



Faça login em Medium com o Google



Rafael Estrela

rafael.r.estrela@gmail.com



Brizade youtube

brizadeemail@gmail.com

# Programação Reativa WebFlux e MongoDB

Software Architect da Netshoes explica tudo sobre o tema



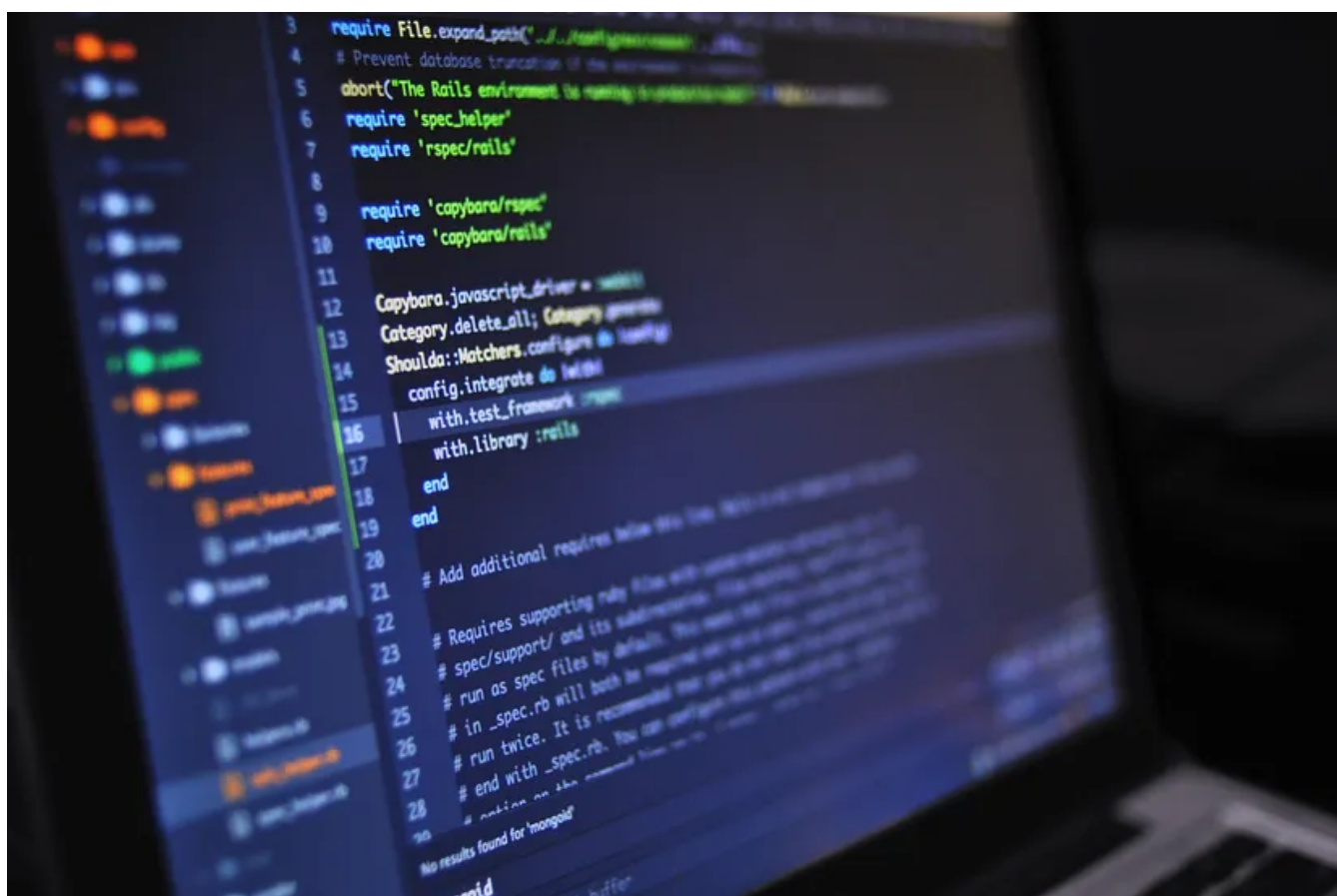
Eder Magalhães · Follow

Published in NSTech

17 min read · Jul 30, 2018



Share



## Introdução

Em tecnologia já nos acostumamos com a constante busca por qualidade, agilidade e melhor performance para atender requisitos que evoluem a todo momento. Ao focar no aspecto *capacidade de processamento*, atualmente contamos com processadores que entregam vários *cores* que potencializam a capacidade

computacional. Se subirmos o nível, podemos explorar a escalabilidade horizontal na nuvem e criar uma infraestrutura com milhares de processadores sem qualquer dificuldade.

Ao fazer uma análise mais detalhada da implementação, ficam algumas questões no ar: o que podemos fazer para aproveitar melhor essa super infraestrutura? Será que realmente precisamos de toda essa capacidade de processamento?

A resposta para essas questões está na Programação Reativa, algo que nos últimos anos vem ganhando maior relevância. Inicialmente, era algo mais popular no front-end para atender a interações complexas de UI, mas agora a abordagem cresce também do lado do back-end, com o objetivo de tirar o melhor proveito de hardware disponível. De acordo com o Manifesto Reativo, uma aplicação reativa deve ser: resiliente, elástica, responsiva e orientada à mensagem. Isso faz com que o modelo de desenvolvimento se torne completamente aderente em aplicações non-blocking que são assíncronas ou orientadas para eventos.

Esse artigo explora algumas características da programação reativa com Java em uma abordagem bem prática. Partindo de um cenário simples, vamos evoluir passo a passo até chegar a um cenário mais complexo. Durante essa jornada iremos implementar um pouco de código usando as tecnologias reativas do Spring Framework 5.

## Paradigma Reativo

O paradigma reativo propõe um modelo diferente para o desenvolvedor, que passa a escrever o código de forma desacoplada, ou seja, os dados ficam isolados das rotinas que fazem a manipulação. O *data stream* (fluxo de dados) é um dos pontos-chave da programação reativa: ele representa um conjunto de dados, e os dados são os eventos. Imagine em fluxo do mundo real, algo que está em movimento e que gera reação em alguém. Do outro lado temos o *subscriber*, que é quem assina o evento, ou melhor, é quem reage ao evento. O subscriber nada mais é do que um conjunto de instruções, uma rotina que é acionada para tratar o evento (dado). Esse acionamento, que, na verdade, é a emissão do evento, fica transparente, sendo abstraído pelo framework. Você vai notar que frameworks reativos na verdade implementam e evoluem o design pattern Observer.

O trecho de código a seguir é um exemplo simples que utiliza o *Spring WebFlux* para demonstrar como criar um data stream a partir de um conjunto de marcas e registra dois subscribers para esse stream:

```
final Flux<String> brands = Flux
    .just("Under Armour", "Asics", "Nike", "Adidas", "Mizuno");
brands.sort()
    .subscribe(System.out::println); //all sorted items

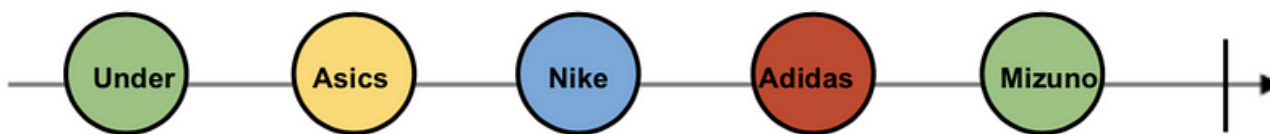
brands.skip(1)
    .groupBy(b -> b.charAt(0))
    .flatMap(group -> group.collectSortedList())
    .subscribe(System.out::println); //items grouped by first char
```

Ainda sem entrar nos detalhes do framework, o interessante é notar que o mesmo stream é consumido por diferentes blocos de código de forma sequencial e desacoplada. O primeiro subscriber ordena os elementos e imprime item a item, e o segundo agrupa os elementos em um subconjunto pela primeira letra da marca e depois imprime esses subconjuntos.

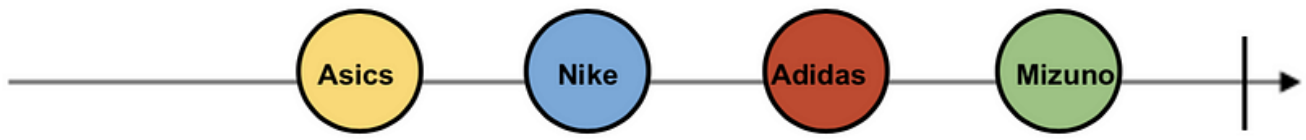
Os desenvolvedores que já trabalham com Java 8 não terão nenhuma dificuldade em compreender esse código, o uso de lambdas expression e a navegação em stream não é mais nenhuma novidade, mas, apesar da semelhança, o conceito é completamente diferente.

A ilustração a seguir demonstra como ocorre a manipulação de streams reativos, de acordo com as instruções do segundo trecho do código demonstrado:

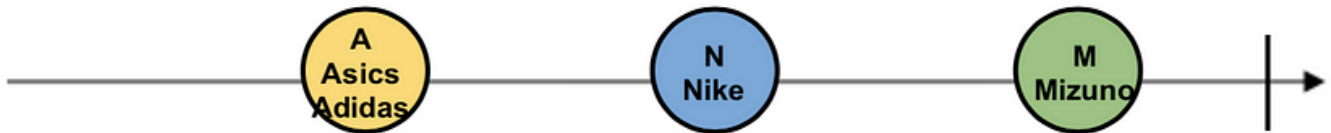
```
Flux.just("Under Armour", "Asics", "Nike", "Adidas", "Mizuno")
```



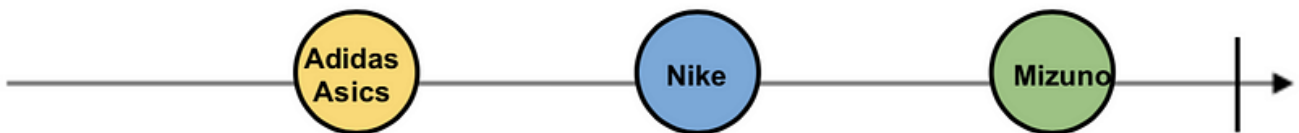
```
brands.skip(1)
```



```
brands.groupBy(b -> b.charAt(0))
```



```
brands.flatMap(group -> group.collectSortedList())
```



Outro conceito-chave de aplicações reativas é o *backpressure*. Imagine um data stream muito grande e um subscriber com uma capacidade de consumo limitada, de forma que a emissão é muito mais rápida do que o consumo dos eventos. Nessa situação corremos um grande risco de *OutOfMemoryError*. O backpressure é um mecanismo para evitar esse problema: ele permite que o emissor controle a cadência no consumo do data stream para que o subscriber não fique sobrecarregado. Um analogia com mundo real seria uma válvula de torneira que delimita o volume de água evitando o transbordo da pia.

Existem duas classificações de data stream:

- *Cold streams*: são fluxos lazy (pull based), formado por um conjunto de dados pré-definidos aguardando por assinantes. Assim que o subscriber é registrado, o cold stream começa a emitir os dados. O trecho de código demonstrado anteriormente é um exemplo de cold stream.
- *Hot streams*: são fluxos ativos (push based), que ficam aguardando dados para encaminhar aos subscribers registrados. Assim que o subscriber for relacionado

ao *hot stream* ele passa a reagir aos dados. Um exemplo seria um listener de um clique de botão (UI).

## Spring WebFlux

O Spring Framework 5 suporta programação reativa via Reactive Stream, a especificação que padroniza o uso de streams reativas dentro da JVM.

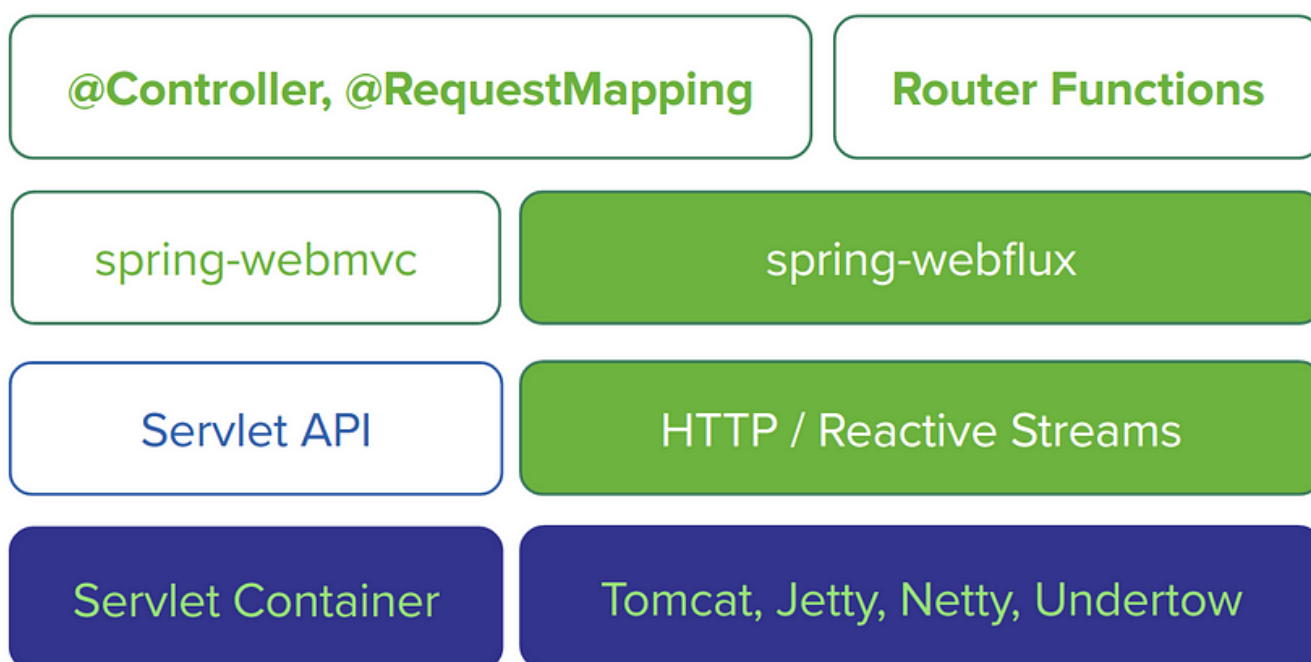
Internamente, o Spring Framework implementa a Reactive Stream através do Reactor. Note que as classes *Flux* e *Mono*, dois publishers de stream reativas, são parte do Reactor.

As classes *Flux* e o *Mono* são expostas nas APIs reativas do Spring Framework:

- *Flux* é um stream reativo formado 0 ou N elementos.
- *Mono* é um stream reativo formado por 0 ou 1 elemento.

O webflux é um módulo do Spring Framework 5, que provê suporte para aplicações web reativas do lado servidor. Porém, além disso, o módulo evolui a stack de tecnologias Spring para web, colocando uma outra abordagem para construção de serviços web, paralela à API Servlets. Apesar de compatível com API Servlet 3.1 non-blocking, não precisamos do Servlet container para executar uma aplicação webflux.

Visão da stack web c/ webflux:



Desenho retirado da documentação do WebFlux

Outra novidade do Spring Framework 5 é que, a partir do webflux, podemos definir os componentes web server-side de duas formas: a forma tradicional, via anotações, amplamente adotada no Spring webmvc, e o novo estilo funcional, onde os componentes são declarados via funções.

Para ilustrar, o código a seguir é um clássico *Hello World*, demonstrando como declarar uma rota em <http://localhost:8080/hello> de forma funcional e subir um servidor web reativo:

```
public class Server {
    private static final String HOST = "localhost";
    private static final int PORT = 8080;
    public void startReactorServer() {
        RouterFunction<ServerResponse> route =
            route(GET("/hello"),
                request -> ok().body(fromObject("Hello reactive world!")));
        HttpServer server = HttpServer.create(HOST, PORT);
        server.newHandler(new
            ReactorHttpHandlerAdapter(toHttpHandler(route)))
            .block();
    }
    public static void main(String[] args) throws Exception {
        Server server = new Server();
        server.startReactorServer();
        System.out.println("Press ENTER to exit.");
        System.in.read();
    }
}public class Server {
```

*Os imports foram omitidos propositalmente por simplicidade.*

O servidor web reativo desse exemplo é o Netty, onde programaticamente criamos uma instância de `HttpServer` na porta 8080. O Netty é um framework que oferece a infraestrutura *client server non-blocking IO*, e continuaremos usando ele nos próximos exemplos do artigo. Outro recurso do webflux é a `WebClient`, uma API funcional, reativa e non-blocking, que tornou-se uma alternativa para o `RestTemplate` — na sequência do artigo veremos um caso de uso desse componente.

## Spring Data

O Spring Data é outro módulo do Spring Framework 5 com suporte a programação reativa. É possível usar o modelo reativo nos providers NoSQL: MongoDB, Apache Cassandra e Redis. Com drivers assíncronos, esses bancos se tornaram candidatos naturais para o *test-drive* do modelo reativo.

O código a seguir demonstra como definir um repositório reativo para o MongoDB:

```
public interface ProductRepository
    extends ReactiveCrudRepository<Product, String> {

    Flux<Product> findByBrand(Mono<String> brand);
}
```

A mudança de maior impacto é o uso dos tipos reativos Mono e Flux, que podem ser mapeados como resultados ou parâmetros dos métodos do repositório. No restante, a interface *ReactiveCrudRepository* define as operações básicas de CRUD, reativas, sob uma entidade.

## Caso de uso simples

Nada melhor do que colocar em prática os recursos reativos do Spring! Como referência, de ponta a ponta nós vamos usar um serviço web simples. Esse serviço implementa um CRUD de Produtos e uma consulta de Produtos com disponibilidade no estoque, sendo que o estoque é gerenciado por outro serviço. Vamos desmembrar os pontos de código mais relevantes dessa demonstração, começando pelo POST para criar Produtos. O código a seguir é a versão imperativa, apenas para efeito comparativo:

```
@RestController
public class ProductController {
    ...
    @PostMapping("/products")
    @ResponseStatus(HttpStatus.CREATED)
    public Product save(@Valid @RequestBody Product product) {
        return gateway.save(product);
    }
    ...
}
```

O *gateway* é um componente que abstrai o *repository* do Spring Data, que, nesse caso, não usa a API reativa do MongoDB. Ao fazer pequenas mudanças, chegamos à primeira versão de método reativo:

```
@PostMapping("/products")
@ResponseStatus(HttpStatus.CREATED)
```

```
public Mono<Product> create(@Valid @RequestBody Product product) {  
    return gateway.save(product);  
}
```

Ambos os métodos do *controller* e *gateway* mudaram a assinatura e o retorno foi alterado para *Mono<Product>*. O *Mono* é o resultado do *repository*: depois de persistir o documento no MongoDB, o driver responde com uma nova referência do objeto. Diferente da primeira versão, quando a thread da requisição (Servlet) que atuava sob *controller* ficava bloqueada esperando o resultado do banco, na versão reativa a *controller* apenas joga pra cima o evento com o produto atualizado.

Como já mencionado, boa parte do Spring Framework foi adaptado para suporte reativo. Isso ocorre nos serializadores e deserializadores do *spring-web* junto com Jackson (JSON). Veja como foi implementado o método PUT para atualizar o Produto:

```
@PutMapping("/products")  
@ResponseStatus(HttpStatus.OK)  
public Mono<Product> update(@Valid @RequestBody Mono<Product>  
product) {  
    return product.flatMap(p -> gateway.save(p));  
}
```

Uma outra abordagem reativa, o método *update* recebe um argumento do tipo *Mono<Product>*. Parece irrelevante, mas, com essa mudança, o *controller* passa apenas descrever o que vai acontecer de forma totalmente non-blocking, ou seja, ele só delega as ações. Note o *@Valid* de “*beans validation*” — na versão anterior, a validação do Produto ocorria antes de passar pelo código do *controller*, mas na versão atual a validação só vai acontecer posteriormente, quando o stream do Produto for consumido (no caso, pelo Spring Data) antes de chegar ao MongoDB.

Outro ponto relevante desse código é o uso do *flatMap*, um dos principais operadores da biblioteca reativa. O *flatMap* faz um merge transformando um ou mais streams em um novo stream. No código acima, temos duas etapas: a primeira faz a transformação da request no objeto *Product* e o valida, e a segunda faz a persistência desse objeto. As duas etapas se tornam uma etapa só e as ações são executadas em sequência. Todo esse desacoplamento impacta bastante na forma em que estamos acostumados a programar, pois não temos mais controle sobre a



execução. Vamos explorar um outro ponto, agora passando pelo código de consulta de Produto, o GET por código:

```
@GetMapping("/products/{code}")
@ResponseStatus(HttpStatus.OK)
public Mono<Product> findOne(@PathVariable String code) {
    return gateway.findByCode(code)
        .switchIfEmpty(Mono.error(new ProductNotFoundException(code)));
}
```

A questão dessa API é que, quando um Produto com o código informado não existe, o retorno HTTP deve ser NOT\_FOUND (404). A estratégia no código foi criar uma exceção (Runtime) exclusiva para esse tipo de retorno. Caso o stream de retorno do banco seja vazio, o operador *switchIfEmpty* dispara um novo stream. Esse novo stream é um Mono, que vai lançar uma exception para o assinante.

Uma exception pode ser lançada em qualquer ponto do stream. Quando o erro acontece, como no código anterior, o subscriber é notificado com a exception no *onError*. Os componentes de *error handling* do Spring Web, como *ControllerAdvice*, funcionam, mas com algumas limitações. Como stream pode ser manipulado por diferentes grupos de thread, caso um erro aconteça em uma thread que não é a que operou a controller, o *ControllerAdvice* não vai ser notificado ([saiba mais aqui](#)). No caso do *findByCode* o erro vai ser lançado na thread da *controller*, então não teremos problemas.

Analisando com mais cautela o Flux (um stream composto por vários elementos), quando ocorre uma exceção por padrão o consumo desse stream é interrompido e nenhuma outra emissão é realizada. Apesar disso, a biblioteca reativa oferece alguns gatilhos para tentar recuperar o stream. Por exemplo, o método *onErrorReturn* poderia ser usado como fallback em caso de falha de algum elemento do stream. Outra opção é o método *retry*, que pode ser usado para realizar novamente a emissão de um elemento que falhou. Através de um parâmetro do método indicamos quantas tentativas de recuperação podem ser executadas.

O próximo método é o DELETE. Veja o código:

```
@DeleteMapping("/products/{code}")
@ResponseStatus(HttpStatus.ACCEPTED)
```

```
public Mono<Void> delete(@PathVariable String code) {  
    return gateway.deleteByCode(code);  
}
```

Aqui o detalhe é o uso do `Mono<Void>`, que representa um stream sem elementos. Essa estratégia é usada em situações onde não temos uma resposta ação. O ponto que deve ficar claro é que, caso mudássemos a assinatura do método colocando o operador `void`, esse código ficaria quebrado pois a exclusão não seria realizada. Para resolver teríamos de assinar o stream, ou delegar o `Mono<Void>` para o framework.

O último método do CRUD, em teoria o mais simples, é o GET, que retorna uma lista com todos os produtos. Porém, vamos incrementar um pouco o código para explorar outras características reativas:

```
@GetMapping(path = "/products", produces =  
    "application/stream+json")  
@ResponseStatus(HttpStatus.OK)  
public Flux<Product> findAll() {  
    return gateway.findAll().delayElements(Duration.ofMillis(300));  
}
```

A primeira observação desse código é a configuração que indica que a response produz “application/stream+json”. A ideia é usar esse recurso quando o data stream for grande e/ou lento o suficiente para fazer a escrita da response parcial. O operador `delayElements` nesse exemplo é usado para retardar a emissão de cada elemento do stream em 300 milissegundos. Note que o resultado dessa API não é um array de produtos no formato JSON, ao invés disso o cliente deve estar preparado para receber objeto a objeto sem correlação. O WebClient consegue tratar response com esse formato e também consegue fazer request com stream:

```
$ curl http://localhost:8080/products  
{  
  "code": "ABC-123", "brand": "Nike", "description": "Tenis  
Nike", "color": "preto", "price": 29900  
}  
{  
  "code": "ABC-222", "brand": "Nike", "description": "Tenis  
Nike", "color": "vermelho", "price": 29900  
}  
{  
  "code": "ABC-333", "brand": "Nike", "description": "Tenis  
Nike", "color": "azul", "price": 29900  
}
```

```
{"code":"ABC-666","brand":"Nike","description":"Tenis Nike","color":"azul","price":29900}
```

## Caso de uso complexo

Ao evoluir nosso serviço de gestão de Produtos, vamos adicionar a funcionalidade que consolida o cadastro com a informação de disponibilidade. Podemos abstrair o gateway que faz essa integração com o seguinte código:

```
public interface InventoryGateway {  
    Mono<Long> getAvailability(String productCode);  
}
```

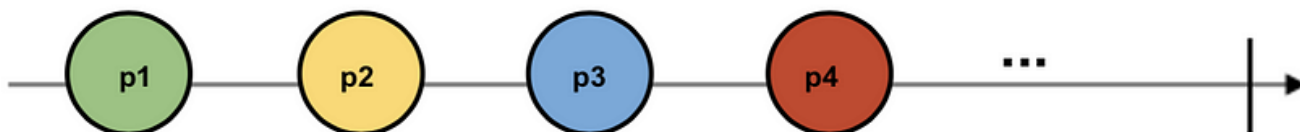
De acordo com o código acima, a chamada para obter a disponibilidade do estoque ocorre por produto, ao usar como chave o código. Podemos implementar a consulta com o uso do RestTemplate do Spring:

```
@Override  
public Mono<Long> getAvailability(String productCode) {  
    final String url = serviceUrl+serviceEndpoint;  
    return Mono.just(getValueAsString(url)) //código do restTemplate...  
        .map(c -> Long.valueOf(c));  
}
```

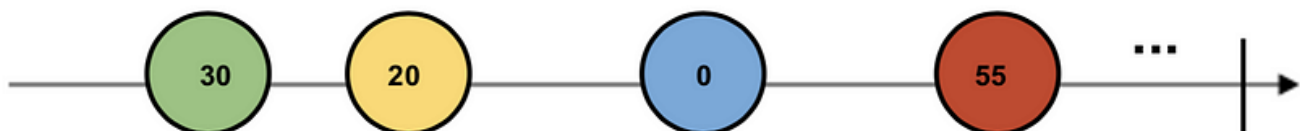
Para focar no problema, abstraí o código do método `getValueAsString`. Nele, nós temos o uso do `RestTemplate` — o que é relevante, já que a chamada desse método que, a priori, é bloqueante, foi encapsulada em um stream reativo, mais explicitamente no `Mono<String>`. Como a quantidade devolvida pelo serviço de estoque é uma `String`, nós precisamos aplicar uma conversão. Entra em cena aqui o operador reativo `map`, usado para transformar um stream de `String` em outro stream de `Long`.

Agora chega o ponto onde é necessário vincular o resultado da busca de produtos com a busca do estoque. Precisamos fazer a junção do stream com resultado do banco de dados com um outro stream que representa as consultas de estoque. A próxima ilustração demonstra a junção dos streams:

```
productGateway.findAll()
```



```
inventoryGateway.getAvailability(product.getCode())
```



```
Flux.combineLatest(
    Mono.just(product),
    inventoryGateway.getAvailability(pr.getCode()),
    (p, a) -> new ProductAvailability(p, a))
```



Isso requer uma estrutura de código mais elaborada do que o simples CRUD. Esse código ficará dentro de um use-case exclusivo para consultar produtos e disponibilidade. A primeira versão poderia ser:

```
public class GetAvailability {
    private ProductGateway productGateway;
    private InventoryGateway inventoryGateway;
    ...
    public Flux<ProductAvailability> execute() {
        return productGateway.findAll()
            .flatMap(product ->
                Flux.combineLatest(
                    Mono.just(product),
                    inventoryGateway.getAvailability(product.getCode()),
```

```
(p, a) -> new ProductAvailability(p, a))
);
}
}
```

Nesse código usamos novamente o operador `flatMap`: ele itera sob todos os elementos do stream inicial. O operador `combineLatest` recebe dois streams e realiza a junção em um terceiro stream formado por objetos “`ProductAvailability`”. Cada elemento mantém a referência do produto e a quantidade disponível. Até o momento foi feito um mix de código bloqueante com código reativo.

Com esse código, já teríamos a nossa solução proposta, porém durante os testes encontramos algumas limitações dessa versão: a consulta de disponibilidade é sequencial e a API é lenta. Em média, cada request leva 1.3 segundos. Desta forma, o fluxo levará mais de um minuto caso a base de produtos tenha mais de 50 documentos. Com isso, entramos em outro ponto interessante que a estrutura do Reactor fornece, que é a possibilidade de consumir os streams de forma paralela. É possível ainda configurar um thread pool exclusivo para isso. Sendo assim, adaptamos a versão do método `execute` para trabalhar de forma paralela:

```
public Flux<ProductAvailability> execute() {
    int coreCount = Runtime.getRuntime().availableProcessors();
    AtomicInteger assigner = new AtomicInteger(0);
    return productGateway.findAll()
        .groupBy(p -> assigner.incrementAndGet() % coreCount) //by core
        .flatMap(grp->
            grp.publishOn(Schedulers.parallel()) // thread pool
            .map(product ->
                Flux.combineLatest(
                    Mono.just(product),
                    inventoryGateway.getAvailability(product.getCode()),
                    (p, a) -> new ProductAvailability(p, a))
                )
            ).flatMap(f -> f)
        ).filter(pa -> pa.getAvailability() > 0l);
}
```

Esse código é um pouco mais complexo: no primeiro trecho, recuperamos a quantidade de processadores na máquina para fazer uma conta e quebrar as requisições em lotes menores, de acordo com a capacidade de processadores. Se a máquina tem cinco processadores, o operador `groupBy` vai gerar um stream com

cinco sub-streams de produtos. No flatMap do stream principal acessamos o grupo através do publishOn, e indicamos qual thread pool (*Schedulers.parallel()*) será responsável pela publicação do stream. O publishOn, nesse caso, faz com que as requisições para o outro serviço ocorram de forma paralela. Depois de configurar o thread pool, usamos map para, enfim, combinar o produto com a disponibilidade. O uso do groupBy gera um stream de Flux<Flux<ProductAvailability>> — o último flatMap unifica esses sub-streams em uma estrutura: o único stream Flux<ProductAvailability>.

A boa notícia é que existe uma maneira bem mais simples de resolver esse problema. Ao usar o WebClient no lugar do RestTemplate, encapsulamos a requisição ao serviço de estoque em uma estrutura assíncrona e reativa. Veja como fica a nova versão do método getAvailability:

#### @Override

```
public Mono<Long> getAvailability(final String productCode) {
    return getValueAsString(serviceEndpoint)
        .map(c -> Long.valueOf(c));
}
private Mono<String> getValueAsString(final String url) {
    return client.get()
        .uri(url)
        .retrieve()
        .bodyToMono(String.class)
        .defaultIfEmpty("0")
        .onErrorReturn("0");
}
```

Note o *fallback* para retornar um valor padrão caso ocorra alguma falha na chamada da API. Essa mudança impacta diretamente o use-case: como não é mais necessário fazer a configuração de paralelismo o código fica bem mais simples:

```
public Flux<ProductAvailability> execute() {
    return productGateway.findAll()
        .flatMap(product ->
            Flux.combineLatest(
                Mono.just(product),
                inventoryGateway.getAvailability(product.getCode()),
                (p, a) -> new ProductAvailability(p, a))
            ).filter(pa -> pa.getAvailability() > 0l);
}
```

O WebClient define uma interface extremamente flexível que, por natureza, é reativa e realiza as requisições para o outro serviço de forma paralela. O WebClient usa o mesmo thread pool das requisições HTTP — eventualmente, isso pode ser uma limitação.

## Testabilidade

O Spring Framework sempre ofereceu funcionalidades para que os desenvolvedores criem suas rotinas de testes. Como a abordagem do modelo reativo impacta diretamente na forma que escrevemos os códigos, o mesmo acontece com as rotinas de testes. Justamente por esse impacto, o framework criou novos componentes para suportar streams durante os testes. A anotação `@WebFluxTest` é usada para testes de API / contrato em controllers do webflux. Ela habilita a infraestrutura mínima do webflux para executar os testes — os tradicionais `@Component` são descartados. O `@WebFluxTest` também habilita o `WebTestClient`, outro novo recurso que permite a realização dos testes em controladores sem a necessidade de HTTP web server. O código a seguir é o exemplo um cenário de teste da API GET por código de produto:

```
@RunWith(SpringRunner.class)
@WebFluxTest(controllers = ProductController.class)
public class ProductControllerTest {
    @Autowired WebTestClient client;
    @MockBean ProductGateway gateway;
    @Test
    public void testGetProductByCodeShouldBeOk() {
        final Product product =
            new Product("BBB-1111", "Adidas", "Tenis Adidas", "Branco", 59900);
        given(gateway.findByCode("BBB-1111"))
            .willReturn(product);
    }
}
```

Open in app ↗

Sign up

Sign In



Search



```
.jsonPath("$.code").isEqualTo("BBB-1111")
.jsonPath("$.brand").isEqualTo("Adidas")
.jsonPath("$.description").isEqualTo("Tenis Adidas")
.jsonPath("$.color").isEqualTo("Branco")
.jsonPath("$.price").isEqualTo(59900);
}
...
}
```

Note que o `@WebFluxTest` já é vinculado com a *controller*, o que indica que os cenários de testes serão direcionados para `ProductController`. Usamos o Mockito

para injetar o mock do *gateway* e definir (BDD) os dados que o mock vai retornar. Outro destaque é a estrutura do `WebTestClient`, que declara os critérios de aceite de acordo com JSON com a resposta da API.

Tão importante quanto o código que implementa funcionalidades de negócio é o código que testa essas funcionalidades. O modelo reativo também impõe mudanças na forma em que pensamos e escrevemos nossos testes. O Reactor disponibiliza o `StepVerifier`, um componente que consegue validar passo a passo a execução de stream. O próximo código define um cenário de teste integrado com banco de dados para testar a exclusão de um documento:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ProductRepositoryIntegrationTest {
    @Autowired ProductRepository repository;
    @Test
    public void deleteAndFindOneProductShouldOk() {
        Final Flux<Product> deleteAndFind =
            repository.deleteById(Mono.just("BBB-3333"))
                .thenMany(repository.findById("BBB-3333"));
        StepVerifier.create(deleteAndFind)
            .expectSubscription()
            .expectNextCount(0L)
            .expectComplete();
    }
    ...
}
```

Esse cenário envia duas instruções para o banco de dados. Conforme já vimos anteriormente, o delete retorna um `Mono<Void>`. O operador `thenMany` permite substituir o stream atual por outro. Nesse caso, o stream original com resultado do delete é substituído por outro, com o resultado da consulta por código de produto, um `Flux<Product>`. O `StepVerifier` é um *builder* que usamos para instruir a sequência dos testes: no exemplo acima, ele inspeciona o stream e verifica se o resultado da consulta não retornou o objeto depois da exclusão.

## Frameworks Reativos

O ReactiveX, criado pela Microsoft, foi uma das tecnologias precursoras em programação reativa. O projeto evoluiu se tornou algo como um padrão reativo: atualmente é conhecido como Rx e a comunidade abraçou o conceito e portou a tecnologia em várias linguagens (JavaScript, Python, Swift e Java).



Entrando no universo Java, existem diversas implementações reativas para a JVM: RxJava, RxKotlin, RxJavaFX, RxNetty e RxGroovy, por exemplo. Outro movimento criou o Reactive Streams, uma especificação de streams reativos para a JVM. A Oracle, inclusive, lançou no JDK 9 a Flow API como uma implementação de Reactive Streams.

O Spring Framework 5, além do Reactor, também suporta o RxJava como provider, apesar de algumas diferenças na API. Os dois frameworks são muito similares.

## Conclusão

Definitivamente a criação de aplicações reativas no back-end é uma realidade. O esforço da Pivotal para suportar o modelo reativo no Spring Framework já é uma prova disso. A programação reativa traz muito da programação funcional: mesmo escrevendo com código Java, a sensação é de que estamos trabalhando com outra linguagem. Um código relativamente simples no modelo imperativo pode ser bem mais complexo na versão reativa. Nós vimos isso no exemplo que cria uma sub-stream para paralelizar requests.

Outro aspecto é se acostumar a não ter mais controle de quando as instruções são executadas: nós registramos o código e empilhamos no consumo do stream. No começo isso é estranho e complexo para desenvolvedores, pois nosso cérebro está acostumado com instruções blocantes: fazer a chamada da função, aguardar o processamento e o obter o resultado. Conseguimos explorar alguns operadores reativos no artigo — acesse os links complementares para descobrir o que mais o framework reativo pode oferecer.

Na Netshoes sempre exploramos novas tecnologias que são aderentes ao negócio da companhia. Nós já estamos desenvolvendo serviços reativos usando o Spring Framework 5. A arquitetura de microserviços pode tornar viável uma experiência com a programação reativa, com impacto controlado.

## Projeto de Referência

Todo o código demonstrado nesse artigo está a disposição no projeto de referência para complementar seu estudo. A aplicação usa o MongoDB e deixamos docker-compose para gerar o ambiente.

- Repositório: <https://github.com/netshoes/blog-spring-reactive>

## Leitura Complementar

- [Projeto Reactor](#)
- Documentação do [webflux](#)
- Documentação do [Spring Data MongoDB](#)
- Projeto [RxJava](#)
- Projeto [Netty](#)



[Entre para nosso time](#)

Reactive Programming

Spring Boot

Como Fazemos

Java

Spring Data



Follow

## Written by Eder Magalhães

87 Followers · Writer for NSTech

I see dead code!

---

More from Eder Magalhães and NSTech



 Eder Magalhães

## Lições após o primeiro ano reativo!

Iniciamos nossa jornada reativa com o Spring WebFlux na Netshoes há pouco mais de um ano. De lá pra cá nós estudamos, colocamos a mão na...

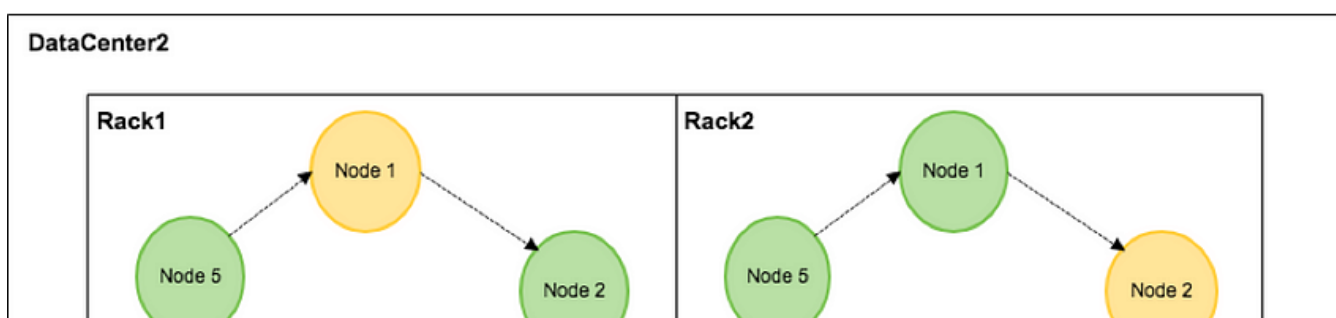
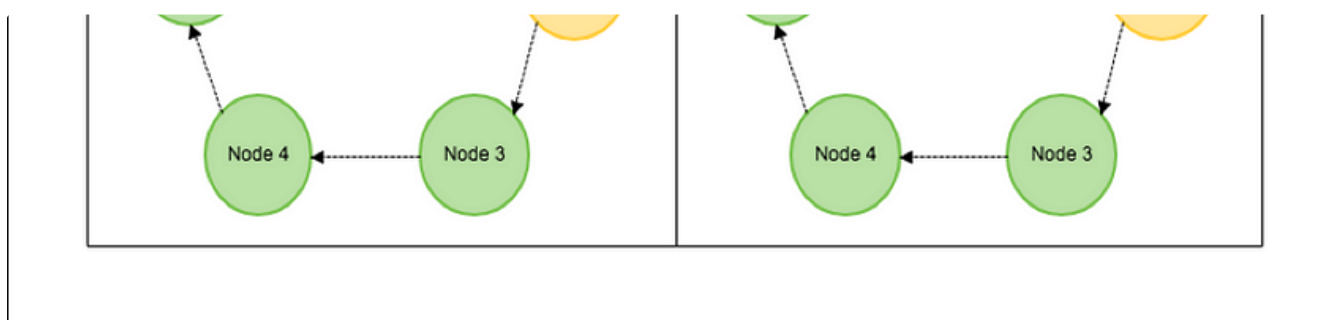
8 min read · Sep 15, 2019




3



2



 alexandre sabino in NSTech

## Apache Cassandra: confira absolutamente tudo a respeito

## Software Architect da Netshoes dissection o banco de dados e passa o que você precisa saber

12 min read · Aug 1, 2018



257



NSTech in NSTech

# Tutorial: Spring Boot - Movendo para o Kubernetes

O Software Architect Guilherme Roveri mostra como desenvolver dois microserviços em Java com Stack Spring Boot com Kubernetes

15 min read · Jun 26, 2018



68





 NSTech in NSTech

## Primeiros passos com Spring State Machine

Este artigo tem como objetivo apresentar a Spring State Machine como uma opção de desenvolvimento de máquinas de estados de entidades

7 min read · Mar 22, 2019



42



1




See all from Eder Magalhães

See all from NSTech

## Recommended from Medium



 Anurag Rana in Naukri Engineering

## RestTemplate is deprecated. Use WebClient.

RestTemplate has been deprecated in favor of the newer WebClient in Spring Framework 5.0 and later versions. This means that while the...

6 min read · May 10



624



6



# WebClient vs RestTemplate

 Pushkar Kumar

## Spring WebClient vs RestTemplate: What's better in 2023?

Communication is the key — we come across this term many times in our life and this is so true. The whole of mankind survives by...

5 min read · Sep 17



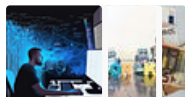
23



1



## Lists



### It's never too late or early to start something

15 stories · 161 saves



### General Coding Knowledge

20 stories · 437 saves



### New\_Reading\_List

174 stories · 150 saves



Bubu Tripathy

## Best Practices: Entity Class Design with JPA and Spring Boot

In the world of modern software development, efficient design and implementation of entity classes play a crucial role in building robust...

8 min read · Aug 10



305



6



Anuj Dekavadiya

## Migrating Spring Boot Microservice from Java 8 to Java 17

In this article, We'll explore the reasons for migration, the benefits and the steps to migrate the Spring Boot Microservice.

4 min read · Oct 9



8







Svosh

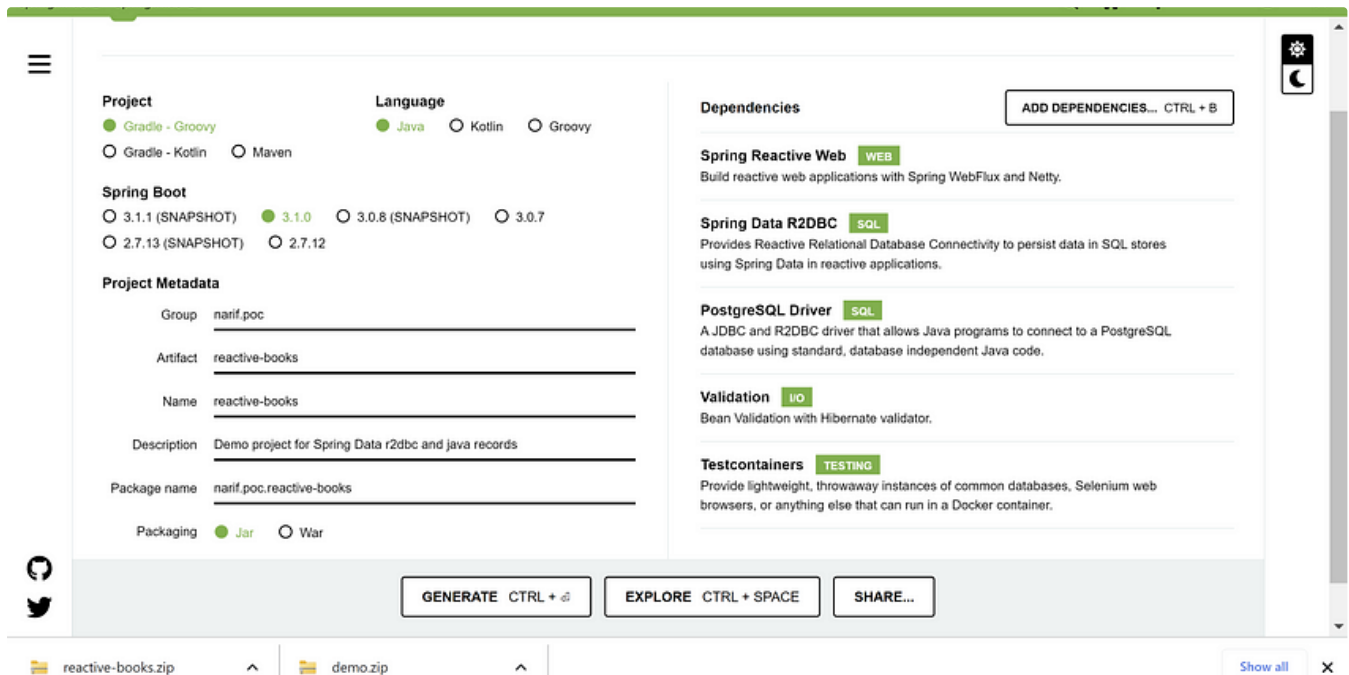
## Java concurrency in practice: synchronization and locks

In Java, locks and synchronization mechanisms are used to coordinate the access to shared resources and protect critical sections of code...

14 min read · Oct 8



35



Najeeb Arif in Dev Genius

## Java Records and Spring Data: Reactive Data Access

In this blog post we will learn how we can use Java Records with Spring Data, specifically Spring Data R2DBC. In this blog post we will...

13 min read · May 28



5



See more recommendations