UNIVERSIDADE DO SUL DE SANTA CATARINA

NIKOLAS CORTES ESTEVES – 1072419809

LUCAS MENDES LUIZ – 10724111935

MATHEUS INÁCIO SOUZA DA SILVA- 1072418296

ARTHUR DA SILVA MARQUES – 10723112096

MATEUS DUARTE DA ROSA – 1072412899

PROJETO PORTO RIO TUBARÃO

PROJETO A3

Orientador: *Pof Hélton Ribeiro Nunes*

Tubarão – Santa Catarina 2025

1. Introdução

As integrações assíncronas permitem que sistemas troquem informações sem a necessidade de resposta imediata. Isso é comumente realizado por meio de filas de mensagens (como RabbitMQ, Kafka, SQS) ou brokers que recebem a mensagem enviada e a entregam quando o destinatário estiver pronto. Dessa forma, o sistema remetente continua seu fluxo sem bloqueios, e o processamento pode ocorrer em segundo plano.

Essa abordagem traz várias vantagens. Primeiro, aumenta a resiliência dos sistemas, pois o remetente não depende da disponibilidade imediata do receptor e evita timeouts. Em caso de falhas do sistema receptor, as mensagens ficam na fila até serem processadas, garantindo alta disponibilidade. Além disso, é possível processar os dados em lote, otimizando o uso de recursos computacionais, ao invés de tratar cada mensagem isoladamente .

Outro benefício é o desacoplamento entre sistemas: os componentes que enviam e recebem mensagens operam de forma independente. Isso facilita a escalabilidade horizontal – cada serviço pode crescer conforme sua demanda, sem impactar diretamente os demais. Em arquiteturas de microsserviços, esse desacoplamento ajuda a evitar falhas cascata; se um serviço estiver fora do ar, o restante continua funcionando normalmente.

Existem no entanto algumas desvantagens que merecem atenção. Por não haver feedback imediato, é necessário informar aos usuários que o resultado será processado posteriormente, o que exige um design cuidadoso da interface . A latência inerente pode levar à impressão errônea de que o sistema falhou . Além disso, lidar com filas, garantir ordenação, garantir processamento idempotente e implementar sistemas de retry e monitoramento adiciona complexidade ao desenvolvimento.

Em termos práticos, integrações assíncronas são ideais quando o tempo de resposta não é crítico – por exemplo, na sincronização entre CRM e ERP, processamento de relatórios, indexação de dados ou workflows de transcrição. Já integrações síncronas continuam sendo mais apropriadas quando o usuário espera uma resposta imediata, como em APIs REST para exibição instantânea de dados.

Dessa forma, as integrações assíncronas são fundamentais em arquiteturas modernas por oferecerem resiliência, escalabilidade, eficiência e desacoplamento, mesmo que exijam mais cuidado no design dos fluxos, tratamento de exceções e comunicação com o usuário sobre o status das operações.

1. Desenvolvimento

Integrações assíncronas são fundamentais em sistemas distribuídos e aplicações modernas que exigem alta escalabilidade e resiliência. Ao contrário das integrações síncronas (que exigem resposta imediata), nas integrações assíncronas a comunicação entre sistemas ocorre de forma desacoplada, permitindo maior flexibilidade e desempenho.

2.1. Ferramentas para Integrações Assíncronas

A seguir, apresentamos algumas ferramentas que permitem integrações assíncronas com suporte direto para a linguagem Java. Essas ferramentas são amplamente utilizadas em arquiteturas modernas, como microsserviços, e oferecem recursos para gerenciamento eficiente de mensagens, filas, eventos e streams de dados: Apache Kafka e RabbitMQ.

2.1.1. Apache Kafka

O Apache Kafka consiste em uma plataforma distribuída para transmissão e processamento de fluxos de dados em tempo real, originalmente desenvolvida pelo LinkedIn e posteriormente incorporada à Apache Software Foundation. Sua arquitetura foi concebida para suportar altas taxas de ingestão de dados com baixa latência, assegurando escalabilidade horizontal, tolerância a falhas e confiabilidade na comunicação entre sistemas. Tais características tornam o Kafka particularmente adequado para ambientes que exigem integração baseada em eventos e manipulação contínua de grandes volumes de dados. A seguir, apresentam-se os principais contextos de aplicação dessa tecnologia na prática:

2.1.1.1. Integração entre Sistemas (Event Streaming / ETL)

Kafka é usado como um barramento de eventos para conectar sistemas distintos. Em vez de um sistema depender diretamente de outro, eles se comunicam via tópicos Kafka, permitindo uma arquitetura desacoplada e escalável.

Exemplo: Sistema de pedidos envia eventos para um tópico; serviços de faturamento, estoque e notificação consomem esses eventos independentemente.

2.1.1.2. Streaming de Dados em Tempo Real

Ideal para processar e reagir a eventos conforme eles ocorrem. Ferramentas como Kafka Streams, Flink ou Spark podem ser usadas junto com Kafka.

Exemplo: Detecção de fraudes em tempo real em sistemas financeiros (análise de transações conforme ocorrem).

2.2.1.3. Recomendações em E-commerce

Kafka alimenta sistemas de recomendação com dados de cliques, buscas e compras em tempo real.

Exemplo: Amazon ou Mercado Livre usam Kafka para coletar eventos de navegação e adaptar ofertas na hora.

2.1.2. Empresas Onde Apache Kafka é Utilizado

2.1.2.1. Netflix

A Netflix utiliza o Apache Kafka como parte essencial de sua arquitetura de microsserviços. Com bilhões de eventos gerados por seus usuários (como play, pause, navegação e recomendações), a empresa precisa capturar e processar esses dados em tempo real. O Kafka serve como o backbone para transportar eventos entre sistemas, permitindo análises em tempo real, geração de relatórios e melhorias nos algoritmos de recomendação. Além disso, ele ajuda na detecção de falhas e no monitoramento da performance da plataforma.

2.1.2.2. LinkedIn

Criadora original do Kafka, o LinkedIn o utiliza para processar mais de trilhões de eventos por dia. Ele é fundamental para alimentar sistemas de análise de dados, monitoramento de atividade, sistemas antifraude e notificações. Os eventos produzidos pelos usuários (como publicações, curtidas, conexões e mensagens) são enviados para o Kafka, que os distribui para múltiplos consumidores interessados em processar ou armazenar esses dados em diferentes bancos ou sistemas analíticos.

2.1.2.3. Uber

O Uber usa o Kafka para manter a comunicação entre milhares de microsserviços em sua plataforma. Por exemplo, quando um passageiro solicita uma corrida, múltiplos eventos são gerados e precisam ser propagados: localização do motorista, status da viagem, pagamentos, atualizações em tempo real etc. O Kafka permite essa troca

assíncrona e eficiente de mensagens, garantindo escalabilidade mesmo com picos massivos de uso, como em horários de alta demanda ou eventos especiais.

2.1.2.4. Spotify

O Spotify também usa o Apache Kafka para processar eventos relacionados a ações dos usuários, como reprodução de músicas, playlists, likes, e uso do aplicativo em geral. Esses eventos são essenciais para alimentar os sistemas de recomendação musical, personalização de conteúdo e relatórios para artistas. O Kafka garante que todas essas informações sejam transmitidas com baixa latência e alta confiabilidade.

2.1.3. Exemplo: Netflix

O Keystone Pipeline é a principal arquitetura de coleta e processamento de dados em tempo real da Netflix, sendo responsável por transportar trilhões de eventos diariamente. Baseado no Apache Kafka, esse pipeline opera com dezenas de clusters e milhares de brokers espalhados globalmente, garantindo alta disponibilidade e escalabilidade. O Kafka atua como a espinha dorsal dessa estrutura, permitindo que diversos microsserviços da Netflix enviem e consumam eventos de forma assíncrona, com baixa latência e tolerância a falhas.

No Keystone, os produtores Kafka são configurados para priorizar desempenho, utilizando parâmetros como acks=1 (confirmação mínima), block.on.buffer.full=false (evita bloqueios quando o buffer está cheio) e envio assíncrono com callbacks que tratam apenas os erros, sem impactar o fluxo principal da aplicação. Isso garante que o sistema continue processando bilhões de mensagens por hora, mesmo sob picos extremos de uso, como durante lançamentos globais de séries ou filmes. Dessa forma, a Netflix consegue coletar, armazenar e encaminhar dados de navegação, reprodução, falhas, preferências e muito mais, alimentando sistemas de recomendação, monitoramento, antifraude e relatórios internos.

Exemplo de código:

```
@Resource(name = "EXAMPLE_VALIDATION_ALERTS", type =
ConsumerStorageType.DB)

public class ValidationAlert implements Annotatable {

   private final Long customerId;

   private final String reason;

   private final Long daysBehind;


   public ValidationAlert(Long customerId, String reason, long daysBehind) {

      this.customerId = customerId;
```

```
        this.reason = reason;
        this.daysBehind = Long.valueOf(daysBehind);
    }

    @com.netflix.logging.annotations.Column("CustomerId")
    public Long getCustomerId() {
        return customerId;
    }

    @com.netflix.logging.annotations.Column("Reason")
    public String getReason() {
        return reason;
    }

    @com.netflix.logging.annotations.Column("DaysBehind")
    public Long getDaysBehind() {
        return daysBehind;
    }
}
```

Este trecho de código define uma classe chamada ValidationAlert, usada para representar alertas de validação gerados em sistemas de processamento de dados, como os que podem existir no pipeline do Netflix com Kafka. A anotação @Resource no topo da classe especifica que este objeto é um recurso com nome "EXAMPLE_VALIDATION_ALERTS" e será persistido em um tipo de armazenamento do consumidor, indicado por ConsumerStorageType.DB, sugerindo que esses dados são salvos em um banco de dados. A classe implementa a interface Annotatable, provavelmente usada internamente para permitir que metadados sejam aplicados a objetos durante o processo de logging ou serialização. Os três atributos — customerId, reason e daysBehind — são campos imutáveis que representam, respectivamente, o ID do cliente, a razão do alerta e quantos dias está em atraso. O construtor recebe esses valores como parâmetros e inicializa os campos da instância. Já os métodos getCustomerId(), getReason() e getDaysBehind() são métodos de acesso (getters), todos anotados com @com.netflix.logging.annotations.Column, o que indica que esses valores serão registrados ou exportados como colunas identificadas — prática comum para logging estruturado e geração de eventos para sistemas como o Kafka. Em resumo, esta classe encapsula um evento que será transformado em uma mensagem logada ou persistida, sendo parte de uma arquitetura orientada a eventos em um sistema de alto desempenho.

2.1.4. RabbitMQ

O RabbitMQ é uma ferramenta de mensageria amplamente utilizada para viabilizar a comunicação assíncrona entre diferentes aplicações, especialmente em arquiteturas baseadas em microsserviços. Ele atua como um intermediário que recebe, armazena e encaminha mensagens entre produtores (quem envia) e consumidores (quem recebe), garantindo que os dados sejam entregues mesmo que o receptor não esteja disponível no momento do envio. Isso permite um alto grau de desacoplamento entre os sistemas, promovendo maior resiliência e escalabilidade.

Um dos usos mais comuns do RabbitMQ está na comunicação entre serviços distribuídos. Por exemplo, em uma aplicação de e-commerce, o serviço responsável por registrar pedidos pode enviar uma mensagem para o RabbitMQ informando sobre uma nova compra, e outros serviços — como o de estoque ou envio de e-mails — podem consumir essa mensagem e agir de acordo, sem que precisem conhecer diretamente o funcionamento uns dos outros. Isso permite que cada serviço evolua de forma independente.

Outro cenário muito comum é o uso de RabbitMQ como fila de tarefas em segundo plano. Em aplicações web, muitas vezes é necessário executar tarefas demoradas, como envio de e-mails ou geração de relatórios. Ao invés de fazer isso durante a requisição do usuário (o que deixaria a experiência lenta), a aplicação pode colocar a tarefa na fila do RabbitMQ e responder ao usuário imediatamente, enquanto outro processo — o "worker" — retira a tarefa da fila e a executa.

Além disso, RabbitMQ é bastante usado na orquestração de workflows em sistemas complexos. Ele permite que diferentes etapas de um processo sejam coordenadas por meio do envio e recebimento de mensagens. Por exemplo, após um usuário se cadastrar em um site, o sistema pode enviar uma mensagem para criar o perfil do usuário, outra para notificar a equipe interna e mais uma para disparar um e-mail de boas-vindas, tudo de forma desacoplada.

Por fim, o RabbitMQ também é uma excelente ferramenta para integração entre sistemas escritos em diferentes linguagens de programação. Isso porque ele suporta múltiplos protocolos de comunicação, como AMQP, MQTT e STOMP, permitindo que sistemas heterogêneos se comuniquem de forma padronizada e eficiente. Dessa forma, ele contribui não apenas para a escalabilidade, mas também para a interoperabilidade de soluções em ambientes corporativos diversos.

2.1.5. Empresas Onde RabbitMQ é Utilizado

2.1.5.1. Mozilla

A Mozilla usa o RabbitMQ em vários de seus serviços internos, como o sistema de atualização e sincronização de dados entre dispositivos (como o Firefox Sync), onde o desempenho e a confiabilidade da troca de mensagens são cruciais.

2.1.5.2. GitHub

O GitHub já utilizou o RabbitMQ para gerenciar tarefas em background, como notificações, análise de código, e integração contínua. A fila de mensagens ajuda a lidar com grande volume de eventos que ocorrem na plataforma.

2.1.5.3. Instagram (Facebook/Meta)

O Instagram utilizava RabbitMQ para processar tarefas assíncronas, como o envio de notificações, redimensionamento de imagens e processamento de vídeos — todos cenários em que o desempenho assíncrono melhora a experiência do usuário.

Exemplo de código RabbitMQ:

Classe Produtor.java

Esta classe será responsável por criar uma conexão com o RabbitMQ, declarar uma fila e enviar algumas mensagens para ela.

```
import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;


public class Produtor {


   private final static String NOME_DA_FILA = "minhaFilaDeExemplo";


   public static void main(String[] argv) throws Exception {
      ConnectionFactory factory = new ConnectionFactory();
      // Altere para o endereço do seu servidor RabbitMQ, se for diferente
      factory.setHost("localhost");
      factory.setPort(5672); // Porta padrão do RabbitMQ
      factory.setUsername("guest"); // Usuário padrão
      factory.setPassword("guest"); // Senha padrão
```

```
        try (Connection connection = factory.newConnection();
             Channel channel = connection.createChannel()) {

            // Declara a fila. Se ela não existir, será criada.
            // Argumentos: nome da fila, durável, exclusiva, auto-delete, argumentos
            channel.queueDeclare(NOME_DA_FILA, false, false, false, null);

            String[] mensagens = {
                "Olá, RabbitMQ!",
                "Esta é a segunda mensagem.",
                "Processar esta mensagem, por favor."
            };

            for (String mensagem : mensagens) {
                channel.basicPublish("", NOME_DA_FILA, null,
mensagem.getBytes("UTF-8"));
                System.out.println(" [x] Enviado '" + mensagem + "'");
                Thread.sleep(1000); // Pequena pausa para simular envio gradual
            }

            System.out.println(" [x] Todas as mensagens foram enviadas.");
        }
    }
}
```

Classe Consumidor.java

Esta classe irá se conectar ao RabbitMQ, se inscrever na mesma fila e consumir as mensagens que forem enviadas para ela.

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;
```

```
public class Consumidor {

    private final static String NOME_DA_FILA = "minhaFilaDeExemplo";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        factory.setPort(5672);
        factory.setUsername("guest");
        factory.setPassword("guest");

        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        // Declara a fila novamente para garantir que ela exista
        channel.queueDeclare(NOME_DA_FILA, false, false, false, null);
        System.out.println(" [*] Aguardando mensagens. Para sair, pressione CTRL+C");

        // Callback que será executado quando uma mensagem for recebida
        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String mensagem = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [x] Recebido '" + mensagem + "'");

            // Simula um processamento demorado
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                System.out.println(" [x] Concluído o processamento de '" + mensagem + "'");
            }
        };
```

```
        // Começa a consumir mensagens da fila
        // Argumentos: nome da fila, autoAck (reconhecimento automático), callback
        channel.basicConsume(NOME_DA_FILa, true, deliverCallback, consumerTag ->
{});
    }
}
```

2. Referências Bibliográficas

Bielsaar. Workersrabbit. GitHub, 2021. Disponível em: https://github.com/Bielsaar/workersrabbit. Acesso em: 04 jun. 2025.

BLOG, Netflix Technology. Kafka Inside Keystone Pipeline. Medium, 2016. Disponível em: https://netflixtechblog.com/kafka-inside-keystone-pipeline-dd5aeabaf6bb. Acesso em: 03 jun. 2025.

BUG, Café Com. RabbitMQ: Entenda de uma vez por todas. YouTube, 2024. Disponível em: https://www.youtube.com/watch?v=bN0_hk0BRwU. Acesso em: 03 jun. 2025.

CBOHimself. Service-Centric-and-Cloud-Computing-CWK. GitHub, 2022. Disponível em: https://github.com/CBOHimself/Service-Centric-and-Cloud-Computing-CWK. Acesso em: 04 jun. 2025.

KRALJ, Filip. Synchronous vs Asynchronous Integration: A Comprehensive Guide. Ebitools, 2023. Disponível em: https://www.ebitools.com/en/integrations/synchronous-vs-asynchronous-integration-a-comprehensive-guide?t.com. Acesso em: 02 jun. 2025.

MAHILANI, Nick. KafkaSink.java. GitHub, 2019. Disponível em: https://github.com/Netflix/mantis-connectors/blob/master/mantis-connector-kafka/src/main/java/io/mantisrx/connector/kafka/sink/KafkaSink.java?.com. Acesso em: 03 jun. 2025.

ManuelMaia165. Roteamento-de-filas-RabbitMQ. GitHub, 2022. Disponível em: https://github.com/ManuelMaia165/roteamento-de-filas-RabbitMQ. Acesso em: 04 jun. 2025.

SAEPULOH, Acep Muhamad. How can I create Progress for Springboot + Rabbit MQ?. StackOverflow, 2017. Disponível em: https://stackoverflow.com/questions/44017575/how-can-i-create-progress-for-springboot-rabbit-mq. Acesso em: 04 jun. 2025.

SF Big Analytics: Building/Running Netflix's Data Pipeline using Apache Kafka. YouTube, 2016. Disponível em: https://www.youtube.com/watch?v=6ocfbpxBobQ. Acesso em: 03 jun. 2025.

TOV, Jonathan Yom. Microservices Architecture: Asynchronous Communication is Better. sysaid, 2022. Disponível em: https://www.sysaid.com/blog/sysaid-tech/microservices-architecture-asynchronouscommunication-better?.com. Acesso em: 02 jun. 2025.