

Programmation réactive avec Reactor

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-06

Table des matières

1	Introduction à la programmation réactive .	1
1.1	Définition de la programmation réactive	1
1.2	Comparaison avec la programmation impérative et asynchrone	2
2	Des API réactives dans différents langages	2
3	Introduction à la Reactive Streams API . . .	2
4	Exemple de Reactive Streams API en Java avec Project Reactor	3
5	Introduction à Project Reactor	3
5.1	Mono	3
5.2	Flux	3
6	Publisher interface	3
7	Génération de Flux	3
8	Opérateurs de Transformation	4
9	Opérateurs de Filtrage dans Project Reactor	4
10	Opérateurs de Combinaison de Flux	4
11	Interface Subscriber	5
12	Gestion des erreurs avec onErrorReturn .	5
12.1	Gestion des erreurs avec onErrorResume	5
13	Gestion du Backpressure dans Project Reactor	5
14	ConnectableFlux - Flux Partagé et Contrôlé	6
15	Introduction aux tests réactifs avec StepVerifier	6
16	Débogage et log pour le code réactif	6

17	Points Clés à Retenir	6
----	-----------------------------	---

1 Introduction à la programmation réactive

• Évolution des systèmes:

- ▶ **Besoin croissant:** Les applications modernes nécessitent une gestion efficace des ressources et une haute disponibilité.
- ▶ **Charges variables:** Les systèmes doivent s'adapter dynamiquement aux variations de la charge utilisateur.
- ▶ **Événements en temps réel:** Capacité à traiter et à réagir aux événements dès qu'ils se produisent.
- ▶ **Scalabilité:** Les systèmes doivent pouvoir évoluer pour gérer un grand nombre d'utilisateurs et de données.
- ▶ **Efficacité:** Utilisation optimale des ressources pour maximiser les performances et réduire les coûts.

• Reactive Manifesto:

- ▶ Un document fondateur publié en 2013.
- ▶ Définit les principes de la programmation réactive pour construire des systèmes réactifs.

1.1 Définition de la programmation réactive

- **Programmation réactive:** Un paradigme de programmation orienté vers les flux de données et la propagation des changements.

- **Réactivité:** Capacité d'un système à réagir aux événements en temps réel.
- Les principes de la programmation réactive
 - **Responsive:** Le système doit répondre rapidement et de manière cohérente.
 - **Resilient:** Le système doit rester fonctionnel même en cas de défaillance.
 - **Elastic:** Le système doit s'adapter à la variation de la charge.
 - **Message Driven:** Le système doit utiliser des messages asynchrones pour garantir une communication efficace.

1.2 Comparaison avec la programmation impérative et asynchrone

- **Programmation impérative:** Basée sur des instructions séquentielles, moins adaptée aux systèmes réactifs.
- **Programmation asynchrone:** Utilise des callbacks, des promesses ou des futures pour gérer les opérations non bloquantes.
- **Programmation réactive:** Étend la programmation asynchrone avec des flux de données et une gestion plus fine des événements.

2 Des API réactives dans différents langages

- **Java:**
 - Project Reactor: Une bibliothèque pour la programmation réactive basée sur la Reactive Streams API.
 - RxJava: Une bibliothèque pour la programmation réactive basée sur le modèle des Observables.
- **JavaScript:**
 - RxJS: Une bibliothèque pour la programmation réactive en JavaScript, utilisée notamment avec Angular.

- Bacon.js: Une bibliothèque pour la programmation réactive fonctionnelle en JavaScript.

- **Python:**

- RxPY: Une implémentation de Reactive Extensions pour Python.
- Streamz: Une bibliothèque pour la manipulation de flux de données en temps réel.

- **C#:**

- Reactive Extensions (Rx.NET): Une bibliothèque pour la programmation réactive en .NET.
- System.Reactive: Une autre implémentation de Reactive Extensions pour .NET.

- **Kotlin:**

- Kotlin Flow: Une API pour la programmation réactive dans Kotlin, intégrée à Kotlin Coroutines.
- RxKotlin: Une extension de RxJava pour Kotlin.

3 Introduction à la Reactive Streams API

- **Reactive Streams API:**

- Une spécification standard pour le traitement des flux de données asynchrones avec gestion de la pression (backpressure).
- Conçue pour assurer une communication fluide et efficace entre les composants réactifs.

- **Objectifs:**

- **Interopérabilité:** Permettre l'interopérabilité entre différentes bibliothèques réactives, telles que Project Reactor, RxJava, et Akka Streams.
- **Gestion efficace des flux de données:** Assurer que les producteurs de données ne submergent pas les consommateurs, en régulant le flux de données pour éviter les surcharges et les blocages.
- **Asynchronisme:** Faciliter le traitement asynchrone des flux de données pour amé-

liorer les performances et la réactivité des applications.

- **Backpressure:** Introduire des mécanismes pour gérer la pression exercée par les producteurs de données sur les consommateurs, permettant ainsi une meilleure gestion des ressources et une prévention des surcharges.
- Comparaison avec Java Streams API:
 - Java Streams API: Conçue pour le traitement de collections de données en mémoire.
 - Reactive Streams API: Conçue pour le traitement de flux de données asynchrones avec gestion de la pression.
 - Push vs Pull: Java Streams utilise un modèle pull, tandis que Reactive Streams utilise un modèle push avec gestion de la pression.

4 Exemple de Reactive Streams API en Java avec Project Reactor

5 Introduction à Project Reactor

- **Présentation de Project Reactor:**
 - Project Reactor est une bibliothèque pour la programmation réactive en Java, basée sur la Reactive Streams API.
 - Elle permet de créer des applications réactives, non bloquantes et hautement performantes.
- **Installation et configuration de Reactor dans un projet Java:**
 - Ajouter la dépendance Maven suivante à votre fichier `pom.xml`:

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

```
<artifactId>reactor-core</artifactId>
  <version>3.7.0</version>
</dependency>
```

5.1 Mono

- représente une séquence asynchrone qui émet zéro ou un élément.
- Utilisé pour les opérations qui retournent un seul résultat ou aucun.

5.2 Flux

- Représente une séquence asynchrone qui émet zéro, un ou plusieurs éléments.
- Utilisé pour les opérations qui retournent plusieurs résultats.

6 Publisher interface

- Interface fondamentale de Reactive Streams
- Source de données qui émet des éléments de manière asynchrone
- Point de départ du flux de données réactif
- `Mono` et `Flux` implémentent l'interface `Publisher`

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

7 Génération de Flux

- Création depuis des Données
 - **Valeurs Discrètes**
 - `Flux.just(1, 2, 3)`
 - `Flux.fromArray(array)`
 - `Flux.fromIterable(list)`
 - `Flux.fromStream(stream)`
 - **Génération Programmée**
 - `Flux.range(1, 10)` : Séquence de nombres
 - `Flux.interval(Duration)` : Émission périodique

- `Flux.generate()` : Génération synchrone
- `Flux.create()` : Génération avec backpressure
- Sources Asynchrones
 - **Temps**
 - `Flux.interval(Duration.ofSeconds(1))`
 - `Flux.timer(Duration.ofMillis(500))`
 - **Callbacks**
 - `Flux.create(sink -> {...})`
 - `Flux.push(sink -> {...})`
 - **CompletableFuture**
 - `Mono.fromFuture(future)`
 - `Flux.fromCompletionStage(stage)`

8 Opérateurs de Transformation

1. **map**: Transformation synchrone élément par élément
 - Conversion de types
 - Formatage de données
 - Calculs simples
2. **flatMap**: Transformation asynchrone avec aplatissement
 - Appels services externes
 - Opérations I/O
 - Flux imbriqués
3. **transform**: Modification globale du flux
 - Composition de transformations
 - Réutilisation de logique
 - Modification de la chaîne
4. **switchMap**: Gestion des changements rapides
 - Annulation automatique
 - Dernier flux uniquement
 - Recherche interactive

9 Opérateurs de Filtrage dans Project Reactor

Filtrage Simple et Déduplication - `filter` : Filtre selon prédicat - `filter(x -> x > 0)`
 - `filter(String::isEmpty)`

- `distinct` : Élimine doublons
 - Utilise equals/hashCode
 - Option keySelector
- `distinctUntilChanged`
 - Élimine doublons consécutifs
 - Compare éléments adjacents

Limitation - `take(n)` : n premiers éléments - `takeLast(n)` : n derniers - `takeWhile` : condition vraie - `takeUntil` : jusqu'à condition

Saut d'Éléments - `skip(n)` : Ignore n premiers - `skipLast(n)` : Ignore n derniers - `skipWhile` : Ignore si vrai - `skipUntil` : Ignore jusqu'à vrai

Sélection Spécifique - `elementAt` : Position précise - `next` : Premier élément - `last` : Dernier élément - `single` : Vérifie unicité

10 Opérateurs de Combinaison de Flux

Fusion de Flux

- `merge` : Combine en intercalant
 - Ordre d'arrivée préservé
 - Pas de synchronisation
 - Parallèle
- `concat` : Concatène séquentiellement
 - Ordre strict respecté
 - Séquentiel
 - Sans entrelacement
- `mergeSequential` : Fusion ordonnée
 - Souscription parallèle
 - Émission ordonnée
 - Préserve ordre

Combinaison Synchronisée

- `zip` : Combine par paires
 - Attend les deux sources
 - Transformation possible
 - 1:1 matching
- `combineLatest` : Dernières valeurs
 - Combine valeurs récentes
 - Réactif aux changements
 - N:M possible

Utilitaires

- `withLatestFrom` : Enrichissement
- `startWith` : Valeurs initiales
- `sample` : Échantillonnage

11 Interface Subscriber

- Interface fondamentale de Reactive Streams
- Consommateur de données qui reçoit les éléments de manière asynchrone
- Point de terminaison du flux de données réactif

```
public interface Subscriber<T> {
    // Called first when subscribing
    // to a Publisher
    // Provides the Subscription
    // object to control the flow
    // Should call
    // subscription.request() to start
    // receiving items
    void onSubscribe(Subscription s);

    // Called for each item emitted
    // by the Publisher
    // Can be called 0 to N times
    // based on the number of items
    // Will not be called after onError
    // or onComplete
    void onNext(T t);

    // Called when an error occurs in
    // the stream
    // Terminal operation - no more
```

```
calls will happen after this
    // Must not be called after
    // onComplete
    void onError(Throwable t);

    // Called when the Publisher has
    // no more items to emit
    // Terminal operation - no more
    // calls will happen after this
    // Must not be called after onError
    void onComplete();
}
```

12 Gestion des erreurs avec `onErrorReturn`

12.1 Gestion des erreurs avec `onErrorResume`

13 Gestion du Backpressure dans Project Reactor

Concept du Backpressure

- Régulation du flux producteur/consommateur
- Évite la surcharge du consommateur
- Communication bidirectionnelle

Contrôle par le Subscriber

- `request(n)`: Demande n éléments
- `request(Long.MAX_VALUE)`: Flux illimité
- `cancel()`: Arrêt immédiat

Stratégies de Gestion

- `buffer()`
 - Stockage temporaire
 - Mémoire tampon configurable
 - Préserve les données
- `drop()`
 - Suppression des excédents
 - Sans impact mémoire

- Perte acceptée
- `latest()`
 - Garde valeur récente
 - Mise à jour continue
 - Sampling
- `error()`
 - Signalement erreur
 - Protection système
 - Circuit breaker

14 ConnectableFlux - Flux Partagé et Contrôlé

Concept et Utilisation

- Variante de `Flux` avec émission contrôlée
- Partage un flux entre multiples abonnés
- « Hot » publisher - émission indépendante des abonnements

Méthodes de Contrôle

- `connect()`: Démarre l'émission
- `autoConnect(n)`: Connexion après n abonnés
- `refCount(n)`: Gestion automatique connexion/déconnexion

Patterns d'Utilisation

- `publish()`: Crée un `ConnectableFlux`
- `replay()`: Met en cache les éléments
- `share()`: Combine `publish` et `refCount`

Cas d'Usage

- Diffusion d'événements
- Partage de données en temps réel
- Cache partagé
- Multicast de flux

15 Introduction aux tests réactifs avec StepVerifier

Principes Fondamentaux - Test déclaratif des flux réactifs - Vérification pas à pas du

comportement - Support du backpressure - Tests synchrones et asynchrones

Méthodes Principales - `expectNext()`: Vérifie valeur suivante - `expectNextCount()`: Nombre d'éléments - `expectComplete()`: Vérifie fin normale - `expectError()`: Vérifie erreur - `verifyTimeout()`: Test avec timeout - ajouter la dépendance suivante à votre fichier `pom.xml`:

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-test</artifactId>
  <version>3.7.0</version>
  <scope>test</scope>
</dependency>
```

16 Débogage et log pour le code réactif

Opérateur `log()` - Trace tous les événements du flux - Signaux: `onNext`, `onError`, `onComplete` - Requêtes backpressure - Options de configuration: - Niveau de log - Logger personnalisé - Catégorie

Points de Contrôle - `checkpoint()` - Capture stack trace - Identifie source d'erreurs - Impact performance minimal

17 Points Clés à Retenir

1. **Évolution des Paradigmes**
 - Du synchrone à l'asynchrone
 - Des threads aux flux réactifs
 - Performance et scalabilité
2. **Approche Moderne**
 - Non-blocking par défaut
 - Gestion des ressources optimisée
 - Tests et monitoring intégrés

3. **Bonnes Pratiques**

- Gestion des erreurs systématique
- Contrôle du backpressure
- Découpage en flux atomiques
- Monitoring continu