSON)  
    .syncBody(person)  
    .retrieve()  
    .bodyToMono(Void.class);
```



# Formulaire

---

Pour poster des données de formulaire, il faut fournir une *MultiValueMap<String, String>* dans le corps.

Le content type est alors positionné automatiquement à : *"application/x-www-form-urlencoded"*

```
MultiValueMap<String, String> formData = ... ;
```

```
Mono<Void> result = client.post()  
    .uri("/path", id)  
    .syncBody(formData)  
    .retrieve()  
    .bodyToMono(Void.class);
```



# Filtre client

---

Le ***WebClient.Builder*** permet  
d'enregistrer un filtre qui intercepte les  
requêtes

## Exemple : Authentification Basique

```
WebClient client = WebClient.builder()  
    .filter(basicAuthentication("user", "password"))  
    .build();
```



# Pile réactive

---

*Spring Data*

*Spring Webflux*

*WebClient*

***Server-side Events***

*Reactive Web Sockets*

*Sécurité*



# Server-side events

---

SSE permet une communication asynchrone d'un serveur vers un client via HTTP.

- Le serveur peut envoyer un flux d'événements qui met à jour le client de manière asynchrone.
- Presque tous les navigateurs supportent le SSE
- Il est très simple de s'abonner au flux en Javascript



# Spring Webflux et SSE

---

Il est possible d'implémenter SSE dans une méthode contrôleur en renvoyant un *Flux* ou une entité *ServerSentEvent*

- Avec le Flux, il faut préciser le mime-type

`MediaType.TEXT_EVENT_STREAM_VALUE`

Spring offre également du support pour le WebClient



# Examples

---

```
@GetMapping(path = "/stream-flux", produces =  
    MediaType.TEXT_EVENT_STREAM_VALUE)  
public Flux<String> streamFlux() {  
    return Flux.interval(Duration.ofSeconds(1))  
        .map(sequence -> "Flux - " + LocalDateTime.now().toString());  
}
```

----

```
@GetMapping("/stream-sse")  
public Flux<ServerSentEvent<String>> streamEvents() {  
    return Flux.interval(Duration.ofSeconds(1))  
        .map(sequence -> ServerSentEvent.<String> builder()  
            .id(String.valueOf(sequence))  
            .event("periodic-event")  
            .data("SSE - " + LocalDateTime.now().toString())  
            .build());  
}
```



# Exemple *WebClient*

---

```
public void consumeServerSentEvent() {
    WebClient client = WebClient.create("http://localhost:8080/sse-server");
    ParameterizedTypeReference<ServerSentEvent<String>> type
    = new ParameterizedTypeReference<ServerSentEvent<String>>() {};

    Flux<ServerSentEvent<String>> eventStream = client.get()
        .uri("/stream-sse")
        .retrieve()
        .bodyToFlux(type);

    eventStream.subscribe(
        content -> logger.info("Time: {} - event: name[{}], id [{}], content[{}] ",
            LocalTime.now(), content.event(), content.id(), content.data()),
        error -> logger.error("Error receiving SSE: {}", error),
        () -> logger.info("Completed!!!"));
}
```





# Exemple Javascript

---

<script>

```
var source = new EventSource("/orders/status");  
source.onmessage = (event) => {  
    console.log("An event "+event);
```

```
    var json = JSON.parse(event.data);  
    console.log("JSON "+JSON.stringify(json));
```

```
};  
</script>
```



# Pile réactive

---

*Spring Data*  
*Spring Webflux*  
*WebClient*  
*Server-side events*  
***Reactive Web Sockets***  
*Sécurité*



# Introduction

---

Le protocole **WebSocket** (RFC 6455) définit un standard pour une communication full-duplex entre un client et un serveur.

C'est un protocole TCP différent de HTTP mais qui utilise les ports 80 et 443 pour passer les firewall.

Spring Framework fournit une API WebSocket permettant d'écrire du code client ou serveur.



# Cas d'usage

---

Une combinaison d'Ajax et de streaming HTTP ou du polling peuvent souvent avoir les mêmes effets

Les *WebSockets* sont utilisées lorsque le client et le serveur doivent échanger des événements à une haute fréquence avec peu de latence.

Attention, la configuration de beaucoup de proxy bloque les websockets !!



# URL unique

---

A la différence des architectures HTTP ou REST, les *WebSockets* n'utilisent qu'**une seule URL** pour la connexion initiale

Tous les messages applicatifs utilisent alors la même connexion TCP.

=> Cela conduit à une architecture asynchrone et piloté par les événements



# Mise en place

---

La mise en place consiste à :

- Définir côté serveur un ***WebSocketHandler***
- L'associer à une URL via un ***HandlerMapping*** et un ***WebSocketHandlerAdapter***
- Utiliser un ***WebSocketClient*** pour démarrer un session



# WebSocketHandler

**WebSocketHandler** définit la méthode *handle()* qui prend une *WebSocketSession* et retourne *Mono<Void>* lorsque le traitement de la session est terminée

La session est traitée via 2 streams de *WebSocketMessage* :  
1 pour le message d'entrée, 1 pour le message de sortie

La session propose donc 2 méthodes :

- ***Flux<WebSocketMessage> receive()*** : Accès au flux d'entrée se termine à la fermeture de connexion.
- ***Mono<Void> send(Publisher<WebSocketMessage>)*** : Prend un source pour les messages de sortie, écrit les messages et retourne *Mono<Void>* lorsque la source est tarie.



# Exemple

---

```
@Component
public class ReactiveWebSocketHandler implements WebSocketHandler {

    // private fields ...

    @Override
    public Mono<Void> handle(WebSocketSession webSocketSession) {
        return webSocketSession.send(intervalFlux
            .map(webSocketSession::textMessage))
            .and(webSocketSession.receive()
                .map(WebSocketMessage::getPayloadAsText)
                .log());
    }
}
```





# Mapping

---

@Autowired

```
private WebSocketHandler webSocketHandler;
```

@Bean

```
public HandlerMapping webSocketHandlerMapping() {  
    Map<String, WebSocketHandler> map = new HashMap<>();  
    map.put("/event-emitter", webSocketHandler);  
  
    SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();  
    handlerMapping.setOrder(1);  
    handlerMapping.setUrlMap(map);  
    return handlerMapping;  
}
```

@Bean

```
public WebSocketHandlerAdapter handlerAdapter() {  
    return new WebSocketHandlerAdapter();  
}
```



# Client

Spring WebFlux fournit une interface ***WebSocketClient*** avec des implémentations pour Reactor Netty, Tomcat, Jetty, Undertow, et Java standard (i.e. JSR-356).

Pour démarrer une session *WebSocket*, créer une instance du client et utiliser sa méthode ***execute*** :

```
WebSocketClient client = new ReactorNettyWebSocketClient();
```

```
URI url = new URI("ws://localhost:8080/path");  
client.execute(url, session ->  
    session.receive()  
        .doOnNext(System.out::println)  
        .then());
```



# Exemple avec envoi de message vers le serveur

---

```
WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute(
    URI.create("ws://localhost:8080/event-emitter"),
    session -> {
        // Receive messages and for each message, send a response
        Flux<Void> receiveAndRespond = session.receive()
            .map(webSocketMessage -> webSocketMessage.getPayloadAsText())
            .flatMap(message -> {
                System.out.println("Received: " + message);
                return session.send(Mono.just(session.textMessage("response
to: " + message))).then();
            })
            .log();
        return receiveAndRespond.then();
    })
    .block(Duration.ofSeconds(60));
```



# Pile réactive

---

*Spring Data*  
*Spring Webflux*  
*WebClient*  
*Server-side events*  
*Reactive Web Sockets*  
***Sécurité***



# Introduction

---

*Spring Security* apporte les nouveautés suivantes :

- Meilleur support de *OAuth 2.0*
- Support pour la programmation réactive
  - *@EnableWebFluxSecurity*
  - *@EnableReactiveMethodSecurity*
  - *ReactiveUserDetailsService*
  - Test de WebFlux
- Nouveaux encodages de mots de passe



# OAuth 2.0

---

**OAuth 2.0** permet aux utilisateurs de se connecter sur une application en se connectant avec un compte existant d'un fournisseur OAuth2.0

- Se logger avec son compte GitHub ou Google
- Protéger des micro-services dans une architecture micro-services

*OAuth2.0* est un protocole avec bcp de variantes d'implémentations



# SpringBoot / Authentification Google

---

*SpringBoot* permet de facilement mettre en place une authentification Google par exemple :

- Obtenir un *clientId* et un *clientSecret* chez Google
- Positionner l'URL de redirection permettant à Google de fournir le jeton d'autorisation
- Configurer *application.yml* en indiquant les créidentiels client
- Se connecter à l'application et consentir que celle-ci accède à l'adresse email google et les informations basiques de profil



# *WebFlux Security*

---

La sécurité pour *Webflux* est consistante avec celle de Spring MVC

- Elle est implémentée sous forme de **filtre** (WebFilter) que l'on peut configurer finement
- L'annotation **@EnableWebFluxSecurity** permet d'avoir une configuration par défaut.
- La configuration par défaut peut être personnalisée via la classe **ServerHttpSecurity**

Les Beans de personnalisation sont réactifs





# Configuration minimale

---

```
/* Configuration fournissant un authentification basique
 * via une page de login et de logout
 * Toutes les URLs sont protégées
 * Les entêtes HTTP relatifs à la sécurité sont positionnés (CSRF, ...)
 */
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {

    /*
     * MapReactiveUserDetailsService implémente ReactiveUserDetailsService
     */
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```



# Configuration personnalisée

---

```
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {

    @Bean
    public SecurityWebFilterChain
        springSecurityFilterChain(ServerHttpSecurity http) {
        http.csrf(csrfSpec -> csrfSpec.disable())
            .authorizeExchange( acl -> acl
                .pathMatchers(HttpMethod.POST, "/accounts").hasRole("ADMIN")
                .pathMatchers("/accounts").authenticated()
                .anyExchange().permitAll())
            .formLogin(Customizer.withDefaults())

        return http.build();
    }
}
```



# Sécurité au niveau des méthodes

---

En programmation réactive, la sécurité au niveau méthodes est possible en utilisant des **Reactor's context**

Elle peut être cumulée avec la sécurité sur les URLs

De la même façon, on obtient une configuration par défaut en utilisant l'annotation

***@EnableReactiveMethodSecurity***



# Exemple minimal

---

## **@EnableReactiveMethodSecurity**

```
public class SecurityConfig {
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
        UserDetails rob =
            userBuilder.username("rob").password("rob").roles("USER").build();
        UserDetails admin =
            userBuilder.username("admin").password("admin").roles("USER","ADMIN").build();
        return new MapReactiveUserDetailsService(rob, admin);
    }
}
```

## **@Component**

```
public class HelloWorldMessageService {

    @PreAuthorize("hasRole('ADMIN')")
    public Mono<String> findMessage() {
        return Mono.just("Hello World!");
    }
}
```



# Test de la sécurité

---

Spring Security 5 offre un support pour tester la sécurité pour tout type de combinaison :  
Impératif/Réactif et URL/méthodes

Annotations : *@WithMockUser*,  
*@WithAnonymousUser*, *@WithUserDetails*,  
*MockMvc*

Dans l'univers réactif : Utilisation de  
***StepVerifier*** provenant de *Reactor*  
permettant d'exprimer les événements  
attendus d'un *Publisher* lors d'un abonnement



# Spring Coeur et les tests

---

*Spring Coeur*  
Annotations Tests  
***WebTestClient***



# Support pour Webflux

---

Spring5 propose des mock implémentations de *ServerHttpRequest* , *ServerHttpResponse*, et *ServerWebExchange* à utiliser dans des applications WebFlux.

Le ***WebTestClient*** permet de tester une application WebFlux

- Sans serveur : Test unitaire
- Avec serveur : Test e2e



# Mise en place

---

La création d'un ***WebTestClient*** s'effectue de différentes façons.

En fonction du type de test que l'on veut effectuer on peut l'associer à :

- Un unique contrôleur
- Une *RouterFunction*
- Un contexte Spring
- Un serveur





# Contrôleur unique

---

```
client = WebTestClient.bindToController(new  
    TestController()).build();
```

- Charge la configuration de WebFlux
- Enregistre le contrôleur fournit .

L'application WebFlux peut être testée sans serveur HTTP en utilisant des requêtes et des réponses mockées.

D'autres méthodes sur le builder permettent de personnaliser la config.



# *RouterFunction*

---

```
RouterFunction<?> route = ...
```

```
    client =  
    WebClient.bindToRouterFunction(route).build();
```

L'application *WebFlux* peut également être testée sans serveur HTTP en utilisant des requêtes et des réponses mockées.



# *ApplicationContext*

---

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = WebConfig.class)
public class MyTests {
    @Autowired
    private ApplicationContext context;

    private WebTestClient client;

    @Before
    public void setUp() {
        client =
        WebTestClient.bindToApplicationContext(context).build();
    }
}
```

L'application *WebFlux* peut également être testée sans serveur HTTP en utilisant des requêtes et des réponses mockées.



# Serveur

---

```
client = WebTestClient.bindToServer().  
    baseUrl("http://localhost:8080").build();
```

=> Connexion à un serveur démarré



# Client Builder

---

On peut configurer les options du client comme la base URL, les entêtes par défaut, les filtres, ..

Ces options sont directement accessible lorsque l'on a associé le client à un serveur, pour les autres association, il faut utiliser

***configureClient()***

```
client = WebTestClient.bindToController(new  
    TestController())  
        .configureClient()  
        .baseUrl("/test")  
        .build();
```



# Ecriture des tests

---

*WebTestClient* fournit une API identique à *WebClient*

L'exécution d'une requête peut se faire par ***exchange()***

Ensuite, une chaîne de vérification peut être combinée.

- Tester le statut et les entêtes
- Extraire le corps de la réponse et y effectuer des assertions



# Exemple

---

```
client.get().uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON_UTF8)
    .exchange()
    .expectStatus().isOk()
    .expectHeader()
        .contentType(MediaType.APPLICATION_JSON_UTF8)
    .expectBodyList(Person.class) // Extraction
        .hasSize(3).contains(person); // Assertions
```



# Extraction du corps de requête

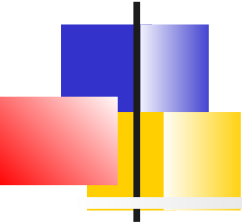
---

Typiquement, 3 possibilités pour extraire le corps de requête :

- ***expectBody(Class<T>)*** : Décoder en un simple objet.
- ***expectBodyList(Class<T>)*** : Décoder en une List<T>.
- ***expectBody()*** : Décoder en un tableau d'octets pour le contenu JSON ou un corps vide

Le résultat effectif peut être récupéré





# Exemple avec obtention du résultat

---

```
EntityExchangeResult<Person> result = client.get().uri("/persons/1")  
    .exchange()  
    .expectStatus().isOk()  
    .expectBody(Person.class)  
    .returnResult();
```



# Réponse sans contenu

---

**// Permet de garantir que les ressources sont libérées**

```
client.get().uri("/persons/123")  
    .exchange()  
    .expectStatus().isNotFound()  
    .expectBody(Void.class);
```

**// Assertion pour contenu vide**

```
client.post().uri("/persons")  
    .body(personMono, Person.class)  
    .exchange()  
    .expectStatus().isCreated()  
    .expectBody().isEmpty();
```



# Réponse JSON

---

Possibilité d'utiliser *JSONAssert* et *JSONPath*

```
client.get().uri("/persons")
    .exchange()
    .expectStatus().isOk()
    .expectBody()
    .jsonPath("$.name").isEqualTo("Jane")
    .jsonPath("$.name").isEqualTo("Jason");
```



# Réponses Stream

---

Le test des réponses sous forme de flux, s'effectue en 2 étapes :

- Récupérer un Flux via la méthode *returnResult*
- Utiliser StepVerifier de Reactor pour vérifier les événements du Flux.



# Exemple

```
FluxExchangeResult<MyEvent> result = client.get().uri("/events")
    .accept(TEXT_EVENT_STREAM)
    .exchange()
    .expectStatus().isOk()
    .returnResult(MyEvent.class);
```

```
Flux<Event> eventFux = result.getResponseBody();
```

```
StepVerifier.create(eventFlux)
    .expectNext(person)
    .expectNextCount(4)
    .consumeNextWith(p -> ...)
    .thenCancel()
    .verify();
```



# Merci!!!

---

❖ MERCI DE VOTRE ATTENTION



# Références

---

Spring Reference : Web Reactive Stack :

<https://docs.spring.io/spring/docs/5.1.0.RC3/spring-framework-reference/web-reactive.html#spring-webflux>

Spring Reactor :

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

Tutoriaux Baeldung

<https://www.baeldung.com/spring-5>

Présentations

<https://fr.slideshare.net/fbeaufume/programmation-reactive-avec-spring-5-et-reactor-81924228>

<https://fr.slideshare.net/Pivotal/reactive-data-access-with-spring-data>

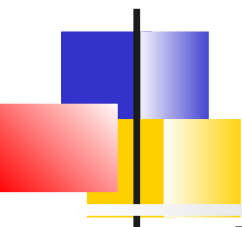


# Reactor

---

Cœur de l'API  
Threads et Scheduler  
Traitement des Erreurs  
**Sink**  
Test, Debug  
Propagation de contexte





# Introduction

---

Un Sink est une classe qui permet le déclenchement manuel sécurisé de signaux de manière autonome, créant une structure de type Publisher capable de gérer plusieurs abonnés.

Les implémentations fournies par Reactor garantissent que l'utilisation multithread est détectée et ne puisse pas conduire à des comportements non voulus.

Les Sinks peuvent être facilement transformés en Flux ou Mono



# Catégories de Sink

---

***many().multicast()*** : Transmet uniquement les données nouvellement poussées à ses abonnés, respectant leur back-pressure (données générées après l'abonnement).

***many().unicast()*** : Idem avec un seul abonné.

***many().replay()*** : Rejoue un certains nombres d'événements pour les nouveau abonnés (Retour arrière).

***one()*** : Produit un seul élément

***empty()*** : Seulement un élément terminal.