





Programmation Reactive Spring

David THIBAU – 2023

david.thibau@gmail.com



Agenda

- Introduction

- Offre Spring
- Motivation pour le réactif
- La programmation réactive

- Reactor

- Cœur de l'API : Mono et Flux
- Threads, Scheduler
- Traitement des erreurs
- Test, Debug
- Propagation de contexte

- Pile réactive

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

- Tests

- *WebTestClient*



Introduction

Offre Spring

Motivation pour le réactif
Programmation Reactive



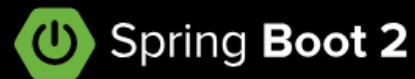
Introduction

Depuis Spring5, Spring propose un mode de programmation réactive

- Ce mode de programmation s'appuie sur la librairie de bas-niveau **Reactor** qui est présente dans les projets : *Spring Data Reactive, WebFlux, Spring Cloud Gateway* , ...

Cela vise des systèmes à faible latence et de gros débits grâce à une meilleure utilisation des ressources CPU

2 stacks



Optional Dependency

Reactive Stack

Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.

Netty, Servlet 3.1+ Containers

Reactive Streams Adapters

Spring Security Reactive

Spring WebFlux

Spring Data Reactive Repositories

Mongo, Cassandra, Redis, Couchbase, R2DBC

Servlet Stack

Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.

Servlet Containers

Servlet API

Spring Security

Spring MVC

Spring Data Repositories

JDBC, JPA, NoSQL



Autres nouveautés Spring 5

- Java 8+
- Support avec Kotlin 1.1+
- Compatibilité avec les extensions de JUnit 5
- JPA 2.1 et Hibernate 5
- Abandon du support pour de nombreuses librairies :
PortletMVC, JDO, Guava caching, JasperReports, OpenJPA, Tiles 2, XMLBeans, Velocity.
- Java EE 8 API level (e.g. Servlet 4.0 et HTTP/2)
- Compatible Spring Boot 2.x



Nouveautés Spring 6

- Java 17+
- Kotlin 1.7+
- Gradle 7.3+
- Lombok 1.18.22+
- Natifs exécutables
- Observabilité basée sur MicroMeter
- Jakarta EE 9 (javax.* devient jakarta.*)
- Compatible Spring Boot 3.x



Introduction

Offre Spring
Motivation pour le réactif
Programmation Reactive



Performance des applications

Les applications modernes peuvent atteindre un grand nombre d'utilisateurs simultanés

- La performance du matériel pour ce type d'application reste cruciale

2 façons d'augmenter la performance :

- Paralléliser afin d'utiliser plus de threads et plus de ressources matérielles.
- Optimiser l'utilisation des ressources disponibles.



Parallélisation

Augmenter le nombre de threads et y exécuter du code bloquant peut introduire des problèmes de conflit et de concurrence.

Mais surtout, le code bloquant gaspille également des ressources.

- Dès qu'un programme implique une certaine latence (E/S, comme requête BD ou appel réseau), les ressources sont gaspillées car les threads restent inactifs, attendant des données.

La parallélisation n'est donc pas une solution optimale pour augmenter la performance



Code asynchrone

Le code asynchrone permet une meilleure utilisation des ressources :

- Avec du code asynchrone et non bloquant, l'exécution bascule vers une autre tâche active utilisant les mêmes ressources sous-jacentes et revient plus tard au processus en cours lorsque le traitement asynchrone est terminé.

Java propose 2 modèles de programmation asynchrone :

- Les **Callbacks**
- Les **Future** et **CompletableFuture**



Callbacks

Principe :

Les méthodes asynchrones n'ont pas de valeur de retour mais prennent un paramètre de callback (une classe lambda ou anonyme) qui est appelée lorsque le résultat est disponible.

Ex : EventListener de Swing.

Inconvénient :

Les callbacks sont difficiles à composer, ce qui conduit rapidement à un code difficile à lire et à maintenir : "Callback Hell".



Exemple Callback-Hell

```
userService.getFavorites(userId, new Callback<List<String>>() {
    public void onSuccess(List<String> list) {
        if (list.isEmpty()) {
            suggestionService.getSuggestions(new Callback<List<Favorite>>() {
                public void onSuccess(List<Favorite> list) {
                    UiUtils.submitOnUiThread(() -> { list.stream().limit(5).forEach(uiList::show); });
                }
                public void onError(Throwable error) {
                    UiUtils.errorPopup(error);
                }
            });
        } else {
            list.stream().limit(5).forEach(favId -> favoriteService.getDetails(favId, new Callback<Favorite>() {
                public void onSuccess(Favorite details) {
                    UiUtils.submitOnUiThread(() -> uiList.show(details));
                }
                public void onError(Throwable error) {
                    UiUtils.errorPopup(error);
                }
            }));
        }
    }
    public void onError(Throwable error) {
        UiUtils.errorPopup(error);
    }
});
```



Équivalent Reactor

```
userService.getFavorites(userId)
    .flatMap(favoriteService::getDetails)
    .switchIfEmpty(suggestionService.getSuggestions())
    .take(5)
    .publishOn(UiUtils.uiThreadScheduler())
    .subscribe(uiList::show, UiUtils::errorPopup);
```



Future<T>

Principe:

Les méthodes asynchrones renvoient immédiatement un *Future<T>*.

La valeur calculée n'est pas immédiatement disponible.

L'objet peut être interrogé jusqu'à ce que la valeur soit disponible.

Exemple : *ExecutorService* exécutant des tâches *Callable<T>* utilise des objets *Future*.

Inconvénients :

- Composer plusieurs objets *Future* est faisable mais pas facile.
- La méthode *get()* rend le code bloquant.
- Pas possible de coder du code lazy.
- Ne prennent pas en charge plusieurs valeurs ni une gestion avancée des erreurs.



Exemple CompletableFuture

```
// Liste d'IDs à traiter
CompletableFuture<List<String>> ids = ifhIds();
// A partir de l'id retrouver un nom et une tâche
CompletableFuture<List<String>> result = ids.thenComposeAsync(l -> {
    Stream<CompletableFuture<String>> zip = l.stream().map(i -> {
        CompletableFuture<String> nameTask = ifhName(i);
        CompletableFuture<Integer> statTask = ifhStat(i);
        return nameTask.thenCombineAsync(statTask,
            (name, stat) -> "Name " + name + " has stats " + stat);
    });
// Pour exécuter les tâches conversion list vers array.
List<CompletableFuture<String>> combinationList = zip.collect(Collectors.toList());
CompletableFuture<String>[] combinationArray = combinationList.toArray(new
CompletableFuture[combinationList.size()]);
// allOf Génère un Future qui se termine lorsque toutes les tâches sont terminées.
CompletableFuture<Void> allDone = CompletableFuture.allOf(combinationArray);
return allDone.thenApply(v -> combinationList.stream()
    .map(CompletableFuture::join)
    .collect(Collectors.toList()));
});
// Attente que la pipeline se termine pour obtenir les résultats
List<String> results = result.join();
```



Équivalent Reactor

```
Flux<String> ids = ifhrIds();

Flux<String> combinations = ids.flatMap(id -> {
    Mono<String> nameTask = ifhrName(id);
    Mono<Integer> statTask = ifhrStat(id);

    return nameTask.zipWith(statTask,
        (name, stat) -> "Name " + name + " has stats " +
            stat);
});

Mono<List<String>> result = combinations.collectList();

List<String> results = result.block();
```



Bibliothèques réactives

Les bibliothèques réactives, telles que Reactor, visent à pallier les inconvénients des approches asynchrones "classiques" tout en se concentrant sur quelques aspects supplémentaires :

- Composabilité et lisibilité
- Le flux de données est manipulé avec un riche vocabulaire d'opérateurs
- Rien ne se passe tant que vous n'êtes pas abonné
- Back-pressure : Capacité pour le consommateur de signaler au producteur que le taux d'émission est trop élevé
- Abstraction de haut niveau indépendante du modèle de concurrence



Introduction

Offre Spring
Motivation pour le réactif
Programmation Reactive



Définition Wikipedia

La programmation réactive est un paradigme de programmation asynchrone concerné par les flux de données et la propagation du changement.

Cela signifie qu'il devient possible d'exprimer facilement des flux de données statiques (par exemple des tableaux) ou dynamiques (par exemple des émetteurs d'événements) via le(s) langage(s) de programmation utilisé(s).



Pattern et *ReactiveX*

La programmation réactive :

- se base sur le pattern **Observable** qui est une combinaison des patterns *Observer* et *Iterator*
- utilise la programmation fonctionnelle pour définir facilement des opérateurs sur le flux
- est formalisée par l'API **ReactiveX**.
De nombreuses implémentations existent (*RxJS*, *RxJava*, *Rx.NET*)



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 ne définit pas les opérateurs (chaque implémentation à ses propres opérateurs)
- Une implémentation existe dans Java9 : *Flow API*



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> {  
}
```




Analogie : Chaîne de montage

Une application réactive ressemble à chaîne de montage. Reactor joue le rôle du convoyeur et des postes de travail

La matière première provient d'une source (**Publisher**) et finit comme un produit fini prêt à être poussé vers le consommateur (**Subscriber**).

- La matière première peut passer par diverses transformations ou faire partie d'une chaîne de montage plus large qui agrège des pièces intermédiaires.
- S'il y a un problème à un moment donné , le poste de travail affecté peut signaler en amont pour limiter le flux de matière première.





Opérateurs

Les **opérateurs** sont les postes de travail de la chaîne de montage.

Chaque opérateur ajoute un comportement à un *Publisher* et encapsule le *Publisher* de l'étape précédente dans une nouvelle instance.

- Les données provenant du premier *Publisher* traversent la chaîne d'opérateur en étant transformé par chaque maillon.

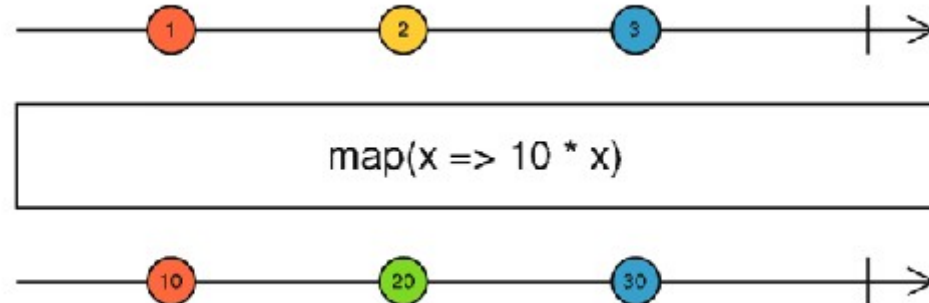
L'apport des bibliothèques réactives comme Reactor est de fournir un ensemble riche d'opérateurs couvrant un large spectre : simple transformation, filtre, orchestration, traitement d'erreur, ...



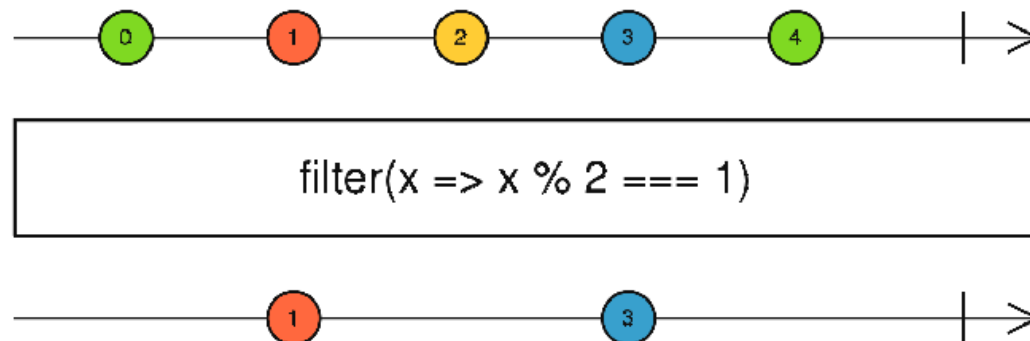
Description opérateurs

La documentation des opérateurs utilise des ***marble diagrams***

Exemple *map* :



Exemple *filter*





Déclenchement via *subscribe*

Lorsque l'on définit une chaîne de *Publisher*, les données ne commencent pas à circuler.

- Ce n'est qu'une description abstraite du processus asynchrone

Lors d'un abonnement via la méthode ***subscribe***, le flux est déclenché dans toute la chaîne

- En interne, le Subscriber invoque la méthode ***request*** qui est propagée jusqu'à la source



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 le back pressure 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.



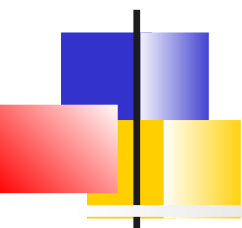
Utilisation back-pressure

Un Subscriber peut :

- travailler en mode illimité et laisser la source pousser toutes les données le plus rapidement possible
- ou il peut utiliser la méthode *request(n)* pour signaler à la source qu'il est prêt à traiter au plus n éléments

Les opérateurs intermédiaires peuvent également modifier la demande en transit.

- Par exemple, un opérateur *buffer* regroupe des éléments par lots.
Si il recoit *request(1)* alors
il effectue 1 appel à *request(tailleBuffer)*



Séquence chaude ou froide

On peut distinguer 2 types de séquences en fonction dont le flux réagit aux abonnés :

- Une séquence **Cold** recommence à la source pour chaque *Subscriber*.
Exemple : si la source encapsule un appel HTTP, une nouvelle requête est effectuée pour chaque abonnement.
- Une séquence **Hot** ne repart pas de zéro pour chaque *Subscriber*. Les abonnés en retard reçoivent les signaux émis après leur abonnement.
 - Certains flux réactifs chauds peuvent cependant mettre en cache ou rejouer l'historique des émissions.
 - Une séquence chaude peut même émettre lorsqu'aucun abonné n'écoute (une exception à la règle « rien ne se passe avant la souscription »)



Reactor

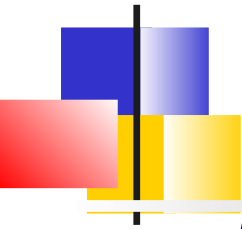
Cœur de l'API

Threads et Scheduler

Traitement des Erreurs

Test, Debug

Propagation de contexte

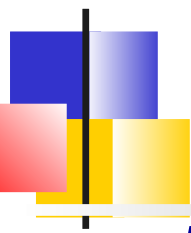


Reactor

Reactor se concentre sur la programmation réactive côté serveur.

Il est développé conjointement avec Spring.

- Il fournit principalement les types **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, ...*



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

Ce sont des implémentations de ***Publisher*** de *Reactive Stream* qui définit 1 méthode :

```
void subscribe(Subscriber<? super T> s)
```

Le flux commence à émettre seulement si il y a un abonné



Flux

Un ***Flux*** $\langle T \rangle$ représente une séquence asynchrone de 0 à N événements, optionnellement terminée par un signal de fin ou une erreur.

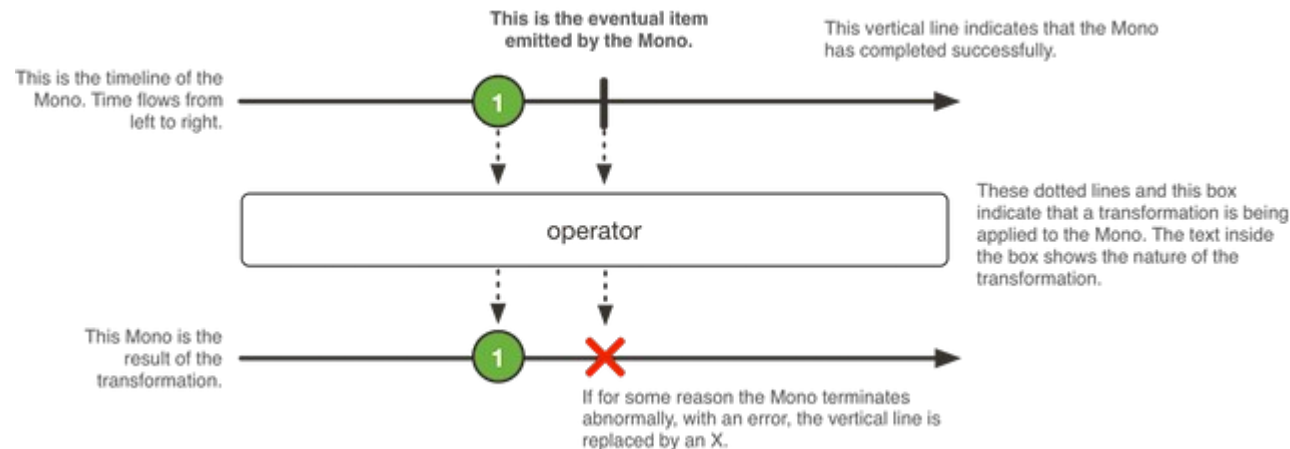
Les événements sont traduits par des appels de méthode sur les abonnés :

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

Mono

Mono $\langle T \rangle$ représente une séquence de 0 à 1 événement, optionnellement terminée par un signal de fin ou une erreur

Mono offre un sous-ensemble des opérateurs de Flux





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 BD 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 les lambda-expression, on peut fournir une implémentation de ***Subscriber*** qui définit 4 méthodes :

// Appelée lors de l'abonnement

```
void onSubscribe(Subscription s)
```

```
void onNext(T t)
```

```
void onComplete()
```

```
void onError(java.lang.Throwable t)
```



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() {}
    });
```



Opérateurs

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

- Transformer
- Filtrer
- Exécuter du code à chaque événements
- Opérateurs temporels
- Séparer un flux
- Revenir au mode synchrone
- Gérer des erreurs



Transformation

1 vers 1 :

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

1 vers N :

flatMap + une méthode factory, *handle*

Ajouter des éléments à une séquence :

startsWith, *concatWith*, *concatWithValues*

Agréger :

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

Agréger en booléen :

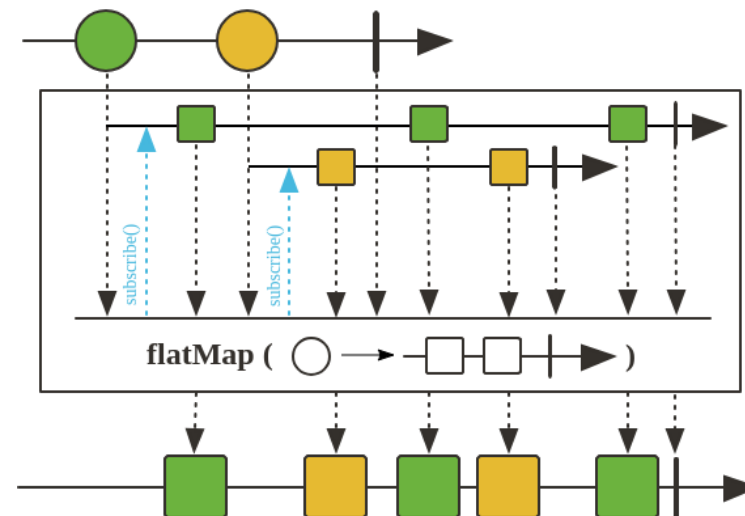
all, *any*, *hasElements*, *hasElement*

Combiner plusieurs flux :

concat, *merge*, *zip*

flatMap

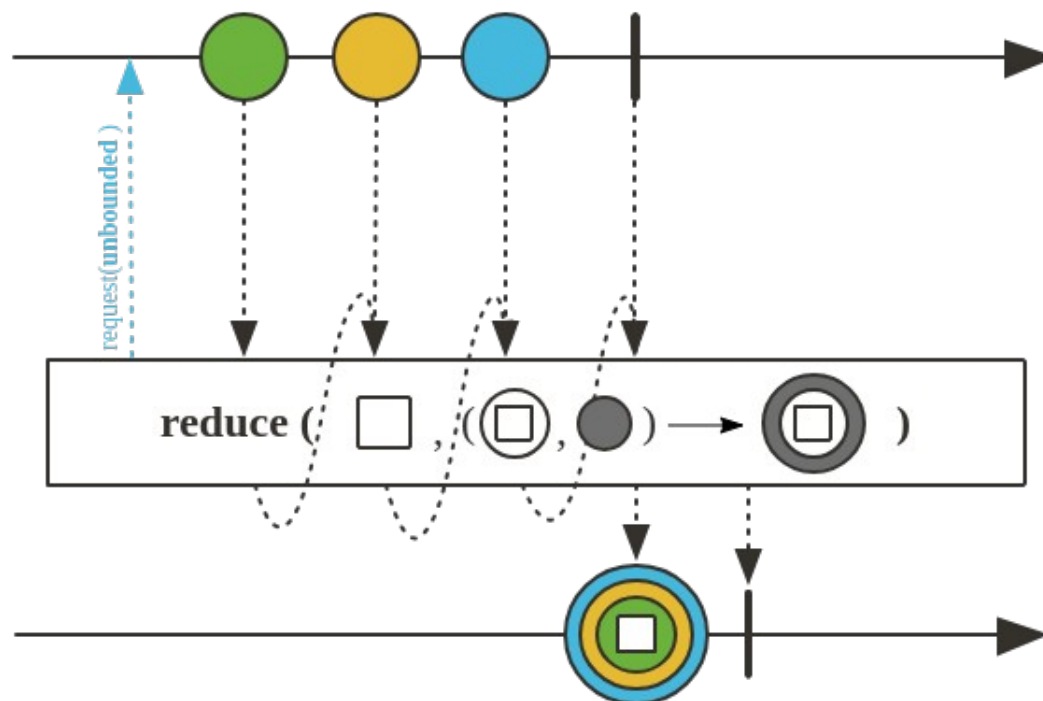
```
Function<String, Publisher<String>> mapper =  
    s -> Flux.just(s.toUpperCase().split(""));  
Flux<String> inFlux = Flux.just("baeldung", ".", "com");  
Flux<String> outFlux = inFlux.flatMap(mapper);  
  
List<String> output = new ArrayList<>();  
outFlux.subscribe(output::add);  
assertThat(output).containsExactlyInAnyOrder("B", "A", "E", "L", "D",  
    "U", "N", "G", ".", "C", "O", "M");
```





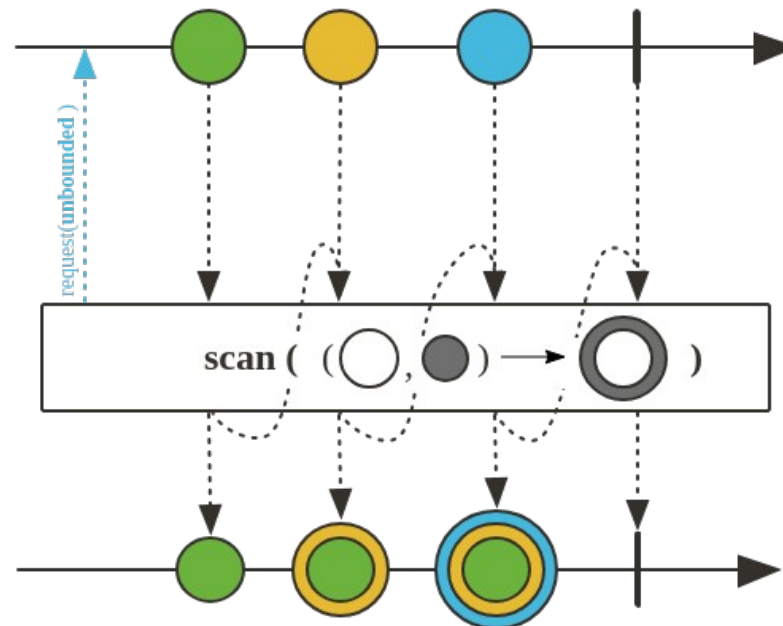
reduce

```
Flux.just(1, 2, 3, 4)
    .reduce(0, (x1, x2) -> x1 + x2)
    .subscribe(System.out::println) ;
```

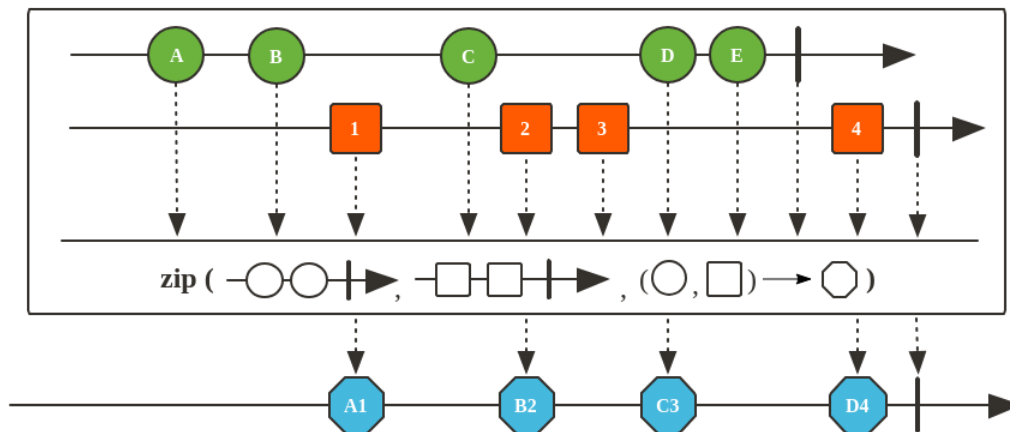
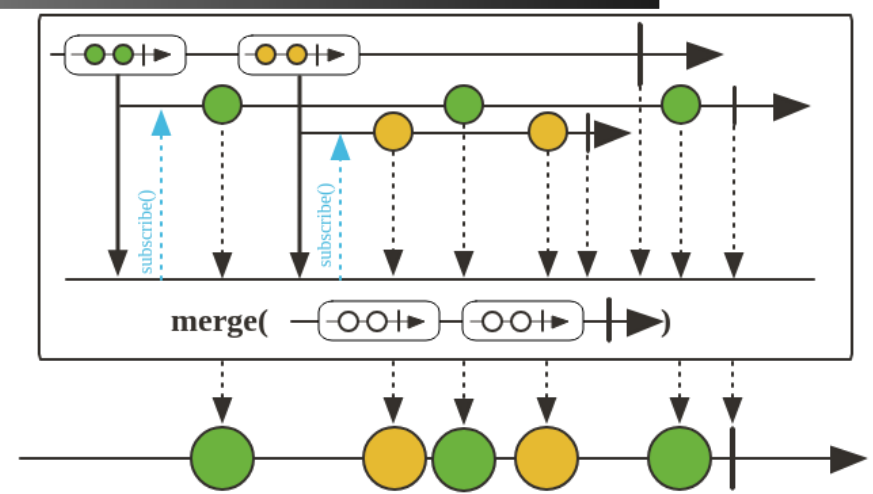
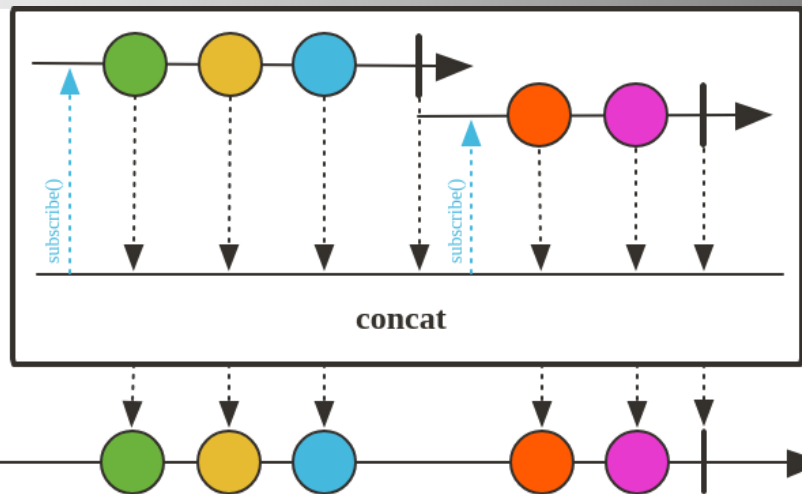


scan

```
Flux.just(1, 2, 3, 4)
  .scan((x1, x2) -> x1 + x2)
  .subscribe(System.out::println) ;
```



Combinaison





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



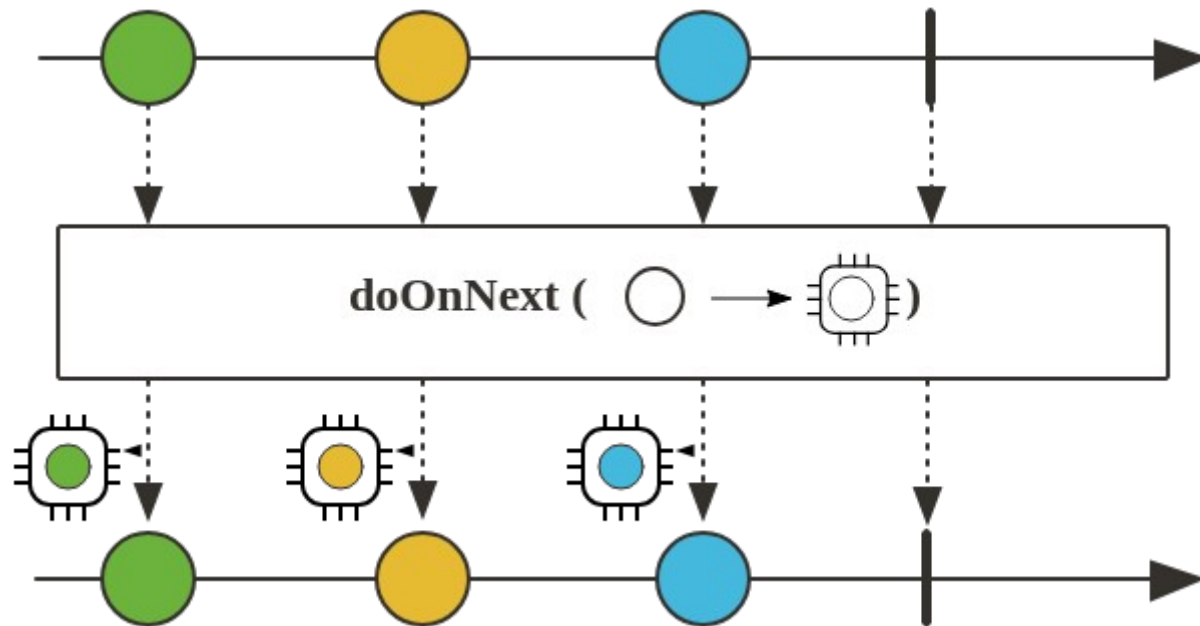
Opérateurs doXXX

Les opérateurs doXXX permettent d'être notifié ou exécuter du code supplémentaire (parfois appelé « effets secondaires ») à chaque événement

- doOnNext : Chaque émission de donnée
- doOnComplete
- doOnError
- doOnCancel
- doOnFirst
- doOnSubscribe
- ...

doOnNext

`Flux<T> doOnNext(Consumer<? super T> onNext)`





Opérateurs temporels

Associer l'événement à un timestamp :

elapsed, timestamp

Séquence interrompue si délai trop important
entre 2 événements :

timeout

Séquence à intervalle régulier :

interval

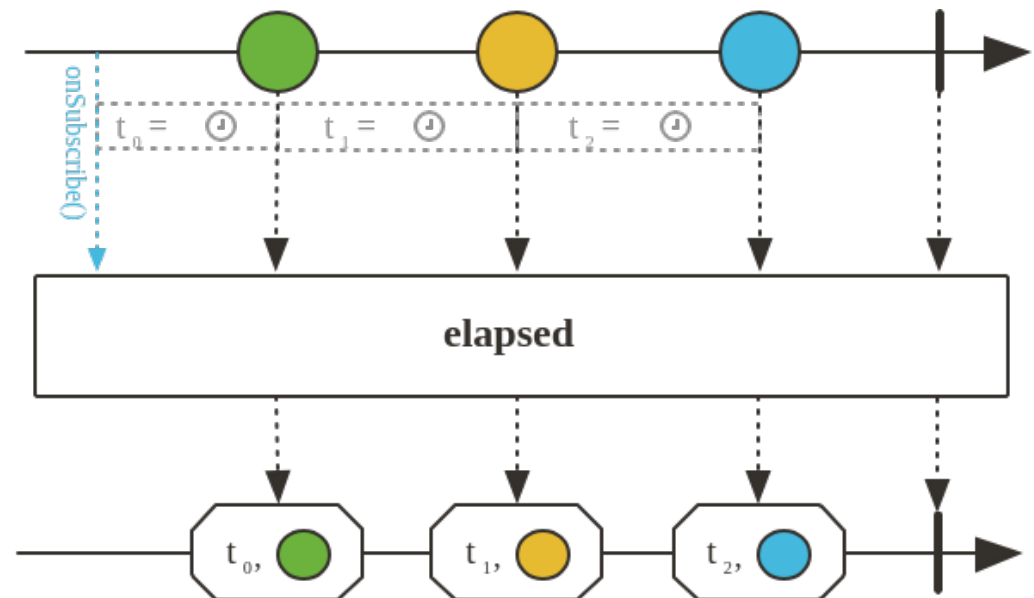
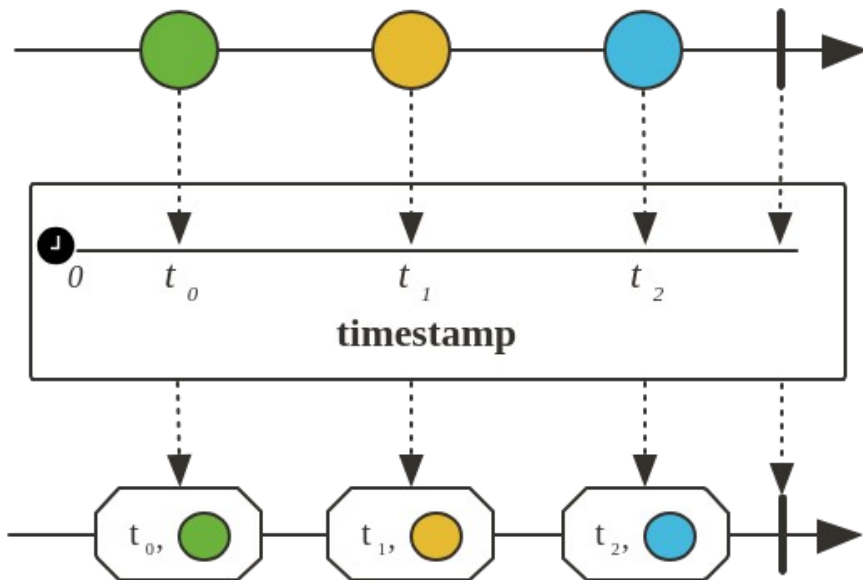
Ajouter des délais

Mono.delay, delayElements, delaySubscription

timestamp / elapsed

`Flux<Tuple2<Long, T>> timestamp()`

`Flux<Tuple2<Long, T>> elapsed()`





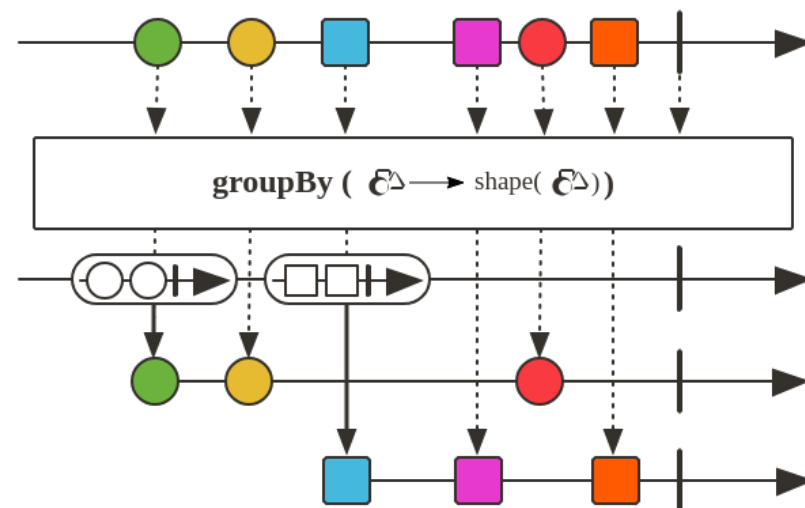
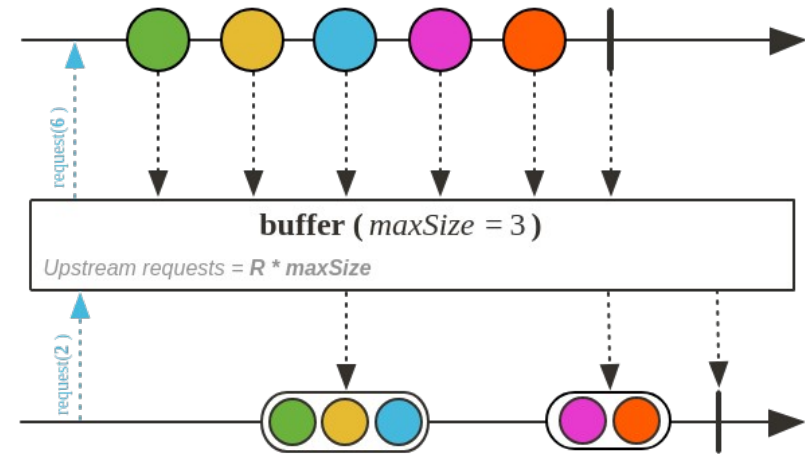
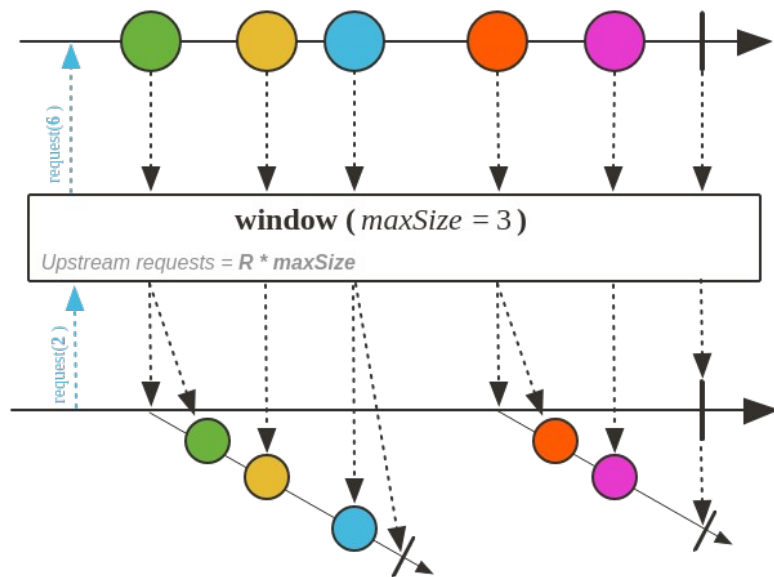
Séparer un flux

Séparer un $\text{Flux}\langle T \rangle$ dans $\text{Flux}\langle \text{Flux}\langle T \rangle \rangle$
window

Séparer un $\text{Flux}\langle T \rangle$ dans une
 $\text{Flux}\langle \text{List}\langle T \rangle \rangle$
buffer

Séparer un Flux en des sous-flux dont les
éléments partagent une caractéristique
groupBy

Séparation





Retour au mode synchrone

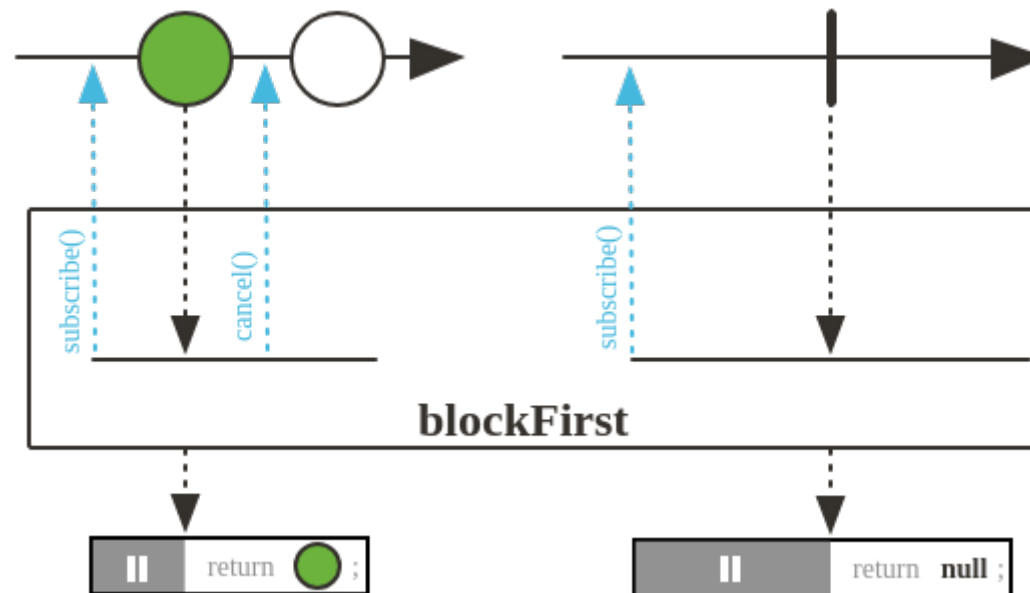
Ces opérateurs peuvent lancer une *UnsupportedOperationException* si ils sont appelés dans un Scheduler marqué comme non-bloquant.

blockFirst, blockLast, Mono.block

tolerable, toStream

toFuture

blockFirst





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



Introduction

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 threads séparés

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

L'interface ***Scheduler*** fournit une abstraction pour du code asynchrone

De nombreuses implémentations 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()*** : Optimisé pour des exécutions avec de faibles latences
- ***Schedulers.immediate()*** : Exécution immédiate des tâches sur la thread de l'appelant.



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 *subscribeOn* le plus tôt dans la chaîne est pris en compte.



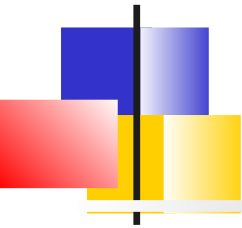
Clean up

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");
```

// Flux 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 de l'erreur (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);
```



StackTrace

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*



Exemple StackTrace

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
    at reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:129)
    at reactor.core.publisher.FluxFlatMap$FlatMapMain.tryEmitScalar(FluxFlatMap.java:445)
    at reactor.core.publisher.FluxFlatMap$FlatMapMain.onNext(FluxFlatMap.java:379)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:121)
    at reactor.core.publisher.FluxRange$RangeSubscription.slowPath(FluxRange.java:154)
    at reactor.core.publisher.FluxRange$RangeSubscription.request(FluxRange.java:109)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)
    at reactor.core.publisher.FluxFlatMap$FlatMapMain.onSubscribe(FluxFlatMap.java:332)
    at reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)
    at reactor.core.publisher.FluxRange.subscribe(FluxRange.java:68)
    at reactor.core.publisher.FluxMapFuseable.subscribe(FluxMapFuseable.java:63)
    at reactor.core.publisher.FluxFlatMap.subscribe(FluxFlatMap.java:97)
    at reactor.core.publisher.MonoSingle.subscribe(MonoSingle.java:58)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3096)
    at reactor.core.publisher.Mono.subscribeWith(Mono.java:3204)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3090)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3057)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3029)
    at reactor.guide.GuideTests.debuggingCommonStacktrace(GuideTests.java:995)
```



Analyse

Le haut de la stack mentionne *MonoSingle* qui doit n'envoyer qu'un seul élément.

Les autres niveaux ne sont que des successions d'appels à *subscribe* et *request*

La chaîne semble impliquer un *MonoSingle*, un *FluxFlatMap* et un *FluxRange*

La dernière ligne référence le code applicatif qui peut être :

```
toDebug
    .subscribeOn(Schedulers.immediate())
    .subscribe(System.out::println, Throwable::printStackTrace);
```

Mais le Flux n'est pas déclaré ni instancié au même endroit du code

Peut-être même y a t il plusieurs façons d'instancier ce Flux.

Lequel pose problème ?

Ce que nous voudrions savoir plus facilement, c'est où le flux a été déclaré et assembler



Debug

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
- L'activation globale s'effectue par :

Hooks.onOperatorDebug();

- La désactivation s'effectue par :

Hooks.resetOnOperatorDebug();

- L'activation/désactivation n'affecte pas les Flux/Mono déjà instanciés.



Debug StackTrace

java.lang.IndexOutOfBoundsException: Source emitted more than one item

at reactor.core.publisher.MonoSingle\$SingleSubscriber.onNext(MonoSingle.java:127)

Suppressed: The stacktrace has been enhanced by Reactor, refer to additional information below:

Assembly trace from producer [reactor.core.publisher.MonoSingle] :

reactor.core.publisher.Flux.single(Flux.java:7915)

reactor.guide.GuideTests.scatterAndGather(GuideTests.java:1017)

Error has been observed at the following site(s):

* _____ Flux.single → at reactor.guide.GuideTests.scatterAndGather(GuideTests.java:1017)

|_ Mono.subscribeOn → at reactor.guide.GuideTests.debuggingActivated(GuideTests.java:1071)

Original Stack Trace:

at reactor.core.publisher.MonoSingle\$SingleSubscriber.onNext(MonoSingle.java:127)

...

...

at reactor.core.publisher.Mono.subscribeWith(Mono.java:4363)

at reactor.core.publisher.Mono.subscribe(Mono.java:4223)

at reactor.core.publisher.Mono.subscribe(Mono.java:4159)

at reactor.core.publisher.Mono.subscribe(Mono.java:4131)

at reactor.guide.GuideTests.debuggingActivated(GuideTests.java:1067)



Analyse

La trace de Debug

- tronque la trace originale
- Ajoute des détails sur où la chaîne a été assemblée
- Indique une notion de chaîne(s) d'opérateur(s) à travers laquelle l'erreur s'est propagée, du premier au dernier
Chaque opérateur qui a vu l'erreur est mentionné avec la classe d'utilisateur et la ligne où il a été utilisé

Dans l'exemple précédent, le single qui a causé le problème a été créé dans la méthode *scatterAndGather*.



Reactor

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



Introduction

La propagation de contexte permet de conserver et propager un état dans le pipeline 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.



API

Depuis la version 3.1.0, Reactor propose l'interface **Context** (~Map) permettant de stocker des paires clés-valeurs.

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

Context expose une version en read-only :
ContextView



Remplissage du Context

Pour qu'un Contexte soit utile, il doit être lié à une séquence spécifique et être accessible par chaque opérateur d'une chaîne.

Le remplissage du contexte ne peut être fait qu'au moment de l'abonnement via l'opérateur ***contextWrite(contextView)*** qui fusionne le *ContextView* fourni avec le *Context* existant

- Cela est fait via la méthode *putAll()* qui crée une nouvelle instance



Lecture du contexte

Plusieurs façon d'accéder en lecture au Contexte :

- A partir d'une opérateur source, on peut utiliser la méthode ***deferContextual***

Ex : `Mono.deferContextual(ctx →
Mono.just(ctx.get(key)))` ;

- 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));  
}))
```

- Une construction classique est l'utilisation de l'opérateur `flatMap` en milieu de séquence
Voir slide suivant



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 pour l'instant une API 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.x+)



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());

```

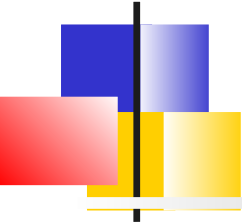


Reactive Repository

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)  
..
```



Mélange bloquant non-bloquant

S'il on doit exécuter code bloquant, il faut absolument éviter de bloquer la thread principale (qui typiquement exécute la boucle de réception d'événements comme la réception de requête http).

Une possibilité est d'utiliser les *Scheduler*.



Requêtes

Les requêtes peuvent être déduites du nom des fonctions :

```
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>${spring-boot.version}</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 une base relationnelle en mode réactif

Les principales BD offrent des drivers réactifs :
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

@Column: Mapper un attribut vers une colonne.

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



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***

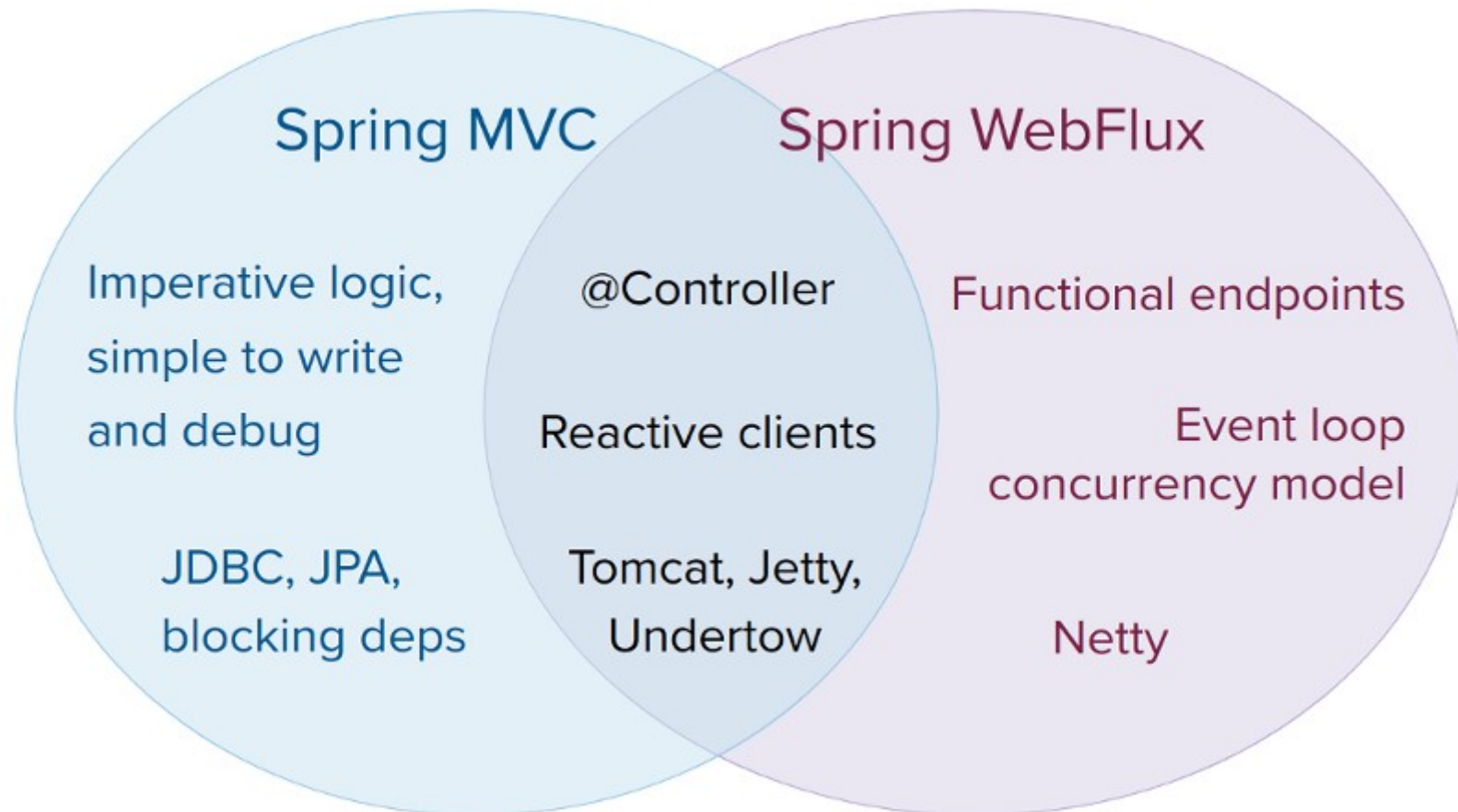
Spring Webflux offre en plus 2 modèles de programmation :

- **Contrôleurs annotés** : Idem à Spring MVC avec les mêmes annotations.
Les méthodes des contrôleurs retournent des types réactifs, des arguments réactifs sont associés à *@RequestBody*.
- **Endpoints fonctionnels** : Programmation fonctionnelle basée sur les lambdas.
Idéal pour de petites libraires 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 les versions récentes 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*



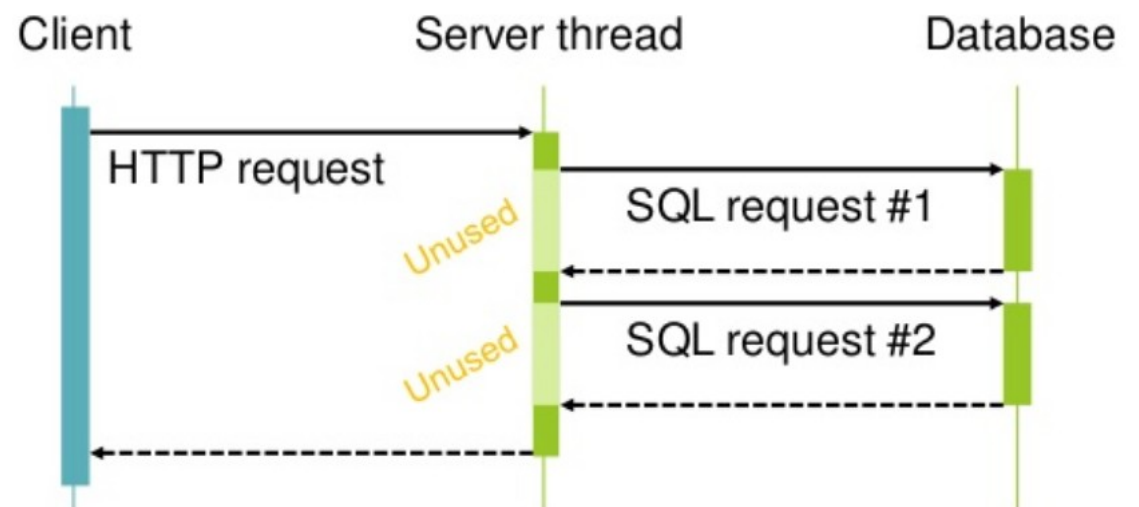
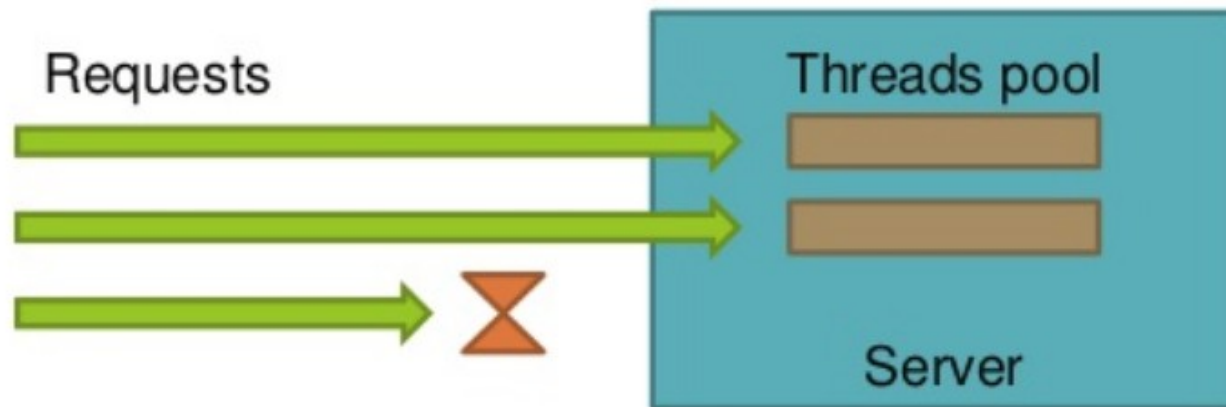
Performance et Scaling

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)

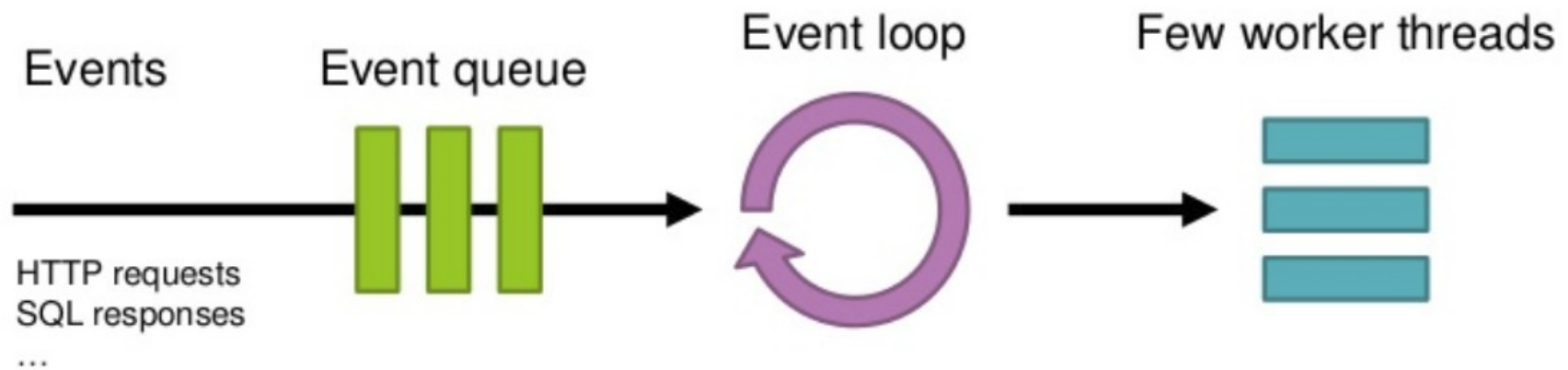
Le bénéfice attendu est la possibilité de **scaler** avec un petit nombre de threads fixes et moins de mémoire. Cela rend les applications plus résistantes à la charge.

Pour bénéficier du modèle réactif, l'application doit présenter de la latence provoquée par des appels I/O. (Requêtes réseau des applications micro-services par exemple)

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***, ***ResponseEntity*** : 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.

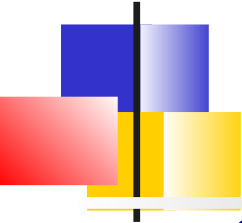


Endpoints fonctionnels

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 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 :

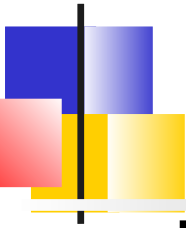
```
HandlerFunction<ServerResponse> helloWorld =  
    request ->  
    ServerResponse.ok().body(fromObject("Hello World"));
```

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);
```



Exécution sur un serveur

Pour exécuter une *RouterFunction* sur un serveur, il faut le convertir en *HttpHandler*.

Dans un contexte SpringBoot cela est fait automatiquement



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*** : alternative non-bloquante à *RestTemplate*.

- 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*)



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 est une technologie qui fournit une communication asynchrone du serveur vers le 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



SpringWebflux 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());
```



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
            .authorizeExchange()
            .anyExchange().authenticated()
            .and()
            .httpBasic().and()
            .formLogin();
        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



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
```

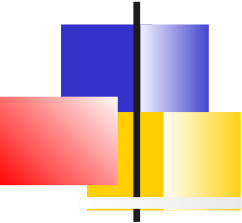


Extraction du corps de requête

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>



Annexes

Java8 et Java9
JUnit5
Annotations Tests



Lambda expressions

Les ***expressions Lambda*** (ou closure ou anonymous function) sont un moyen de simplifier le code

Exemple:

```
button.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        operationQuelconque(e);  
    }  
});
```

Équivalent à:

```
button.addActionListener( e ->operationQuelconque(e) );
```

L'expression Lambda s'écrit entre les parenthèses d'appel à une méthode

La méthode respecte une interface fonctionnelle (*Functional interface*)



Utilisation

// Appel de méthodes

```
Arrays.sort(testStrings, (s1,s2) -> s1.length() - s2.length())
```

// Variables

```
AutoCloseable c = () -> cleanupForTryWithResources() ;
```

// Retour implicite

```
(arg1,arg2) -> qqch;
```

// Parenthèses omises si 1 seul argument

```
button1.addActionListener(event->setBackground(Color.BLUE));
```

// Accès en lecture à une variable locale

```
Color b2c = Color.GREEN;  
boutton2.addActionListener( event -> setBackground(b2c));
```




Référence de méthode

Les **références de méthode** sont utilisées lorsque la lambda expression ne fait qu'appeler une méthode.

// Référence à méthode statique

String::valueOf

// Méthode d'instance sur un objet

s::toString

// Méthode d'instance non liée à un objet particulier

String::toUpperCase

// Constructeur

String::new



Interface Fonctionnelle

Le package ***java.util.function*** contient plus de 40 interfaces fonctionnelles destinées à faciliter l'écriture d'expressions Lambdas ou de références de méthodes

Par exemple :

Consumer : Fonction sans retour acceptant un type T en argument

Supplier : Fonction sans argument retournant un type T

Predicate : Test d'une condition

Function : Fonction prenant un type T en argument et retournant un type R



Stream

Un ***Stream*** permet d'envelopper temporairement des données issues de tableaux ou de collections afin de les traiter massivement d'une manière efficace

- il permet d'effectuer des opérations séquentielles ou en parallèle sur les données
- Adapté aux expressions Lambda
- Pas d'accès indexé aux données

Un *Stream* est consommable, et de ce fait il n'est pas possible de créer une référence sur un stream pour un usage ultérieur



Types d'opérations

Une opération **intermédiaire** retourne un nouveau Stream

- *filter* et *map* par exemple

Une opération **terminale** retourne un résultat ou produit un effet de bord

- *forEach*, *toArray*, *min*, *max*, *findFirst*, *anyMatch*, *allMatch* etc...

Une opération **intermédiaire est appelée court-circuit** si elle produit un Stream fini à partir d'un Stream infini

- exemple *limit()* et *skip()*

Une opération **terminale est appelée court-circuit** si elle peut terminer une opération dans un temps fini sur un stream infini

- exemple *anyMatch*, *allMatch*, *noneMatch*, *findFirst* et *findAny*



Exemples

// forEach

```
Stream<Employe> employes = getEmployes().stream();  
employes.forEach(e -> e.setSalaire(e.getSalaire() * 11/10)) ;
```

// map

```
Double[] carres = Stream.of(valeurs).map(n ->  
                                         n * n).toArray(Double[]::new);
```

// filter

```
Integer[] evens = Stream.of(nums).filter(n ->  
                                         n%2 == 0).toArray(Integer[]::new);
```

// findFirst

```
Stream<String> amis = Stream.of("Nicolas", "Sophie", "Chloe");  
Optional<String> prenom = amis.filter(i ->  
                                     i.startsWith("S")).findFirst();
```

// Reduce

```
List<Double> nums = Arrays.asList(1.2, -2.3, 4.5, -5.6);  
double max = nums.stream().reduce(Double.MIN_VALUE, Double::max);  
double produit = nums.stream().reduce(1, (n1, n2) -> n1 * n2);
```



CompletableFuture

CompletableFuture améliore *Future* avec les lambda expression, la gestion des erreurs, la possibilité de chaîner les traitements asynchrones, ou de les exécuter en //

```
CompletableFuture<String> completableFuture  
    = CompletableFuture.supplyAsync(() -> "Hello");
```

```
CompletableFuture<String> future = completableFuture  
    .thenApply(s -> s + " World");
```

```
assertEquals("Hello World", future.get());
```



Modules Java9

Similaire à OSGi : Les **modules** ont des dépendances et peuvent exporter une API publique et garder les détails d'implémentation cachés/privés

Pas nécessaire d'avoir tout Java pour exécuter des programmes
=> Permet d'exécuter Java sur de plus petit devices



Flow API Java 9

Implémentation de *ReactiveStream* avec en particulier :

- *Publishers* : Produit des Items
- *Subscribers* : Consomme les Items
- *Subscription* : Gère la relation entre *Publishers* et *Subscribers*
- *Processor* : A la fois *Publisher* et *Subscriber*



Autres nouveautés Java9

- Un nouveau client HTTP supportant HTTP/2 et les WebSocket
- API Process pour contrôler les processus natifs
- Logging de la JVM (GC, Threads, Compiler, Memory, ...)
- *Immutable Set, Optional Stream*



Annexes

Java8 et Java9
JUnit5
Annotations Tests



Le framework *JUnit5*

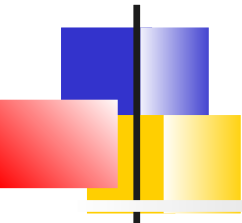
Avec *JUnit 5*, *JUnit* devient compatible Java8 et Lambda expression

JUnit 5 est composé de 3 sous-projets :

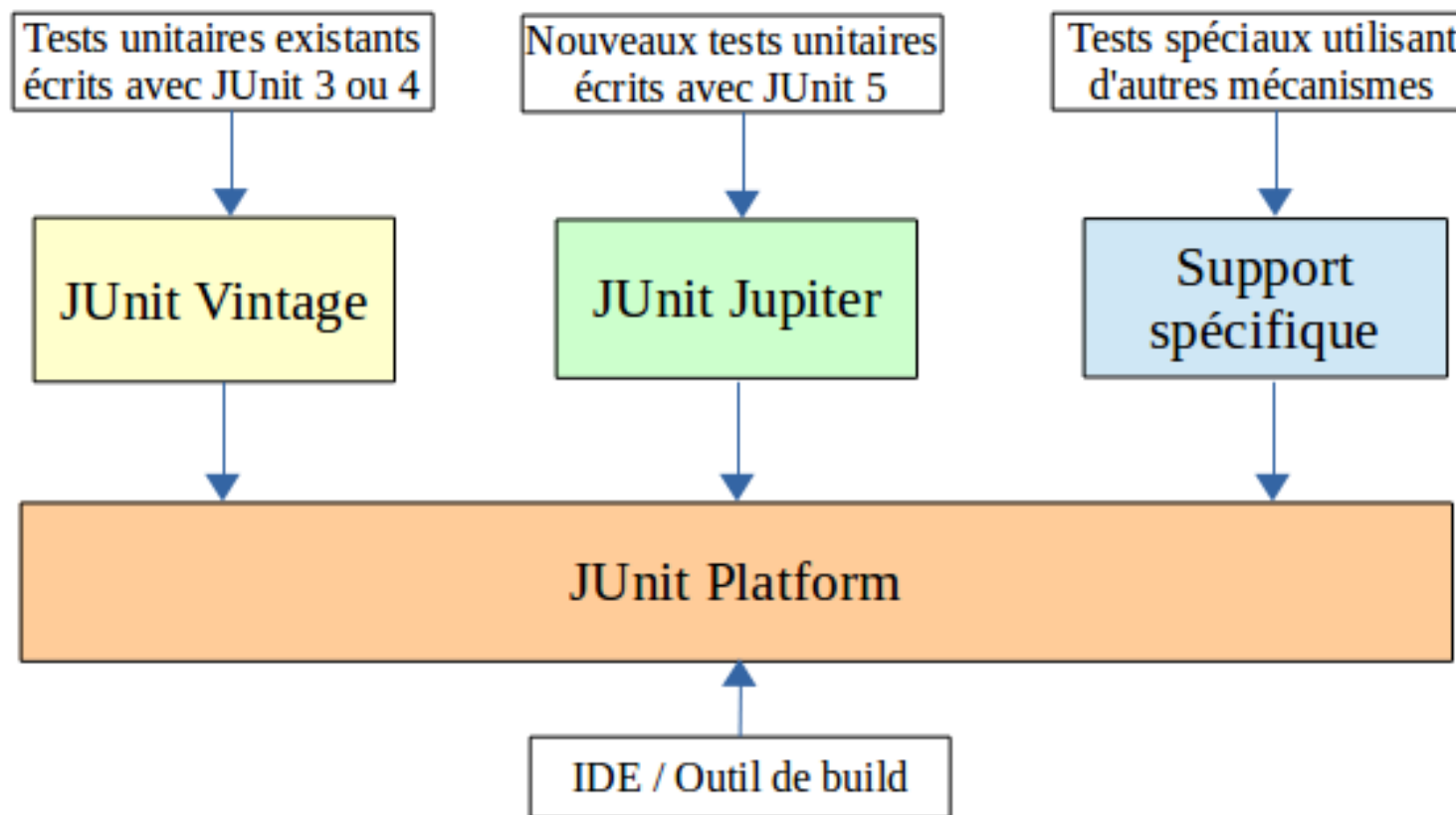
- ***JUnit Platform*** : Fondation pour lancer les tests.
Définit l'API *TestEngine*. Intégration avec outils de build.
Permet d'exécuter des tests *JUnit4*
- ***JUnit Jupiter*** : Permet l'exécution de tests *JUnit5*
- ***JUnit Vintage*** : Permet l'exécution de tests *JUnit3* et 4

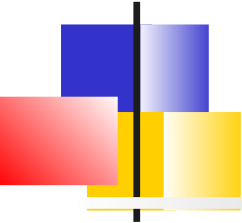
Projet exemples :

<https://github.com/junit-team/junit5-samples>



Compatibilité descendante





Annotations JUnit5

After, Before

@BeforeAll, @AfterAll : Méthodes exécutées une fois avant/après toutes les méthodes de Test.

Équivalent à *@BeforeClass, @AfterClass* de JUnit4

@BeforeEach, @AfterEach : Méthodes exécutées avant/après chaque méthode de test.

Équivalent à *@Before* et *@After* de JUnit4



Injection de dépendance

Afin que les méthodes de test soient exécutées en isolation, *JUnit* crée une nouvelle instance de la classe de test avant l'exécution de chaque méthode de test.

Jusqu'à *JUnit* 5, les constructeurs de classe de test ne pouvaient pas prendre d'arguments

Avec JUnit 5, on peut profiter **d'injection de dépendance** lors des constructeurs ou appel de méthodes de test



Example

```
@DisplayName("TestInfo Demo")
class TestInfoDemo {

    TestInfoDemo(TestInfo testInfo) {
        assertEquals("TestInfo Demo", testInfo.getDisplayName());
    }

    @BeforeEach
    void init(TestInfo testInfo) {
        String displayName = testInfo.getDisplayName();
        assertTrue(displayName.equals("TEST 1") || displayName.equals("test2()"));
    }

    @Test
    @DisplayName("TEST 1")
    @Tag("my-tag")
    void test1(TestInfo testInfo) {
        assertEquals("TEST 1", testInfo.getDisplayName());
        assertTrue(testInfo.getTags().contains("my-tag"));
    }
}
```



Assertions

Assert est remplacée par *Assertions* qui ajoute 4 nouvelles méthodes :

- ***assertAll*** qui regroupe en argument des lambdas exécutant d'autres assertions
- ***assertThrows*** pour indiquer qu'on s'attend à voir survenir une exception
- ***assertTimeout*** ou ***assertTimeoutPreemptively*** selon que l'on souhaite attendre ou non la fin d'exécution d'un traitement testé par rapport à une contrainte de temps

Les messages optionnels d'échec sont en dernier paramètre (lazy initialisation)

Hamcrest n'est plus inclus avec JUnit.



Exemples *JUnit5*

```
assertAll("address",  
() -> assertEquals("John", address.getFirstName()),  
() -> assertEquals("User", address.getLastName())) );
```

```
Throwable exception =  
    assertThrows(IllegalArgumentException.class, () ->  
    {  
        throw new IllegalArgumentException("a message");  
    });
```

```
assertTimeoutPreemptively(ofMillis(10), () -> {  
    Thread.sleep(100); });
```



Modèle d'extension JUnit5

Le modèle d'extension de JUnit5 repose dorénavant sur l'interface marqueur ***Extension*** avec l'annotation ***@ExtendWith***

Le moteur *JUnit* enregistre les *Extensions* présentes dans le classpath et les applique lorsqu'il voit une annotation *@ExtendWith* sur une classe ou sur une méthode.



Sous-classes d'extension

Les sous-classes des extensions sont :

- *BeforeAllCallback*
- *BeforeEachCallback*
- *BeforeTestExecutionCallback*
- *TestExecutionExceptionHandler*
- *AfterTestExecutionCallback*
- *AfterEachCallback*
- *AfterAllCallback*

Toutes les méthodes définies reçoivent en argument une classe de contexte englobant les informations nécessaires.



Annexes

Java8 et Java9
JUnit5

Annotations Tests



JUnit5

Spring 5 ajoute de nouvelles annotations compatibles exclusivement avec JUnit5

- **@SpringJUnitConfig** et **@SpringJUnitWebConfig** facilitant la création du contexte Spring
- **@EnabledIf/@DisabledIf** : Permettant de activer/désactiver des tests selon des conditions



@SpringJUnitConfig

@SpringJUnitConfig combine 2 annotations:

- **@ExtendWith(SpringExtension.class)** de JUnit 5 pour exécuter les tests avec l'extension *SpringExtension*
- **@ContextConfiguration** de Spring Testing pour charger le contexte Spring



Exemple

```
@SpringJUnitConfig(SpringJUnitConfigIntegrationTest.Config.class)
```

```
public class SpringJUnitConfigIntegrationTest {
```

```
    @Configuration
```

```
    static class Config {}
```

```
    @Autowired
```

```
    private ApplicationContext applicationContext;
```

```
    @Test
```

```
    void givenAppContext_WhenInjected_ThenItShouldNotBeNull() {
```

```
        assertNotNull(applicationContext);
```

```
    }
```

```
}
```



@SpringJUnitWebConfig

@SpringJUnitWebConfig est une combinaison de *@SpringJUnitConfig* + *@WebAppConfiguration*

Il permet de charger un *WebApplicationContext*.

```
@SpringJUnitWebConfig(TestConfig.class)
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```




Activer/Désactiver des tests

Les annotations **@EnabledIf** et **@DisabledIf** permettent d'activer ou désactiver des tests selon une expression :

- Spring Expression Language (SpEL)

```
@EnabledIf("#{systemProperties['os.name'].contains('Mac')}")
```

- Référence à une propriété de l'environnement Spring

```
@EnabledIf("${smoke.tests.enabled}")
```

- Texte

```
@EnabledIf("true")
```