UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Spring Webflux

Sommaire

- Introduction à Spring WebFlux
- REST classique VS une application WebFlux
- Project Reactor
- API REST réactives
- Gestion des bases de données réactives
- Gestion des erreurs et sécurité réactive
- Tests et optimisation des performances

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction à Spring WebFlux

Introduction à Spring WebFlux

Qu'est-ce que Spring WebFlux ?

Spring WebFlux est un framework réactif de Spring qui permet de construire des applications web asynchrones et non bloquantes. Il repose sur le standard **Reactive Streams**, introduit avec **Spring 5**.

- **Objectif principal :** Optimiser l'utilisation des ressources en gérant efficacement les opérations d'E/S, indépendamment du nombre de requêtes concurrentes.

Introduction à Spring WebFlux

- **Pourquoi l'utiliser ?**
  - Scalabilité accrue grâce au modèle non bloquant.
  - Idéal pour des applications modernes comme les API REST, le streaming temps réel et les microservices.

Introduction à Spring WebFlux

Mots-clés importants :

- **Non bloquant :** Pas d'attente active pour qu'une ressource soit disponible.
- **Asynchrone :** Les opérations se produisent en parallèle, sans dépendance immédiate à un thread.
- **Réactif :** Le système répond de manière efficace à un grand nombre de requêtes.

Différences entre Spring MVC et Spring WebFlux

Critère	Spring MVC	Spring WebFlux
Modèle	Bloquant (basé sur Servlet)	Non bloquant (basé sur Reactive Streams)
Serveur	Tomcat, Jetty (bloquant)	Netty, Undertow (non bloquant)
Performance	Limité par les threads	Scalabilité grâce au modèle non bloquant
Flux de données	Objets Java classiques	Flux réactifs (Mono, Flux)

Concepts clés de Spring WebFlux 1/2

1. **Reactive Streams API :** Un standard permettant de travailler avec des flux de données asynchrones.
   - **Objectif :** Simplifier la communication entre un producteur (génère des données) et un consommateur (traite ces données).
2. **Mono et Flux :**
   - **Mono :** Représente une donnée unique ou aucune donnée (équivalent de `Optional` ou `Future`).
   - **Flux :** Représente plusieurs éléments (équivalent d'un `List` réactif).

Concepts clés de Spring WebFlux 2/2

3. **Backpressure :**
   Gestion de la vitesse entre un producteur et un consommateur pour éviter une surcharge.

4. **Serveurs réactifs :**
   Spring WebFlux s'appuie sur des serveurs comme **Netty** ou **Undertow**, qui permettent des communications non bloquantes.

Cas d'utilisation concrets

Spring WebFlux est adapté à des cas d'utilisation où la scalabilité et la réactivité sont cruciales :

- **Applications API REST à forte charge :** Répondre à des milliers de requêtes simultanément.
- **Services temps réel :** Notamment avec WebSocket pour les notifications en direct.
- **Streaming de données :** Exemple, applications de trading ou systèmes IoT.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

REST classique

VS

une application WebFlux

Différence concrète entre une application REST classique et une application WebFlux

1. Modèle bloquant vs non bloquant

- **REST classique (Spring MVC) :**
  - Utilise un modèle **bloquant**.
  - Chaque requête est traitée par un thread du pool de threads. Si une opération (comme une requête à la base de données) est lente, le thread reste bloqué jusqu'à ce que la réponse soit prête.
  - Limité par le nombre de threads disponibles : plus il y a de requêtes simultanées, plus le serveur peut être ralenti.

1. Modèle bloquant vs non bloquant

- **Spring WebFlux :**
  - Utilise un modèle **non bloquant**.
  - Les threads ne sont jamais bloqués en attente d'une opération lente (comme un appel à une base de données ou à un autre service).
  - Permet de gérer un grand nombre de requêtes simultanées, même avec un nombre limité de threads, en optimisant l'utilisation des ressources.

1. Modèle bloquant vs non bloquant

Exemple d'intérêt :

Imaginez un service recevant 1000 requêtes simultanées, dont chacune implique une requête lente à une API externe :

- En REST classique : Le serveur pourrait s'essouffler rapidement, car chaque requête bloque un thread.
- En WebFlux : Les threads peuvent être réutilisés pendant que l'application attend la réponse des opérations lentes, permettant une meilleure scalabilité.

2. Traitement progressif des données

- **REST classique :**
  - Les données sont collectées, transformées, puis retournées en une seule fois.
  - Le client ne reçoit rien tant que la réponse complète n’est pas prête.

2. Traitement progressif des données

- **Spring WebFlux :**
  - Les données sont envoyées **au fur et à mesure qu'elles sont prêtes** (streaming).
  - Cela permet de commencer à traiter les données côté client avant que tout le flux ne soit envoyé.

2. Traitement progressif des données

Exemple d'intérêt :

Imaginez un service REST qui retourne 10 000 lignes d'une base de données.

- En REST classique : Le client doit attendre que les 10 000 lignes soient chargées en mémoire et envoyées d'un coup.
- En WebFlux : Les données peuvent être **streamées ligne par ligne**, permettant au client de commencer à traiter immédiatement.

4. Flux infinis ou temps réel

- **REST classique :**
  - Ne gère pas bien les flux infinis ou les mises à jour temps réel.
  - Généralement, des solutions comme WebSocket ou Server-Sent Events nécessitent un ajout complexe au-dessus de Spring MVC.

4. Flux infinis ou temps réel

- **Spring WebFlux :**
  - Conçu pour gérer des **flux de données continus**, comme des notifications en temps réel ou des mises à jour IoT.
  - Intègre nativement des mécanismes comme **Server-Sent Events (SSE)** ou **WebSocket**.

4. Flux infinis ou temps réel

Exemple d'intérêt :

Imaginez un tableau de bord de trading qui reçoit des mises à jour en temps réel sur les cours des actions.

- En REST classique : Nécessite une architecture complexe pour simuler un comportement en temps réel.
- En WebFlux : Les flux réactifs peuvent envoyer les mises à jour en continu via SSE ou WebSocket.

Intérêts pratiques de Spring WebFlux

Aspect	REST classique (Spring MVC)	Spring WebFlux
Scalabilité	Limitée par les threads	Haute scalabilité avec peu de threads
Traitement des données	Tout ou rien (pas de streaming)	Streaming progressif des données
Opérations lentes	Threads bloqués	Threads non bloqués (modèle réactif)
Flux temps réel	Complexe à implémenter	Gestion native des flux et des données temps réel
Utilisation des ressources	Moins efficace sous forte charge	Optimisée pour une grande charge simultanée

Quand utiliser Spring WebFlux ?

- Si votre application gère **beaucoup de requêtes simultanées**.
- Si vos dépendances externes (base de données, API) sont parfois lentes.
- Si vous avez besoin de **streaming de données** ou de **flux continus** (comme WebSocket).
- Si vous développez une architecture basée sur des **microservices**.

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Project Reactor

Programmation réactive avec Project Reactor

Spring WebFlux repose sur **Project Reactor**, une bibliothèque Java qui implémente les **Reactive Streams**. Ce chapitre explore les bases de cette bibliothèque, notamment les concepts de **Mono** et **Flux**, ainsi que les opérateurs permettant de manipuler ces flux.

Introduction à Project Reactor

Project Reactor est une bibliothèque réactive pour Java qui suit les principes définis par la spécification **Reactive Streams**. Elle offre deux types principaux pour travailler avec des données réactives :

- **Mono** : Contient une donnée unique ou aucune.
- **Flux** : Contient plusieurs données (0, 1 ou plus).

Pourquoi Project Reactor ?

- Il permet de travailler de manière non bloquante et asynchrone.
- Il optimise l'utilisation des ressources pour des applications nécessitant une forte scalabilité.

Les types réactifs : Mono et Flux

Mono

- Utilisé lorsqu'il y a **une seule donnée à traiter**.
- Exemple typique : une réponse HTTP avec un seul objet JSON.

Exemples :

```
Mono<String> mono = Mono.just("Hello, Reactor!");
Mono<Integer> emptyMono = Mono.empty(); // Mono sans données
Mono<String> errorMono = Mono.error(new RuntimeException("An error occurred"));
```

Les types réactifs : Mono et Flux

Flux

- Utilisé lorsqu'il y a **plusieurs données**.
- Exemple typique : une réponse contenant une liste d'objets ou un flux continu.

Exemples :

```
Flux<String> flux = Flux.just("Data1", "Data2", "Data3");
Flux<Integer> rangeFlux = Flux.range(1, 5); // Flux de 1 à 5
Flux<String> emptyFlux = Flux.empty(); // Flux sans données
```

Opérations sur les flux

Project Reactor offre une variété d'opérateurs pour transformer, filtrer et combiner les flux.

1. Filtrage des données : `filter`

- Filtre les éléments selon une condition.

```
Flux<Integer> numbers = Flux.range(1, 10);
Flux<Integer> evenNumbers = numbers.filter(n -> n % 2 == 0);
// Résultat : 2, 4, 6, 8, 10
```

2. Transformation des données : `map` et `flatMap`

- **`map`** : Transforme chaque élément du flux.

```
Flux<Integer> numbers = Flux.just(1, 2, 3);
Flux<String> transformed = numbers.map(n -> "Number: " + n);
// Résultat : "Number: 1", "Number: 2", "Number: 3"
```

- **`flatMap`** : Transforme chaque élément en un nouveau flux.

```
Flux<String> words = Flux.just("Hello", "World");
Flux<String> letters = words.flatMap(word -> Flux.fromArray(word.split("")));
```

3. Combinaison de flux : `merge`, `concat`, `zip` 1/2

- `merge` : Combine plusieurs flux en parallèle.

```
Flux<String> flux1 = Flux.just("A", "B");
Flux<String> flux2 = Flux.just("C", "D");
Flux<String> merged = Flux.merge(flux1, flux2);
// Résultat : A, C, B, D (ordre non garanti)
```

- `concat` : Combine plusieurs flux séquentiellement.

```
Flux<String> concatenated = Flux.concat(flux1, flux2);
// Résultat : A, B, C, D
```

3. Combinaison de flux : `merge`, `concat`, `zip` 2/2

- **`zip`** : Combine les éléments des flux correspondants.

```
Flux<String> letters = Flux.just("A", "B");
Flux<Integer> numbers = Flux.just(1, 2);
Flux<String> zipped = Flux.zip(letters, numbers, (letter, number) -> letter + number);
// Résultat : "A1", "B2"
```

Gestion des erreurs

Les flux réactifs offrent des opérateurs pour gérer les erreurs de manière élégante.

1. Gestion basique avec `onErrorReturn`

- Retourne une valeur par défaut en cas d'erreur.

```
Mono<String> fallback = Mono.error(new RuntimeException("Error"))
                            .onErrorReturn("Default value");
// Résultat : "Default value"
```

Gestion des erreurs

2. Gestion avancée avec `onErrorResume`

- Fournit un flux de remplacement en cas d'erreur.

```
Mono<String> fallback = Mono.error(new RuntimeException("Error"))
                            .onErrorResume(e -> Mono.just("Recovered from error"));
// Résultat : "Recovered from error"
```

3. Gestion continue avec `onErrorContinue`

- Permet de continuer le traitement malgré une erreur.

```
Flux<Integer> numbers = Flux.just(1, 2, 0, 4)
                            .map(n -> 10 / n)
                            .onErrorContinue((e, value) -> System.out.println("Error with: " + value));
// Résultat : 10, 5, Error with: 0, 2
```

Exemple : Notification réactive

Pour illustrer l'utilisation de **Flux** et des opérateurs, imaginons un système de notifications qui envoie des messages aux utilisateurs.

```
Flux<String> notifications = Flux.just("Notification 1", "Notification 2", "Notification 3");

// Transformer les messages (ajout d'un timestamp)
Flux<String> transformed = notifications.map(msg -> "[" + System.currentTimeMillis() + "] " + msg);

// Simuler l'envoi (affichage dans la console)
transformed.subscribe(System.out::println);
```

Résumé :

- **Mono** et **Flux** sont les bases de Project Reactor.
- Les opérateurs comme `map`, `flatMap`, `filter`, et `merge` permettent de manipuler les flux.
- La gestion des erreurs est flexible avec des opérateurs comme `onErrorResume` et `onErrorContinue`.
- Project Reactor permet de travailler efficacement avec des flux de données asynchrones, offrant ainsi un environnement idéal pour des applications réactives.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

API REST réactives

Création d'API REST réactives

Spring WebFlux permet de créer des API REST réactives en utilisant un modèle non bloquant. Dans cette section, nous abordons les bases pour configurer des contrôleurs réactifs et traiter les requêtes HTTP de manière asynchrone.

Contrôleurs réactifs avec annotations

Spring WebFlux propose deux façons principales de créer des contrôleurs REST réactifs :

- **Avec des annotations (@RestController et @RequestMapping)** : similaire à Spring MVC, mais avec des types réactifs (Mono et Flux).
- **Avec un modèle fonctionnel (RouterFunction et HandlerFunction)** : une approche plus déclarative et fonctionnelle.

Contrôleurs réactifs avec annotations

Les contrôleurs annotés permettent de définir des endpoints en utilisant des annotations comme `@GetMapping` ou `@PostMapping`, mais les méthodes doivent retourner des types réactifs comme **Mono** et **Flux**.

Exemple : Endpoint GET

```java
@RestController
@RequestMapping("/api/items")
public class ItemController {

    @GetMapping // Déclare un endpoint accessible via une requête HTTP GET.
    public Flux<String> getAllItems() {
      //Retourne un `Flux` ou un `Mono`, permettant un traitement réactif des données.
        return Flux.just("Item1", "Item2", "Item3");
    }

    @GetMapping("/{id}")
    public Mono<String> getItemById(@PathVariable String id) {
        return Mono.just("Item " + id);
    }
}
```

Contrôleurs réactifs avec RouterFunction

Le modèle fonctionnel offre une approche déclarative pour définir les routes et leurs handlers.

Détails :

- `RouterFunctions.route` : Définit les routes et lie chaque route à un handler.
- `ServerRequest` et `ServerResponse` : Utilisés pour accéder aux informations de la requête et construire une réponse.

Exemple : RouterFunction

```
@Configuration
public class RouterConfig {

    @Bean
    public RouterFunction<ServerResponse> route(ItemHandler handler) {
        return RouterFunctions
                .route(GET("/api/items"), handler::getAllItems)
                .andRoute(GET("/api/items/{id}"), handler::getItemById);
    }
}
```

Exemple : HandlerFunction

```
@Component
public class ItemHandler {

    public Mono<ServerResponse> getAllItems(ServerRequest request) {
        Flux<String> items = Flux.just("Item1", "Item2", "Item3");
        return ServerResponse.ok().body(items, String.class);
    }

    public Mono<ServerResponse> getItemById(ServerRequest request) {
        String id = request.pathVariable("id");
        Mono<String> item = Mono.just("Item " + id);
        return ServerResponse.ok().body(item, String.class);
    }
}
```

Gestion des requêtes et réponses réactives

Dans Spring WebFlux, les requêtes et réponses sont traitées de manière réactive :

- **Requêtes HTTP :** Les données reçues (par exemple, JSON) sont converties automatiquement en objets Java grâce à des convertisseurs réactifs.
- **Réponses HTTP :** Les données renvoyées sont converties en JSON et envoyées au client.

Lecture de requêtes (POST/PUT)

Pour lire le corps de la requête, utilisez des types réactifs comme `Mono`.

Détails :

- `@RequestBody` : Lit les données du corps de la requête en tant que `Mono` ou `Flux`.
- Le traitement est asynchrone : les données sont transformées ou enregistrées sans bloquer les threads.

Exemple : Endpoint POST

```
@RestController
@RequestMapping("/api/items")
public class ItemController {

    @PostMapping
    public Mono<String> createItem(@RequestBody Mono<String> item) {
        return item.map(i -> "Created: " + i);
    }
}
```

Streaming de réponses

Spring WebFlux permet de streamer les réponses directement au client en utilisant **Flux**.

Détails :

- `Flux.interval` : Génère un flux d'éléments à des intervalles réguliers.
- Les données sont envoyées en continu au client.

Streaming de réponses

Exemple : Streaming avec Flux

```
@GetMapping("/stream")
public Flux<String> streamItems() {
    return Flux.interval(Duration.ofSeconds(1))
               .map(i -> "Item " + i);
}
```

Gestion des routes avec RouterFunction

Pourquoi utiliser le modèle fonctionnel ?

- Approche déclarative qui permet une gestion fine des routes.
- Idéal pour les microservices ou les APIs où la configuration des routes est essentielle.

Exemple complet avec RouterFunction

```
@Configuration
public class RouterConfig {

    @Bean
    public RouterFunction<ServerResponse> routes(ItemHandler handler) {
        return RouterFunctions
                .route(GET("/api/items"), handler::getAllItems)
                .andRoute(POST("/api/items"), handler::createItem);
    }
}
```

Exemple complet avec RouterFunction

Correspondant HandlerFunction :

```
@Component
public class ItemHandler {

    public Mono<ServerResponse> getAllItems(ServerRequest request) {
        Flux<String> items = Flux.just("Item1", "Item2", "Item3");
        return ServerResponse.ok().body(items, String.class);
    }

    public Mono<ServerResponse> createItem(ServerRequest request) {
        Mono<String> item = request.bodyToMono(String.class);
        return item.flatMap(i -> ServerResponse.ok().bodyValue("Created: " + i));
    }
}
```

Conclusion :

- Les contrôleurs réactifs peuvent être définis avec des annotations ou en utilisant le modèle fonctionnel.
- Les réponses peuvent être streamées (Flux) ou retournées individuellement (Mono).
- Spring WebFlux permet un traitement asynchrone efficace des requêtes et des réponses.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Gestion des bases de données réactives

Gestion des bases de données réactives

Spring WebFlux s'intègre parfaitement avec des bases de données réactives. Cette partie explore la configuration, l'utilisation et les bonnes pratiques pour interagir avec des bases de données de manière non bloquante.

Introduction aux bases de données réactives

Les bases de données réactives sont conçues pour prendre en charge les opérations non bloquantes, offrant une communication asynchrone et un meilleur support pour les applications réactives. Deux exemples populaires :

- **MongoDB réactif** : Adapté aux bases NoSQL.
- **R2DBC** : Une spécification réactive pour les bases de données relationnelles comme PostgreSQL, MySQL, etc.

Configuration de Spring Data Reactive avec MongoDB réactif

Étape 1 : Ajouter la dépendance

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

Étape 2 : Configurer MongoDB

Dans le fichier `application.yml` :

```
spring:
  data:
    mongodb:
      uri: mongodb://localhost:27017/mydatabase
```

Configuration de Spring Data Reactive avec R2DBC (bases relationnelles) 1/2

Étape 1 : Ajouter les dépendances

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>
</dependency>
<dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-postgresql</artifactId> <!-- ou un autre pilote compatible -->
</dependency>
```

Configuration de Spring Data Reactive avec R2DBC (bases relationnelles) 2/2

Étape 2 : Configurer R2DBC

Dans le fichier `application.yml` :

```
spring:
  r2dbc:
    url: r2dbc:postgresql://localhost:5432/mydatabase
    username: myuser
    password: mypassword
```

Création des entités et des repositories réactifs

1. Définir une entité exemple avec une entité pour un système de gestion de tâches :

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

@Table("tasks") // Utilisez @Document pour MongoDB
public class Task {

    @Id
    private String id;
    private String title;
    private boolean completed;

    // Getters et Setters
}
```

Création des entités et des repositories réactifs

2. Créer un repository réactif

Spring Data Reactive fournit des interfaces spécifiques pour les bases réactives.

Repository réactif pour MongoDB ou R2DBC :

```
import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import reactor.core.publisher.Flux;

public interface TaskRepository extends ReactiveCrudRepository<Task, String> {

    // Requêtes personnalisées réactives
    Flux<Task> findByCompleted(boolean completed);
}
```

Utilisation des repositories réactifs dans les services

Avec Spring WebFlux, les repositories réactifs peuvent être utilisés directement dans les services pour effectuer des opérations CRUD.

Gestion des transactions réactives

Pour les bases relationnelles (R2DBC), Spring fournit un support pour les transactions réactives.

Configurer les transactions réactives

Dans un service, les transactions réactives peuvent être configurées avec `TransactionalOperator`.

Exemple d’un service réactif :

```java
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class TaskService {

    private final TaskRepository repository;

    public TaskService(TaskRepository repository) {
        this.repository = repository;
    }

    public Flux<Task> getAllTasks() {
        return repository.findAll();
    }

    public Mono<Task> getTaskById(String id) {
        return repository.findById(id);
    }

    public Mono<Task> createTask(Task task) {
        return repository.save(task);
    }

    public Mono<Task> updateTask(String id, Task task) {
        return repository.findById(id)
                .flatMap(existingTask -> {
                    existingTask.setTitle(task.getTitle());
                    existingTask.setCompleted(task.isCompleted());
                    return repository.save(existingTask);
                });
    }

    public Mono<Void> deleteTask(String id) {
        return repository.deleteById(id);
    }
}
```

Exemple transactions réactives :

```java
import org.springframework.r2dbc.connection.TransactionalOperator;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;

@Service
public class TransactionalService {

    private final TaskRepository repository;
    private final TransactionalOperator transactionalOperator;

    public TransactionalService(TaskRepository repository, TransactionalOperator transactionalOperator) {
        this.repository = repository;
        this.transactionalOperator = transactionalOperator;
    }

    public Mono<Void> createTaskWithTransaction(Task task) {
        return repository.save(task)
                .then(repository.save(new Task("id2", "Another Task", false)))
                .as(transactionalOperator::transactional); // Déclare une transaction
    }
}
```

Résumé

- **MongoDB et R2DBC réactifs :** Configurés avec des dépendances spécifiques et connectés à Spring WebFlux.
- **Repositories réactifs :** Gèrent les opérations CRUD en mode réactif.
- **Transactions réactives :** Assurent une cohérence des données dans les bases relationnelles.

Gestion des erreurs et sécurité réactive

Gestion des erreurs et sécurité réactive

Dans une application Spring WebFlux, la gestion des erreurs et la sécurité doivent être adaptées au modèle réactif pour garantir une expérience utilisateur fluide et une sécurisation robuste.

Gestion des erreurs spécifiques :

Gestion des erreurs au niveau des flux

Dans Spring WebFlux, la gestion des erreurs se fait directement sur les flux réactifs à l'aide des opérateurs comme `onErrorResume`, `onErrorReturn`, ou `onErrorContinue`.

Gestion des erreurs au niveau des flux

Exemple : Gestion des erreurs dans un Flux

```
Flux<Integer> numbers = Flux.just(1, 2, 0, 4)
    .map(n -> 10 / n)
    .onErrorResume(e -> Flux.just(-1, -2, -3)); // Flux de remplacement en cas d'erreur
```

Exemple : Mono avec valeur par défaut en cas d’erreur

```
Mono<String> result = Mono.error(new RuntimeException("Error"))
    .onErrorReturn("Default value"); // Retourne une valeur par défaut
```

Gestion globale des erreurs avec WebExceptionHandler

Spring WebFlux permet de gérer globalement les exceptions grâce à une implémentation de l'interface `WebExceptionHandler`.

Exemple : Gestionnaire d'erreurs global

```java
@Component
public class GlobalErrorHandler implements ErrorWebExceptionHandler {

    @Override
    public Mono<Void> handle(ServerWebExchange exchange, Throwable ex) {
        ServerHttpRequest request = exchange.getRequest();

        // Journalisation de l'erreur
        System.err.println("Erreur interceptée : " + ex.getMessage() + " pour la requête " + request.getPath());

        // Définir un code HTTP
        exchange.getResponse().setStatusCode(HttpStatus.INTERNAL_SERVER_ERROR);

        // Construire une réponse JSON personnalisée
        String errorMessage = String.format(
                "{\"error\": \"%s\", \"path\": \"%s\"}",
                ex.getMessage(),
                request.getPath()
        );

        byte[] bytes = errorMessage.getBytes();

        // Retourner la réponse JSON
        return exchange.getResponse()
                .writeWith(Mono.just(exchange.getResponse().bufferFactory().wrap(bytes)));
    }
}
```

Personnalisation des réponses en cas d'erreur

Pour fournir une réponse JSON détaillée en cas d'erreur :

```
public Mono<ServerResponse> handleError(Throwable ex) {
    return ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR)
        .bodyValue(new ErrorResponse("INTERNAL_ERROR", ex.getMessage()));
}
```

Validation réactive des données

Spring WebFlux prend en charge la validation des données avec **Hibernate Validator** et l'annotation `@Valid`.

Exemple : Validation dans un contrôleur

```
@RestController
@RequestMapping("/tasks")
public class TaskController {

    @PostMapping
    public Mono<Task> createTask(@Valid @RequestBody Task task) {
        return taskService.createTask(task);
    }
}
```

Entité avec contraintes de validation :

```
import jakarta.validation.constraints.NotBlank;

public class Task {

    @NotBlank(message = "Title cannot be empty")
    private String title;
    private boolean completed;

    // Getters et Setters
}
```

Gestion des erreurs de validation :

Spring retourne automatiquement un code d'erreur **400 BAD REQUEST** si les contraintes ne sont pas respectées.

Introduction à Spring Security réactif

Spring Security réactif est adapté à la nature non bloquante de Spring WebFlux. Il permet :

- Une gestion réactive des utilisateurs et des permissions.
- L'utilisation de mécanismes comme JWT pour l'authentification.

1. Ajouter Spring Security au projet

Dépendance Maven :

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

Configuration de la sécurité réactive

Spring Security utilise un `SecurityWebFilterChain` pour configurer la sécurité.

Exemple : Configuration de base

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;
@Configuration
public class SecurityConfig {
    @Bean
    public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
        return http
                .authorizeExchange()
                .pathMatchers("/public/**").permitAll() // Routes publiques
                .anyExchange().authenticated()         // Tout le reste nécessite une authentification
                .and()
                .httpBasic() // Authentification HTTP Basic
                .and()
                .build();
    }
}
```

Utilisation des utilisateurs en mémoire

Pour les environnements de développement ou des tests rapides :

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.userdetails.MapReactiveUserDetailsService;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;

@Configuration
public class UserConfig {
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
                .username("user")
                .password("password")
                .roles("USER")
                .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```

Authentification avec JWT 1/2

Pour sécuriser une API REST avec JWT, Spring Security permet de valider et de décoder les jetons.

Exemple simplifié :

1. Ajouter la dépendance JWT :

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
```

Authentification avec JWT 2/2

2. Générer un JWT dans une méthode de login :

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

public String generateToken(String username) {
    return Jwts.builder()
        .setSubject(username)
        .signWith(SignatureAlgorithm.HS512, "secretKey")
        .compact();
}
```

3. Valider le JWT dans un filtre de sécurité.

Résumé

- **Gestion des erreurs réactives :** Utilisation de `onErrorResume`, `WebExceptionHandler`, ou des gestionnaires globaux.
- **Validation réactive :** Validation des entrées utilisateur avec Bean Validation.
- **Sécurité réactive :** Spring Security réactif permet une sécurisation adaptée aux applications non bloquantes, avec prise en charge de mécanismes comme JWT et HTTP Basic.

Tests et optimisation des performances

Tests et optimisation des performances

Les tests et l'optimisation des performances sont essentiels pour garantir la qualité et l'efficacité des applications Spring WebFlux. Cette section détaille les bonnes pratiques pour tester des applications réactives et optimiser leur comportement.

Tester des applications réactives nécessite des outils adaptés pour vérifier le comportement des flux de données asynchrones.

Tests unitaires avec StepVerifier

`StepVerifier` est un outil de **Project Reactor** utilisé pour vérifier le comportement des flux réactifs.**Exemple : Tester un Mono**

```
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Mono;
import reactor.test.StepVerifier;

public class MonoTest {

    @Test
    public void testMono() {
        Mono<String> mono = Mono.just("Hello, WebFlux!");

        StepVerifier.create(mono)
            .expectNext("Hello, WebFlux!")
            .verifyComplete();
    }
}
```

Exemple : Tester un Flux

```
@Test
public void testFlux() {
    Flux<String> flux = Flux.just("A", "B", "C");

    StepVerifier.create(flux)
        .expectNext("A")
        .expectNext("B")
        .expectNext("C")
        .verifyComplete();
}
```

Tests des contrôleurs avec WebTestClient

`WebTestClient` est une classe de test intégrée à Spring WebFlux pour tester les contrôleurs et les endpoints.**Exemple : Tester un endpoint GET**

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.test.web.reactive.server.WebTestClient;

@WebFluxTest
public class ControllerTest {
    @Autowired
    private WebTestClient webTestClient;
    @Test
    public void testGetAllItems() {
        webTestClient.get()
            .uri("/api/items")
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$[0]").isEqualTo("Item1");
    }
}
```

Tests d'intégration réactifs

Les tests d'intégration permettent de valider le fonctionnement de l'ensemble de l'application, y compris les bases de données réactives.**Exemple : Test avec MongoDB réactif**

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import reactor.test.StepVerifier;

@DataMongoTest
public class RepositoryTest {

    @Autowired
    private TaskRepository repository;

    @Test
    public void testSaveAndFind() {
        Task task = new Task(null, "Task1", false);

        repository.save(task)
            .thenMany(repository.findAll())
            .as(StepVerifier::create)
            .expectNextMatches(t -> t.getTitle().equals("Task1"))
            .verifyComplete();
    }
}
```

Introduction à la backpressure

La **backpressure** est un mécanisme qui contrôle le flux de données entre un producteur et un consommateur pour éviter une surcharge.

Gestion native de la backpressure dans Reactor

- Lorsque le consommateur ne peut pas traiter les données assez vite, Reactor peut ralentir ou interrompre temporairement le producteur.

Introduction à la backpressure

Exemple : Limiter la vitesse des données

```
Flux<Long> flux = Flux.interval(Duration.ofMillis(10))
    .onBackpressureDrop() // Ignore les éléments excédentaires
    .doOnNext(System.out::println)
    .take(10);
```

Bonnes pratiques pour optimiser les performances

1. Utiliser les bons Schedulers

Les Schedulers permettent de contrôler sur quel thread les tâches réactives s'exécutent.

- **Parallel Scheduler :** Utilisé pour les tâches parallèles.
- **Elastic Scheduler :** Pour des opérations bloquantes comme des appels à des API externes.

Bonnes pratiques pour optimiser les performances

Exemple : Utiliser un Scheduler pour un traitement parallèle

```
Flux.range(1, 10)
    .parallel()
    .runOn(Schedulers.parallel())
    .map(i -> i * 2)
    .sequential()
    .subscribe(System.out::println);
```

Bonnes pratiques pour optimiser les performances

2. Réduire les conversions bloquantes

- Évitez d'utiliser des appels bloquants (`Thread.sleep`, `blockingQueue.take`, etc.) dans une application réactive.
- Si nécessaire, encapsulez-les dans un `Scheduler`.

3. Optimisation des ressources

- **Taille du pool de threads :** Assurez-vous que le pool de threads est configuré pour répondre à la charge.
- **Netty tuning :** Si vous utilisez Netty, vous pouvez ajuster les paramètres pour gérer davantage de connexions simultanées.

Bonnes pratiques pour optimiser les performances

4. Surveillance des performances

Utilisez des outils comme :

- **Spring Actuator :** Ajoutez des métriques pour surveiller l'état de l'application.
- **Micrometer et Prometheus :** Collectez des métriques pour analyser les performances réactives.

Bonnes pratiques pour optimiser les performances

Exemple : Ajout d'Actuator

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

Dans `application.yml` :

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

Résumé

- **Tests réactifs :** Utilisation de `StepVerifier` pour les flux et de `WebTestClient` pour les contrôleurs.
- **Backpressure :** Contrôle du flux de données pour éviter la surcharge.
- **Optimisation des performances :** Utilisation efficace des Schedulers, réduction des conversions bloquantes, et surveillance des métriques.

Merci pour votre attention

Des questions ?