POLITECNICO DI TORINO

Functional Reactive Web Programming with Spring WebFlux

Cadidata: Elena Civera

Relatore: Giovanni Malnati

Corso di Laurea Magistrale in Ingegneria Informatica

Specializzazione Reti

Dicembre 2018

A te,
Amore mio,
che mi hai dato la forza per realizzare questo bellissimo sogno.

Table of contents

1

Introduzione

In un ambiente moderno in cui “alle persone non piace aspettare” è importante avere applicazioni che siano interattive, cioè sistemi che reagiscano in modo reattivo e quindi con un tempo di latenza minimo.

Il lavoro di questa tesi è quello di studiare il concetto di “reactive” così come è stato definito dallo standard ReactiveStream e di applicarlo ad un’architettura che lo supporti, al fine di confrontare le prestazioni di due applicazioni identiche, dove una supporta la programmazione reattiva, mentre l’altra no.

Nel secondo capitolo si definisce quindi cosa si intende con reactive. Dietro di essa infatti si nascondono la programmazione asincrona e concorrente, che permetta al sistema di non bloccarsi, gestendo quindi al meglio le risorse e fornendo già i primi risultati all’utente quando ancora non tutto il risultato è definitivamente pronto. Questo è fonte di guadagno sotto molti aspetti: meno spreco di risorse, bassi tempi di latenza e si evita quindi che l’utente cominci a visualizzare altri siti web nell’attesa del caricamento dell’intera pagina. Maggiore è la reattività del sito, maggiore sarà il tempo di permanenza dell’utente su di esso.

Nel terzo capitolo, si sperimentano i concetti sopra definiti applicandoli al framework Spring WebFlux. In particolare ci si focalizza sulla struttura del framework e sugli operatori principali definiti dalla libreria Reactor, ovvero la libreria scelta da Spring per il framework WebFlux.

Nel quarto capitolo testo due applicazioni completamente identiche, con la sola ed unica differenza che una applica il concetto di reactive, mentre l’altra no. La prima è realizzata con Spring WebFlux, mentre la seconda con Spring MVC. Quest’ultima è stata scelta perché ha un’architettura quasi identica alla prima, infatti Spring definisce MVC e WebFlux come due stack protocollari alternativi, dove uno non sostituisce l’altro, quindi completamente compatibili. Esse verranno testate mediante l’utilizzo di due Tool: Gatling e Jmeter al fine di riuscire effettivamente a dimostrare quanto teoricamente detto.

In conclusione, presenterò i vantaggi che ne derivano dalla programmazione reattiva. Specificando quando e perché sia utile approcciarsi ad essa.

2

Il pattern di programmazione reattiva

2.1 Il problema tipico della programmazione sincrona

Un'applicazione scritta attraverso l'uso di sole chiamate sincrone è sicuramente più facile da implementare ma rende l'interfaccia utente bloccante.

Consideriamo un esempio tipico, di un'applicazione web che riceve richieste da utenti e deve potergli fornire una risposta:

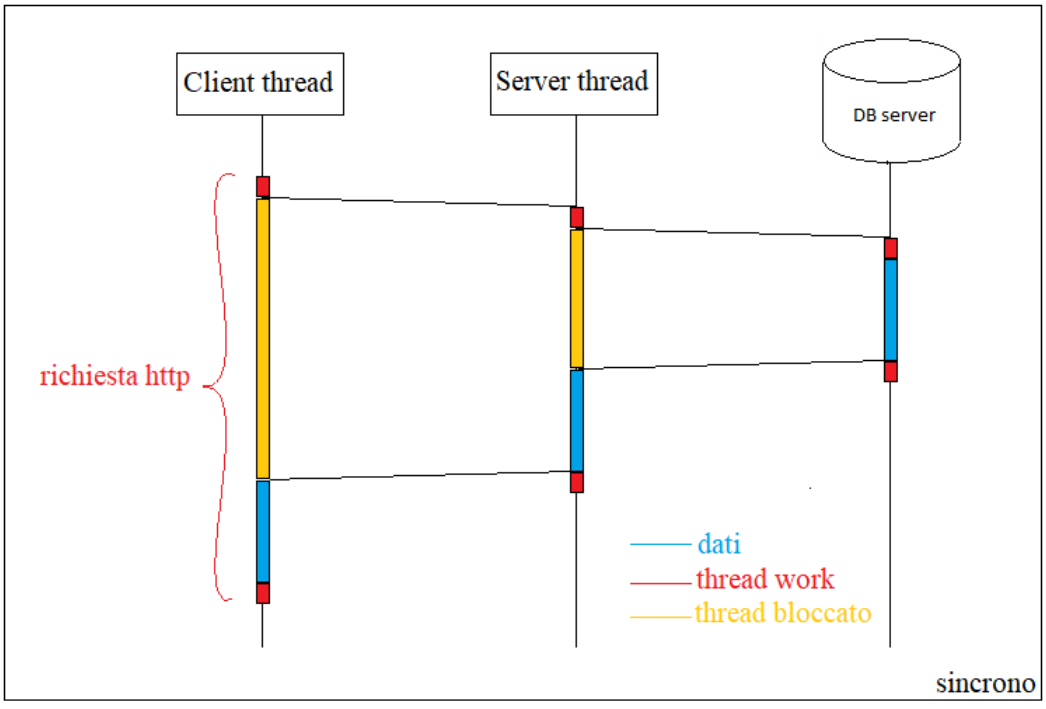


Figure 2.1: Tipiche chiamate sincrone

Avviene quindi che:

1. L'utente invia una richiesta HTTP alla nostra applicazione per ottenere una certa quantità di dati, rimanendo bloccato in attesa della risposta
2. La nostra applicazione attiva un thread, il quale anch'esso si blocca in attesa che il DB server risponda
3. Il thread che gestisce il DB recupera quindi i dati e li fornisce al thread server
4. Il thread server deve attendere di ricevere tutti i dati prima di poterli inviare al client e allo stesso modo il client deve attendere di riceverli tutti per farli vedere all'utente.

Ogni chiamata è sincrona, quindi i thread sono bloccati nell'attesa di ricevere risposta. Possiamo quindi facilmente notare che una risorsa (thread) non può essere più utilizzata da nessun altro client finché non ottiene la risposta dal metodo chiamato (oppure se scade un timeout) nonostante il thread non stia lavorando. Questo è sicuramente uno spreco di risorse.

2.2 La programmazione asincrona

Con la programmazione asincrona i thread non rimangono bloccati e sono liberi di svolgere altro nell'attesa di ricevere la risposta dal servizio chiamato, e quando la riceveranno, una call-back verrà invocata per rispondere al chiamante.

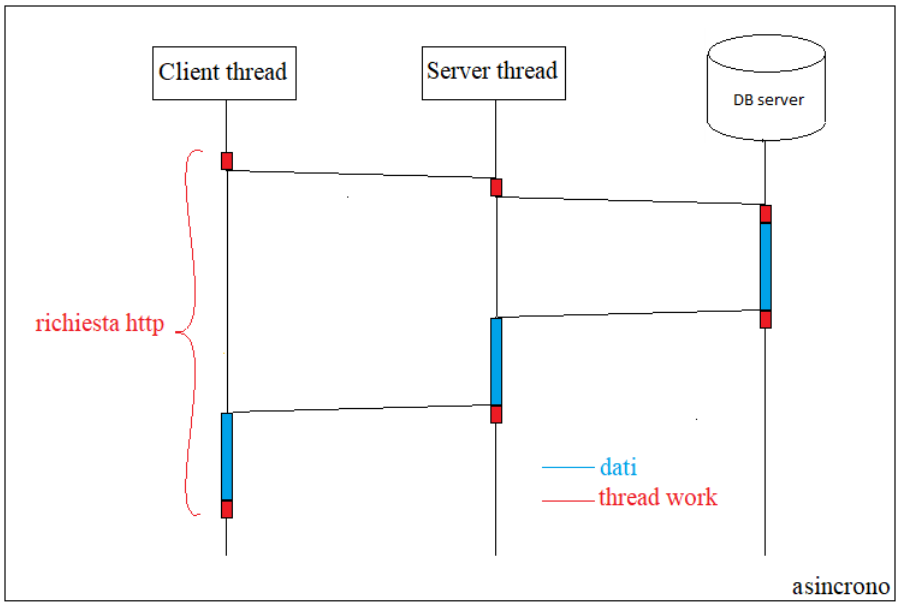


Figure 2.2: Tipiche chiamate asincrone

Con la programmazione asincrona possiamo notare che:

- è cambiato il modo con cui le risorse (thread in particolare) vengono utilizzate, permettendo quindi una gestione contemporanea di un numero di utenti maggiore. Infatti in framework come Spring MVC, che di base è sincrono, viene attivato un thread per ogni richiesta http, mentre in WebFlux (asincorno), un thread può gestire anche più di una richiesta http nell'attesa di una risposta del servizio chiamato.
- il tempo complessivo richiesto per rispondere all'utente in realtà è leggermente aumentato perchè il cambio di contesto di un thread non è gratis, ma è praticamente trascurabile rispetto alla durata dell'intera richista http.

Questa ottimizzazione di risorse costa al programmatore uno sforzo maggiore nella scrittura del codice perché sarà appunto più complesso a causa di:

- librerie che non rendono facile la scrittura per l'invocazione di chiamate asincrone. Considerando il linguaggio Java, che è quello che ho utilizzato per questa tesi, esistono alcuni classi che permettono di scrivere codice asincrono ma che in alcuni casi particolarmente complessi diventa quasi impossibile da gestire e manutenere nel tempo (Sezione 3.6)
- problematiche derivate dalla programmazione concorrente e quindi: deadlock, accesso a risorse condivise, problemi tipici di comunicazione tra un produttore troppo veloce e consumatore lento, e così via....

Nonostante queste difficoltà, la programmazione concorrente di tipo asincrono è però fondamentale per la creazione di un applicazione molto reattiva.

2.3 La programmazione reattiva

L'idea della reattività nasce dalla necessità di osservare il cambiamento di un flusso asincrono di elementi che variano nel tempo e quindi di poter reagire attraverso delle azioni nell'esatto istante di tempo in cui essi cambiano.

Questo fa si che il programmatore abbia a disposizione delle variabili il cui valore varia nel tempo al variare di altri componenti senza che questi siano strettamente legati. Si immagini una variabile che resti legata alla posizione del mouse sullo schermo. Appena il mouse si sposta, il valore delle variabili cambia con esso dinamicamente.

In un sistema reattivo abbiamo quattro caratteristiche fondamentali:

- Concorrenza: un sistema concorrente è necessario per poter rispondere agli eventi che si generano in modo asincrono nel tempo.
- Evoluzione dello stato nel tempo: un sistema reattivo ha componenti il cui valore dipende dal tempo
- Inversione del controllo: questo permette di evitare l'accoppiamento stretto tra gli oggetto rendendoli il più possibile l'uno indipendente dall'altro. Questo è ottenuto evitando l'invocazione diretta dell'oggetto, ma utilizzando gli eventi e quindi l'invocazione di call-back quando questi si verificano.

Si immagini un interruttore collegato a una lampadina. Chi è che comanda la lampadina affinchè essa si accenda o spenga?

- In un sistema non reattivo abbiamo la lampadina che viene controllata direttamente dallo switch. Quando lo switch cambia stato, lui stesso si preoccuperà di dare corrente alla lampadina, la quale si accenderà. Quindi l'oggetto "Switch" avrà al suo interno un oggetto "Lampadina" su cui invocherà un metodo diretto per dargli corrente.

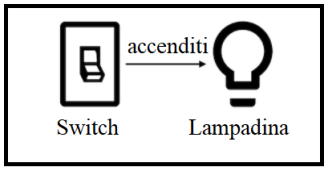


Figure 2.3: Componenti non reattivi

- In un sistema reattivo, lo switch e la lampadina sono due oggetti indipendenti, il primo emetterà un evento comunicando il proprio cambiamento di stato, il secondo resta in ascolto di eventi e quindi si illumina di conseguenza. Quindi l'oggetto "Switch" invocherà un evento "attivazione corrente elettrica" che, quando la lampadina lo ascolterà invocherà lei stessa una call-back per fornirsi luce. Quindi la lampadina non è comandata dallo switch, ma è in grado di autogestirsi.

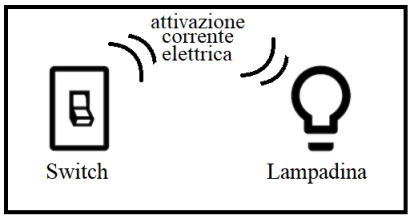


Figure 2.4: I due componenti sono reattivi

Tornando a considerare l'esempio di un applicazione web che riceve richieste http dai client, come si comporterebbero il client e il server se fossero reattivi?
Si avrebbe questa situazione:

- I thread non sono bloccati
- Non appena il DB server riesce a leggere il primo dei tanti item da inviare al client lo emette come se fosse un evento che viene immediatamente ricevuto dal server thread, il quale anch'esso lo emette verso il client
- Il Client thread riceve quasi immediatamente già il primo evento e lo mostra all'utente attraverso l'interfaccia grafica

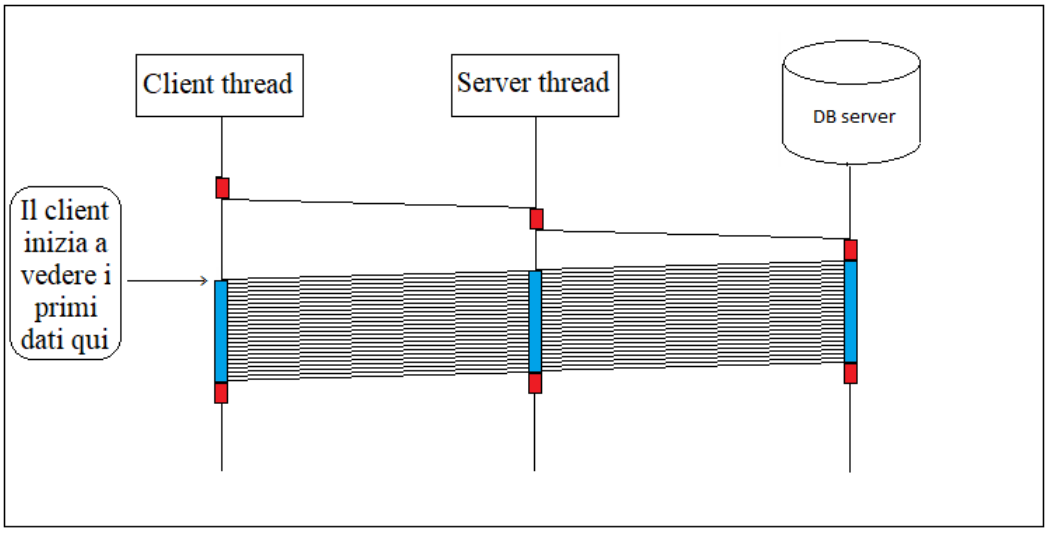


Figure 2.5: Client e server reattivi

Il risultato che si ottiene è straordinario: il client permette all'utente di leggere una grossa quantità di dati con una latenza talmente piccola che l'utente tipicamente non è in grado di percepire. E mentre l'utente è impegnato a leggere i primi dati ottenuti, intanto il client mostra anche gli altri dati che sta ricevendo.

Se la mole di dati è enorme e le operazioni nel server non sono bloccanti, oppure se vi sono molti servizi di back-end da contattare uno in cascata all'altro, la programmazione reattiva mostra davvero dei risultati straordinari, impossibili da ottenere attraverso l'uso di codice sicrono, ottenendo:

- Tempo di latenza minimi
- Tempo complessivo di invio della risposta non è pari alla somma dei tempi di ogni singola operazione (questo lo sarebbe con la programmazione sincrona) ma sarebbe pari al tempo richiesto dall'operazione più lunga più un piccolo delta di elaborazione e trasmissione della richiesta http.

Affinchè un sistema sia reattivo, è necessario che esso utilizzi 4 concetti fodamentali:

- Eventi asincroni: i componenti devono comunicare fra loro attraverso eventi asincroni
- Flow API: con java 9 viene introdotto la libreria java.util.concurrent.Flow che definisce le API per il controllo del flusso tra due o più componenti
- Back pressure non bloccante: è la strategia che permette a un ascoltatore (quindi consumatore di eventi) di non essere mai sovraccaricato da un eccessiva produzione di eventi emessi dal produttore. Sarà proprio con l'operazione di "request" (API definita dalle specifiche Reactive Stream) che verrà garantita la back-pressure.
- Programmazione funzionale: adottando uno stile di programmazione basato su funzioni prive di effetti collaterali è possibile evitare di condividere stati e rendere le operazioni componibili portando a un approccio più dichiarativo che facilita il riuso e consente ottimizzazioni anche in fase di compilazione.

2.4 Reactive Manifesto

Quali caratteristiche architetturali deve avere un sistema per essere considerato reattivo? Nel "Reactive Manifesto" è stato definito che un sistema reattivo deve necessariamente avere la seguente architettura:

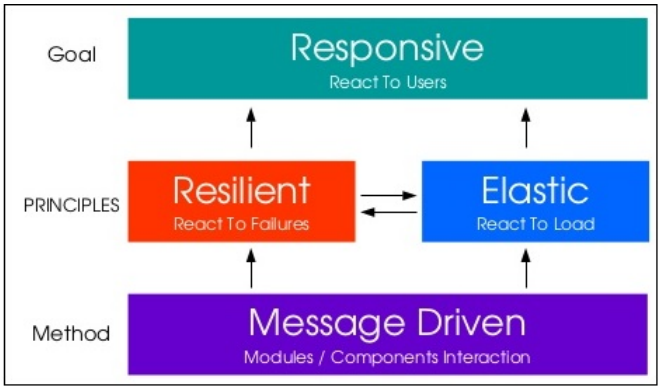


Figure 2.6: Architettura di un sistema reattivo

- Responsive: i sistemi reattivi devono fornire una risposta in maniera tempestiva, definendo un limite massimo se necessario, che permette di garantire una specifica qualità del servizio.

- Resilienti: i sistemi reattivi deve essere in grado di rispondere alle richieste anche nel caso in cui avviene un guasto. Viene usato il termine guasto in contrapposizione a quello d errore. Il guasto infatti rappresenta un problema che si verifica all'interno di un qualsiasi componente. Se infatti il componente è isolato, l'eventuale guasto non comporterà problemi all'intero sistema. In questo caso il recupero del guasto dovrà essere gestito da un componente esterno. Questo permette al singolo componente di non doversene preoccupare , perché delegato ad un altro.

- Elastici: i sistemi reattivi devono essere in grado di adattarsi, incrementando e decrementando le risorse in funzione della richiesta. Questo porta alla creazione di architetture che non abbiamo colli di bottiglia fornendo all'utente sempre una buona percezione di interattività.

- Orientato ai messaggi: I sistemi reattivi si basano sullo scambio di messaggi asincroni. Tali messaggi devono basarsi anche su un controllo del flusso, gestito grazie alla back-pressure.

2.5 Reactive Stream

Reactive Stream è un iniziativa che ha lo scopo di fornire uno standard per l'elaborazione degli stream asincroni con back-pressure non bloccante, ovvero per i sistemi reattivi. Le sue API sono state pensate a partire dall'ambiente JVM, Javascript e sui protocolli di rete.

Reactive Stream definisce come le librerie devono processare un potenziale numero infinito di elementi, in sequenza, in modo asincrono passando gli item tra i vari componenti con una back-pressure non bloccante. Queste sono definite su due parti:

- Le API, che specifica come implementare flussi reattivi e ottenere l'interoperabilità tra le diverse implementazioni
- Il TCK, che specifica uno standard per i test.

La comunicazione tra subscriber e publisher è quindi la seguente:

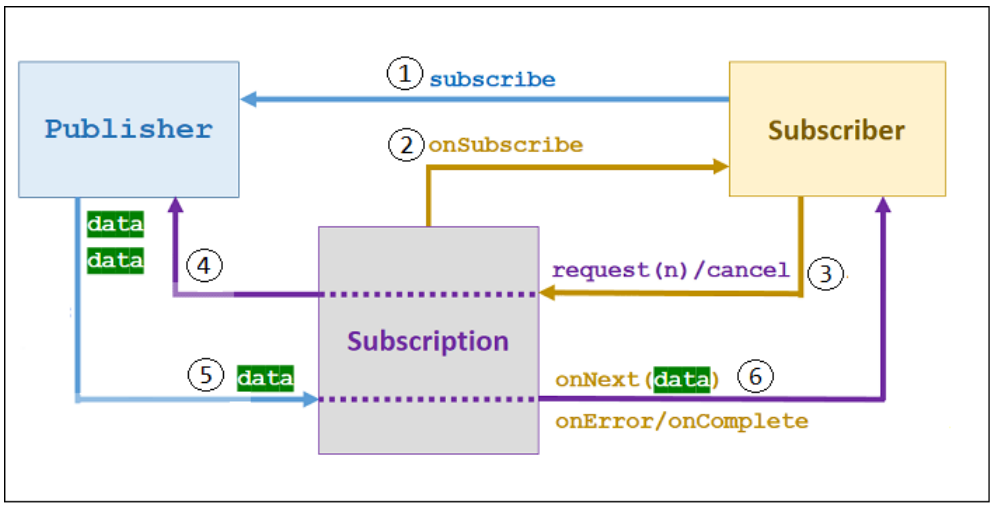


Figure 2.7: Comunicazione tra publisher e subscriber

1. Il Subscriber può utilizzare il metodo pubblico del Publisher per iscriversi a lui.
2. Viene invocata la callback OnSubscribe contenete il referimento alla Subscription, in questo modo il Publisher e il Subscriber hanno ora un legame.
3. Il Subscriber rende noto alla Subscription quanti elementi al massimo vuole ricevere utilizzando il metodo request(n). Si tenga presente che in qualsiasi momento il Subscriber può annullare la richiesta di elementi mediante il metodo cancel(), e non appena possibile la Subscription smetterà di inviargliene.

4. La subscription richiede (pull) gli n elementi al Publisher.
5. Il Publisher quindi invierà i dati alla Subscription
6. La Subscription invocherà verso il subscriber il metodo:
    - onNext() per fornire l'item ottenuto dal Publisher
    - onError() per informare il Subscriber che il Publisher ha riscontrato un errore
    - onComplete() per informare il Subscriber che il Publisher ha terminato gli elementi da inviargli.

Abbiamo però appena visto solo tre delle quattro interfacce definite dalle API reactive. Vediamole tutte insieme: Le interfacce sono quindi quattro:

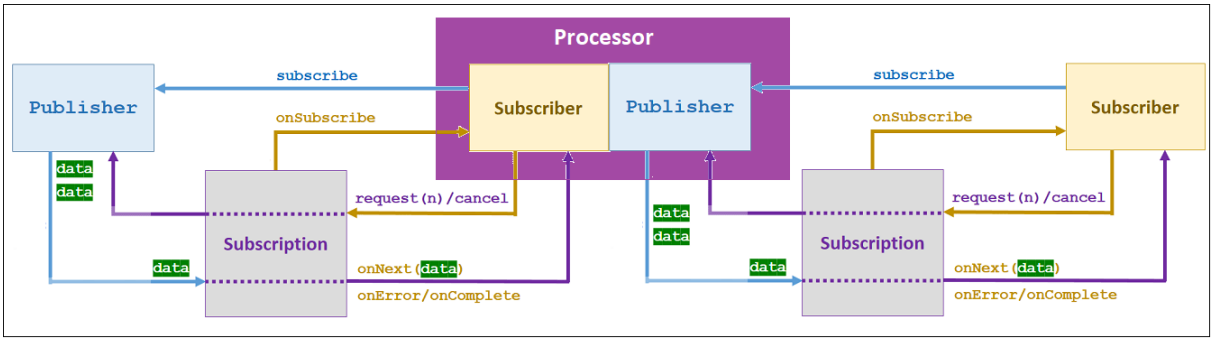


Figure 2.8: Le quattro interfacce

1. Il Publisher: ovvero colui che fornisce il flusso di elementi, quindi il produttore che "pubblica" dati a fronte di specifiche request().

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

Figure 2.9: API publisher

2. Subscriber: ovvero colui che riceve il flusso di elementi, quindi il consumatore.

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

Figure 2.10: API subscriber

3. Subscription: rappresenta il legame tra il publisher e il subscriber. Infatti grazie ad essa, il subscriber ha la possibilità di comunicare al publisher che vuole altri dati o cancellare le richieste fatte in precedenza, mentre il publisher ha la possibilità di inviare item al subscriber, oppure di comunicargli eventuali errori o terminazione dei dati.

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

Figure 2.11: API subscription

4. Il Processor: è un oggetto particolare che contiene al suo interno sia un publisher che un subscriber. Esso è infatti utilizzato in situazioni in cui vari componenti sono messi in cascata l'uno all'altro similmente a una catena di montaggio: un componente si trova ad essere un subscriber per uno specifico publisher, ma allo stesso tempo lui consuma gli item, li modifica e li invia a un altro subscriber, diventando quindi anche un publisher.

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

Figure 2.12: API processor

2.6 Le specifiche Reactive Stream per la jvm

Le specifice reactive stream vogliono definire alcune regole per il comportamento delle quattro interfacce sopra descritte.

Le specifiche reactive stream opportunamente pensate per la jvm sono state definite in un insieme di regole riassunte qui:

1. Regole a cui deve sottostare il publisher:
   (a) Può inviare un numero di item minore o uguale al numero di item richiesti dal subscriber
   (b) Se decide di inviare meno item di quelli richiesti, può (senza obbligo) invocare onComplete o onError, ma se:
      i. Termina con successo, allora DEVE segnalare onComplete
      ii. Termina con errore, DEVE segnalare onError
   (c) La segnalazione degli eventi onNext, onComplete, onError e onSubscribe DEVE essere thread safe, ovvero se il publisher usa più di un thread per fare tali chiamate non può iniziare una nuova chiamata finché quella precedente non è terminata.
   (d) L'invocazione di onError, onComplete o cancel provoca la terminazione della comunicazione tra publisher e subscriber, quindi nessun altro segnale tra questi due componenti (publischer e subscription) potrà più verificarsi.
   (e) Un publisher può decidere di avere più di un subscriber e quindi scegliere se la subscription è unicast o multicast.
2. Regole a cui deve sottostare il subscriber:
   (a) Per ricevere segnali di onNext DEVE prima invocare la request. Riceverà quindi un numero di onNext pari al massimo alla somma di tutte le richieste effettuate
   (b) Deve invocare cancel() se:
      i. possiede già una Subscription attiva e intende iscriversi a un altro publisher. Infatti un Subscriber non può avere più di un Publisher.
      ii. la Subscription che possiede non gli è più utile, così da comunicare al Publisher che non vuol più ricevere da lui alcun dato.
   (c) Le chiamate effettuate sulla Subscription devono essere invocate sempre dallo stesso thread, oppure deve essere implementata una opportuna sincronizzazione affinchè la subscription venga usata da un solo thread alla volta

(d) Deve essere preparato a rivecere più segnali di onNext dopo aver invocato la cancel, perché possono essercene rimasti ancora alcuni in coda

(e) Deve essere sempre preparato a ricevere la onComplete anche senza aver inviato alcuna request

(f) Deve assicurarsi che i metodi onNext, onComplete, onError o onSubscribe che lui crea siano tutti thread safe

(g) Le invocazioni di onNext, onComplete, onError o onSubscribe, deve sempre ritornare normalmente. Se si verificasse un problema, allora:

i. dovrà generata una java.lang.NullPointerException solo se uno dei parametri passati vale null

ii. per tutte le altre situazioni di errore, il modo corretto di gestire il problema è invocare la cancel

3. Le Regole a cui deve sottostare il processor: deve obbedire a entrambe le regole imposte sia al publisher che al subscriber.

2.7 La programmazione funzionale

La programmazione funzionale è uno stile di programmazione che mira a dichiarare, ad un livello più astratto, le operazioni chiave da eseguire permettendo di ottenere un codice più leggibile e quindi riusabile, anche se un poco meno manutenibile perché al verificarsi di un errore non è facile definirne la causa.

Due delle caratteriste base della programmazione funzionale sono:

- Immutabilità dello stato: permette di garantire che le variabili all'interno di un programma, non variano con l'esecuzione di codice scritto con uno stile di programmazione funzionale.

- Lazy evalutation: è la strategia utilizzata per valutare un'espressione solo quando il suo valore è davvero necessario, evitando quindi di valutare un'espressione che, al variare di eventi, potrebbe non servire più.

Questo tipi di programmazione fu introdotto già in Java 8 con l'introduzione della libreria java.util.Stream, e quindi anche mantenuto in Java 9 con la libreria java.util.concurrent.Flow per lo scambio asincrono di dati.

2.8 Adozioni

Reactive stream nacque nel 2013 da alcune delle più influenti società di applicazioni web (incluso Netflix) come mezzo per standardizzare lo scambio di dati asincroni tra componenti software.

Mano a mano nacquero varie librerie e quindi varie implementazioni Java.
Reactive Stream, riuscì a trovare un modo semplice per definire un numero minimo di interfacce, metodi e protocolli che permettesse di descrivere tutte le operazioni e le entità necessarie per la gestione asincrona di flussi di dati attraverso una backpressure non bloccante.

Questo permise l'introduzione di Java 9, la quale cercò di ridurre questa incompatibilità includendo le interfacce di base nella libreria Flow Concurrency, così da non dipendere da un'implementazione specifica di Reactive Stream.

Le diverse librerie che sono state implementate per la jvm sono:

- Reactor, è la libreria implementata da Pivotal per il framework Spring WebFlux
- RxJava, è la libreria utlizzata da Netflix
- Akka Stream, un'implementazione ad attori dei Reactive Stream per i linguaggi Java e Scala
- molte altre...

La creazione di un'implementazione personalizzata dello standard Reactive Stream può essere soggetta a errori e deve essere verificata attraverso il Reactive Streams Technology Compatibility Kit (TCK). E' quindi consigliato che le applicazioni utilizzino un'implementazione standard per gli ambienti di produzione.

3

Il framework Spring WebFlux e la libreria Reactor

3.1 Il framework Spring WebFlux

Il nuovo modulo offerto da Spring 5 si chiama “spring-webflux” ed è in grado di dare supporto alla programmazione reattiva per le Applicazioni web.

Per riuscire ad ottenere tutte i privilegi derivanti dalla programmazione reattiva è necessario avere uno stack completamente reattivo, quindi sia la parte client che quella server. Lato server è stata mantenuta un’architettura del tutto parallela a Spring MVC (versione non reattiva di Spring) ma compatibile con essa:

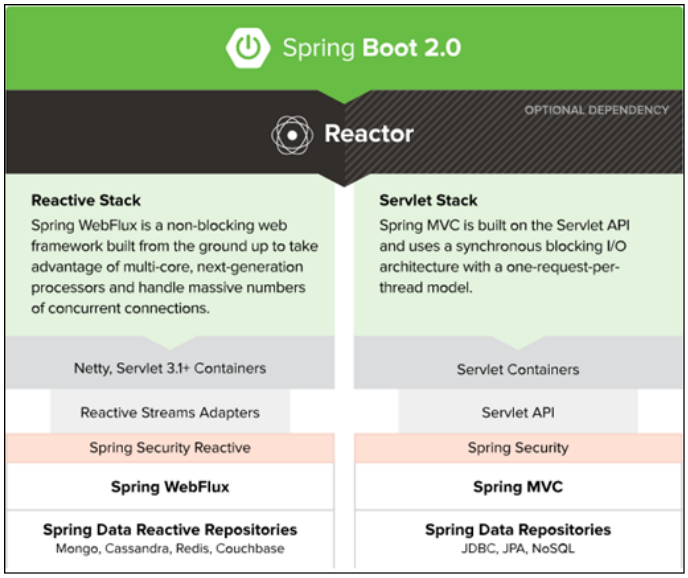


Figure 3.1: Stack webFlux e MVC a confornto

Nonostante lo stack sia stato mantenuto parallelo e del tutto compatibile, in realtà il modulo spring-webflux è stato completamente riscritto utilizzando un architettura basata sui messaggi, esattamente come le specifiche Reactive Stream richiedevano.

Infatti confrontando con più attenzione lo stack possiamo appunto notare che Spring WebFlux:

- Supporta anch'esso i controllori annotati tipici di Spring MVC, ma aggiunge un ulteriore modalità per l'implementazione della logica di business: le RouterFunction.
- Si basa sulla libreria reactive Stream eliminando quindi completamente l'utilizzo dei Servlet sopra cui MVC è costruito, ed introducendo gli elementi reattivi Flux e Mono.

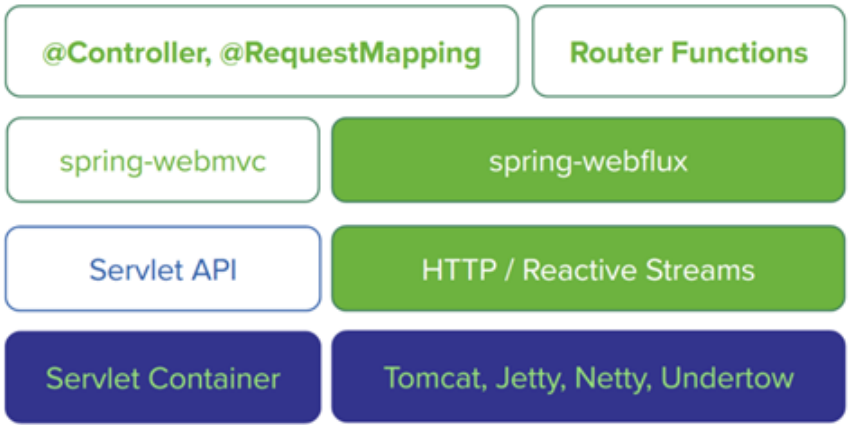


Figure 3.2: Lo stack MVC e WebFlux

3.2 Il server reattivo

All'interno di un'applicazione web, un thread (il dispatcher) sta sempre in ascolto di nuove richieste dai Client, inoltrando la richiesta al giusto Controller. Quando il controller riceve una richiesta utilizzerà un thread tra quelli disponibili del Container per l'esecuzione della richiesta http.

Tale thread poi avrà bisogno (tipicamente) di accedere a uno o più servizi e quindi si sottoscriverà ad esso attraverso una Subscription, ma non sarà bloccato nell'attesa che il servizio risponda, quindi si rimetterà a dormire nell'attesa di ricevere i dati da inviare al Client. Ogni elemento è collegato l'uno all'altro attraverso una sottoscrizione, similmente a una catena di montaggio.

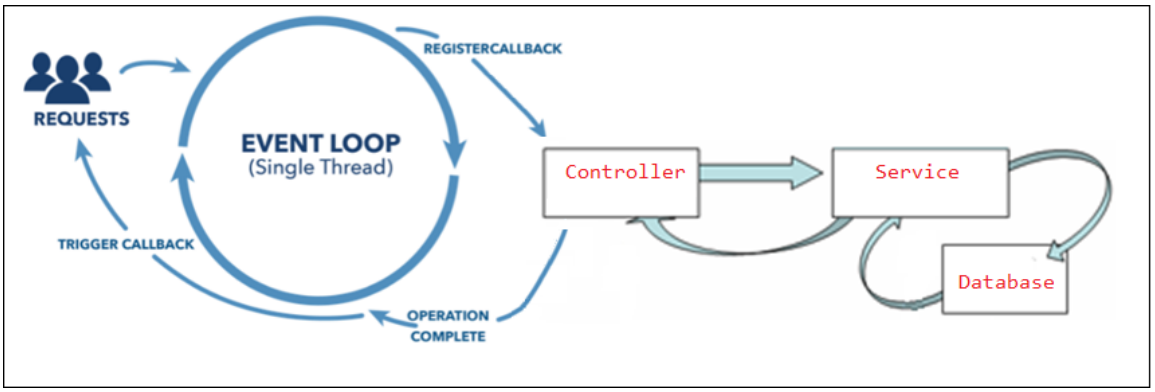


Figure 3.3: Achitettura server reattivo

L'intera architettura di Spring WebFlux è costruita quindi da una catena di Publisher e Subscriber, ognuno dei quali comunica attraverso uno scambio di messaggi (onNext, onComplete, onError e onSubscribe).

Confrontiamolo ora con l'architettura di Spring MVC: Spring MVC ha un'architettura in grado di gestire un solo thread per connessione. Non appena una richiesta http arriva viene messa all'interno di una coda in attesa che il dispatcher thread la prelevi e gli assegni un thread del thread pool in grado di portare avanti la richiesta (controller, service, ...):

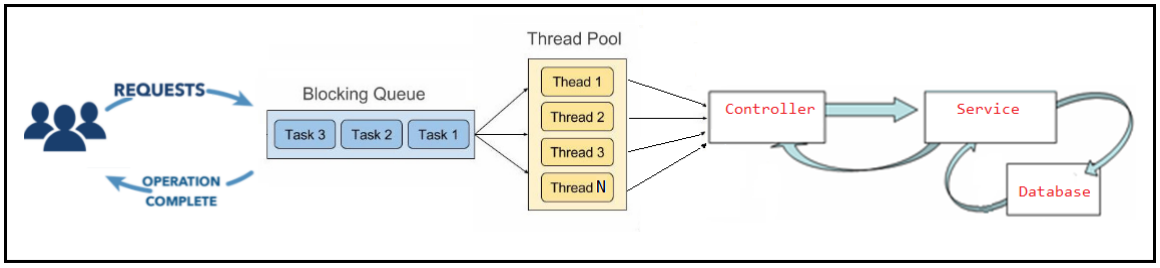


Figure 3.4: Achitettura server non reattivo

A causa della natura bloccate dalle API sincrone i thread del framework Spring MVC si bloccano, ma la cosa ancor peggiore è che non possono svolgere altro lavoro. Analizziamo infatti cosa succede se il thread di trova all'interno di un metodo e deve invocare più servizi, uno consegutivo all'altro:

- MVC: il thread invocherà il primo servizio e rimarrà bloccato in attesa della risposta, poi invocherà il secondo, e così via, ottenendo quanto già discusso in Fig.2.1, e dimostrato successivamente in fase di test nella Sez. 4.5.3

- WebFlux: il thread invoca la chiamata al primo servizio, poi può scegliere di proseguire con l'invocazione della seconda chiamata, sempre che questa non richieda l'uso del

risultato dato dal servizio precedente, e se così fosse è costretto a fermarsi, MA ritorna ad essere disponibile nel thread pool. Quando poi il risultato sarà disponibile un thread proseguirà dal punto in cui l'altro si era interrotto.

Con ciò possiamo quindi capire che webFlux se può parallelizza in modo automatico, mentre con Spring MVC questo potrebbe essere ottenuto solo se il programmatore espliticamente lo scrive mediante del codice, soluzione che in java spesso risulta critica.

Abbiamo quindi già potuto notare due grosse diversità in queste due architetture:

- se in MVC era il programmatore a dover implementare esplicitamente richieste asincrone, in webflux questo è fatto automaticamente dal framework permettendo:
  - Meno errori dovuti al programmatore
  - Maggior riusabilità del codice grazie a una migliore leggibilità del codice stesso.
- I thread in webflux eseguono un cambio di contesto dovuto al fatto che un thread può portare avanti richieste di client differenti. Questo è sicuramente più costoso, ma in un ambito in cui si fanno spesso richieste a servizi esterni, a parità di thread disponibili nel container, MVC rischierebbe di servire un numero limitato di client (1 connessione client = 1 thread) mentre webflux riuscirebbe a servire più client (1 thread = piu connessioni client).

3.3 Il client reattivo

Per poter sfruttare al meglio il concetto di reactive è necessario avere uno l'intero stack completamente reattivo.

Il comportamento sincrono si manifesta non solo nelle chiamate a servizi o connessione sulle basi di dati, ma anche in tutti quelli che sono gli accessi di rete. Quando un server si trova a fare richieste http a un altro server può rimanere bloccato in attesa delle risposte. Occorre pertanto passare da un modello di richiesta sincrona, come quello offerto per esempio da RestTemplate di MVC a un modello asincrono in cui sia possibile registarsi come ascoltatori delle risposte provenienti dai server che sono stati interrogati: per questa ragione Spring webFlux ha introdotto il componente WebClient che è reattivo non bloccante.

Tale oggetto quindi permette di:

- Fare una richiesta a un server
- Applicare trasformazioni e azioni sulla risposta quando essa ritornerà
- Non bloccare le altre operazioni del nostro codice

3.4 La libreria Reactor

La libreria scelta da Spring webFlux per l'implementazione delle specifiche reattive è Reactor.
Esso fornisce due diverse implementazioni dell'interfaccia Publisher:

- Flux<T> che rappresenta un flusso di elementi (0....N) di tipo T, come descritto questo diagramma a marble, all'interno della documentazione Reactor:

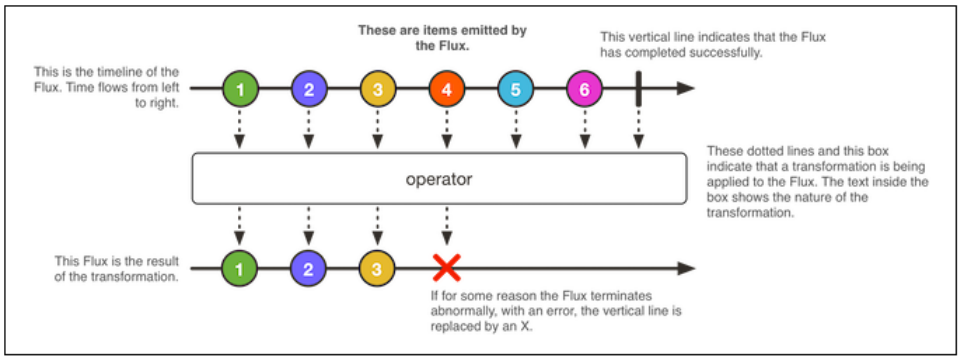


Figure 3.5: Flux<T>

- Mono<T> che rappresenta (0...1) elementi di tipo T, descritto nella documentazione così:

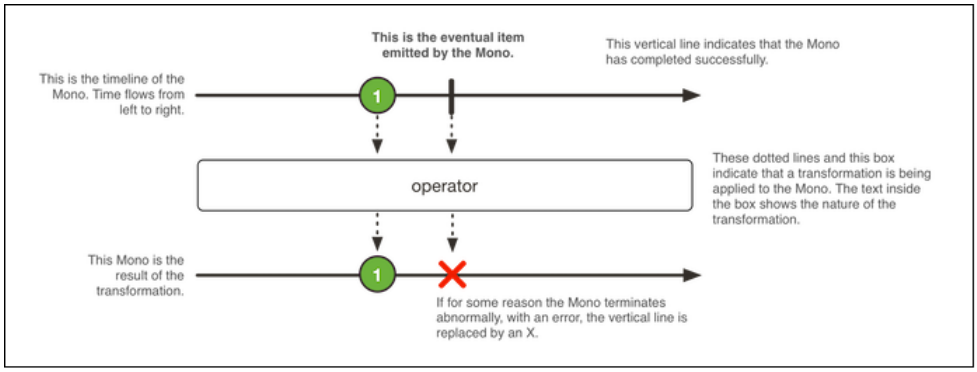


Figure 3.6: Mono<T>

Flux e Mono lavorano attraverso un modello deferred, push o pull:

- Deferred significa che nessun dato viene generato finchè non è necessario. Il produttore non potrà inviare dati finchè il ricevitore non gli indicherà quando è disponibile a ricevere.

- Pull significa che quando il consumatore è pronto per i dati lo segnala al produttore e si prende i dati
- Push significa che quando il produttore sarà pronto spingerà verso il consumatore i dati finché il consumatore non gli dice di fermarsi

3.4.1 Esempio semplice

Vediamo un esempio di coppia publisher-subscriber che si scambaiano dati.

```
@GetMapping(value="/getInteger", produces = {MediaType.APPLICATION_STREAM_JSON_VALUE, MediaType.TEXT_EVENT_STREAM_VALUE})
public Flux<Integer> getInteger(ServerHttpResponse h )
{
    return Flux.range(3,  5)                  // Crea 5 numeri a partire dal 3:                          3, 4, 5, 6, 7, 8
                .map(i -> i+3)                // Esegui su ogni numero un operazione:                    6, 7, 8, 9, 10, 1
                .filter(i -> i % 2 == 0 )     // Mantiene solo i numeri pari:                            6, 8, 10
                .filter(i -> i % 5 == 0 )     // Mantiene solo i numeri multipli di 5:                   10
                .switchIfEmpty(Flux.just(0));  // Se il flusso è vuoto lo sostituisce con un nuovo flusso:  10

}
```

Figure 3.7: Esempio

Possiamo notare che:

1. La sintassi tipica è quella della programmazione funzionale, molto simile agli stream() introdotti con Java 8 (vedi Sez.3.5)
2. Qui non è visibile il metodo subscribe(). In realtà per come è fatta l'architettura di spring webflux si ha che quando una richiesta http arriva al container, qualcuno si iscrive al controller affinchè lui gli restituisca il flusso (Flux<T>). In generale, l'operatore subscribe() deve essere invocato, altrimenti il flusso non verrà mai eseguito, e quello che si legge è solo la sua dichiarazione.
3. È necessario che il flusso inviato al client sia di tipo `TEXT_EVENT_STREAM_VALUE` oppure `APPLICATION_STREAM_JSON_VALUE`, altrimenti prima che la risposta venga scritta sul socket, il server attende di ricevere tutto il flusso dal controller per generare un unico json, bloccando quindi la risposta. Invece esplicitando uno di questi due MediaType, il socket del server invia al client gli elementi non appena li possiede.

Potrebbe quindi anche succedere che se il server ha molti dati da elaborare, potrebbe iniziare a elaborarli e nel frattempo inviare quelli già elaborati al client. Quindi il client riuscirebbe a vedere già i primi elementi quando in realtà il server sta ancora elaborando la rimanente risposta http.

3.4.2 Esempio focalizzato sulle API Reactive Stream

Tutta l'architettura del modulo spring-webflux prevede che ogni elemento sia collegato l'uno all'altro similmente a una catena di montaggio, quindi anche gli operatori dichiarati attraverso la programmazione funzionale.

Si supponga di dichiarare il seguente flusso utilizzando la programmazione funzionale:

```
Flux.range(5,  3)                 //   5, 6, 7
  . map(i -> i+3)                 //   8, 9, 10
  . filter(i -> i % 2 == 0 )      //   8, 10
  . buffer(3);
```

Figure 3.8: Esempio

Questa scrittura concettualmente corrisponde a dichiarare una sequenza di operatori collegati l'uno all'altro. Nessun elemento scorrerà però tra questi operatori finchè non si presenta un subscriber che vuole consumare i dati:

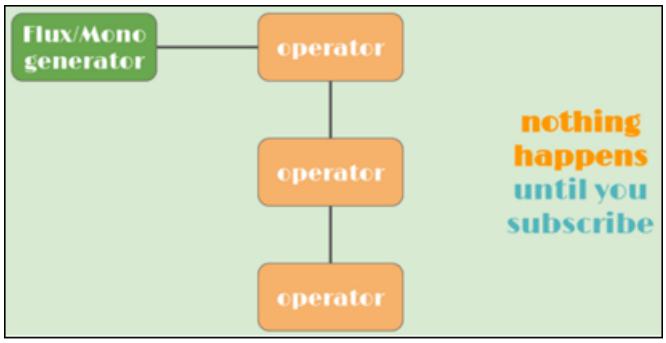


Figure 3.9: Esempio

Un subscriber può sottoscriversi al flusso utilizzando l'operatore subscribe(), che ha come parametro il metodo che consumerà i dati prodotti dal flusso. Se non è presente alcun metodo i dati verranno emessi ma nessuno li consumerà.

```
Flux.range(5,  3)                 //   5, 6, 7
    .map(i -> i+3)                //   8, 9, 10
    .filter(i -> i % 2 == 0 )     //   8, 10
    .buffer(3)
    .subscribe();
```

Figure 3.10: Esempio

Non appena un subscriber si sottoscrive al flusso , succedono due cose:

1. Il subscriber si iscrive all'operatore a lui collegato (buffer), quindi l'operatore "buffer" sarà un publisher per il "subscriber". Inoltre buffer si sottoscrive all'operatore filter, quindi "buffer" sarà il subscriber e "filter" il publisher, e così via fino al generatore del Flux, ovvero "range". Quindi viene instaurata quella che è la catena produttore-consumatore, dove ogni operatore è un produttore per l'operatore successivo, ma un consumatore per l'elemento precedente.

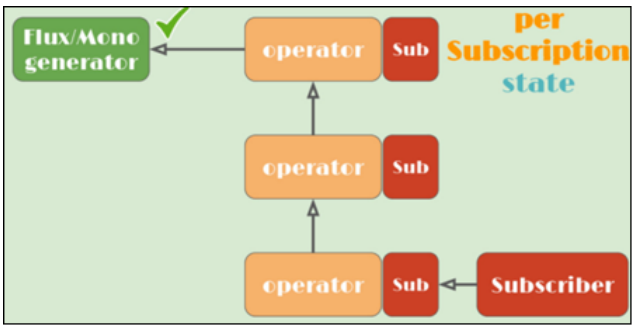


Figure 3.11: Esempio

2. Dopo che la catena è stata creata, il generatore range, inizierà ad inviare un elemento alla volta all'operatore successivo, così che ogni elemento possa quindi scorrere lungo la catena. Prendendo un istante a caso durante lo scorrimento dei dati tra i vari operatori, potremmo per esempio vedere:

   - Il publisher range che sta facendo onNext(7) verso il subscriber map
   - Il publisher map che sta facendo onNext(9) verso il subscriber filter
   - Il publisher filter che sta facendo onNext(8) verso il subscriber buffer

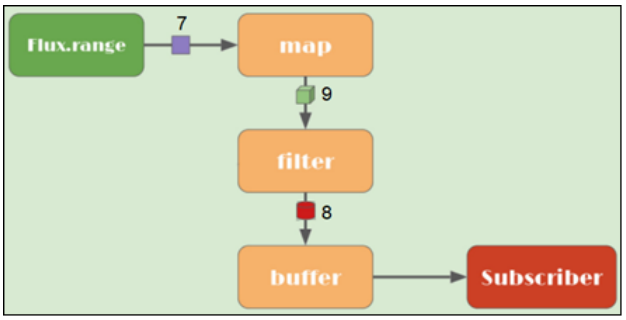


Figure 3.12: Esempio

Gli operatori intermedi rappresentano quindi i processor.

3.4.3 Esempio: operatore log

In questo esempio vogliamo provare a vedere più nel dettaglio un esempio di quali sono i messaggi scambiati tra l'operatore log e gli operatori map:

Si supponga di scrivere il seguente pezzo di codice all'interno di un metodo di un controller:

```
33  System.out.println("Dichiarazione del flusso....");
34
35  Flux<String> source = Flux.just(10, 20, 30, 40)
36                            .map(e -> e * 2)
37                            .log("L'intero prima che diventi stringa:")
38                            .map(i -> i + "; ");
39
40
41  System.out.println("Inizio del primo flusso: ");
42
43  source.subscribe(System.out::println);
44
45  System.out.println("Inizio secondo flusso....");
46  return source;
```

Figure 3.13: Esempio

Ciò che in console verrà quindi generato, sarà il seguente output:

```
Dichiarazione del flusso....
Inizio del primo flusso:
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onSubscribe([Fuseable] FluxMapFuseable.MapFuseableSubscriber)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | request(unbounded)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(20)
20;
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(40)
40;
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(60)
60;
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(80)
80;
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onComplete()
Inizio secondo flusso....
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onSubscribe([Fuseable] FluxMapFuseable.MapFuseableSubscriber)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | request(1)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(20)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | request(1)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(40)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | request(1)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(60)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | request(1)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onNext(80)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | request(1)
[nio-9000-exec-1] L'intero prima che diventi stringa:      : | onComplete()
```

Figure 3.14: Esempio

L'operatore log è un oggetto particolare che permette di stampare a console i messaggi che lui invia agli altri. Non appena il thread invoca il metodo subscriber(System.out::println) ogni elemento della catena si sottoscrive e i dati cominciano a fluire, infatti possiamo notare quelli che sono i segnali inviati dall'operatore log agli altre due operatori map a lui collegati:

1. onSubscribe: log si sta iscrivendo all'operatore map(e -> e * 2)

2. request (unbounded): log sta chiedendo all'operatore map (e -> e *2) un numero unbounded di elementi. La quantità di elementi richiesti (in questo caso è unbounded) è scelto dal framework

3. onNext(20): log sta inviando a map (i -> i + "; ") l'elemento "20".

4. System.out::println è il consumatore, quindi consuma il dato scrivendolo a console.

5. ...

6. onComplete: log specifica a map (e-> e*2) che non ha più elementi da dargli, così che cessa la comunicazione tra loro.

Quando il thread quindi termina l'esecuzione del flusso precedente, trova un'altra subscribe che è quella implicita tra il client e il controller, perché il flusso deve essere ritornato al client. Notiamo infatti che il flusso viene nuovamente eseguito, questa volta eseguendo una request di un solo elemento alla volta. Questo stato fatto perché si è scelto di mettere nel controller un producer di tipo `APPLICATION_STREAM_JSON_VALUE`, così che ogni elemento sul socket venga subito inviato al client.

Possiamo quindi notare che:

1. Il thread incontra la dichiarazione del flusso source, ma non lo esegue, perché ancora nessun subscriber si è iscritto ad esso.

2. Il flusso è riutilizzabile, infatti abbiamo due subscriber per lo stesso flusso:

   (a) Il primo è definito dal metodo (della classe System.out) passato come parametro alla subscriber(), infatti possiamo vedere che il risultato effettivo (20; 40; 60; 80;) lo consuma la console.

   (b) Il secondo è definito dalla richiesta http che ha prodotto il Client:

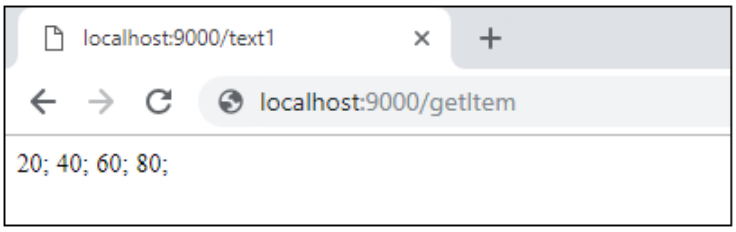


Figure 3.15: Esempio

3.5 Stream di Java 8

Attraverso il concetto di Stream lo sviluppatore può elaborare flussi di dati in modo dichiarativo (attraverso l'utilizzo della programmazione funzionale) con un alto livello di astrazione e quindi sfruttare l'architettura multi-core senza la necessità di scrivere alcun codice specifico per esso. Proviamo a confrontare gli stream di Java 8 con l'oggetto Flux:

	Caratteristiche di Java 8	**Caratteristiche Reactor**
Cosa producono?	Possono manipolare un insieme di dati producendo in uscita raccolte (Collection) oppure oggetti semplici	Possono manipolare un insieme di dati producendo in uscita oggetti di tipo Flux<T> o Mono<T>
Come gestiscono lo scambio dati?	Sono progettati per avere un'unica sorgente di input arbitraria che produca dati in modo inaffidabile, ovvero non è in grado di rallentare la produzione dei dati se il consumatore non riesce a consumarli abbastanza velocemente, generando un overflow che scatenerà un'eccezione	Sono progettati per avere una sorgente arbitraria che produca un numero infinito di valori, in un periodo di tempo sconosciuto, con una gestione affidabile del flusso stesso
Sono riutilizzabili?	Gli stream sono utilizzabili una sola volta, infatti se si provasse a riutilizzarli, si otterrebbe una IllegalStateException. (Fig. 3.17)	I flussi reattivi sono riutilizzabili per un numero indefinito di volte (Fig. 3.16)
Che modello seguono?	Utilizza un modello di tipo pull. Quindi ogni singolo elemento del flusso deve essere richiesto e rianendo bloccato in attesa della risposta se il produttore non ha il dato subito pronto.	Può utilizzare sia un modello pull che push: • Pull, quando vengono richiesti n elementi e il publisher li emette subito. • Push, quando vengono richiesti n elementi che però ancora non sono disponibili, quindi il publisher ne farà il push non appena potrà.

Esempio di flusso reattivo riutilizzabile:

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

j.map(i -> i * 10)
 .subscribe(System.out::println);

j.map(i -> i + 5)
 .subscribe(System.out::println);
```

Figure 3.16: Esempio

Esempio di flusso non reattivo (stream di Java 8) che produce eccezione perché non è riutilizzabile:

```
Stream<Integer> j = Arrays.asList(1, 2, 3, 4, 5).stream();

j.map(i -> i * 10)
  .forEach(System.out::println);

j.map(i -> i + 5)
  .forEach(System.out::println);
```

Figure 3.17: Esempio

3.6 Librerie Java Asincrone

Purtroppo in java non esistono molte librerie asincrone e i tipici modi per l'uso della programmazione asincrona sono:

1. Callback: ovvero per rendere un metodo asicnrono basta assegnargli come parametro una callback (classe astratta o lambda function), la quale verrà invocata quando il risultato è disponibile. Questi sono i tipici esempi di EventListener.

2. Future: ovvero esiste un oggetto Future<T> che incapsula l'oggetto T che è il valore di ritorno della chiamata asincrona e basta interrogarlo per ottenerne il valore.

Il problema d queste due modalità è che hanno entrambe delle limitazioni:

1. Le callback: non è facile comporre insieme il risultato dato da più callback. E anche se il programmatore riuscisse a farlo, si porta il codice ad essere difficile da gestire e poco leggibile. Questo problema è noto come "Callback Hell". Vedi Esempio.

2. Future<T> : con i future si possono gestire i metodi asincroni, ma prima dell'introduzione di Java 8 con i CompletableFuture, non era possibile sapere se il l'oggetto T era già disponibile e non appena si cercava di leggerlo con il metodo get() si poteva rimaneva bloccati se T ancora non era disponibile. Con i CompletableFuture, questo è stato risolto, ma ancora non c'è supporto per la gestione avanzata degli errori e per la combinazione di più valori T ritornati da servizi diversi.

3.6.1 Esempio: limitazione delle callback

Consideriamo un esempio: si immagini di voler legere i primi 5 preferiti dell'utente, oppure dei suggerimenti se non ci sono preferiti. Questo viene fatto richiamando tre servizi: uno che recupera gli ID degli utenti, uno che preleva i preferiti con dettagli e un altro che preleva dei suggerimenti.
Con le callback dovremmo scrivere il seguente codice:

```
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);
  }
});
```

Figure 3.18: codice asincrono mediante Callback

Con Reactor invece avremmo un codice nettamente più snello e leggibile:

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

Figure 3.19: codice asincrono mediante Reator

La "Callback Hell" è infatti noto come la difficoltà che incontra lo sviluppatore nel riuscire a comporre il risultato di più callback, che rendono il codice poco leggibile e manutenibile.

3.6.2 Esempio: limitazione dei CompletableFuture

Consideriamo un esempio: si immagini di voler recuperare un elenco di ID per ottenere di conseguenza il nome associato e una statistica e infine combinarli insieme, il tutto in modo completamente asincrono.

Con i CompletableFluture dovremmo scrivere il seguente codice:

```
CompletableFuture<List<String>> ids = ifhIds();

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);
                                        });
        List<CompletableFuture<String>> combinationList = zip.collect(Collectors.toList());

        CompletableFuture<String>[] combinationArray = combinationList.toArray(
                                                                new CompletableFuture[combinationList.size()]);

        CompletableFuture<Void> allDone = CompletableFuture.allOf(combinationArray);

        return allDone.thenApply(v -> combinationList.stream()
                                                .map(CompletableFuture::join)

                                                .collect(Collectors.toList()));
                                });
List<String> results = result.join();
assertThat(results).contains(
                                "Name NameJoe has stats 103",
                                "Name NameBart has stats 104",
                                "Name NameHenry has stats 105",
                                "Name NameNicole has stats 106",
                                "Name NameABSLAJNFOAJNFOANFANSF has stats 121");
```

Figure 3.20: codice asincrono mediante CompletableFluture

Con Reator invece avremmo un codice più leggibile:

```
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();
assertThat(results).containsExactly(
                "Name NameJoe has stats 103",
                "Name NameBart has stats 104",
                "Name NameHenry has stats 105",
                "Name NameNicole has stats 106",
                "Name NameABSLAJNFOAJNFOANFANSF has stats 121"
);
```

Figure 3.21: codice asincrono con Reactor

Con questi due esempi di programmazione asincrone abbiamo quindi potuto notare quanto, in alcune situazioni, possa diventare particolarmente complessa o limitata la scrittura di codice non bloccante.Un altro motivo per cui la programmazione reattiva offre un vantaggio considerevole!

3.7 La request in Reactor

Abbiamo visto le quattro interfacce, di cui Reactor ci fornisce due iplementazioni di base. La back-pressure è anch'essa implementata dal framework, grazie all'utilizzo della request. Se infatti non viene inviato il segnale di request, il componente precedente non può inviare dati a quello successivo.

Il segnale di request, che tipicamente è inviato in automatico dal framework grazie alle implementazioni Reactor, ppuò essere modificato da ogni singolo operatore.

Se consideriamo per esempio buffer(2) si avrà che, affinché lui possa emettere un item dovrà ricevere almeno 2 elementi alla volta, quindi la request invece che essere n dovrà essere 2*n.

Inoltre la request, se non ha n pari a long.MAX_VALUE, gestisce in un modo ottimizzato le richieste di item all'operatore precedente. Infatti il subscriber non aspetta di ricvere tutti gli item prima di inviare una nuova request(), ma non appena riceve il 25% degli elementi, esegue immediatamente una request pari al 25% di n elementi. Questa è un'ottimizzazione basata su un'euristica, che fa si che gli operatori anticipino le richieste in modo proattivo.

Esistono inoltre degli operatori in grado di regolare le richiesta:

- limitRate: se un elemento a valle della catena ricevesse una request(n) con n molto grande, l'introduzione della limitRate(x) fa sì che la request venga invece propagata a monte inoltrando più segnali di request con n pari a n/x, ovvero effettuando più request, ma con un n più piccolo.
- limitRequest(n): fa si che all'elemento più a monte nella catena (quello più vicino alla sorgente) possa inviare al massimo n elementi. Una volta che vengono forniti n elementi, quel flusso termina inviando un segnale di onComplete verso il subscriber.

3.8 Threading model

La libreria Reactor è considerata "concurrency agnostic" perché non forza un modello di tipo push o pull, ma lascia tale controllo allo sviluppatore attraverso opportuni operatori. Gli operatori limitRate e limitRequest, come abbiamo appena visto, modificano infatti la comunicazione tra un operatore e un altro.

Tipicamente l'intero flusso viene eseguito sul thread che invoca la subscriber a meno che vanga definito espliticamente o implicitamente uno scheduler su cui eseguirlo. Per esempio attraverso l'uso di un operatore (publishOn() o subscribeOn()) oppure specificando lo scheduler come parametro di un generico operatore, se esso lo permette. Vediamo un esempio: Flux.interval(Duration.ofMillis(300), Schedulers.newSingle("test"))

E' possibile definire degli scheduler personalizzati, altrimenti si possono utilizzare quelli messi a disposizione dal framework stesso:

1. Schedulers.immediate(), è uno scheduler che permette di eseguire l'intero flusso all'interno dello stesso Thread chiamante
2. Schedulers.single(), è uno scheduler che usa lo stesso thread per ogni subscriber
3. Scheduler.newSingle(), è uno scheduler che genera un thread dedicato per ogni Subscriber.
4. Scheduler.elastic(), è uno scheduler che crea un thread per l'esecuzione quando ne ha bisogno, e non appena esso termina il proprio lavoro cerca di riutilizzarlo per altro. Dopo 60 secondi che tale thread è nello stato di idle per mancanza di lavoro da svolgere, esso verrà eliminato (dispose). Questo scheduler è consigliato per l'esecuzione di chiamate bloccanti, così che il thread si dedichi a quel task senza bloccare risorse.

5. Scheduler.parallel(), è uno scheduler contenente un numero limitato di thread che hanno lo scopo di cercare di ottimizzare i task paralleli. Tipicamente il numero di questi thread (4 thread per le macchine comuni) è variabile a seconda del hardware sottostante. A causa del numero limitato di thread in questo scheduler è opportuno che tali thread non gestiscano task bloccanti per evitare di bloccare il sistema.

Reactor definisce due operatori per cambiare lo scheduler durante l'esecuzione di un flusso:

- publishOn(<Schedulers>)
- subscribeOn(<Schedulers>)

3.8.1 Esempio: operatore publishOn

PublishOn influenza il processo di produzione degli item. Il punto in cui viene posizionato l'operatore publishOn è significativo perché esso cambia lo scheduler a tutti e soli gli operatori a lui successivi lungo la catena, finché non si incontra un altro publishOn. Vediamo un esempio:

```
Flux<Integer> source = Flux.range(0, 5)
                           .filter(i -> i%2 == 0)
                           .doOnNext(s -> System.out.println
                                          (s + " before publishOn using thread: "
                                             + Thread.currentThread().getName()))
                           .publishOn(Schedulers.elastic())
                           .doOnNext(s -> System.out.println
                                          (s + " after publishOn using thread****: "
                                             + Thread.currentThread().getName()));

 Flux.range(0, 1)
     .doOnEach(value -> source.subscribe(so -> System.out.println
                                         (value + " consumer processed "
                                         + so + " using thread: "
                                         + Thread.currentThread().getName())))
     .subscribe();
```

Figure 3.22: Utilizzo dello scheduler elastic(): il codice

In console otterremmo quindi che tutti operatori successivi a publishOn fanno uso di un nuovo thread elastic() per ogni ri-esecuzione del codice:

```
0 before publishOn using thread: http-nio-9000-exec-1
2 before publishOn using thread: http-nio-9000-exec-1
4 before publishOn using thread: http-nio-9000-exec-1
0 after publishOn using thread****: elastic-2
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@72ddf9b4
        consumer processed 0 using thread: elastic-2
2 after publishOn using thread****: elastic-2
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@72ddf9b4
        consumer processed 2 using thread: elastic-2
4 after publishOn using thread****: elastic-2
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@72ddf9b4
        consumer processed 4 using thread: elastic-2
0 before publishOn using thread: http-nio-9000-exec-1
2 before publishOn using thread: http-nio-9000-exec-1
4 before publishOn using thread: http-nio-9000-exec-1
0 after publishOn using thread****: elastic-3
onComplete() consumer processed 0 using thread: elastic-3
2 after publishOn using thread****: elastic-3
onComplete() consumer processed 2 using thread: elastic-3
4 after publishOn using thread****: elastic-3
onComplete() consumer processed 4 using thread: elastic-3
```

Figure 3.23: Utilizzo dello scheduler elastic(): la console

Vediamo invece un esempio in cui si utilizza lo Scheduler single():

```
Flux<Integer> source = Flux.range(0, 5)
                          .filter(i -> i%2 == 0)
                          .doOnNext(s -> System.out.println
                                        (s + " before publishOn using thread: "
                                           + Thread.currentThread().getName()))
                          .publishOn(Schedulers.single())
                          .doOnNext(s -> System.out.println
                                        (s + " after publishOn using thread****: "
                                           + Thread.currentThread().getName()));

 Flux.range(0, 1)
     .doOnEach(value -> source.subscribe(so -> System.out.println
                                        (value + " consumer processed "
                                        + so + " using thread: "
                                        + Thread.currentThread().getName())))
     .subscribe();
```

Figure 3.24: Utilizzo dello scheduler sigle(): il codice

Si ottiene quindi un risultato simile a quello di prima, ma usando lo scheduler “Single”, si ottiene che la ri-esecuzione del flusso fa si che il thread utilizzato sia sempre single-1.

```
0 before publishOn using thread: http-nio-9000-exec-1
2 before publishOn using thread: http-nio-9000-exec-1
0 after publishOn using thread****: single-1
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@6f294900
        consumer processed 0 using thread: single-1
4 before publishOn using thread: http-nio-9000-exec-1
2 after publishOn using thread****: single-1
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@6f294900
        consumer processed 2 using thread: single-1
4 after publishOn using thread****: single-1
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@6f294900
        consumer processed 4 using thread: single-1
0 before publishOn using thread: http-nio-9000-exec-1
2 before publishOn using thread: http-nio-9000-exec-1
4 before publishOn using thread: http-nio-9000-exec-1
0 after publishOn using thread****: single-1
onComplete() consumer processed 0 using thread: single-1
2 after publishOn using thread****: single-1
onComplete() consumer processed 2 using thread: single-1
4 after publishOn using thread****: single-1
onComplete() consumer processed 4 using thread: single-1
```

Figure 3.25: Utilizzo dello scheduler single(): la console

3.8.2 Esempio: operatore subscribeOn

L’operatore SubscribeOn influenza invece il processo di sottoscrizione a partire dalla sorgente, ma non influenza mai il comportamento delle chiamate successive a PublishOn. Vediamo un esempio:

```
Flux<Integer> source = Flux.range(0, 5)
                          .filter(i -> i%2 == 0)
                          .doOnNext(s -> System.out.println
                                        (s + " before publishOn using thread: "
                                           + Thread.currentThread().getName()))
                          .publishOn(Schedulers.elastic())
                          .doOnNext(s -> System.out.println
                                              (s + " after publishOn using thread****: "
                                                 + Thread.currentThread().getName()));
                          .subscribeOn(Schedulers.single());

 Flux.range(0, 1)
     .doOnEach(value -> source.subscribe(so -> System.out.println
                                        (value + " consumer processed "
                                        + so + " using thread: "
                                        + Thread.currentThread().getName())))
     .subscribe();
```

Figure 3.26: Utilizzo dello scheduler single(): il codice

Come mostra la console in Fig.3.27 abbiamo che la SubscribeOn influenza tute le sottoscrizioni precedenti alla publishOn.

```
0 before publishOn using thread: single-1
2 before publishOn using thread: single-1
4 before publishOn using thread: single-1
0 after publishOn using thread****: elastic-2
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@1f67f2fc
        consumer processed 0 using thread: elastic-2
2 after publishOn using thread****: elastic-2
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@1f67f2fc
        consumer processed 2 using thread: elastic-2
4 after publishOn using thread****: elastic-2
0 before publishOn using thread: single-1
2 before publishOn using thread: single-1
4 before publishOn using thread: single-1
reactor.core.publisher.FluxDoOnEach$DoOnEachSubscriber@1f67f2fc
        consumer processed 4 using thread: elastic-2
0 after publishOn using thread****: elastic-3
onComplete() consumer processed 0 using thread: elastic-3
2 after publishOn using thread****: elastic-3
onComplete() consumer processed 2 using thread: elastic-3
4 after publishOn using thread****: elastic-3
onComplete() consumer processed 4 using thread: elastic-3
```

Figure 3.27: Utilizzo dello scheduler single(): la console

Si tenga conto quindi che:

1. il risultato non cambierebbe se si spostasse la subscribeOn in qualsiasi altra posizione lungo la catena

2. la subscribeOn influenza SOLO le sottoscrizioni precedenti alla publishOn() oppure tutta la catena se tale chiamata non è presente.

3.9 Il routing delle pagine

In Spring MVC, il dispatcher distribuiva le richieste ai controller opportuni attravarso l'uso di annotazioni come @Controller o @RestController che permettevano appunto di dichiarare al framework lo specifico metodo di un controller in grado di rispondere alla richiesta http del client.

Una particolarità molto interessate del modulo spring-webflux è che, nonostante l'architettura sia cambiata, la sintassi è rimasta sostanzialmente la stessa, ma internamente sono stati modificati due componenti: l'HandlerMapping e l'HandlerAdapter che in MVC utilizzavano HttpServletRequest e HttpServletResponse, in webFlux utilizzano ServerHttpRequest e ServerHttpResponse che sono due oggetti reattivi quindi non bloccanti.

Vediamo quindi un esempio:

```
@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);
	}
}
```

Figure 3.28: Controllori annotati

Con le annotazioni classiche di MVC è possibile esprimere solo alcune caratteristiche che la richiesta http deve avere, mentre altre più complesse no. Se dovessimo per esempio aggiungere della logica un poco più complessa si possono invece usare le RouterFunction.

Queste consentono di esprimere in modo più astratto una pluralità di vincoli sulla struttura della richiesta ricevuta demandando eventualmente ad apposite lambda function, che verranno eseguite all'atto della ricezione della richiesta, l'esecuzione di controller spcifici volte a determinare se una data richiesta debba essere o meno presa in carico da una specifica funzione.

In MVC infatti non è possibile esprimere dei filtri mediante l'uso di annotazioni, ma occorre definire una classe specifica e interporla prima dell'invocazione del controller. Questo è il motivo principale per cui Spring webFlux ha messo a disposizione un'altra modalità per gestire il routing delle pagine: le routerFunction.

Le routerFunction sono dei bean che vengono definiti all'avvio del server e che cercano di sostituire le annotazioni @RequestMapping. Esse possono essere messe in un unico punto centralizzato, oppure no, a scelta dello sviluppatore.

Inoltre, nella nuova notazione i Controller vengono chiamati HandlerFunction perché di fatto sono delle funzioni che prendono come parametro la request http e rispondono con una response http.

Vediamo un esempio:

```
@Bean
  PersonHandlerInterface person() {
          return new PersonHandler();
  }

  @Bean
  RouterFunction<ServerResponse> helloRouterFunction(PersonHandlerInterface p)
  {

    RouterFunction<ServerResponse> router = RouterFunctions

                    .route(RequestPredicates.GET("/getListPersons")
                                    .or(RequestPredicates.GET("/getPersons")), p::getListPersons)
                    .andRoute(RequestPredicates.GET("/Person/{id}"), p::findOnePerson)
                    .filter((request, next) ->
                                    {
                                            if (!(request.pathVariables().size()<=1))
                                                    return  ServerResponse.status(HttpStatus.FORBIDDEN).build();
                                            else
                                                    return next.handle(request);
                                    });
    return router;
  }
```

Figure 3.29: Esempio di routerFunction

In questa routerFunction ho definito:

- l'equivalente dell'annotazione @RequestMapping con metodo GET attraverso un RequestPredicate
- un metodo filter che permette di filtrare tutte le richieste http che hanno più di un parametro. Questo è stato fatto solo a scopo di esempio, ma è possibile creare qualsiasi tipo di filtro in grado di fare un controllo degli accessi, nonché la profilazione.

Un filtro lo si può creare in due modi:

1. utilizzando una lambda, come nell'esempio, che prende in input la richiesta http e il prossimo HandlerFunction<Response>

2. creando un apposita classe che implementa l'interfaccia HandlerFilterFunction contenente un solo metodo: filter

Una RouterFunction è tipicamente associata ad una o più HandleFunction, componenti che hanno il compito di generare l'opportuna risposta implementando la necessaria logica applicativa.

L'HandlerFunction rappresenta la funzione che viene invocata quando esiste il match con il RequestPredicates opportuno. In particolare la classe HandlerFunction che ho creato per il routing delle pagine sopra descritto è il seguente:

```
public class PersonHandler implements PersonHandlerInterface
{
	@Autowired
	private PersonService personService;


	@Override
	public Mono<ServerResponse> getListPersons(ServerRequest request)
	{
		Flux<Person> c = Flux.from(personService.findAll());

		Mono<ServerResponse> monoServResp = ServerResponse.ok()
						.contentType(MediaType.TEXT_EVENT_STREAM)
						.body(c, Person.class);
		return monoServResp;
	}


	@Override
	public Mono<ServerResponse> findOnePerson(ServerRequest request)
	{
		Mono<Person> c = personService.findById(request.pathVariable("id"));

		Mono<ServerResponse> monoServResp = ServerResponse.ok()
						.contentType(MediaType.TEXT_EVENT_STREAM)
						.body(c, Person.class);

		return monoServResp;
	}
}
```

Figure 3.30: Esempio di HandlerFunction

Dove l'interfaccia è la seguente:

```
public interface PersonHandlerInterface
{
        public Mono<ServerResponse> getListPersons(ServerRequest request);

        public Mono<ServerResponse> findOnePerson(ServerRequest request);

}
```

Figure 3.31: PersonHandlerInterface

3.10 Stream Hot e Cold

La libreria reactor distingue due tipi di flussi:

1. Stream Cold
2. Stream Hot

Gli stream Cold sono stream tali per cui il flusso viene ri-eseuito per ogni subscriber. Vediamo un esempio:

```
Flux<String> source = Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))
                          .doOnNext(System.out::println)
                          .filter(s -> s.startsWith("o"))
                          .map(String::toUpperCase);

source.subscribe(d -> System.out.println("Subscriber 1: "+d));
source.subscribe(d -> System.out.println("Subscriber 2: "+d));
```

Figure 3.32: Esempio di stream Cold

Come abbiamo detto, in console troviamo che il flusso è stato eseguito due volte, infatti due sono i subscriber:

```
blue
green
orange
Subscriber 1: ORANGE
purple

blue
green
orange
Subscriber 2: ORANGE
purple
```

Figure 3.33: Esempio di stream Cold

Gli stream Hot sono stream tali per cui il flusso viene eseguito una sola volta anche se non vi è alcun subscriber iscritto. Quindi se il flusso comincia l'esecuzione e un subscriber si aggancia ad esso in un secondo momento, ciò che riceverà potrebbe non essere l'intero flusso.

Per la precisione, ciò che effettivamente riceve un subscriber che si aggancia a un flusso hot è definito da alcune regole che sono diverse a seconda del tipo di oggetto che viene utilizzato come publisher: Reactor ha definito vari publisher che possono produrre flussi hot:

1. La classe ConnectableFlux
2. Le classi Processor (Sez. 3.12)

3.11 La classe ConnectableFlux

La classe ConnectableFlux permette a un publisher di generare una sola volta l'intero flusso e fornirlo a più subscriber, così da evitarne la ri-esecuzione per ogni subscriber.

ConnectableFlux ha delle regole molto rigide:

1. Si può creare un oggetto ConnectableFlux invocando il metodo publish sul flusso che verrà eseguito
2. Per avviare l'esecuzione del flusso è possibile invocare uno di questi metodi:
   (a) connect()
   (b) autoconnect(int n)
   (c) refCount(n)
   (d) refCount(int n, Duration t)
3. Affinché ogni subscriber possa ricevere i dati emessi dal publisher deve sottoscriversi prima dell'avvio del flusso.

Vediamo un esempio:

```
Flux<Integer> source = Flux.range(3, 7);

ConnectableFlux<Integer> hotFlux  = source.publish();

System.out.println("Ora i subscriber si iscrivono...");

hotFlux.subscribe(d -> System.out.println("Subscriber 1 : "+d ));
hotFlux.subscribe(d -> System.out.println("Subscriber 2 : "+d ));
hotFlux.subscribe(d -> System.out.println("------------ : "+d ));


System.out.println("Ora verrà avviato il flusso...");
hotFlux.connect();

hotFlux.subscribe(d -> System.out.println("Non riceverò mai dati : "+d ));
System.out.println("FLUSSO TERMINATO.");
```

Figure 3.34: Esempio di codice con stream Hot

In console otteniamo quindi:

```
Ora i subscriber si iscrivono...
Ora verrà avviato il flusso...
Subscriber 1 : 3
Subscriber 2 : 3
------------ : 3
Subscriber 1 : 4
Subscriber 2 : 4
------------ : 4
Subscriber 1 : 5
Subscriber 2 : 5
------------ : 5
Subscriber 1 : 6
Subscriber 2 : 6
------------ : 6
Subscriber 1 : 7
Subscriber 2 : 7
------------ : 7
Subscriber 1 : 8
Subscriber 2 : 8
------------ : 8
Subscriber 1 : 9
Subscriber 2 : 9
------------ : 9
FLUSSO TERMINATO.
```

Figure 3.35: Esempio di esecuzione di un flusso Hot

Ciò che succede è:

1. Abbiamo creato un flusso "source"

2. Creiamo il ConnectableFlux a partire dal source e invochiamo il metodo publish per creare un oggetto di tipo ConnectableFlux

3. I subscriber si iscrivono al ConnectableFlux invece che al Flux, questo è il motivo per cui l'avvio del flusso non avviene nel momento del subscribe() (come notiamo da console) ma avviene DOPO l'invocazione della connect().

4. I subscriber si devono iscrivere al prima dell'invocazione della connect(), altrimenti non riceveranno mai i dati, come possiamo vedere dalla Fig.3.35

5. Il flusso è eseguito una volta sola, con il thread che invia i dati a ogni subscriber iscritto.

Il metodo autoConnect è anch'esso molto interessante: esso infatti permette l'avvio del flusso solo se ci sono esattamente n subscriber iscritti quando l' autoConnect(n)[1] viene invocata.
Vediamo un esempio:

```
Flux<Integer> source = Flux.range(3, 7);

ConnectableFlux<Integer> hotFlux  = source.publish();

System.out.println("Ora i subscriber si iscrivono...");

hotFlux.subscribe(d -> System.out.println("Subscriber 1 : "+d ));
hotFlux.subscribe(d -> System.out.println("Subscriber 2 : "+d ));
hotFlux.subscribe(d -> System.out.println("------------ : "+d ));


System.out.println("Ora verrà avviato il flusso...");
hotFlux.autoConnect(2);

hotFlux.subscribe(d -> System.out.println("Non riceverò mai dati : "+d ));
System.out.println("FLUSSO TERMINATO.");
```

Figure 3.36: Esempio di esecuzione di un flusso Hot

E in console vedremmo:

```
Ora i subscriber si iscrivono...
Ora verrà avviato il flusso...
FLUSSO TERMINATO.
```

Figure 3.37: Esempio di esecuzione di un flusso Hot

[1]La documentazione ufficiale descrive che n rappresenta il numero minimo di Subscriber, ma come mostra l'esempio, se ci sono più di n Subscriber iscritti, il flusso non viene avviato.

3.12 I processor

I processor sono degli oggetti che sono contemporaneamente sia subscriber sia publisher. Essi infatti ricevono dai da uno o più publisher e sono in grado di emettere item verso uno o più subscriber.

Il loro comportamento è diverso a seconda dall'implementazione del processor che si sceglie di utilizzare.

Ognuno di essi ha in realtà delle criticità e riescono a comportarsi bene solo se davvero vengono utilizzati nel modo corretto, quindi occorre molta attenzione. Analizzeremo ora i 6 tipi di processor definiti nella libreria Reactor di Spring definendoli e mostrando alcune particolarità di essi:

A seconda di alcune caratteristiche sono state suddivisi in tre gruppi:

1. Processor "Direct": questi possono solo fare il push dei dati attraverso un'azione diretta dell'utente chiamando il metodo sink:
   (a) DirectProcessor
   (b) UnicastProcessor
2. Processor "sincroni": questi processor possono o fare il push dei dati attraverso un interazione utente o sottoscrivendosi a un publisher che in modo sincrono gli invia i dati:
   (a) EmitterProcessor
   (b) ReplayProcessor
3. Processor "Asincroni": questi processor possono fare il push dei dati ottenuti da varie sorgenti o anche attraverso l'interazione utente. Sono piu robusti grazie un RingBuffer, una struttura dati che ha lo scopo di accogliere le multiple sorgenti:
   (a) TopicProcessor
   (b) WorkQueueProcessor

3.12.1 Il DirectProcessor

I DirectProcessor sono contemporaneamente un Subscriber e un Publisher, il cui comportamento è il seguente:

- Il subscriber può ricevere il flusso di dati da più sorgenti
- Il publisher è in grado di distribuire lo stesso flusso a più subscriber, eventualmente anche effettuando elaborazioni diverse per ogni publisher (vedi esempio)

Dalla documentazione ufficiale possiamo vedere il diagramma che ne sintetizza il comportamento:

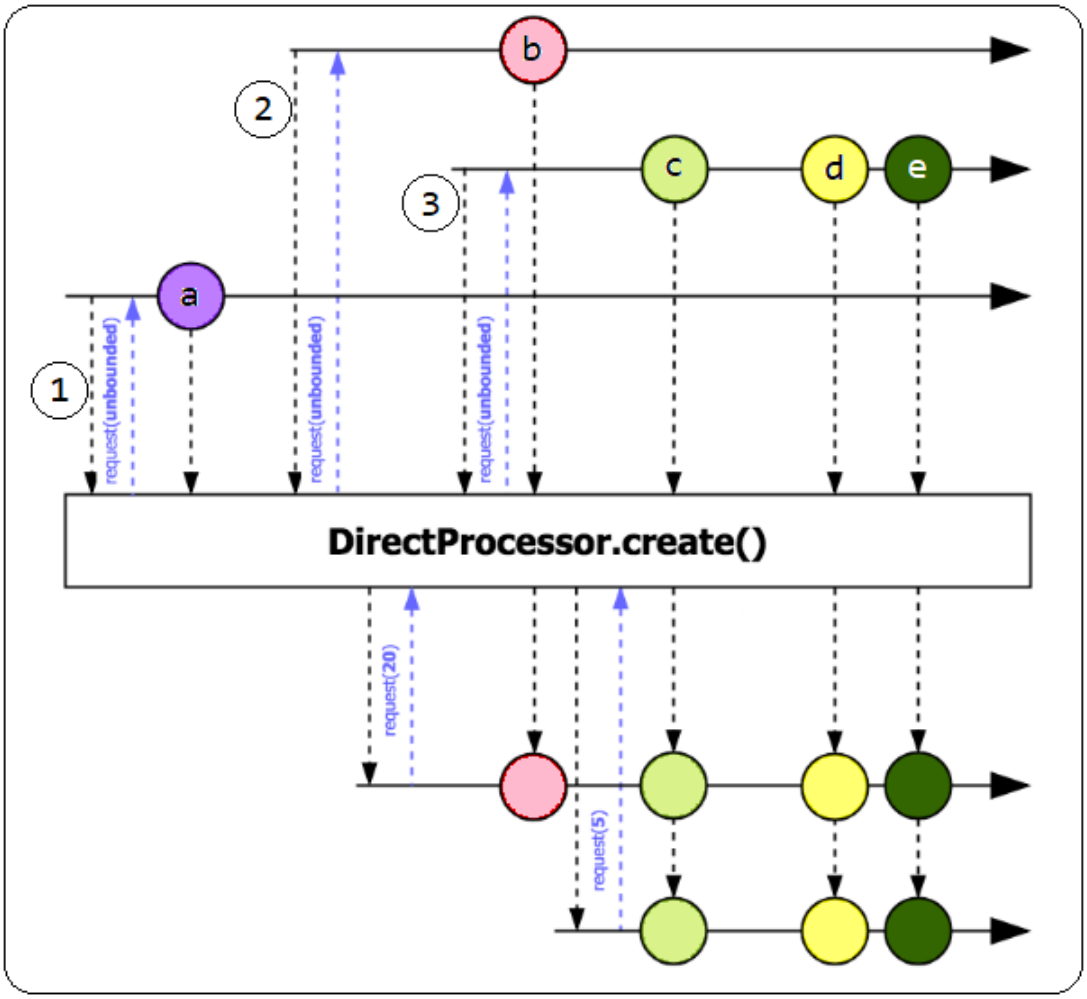


Figure 3.38: Comportamento del DirectProcessor

Possiamo infatti capire che:

- Il directProcessor si sottoscrive a flussi diversi in istanti diversi (1 – 2 – 3 in Fig.3.38), ottenendo così dati da publisher diversi
- Tutti i dati che il DirectProcessor riceve vengono eliminati se nessun Subscriber è iscritto al DirectProcessor (item “a”)
- Vi è un solo thread che gestisce il componente DirectProcessor

Vediamo un esempio:

```
Flux<Integer> c = Flux.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11);

DirectProcessor<Integer> direct = DirectProcessor.create();


direct.filter(d-> d%2==0).subscribe(d ->System.out.printf("Subscriber1      : %s\n", d));
direct.subscribe(d ->System.out.printf("Subscriber2 : %s\n", d));

c.subscribe(direct);
```

Figure 3.39: Esempio di utilizzo del DirectProcessor

Ottendo quindi in console:

```
Subscriber2 : 1
Subscriber1      : 2
Subscriber2 : 2
Subscriber2 : 3
Subscriber1      : 4
Subscriber2 : 4
Subscriber2 : 5
Subscriber1      : 6
Subscriber2 : 6
Subscriber2 : 7
Subscriber1      : 8
Subscriber2 : 8
Subscriber2 : 9
Subscriber1      : 10
Subscriber2 : 10
Subscriber2 : 11
```

Figure 3.40: Utilizzo del DirectProcessor: la console

Vediamo alcune sue caratteristiche:

- Il Flux “c” è il publisher che fornisce i dati al nostro processor.
- Il directProcessor, quindi si sottoscrive a “c” e riceve da lui ogni elemento.
- Ci sono 2 subscriber che si iscrivono al processor direct, quindi per ogni elemento che il processor riceve, lo distribuisce a ogni subscriber in modo sincrono
- Il Publisher distribuisce un item alla volta a ogni subscriber. Infatti dalla console possiamo notare che il dato viene prima fornito al Subsriber 1 e poi al Subscriber 2.
- Pone delle limitazioni sulla gestione della back-pressure: se infatti un subscriber è più lento dell’altro viene generata un’eccezione.

3.12.2 L’UnicastProcessor

L’UnicastProcessor è tipicamente utilizzato quando si ha bisogno di avere un oggetto in grado di realizzare un “multiplexer” di eventi, ovvero un oggetto che raccoglie in input da sorgenti diverse e emette verso un unico subscriber.

Esso ha la caratteristica di bufferizzare all’interno di una coda gli item ricevuti dai suoi publisher, così che possa inviarli immediatamente al subscriber, se ce ne è già uno iscritto, altrimenti non appena se ne iscrive uno.

Il suo comportamento prevede che:

- L’unicast Processor si può scrivere a multipli flussi, così da ricevere dati da più pulisher.
- Tutti gli item che l’Unicast processor riceve quando nessun Subscriber è iscritto a lui, vengono automaticamente bufferizzati e quindi conservati per il subscriber che si sottoscriverà.
- Può sottoscriversi un solo subscriber altrimenti verrà generata un eccezione di questo tipo: java.lang.IllegalStateException: UnicastProcessor allows only a single Subscriber

Nella documentazione ufficiale il processor Unicast viene definito attraverso l'uso di questo diagramma:

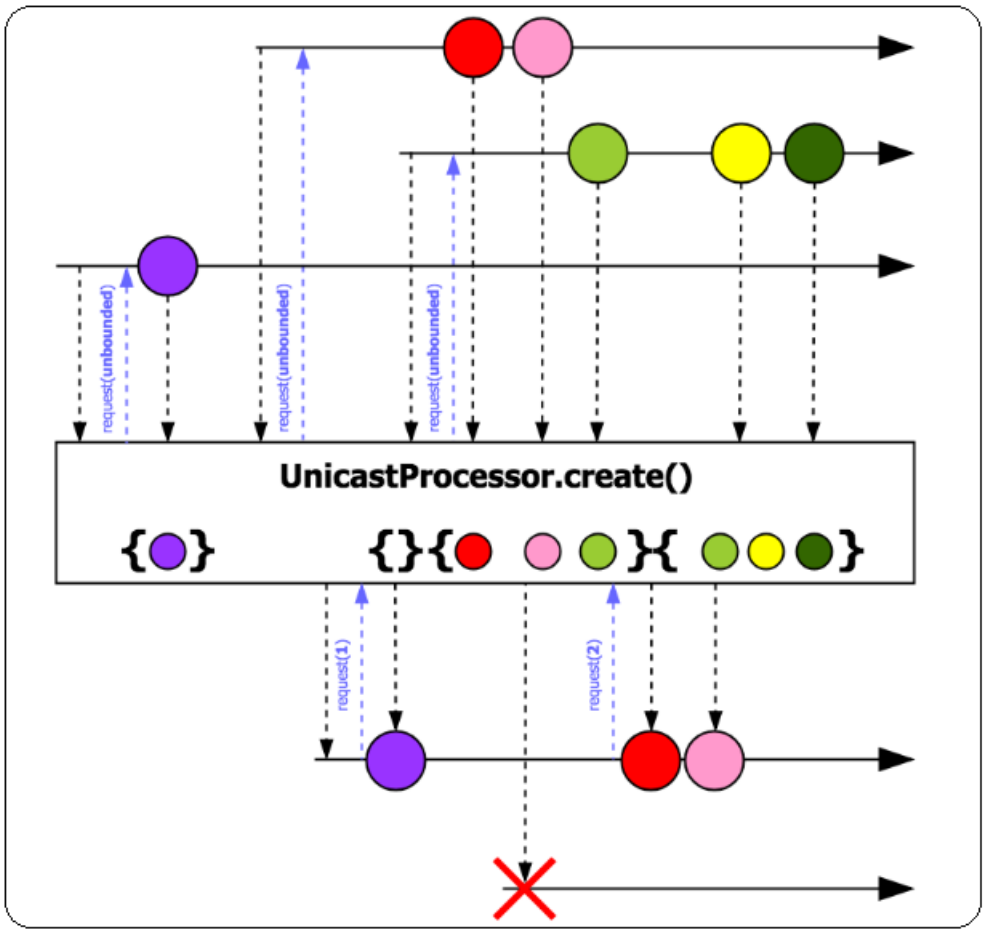


Figure 3.41: Comportamento dell'UnicastProcessor

Attenzione a un problema: Si deve fare attenzione al thread che emette i segnali: infatti se gli item vengono emessi sullo stesso thread esso riceverà gli item da tutti i publisher, altrimenti il thread, non ricevendoli, non potrà emetterli al subscriber.

Un esempio di codice corretto:

```
UnicastProcessor<Integer> unicast = UnicastProcessor.create();

unicast
       .subscribeOn(Schedulers.single())
       .doOnNext(item-> System.out.println("Subscriber: " + item))
       .subscribe();

Flux.range(0, 5)
    .filter(i -> i%2 == 0)            //genera 0, 2, 4
    .subscribe(d-> unicast.onNext(d));

Flux.range(0, 5)
    .filter(i -> i%2 != 0)            //genera 1, 3
    .subscribe(d-> unicast.onNext(d));
```

Figure 3.42: Esempio di codice corretto

Come ci si poteva aspettare, in console otteniamo:

```
Subscriber: 0
Subscriber: 2
Subscriber: 4
Subscriber: 1
Subscriber: 3
```

Figure 3.43: Esempio di codice corretto

In questo caso abbiamo due publisher che generano 5 interi e l'unicastProcessor rappresenta il subscriber di questi ultimi. L'invocazione onNext() sull'oggetto unicast fa si che gli item vengano emessi sullo stesso thread che gestisce l'unicastProcessor.

Vediamo invece ora un'implemnetazione errata:

```
UnicastProcessor<Integer> unicast = UnicastProcessor.create();

unicast
       .subscribeOn(Schedulers.single())
       .doOnNext(item-> System.out.println("Subscriber: " + item))
       .subscribe();

Flux.range(0, 5)
    .filter(i -> i%2 == 0)            //genera 0, 2, 4
    .subscribe(unicast);

Flux.range(0, 5)
    .filter(i -> i%2 != 0)            //genera 1, 3
    .subscribe(unicast);
```

Figure 3.44: Esempio di codice errato

Come possiamo vedere dalla Fig.3.45, eseguendo il flusso più volte potremmo notare che in console possono verificarsi due situazioni: in una ci sono tutti e cinque le stampe che ci aspettavamo (come nella corretta implementazione), ma nell'altra vedremmo solo un risultato parziale. Questo problema rende l'esecuzione non deterministica perché non è detto che gli item vengano emessi sullo stesso thread.

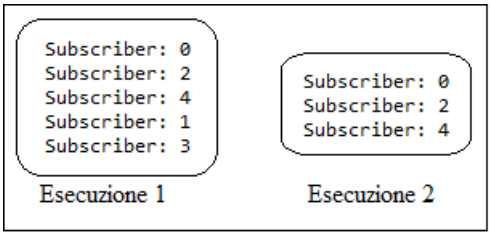


Figure 3.45: Esempio di codice errato

3.12.3 L'EmitterProcessor

L'emitter processor nasce con l'idea di avere un oggetto che si comporta come un "demultiplexer", ovvero in grado di prendere item da una sola sorgente e di distribuirli in modo sincrono bloccante a ogni subscriber iscritto.

Dalla documentazione possiamo vedere il suo comportamento con il diagramma:

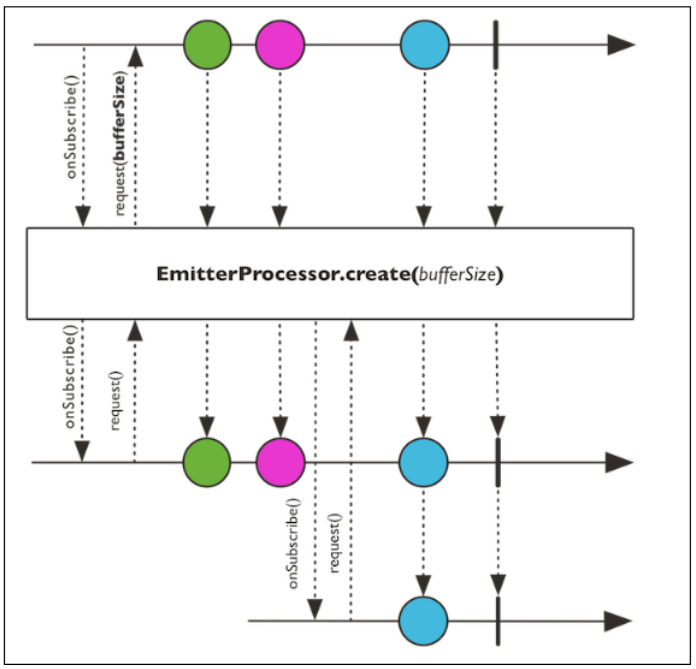


Figure 3.46: Comportamento dell'EmittertProcessor

Vediamo un esempio di utilizzo corretto:

```
EmitterProcessor<Integer> emitter = EmitterProcessor.create();

emitter.log("a").subscribe(d ->System.out.printf("Subscriber1 : %s\n", d));
emitter.log("b").subscribe(d ->System.out.printf("Subscriber2 : %s\n", d));

Flux.range(0, 10)
.filter(i -> i%2 == 0)
.subscribe(d->emitter.onNext(d));          //genera 0, 2, 4, 6, 8, 10
```

Figure 3.47: Esempio di uso corretto dell'EmitterProcessor

Cioè:

1. L'emitterProcessor viene creato con il metodo create.
2. Due subscriber si iscrivono al flusso e rimangono in attesa di item
3. Quando il publisher emette gli item, li propaga in modo sincrono prima al Subscriber1 e poi al Subscriber2. Come possiamo anche vedere in Fig.3.48, il thread che emette onNext(attraverso l'operatore Log) è sempre lo stesso "nio-9000-exec-1".

```
2018-10-24 19:06:24.292  INFO 2280 --- [nio-9000-exec-1] a  : onNext(0)
Subscriber1 : 0
2018-10-24 19:06:24.292  INFO 2280 --- [nio-9000-exec-1] b  : onNext(0)
Subscriber2 : 0
2018-10-24 19:06:24.293  INFO 2280 --- [nio-9000-exec-1] a  : onNext(2)
Subscriber1 : 2
2018-10-24 19:06:24.293  INFO 2280 --- [nio-9000-exec-1] b  : onNext(2)
Subscriber2 : 2
2018-10-24 19:06:24.293  INFO 2280 --- [nio-9000-exec-1] a  : onNext(4)
Subscriber1 : 4
2018-10-24 19:06:24.293  INFO 2280 --- [nio-9000-exec-1] b  : onNext(4)
Subscriber2 : 4
2018-10-24 19:06:24.293  INFO 2280 --- [nio-9000-exec-1] a  : onNext(6)
Subscriber1 : 6
2018-10-24 19:06:24.294  INFO 2280 --- [nio-9000-exec-1] b  : onNext(6)
Subscriber2 : 6
2018-10-24 19:06:24.294  INFO 2280 --- [nio-9000-exec-1] a  : onNext(8)
Subscriber1 : 8
2018-10-24 19:06:24.294  INFO 2280 --- [nio-9000-exec-1] b  : onNext(8)
Subscriber2 : 8
```

Figure 3.48: Esempio corretto: Visualizzazione della consoele

Diverso è però il comportamento quando il produttore di item inizia a generarli prima che i subscriber si siano iscritti al processor. Ciò che succede è che SOLO il primo subscriber riceve tutti gli item, mentre il secondo riesce a iscriversi solo quando ormai tutto il flusso è stato generato, infatti riceverà solo il segnale di onComplete. Questo succede perché l'emitterProcessor è sincrono ed è gestito da un unico thread, quindi appena un Subscriber si iscrive, lui è impegnato a fornirgli gli item e solo dopo vedrà la richiesta di OnSubscribe da parte del secondo sottoscrittore.
Possiamo infatti verificarlo con questo esempio:
Infatti, possiamo verificare quanto appena detto, guardando la console (Fig.4.50):

```
EmitterProcessor<Integer> emitter = EmitterProcessor.create(1);
Flux.range(0, 10)
    .filter(i -> i%2 == 0)
    .subscribe(emitter);            //genera 0, 2, 4, 6, 8

emitter.log("a").subscribe(d ->System.out.printf("Subscriber1 : %s\n", d));
emitter.log("b").subscribe(d ->System.out.printf("Subscriber2 : %s\n", d));
```

Figure 3.49: Esempio particolare di uso dell'emitterProcessor

```
2018-10-25 10:27:31.582  INFO 7048 --- [nio-9000-exec-1] a  : onSubscribe(EmitterProcessor.EmitterInner)
2018-10-25 10:27:31.584  INFO 7048 --- [nio-9000-exec-1] a  : request(unbounded)
2018-10-25 10:27:31.585  INFO 7048 --- [nio-9000-exec-1] a  : onNext(0)
Subscriber1 : 0
2018-10-25 10:27:31.586  INFO 7048 --- [nio-9000-exec-1] a  : onNext(2)
Subscriber1 : 2
2018-10-25 10:27:31.586  INFO 7048 --- [nio-9000-exec-1] a  : onNext(4)
Subscriber1 : 4
2018-10-25 10:27:31.586  INFO 7048 --- [nio-9000-exec-1] a  : onNext(6)
Subscriber1 : 6
2018-10-25 10:27:31.587  INFO 7048 --- [nio-9000-exec-1] a  : onNext(8)
Subscriber1 : 8
2018-10-25 10:27:31.588  INFO 7048 --- [nio-9000-exec-1] a  : onComplete()
2018-10-25 10:27:31.589  INFO 7048 --- [nio-9000-exec-1] b  : onSubscribe(EmitterProcessor.EmitterInner)
2018-10-25 10:27:31.589  INFO 7048 --- [nio-9000-exec-1] b  : request(unbounded)
2018-10-25 10:27:31.589  INFO 7048 --- [nio-9000-exec-1] b  : onComplete()
```

Figure 3.50: Esempio particolare di uso dell'emitterProcessor

Attenzione. Analizziamo invece che cosa succede in quest'altra situazione:

```
EmitterProcessor<Integer> emitter = EmitterProcessor.create(1);
Flux.range(0, 10)
    .filter(i -> i%2 == 0)
    .subscribe(d->emitter.onNext(d));          //genera 0, 2, 4, 6, 8

emitter.log("a").subscribe(d ->System.out.printf("Subscriber1 : %s\n", d));
emitter.log("b").subscribe(d ->System.out.printf("Subscriber2 : %s\n", d));
```

Figure 3.51: Esempio errato di uso del EmitterProcessor

Si ottiene che la richiesta http rimane pendente(Fig.3.52).

Ciò che è stato cambiato rispetto all'ultima versione è solamente il consumatore del flusso. Prima infatti veniva dichiarato Emitter, ora invece viene dichiarato il metodo onNext dell'emitterProcessor. L'unica spiegazione plausibile la si può attribuire al fatto che l'emitter processor sia bloccato sulla onNext() perché al momento non ha sottoscrittori a cui effettivamente inviare quel dato, ma allo stesso tempo quel thread non può invocare il metodo subscribe() perché bloccato sulla OnNext(), realizzando quindi un deadlock.

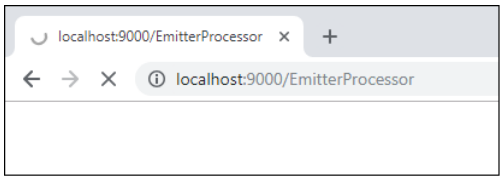


Figure 3.52: Esempio errato di uso del EmitterProcessor

3.12.4 Il ReplayProcessor

Il processor "ReplayProcessor" permette di mantenere una storia sugli ultimi N item ricevuti da un publisher così che possa replicarli (replay) a ogni subscriber che si iscrive.
Vediamo come la documentazione descrive questo processor:

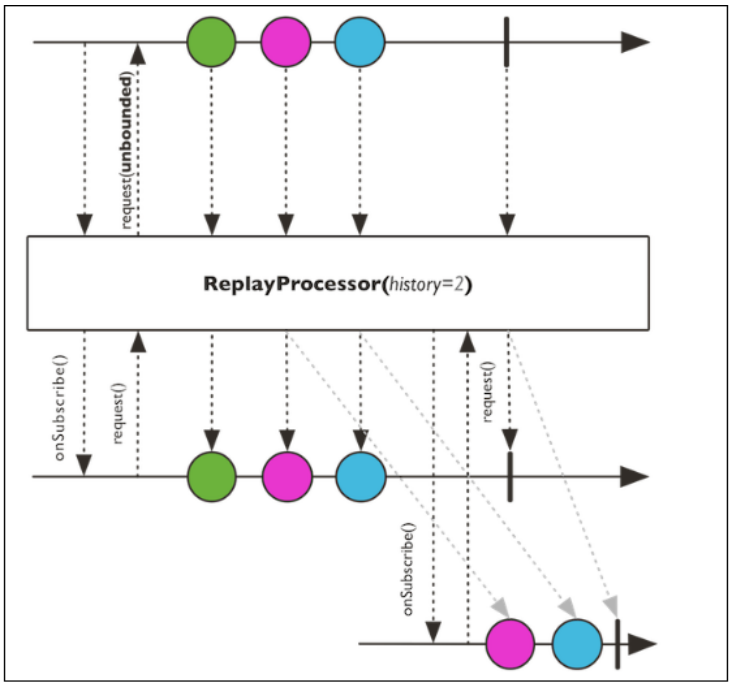


Figure 3.53: Comportamento del ReplayProcessor

Si supponga di utilizzare il ReplayProcessor in cui l'avvio del flusso avviene DOPO che i subscriber si sono iscritti: Come ci possiamo aspettare tutti i subscriber ricevono entrambi i

```
ReplayProcessor<Integer> reply = ReplayProcessor.create(3);

reply.log("a").subscribe(d ->System.out.printf("Subscriber1 : %s\n", d));
reply.log("b").subscribe(d ->System.out.printf("Subscriber2 : %s\n", d));

Flux.range(1, 5)
.map(i -> i*2+1)
.subscribe(d-> reply.onNext(d));            //genera 3, 5, 7, 9, 11
```

Figure 3.54: Esempio 1 di uso del ReplayProcessor

flussi, ottenendo lo stesso cmportamento dell'emitterProcessor:

```
2018-10-25 11:08:15.678  INFO 8276 --- [nio-9000-exec-1] a : | onSubscribe([Fuseable] ReplayProcessor.ReplayInner)
2018-10-25 11:08:15.680  INFO 8276 --- [nio-9000-exec-1] a : | request(unbounded)
2018-10-25 11:08:15.681  INFO 8276 --- [nio-9000-exec-1] b : | onSubscribe([Fuseable] ReplayProcessor.ReplayInner)
2018-10-25 11:08:15.681  INFO 8276 --- [nio-9000-exec-1] b : | request(unbounded)
2018-10-25 11:08:15.684  INFO 8276 --- [nio-9000-exec-1] a : | onNext(3)
Subscriber1 : 3
2018-10-25 11:08:15.685  INFO 8276 --- [nio-9000-exec-1] b : | onNext(3)
Subscriber2 : 3
2018-10-25 11:08:15.685  INFO 8276 --- [nio-9000-exec-1] a : | onNext(5)
Subscriber1 : 5
2018-10-25 11:08:15.685  INFO 8276 --- [nio-9000-exec-1] b : | onNext(5)
Subscriber2 : 5
2018-10-25 11:08:15.685  INFO 8276 --- [nio-9000-exec-1] a : | onNext(7)
Subscriber1 : 7
2018-10-25 11:08:15.686  INFO 8276 --- [nio-9000-exec-1] b : | onNext(7)
Subscriber2 : 7
2018-10-25 11:08:15.686  INFO 8276 --- [nio-9000-exec-1] a : | onNext(9)
Subscriber1 : 9
2018-10-25 11:08:15.686  INFO 8276 --- [nio-9000-exec-1] b : | onNext(9)
Subscriber2 : 9
2018-10-25 11:08:15.686  INFO 8276 --- [nio-9000-exec-1] a : | onNext(11)
Subscriber1 : 11
2018-10-25 11:08:15.686  INFO 8276 --- [nio-9000-exec-1] b : | onNext(11)
Subscriber2 : 11
```

Figure 3.55: Esempio 1 di uso del ReplayProcessor

Il ReplayProcessor fornisce un risultato diverso quando i subscriber si sottoscrivono DOPO l'inizio dell'esecuzione del flusso:

```
ReplayProcessor<Integer> replay = ReplayProcessor.create(3);

Flux.range(1, 5)
.map(i -> i*2+1)
.subscribe(d-> replay.onNext(d));           //genera 3, 5, 7, 9, 11

replay.log("a").subscribe(d ->System.out.printf("Subscriber1 : %s\n", d));
replay.log("b").subscribe(d ->System.out.printf("Subscriber2 : %s\n", d));
```

Figure 3.56: Esempio 2 di uso del ReplayProcessor

Il flusso comincia quando ancora non ci sono sottoscrittori, quindi replayProcessor emettere gli item, ma nessuno li riceverà. Quando i due sottoscrittori riescono a iscriversi riceveranno solamente gli ultimi 3 elementi, dove 3 è il parametro impostato nella create che indica la dimensione del buffer mantenuta dal replayProcessor per mantenerne uno storico (vedi Fig. 3.57)

```
2018-10-25 11:17:41.189  INFO 9312 --- [nio-9000-exec-1] a : | onSubscribe([Fuseable] ReplayProcessor.ReplayInner)
2018-10-25 11:17:41.191  INFO 9312 --- [nio-9000-exec-1] a : | request(unbounded)
2018-10-25 11:17:41.192  INFO 9312 --- [nio-9000-exec-1] a : | onNext(7)
Subscriber1 : 7
2018-10-25 11:17:41.192  INFO 9312 --- [nio-9000-exec-1] a : | onNext(9)
Subscriber1 : 9
2018-10-25 11:17:41.193  INFO 9312 --- [nio-9000-exec-1] a : | onNext(11)
Subscriber1 : 11
2018-10-25 11:17:41.193  INFO 9312 --- [nio-9000-exec-1] b : | onSubscribe([Fuseable] ReplayProcessor.ReplayInner)
2018-10-25 11:17:41.194  INFO 9312 --- [nio-9000-exec-1] b : | request(unbounded)
2018-10-25 11:17:41.194  INFO 9312 --- [nio-9000-exec-1] b : | onNext(7)
Subscriber2 : 7
2018-10-25 11:17:41.194  INFO 9312 --- [nio-9000-exec-1] b : | onNext(9)
Subscriber2 : 9
2018-10-25 11:17:41.195  INFO 9312 --- [nio-9000-exec-1] b : | onNext(11)
Subscriber2 : 11
```

Figure 3.57: Esempio 2 di uso del ReplayProcessor

3.12.5 Il TopicProcessor

Il topicProcessor è un processor asincrono in grado di mantenere e gestire internamente una coda circolare (ringBuffer) per storicizzare gli item. Nella documentazione possiamo vedere che il TopicProcessor può assumere comportamenti diversi a seconda del metodo con cui viene creato. Vediamo i due diagrammi:

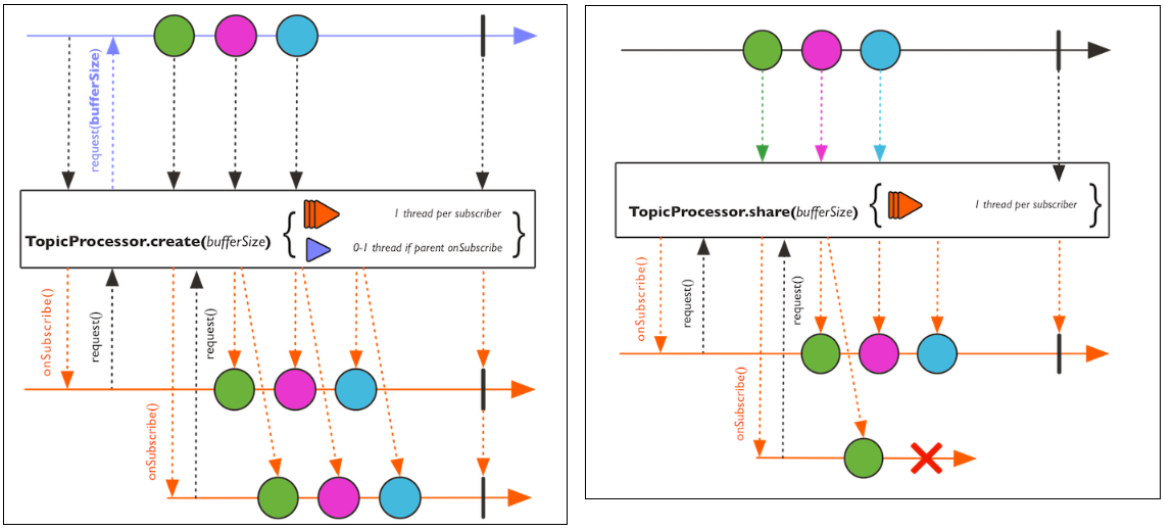


Figure 3.58: Comportamento del TopicProcessor

Il processorTopic creato attraverso il metodo create() permette di gestire un subscriber con un thread diverso, prendendo come parametro i nome del buffer e la sua dimensione che deve necessariamente essere una potenza di due altrimenti verrà generata la seguente eccezione: java.lang.IllegalArgumentException: bufferSize must be a power of 2

Il topicProcessor utilizza un event-loop asincrono per emettere item verso i suoi subscriber ed è autorizzato ad interrompere la pubblicazione degli item non appena rileva un errore senza emettere una onError.
Il topicProcessor può ricevere item:

1. Da un solo produttore, se è stato creato attraverso il metodo create()

2. Da più produttori, se è stato creato attraverso il metodo shared()

Vedimao un esempio di utilizzo del TopicProcessor, ottenuto creandolo attraverso il metodo create():

```
TopicProcessor<Integer> topic = TopicProcessor.create("bufferTopicProcessor", 2);

	topic
		.doOnNext(item-> System.out.println("Subscriber1: " + item + " --> " + Thread.currentThread().getName()))
		.subscribe();

	topic
		.doOnNext(item-> System.out.println("Subscriber2: " + item+ " --> " + Thread.currentThread().getName()))
		.subscribe();

	Flux.range(0, 7)
		.subscribe(topic);
```

Figure 3.59: Esempio di uso del TopicProcessor

Come possiamo vedere dalla Fig.3.60, si ottene un comportamento completamente asincrono, grazie all'uilizzo di più thread.

```
Subscriber1: 0 --> bufferTopicProcessor-1
Subscriber1: 1 --> bufferTopicProcessor-1
Subscriber2: 0 --> bufferTopicProcessor-2
Subscriber2: 1 --> bufferTopicProcessor-2
Subscriber2: 2 --> bufferTopicProcessor-2
Subscriber1: 2 --> bufferTopicProcessor-1
Subscriber2: 3 --> bufferTopicProcessor-2
Subscriber2: 4 --> bufferTopicProcessor-2
Subscriber1: 3 --> bufferTopicProcessor-1
Subscriber1: 4 --> bufferTopicProcessor-1
Subscriber1: 5 --> bufferTopicProcessor-1
Subscriber1: 6 --> bufferTopicProcessor-1
Subscriber2: 5 --> bufferTopicProcessor-2
Subscriber2: 6 --> bufferTopicProcessor-2
```

Figure 3.60: Esempio di uso del TopicProcessor

Attenzione. Cambiando leggermente questo comportamento otteniamo qualcosa di inaspettato. Vediamolo con un esempio:

```
TopicProcessor<Integer> topic = TopicProcessor.create("bufferTopicProcessor", 2);

	topic.onNext(19);
	topic.onNext(11);
	topic.onNext(15);
	topic.onNext(120);

	topic
		.doOnNext(item-> System.out.println("Subscriber1: " + item + " --> " + Thread.currentThread().getName()))
		.subscribe();

	topic
		.doOnNext(item-> System.out.println("Subscriber2: " + item+ " --> " + Thread.currentThread().getName()))
		.subscribe();

	Flux.range(0, 7)
		.log("produttore: ")
		.subscribe(topic);
```

Figure 3.61: Esempio di uso errato del TopicProcessor

Ottenendo in console:

```
Subscriber1: 15 --> bufferTopicProcessor-1
Subscriber1: 120 --> bufferTopicProcessor-1
Subscriber1: 15 --> bufferTopicProcessor-1
Subscriber1: 120 --> bufferTopicProcessor-1
2018-10-25 12:16:54.986  INFO 8612 --- [nio-9000-exec-1] produttore : | onSubscribe([Synchronous Fuseable]
                                                                         FluxRange.RangeSubscription)
2018-10-25 12:16:55.008  INFO 8612 --- [r[request-task]] produttore : | request(2)
2018-10-25 12:16:55.009  INFO 8612 --- [r[request-task]] produttore : | onNext(0)
Subscriber1: 0 --> bufferTopicProcessor-1
2018-10-25 12:16:55.009  INFO 8612 --- [r[request-task]] produttore : | onNext(1)
Subscriber1: 1 --> bufferTopicProcessor-1
2018-10-25 12:16:55.012  INFO 8612 --- [r[request-task]] produttore : | request(4)
2018-10-25 12:16:55.012  INFO 8612 --- [r[request-task]] produttore : | onNext(2)
Subscriber2: 0 --> bufferTopicProcessor-2
Subscriber2: 1 --> bufferTopicProcessor-2
```

Figure 3.62: Esempio di uso errato del TopicProcessor

Cerchiamo di capire cosa è successo:

- L'item 19 e 11 che vengono persi dai due subscriber, ma questo potevamo aspettarcelo perché la sottoscrizione al flusso avviene dopo l'avvio effettivo dello stesso.
- Effettuando esplicitamente il metodo onNext si ottiene che gli item 15 e 120 vengono ricevuti per ben due volte dal subscriber1 mentre il subscriber 2 non li riceve. La ragione per cui questo avviene penso sia sempre un problema del thread che viene invocato per effettuare la onNext che in questo caso è il thread che invia i dati al

subscriber1. Quindi invocare in metodo onNext in questo modo sicuramente non è corretto.

- **Attenzione**: il produttore degli item riesce a fare correttamente la onNext solo degli item 0 e 1, dopodichè prova a fare la onNext dell'item2 ma né il Subscriber1, né il subscriber2 la catturano, quindi non emette più elementi e come possiamo vedere non viene generato alcun segnale di onError.

3.12.6 Il WorkQueueProcessor

Il WorkQueueProcessor è quasi del tutto simile al Topic Processor, solo che se prima i topicProcessor inoltravano i dati a entrambi i subscriber, ora invece ogni item viene dato a uno dei tanti subscriber, cercando quindi di simulare l'algoritmo roundRobin.

Vediamo graficamente come la documentazione di reactor definisce il coportamento di questo processor, distinguendone il comportamento a seconda del metodo di creazione:

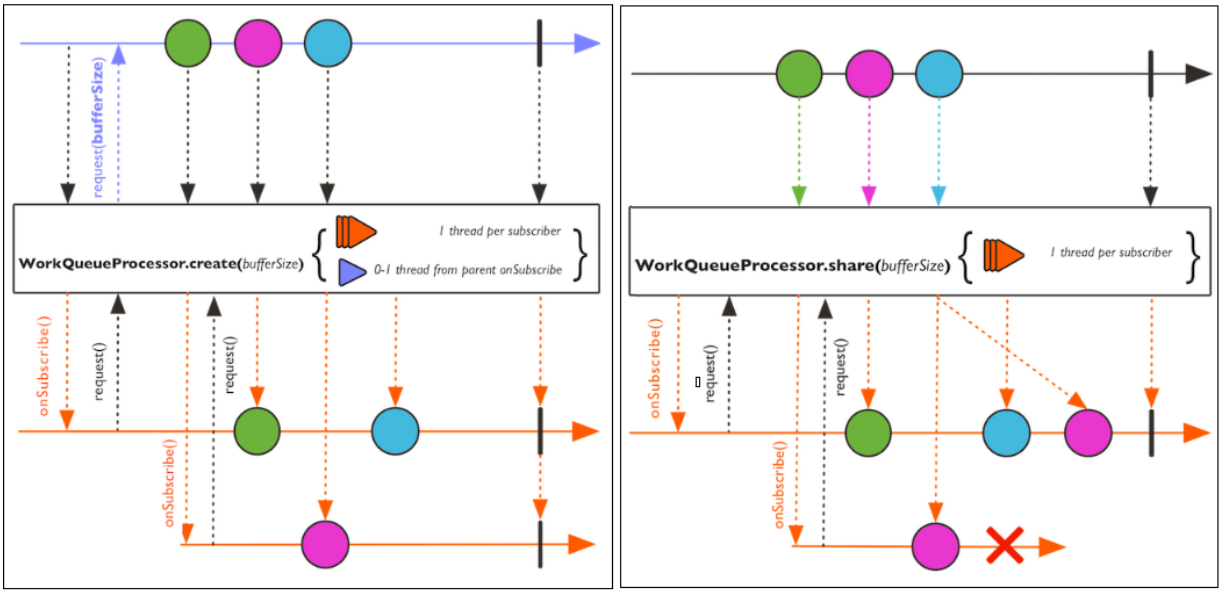


Figure 3.63: Comportamento del WorkQueueProcesor

Possiamo quindi anche notare che al presentarsi di un errore, il processor è in grado di rilevarlo e quindi emettere l'item su un altro subscriber.

Vediamo un esempio di utilizzo, creando il processor attraverso il metodo create:

```
WorkQueueProcessor<Integer> workQueue = WorkQueueProcessor.share("bufferWorkQueueProcessor", 2);

workQueue
    .doOnNext(item-> System.out.println
            ("Subscriber1: " + item + " --> " + Thread.currentThread().getName()))
    .subscribe();

workQueue
    .doOnNext(item-> System.out.println
            ("Subscriber2: " + item+ " --> " + Thread.currentThread().getName()))
    .subscribe();


    Flux.range(0, 7)
    .onErrorReturn(new Integer(-10))
    .subscribe(workQueue);
```

Figure 3.64: Esempio di uso del WorkQueueProcesor

Ottenendo in console, che ogni item viene ricevuto da uno solo dei subscriber:

```
Subscriber2: 1 --> bufferWorkQueueProcessor-2
Subscriber1: 0 --> bufferWorkQueueProcessor-1
Subscriber2: 2 --> bufferWorkQueueProcessor-2
Subscriber1: 3 --> bufferWorkQueueProcessor-1
Subscriber2: 4 --> bufferWorkQueueProcessor-2
Subscriber1: 5 --> bufferWorkQueueProcessor-1
Subscriber2: 6 --> bufferWorkQueueProcessor-2
```

Figure 3.65: Esempio di uso del WorkQueueProcesor

4

Testing

Ho messo a confronto l'approccio bloccante della libreria di Spring MVC con l'approccio reattivo offerto dalla libreria Reactor di WebFlux attraverso un caso d'uso concreto, per verificarne effettivamente i vantaggi, prestando particolare attenzione ai pregi che derivano da una programmazione reattiva.

La programmazione reattiva nasce con l'intento di riuscire a scalare e servire più utenti con la stessa quantità di risorse e diminuire i tempi di lateza quando possibile. Questo è effettivamente il risultato che vorrei verificare.

4.1 Caso D'uso

Si supponga di trovarsi all'interno di un autobus nella città di Torino e di rilevare la salita e la discesa degli utenti su di esso. Queste informazioni vengono mantenute all'interno di un database come segue:

- Identificativo utente: per identificare un'utente si supponga di utilizzarne il MAC del proprio smarphone.
- Identificativo del pullman: ogni autobus viene identificato attraveso la linea a cui appartiene più un numero in sequenza che ne identifica gli autobus della stessa linea.
- Timestamp1: questo rappresenta l'istante in cui l'utente è salito sul pullman.
- Timestamp2: questo rappresenta l'istante in cui l'utente è sceso dal pullman.
- Latitudine e longitudine della fermata di partenza dell'utente
- Latitudine e longitudine Latitudine e longitudine della fermata di arrivo dell'utente

Il modello utilizzato è il seguente:

```
@Document(collection = "packet")
public class Packet
{
   @Id
   private ObjectId id;
   private String busId;
   private String macAddress;
   private GregorianCalendar timestamp1;
   private GregorianCalendar timestamp2;
   private Double lat1;
   private Double lng1;
   private Double lat2;
   private Double lng2;

   public Packet(){}

   public Packet(String macAddress, String busId, GregorianCalendar
       timestamp1, GregorianCalendar timestamp2, Double lat1, Double lng1,
       Double lat2, Double lng2)
   {
      this.macAddress = macAddress;
      this.busId = busId;
      this.timestamp1 = timestamp1;
      this.timestamp2 = timestamp2;
      this.lat1 = lat1;
      this.lng1 = lng1;
      this.lat2 = lat2;
      this.lng2 = lng2;
   }

   //getter and setter...

}
```

Si supponga che venga creata un'applicazione che permetta di vedere quanto frequentemente quel pullman sia popolato in determinati orari.

Si tenga in considerazione che i dati raccolti da ogni pullman per il rilevamento dei passeggeri che salgono e scendono siano migliaia ogni giorno.

Ho quindi realizzato due applicazioni il più possibile identiche: una realizzata con Spring MVC e l'altra con Spring WebFlux, così da confrontarne le prestazioni.

4.2 Generazione della base di dati

Per poter testare l'applicazione ho dovuto generare una base di dati che permettesse di testare le prestazioni delle due applicazioni.

Questo è stato fatto con un modello casuale considerando che l'utente possa aver fatto più salite/discese dallo stesso pullman in vari punti della città, considerando degli orari di salita e discesa realistici.

Il seguente codice è stato utilizzato per generare una base di dati abbastanza grande [circa 50 mila record] per poter testare le performance dei framework.

```
public void generateDB()
{
   //Linea 15: 81 fermate
   Double[][] latLong_Bus15 = {{45.06258,7.62118}, /* ... */ };

   //Linea 10N: 32 fermate
   Double[][] latLong_Bus10N = {{45.10661,7.67982}, /* ... */ };

   //Linea 55: 93 fermate
   Double[][] latLong_Bus55 = {{45.04435,7.61775}, /* .... */ };

   Random random = new Random();
   Double[][] latLong_bus = new Double[100][2];

   int iterableList = 0;
   int size_array = 0;

    int m = 50000;
   for(int i=0;i<m; i++)
   {
       //scelgo in mdo random uno del pullman
      int idBus = random.nextInt(3);
```

```
String IDrouter = "";
switch (idBus)
{
   case 1 :
      latLong_bus = latLong_Bus55;
      IDrouter = "55_ID" + random.nextInt(7);
      size_array = 93;
      break;
   case 2 :
      latLong_bus = latLong_Bus10N;
      IDrouter = "10N_ID" + random.nextInt(7);
      size_array = 32;
      break;
   case 0 :
      latLong_bus = latLong_Bus15;
      IDrouter = "15_ID" + random.nextInt(7);
      size_array = 81;
      break;
}

Generex generexMac = new
    Generex("([0-9]{2}|[A-F]{2})(:([0-9]{2}|[A-F]{2})){5}");
String mac = generexMac.random();

Long timestampSalita = null;
Long timestampDiscesa = null;

GregorianCalendar gregorianSalita = new GregorianCalendar();
GregorianCalendar gregorianDiscesa = new GregorianCalendar();

//0-86399 rappresenta il range in secondi di una giornata dalle ore
    00:00 ale ore 23:59
//1534716000 rappresenta in secondi la giornata del 20/08/2018
timestampSalita = new Long(1534716000 + random.nextInt(86400));
gregorianSalita.setTimeInMillis(timestampSalita*1000);

//fermata in cui l'utente sale:
int n1 = random.nextInt(size_array);
```

```
        //fermata in cui l'utente scende:
        int n2 = random.nextInt(size_array);

        //utilizzo il numero delle fermate per definire il timestamp di
            discesa
        int distanza = Math.abs(n2-n1);

        timestampDiscesa = new Long(timestampSalita + distanza*60);
        gregorianDiscesa.setTimeInMillis(timestampDiscesa*1000);

        mongo.save(new Packet
              (mac,
              IDrouter,
              gregorianSalita,
              gregorianDiscesa,
              latLong_bus[n1][0],
              latLong_bus[n1][1],
              latLong_bus[n2][0],
              latLong_bus[n2][1]));

    }
```

4.3 Configurazione del progetto SpringWebFlux

Il progetto è basato su SpringBoot è stato scaricato dal sito ufficiale di Spring (https://start.spring.io/). Ho strutturato il progetto nel seguente modo:

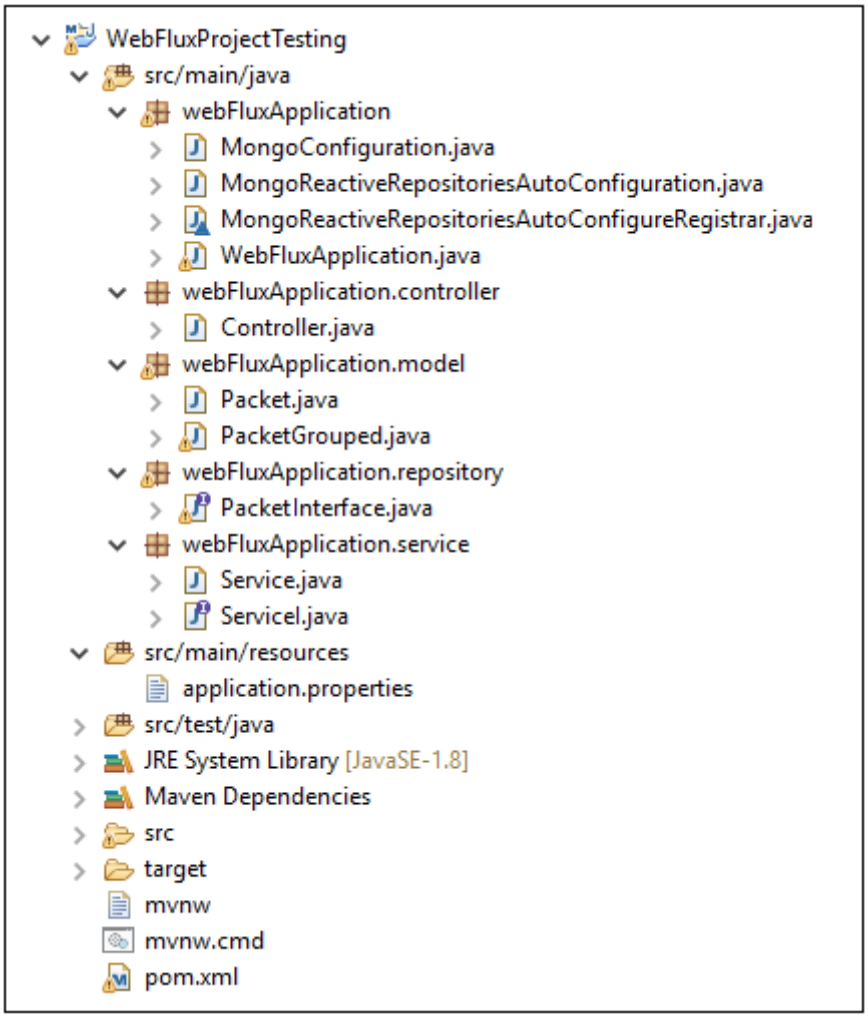


Figure 4.1: Struttura del progettoe

All'interno del progetto ho utilizzato come gestore di pacchetti, maven, che consente di definire nel file pom.xml le dipendenze necessarie.

Quelle usate per questo progetto sono:

```
<!-- Spring Boot -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>

<!-- Spring Data Reactive Repository, per avere il Repository reattivo -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>

<!-- Libreria Reactor -->
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
</dependency>

<!-- Libreria Generx, per la generazione random -->
<dependency>
    <groupId>com.github.mifmif</groupId>
    <artifactId>generex</artifactId>
    <version>1.0.1</version>
</dependency>

<!-- Tomcat -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
```

Per la configurazione dell'intero progetto sono state necessarie quattro classi:

- MongoConfiguration: è una classe all'interno della quale ho configurato l'accesso alla base di dati.
- MongoReactiveRepositoriesAutoConfiguration
- MongoReactiveRepositoriesAutoConfigureRegistrar
- WebFluxApplication, la classe che permette di avviare il server.

4.3.1 La classe WebFluxApplication

La classe WebFluxApplication è quella che permette di avviare il server.

```
@SpringBootApplication()
@EnableReactiveMongoRepositories
@ComponentScan(basePackages = "webFluxApplication")
@EnableWebFlux
public class WebFluxApplication
{
   public static void main(String[] args)
   {
      SpringApplication.run(WebFluxApplication.class, args);
   }

}
```

Le annotazioni utilizzate servono per la configurazione del progetto, vediamole:

- @SpringBootApplication: permete di abilitare l'auto-configurazione di Spring Boot
- @EnableReactiveMongoRepositories: permette di attivare lo scanning sul package per poter abilitare il reactive repository di MongoDB
- @ComponentScan prmette di specificare il package di base da cui partire per fare uno scanning dei @Bean presenti del progetto.
- @EnableWebFlux permette di importare la configurazione Spring Web Flux da WebFlux-ConfigurationSupport permettendo quindi l'uso di controller annotati e functional endpoints.

4.3.2 La classe MongoConfiguration

La classe MongoConfiguration contiene i metodi necessari al framework per contattare la base dati. Nel mio progetto la base di dati scelta è MongoDB, essa:

- è grado di suportare applicazioni attraverso l'uso di chiamate sincrone e asincrone.
- si trova su una macchina Linux differente dall'Application Server virtualizzata attraverso l'utilizzo di Docker.

Possiamo infatti notare che il database è accessibile all'indirizzo "mongodb://192.168.99.100:27017" e il nome del database è "trasporti"

```
@Configuration
@EnableReactiveMongoRepositories
@Import(value = MongoAutoConfiguration.class)
public class MongoConfiguration extends AbstractReactiveMongoConfiguration
{
    @Override
    public @Bean MongoClient reactiveMongoClient() {
        return MongoClients.create("mongodb://192.168.99.100:27017");
    }

    public ReactiveMongoDatabaseFactory mongoDbFactory() {
        return new SimpleReactiveMongoDatabaseFactory(
                          reactiveMongoClient(), getDatabaseName());
    }

    @Bean
    public ReactiveMongoTemplate reactiveMongoTemplate() {
        return new ReactiveMongoTemplate(mongoDbFactory());
    }

    @Override
    protected String getDatabaseName() {
        return "trasporti";
    }
}
```

4.3.3 La classe MongoReactiveRepositoriesAutoConfiguration

La classe MongoReactiveRepositoriesAutoConfiguration è definità all'interno della documentazione ufficiale di Spring ed è in grado di eseguire un auto-configurazione per Spring data reactive Repository specifico per MongoDB, che abilita il comportamento reattivo di MongoDB. Questa classe è del tutto vuota perché la configurazione viene attivata attraverso l'uso di annotazioni.

```
@Configuration
@ConditionalOnClass({MongoClient.class, ReactiveMongoRepository.class})

@ConditionalOnMissingBean({
           ReactiveMongoRepositoryFactoryBean.class,
           ReactiveMongoRepositoryConfigurationExtension.class})

@ConditionalOnProperty(
                  prefix = "spring.data.mongodb.reactiverepositories",
                  name = "enabled",
                  havingValue = "true",
                  matchIfMissing = true)
@Import(MongoReactiveRepositoriesAutoConfigureRegistrar.class)
@AutoConfigureAfter(MongoReactiveDataAutoConfiguration.class)
public class MongoReactiveRepositoriesAutoConfiguration
{

}
```

Cerchiamo di capire il significato delle annotazioni usate:

- @Configuration: indica che la classe dichiara una o più metodi @Bean che potrebbero essere processati dal container Spring affinché siano poi disponibili a runtime.
- @ConditionalOnClass: esprime il vincolo che la configurazione corrente deve essere attivata solo se le classi indicate come parametro vengono trovate nel classpath
- @ConditionalOnMissinBean: esprime il vincolo che la configurazione corrente deve essere attivata solo se mancano i Bean specifici nel BeanFanctory
- @ConditionalOnProperty permette di abilitare la configurazione in base al valore assunto da alcune proprietà:

  - prefix e name indicano la proprietà da verificare. Nel mio caso ho bisogno di verificare che sia "enable" il repository REACTIVE di Spring
  - havingValue e matchIfMissing sono utilizzati per creare dei controlli più avanzati, in particolare:
    * havingValue indica il valore che la proprietà dovrebbe avere.
    * matchIfMissing indica cosa dovrebbe fare il framework se la proprietà non assume il valore specificato. Se matchIfMissing è true, la proprietà verrà creata così che sia disponibile.

- @Import permette di importare classi tipicamente di configurazione (MongoReactiveRepositoriesAutoConfigureRegistrar nel mio progetto)
- @AutoConfigureAfter vuole suggerire che le classi specificate dovrebbero essere autoconfigurate dopo le altre classi di autoconfigurazione

4.3.4 La classe MongoReactiveRepositoriesAutoConfigureRegistrar

La classe MongoReactiveRepositoriesAutoConfigureRegistrar è una classe che definisce il nome di specifiche classi necessarie per la configurazione.

```
class MongoReactiveRepositoriesAutoConfigureRegistrar extends
    AbstractRepositoryConfigurationSourceSupport
{
   @Override
   protected Class<? extends Annotation> getAnnotation(){
      return EnableReactiveMongoRepositories.class;
   }

   @Override
   protected Class<?> getConfiguration(){
      return EnableReactiveMongoRepositoriesConfiguration.class;
   }

   @Override
   protected RepositoryConfigurationExtension
       getRepositoryConfigurationExtension(){
      return new ReactiveMongoRepositoryConfigurationExtension();
   }
```

```
    @EnableReactiveMongoRepositories
    private static class EnableReactiveMongoRepositoriesConfiguration{
    }
}
```

4.4 Il Repository

Per l'accesso alla base di dati ho utilizzato un'implmenetazione di Spring Data Reactive Repositories, così che possa definire solamente l'interfaccia, godendo dei metodi ereditati già implementati della classe astratta estesa, come findAll, findOne, ...

```
public interface PacketInterface extends
                            ReactiveMongoRepository<Packet, String>
{

}
```

Per l'utilizzo di Spring Data Reactive Repository ho aggiunto la dipendenza maven:

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

All'interno del mio progetto ho scelto di non creare una classe che implementi questa interfaccia perché ho volutamente deciso utilizzare solo la findAll per leggere i dati dal database.

Per testare quanto effettivamente il framework Spring WebFlux sia in grado di reagire bene a fronte di un enorme sovraccarico, ho scelto di simularlo utilizzando una mole di dati enorme, evitanto quindi di far eseguire le query al database stesso, ma dando questo compito alla mia applicazione, pertanto ho usato solo la findAll per leggere la base di dati senza alcun filtro nè paginazione.

Proprio come per Spring WebFlux, anche il repository creato con Spring MVC è del tutto simile:

```
@Repository
public interface PacketRepositoryInterface extends
                                    MongoRepository<Packet, String>
{

}
```

Mentre invece, la dipendenza maven è la seguente:

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

4.5 Test 1: Raccolta di informazioni raggruppate

Il primo test che voglio fare è quello di testare l'asincronicità e backpressure tipiche di Spring WebFlux e mostrare con quanta facilità sia possibile scrivere applicazioni reattive.

Vediamo quindi ora:

- Come sono stati scritti i Service e i Controller
- Cosa ci si aspetta teoricamente
- Cosa ho ottenuto sperimentalmente

4.5.1 I Services

Il Service contiene un metodo che permette di contare quanti utenti si trovavano su uno specifico pullman in un determinato istante di tempo.

Nel progetto di Spring WebFlux ho quindi creato il seguente metodo:

```
@Override
public Mono<PacketGrouped> getById(String id)
{
   return Flux.from(packetRepository.findAll())
         .delayElements(Duration.ofNanos(1))
            .filter(f -> f.getBusId().equals(id))
            .filter(f -> (
                   (f.getTimestamp1().getTimeInMillis()/1000)<long1 &&
                       (f.getTimestamp2().getTimeInMillis()/1000)>long1))
            .distinct()
            .count()
            .map(d -> new PacketGrouped(id, d) );
}
```

Invece nel progetto di Spring MVC ho quindi creato il seguente metodo:

```
@Override
public long getById(String id)
{
   return packetRepository.findAll()
         .stream()
         .filter(f -> f.getBusId().equals(id))
         .filter(f -> (
               (f.getTimestamp1().getTimeInMillis()/1000)<long1
               && (f.getTimestamp2().getTimeInMillis()/1000)>long1)
               )
         .distinct()
         .count();
}
```

Come possiamo notare, ho cercato di mantenere i due servizi il più possibile simili, infatti:

- entrambi utilizzano la programmazione funzonale, uno attraverso gli Stream di Java 8, mentre l'altra utilizzando i concetti di Reactive Stream.

- entrambi invocano il metodo findAll() sull interfaccia del Repository ed eseguono loro i filtraggi e il conteggio.

Possiamo però notare una piccola differenza. La classe Stream di Java 8 utilizza come terminatore finale per l'avvio del flusso alcuni operatori, come per esempio "count". Quindi quando tale operatore viene utilizzato non è più possibile eseguire altre operazioni su tale flusso. Questo è il motivo per cui il pacchetto PacketGrouped è nel progetto Spring MVC è stato creato poi nel Controller, ma fondamentalmente entrambi fanno le stesse operazioni.

4.5.2 I Controllers

La classe Controller contiene i metodi che permettono l'invocazione di un servizio REST.

Il Controller in Spring Weblux:

```
@GetMapping(value="/getPacketgrouped", produces = {
    MediaType.APPLICATION_STREAM_JSON_VALUE,
    MediaType.TEXT_EVENT_STREAM_VALUE})
public Flux<PacketGrouped> dispatchGrouped() {
   Flux<PacketGrouped> f1 = Flux.from(service.getById("55_ID0"));
   Flux<PacketGrouped> f2 = Flux.from(service.getById("55_ID1"));
   Flux<PacketGrouped> f3 = Flux.from(service.getById("55_ID3"));
   return f1.mergeWith(f3).mergeWith(f2);
}
```

Il controller in Spring MVC è invece:

```
@GetMapping(value="/getPacketgrouped")
   public List<PacketGrouped> dispatchGrouped()
   {
      Stream<PacketGrouped> s1 = Stream.of(service.getById("55_ID0"))
            .map(s-> new PacketGrouped("55_ID0", s));
      Stream<PacketGrouped> s2 = Stream.of(service.getById("55_ID1"))
            .map(s-> new PacketGrouped("55_ID1", s));
      Stream<PacketGrouped> s3 = Stream.of(service.getById("55_ID3"))
            .map(s-> new PacketGrouped("55_ID3", s));

      return Stream.concat( Stream.concat(s1, s2), (s3))
                  .collect(Collectors.toList());
   }
```

4.5.3 I risultati attesi teoricamente

Dopo quanto detto teoricamente, posso descrivere quindi il modello di gestione dei thread all'interno del framework.

Quindi, se volessimo rappresentare ciò che si ottiene con il codice scritto sopra all interno di Spring MVC, avremmo che:

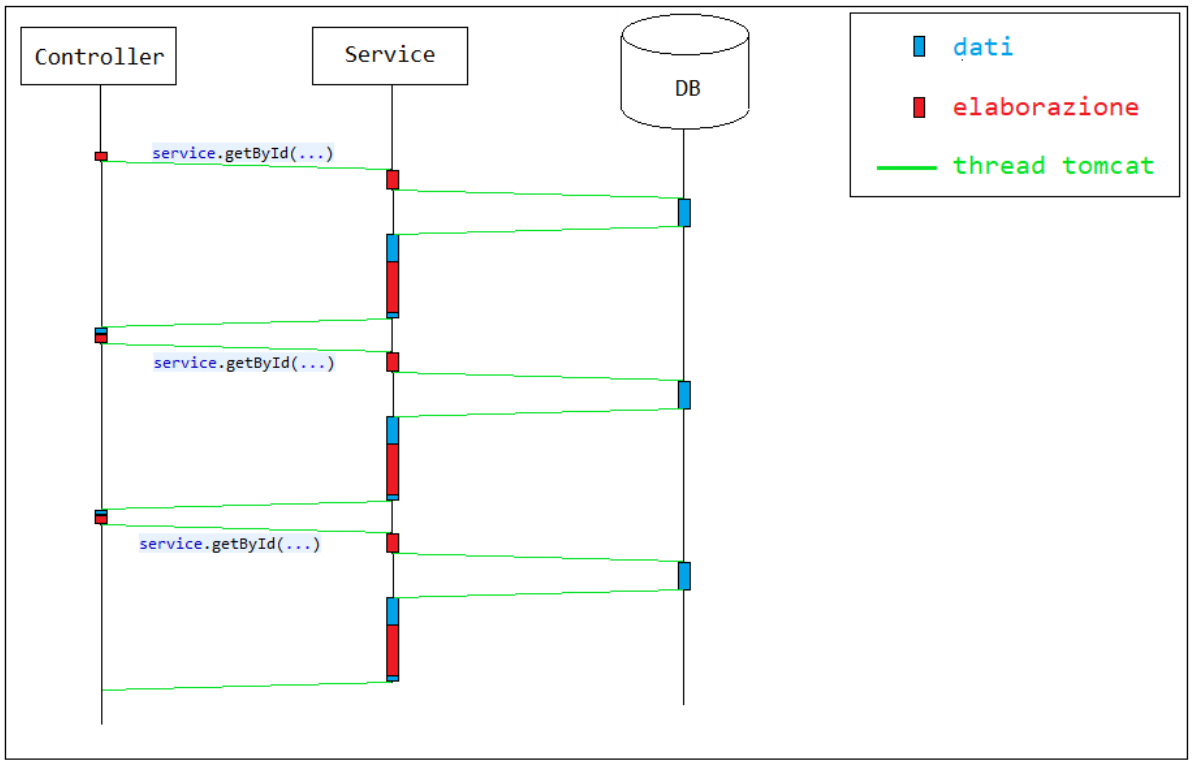


Figure 4.2: Modello Thread ed invocazione delle chiamate in Spring MVC

Ovvero abbiamo che:

- Ogni singola riga di codice all'interno del Controller richiede un operazione che di base è bloccante perché si sta cercando di fare quello che all'interno di una base di dati avremmo risolto con una GroupBy.
- nel modello di programmazione tipica MVC, la programmazione è sincrona, quindi il thread del controller quando invoca un metodo su un Servizio:
  - se esso si trova all'interno dello stesso container (come nella mia applicazione) allora è il thread stesso che va ad eseguire il metodo all'interno del servizio e quindi l'invocazione al DB per ottenere i dati attraverso la findAll.

– se esso invece si trova all'esterno allora rimarrà bloccato nell'attesa che il servizio risponda.

Ma in entrambi i casi il modello risultante, in termini di latenza e tempo di esecuzione della richiesta HTTP, sarà lo stesso.

Cerchiamo invece di capire cosa succede eseguendo lo stesso metodo REST ma all'interno di Spring WebFlux:

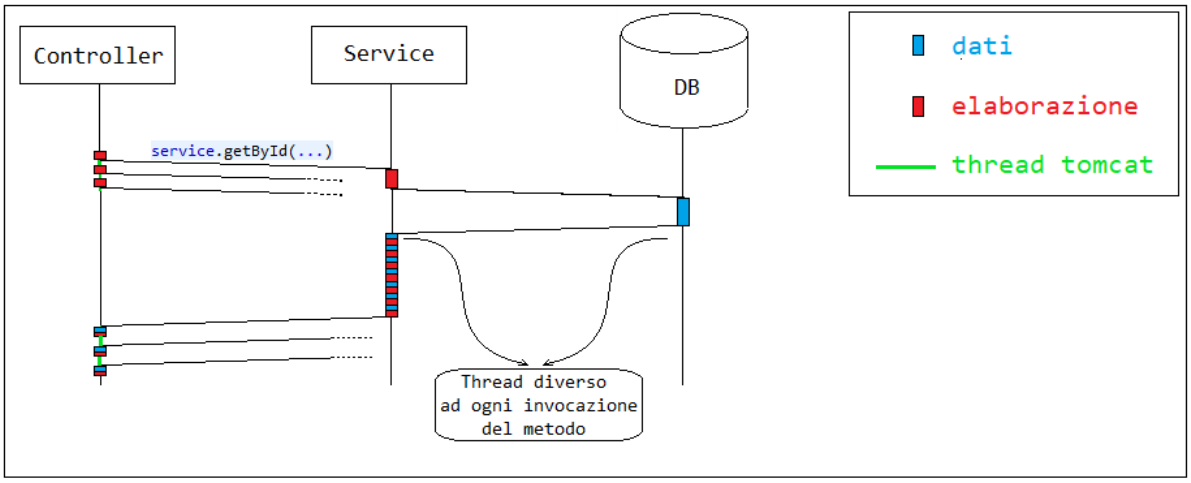


Figure 4.3: Modello Thread ed invocazione delle chiamate in Spring WebFlux

Ovvero:

- Ogni singola riga di codice all'interno del Controller richiede un operazione che di base NON è bloccante, infatti il thread invoca tutti i servizi, uno dopo l'altro senza bloccarsi nell'attesa della risposta. Quando poi arriva sull'operatore di merge si acorge che di non avere i dati necessari per inviare una risposta HTTP, quindi ritorna ad essere disponibile nel thread Pool per altri user, ma non appena uno solo del dati sarà pronto lo invierà al client come un flusso.

- Ogni volta che viene invocato il servizio, esso non è mai eseguito dal thread di Tomcat, ma da un thread diverso. Quindi nel programma che ho scritto avremmo che contemporaneamente 3 thread stanno eseguendo il metodo del service parallelamente.

Ci si aspetta infatti che al crescere del numero di servizi invocati, ad esempio n, le performance di Spring WebFlux abbiano un tempo complessivamente n volte migliore di spring MVC.

4.5.4 I risultati sperimentali

Cercando di verificare quanto teroricamente ho spiegato sopra, vediamo quello che realmente si ottiene.

MVC ha avuto i seguenti risultati:

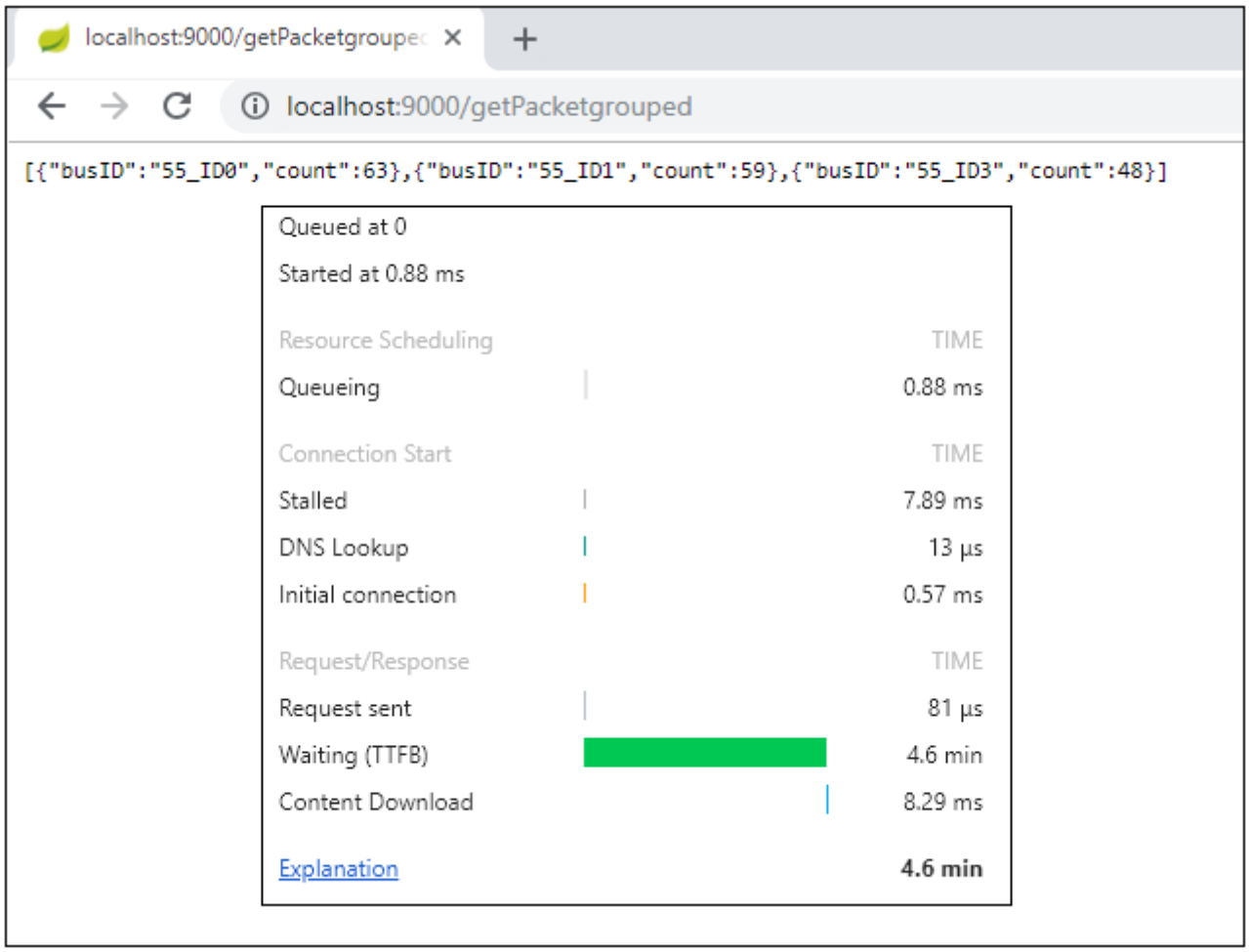


Figure 4.4: Spring MVC:Risposta Http e Tempi di risposta

E possiamo anche notare che è davvero il thread di Tomcat ad arrivare a richiedere l'apertura di UNA connessione al DB. Possiamo vederlo dalla Console:

```
[nio-9000-exec-1] org.mongodb.driver.connection          : Opened connection [connectionId
                                                        {localValue:2, serverValue:15}] to 192.168.99.100:27017
```

Figure 4.5: Connessione al DB in Spring MVC

WebFlux ha avuto i seguenti risultati:

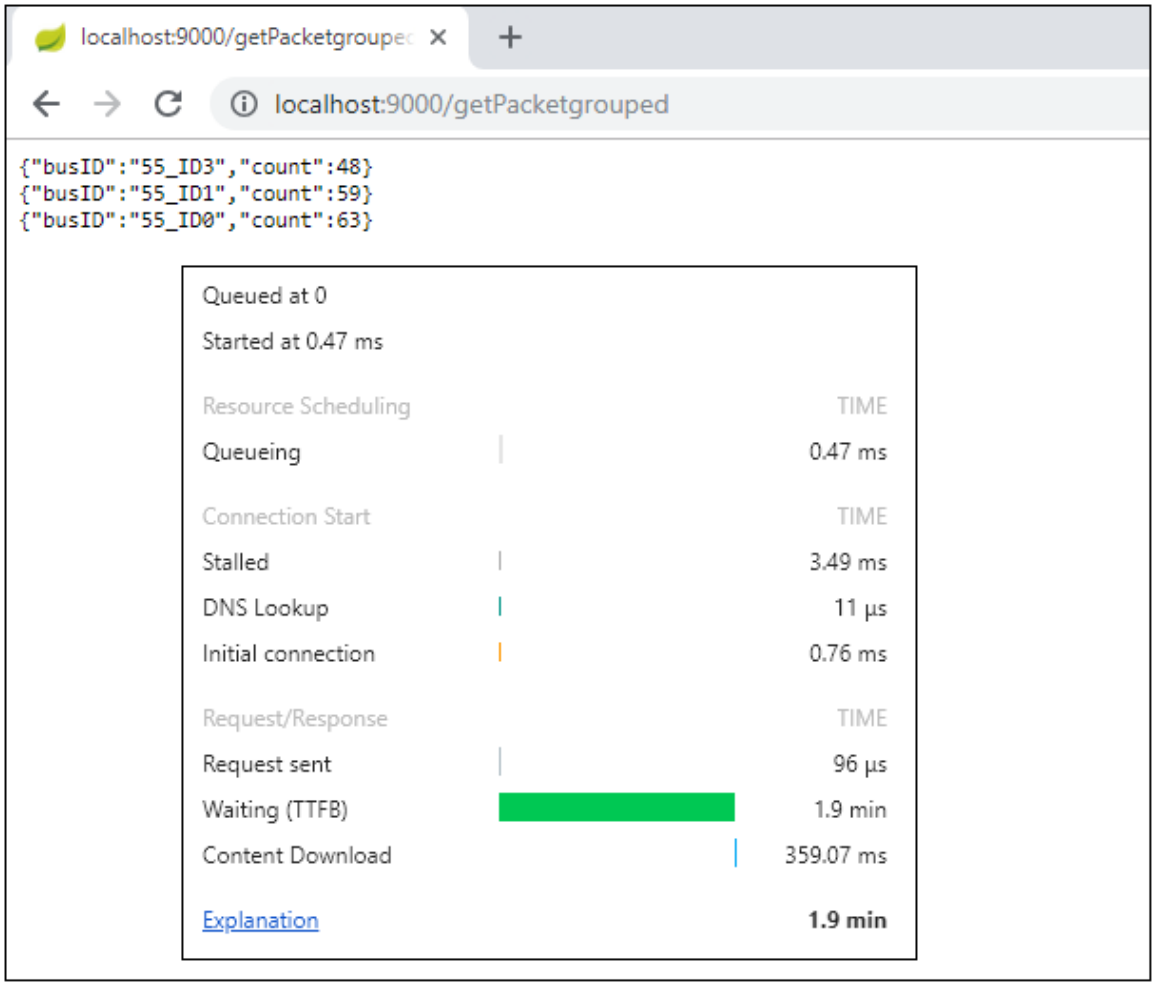


Figure 4.6: Spring WebFlux: Risposta Http e Tempi di risposta

Possiamo anche qui verificare, come spiegato teoricamente, che NON è il thread di Tomcat ad arrivare a richiedere l'apertura di una connessione al DB. Inoltre viene aperta più di una connessione, perchè è più di un thread che contemporaneamente ne ha bisogno. Possiamo vederlo dalla Console:

```
[  Thread-9] org.mongodb.driver.connection          : Opened connection [connectionId
{localValue:3, serverValue:9}] to 192.168.99.100:27017
[ Thread-10] org.mongodb.driver.connection          : Opened connection [connectionId
{localValue:4, serverValue:10}] to 192.168.99.100:27017
[ Thread-11] org.mongodb.driver.connection          : Opened connection [connectionId
{localValue:5, serverValue:11}] to 192.168.99.100:27017
```

Figure 4.7: Connessione al DB in Spring WebFlux

Inoltre ho provato a vedere quanto tempo entrambi ci mettessero a eseguire la count sul solo busID `55_ID0` e ho ottenuto esattamente quanto detto teoricamente:

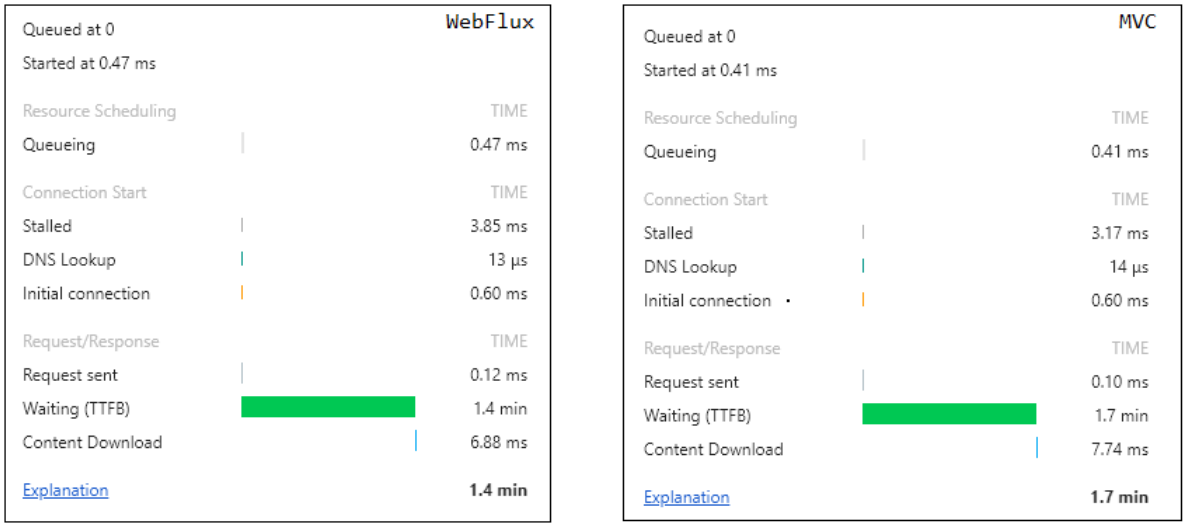


Figure 4.8: Tempi delle risposte Http di Spring e WebFlux a confronto

Possiamo infatti notare che:

- MVC per eseguire UN conteggio ci ha messo 1.7 min, mentre per eseguirne 3 ne ha impiegati 4.7 min, che è quasi 3 volte tanto.
- WebFlux per eseguire UN ci ha messo 1.4 min, mentre per eseguirne 3 ne ha impiegati 1.9 minuti!

Questo dimostra quanto dichiarato teoricamente alla Sez.2.3, quando ho scritto: "Il tempo complessivo di invio della risposta non è pari alla somma dei tempi di ogni singola operazione (questo lo sarebbe con la programmazione sincrona) ma sarebbe pari al tempo richiesto dall'operazione più lunga più un piccolo delta di elaborazione e trasmissione della richiesta http".

4.6 Test 2: Raccolta e informazioni sui dati

Lo scopo di questo test è dimostrare quanto rapidamente il framework SpringWebFlux è in grado di gestire meglio le risorse che possiede per ottenere migliori performance e tempi di latenza incredibimente inferiori.

4.6.1 I Services

La richiesta Http che verrà eseguita dai client per il test è la stessa di prima, ma invece di eseguire una count, vengono ritornate le informazioni di ogni elemento raccolto che soddisfa i filtri inseriti.

In Spring MVC il servizio utilizzato, ritorna una lista di Packet:

```
@Override
public List<Packet> getListById(String id)
{
   return packetRepository.findAll()
         .stream()
         .filter(f -> f.getBusId().contains(id))
         .filter(f -> ((f.getTimestamp1().getTimeInMillis()/1000)<long1
              && (f.getTimestamp2().getTimeInMillis()/1000)>long1))
         .distinct()
         .collect(Collectors.toList());
}
```

In Spring WebFlux invece non vengono utilizzate le liste ma le API reattive di Reactive Stream, ovvero il seguente servizio:

```
@Override
public Flux<Packet> getFluxById(String id)
{
   return Flux.from(packetRepository.findAll())
         .filter(f -> f.getBusId().contains(id))
         .filter(f -> ((f.getTimestamp1().getTimeInMillis()/1000)<long1 &&
            (f.getTimestamp2().getTimeInMillis()/1000)>long1))
         .distinct();
}
```

4.6.2 I Controllers

La classe Controller contiene i metodi che permettono l'invocazione di un servizio REST.

Il Controller in Spring Weblux:

```
@GetMapping(value="/getPacket", produces =
    {MediaType.APPLICATION_STREAM_JSON_VALUE,
    MediaType.TEXT_EVENT_STREAM_VALUE})
public Flux<Packet> dispatch()
{
   return Flux.from( service.getFluxById("15_ID"));
}
```

Il Controller in Spring MVC è invece:

```
@GetMapping(value="/getPacket")
public List<List<Packet>> dispatchPacket()
{
   return
       Stream.of(service.getListById("15_ID")).collect(Collectors.toList());
}
```

4.6.3 I Risultati attesi teoricamente

Con questo test voglio esattamente verificare quello di cui parlavo quando nella Sezione 2.3 ho spiegato il concetto basilare della programmazione Reattiva.

Confrontiamo le due immagni già spiegate precedentemente, notando quanto effettivamente sia la latenza ch il tempo compressivo di risposta siano nttamente inferiori.

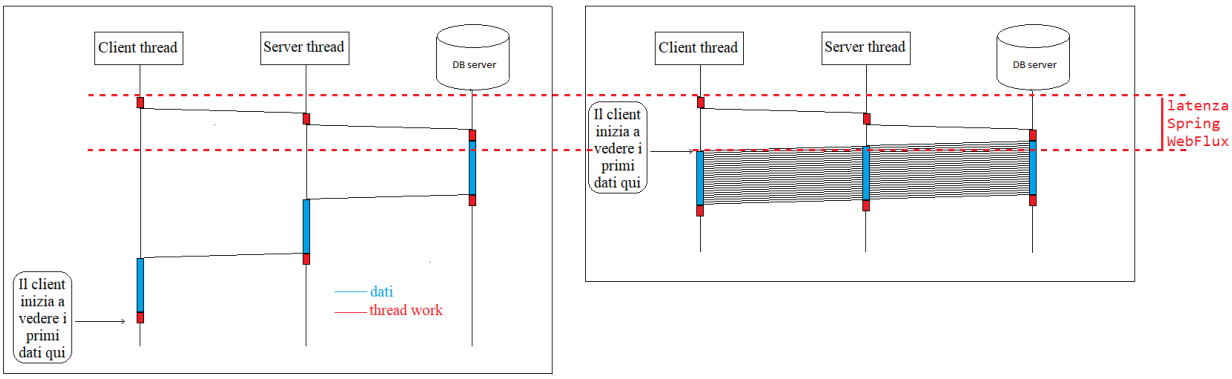


Figure 4.9: Tipiche chiamate asincrone

4.6.4 I Risulti Sperimentali

Sperimentalmente ho effetuato due tipologie di test:

- Test sui tempi di risposta di una sola richiesta HTTP
- Test sui tempi di risposta su n richieste HTTP invocabili attraverso una rampa di T secondi.

Considerando una sola richiesta HTTP il risultato è stato il seguente:

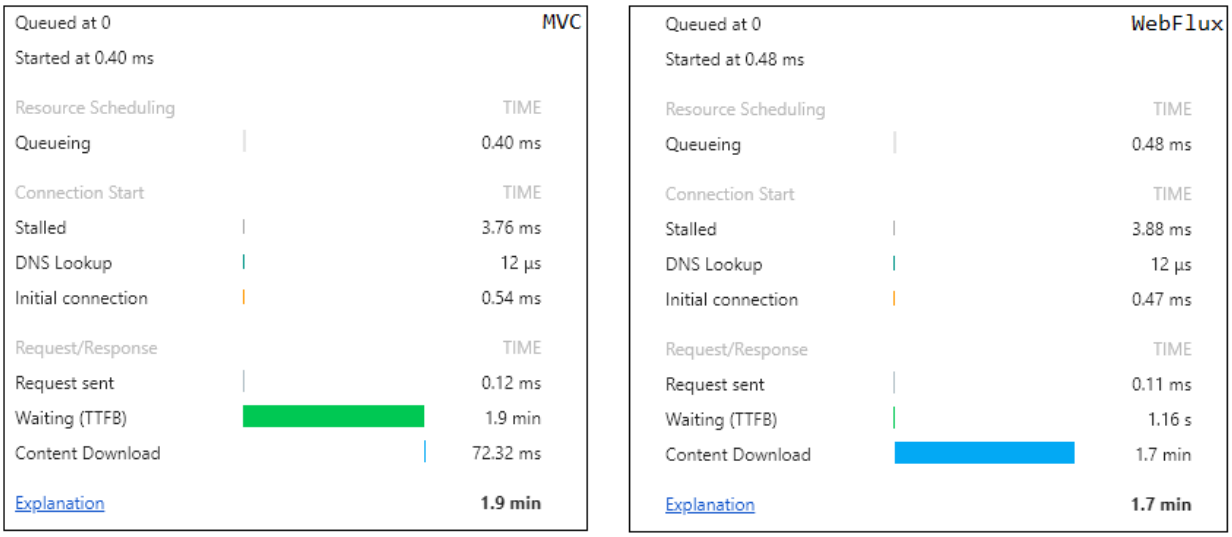


Figure 4.10: Risultati della singola richiesta Http

A conferma di quanto teoricamente ci si aspettava, le differenze si trovano nel:

- tempo di latenza di 1.9 minuti di MVC a confronto dei 1.16 secondi di Spring WebFlux.
- tempo complessivo che è abbastanza paragonabile ma pur sempre inferiore in Spring WebFlux

Si deve anche tenere conto che tali risultati si ottengono perché le operazioni sono effettuate su una quantità di dati molto grande e su operazioni che non sono bloccanti.

Per enfatizzare meglio quanto WebFlux sia migliore in questo tipo di contesto, possiamo fare un ulteriore test utilizzando Gatling e Jmeter. Entrambi sono dei tool che permettono di effettuare un insieme di richieste Http allo stesso server in un certo intervallo di tempo. Ne ho utilizzati due per verificare che l'uso dell'uno o dell'altro non cambi il risultato, anche se qui di seguito mostrerò solamente il risultato ottenuto da Jmeter.

Ho utilzzato Jmeter per effettuare 3 richieste HTTP in un intervallo totale di 3 secondi, quindi una richiesta al secondo, e i risultati ottenuti sono stati già significativi: una delle tre richieste HTTP effettuate a Spring MVC è andata in KO:

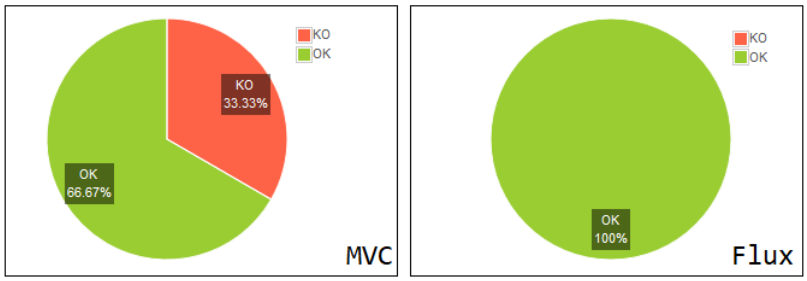


Figure 4.11: Risultati delle 3 richieste HTTP

L'errore generato dal server per l'applicazione Spring MVC è stato un OutOfMemory:

```
java.lang.OutOfMemoryError: GC overhead limit exceeded
        at org.springframework.data.mongodb.core.convert.CustomConversions.getCustomReadTarget(CustomConversions.java:327) ~[spring-data-mongodb-1.10.9.RELEASE.jar:na]
        at org.springframework.data.mongodb.core.convert.CustomConversions.hasCustomReadTarget(CustomConversions.java:310) ~[spring-data-mongodb-1.10.9.RELEASE.jar:na]
        at org.springframework.data.mongodb.core.convert.MappingMongoConverter.getPotentiallyConvertedSimpleRead(MappingMongoConverter.java:824) ~[spring-data-mongodb-1.10.9.RELEASE.jar:na]
        at org.springframework.data.mongodb.core.convert.MappingMongoConverter.readCollectionOrArray(MappingMongoConverter.java:926) ~[spring-data-mongodb-1.10.9.RELEASE.jar:na]
        at org.springframework.data.mongodb.core.convert.MappingMongoConverter.readValue(MappingMongoConverter.java:1220) ~[spring-data-mongodb-1.10.9.RELEASE.jar:na]
        at org.springframework.data.mongodb.core.convert.MappingMongoConverter.access$200(MappingMongoConverter.java:85) ~[spring-data-mongodb-1.10.9.RELEASE.jar:na]
        ........
```

Figure 4.12: Errore Server Tomcat in Spring MVC

Purtroppo l'OutOfMemory è un problema non sempre risolvibile, infatti si possono utilizzare solamente soluzioni palliative, come per esempio l'uso della paginazione a livello sw oppure aumentare la quantità di memoria nel Server. Anche qui abbiamo verificato effettivamente che Spring MVC ha bisogno di mantenere i dati per più tempo in memoria non riuscendo quindi a reggere un carico eccessivo. Spring WebFlux invece si libera subito di ogni singolo dato senza dover attendere di ricevere l'intera lista dal produttore.

Durante il test ho anche effettuato un monitoraggio delle risorse utilizzate: CPU, Memoria e Disco. Questa è una fotografia efettuata in un istante di tempo ma in realtà durante il test è stata pressoché costante e possiamo anche qui notare che in ogni caso WebFlux le gestiva meglio:

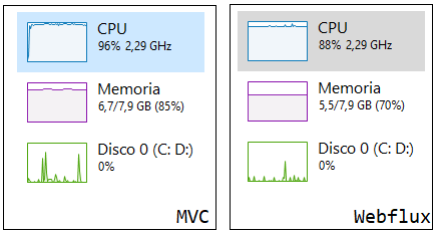
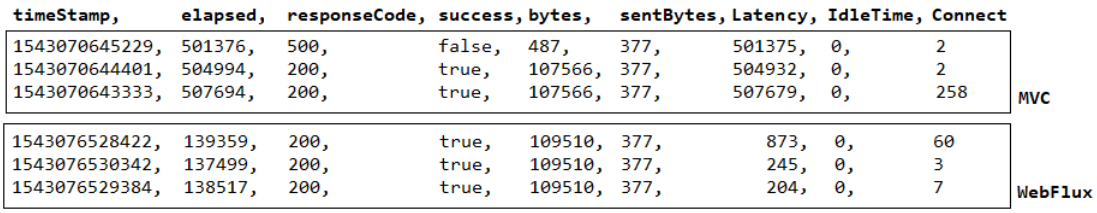


Figure 4.13: Confronto delle risorse utilizzate

I risultati ottenuti sono stati i seguenti:



timeStamp,	elapsed,	responseCode,	success,	bytes,	sentBytes,	Latency,	IdleTime,	Connect	
1543070645229,	501376,	500,	false,	487,	377,	501375,	0,	2	
1543070644401,	504994,	200,	true,	107566,	377,	504932,	0,	2	
1543070643333,	507694,	200,	true,	107566,	377,	507679,	0,	258	MVC
1543076528422,	139359,	200,	true,	109510,	377,	873,	0,	60	
1543076530342,	137499,	200,	true,	109510,	377,	245,	0,	3	
1543076529384,	138517,	200,	true,	109510,	377,	204,	0,	7	WebFlux

Figure 4.14: Confronto dei risultati ottenuti [ms]

Dove possiamo notare:

- i tempi di latenza:
  - WebFlux: meno di un secondo
  - MVC: 8 minuti e mezzo circa
- i tempi complessivi di ogni risposta:
  - WebFlux: un minuto e mezzo circa
  - MVC: 8 minuti e mezzo circa

5

Conclusioni

L'utilizzo della programmazione reattiva porta benefici enormi in situazioni ben specifiche. Essa infatti non è adatta a ogni situazione. Il confronto fatto tra i due framework non vuole evidenziare che WebFlux sia sempre migliore di MVC. Infatti ognuno ha prestazioni migliori rispetto all'altro a seconda del contesto in cui viene utilizzato. Quando si inizia a costruire una WebApplication, bisogna quindi rendersi conto se e quando conviene l'utilizzo di uno o dell'altro.

Ecco quindi che, a seguito di quanto ho studiato posso quindi affermare quando questa tecnologia porta dei benifici migliori e quali:

1. Quando utilizzare reactive?
    (a) Solo in contesti in cui le operazioni sono per lo più non bloccanti, altrimenti si potrebbero ottenere prestazioni verosimilmente identiche se non leggermente peggiorative considerando che l'attivazione di thread richiede un costo maggiore
    (b) Solo se anche il client è reattivo. Se il client non è reattivo, rimarrà bloccato nell'attesa di ricevere tutti i dati prima di visualizzarli all'utente, quindi se ne perderebbero una gran parte dei benefici.
    (c) Solo se la quantità di dati da elaborare o inviare al Client è davvero grande.
2. Quali benifici porta?
    (a) Permette di ridurre i tempi di latenza (Sez.4.6.3) perché non appena il primo item è disponibile viene subito inviato al client senza dover attendere che l'intero set di dati si disponibile.
    (b) Permette di ottenere un tempo complessivo di invio della risposta che non è pari alla somma dei tempi di ogni singola operazione (questo lo sarebbe con la

programmazione sincrona) ma è pari al tempo richiesto dall'operazione più lunga, più un piccolo delta di elaborazione e trasmissione della richiesta http

(c) Permette di gestire al meglio la gestione asincrona di scambio dati tra produttore e consumatore evitando quindi che i thread si blocchino nell'attesa o che un thread sovraccarichi l'altro a causa delle diverse velocità.

(d) Permette il riutilizzo dei flussi, che purtroppo Java 8 non lo rendeva possibile

(e) Utilizza la programmazione funzionale, con tutti i benefici che ne derivano

(f) Evita l'uso di callback o ComputableFuture che rendono la scrittura della programmazione asincrona critica, ma fornisce al programmatore metodi e oggetti asincroni, così che non debba preoccuparsi lui di implementarla!

(g) Permette la gestione di un maggior numero di utenti grazie alla completa asincronicità del server e senza alcuno sforzo del programmatore.

Bibliography

[1] *Il Manifesto dei Sistemi Reattivi*
Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson.
https://www.reactivemanifesto.org/it

[2] *Reactive Streams*
http://www.reactive-streams.org/

[3] *The Reactive Manifesto*
Reza Samei
https://www.slideshare.net/RezaSamee/the-reactive-manifesto-49897385

[4] *Reactive Streams*
https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.2/README.md

[5] *Java 9: Learning the new features - Part 3*
https://alexandreesl.com/2017/11/16/java-9-learning-the-new-features-part-3/

[6] *Reactive Streams in Java 9*
Justin Albano
https://dzone.com/articles/reactive-streams-in-java-9

[7] *Java 9 new Features: Reactive Streams*
Mohammed Aboullaite
https://aboullaite.me/java-9-new-features-reactive-streams/

[8] *Understanding Reactor Pattern: Thread-Based and Event-Driven*
Rafael Salerno
https://dzone.com/articles/understanding-reactor-pattern-thread-based-and-eve

[9] *Doing Reactive Programming with Spring 5*
Eugen Paraschiv
https://stackify.com/reactive-spring-5/

[10] *Java 8 - Streams*
https://www.tutorialspoint.com/java8/java8_streams.htm

[11] *Reactor 3 Reference Guide*
Stephane Maldini, Simon Baslé
https://projectreactor.io/docs/core/release/reference/

[12] *SubscribeOn and publishOn operators in Reactor*
Zoltan Altfatter
https://zoltanaltfatter.com/2018/08/26/subscribeOn-publishOn-in-Reactor/

[13] *Spring Data Reactive Repositories with MongoDB*
https://www.baeldung.com/spring-data-mongodb-reactive

[14] *MongoReactiveRepositoriesAutoConfigureRegistrar.java*
Mark Paluch
https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/mongo/MongoReactiveRepositoriesAutoConfigureRegistrar.java

[15] *Class MongoReactiveRepositoriesAutoConfiguration*
https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/data/mongo/MongoReactiveRepositoriesAutoConfiguration.html

[16] *The magic behind the magic: Spring Boot Autoconfiguration*
Mohammed Aboullaite
https://aboullaite.me/the-magic-behind-the-magic-spring-boot-autoconfiguration/

[17] *Reactive systems using Reactor*
Matt Sicker
https://musigma.blog/2016/11/21/reactor.html

[18] *Reactive RESTful service with Spring 5, Spring Boot 2 and MongoDB (part 1)*
Djallal Serradji
https://dserradji.wordpress.com/2017/04/28/reactive-restful-service-with-spring-5-spring-boot-2-and-mongodb-part-1/

[19] *Advanced Reactive Java*
Dávid Karnok
https://akarnokd.blogspot.com/2016/03/operator-fusion-part-1.html

[20] *Going reactive with Spring Data*
Mark Paluch
https://spring.io/blog/2016/11/28/going-reactive-with-spring-data

[21] *Understanding Reactive types*
Sébastien Deleuze
https://spring.io/blog/2016/04/19/understanding-reactive-types

[22] *Notes on Reactive Programming Part I: The Reactive Landscape*
Dave Syer
https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape

[23] *Notes on Reactive Programming Part II: Writing Some Code*
Dave Syer
https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code

[24] *Notes on Reactive Programming Part III: A Simple HTTP Server Application*
Dave Syer
https://spring.io/blog/2016/07/20/notes-on-reactive-programming-part-iii-a-simple-http-server-application

[25] *Functional Reactive Programming - Streams on steroids*
Wojciech Marusarz
https://www.nexocode.com/blog/posts/reactive-programming/

[26] *Reactor Core*
https://github.com/reactor/reactor-core

[27] *JMETER VS GATLING TOOL*
Jérôme Loisel
https://octoperf.com/blog/2015/06/08/jmeter-vs-gatling/

[28] *Spring, Reactor and Elasticsearch: from callbacks to reactive streams*
Tomasz Nurkiewicz
https://www.nurkiewicz.com/2018/01/spring-reactor-and-elasticsearch-from.html

[29] *The introduction to Reactive Programming you've been missing*
André Staltz
https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

[30] *Testing RxJava*
Andres Almiray
https://www.infoq.com/articles/Testing-RxJava

[31] *Reactor by Example*
Simon Baslé
https://www.infoq.com/articles/reactor-by-example

[32] *RXJava by Example*
Victor Grazi
https://www.infoq.com/articles/rxjava-by-example

[33] *The Reactive Web approach – Introduction*
Moisés Macero
https://thepracticaldeveloper.com/2017/11/04/full-reactive-stack-introduction/Introduction_Reactive_Web_and_The_demo_application

[34] *Full Reactive Stack Backend: WebFlux, Reactive MongoDB and Spring Boot*
Moisés Macero
https://thepracticaldeveloper.com/2017/11/04/full-reactive-stack-with-spring-webflux-and-angularjs/

[35] *Full Reactive Stack: Conclusions and Comparison between WebFlux and MVC*
Moisés Macero
https://thepracticaldeveloper.com/2018/01/02/full-reactive-stack-conclusions/

[36] *Zuul 2 : The Netflix Journey to Asynchronous, Non-Blocking Systems*
https://medium.com/netflix-techblog/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c

[37] *SpringData Reactive MongoDB Repositories | SpringBoot*
http://javasampleapproach.com/reactive-programming/springdata-reactive-mongodb-repositories-springboot

[38] *Developers guide to Webflux*
Adam Szabo
https://www.kotlindevelopment.com/kotlin-webflux/

[39] *Building Reactive Rest APIs with Spring WebFlux and Reactive MongoDB*
Rajeev Kumar Singh
https://www.callicoder.com/reactive-rest-apis-spring-webflux-reactive-mongo/

[40] *Servlet vs. Reactive: Choosing the Right Stack - Rossen Stoyanchev Presents at QCon SF 2017*
Amit K Gupta
https://www.infoq.com/news/2017/12/servlet-reactive-stack

[41] *Reactive programming vs. Reactive systems*
Jonas Bonér and Viktor Klang
https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems

[42] *Reactive Java Performance Comparison*
Rene Schakmann
https://tech.willhaben.at/reactive-java-performance-comparison-c4d248c8d21f

[43] *Reactive Programming 101*
Simon Baslé
http://next.projectreactor.io/learn

[44] *Processing streaming data with Spring WebFlux*
Nithin Mallya
https://medium.com/@nithinmallya4/processing-streaming-data-with-spring-webflux-ed0fc68a14de

[45] *An Introduction to Functional Reactive Programming*
Dan Lew
http://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/

[46] *Spring 5 WebClient*
https://www.baeldung.com/spring-5-webclient

[47] *5 Things to Know About Reactive Programming*
Clement Escoffier
https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/

[48] *WebFlux framework*
https://docs.spring.io/spring-framework/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/web-reactive.html

[49] *Spring Data MongoDB - Reference Documentation*
Mark Pollack, Thomas Risberg, Oliver Gierke, Costin Leau, Jon Brisbin, Thomas Darimont, Christoph Strobl, Mark Paluch, Jay Bryant
https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/mongo.reactive

[50] *Testing Reactive Microservices With Spring-Webflux*
https://workwiththebest.intraway.com/white-paper/Testing-Reactive-Microservices-With-Spring-Webflux/

[51] *Reactive vs. Synchronous Performance Test with Spring Boot 2.0*
Gonçalo Trincao Cunha
https://dzone.com/articles/spring-boot-20-webflux-reactive-performance-test

Ringraziamenti

Alla mia famiglia e ai miei parenti, per essermi stati vicini sempre.

Al mio ragazzo, per aver sempre creduto in me.

A Veronica, per avermi mostrato la strada giusta quando mi sentivo persa.

A Roberto, per esserci stato sempre, ogni singolo giorno di questa grande avventura al Poli.

A tutti i miei amici, per avermi dato la felicità di cui avevo bisogno.

Al mio relatore, per la passione che mi ha trasmesso in ciò che ho studiato.