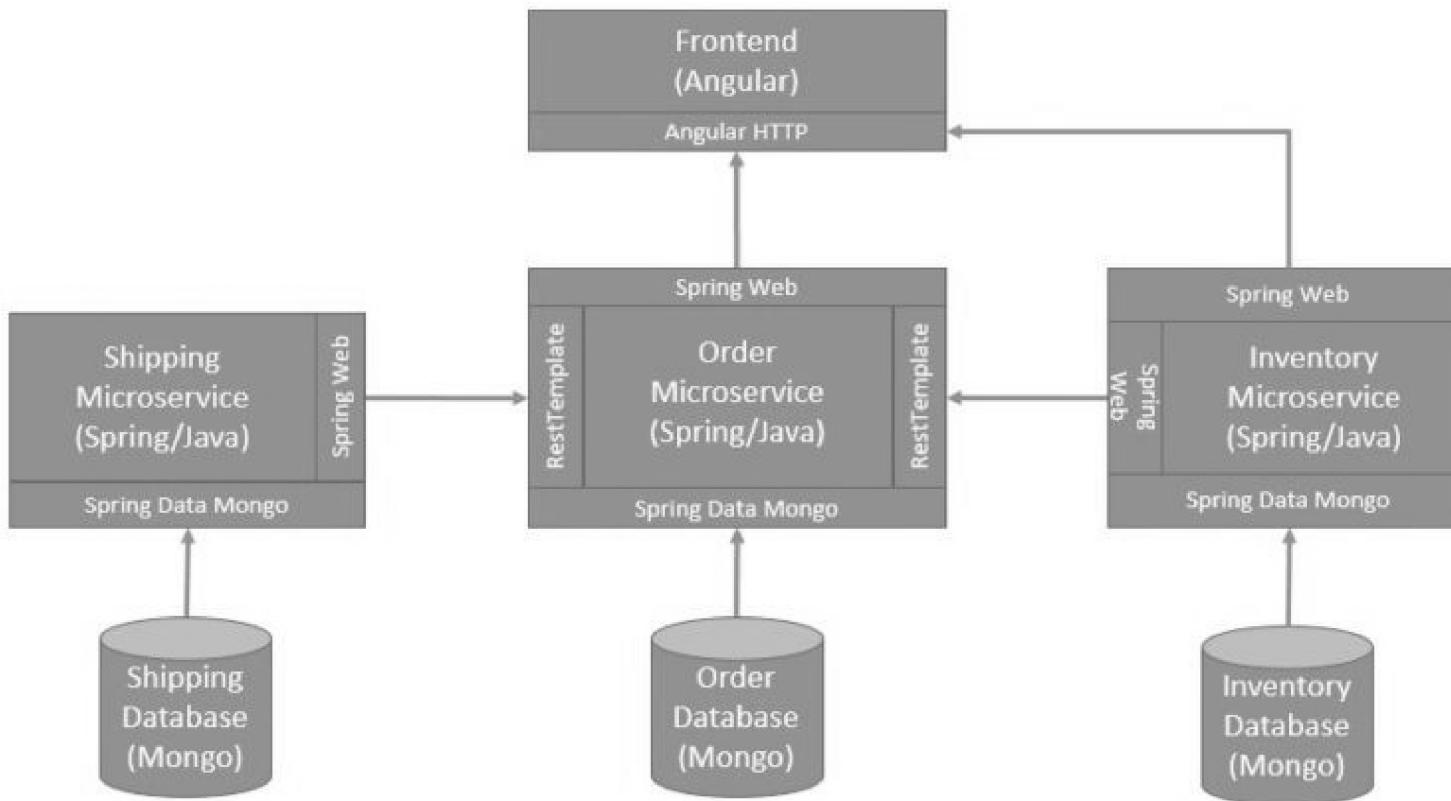


Sistemas reativos anotações de estudos.

ESSAS SÃO AS ANOTAÇÕES QUE FIZ DOS PONTOS QUE MAIS ME CHAMARAM ATENÇÃO DESSE ARTIGO e outras fontes.

O código-fonte deste artigo pode ser encontrado no GitHub.

Sistema usado como exemplo:



No ano de 2013, **uma equipe de desenvolvedores, liderada por Jonas Boner, se reuniu para definir um conjunto de princípios básicos** em um documento conhecido como [Manifesto Reativo](#). Isto é o que lançou as bases para um estilo de arquitetura para criar Sistemas Reativos. Desde então, este manifesto atraiu muito interesse da comunidade de desenvolvedores.

Basicamente, este documento prescreve **a receita para um sistema reativo ser flexível, fracamente acoplado e escalável**. Isso torna esses sistemas fáceis de desenvolver, tolerantes a falhas e, o mais importante, altamente responsivos, a base para experiências incríveis do usuário.

Então, qual é essa receita secreta? Bem, dificilmente é segredo! O manifesto define as características ou princípios fundamentais de um sistema reativo:

- *Responsivo* : um sistema reativo deve fornecer um tempo de resposta rápido e consistente e, portanto, uma qualidade de serviço consistente.
- *Resiliente* : um sistema reativo deve permanecer responsável em caso de falhas aleatórias por meio de replicação e isolamento
- *Elástico* : esse sistema deve permanecer responsável sob cargas de trabalho imprevisíveis por meio de escalabilidade econômica
- *Orientado a mensagens* : deve contar com a passagem de mensagens assíncronas entre os componentes do sistema

Esses princípios parecem simples e sensatos, mas nem sempre são mais fáceis de implementar em uma arquitetura corporativa complexa. Neste tutorial, desenvolveremos um sistema de amostra em Java com esses princípios em mente!

Contrapressão:

A contrapressão em sistemas de software é a capacidade de sobrecarregar a comunicação de tráfego.

Em outras palavras, os emissores de informações sobrecarregam os consumidores com dados que eles não são capazes de processar.

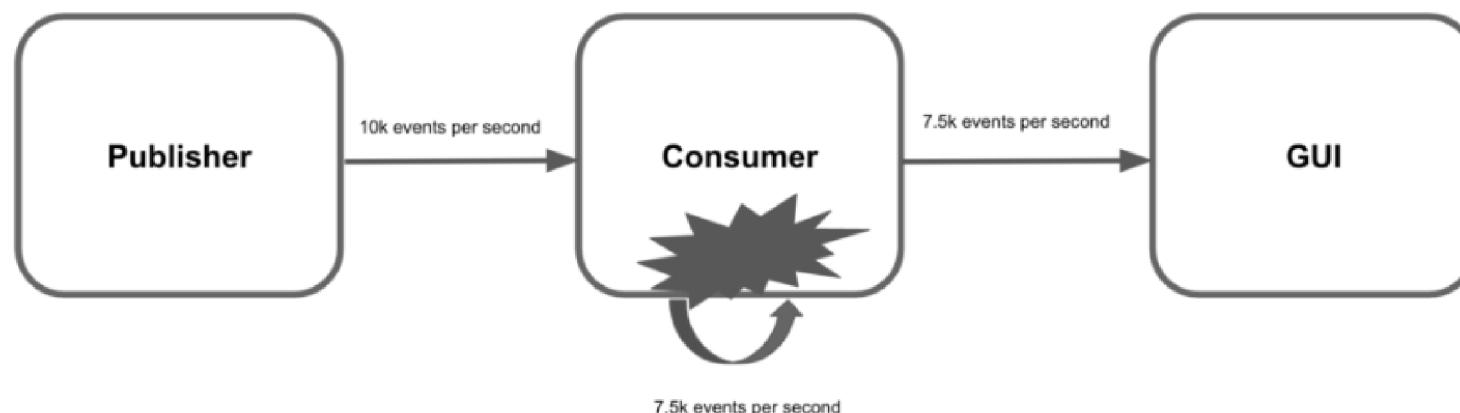
Eventualmente, as pessoas também aplicam esse termo como o mecanismo para controlar e lidar com isso. São as ações de proteção tomadas pelos sistemas para controlar as forças a jusante.

2.1. O que é contrapressão?

Em fluxos reativos, a contrapressão também define como regular a transmissão dos elementos do fluxo . Em outras palavras, controle quantos elementos o destinatário pode consumir.

Vamos usar um exemplo para descrever claramente o que é:

- O sistema contém três serviços: o Publicador, o Consumidor e a Interface Gráfica do Usuário (GUI).
- O Publicador envia 10.000 eventos por segundo para o Consumidor
- O consumidor os processa e envia o resultado para a GUI
- A GUI exibe os resultados para os usuários
- O consumidor só pode lidar com 7500 eventos por segundo

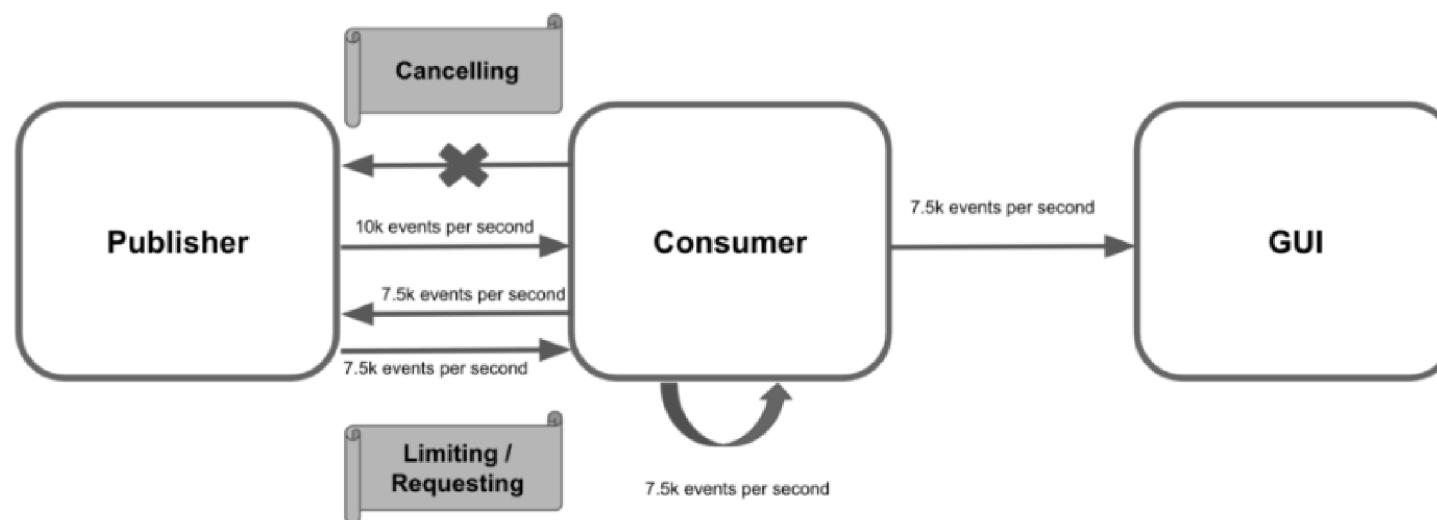


Nessa velocidade, o consumidor não consegue gerenciar os eventos (contrapressão) . Consequentemente, o sistema entraria em colapso e os usuários não veriam os resultados.

2.3. Controlando a contrapressão

Vamos nos concentrar em controlar os eventos emitidos pelo editor. Basicamente, existem três estratégias a seguir:

- **Envie novos eventos somente quando o assinante os solicitar**. Esta é uma estratégia pull para reunir elementos na solicitação do emissor
- **Limitando o número de eventos a serem recebidos no lado do cliente**. Trabalhando como uma estratégia de push limitado, o publisher só pode enviar uma quantidade máxima de itens para o cliente de uma só vez
- **Cancelamento do streaming de dados quando o consumidor não puder processar mais eventos**. Nesse caso, o receptor pode abortar a transmissão a qualquer momento e assinar o fluxo novamente mais tarde.



Front-end (ANGULAR):

Precisamos **criar um componente simples em Angular para lidar com create e fetch orders**. De importância específica é a parte em que chamamos nossa API para criar o pedido:

```
createOrder() {  
    let headers = new HttpHeaders({'Content-Type': 'application/json'});  
    let options = {headers: headers}  
    this.http.post('http://localhost:8080/api/orders', this.form.value, options)  
        .subscribe(  
            (response) => {  
                this.response = response  
            },  
            (error) => {  
                this.error = error  
            }  
        )  
}
```

O trecho de código acima **espera que os dados do pedido sejam capturados em um formulário e disponibilizados no escopo do componente**. Angular oferece suporte fantástico para criar formulários simples a complexos usando formulários reativos e orientados a modelos.

Também é importante a parte onde obtemos os pedidos criados anteriormente:

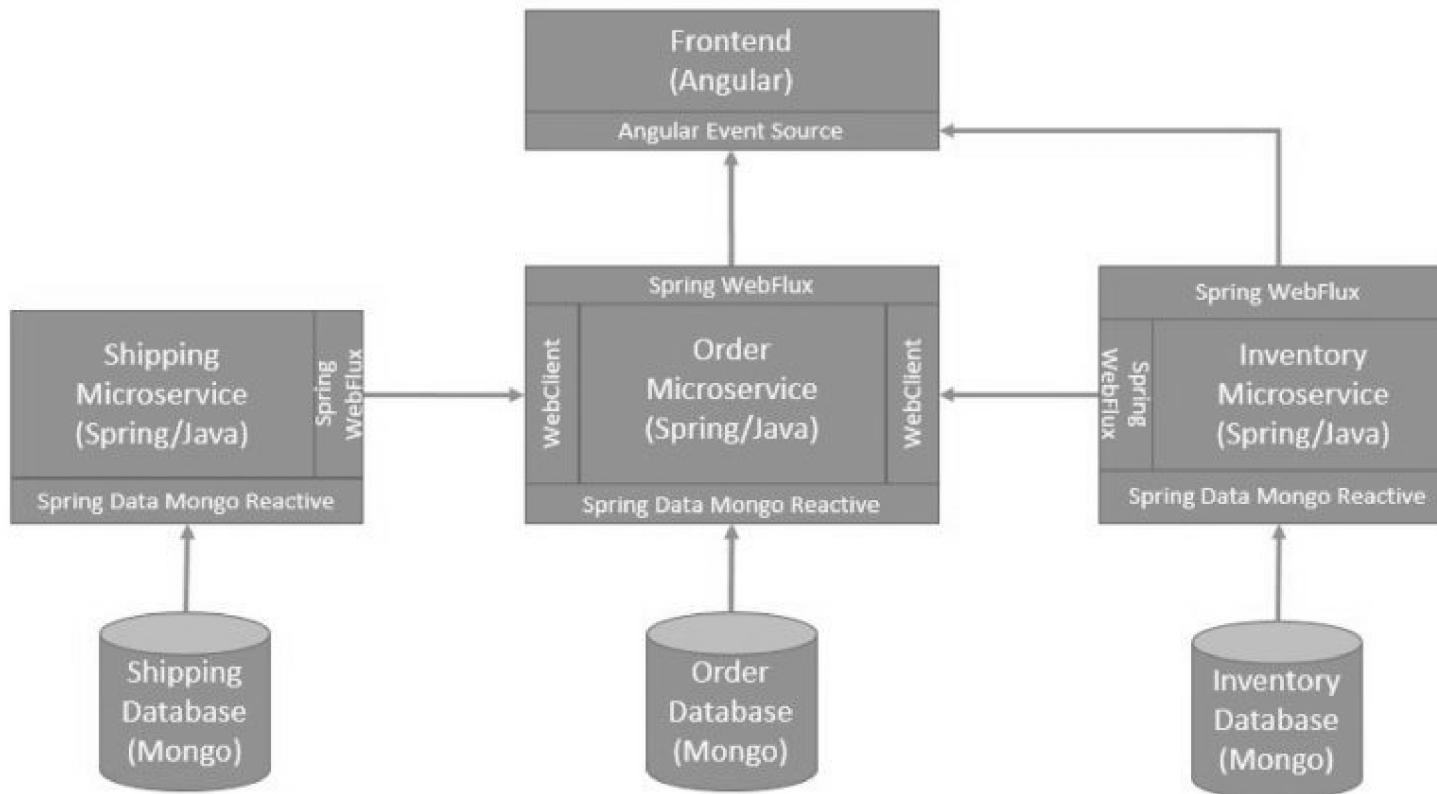
```
getOrders() {  
    this.previousOrders = this.http.get('http://localhost:8080/api/orders')  
}
```

Programação reativa:

A programação reativa é um paradigma de programação em que o foco está no desenvolvimento de componentes assíncronos e sem bloqueio.

Bloquear chamadas em qualquer programa geralmente **resulta em recursos críticos apenas esperando que as coisas aconteçam**. Isso inclui chamadas de banco de dados, chamadas para serviços da Web e chamadas de sistema de arquivos. Se pudermos liberar os threads de execução dessa espera e fornecer um mecanismo para retornar assim que os resultados estiverem disponíveis, isso resultará em uma utilização de recursos muito melhor.

Isso é o que a adoção do paradigma de programação reativa faz por nós. Embora seja possível mudar para uma biblioteca reativa para muitas dessas chamadas, pode não ser possível para tudo. Para nós, felizmente, o Spring torna muito mais fácil usar programação reativa com MongoDB e APIs REST:



Spring Data Mongo tem suporte para acesso reativo por meio do MongoDB Reactive Streams Java Driver. Ele fornece *ReactiveMongoTemplate* e *ReactiveMongoRepository*, ambos com ampla funcionalidade de mapeamento.

O Spring WebFlux fornece a estrutura da Web de pilha reativa para Spring, permitindo código sem bloqueio e contrapressão de fluxos reativos. Ele aproveita o Reactor como sua biblioteca reativa. Além

disso, ele fornece o *WebClient* para executar solicitações HTTP com contrapressão de fluxos reativos. Ele usa o Reactor Netty como a biblioteca cliente HTTP.

Ao usarmos essas dependências em nosso projeto Spring teremos de alterar o nosso código um pouco para se tornar um código digno da programação reativa, veremos abaixo essa mudança:

```
@Transactional
public Order handleOrder(Order order) {
    order.getLineItems()
        .forEach(l → { Antes do service do micro serviços de inventário.
            Product> p = productRepository.findById(l.getProductId())
                .orElseThrow(() → new RuntimeException("Could not find the product: " + l.
                    getProductId()));
            if (p.getStock() ≥ l.getQuantity()) {
                p.setStock(p.getStock() - l.getQuantity());
                productRepository.save(p);
            } else {
                throw new RuntimeException("Product is out of stock: " + l.getProductId());
            }
        });
    return order.setOrderStatus(OrderStatus.SUCCESS);
}

@Transactional
public Order revertOrder(Order order) {
    order.getLineItems()
        .forEach(l → {
            Product p = productRepository.findById(l.getProductId())
                .orElseThrow(() → new RuntimeException("Could not find the product: " + l.
                    getProductId()));
            p.setStock(p.getStock() + l.getQuantity());
            productRepository.save(p);
        });
    return order.setOrderStatus(OrderStatus.SUCCESS);
}
```

```
@Transactional
public Mono<Order> handleOrder(Order order) {
    return Flux.fromIterable(order.getLineItems())
        .flatMap(l -> productRepository.findById(l.getProductId()))
        .flatMap(p -> {
            int q = order.getLineItems().stream()
                .filter(l -> l.getProductId().equals(p.getId()))
                .findAny().get()
                .getQuantity();
            if (p.getStock() >= q) {
                p.setStock(p.getStock() - q);
                return productRepository.save(p);
            } else {
                return Mono.error(new RuntimeException("Product is out of stock: " + p.getId()));
            }
        })
        .then(Mono.just(order.setOrderStatus("SUCCESS")));
}

@Transactional
public Mono<Order> revertOrder(Order order) {
    return Flux.fromIterable(order.getLineItems())
        .flatMap(l -> productRepository.findById(l.getProductId()))
        .flatMap(p -> {
            int q = order.getLineItems().stream()
                .filter(l -> l.getProductId().equals(p.getId()))
                .findAny().get()
                .getQuantity();
            p.setStock(p.getStock() + q);
            return productRepository.save(p);
        })
        .then(Mono.just(order.setOrderStatus("SUCCESS")));
}
```

Percebemos que agora nos nossos retornos e métodos tem algo de diferente, sim estamos usando um retorno do Spring WebFlux nesse caso.

Também alteramos o retorno dos nossos controllers respectivamente.

No micro serviço de pedido que estamos enviando as informações para fora (Front-end) temos que mudar mais o service, pois temos de usar o Spring WebClient nele para invocar os endpoints reativos de inventário e envio.

```
public Order createOrder(Order order) {
    boolean success = true;
    Order savedOrder = orderRepository.save(order);
    Order inventoryResponse = null; Antes do service do ms de pedido
    try {
        inventoryResponse = restTemplate.postForObject(
            inventoryServiceUrl, order, Order.class);
    } catch (Exception ex) {
        success = false;
    }
    Order shippingResponse = null;
    try {
        shippingResponse = restTemplate.postForObject(
            shippingServiceUrl, order, Order.class);
    } catch (Exception ex) {
        success = false;
        HttpEntity<Order> deleteRequest = new HttpEntity<>(order);
        ResponseEntity<Order> deleteResponse = restTemplate.exchange(
            inventoryServiceUrl, HttpMethod.DELETE, deleteRequest, Order.class);
    }
    if (success) {
        savedOrder.setOrderStatus(OrderStatus.SUCCESS);
        savedOrder.setShippingDate(shippingResponse.getShippingDate());
    } else {
        savedOrder.setOrderStatus(OrderStatus.FAILURE);
    }
    return orderRepository.save(savedOrder);
}

public List<Order> getOrders() {
    return orderRepository.findAll();
}
```

```
public Mono<Order> createOrder(Order order) {
    return Mono.just(order)
        .flatMap(orderRepository::save)
        .flatMap(o -> {
            return webClient.method(HttpMethod.POST)
                .uri(inventoryServiceUrl)
                .body(BodyInserters.fromValue(o))
                .exchange();
        })
        .onErrorResume(err -> {
            return Mono.just(order.setOrderStatus(OrderStatus.FAILURE)
                .setResponseMessage(err.getMessage()));
        })
        .flatMap(o -> {
            if (!OrderStatus.FAILURE.equals(o.getOrderStatus())) {
                return webClient.method(HttpMethod.POST)
                    .uri(shippingServiceUrl)
                    .body(BodyInserters.fromValue(o))
                    .exchange();
            } else {
                return Mono.just(o);
            }
        })
        .onErrorResume(err -> {
            return webClient.method(HttpMethod.POST)
                .uri(inventoryServiceUrl)
                .body(BodyInserters.fromValue(order))
                .retrieve()
                .bodyToMono(Order.class)
                .map(o -> o.setOrderStatus(OrderStatus.FAILURE)
                    .setResponseMessage(err.getMessage()));
        })
        .map(o -> {
            if (!OrderStatus.FAILURE.equals(o.getOrderStatus())) {
                return order.setShippingDate(o.getShippingDate())
                    .setOrderStatus(OrderStatus.SUCCESS);
            } else {
                return order.setOrderStatus(OrderStatus.FAILURE)
                    .setResponseMessage(o.getResponseMessage());
            }
        })
        .flatMap(orderRepository::save);
}

public Flux<Order> getOrders() {
    return orderRepository.findAll();
}
```

Front-end:

Agora que nossas APIs são capazes de transmitir eventos à medida que ocorrem, é bastante natural que possamos aproveitar isso também em nosso front-end. Felizmente, o Angular oferece suporte a [EventSource](#), a interface para eventos enviados pelo servidor.

Vamos ver como podemos extrair e processar todos os nossos pedidos anteriores como um fluxo de eventos:

```
/**  
 * getOrderStream()  
 *  
 * This function creates an Observable that will listen to an EventSource for orders.  
 *  
 * @returns {Observable} An observable that will emit the orders received from the EventSource.  
 */  
getOrderStream() {  
    return Observable.create((observer) => {  
        // Create an EventSource to listen to orders  
        let eventSource = new EventSource('http://localhost:8080/api/orders')  
        // On message, parse the data and push it to the orders array  
        eventSource.onmessage = (event) => {  
            let json = JSON.parse(event.data)  
            this.orders.push(json)  
            // Run the observer's next() method in the Angular zone  
            this._zone.run(() => {  
                observer.next(this.orders)  
            })  
        }  
        // On error, check the readyState of the EventSource  
        eventSource.onerror = (error) => {  
            if(eventSource.readyState === 0) {  
                observer.error(error)  
            }  
        }  
    })  
}
```

```
// If the readyState is 0, close the EventSource and complete the observer
eventSource.close()
this._zone.run(() => {
    observer.complete()
})
} else {
    // Otherwise, emit an error
    this._zone.run(() => {
        observer.error('EventSource error: ' + error)
    })
}
})
}
}
```

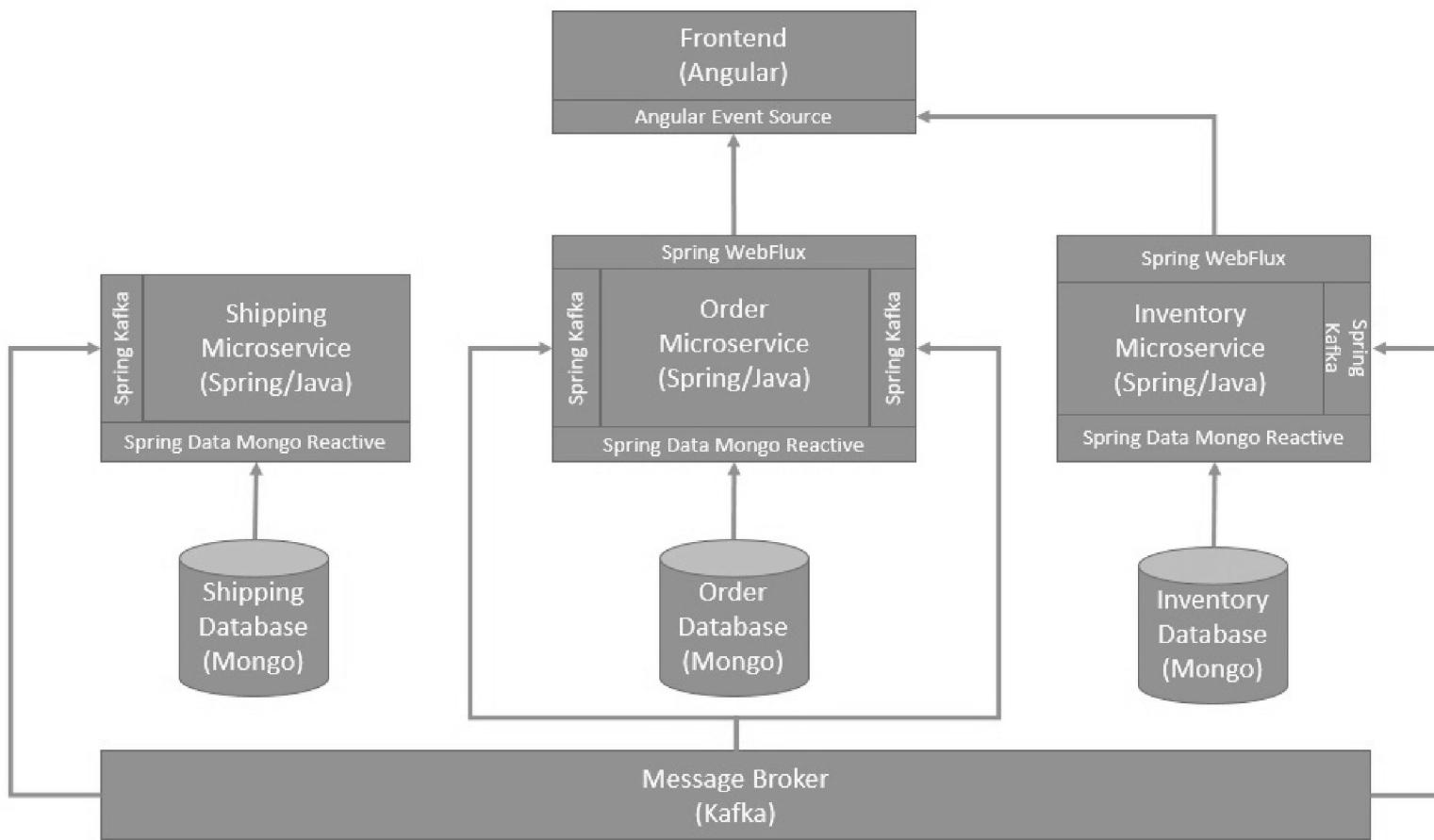
Se quiser a explicação de cada detalhe com mais especificidade dessa função acima podes usar [ocodepal.ai](#).

Arquitetura Orientada a Mensagens:

O primeiro problema que abordaremos está relacionado à comunicação serviço a serviço. No momento, **essas comunicações são síncronas, o que apresenta vários problemas**. Isso inclui falhas em cascata, orquestração complexa e transações distribuídas, para citar alguns.

Qual a diferença entre comunicação síncrona e assíncrona? Como você viu, a principal diferença entre a comunicação assíncrona e síncrona é que a primeira indica uma mensagem que não vai, ou não precisa, ser respondida de imediato. Já a síncrona são as mensagens respondidas em simultâneo.

Uma maneira óbvia de resolver esse problema é tornar essas comunicações assíncronas. Um **intermediário de mensagem para facilitar toda a comunicação serviço a serviço** pode fazer o truque para nós. Usaremos o Kafka como nosso agente de mensagens e [o Spring for Kafka](#) para produzir e consumir mensagens:



Usaremos um único tópico para produzir e consumir mensagens de pedido com diferentes status de pedido para que os serviços reajam.

Vamos ver como cada serviço precisa mudar:

Serviço de Inventário:

Vamos começar definindo o produtor de mensagem para nosso serviço de inventário:

```
@Autowired
private KafkaTemplate<String, Order> kafkaTemplate;
public void sendMessage(Order order) {
    """
    This function sends a message to a Kafka topic using the KafkaTemplate.
    Args:
    order (Order): An instance of the Order class to be sent as a message.
    Returns:
    None
    """
    # Send the message to the "orders" topic using the KafkaTemplate
    this.kafkaTemplate.send("orders", order)
}
```

A seguir, teremos que definir um consumidor de mensagem para o serviço de inventário para reagir a diferentes mensagens sobre o tema:

```
@KafkaListener(topics = "orders", groupId = "inventory")
public void consume(Order order) throws IOException {
    """
    This function consumes messages from a Kafka topic named "orders" and processes them based on the order status.
    If the order status is "RESERVE_INVENTORY", it calls the "handleOrder" function of the "productService" object
    to reserve inventory.
    If the order status is "REVERT_INVENTORY", it calls the "revertOrder" function of the "productService" object to
    revert inventory.
    Args:
    order (Order): An instance of the Order class.
    Returns:
    None
}
```

```

"""
if (OrderStatus.RESERVE_INVENTORY.equals(order.getOrderStatus())):
    # If the order status is "RESERVE_INVENTORY", call the "handleOrder" function of the "productService" object
    productService.handleOrder(order)
    .doOnSuccess(o -> {
        # If the operation is successful, send a message to the "orderProducer" object with the order status
        set to "INVENTORY_SUCCESS"
        orderProducer.sendMessage(order.setOrderStatus(OrderStatus.INVENTORY_SUCCESS));
    })
    .doOnError(e -> {
        # If the operation fails, send a message to the "orderProducer" object with the order status set to
        "INVENTORY_FAILURE" and the error message
        orderProducer.sendMessage(order.setOrderStatus(OrderStatus.INVENTORY_FAILURE)
            .setResponseMessage(e.getMessage()));
    }).subscribe();
elif (OrderStatus.REVERT_INVENTORY.equals(order.getOrderStatus())):
    # If the order status is "REVERT_INVENTORY", call the "revertOrder" function of the "productService" object
    to revert inventory
    productService.revertOrder(order)
    .doOnSuccess(o -> {
        # If the operation is successful, send a message to the "orderProducer" object with the order status
        set to "INVENTORY_REVERT_SUCCESS"
        orderProducer.sendMessage(order.setOrderStatus(OrderStatus.INVENTORY_REVERT_SUCCESS));
    })
    .doOnError(e -> {
        # If the operation fails, send a message to the "orderProducer" object with the order status set to
        "INVENTORY_REVERT_FAILURE" and the error message
        orderProducer.sendMessage(order.setOrderStatus(OrderStatus.INVENTORY_REVERT_FAILURE)
            .setResponseMessage(e.getMessage()));
    }).subscribe();
}

```

Isso também significa que podemos descartar com segurança alguns dos endpoints redundantes de nosso controlador agora. Essas mudanças são suficientes para alcançar a comunicação assíncrona em nosso aplicativo.

Serviço de entrega:

As mudanças no serviço de remessa são relativamente semelhantes ao que fizemos anteriormente com o serviço de inventário. O produtor da mensagem é o mesmo e o consumidor da mensagem é específico para a lógica de envio:

```
@KafkaListener(topics = "orders", groupId = "shipping")
public void consume(Order order) throws IOException {
    if (OrderStatus.PREPARE_SHIPPING.equals(order.getOrderStatus())) {
        shippingService.handleOrder(order)
            .doOnSuccess(o -> {
                orderProducer.sendMessage(order.setOrderStatus(OrderStatus.SHIPPING_SUCCESS)
                    .setShippingDate(o.getShippingDate()));
            })
            .doOnError(e -> {
                orderProducer.sendMessage(order.setOrderStatus(OrderStatus.SHIPPING_FAILURE)
                    .setResponseMessage(e.getMessage()));
            });
    }
}
```

Podemos descartar com segurança todos os endpoints em nosso controlador agora, pois não precisamos mais deles.

Serviço de pedidos:

As mudanças no serviço de pedidos serão um pouco mais complicadas, pois é onde estávamos fazendo toda a orquestração anteriormente.

No entanto, o produtor da mensagem permanece inalterado e o consumidor da mensagem assume a lógica específica do serviço de pedido:

```
/*
 * This method listens to the "orders" topic and consumes the messages received.
 * It updates the order status and response message based on the received message.
 *
 * @param order: The Order object received from the Kafka message.
 * @throws IOException: If there is an error while consuming the message.
 */
@KafkaListener(topics = "orders", groupId = "orders")
public void consume(Order order) throws IOException {
    // Check if the order status is INITIATION_SUCCESS
    if (OrderStatus.INITIATION_SUCCESS.equals(order.getOrderStatus())) {
        // Find the order by ID and update the order status to RESERVE_INVENTORY
        orderRepository.findById(order.getId())
            .map(o -> {
                // Send a message to the order producer to update the order status to RESERVE_INVENTORY
                orderProducer.sendMessage(o.setOrderStatus(OrderStatus.RESERVE_INVENTORY));
                // Update the order status and response message
                return o.setOrderStatus(order.getOrderStatus())
                    .setResponseMessage(order.getResponseMessage());
            })
            .flatMap(orderRepository::save)
            .subscribe();
    }
    // Check if the order status is INVENTORY-SUCCESS
    else if ("INVENTORY-SUCCESS".equals(order.getOrderStatus())) {
        // Find the order by ID and update the order status to PREPARE_SHIPPING
        orderRepository.findById(order.getId())
            .map(o -> {
                // Send a message to the order producer to update the order status to PREPARE_SHIPPING
                orderProducer.sendMessage(o.setOrderStatus(OrderStatus.PREPARE_SHIPPING));
            })
            .flatMap(orderRepository::save)
            .subscribe();
    }
}
```

```
// Update the order status and response message
return o.setOrderStatus(order.getOrderStatus())
    .setResponseMessage(order.getResponseMessage());
})
.flatMap(orderRepository::save)
.subscribe();
}

// Check if the order status is SHIPPING-FAILURE
else if ("SHIPPING-FAILURE".equals(order.getOrderStatus())) {
    // Find the order by ID and update the order status to REVERT_INVENTORY
    orderRepository.findById(order.getId())
        .map(o -> {
            // Send a message to the order producer to update the order status to REVERT_INVENTORY
            orderProducer.sendMessage(o.setOrderStatus(OrderStatus.REVERT_INVENTORY));
            // Update the order status and response message
            return o.setOrderStatus(order.getOrderStatus())
                .setResponseMessage(order.getResponseMessage());
        })
        .flatMap(orderRepository::save)
        .subscribe();
}

// For any other order status, update the order status and response message
else {
    orderRepository.findById(order.getId())
        .map(o -> {
            return o.setOrderStatus(order.getOrderStatus())
                .setResponseMessage(order.getResponseMessage());
        })
        .flatMap(orderRepository::save)
        .subscribe();
}
}
```

O **consumidor aqui está apenas reagindo a mensagens de pedido com diferentes status de pedido**. É isso que nos dá a coreografia entre diferentes serviços.

Por fim, nosso serviço de pedidos também terá que mudar para suportar esta coreografia:

```
public Mono<Order> createOrder(Order order) {
    /**
     * This function creates an order by saving the order to the database, sending a message to a producer,
     * and updating the order status. If an error occurs, it sets the order status to failure and returns
     * an error message.
     *
     * Args:
     *   order (Order): An instance of the Order class.
     *
     * Returns:
     *   Mono<Order>: A Mono object that emits the saved order.
     *
     * Raises:
     *   None.
     */
    return Mono.just(order)
        .flatMap(orderRepository::save)  # Save the order to the database
        .map(o -> {
            orderProducer.sendMessage(o.setOrderStatus(OrderStatus.INITIATION_SUCCESS));  # Send a message to the
producer
            return o;
        })
        .onErrorResume(err -> {
            return Mono.just(order.setOrderStatus(OrderStatus.FAILURE))  # Set the order status to failure
                .setResponseMessage(err.getMessage());  # Return an error message
        })
        .flatMap(orderRepository::save);  # Save the order to the database
}
```

Observe que isso é muito mais simples do que o serviço que tivemos que escrever com endpoints reativos na última seção. A coreografia assíncrona **geralmente resulta em um código muito mais simples**,

embora tenha o custo de consistência eventual e depuração e monitoramento complexos. Como podemos imaginar, nosso front-end não receberá mais o status final do pedido imediatamente.

Serviço de Orquestração de Contêineres:

A última peça do quebra-cabeça que queremos resolver está relacionada à implantação.

O que queremos na aplicação é uma ampla redundância e uma tendência de aumentar ou diminuir automaticamente dependendo da necessidade.

Já alcançamos a conteinerização de serviços por meio do Docker e estamos gerenciando dependências entre eles por meio do Docker Compose. Embora sejam ferramentas fantásticas por si só, elas não nos ajudam a alcançar o que queremos.

Portanto, **precisamos de um serviço de orquestração de contêineres que possa cuidar da redundância e escalabilidade em nosso aplicativo**. Embora existam várias opções, uma das mais populares inclui o Kubernetes. O Kubernetes nos fornece uma maneira independente de fornecedor de nuvem para obter implantações altamente escaláveis de cargas de trabalho em contêineres.

O Kubernetes envolve contêineres como o Docker em pods, que são a menor unidade de implantação. Além disso, podemos usar Deployment para descrever o estado desejado declarativamente.

A implantação cria ReplicaSets, que internamente é responsável por ativar os pods. Podemos descrever um número mínimo de pods idênticos que devem estar em execução a qualquer momento. Isso fornece redundância e, portanto, alta disponibilidade.

Vamos ver como podemos definir uma implantação do Kubernetes para nossos aplicativos:

The code provided is a Kubernetes deployment configuration file written in YAML format. It defines three deployments for three different services: inventory, shipping, and order.

Each deployment specifies the number of replicas to be created, the labels to be used for selecting the replicas, and the container image and port to be used for each replica.

Below is the code with extensive documentation and comments:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inventory-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      name: inventory-deployment
  template:
    metadata:
      labels:
        name: inventory-deployment
    spec:
      containers:
        - name: inventory
          image: inventory-service-async:latest
          ports:
            - containerPort: 8081
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: shipping-deployment
spec:
  replicas: 3
```

```
selector:
  matchLabels:
    name: shipping-deployment
template:
  metadata:
    labels:
      name: shipping-deployment
spec:
  containers:
    - name: shipping
      image: shipping-service-async:latest
      ports:
        - containerPort: 8082
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      name: order-deployment
  template:
    metadata:
      labels:
        name: order-deployment
  spec:
    containers:
      - name: order
        image: order-service-async:latest
        ports:
```

```
- containerPort: 8080
```

Aqui estamos declarando nossa implantação para manter três réplicas idênticas de pods a qualquer momento. Embora seja uma boa maneira de adicionar redundância, pode não ser suficiente para cargas variadas. O Kubernetes fornece outro recurso conhecido como [horizontal Pod Autoscaler](#), que pode **dimensionar o número de pods em uma implantação com base em métricas observadas**, como a utilização da CPU.

Observe que acabamos de abordar os aspectos de escalabilidade do aplicativo hospedado em um cluster Kubernetes. Isso não implica necessariamente que o próprio cluster subjacente seja escalável. A criação de um cluster Kubernetes de alta disponibilidade não é uma tarefa trivial e está além do escopo deste documento de estudos.

Sistema Reativo Resultante:

Agora que fizemos várias melhorias em nossa arquitetura, talvez seja hora de avaliar isso em relação à definição de um Sistema Reativo. Manteremos a avaliação em relação às quatro características de um Sistema Reativo que discutimos anteriormente no tutorial:

- *Responsivo* : A adoção do paradigma de programação reativa deve nos ajudar a alcançar o não-bloqueio de ponta a ponta e, portanto, um aplicativo responsivo
- *Resiliente* : a implantação do Kubernetes com ReplicaSet do número desejado de pods deve fornecer resiliência contra falhas aleatórias
- *Elástico* : o cluster e os recursos do Kubernetes devem nos fornecer o suporte necessário para sermos elásticos diante de cargas imprevisíveis

- *Orientado a mensagens* : ter toda a comunicação serviço a serviço tratada de forma assíncrona por meio de um corretor Kafka deve nos ajudar aqui

Embora isso pareça bastante promissor, está longe de terminar. Para ser honesto, a **busca por um sistema verdadeiramente reativo deve ser um exercício contínuo de melhorias** . Nunca podemos antecipar tudo o que pode falhar em uma infraestrutura altamente complexa, onde nosso aplicativo é apenas uma pequena parte.

Um sistema reativo, portanto, **exigirá confiabilidade de cada parte que faz o todo** . Desde a rede física até os serviços de infraestrutura, como o DNS, todos devem estar alinhados para nos ajudar a atingir o objetivo final.

Muitas vezes, pode não ser possível para nós gerenciar e fornecer as garantias necessárias para todas essas peças. E é aqui que **uma infraestrutura de nuvem gerenciada ajuda a aliviar nossa dor** . Podemos escolher entre uma série de serviços como IaaS (Infeastrure-as-a-Service), BaaS (Backend-as-a-Service) e PaaS (Platform-as-a-Service) para delegar as responsabilidades a terceiros. Isso nos deixa com a responsabilidade de nossa aplicação, tanto quanto possível.