UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Isadora Rocha de Paula

Testes em tópicos de Kafka: um roteiro de como realizar

Uberlândia, Brasil

2022

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Isadora Rocha de Paula

Testes em tópicos de Kafka: um roteiro de como realizar

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: André Ricardo Backes

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2022

Isadora Rocha de Paula

Testes em tópicos de Kafka: um roteiro de como realizar

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 10 de agosto de 2022:

André Ricardo Backes
Orientador

Professor

Professor

Uberlândia, Brasil
2022

Agradecimentos

Agradeço aos meus pais por sempre me incentivarem a estudar e a seguir meus sonhos; a Deus por me conceder todas as oportunidades que tive de conhecimento durante o período de graduação; à minha tia Amália, que ajudou me apoiando e revisando esse trabalho. Agradeço a todo o corpo docente pelos ensinamentos durante o período de curso de Sistemas de Informação.

Resumo

O objetivo deste trabalho de conclusão de curso é apresentar uma forma de realizar testes para validar a publicação de mensagens em tópicos do Kafka, que é uma das dificuldades encontradas na área da qualidade. Kafka é uma ferramenta de código aberto que faz *streaming* de eventos e é muito utilizada por gigantes da Internet, como o LinkedIn, Netflix entre outras. Porém, apesar de o Kafka ser bastante utilizado, a forma de validar os eventos gerados é pouco conhecida pela equipe de qualidade, mesmo sendo necessária para garantir a qualidade do sistema.

O método escolhido foi a utilização de testes manuais e automatizados. Para testes manuais é apresentado neste trabalho uma ferramenta chamada Kafka Magic; e para a automação de testes foi montado um projeto de automação em Ruby que utiliza uma biblioteca de Kafka para conseguir conectar o projeto ao Kafka e realizar a validação necessária.

Também apresentamos os resultados alcançados, que foram realizar a demonstração de como fazer testes em tópicos do Kafka de maneira manual e automatizada. Foi possível observar que as ferramentas propostas para realização dos testes facilitam o trabalho do engenheiro de qualidade, tanto para acesso às informações, quanto em otimização do tempo de trabalho para execução dos testes.

Este trabalho traz também informações sobre a história do Kafka, seu funcionamento, sua capacidade e as possibilidades de uso.

Lista de ilustrações

Sumário

1 Introdução

A crescente necessidade de retorno imediato pelos usuários dos recursos de Internet (aplicativos, redes sociais, etc.) tem levado ao desenvolvimento de aplicações cada vez mais resilientes e que possam dar respostas em tempo real. Por isso, as empresas que conseguiram atender essa demanda do mercado atual cresceram e se destacaram, tais como Netflix, Spotify, LinkedIn e Nubank.

Para poderem crescer, essas empresas usaram a tecnologia de *streaming*, que consiste em disponibilizar informações em tempo real, algumas delas utilizando o Apache Kafka. Esta plataforma permite que dados sejam transmitidos entre clientes e servidores em alta performance com protocolo TCP, evitando assim que haja qualquer tipo de perda de dados no caso de falha de algum servidor.

Apesar de ser uma ferramenta muito utilizada, principalmente por grandes empresas, tais como Yahoo, LinkedIn, Netflix, Oracle, entre outros, é pouco conhecida, por parte da equipe de qualidade, a forma de realizar a validação dos eventos gerados no Kafka. Por ser uma ferramenta de baixo nível, muitas vezes são realizados apenas testes unitários e de integração, o que nem sempre garante que essas informações estejam sendo enviadas ou recebidas adequadamente por um serviço terceiro. Isso pode resultar em gargalos no processamento de dados, uma vez que são desconhecidos pela aplicação.

Conhecendo a dificuldade de validação deste tipo de implementação por parte da equipe de qualidade de um time, este trabalho tem como objetivo apresentar uma forma de realizar o teste de eventos gerados pelo Kafka. Com isso, pretende-se garantir a qualidade de um sistema.

Dado que a maior parte dos profissionais da área de Tecnologia da Informação que atuam com Kafka são os desenvolvedores e não os engenheiros de qualidade, uma das opções de ferramenta de teste apresentadas neste trabalho tem como finalidade tornar o trabalho dos engenheiros de qualidade bem mais “visual”, para aqueles que têm pouca familiaridade com o Kafka. Espera-se, assim, facilitar a compreensão da ferramenta e a melhoria da atuação dentro de uma equipe de desenvolvimento para validação das alterações, pois para muitos o Kafka é algo obscuro. Em geral, quando os engenheiros de qualidade precisam realizar a validação de alguma alteração no sistema do Kafka, apenas validam o seu bom funcionamento, sem de fato verificar a produção de eventos e criação de mensagens no Kafka.

Se uma informação errada passar adiante e ficar registrada de maneira incorreta, pode haver muita demora para identificá-la e resolver tudo o que já foi processado, com risco de que a informação se perca.

A vantagem de o pessoal da qualidade conseguir entender e testar os sistemas com as ferramentas do Kafka é de fato agregar mais na equipe de desenvolvimento, não deixando apenas a cargo do desenvolvedor realizar esse teste, tendo assim uma validação a mais, evitando *bugs*. Pessoas que trabalham repetidamente em um sistema agir de forma automática e assim deixar de perceber alguns problemas que venham a surgir ou podem ainda esquecer de executar alguns passos. A verificação automatizada nunca vai ignorar os testes, sempre fará o que foi determinado e configurado no código. Além disso, ela vai gerar um relatório com detalhes importantes que vão ajudar a detectar e corrigir problemas (caso sejam encontrados), registrando as evidências de erros e acertos.

Enfim, essas ferramentas de testes servem para facilitar o trabalho dos profissionais da qualidade, independentemente de seu conhecimento prévio sobre o Kafka.

O restante desta monografia encontra-se organizado da seguinte forma: no Capítulo 2, é apresentado o referencial teórico que serviu de base para fundamentação da proposta; no Capítulo 3, é detalhado o funcionamento do Kafka; no Capítulo 4 descreve-se como realizar testes no Kafka e sugestões de ferramentas a serem utilizadas.

2 Referencial teórico

2.1 A arquitetura de microsserviços

Atualmente, as empresas de desenvolvimento ou de consultoria têm optado por implementar a arquitetura de microsserviços em seus sistemas. Isso significa que, para cada função existente dentro da aplicação, pode-se implementar um serviço de forma independente das demais funções daquele sistema. Essa arquitetura permite que cada funcionalidade opere com sucesso ou falha sem comprometer as demais. Apesar da independência de cada microsserviço, eles precisam se comunicar entre si para que a execução de um fluxo acionado pela ação de um usuário seja concluída com êxito.

Como exemplo, imagine um site de compras. Nele é possível pesquisar produtos, adicionar produtos ao carrinho de compras, adicionar uma forma de pagamento e concluir a compra. Cada uma dessas funcionalidades é executada por um serviço independente. Portanto, caso o serviço de busca de produtos fique indisponível por um tempo e existam produtos no carrinho, que o usuário deseja comprar, a ação de compra poderá ser concluída sem que o site fique fora do ar, graças à independência entre os serviços.

As vantagens de trabalhar com essa arquitetura são várias (HAT, 2018). Dentre elas, além da resiliência citada no exemplo anterior, também se destacam:

- O tempo de implementação reduzido: a independência dos microsserviços permite a paralelização de desenvolvimento de cada funcionalidade que irá compor uma determinada aplicação;
- A facilidade de implementação: como as aplicações baseadas em microsserviços são menores, as preocupações de causar danos em outra funcionalidade do sistema são menores;
- A acessibilidade: como a aplicação é decomposta em partes menores, permite que uma equipe de desenvolvimento entenda com maior facilidade cada uma dessas partes, resultando em ciclos mais rápidos de entregas e aprimoramentos.

2.2 Entendendo APIs

Como foi explicado anteriormente, a arquitetura de microsserviços consiste em serviços independentes que se comunicam entre si para realizar um determinado fluxo. Neste tipo de arquitetura é comum utilizar a implementação de APIs (*Application Programming*

Interfaces, em português: interfaces de programação de aplicativos). Cada função dentro de um sistema é executada por uma API, que é vista como um serviço.

Como é citado no site oficial da Red Hat (HAT, 2020a), API é um conjunto de definições e protocolos utilizados no desenvolvimento e na integração de software de aplicações, permitindo que uma solução ou serviço tenha comunicação com outros produtos e serviços, sem necessidade de saber como eles foram implementados (HAT, 2017b). Essa estrutura simplifica o desenvolvimento, permitindo economia de tempo e de recursos, dado que cada time (ou cada recurso) pode atuar no desenvolvimento de uma API de forma independente e paralela.

Por conta do conceito de serviço independente, é possível disponibilizar uma API para qualquer pessoa que queira utilizar esse recurso em sua aplicação. Nesse caso, são nomeadas como APIs públicas. Elas são bem comuns no mercado pois, além de agregarem um sistema, são capazes de executar uma determinada função necessária, reduzindo o tempo de desenvolvimento de uma nova funcionalidade para exercer o mesmo papel. Elas também ajudam na expansão da marca que as criou e facilitam a inovação por meio da colaboração aberta ao público. O Google é um exemplo de empresa que disponibiliza algumas APIs públicas, como o Google Maps e o *captcha* de autenticação.

2.3 Arquitetura orientada a eventos

Event-driven Architecture - EDA (arquitetura orientada a eventos), ou simplesmente *Event-driven*, é um padrão de design arquitetural onde a comunicação entre os componentes é de natureza assíncrona, voltada a eventos, feita com base nos *streams* de eventos (GONÇALVES, 2020). Ela permite autonomia e independência aos componentes, possibilitando uma interação mais ágil e mais inteligente. Os eventos são essenciais para a EDA, pois eles vão atuar como contextualizadores das atualizações das aplicações, podendo direcionar a forma como o software será estruturado.

Para exemplificar melhor como é a arquitetura orientada a eventos podemos retomar o exemplo do site de compras, já que muitos sistemas com arquitetura orientada a microsserviços também podem possuir uma arquitetura orientada a eventos. Então suponhamos que um usuário tenha finalizado a compra. O ato de finalizar a compra (*checkout*) gera um evento que identifica o pagamento dos itens, que pode ser feito em paralelo caso necessário. Estes eventos podem ser consumidos por aplicações diferentes e em paralelo, como o envio de itens, a emissão da nota fiscal e o controle de estoque. A confirmação do pagamento também não precisa acontecer de forma instantânea, por isso ela é assíncrona.

Por outro lado, caso aconteça algum problema no microsserviço responsável pelo estoque, o histórico de eventos fica salvo em outro diretório, como será explicado na sessão 3.3. Assim que esse microsserviço for corrigido pela equipe de desenvolvimento,

ele será capaz de reprocessar as informações acessando esses dados a partir do diretório responsável por guardar os dados dos eventos (tópicos) gerados na aplicação.

Marcelo M. Gonçalves diz que "o principal valor agregado ao negócio ao adotar-se EDA seria a facilidade de estender o ecossistema com novos componentes, de forma modular, prontos para reagir a eventos existentes ou produzir novos sem o risco de comprometer as implementações existentes e seu funcionamento." (GONÇALVES, 2020).

Para entender o que é a arquitetura orientada a eventos é necessário entender o que é comunicação síncrona e assíncrona de serviços, pois a EDA, por muitas vezes, utiliza uma comunicação assíncrona. A comunicação síncrona acontece quando os serviços estão em sincronia, ou seja, um serviço A chama um serviço B e fica esperando a resposta de B. Por outro lado, na comunicação assíncrona, a troca de informação não precisa acontecer instantaneamente, ou seja, um serviço A chama B e não espera a resposta de B para continuar sua execução.

A arquitetura orientada a eventos funciona utilizando plataformas de transmissão assíncrona, pois ela utiliza produtores e consumidores. Um produtor cria um evento (criação de dados) que ocorre em um sistema, registra-o em forma de mensagem e o disponibiliza numa plataforma de transmissão sem direcionar essa mensagem especificamente a um consumidor (ou destinatário). O consumidor, por sua vez, se inscreve para receber mensagens de determinados produtores. Os sistemas de produtores/consumidores possuem um sistema chamado de *broker*, que é o ponto central onde as mensagens são publicadas para facilitar o acesso a esses dados por outros sistemas.

Neste contexto, evento é qualquer mudança de estado de uma aplicação. Se recapitularmos o exemplo do site de compras, quando o carrinho é criado pela inserção do primeiro item, o evento é a criação do carrinho. Quando outro item é adicionado, o evento é a alteração da quantidade de itens do carrinho. E outro evento pode ser registrado, caso um item do carrinho seja retirado ou até mesmo se o carrinho de compras for excluído.

Para que estes acontecimentos fiquem registrados em ordem cronológica, deixando claro qual é o estado atual dos dados, é necessário utilizar o *Event sourcing*, ou fonte de eventos. Isso ocorre porque em muitos casos surgem mudanças rápidas no estado de uma aplicação e por isso existe uma alta quantidade de dados que podem ser registrados pelos produtores de eventos. O *event sourcing* organiza esses eventos de forma cronológica para que os consumidores consigam registrar a ordem correta do acontecimento de cada evento, registrando assim o estado final correto da aplicação.

O site oficial da Red Hat levantou alguns benefícios de se utilizar uma arquitetura orientada a eventos (HAT, 2020b; HAT, 2019). Dentre eles, encontra-se a capacidade de ter um sistema flexível que pode adaptar-se a mudanças e tomar decisões em tempo real, pois decisões de negócio são tomadas a partir de dados que informam o estado atual

da aplicação ou do sistema, devido ao compartilhamento de informações dos produtores para os consumidores. Outra grande vantagem desta arquitetura é um melhor tempo de resposta da aplicação, por conseguir receber eventos simultaneamente e disponibilizar informações de um evento em um único lugar para que vários sistemas possam ter acesso ao mesmo tempo. Assim, o uso do tempo é otimizado, porque o atendimento das requisições enviadas não sofre bloqueios, como costuma acontecer nos microsserviços de comunicação síncrona.

Por sua vez, *Event Sourcing* (GONÇALVES, 2020) é o modo como os dados são estruturados nos bancos de dados. Na maioria dos bancos de dados não relacionais os valores antigos são substituídos pelos valores novos quando há uma mudança. No *Event Sourcing*, todas as alterações que acontecem são registradas individualmente como um evento anexado ao banco de dados. Assim, cada alteração se torna um evento imutável que pode ser usado como informação histórica da movimentação, o que abre um grande leque de possibilidades de utilização, ao contrário do que aconteceria se os dados anteriores fossem apagados durante a subscrição de novos dados. (GONÇALVES, 2020)

2.4 Event-Streaming

Para entender como o Kafka funciona, primeiramente é necessário entender o contexto de *streaming*.

Plataformas que são capazes de transmitir dados em tempo real sem a necessidade de realizar o download do conteúdo completo para interação com o usuário são conhecidas como plataformas de *streaming*. Exemplos bastante conhecidos são Netflix, Spotify, Disney+, entre outros. Os usuários conseguem interação em tempo real com esses conteúdos devido à capacidade de transmissão contínua de dados do servidor para a plataforma em uso. As formas mais conhecidas de dados são em áudio e vídeo, porém no campo de desenvolvimento a transmissão de dados em tempo real é utilizada em muitos outros cenários. Este capítulo vai explicar como essa tecnologia é utilizada para outras finalidades.

No contexto do desenvolvimento, *streaming* nada mais é do que um fluxo contínuo de dados, divididos em pequenas partes de forma independente e enviados separadamente, para serem trabalhados de diversos modos. Uma analogia, para facilitar o entendimento, seria imaginar uma avenida onde passam muitos carros e eles nunca param de passar.

A passagem do carro é o evento. Cada carro é um dado. Dentre eles, serão observados quantos carros vermelhos passam em uma hora. *Streaming* são os carros (dados) passando. A janela de tempo (uma hora) é onde deve ser aplicada a filtragem. Os carros vermelhos são o filtro. Então, *stream processing* é a análise dos dados que estão passando pelo *stream*. Assim, podemos dizer que um *Event Stream Processing - ESP* nada mais é do que o monitoramento em tempo real de dados que estão constantemente em mudança

(eventos).

Uma plataforma de *Event Stream* é aquela que consegue interagir em diferentes janelas de eventos de *streaming* (LUCKHAM, 2021)(LUCKHAM, 2020). *Streaming de eventos* (TIBCO, 2022) é o procedimento para transportar dados de eventos de um lugar a outro com eficiência, de modo que fiquem disponíveis para serem acessados e analisados por outros sistemas. Vale lembrar que no *event processing* (processamento de eventos) os eventos são examinados um por vez, enquanto que no *event stream processing* o trabalho é feito com muitos eventos relacionados juntos. O fluxo de eventos é, portanto, um componente da execução de *Event Stream Processing* (TIBCO, 2022).

Explicando melhor, *streaming* é um conceito usado para representar um conjunto de dados ilimitado e sempre crescente, pois novos registros chegam continuamente. Um *streaming* de eventos pode representar praticamente todas atividades de negócios a serem analisadas, como envio de e-mails, operações com cartões de crédito, rastreamento de encomendas, ou seja, há uma infinidade de exemplos, pois praticamente tudo que é feito em rede supõe uma sequência de eventos. (NARKHEDE; SHAPIRA; PALINO, 2017)

3 Apache Kafka

Diversas aplicações funcionam com a criação, recebimento, envio e manipulação de dados, seja por atividades do usuário, métricas, *logs*, entre outros. Para saber a história por detrás de cada dado trafegado em uma aplicação é necessário que haja o gerenciamento desses dados. Para isso existem ferramentas específicas para esse tipo de tarefa, sendo uma delas o Apache Kafka, que neste trabalho será chamado somente de Kafka. Neste capítulo veremos como ele surgiu, o que ele é e como ele é capaz de fazer o gerenciamento desses dados dentro de uma aplicação.

Dentre muitas ferramentas destinadas a trabalhar com eventos de *streaming*, o Apache Kafka, desenvolvido pela Apache Software Foundation, é o exemplo de ferramenta de *streaming* de eventos deste trabalho. Ele suporta a movimentação de enormes volumes de dados não apenas entre dois pontos, mas entre qualquer quantidade de pontos que forem necessários, ao mesmo tempo (HAT, 2017a). Ele também é usado por grandes empresas tais como *The New York Times*, Cisco, Nextflix, Oracle, entre outros (KAFKA, 2022). E por fim, muitos profissionais da área da qualidade enfrentam dificuldade pra entender e testar aplicações que fazem o uso do Kafka por falta de material na língua portuguesa e que também fale sobre como realizar testes em Kafka, por isso a ideia deste trabalho é explicar de forma sucinta o que é o Kafka, como ele funciona e como ele pode ser testado.

De acordo com Martin Kleppmann (RABELO, 2020), engenheiro de software e empresário, *stream processing* é uma ferramenta com muitas facetas, destinada a trabalhar os dados em tempo real, de forma a tornar um aplicativo mais escalável, mais confiável e mais sustentável. A ideia de estruturar dados como um fluxo de eventos não é novidade e seus princípios têm sido utilizados em muitos lugares, de vários modos e com nomenclatura diversificada. Ele conta que esta ideia surgiu no início dos anos 2000, em empresas da Internet como o LinkedIn, a partir das pesquisas em banco de dados.

O nome deste aplicativo veio de uma correlação entre o fato de ele ser otimizado para escrever e o fato de um de seus criadores gostar muito do escritor Franz Kafka, o qual conheceu nas aulas de literatura em sua faculdade.

3.1 A origem do Kafka

O Kafka surgiu dentro do LinkedIn para resolver o problema de coleta de métricas desta rede social. Até as tarefas mais simples do LinkedIn necessitavam de intervenção humana. Além disso, seu sistema era impreciso e tinha muitos nomes diferentes para

definir o mesmo tipo de métrica, conforme é relatado no livro *Kafka: The Definitive Guide* (NARKHEDE; SHAPIRA; PALINO, 2017).

Desta forma, para entender como o Kafka surgiu, primeiro é necessário conhecer um pouco do LinkedIn, pois sua história e a do Kafka são totalmente conectadas. Na definição do próprio LinkedIn, ele é a maior rede profissional do mundo, registrando atualmente mais de 774 milhões de usuários em cerca de 200 países. Sua missão é conectar profissionais e torná-los mais produtivos e bem-sucedidos (LINKEDIN, 2022).

Lançado na Internet em maio de 2003, o LinkedIn pertence à Microsoft desde 2016 e atualmente é considerado o maior serviço de nuvem profissional e a maior rede profissional e de negócios do planeta, com grande diversificação de negócios e faturamento advindo principalmente do seu uso para recrutamento de recursos humanos, assinaturas e venda de publicidade.

Esta rede possibilita que os usuários criem seus currículos, encontrem oportunidades de trabalho e estejam em contato profissional com pessoas de diversas partes do mundo. As empresas também a utilizam para fazer contratações para seu quadro de pessoal e para busca ativa de potenciais clientes.

Jeff Weiner, CEO do LinkedIn, explica que o Kafka foi criado para resolver o problema do *pipeline*[1] de dados no LinkedIn, tendo sido projetado para fornecer um sistema de mensagens de alta performance, capaz de lidar com vários tipos de dados e de fornecer em tempo real dados claros, construídos sobre as ações dos usuários e as métricas do sistema (NARKHEDE; SHAPIRA; PALINO, 2017). Porém, no início de suas operações, o LinkedIn tinha muitos problemas. A coleta de métricas era baseada em pesquisas e era ineficiente, pois havia grandes intervalos entre elas. Os proprietários de aplicativos não podiam gerenciar as métricas de acordo com suas necessidades.

Até então, o LinkedIn fazia o rastreamento das atividades dos usuários por meio de um serviço HTTP, no qual os servidores *front-end* se conectavam periodicamente e publicavam as mensagens em lotes, no formato XML. Esses lotes eram levados para o processamento *off line*, onde os arquivos eram avaliados e organizados.

Porém, a formatação XML era inconsistente e sua análise era computacionalmente cara, além do fato de que o rastreamento era arquitetado em lotes de hora em hora, e não em tempo real, entre muitos outros problemas, como o formato dos dados, o modelo de sondagem para monitoramento, dificuldade em usar o sistema para métricas e horas de atraso no processamento de lotes de atividades.

Para tentar solucionar essas falhas houve uma testagem massiva das soluções de código aberto existentes até então, com foco em obter acesso aos dados em tempo real e a possibilidade de trabalhar com um tráfego mais intenso de mensagens. Os modelos

[1] Execução de processos e/ou tarefas um seguido do outro.

de sistemas usavam o ActiveMQ, porém esta configuração não era suficiente para lidar com a escala e não atendia as demandas do LinkedIn. Foram descobertas muitas falhas no ActiveMQ, o que tornou clara a necessidade de uma infraestrutura personalizada para o *pipeline* de dados, como explicam Narkhede, N.; Shapira, G. e Palino, T. (2017, p. 14-16) em seu Guia.

Então, a equipe da qual faziam parte optou pela criação de um sistema de mensagens específico para atender às exigências dos sistemas de monitoramento e rastreamento e que pudesse ser usado para escalar para o futuro.

Havia também os objetivos de usar um modelo *push-pull* para separar produtores e consumidores, proporcionando persistência para dados de mensagens dentro do sistema de mensagens, a fim de que aceitasse vários consumidores; aprimorar para alta taxa de transferência de mensagens e, por fim, dar condições para que o dimensionamento horizontal do sistema cresça na mesma proporção em que os fluxos de dados aumentem.

Desta forma, o Kafka nasceu para ser um sistema de mensagens de publicação/assinatura, tendo uma interface típica de sistemas de mensagens. Porém, sua camada de armazenamento é estruturada de forma mais semelhante à de um sistema de agregação de *logs*. Para aumentar sua eficácia no trabalho com as métricas e o rastreamento de atividades dos usuários, foi adotado o Apache Avro, que faz a serialização das mensagens. Isso levou à possibilidade de atuar com bilhões de mensagens por dia, o que fez com que o LinkedIn crescesse em mais de um trilhão de mensagens desde agosto de 2015, consumindo acima de um petabyte de dados diariamente.

Em resumo, o Kafka surgiu em 2010, sete anos depois do lançamento do LinkedIn, como um projeto de código aberto no *GitHub*. No ano seguinte, ele foi acolhido pela comunidade de código aberto como um projeto de incubadora da Apache Software Foundation, passando a se chamar Apache Kafka. Ele foi formado na incubadora em Outubro de 2012 e, a partir daí, vem sendo continuamente aperfeiçoado. Hoje o Kafka tem uma grande comunidade de contribuidores e *committers* fora do LinkedIn, sendo usado em alguns dos maiores *pipelines* de dados do mundo.

Em 2014 alguns membros saíram do LinkedIn e criaram a Confluent, com o objetivo de fornecer desenvolvimento, suporte empresarial e treinamento para o Apache Kafka. Juntas, as duas empresas, contando com contribuições cada vez maiores de outras empresas da comunidade de código aberto, continuam a desenvolver e manter o Kafka. Atualmente, ele é a primeira escolha para *pipelines* de *big data* (NARKHEDE; SHAPIRA; PALINO, 2017).

3.2 Definições e funcionalidades

Kafka é um sistema de mensagem que permite a escrita, leitura, armazenamento e processamento de dados de *streaming*, mantendo o controle de eventos. De acordo com os autores do livro *Kafka: The Definitive Guide*, ele é frequentemente descrito como um *log* de confirmação distribuído ou, mais recentemente, como uma plataforma de distribuição de *streaming*.

Entre as situações em que o Kafka pode ser usado estão o rastreamento de atividades e de comportamento de usuários em uma aplicação (motivo de origem desse sistema), ou seja, ele é capaz de saber quais páginas foram visitadas, quantos cliques um botão recebeu, alterações dentro de um perfil, etc. O Kafka também pode ser usado para enviar notificações aos usuários, colher métricas para sistemas de monitoramentos, emitir alertas, auxiliar na atualização de banco de dados à medida em que os dados chegam e por fim, fazer o processamento de *streaming*.

Frequentemente o Kafka é comparado com outras categorias de tecnologias, tais como sistemas de mensagens corporativas, sistemas de *big data* como o Hadoop[2] e ferramentas de integração de dados ou ETL (sigla em inglês para "extração, transformação, carregamento") [3].

Apesar de já existirem ferramentas que funcionam de uma forma similar ao Kafka, como RabbitMQ, MQSeries da IBM, ActiveMQ, entre outras, o LinkedIn viu a necessidade de incrementar outras funcionalidades que facilitariam o seu processo interno. Por isso, a diferença entre o Kafka e essas outras ferramentas está na capacidade de ele funcionar como um sistema distribuído moderno. Ao invés de executar uma quantidade massiva de mensagens individuais por vez, ele possui uma plataforma central que pode ser dimensionada para lidar com todos os fluxos de dados em uma empresa.

Além disso, ele também foi construído para guardar dados pelo tempo que for configurado durante o desenvolvimento da aplicação. A vantagem dessa funcionalidade consiste na replicação e persistência dos dados por quanto tempo for necessário. E por último, os recursos de processamento de fluxo no Kafka permitem calcular dinamicamente os fluxos derivados e conjuntos de dados, a partir de seus fluxos, com muito menos código, ao contrário dos sistemas de mensagens, que apenas as distribuem.

3.3 Funcionamento

De acordo com os criadores do Kafka, ele foi construído com o intuito de ser uma versão em tempo real do Hadoop, que é um sistema que possibilita o armazenamento e

[2] Plataforma de computação distribuída voltada para o processamento de grandes volumes de dados.
[3] Ferramentas de software que realizam a extração de dados de diversas aplicações.

o processamento periódico de dados de arquivos em grande escala. Da mesma maneira, o Kafka permite armazenar e processar continuamente os fluxos de dados, também em grande escala. Mas, diferente do Hadoop, que realiza o processamento de dados em lote, o Kafka, por realizar o processamento de dados contínuos e de baixa latência, é aplicável em todo tipo de sistema que alimenta diretamente um negócio. Como toda empresa produz eventos a todo momento, é um grande ganho quando elas possuem um sistema capaz de reagir a esses eventos à medida em que eles ocorrem.

Assim como o ETL, o Kafka também é uma ferramenta de integração de dados. Porém, diferente do ETL, ele possui uma arquitetura centrada em fluxo de dados e, por isso, é capaz de conectar sistemas e aplicativos de dados prontos para uso, além de alimentar aplicações personalizadas criadas especificamente para usar esse mesmo fluxo de dados.

De forma mais detalhada, o Kafka armazena dados em ordem, de forma durável, e é capaz de ler esses dados de forma precisa. Esses dados chegam em forma de mensagens. Cada mensagem trafegada na rede possui um custo de processamento. Para otimizar este processo e evitar uma sobrecarga excessiva na rede, o Kafka agrupa as mensagens que são produzidas por um mesmo tópico em lotes compactos, proporcionando a transferência e o armazenamento dos dados de forma mais eficiente. Quanto maior o lote, mais mensagens podem ser tratadas por unidade de tempo.

Para facilitar a leitura dessas mensagens, é recomendado que elas estejam adequadas dentro do formato de algum esquema, podendo ser em *Javascript Object Notation* (JSON) ou em *Extensible Markup Language* (XML), que são mais próximas da leitura humana. Para evitar problemas de versão dos esquemas, os programadores que adotam o Kafka usam uma estrutura de serialização do Hadoop, chamada Apache Avro, que fornece um formato de serialização compacto e não requer a geração de código quando as mensagens serializadas são alteradas, além de ter uma forte tipagem de dados e evolução de esquema, sendo compatível com versões anteriores e posteriores.

As mensagens que chegam ao Kafka são categorizadas e armazenadas em tópicos, portanto normalmente elas são produzidas para um tópico específico. Tais tópicos são equivalentes a um banco de dados ou a uma pasta de arquivos, onde essas mensagens são guardadas em fila (primeira a entrar, primeira a ser lida). Por exemplo: em um sistema de vendas online, pode-se categorizar essas mensagens de acordo com cada funcionalidade do sistema, obtendo-se tópicos relacionados aos eventos que acontecem no carrinho, outro tópico para os eventos que acontecem na parte de pagamento e assim por diante, armazenando mensagens por categoria conforme são produzidas no sistema.

Os tópicos também podem ser divididos em partições que são fisicamente separadas e essas podem ser hospedadas em servidores diferentes, permitindo assim que o tópico seja dimensionado horizontalmente, apresentando um desempenho muito melhor por não

depender de apenas um servidor.

Como pode ser observado na Figura 1, a capacidade da mesma mensagem chegar em todas as partições do tópico proporciona a resiliência do sistema. Em caso de falha ou problemas no servidor principal, há um backup desses dados em outro servidor permitindo que o Kafka continue a realizar a leitura e processamento de mensagens, de forma automática e sem impactar o sistema.

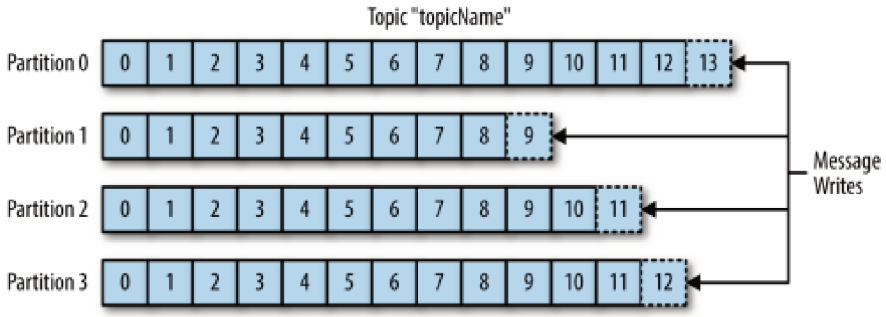

Figura 1 – Representação de um tópico com várias partições. (NARKHEDE; SHAPIRA; PALINO, 2017)

De acordo com o livro *Kafka: the definitive guide*, (NARKHEDE; SHAPIRA; PALINO, 2017), o Kafka possui dois tipos de usuários: os produtores e os consumidores. Os produtores, como explicado no Capítulo 4, produzem as mensagens e as direcionam para um tópico específico. Caso essas mensagens possuam uma chave *hash*, elas podem ser direcionadas para uma partição específica dentro do tópico. Porém, por padrão, o produtor não realiza esse direcionamento.

Os consumidores são os clientes que leem as mensagens criadas pelos produtores na ordem em que são produzidas. Eles também podem se inscrever para receber mensagens produzidas em mais de um tópico por vez.

Como forma de controle das mensagens que o consumidor já leu, um bit de metadados é adicionado a cada mensagem que é produzida, chamado *offset*. Portanto, o consumidor registra no próprio Kafka o *offset* da última mensagem consumida de cada partição. Se o consumidor parar e reiniciar, ele conseguirá consumir a próxima mensagem sem perder o lugar de onde parou.

Para trabalhar de forma mais eficiente, consumindo tópicos com um grande número de mensagens, os consumidores trabalham em grupos, onde cada partição fica responsável por apenas um membro do grupo, como pode ser ilustrado na Figura 2. Além da vantagem de se trabalhar em grupo quanto ao desempenho, também existe a vantagem de resiliência, pois em caso de falha de um dos consumidores, o grupo redistribui as partições entre os membros restantes para substituir o membro ausente.

Cada servidor do Kafka é chamado de *broker*. Portanto, cada mensagem gerada

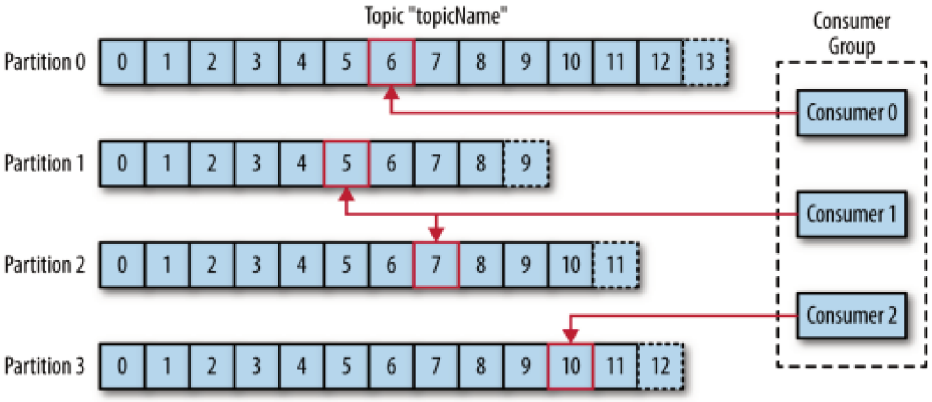

Figura 2 – Representação de um grupo de consumidores. (NARKHEDE; SHAPIRA; PALINO, 2017)

pelos produtores é armazenada em um tópico localizado dentro de um *broker* por um limite padrão de 7 dias ou por capacidade máxima de tópico de 1GB. Esses limites podem ser reconfigurados caso seja de interesse do desenvolvedor ou do cliente. E quando um desses limites é atingido, as mensagens são deletadas do tópico. Esse armazenamento tem como vantagem permitir que os consumidores não necessariamente precisem trabalhar em tempo real. Ou, caso os consumidores tenham algum problema de falha no processamento de dados, é possível acessar novamente as informações no tópico enquanto esses dados estão guardados no Kafka.

É válido lembrar que a função do tópico não é a de salvar mensagens como um banco de dados, mas sim mantê-las disponíveis por um tempo para que outras aplicações que tenham interesse nesses dados possam acessar e salvar essas informações.

O *broker* tem outras funções, como realizar a marcação de *offset* nas mensagens e confirmar o armazenamento da mensagem em disco. É ele também que atende os consumidores quando realizam a busca por partições, respondendo com as mensagens que foram armazenadas em disco.

Os *brokers* foram projetados para atuarem como parte de um *cluster*[4], funcionando como um controlador de *cluster*, atribuindo partições aos *brokers* e monitorando falhas dos mesmos. Uma partição pode ser atribuída a vários *brokers*, o que resultará na replicação da partição. Esse comportamento gera redundância de mensagens nas partições. Desta forma, se um *broker* falhar, outro *broker* é capaz de assumir seu lugar.

[4] Grupo de um ou mais computadores funcionando em conjunto.

4 Realizando testes no Kafka

Quando um profissional da área de qualidade começa a atuar em um projeto que faz o uso de Kafka, muitas vezes, ele não conhece essa ferramenta, enfrenta dificuldades de achar material na língua portuguesa e, às vezes, mais dificuldades ainda de achar material de como realizar os testes em tópicos de Kafka. E como consequência pode acabar afetando o desempenho no projeto e na qualidade do serviço. Por isso, este foi o tema escolhido para este trabalho de conclusão de graduação.

Em um dos projetos que esta aluna participou, houve a oportunidade de desenvolver uma forma de realizar esses testes, os quais não eram realizados antes. Para isso, foram utilizadas duas ferramentas: a primeira, chamada Kafka Magic, criada pela Apache especificamente para realizar esses testes; a segunda, foi uma *gem*[1] do Ruby, chamada também de Kafka.

Cada uma das ferramentas foi utilizada conforme a necessidade do projeto. A primeira facilita uma visualização geral das mensagens nos tópicos com possibilidade de usar *queries*[2] para realizar filtros das mensagens a serem validadas. A segunda ferramenta é utilizada em um projeto de automação que valida a publicação de uma mensagem específica no tópico correspondente de um fluxo de negócio.

Por motivos de *compliance*, não é possível exibir aqui a execução desses testes em grandes sistemas. Por isso, foi criada uma aplicação de exemplo para que fosse possível mostrar a execução do teste.

4.1 Kafka Magic

O Kafka Magic é uma ferramenta de interface gráfica, desenvolvida pela própria Apache especificamente para trabalhar com os *clusters* do Kafka. Como explica a documentação da ferramenta (MAGIC, 2019), ela possui várias funcionalidades, dentre elas: localizar e exibir mensagens, transformar e mover mensagens entre tópicos, revisar e atualizar esquemas, gerenciar tópicos e automatizar tarefas complexas. Por ser uma ferramenta paga, a única funcionalidade que será observada neste trabalho é a capacidade de localizar e exibir mensagens dos tópicos.

Para este trabalho, os testes utilizando o Kafka Magic serão para validar a publicação de mensagens no tópico de forma manual, pois é a funcionalidade que conseguimos usar na versão gratuita. A interface gráfica permite o acesso às mensagens produzidas

[1] Bibliotecas em Ruby.
[2] Comandos e consulta.

em cada tópico. Portanto, o que pode ser feito é criar filtros utilizando JavaScript para buscar exatamente o que será testado e validar se a mensagem foi produzida e consumida como o esperado.

O Kafka Magic é uma ferramenta cuja instalação e configuração é simples e pode ser feita seguindo o manual disponível em seu próprio site (MAGIC, 2019). Portanto, supondo que tudo esteja pronto e configurado na máquina, serão apresentados abaixo exemplos de conexão ao *cluster* do Kafka, realizando consultas em seguida.

A primeira parte é verificar em qual porta o Kafka Magic está funcionando, como ilustra a Figura 3.

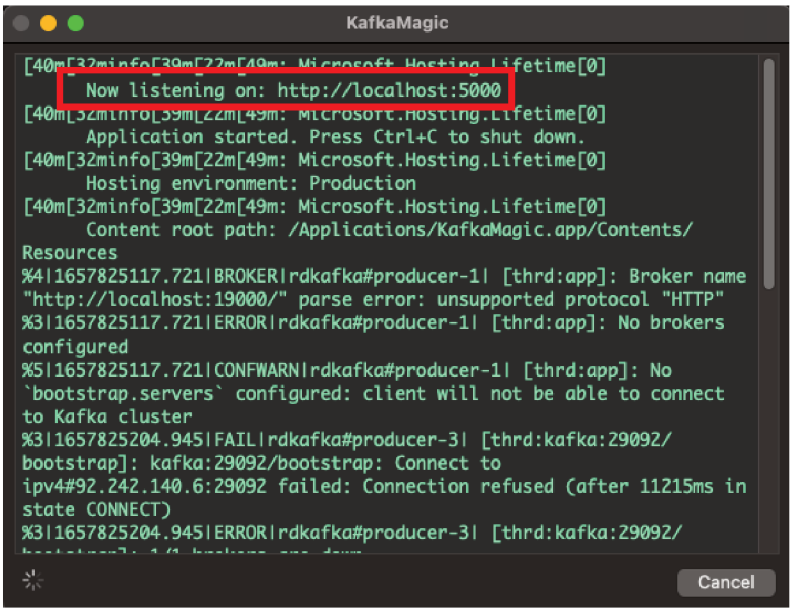

Figura 3 – Inicializando Kafka Magic.

Em seguida, acesse a interface por meio de um navegador, como mostra a Figura 4, acessando o *localhost* seguido do número da porta em que o Kafka Magic foi iniciado como ilustrado na Figura 3.

Para criar uma conexão com um Kafka Magic, clique em *Register new*, como indicado na Figura 5. Em seguida, faça a conexão criando um nome para o *Cluster* de acesso, identificado pelo número 1 da Figura 6. Esse nome pode ser de livre escolha. Em seguida, informe o local de acesso ao *Bootstrap*[3], identificado também pelo número 1 da Figura 6. Em alguns sistemas também é necessário informar o *Schema Registry*[4]. Ao final clique em *Verify* para verificar se a conexão foi bem sucedida, identificado pelo número 2 da Figura 6. E para concluir, clique em *"Register Connection"*, identificado pelo número 4 da Figura 6

[3] Porta que os *brokers* usarão para estabelecer a conexão inicial com o *cluster* Kafka.
[4] Ferramenta de código aberto utilizada para recuperar mensagens em *schema* Avro ou em JSON.

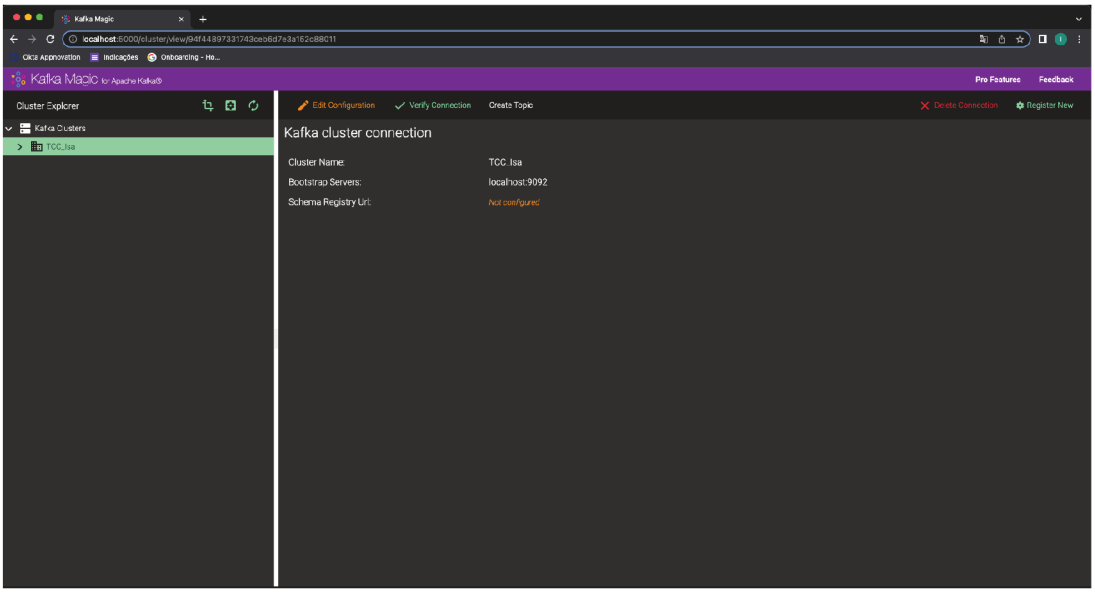

Figura 4 – Acessando o Kafka Magic por meio de um navegador.

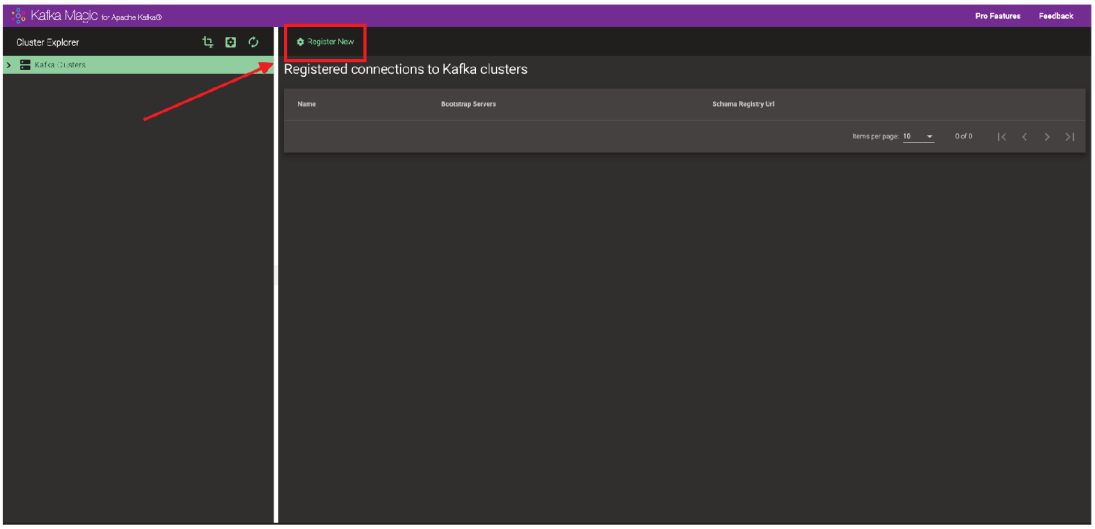

Figura 5 – Realizando configurações para conexão ao Kafka.

Após validar e registrar a conexão, será possível visualizar na barra lateral esquerda o nome do *cluster*, seguido dos tópicos existentes no Kafka conectado, como ilustra o número 1 da Figura 7. Apenas clicando no botão *Run Search*, item 4 da figura 7, ele irá trazer até 10 mensagens daquele tópico, podendo essa quantidade ser alterada no campo *Maximum Results*. O formato como o Kafka Magic mostra essas mensagens está disponível na Figura 8.

Para realizar testes, também é possível inserir alguns filtros utilizando JavaScript. Como mostra a Figura 8, foi feito um filtro para trazer apenas mensagens que contenham a palavra “oi”.

Para realizar a demonstração da consulta de geração de eventos no Kafka pelo

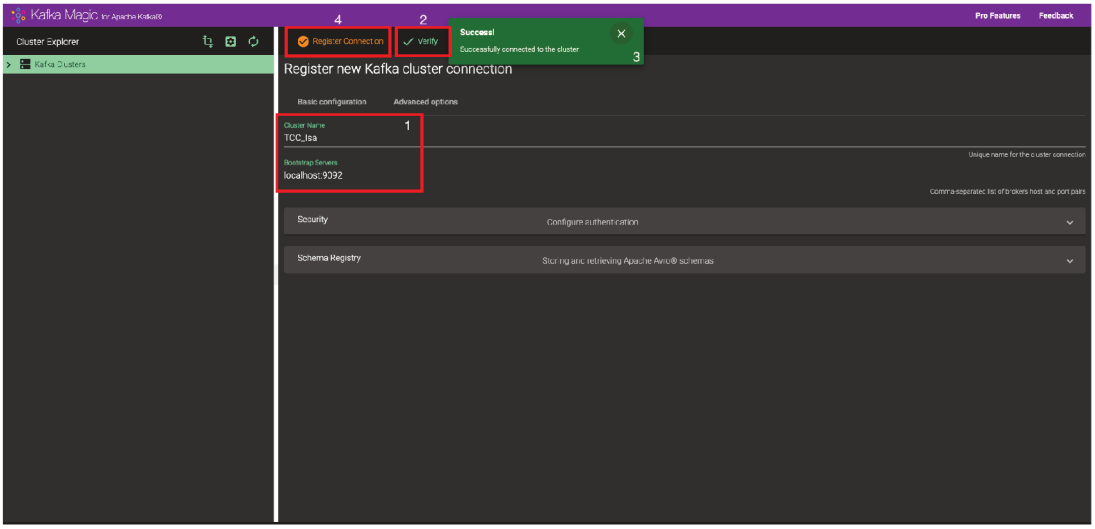

Figura 6 – Configurando uma conexão com o Kafka.

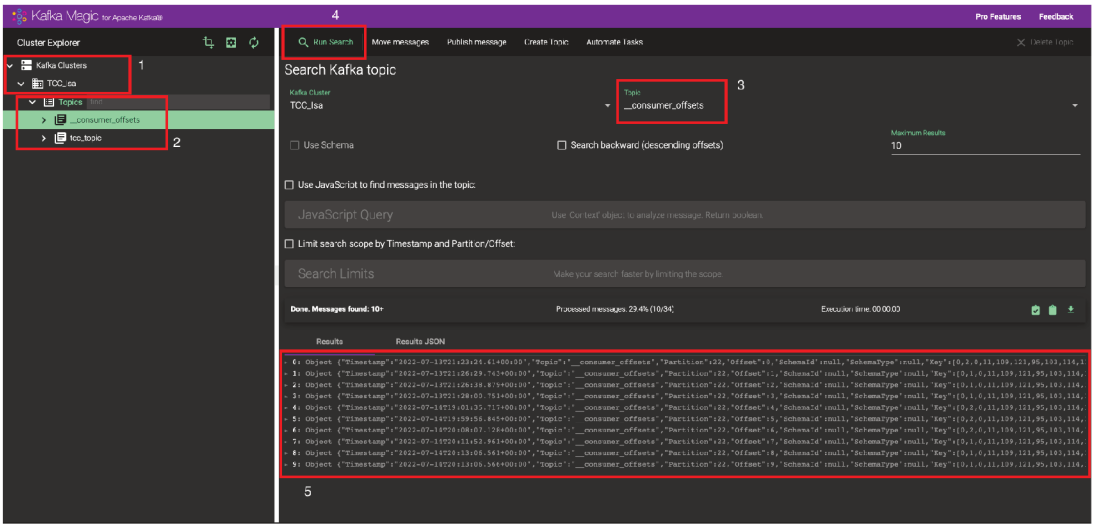

Figura 7 – Visualizando mensagens no tópico do Kafka.

Kafka Magic, foi utilizado uma plataforma de criação de uso de APIs chamada Postman (POSTMAN, 2022) para gerar os eventos no Kafka. Na área de qualidade, o Postman é muito utilizado para realizar testes de API tanto de forma manual quanto automatizada.

Foi utilizado um projeto de código aberto disponível em nuvem (SOUZA, 2022), em que ao rodar a aplicação localmente, foi possível realizar um POST no Kafka (local), por meio do Postman, informando qualquer texto do campo de mensagem para gerar um evento. A Figura 9 mostra a disposição da requisição POST pelo Postman. No campo de envio do corpo da mensagem foi utilizado um recurso da própria ferramenta para gerar nomes próprios de forma aleatória. A resposta da requisição se dá apenas por código de status, sem retornar um corpo de resposta. Na demonstração da Figura 9, o código de retorno foi “200, ok”. Ou seja, a requisição foi feita com sucesso, pois qualquer resposta HTTP que tenha o código de status na casa dos 200 significa uma resposta de sucesso

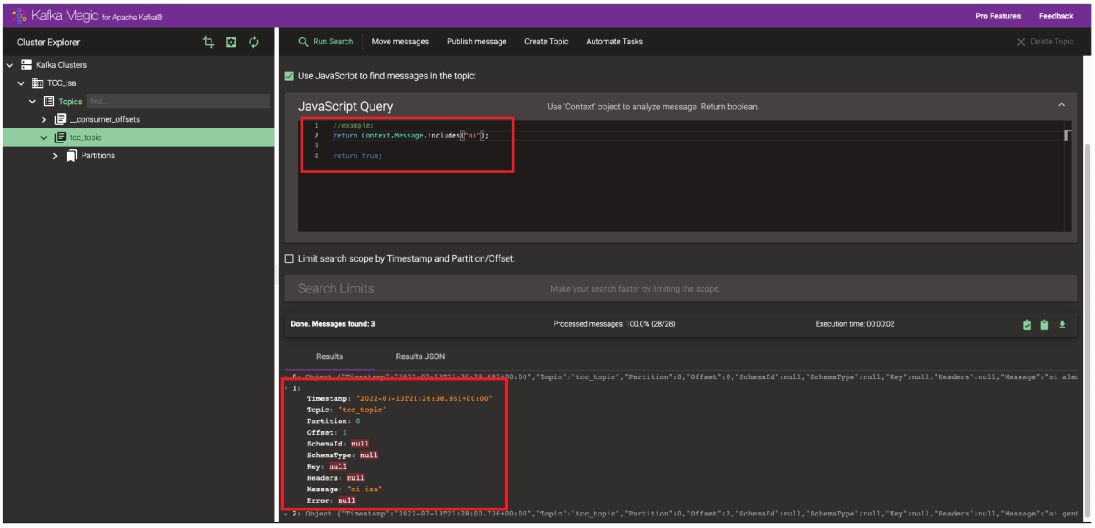

Figura 8 – Realizando consulta com filtros em Java Script no Kafka Magic.

(DOCS, 1998-2022).

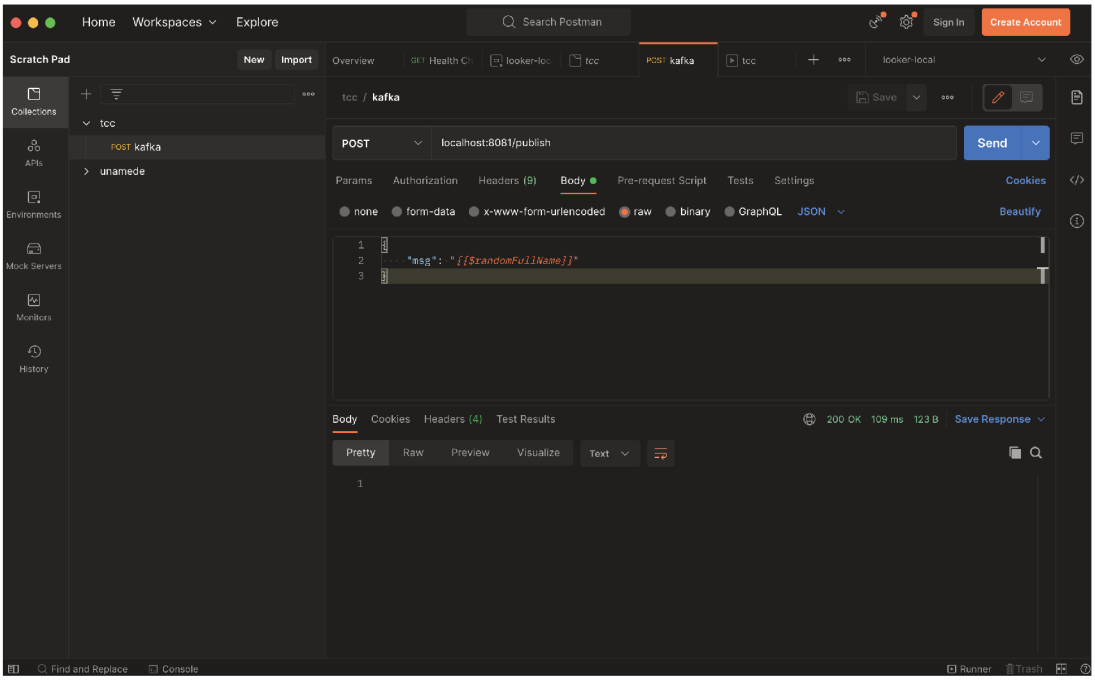

Figura 9 – Requisição POST pelo Postman.

4.2 Testando Kafka no Ruby

O Ruby é uma linguagem de programação orientada a objetos criada por Yukihiro Matsumot e que se tornou pública em 1995. A ideia era criar uma linguagem natural e simples. De acordo com o site oficial do Ruby (RUBY, s.d.), é a junção de Perl, Smalltalk, Eiffel, Ada e Lisp.

Justamente pela flexibilidade e simplicidade do Ruby, e uma grande comunidade de programadores e contribuintes da linguagem, muitos profissionais que atuam com automação de teste optam por essa linguagem de programação para trabalhar em seus testes. Também existem muitas bibliotecas, em Ruby, chamadas de *gem*, que facilitam o trabalho do desenvolvedor. No caso deste trabalho serão utilizadas três principais delas. A primeira é chamada *cucumber*[5]; a seguinte é chamada *ruby-kafka* (ZENDESK, 2015), necessária para montar a automação de testes para tópicos do Kafka; e por último a *report_builder* utilizada para gerar os relatórios dos testes.

O uso dessas *gems* para estudo e execução do projeto deste trabalho (PAULA, 2022) pode ser observado na Figura 10[6].

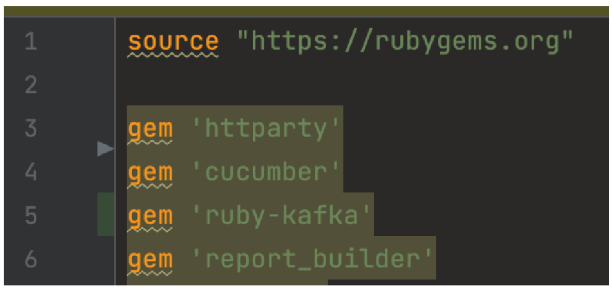

Figura 10 – Lista das principais *gems* usadas no projeto deste trabalho.

4.2.1 *Cucumber*

O *cucumber* é uma ferramenta que permite o automatizador de testes usar o *Behaviour-Driven Development* (BDD) em seus testes, o que traduzido significa Desenvolvimento Orientado ao Comportamento. Ou seja, seu objetivo é aproximar mais ainda a programação ao ser humano, onde as execuções dos testes são descritas em frases, o que porventura acaba servindo como documentação de testes, visto que é algo completamente legível para qualquer pessoa que esteja trabalhando naquele teste.

O BDD, no caso do *Cucumber*, de acordo com seu site oficial, é escrito em *Gherking* (CUCUMBER, 2019), ou seja, um conjunto de regras gramaticais que simplificam a estrutura do texto, para que possa ser entendido pelo *Cucumber*. Ele é estruturado por "Dado", "Quando" e "Então". Neste caso, "Dado" é o cenário inicial do teste; "Quando" é uma ação executada; e o "Então" é a validação do comportamento esperado para aquele cenário.

O exemplo a seguir ilustra melhor a forma como o *Gherking* funciona:

Cenário: Adicionar itens de compra ao carrinho do site

[5] Ferramenta que suporta uma forma de programação orientada a comportamento do usuário.
[6] Captura de tela do projeto de testes para estudo deste trabalho disponível em nuvem.

Dado que acesso um site de compras
Quando realizo a adição de 2 itens de compra ao carrinho
Então valido que a quantidade de itens no carrinho é igual a 2

O *print* de tela ilustrado na Figura 11, além de servir como um segundo exemplo de uso do *Gherking* no BDD, também mostra como esta linguagem foi utilizada na construção do projeto de estudo deste trabalho. Ele utiliza o que é chamado de Esquema de cenário, que permite a execução do mesmo cenário de teste, porém alterando valores durante a sua execução. Apesar de ser apenas um texto descritivo de cenário, o BDD entende que o teste precisa executar o mesmo cenário para cada linha do exemplo do esquema de cenário.

A decisão de escrita de um cenário de teste em formato de esquema de cenário para este trabalho foi para flexibilizar a escolha do tempo que o teste teria para consumir a mensagem desejada. A expectativa na execução do segundo exemplo do esquema de cenário é de falha, pois o tempo estimado para buscar a mensagem no tópico do Kafka é extremamente baixo. Desta forma, o teste não consegue achar a mensagem esperada dentro do tempo estimado e falha. Os casos de sucesso e falha serão ilustrados na sessão 4.2.3 na apresentação do relatório.

```
2
3   @teste
4   Funcionalidade: Consulta de mensagens em tópico do Kafka
5
6     Esquema do Cenario: Consultar a publicação em mensagens no tópico do Kafka local com sucesso
7       Dado que realizo o POST de uma mensagem qualquer
8       Entao valido que houve o consumo dessa mensagem no tópico do kafka local em <seg>
9
10      Exemplos:
11        | seg    |
12        | 20.0   |
13        | 0.0001 |
14
```

Figura 11 – Exemplo de escrita do BDD.

4.2.2 *Gem ruby-kafka*

Esta é uma *gem* do Ruby que permite a criação de um Kafka, número de partições, produtores e consumidores. Também permite a conexão a um Kafka existente para produzir ou consumir mensagens de um tópico para validações da funcionalidade deste Kafka. Este último caso é o cenário utilizado no projeto de estudo deste trabalho, como mostra a Figura 12, conectando a um outro projeto local que está rodando o Kafka e acessando um tópico para ler as mensagens disponíveis nele. Assim, o teste consiste em realizar uma ação para gerar um determinado evento e em seguida validar que a própria automação conseguiu achar e consumir esse evento. A validação se dá na comparação entre a mensagem esperada e consumida no teste.

```ruby
class KafkaTest
  # require "kafka"
  def read_message
    #caminho do broker do kafka
    kafka = Kafka.new(["localhost:9092"])

    # ID do grupo de consumidores
    consumer = kafka.consumer(group_id: "my-consumer")

    #nome do topico a ser consumido
    consumer.subscribe("tcc_topic")

    #variavel responsavel por controle de tempo que o consumidor ficara em funcionamento
    time_start = Time.new

    #loop para leitura das mensagens do kakfa
    consumer.each_message do |message|

      #acessa o valor da mensagem do tópico
      message = message.value

      #printa no console a mensagem que foi consumida
      puts message
```

Figura 12 – Configurações para criação de um consumidor de tópicos do Kafka em um projeto de automação de testes.

4.2.3 *Gem report_builder*

O *report_builder* é uma biblioteca do Ruby que permite a geração de relatórios para projetos que utilizam o *cucumber* para a execução dos testes automatizados. Existem algumas configurações que podem ser feitas para personalização do relatório. No caso deste trabalho, foi feita uma customização no relatório para que ficasse visualmente mais amigável e interativo, fácil de ser entendido e usado por qualquer pessoa.

A Figura 13 demonstra a tela inicial do relatório apresentando a quantidade de testes que foram feitos, juntamente com gráficos que mostram a proporção dos testes que passaram e falharam, o tempo de execução dos testes e a data de execução dos mesmos. A Figura 14 lista os cenários que foram testados. É interessante analisar, nesta figura, o curto tempo que a automação gasta para realizar o teste, se comparado ao tempo que um humano levaria para executar os dois testes. E por fim, ao clicar sobre cada um desses cenários, é possível acessar os BDDs, como ilustram as Figuras 15 e 16.

Figura 13 – Tela principal de visualização do relatório de testes.

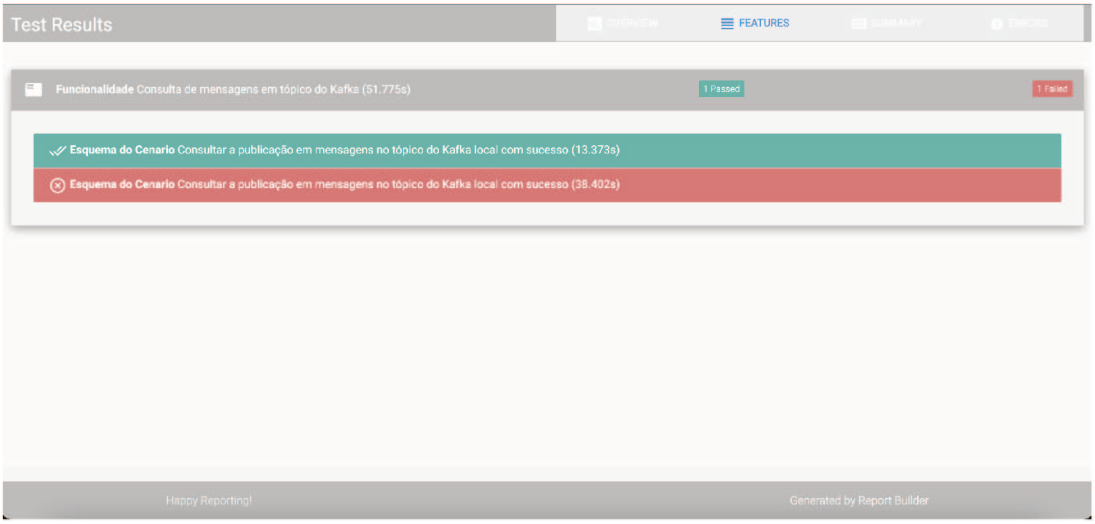

Figura 14 – Visualização dos cenários de teste que foram executados.

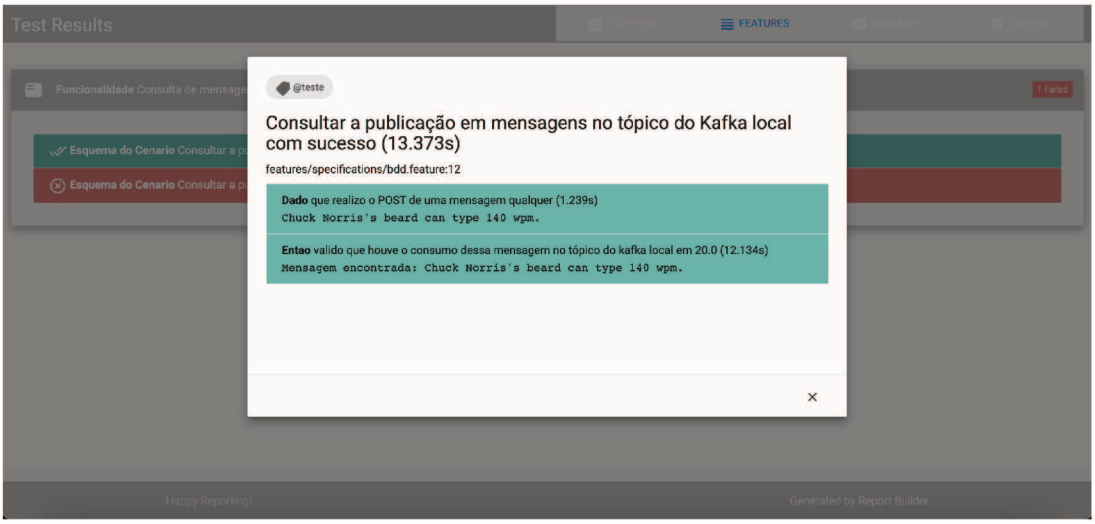

Figura 15 – Visualização do cenário de BDD em que houve sucesso na execução do teste.

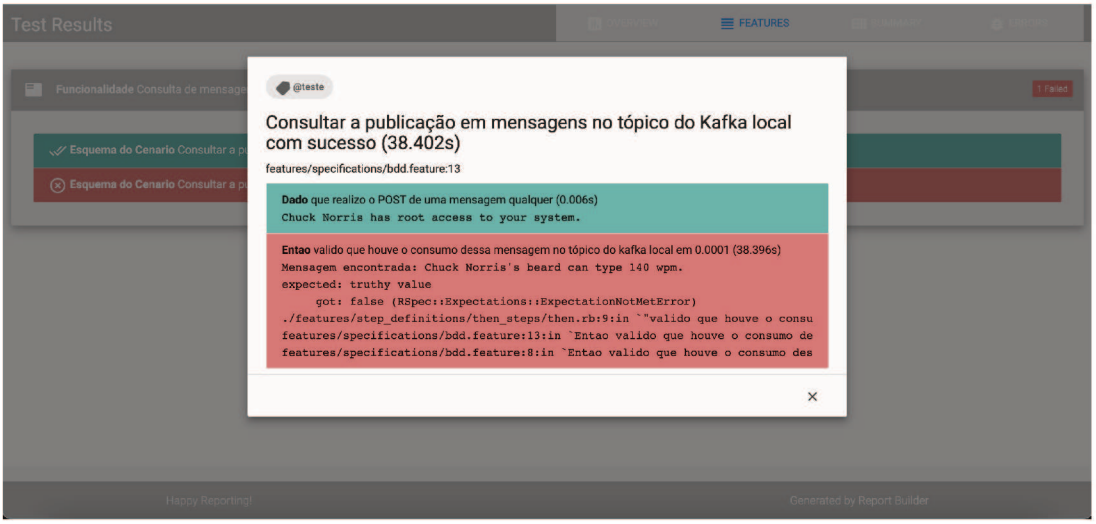

Figura 16 – Visualização do cenário de BDD que houve falha na execução do teste.

5 Conclusão

Para muitos engenheiros de qualidade, os testes de *backend* ainda são complexos e difíceis de serem feitos, ainda mais de forma automatizada. O Kafka, por ser um sistema de mais baixo nível, com pouco conteúdo na língua portuguesa e com poucas ferramentas para auxílio da visualização do seu correto funcionamento, torna o trabalho do engenheiro de qualidade ainda mais complexo e, por isso, muitos deixam a cargo do desenvolvedor realizar esse teste, ou apenas validam o bom funcionamento do sistema como uma validação positiva para qualquer alteração feita no Kafka.

Ao descrever o uso de ferramentas de teste, este trabalho visa servir como um roteiro de teste escrito na língua portuguesa pra contribuir com a comunidade dos engenheiros de qualidade, a fim de promover uma maior proximidade e familiaridade entre os profissionais da área e o Kafka, permitindo, assim, que eles possam realizar um trabalho mais completo quando existir a necessidade de atuar em projetos que façam o uso do Kafka, economizando tempo de teste e garantindo a qualidade desses sistemas.

O Kafka Magic permite o acesso aos dados das mensagens dentro de cada tópico por meio de uma interface gráfica, para a execução de testes manuais. O fato de não poder utilizar a versão de assinante do Kafka Magic foi uma das limitações deste trabalho, pois só foi possível utilizar a versão gratuita, na qual os testes são apenas manuais. Então, uma sugestão para trabalhos posteriores seria explorar as possibilidades da versão paga. E para superar a limitação que o Kafka Magic possui em sua versão gratuita, é possível realizar uma automação de testes utilizando Ruby.

A vantagem da automação é que, quando bem configurada, realizará validações que o profissional da qualidade pode eventualmente se esquecer de executar. Além disso, pode ser integrada a uma esteira de entrega (*Continuous Delivery*), para sempre ser executada antes de uma entrega de projeto e assim otimizar o tempo de execução dos testes e ainda gerar relatórios de maneira automática. Pois o tempo que um humano gasta para executar um teste acaba sendo, quase sempre, maior do que o tempo da máquina. Como pode ser observado na Seção 4.2, que mostra exatamente isso: o tempo de execução de dois cenários de teste foi de aproximadamente 52 segundos, enquanto um humano poderia levar de 3 a 5 minutos para realizar essa validação.

Por fim, fica como sugestão para trabalhos futuros, além de explorar a versão paga do Kafka Magic, já citada, pesquisar como funcionam outras ferramentas de *event stream*, como testá-las e também como realizar a automação de testes em tópicos do Kafka em outra linguagem de programação diferente do Ruby.

Referências

CUCUMBER. *Guia Cucumber*. 2019. Disponível em: <https://cucumber.io/docs/guides/overview/>. Acesso em: 15 de jul. de 2022. Citado na página 26.

DOCS, M. W. *Códigos de status de respostas HTTP*. 1998–2022. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>. Acesso em: 25 de jul. de 2022. Citado na página 25.

GONÇALVES, M. M. *Event-Driven Architecture (EDA) em uma arquitetura de microsserviços*. 2020. Disponível em: <https://imasters.com.br/apis-microsservicos/event-driven-architecture-eda-em-uma-arquitetura-de-microsservicos>. Acesso em: 17 de maio de 2022. Citado 3 vezes nas páginas 10, 11 e 12.

HAT, R. *O que é Apache Kafka?* 2017. Disponível em: <https://www.redhat.com/pt-br/topics/integration/what-is-apache-kafka>. Acesso em: 10 de maio de 2022. Citado na página 14.

HAT, R. *O que é uma API?* 2017. Disponível em: <https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces>. Acesso em: 10 de jun. de 2022. Citado na página 10.

HAT, R. *Arquitetura Orientada a Microsserviços*. 2018. Disponível em: <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>. Acesso em: 12 de set. de 2021. Citado na página 9.

HAT, R. *O que é uma arquitetura orientada por eventos?* 2019. Disponível em: <https://www.redhat.com/pt-br/topics/integration/what-is-event-driven-architecture>. Acesso em: 10 de jun. de 2022. Citado na página 11.

HAT, R. *API REST*. 2020. Disponível em: <https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api>. Acesso em: 10 de jun. de 2022. Citado na página 10.

HAT, R. *Event-driven architecture for a hybrid cloud blueprint*. 2020. Disponível em: <https://www.redhat.com/pt-br/resources/event-driven-architecture-hybrid-cloud-blueprint-detail>. Accessed: 10 de jun. de 2022. Citado na página 11.

KAFKA, A. *POWERED BY*. 2022. Disponível em: <https://kafka.apache.org/powered-by>. Acesso em: 10 de ago. de 2022. Citado na página 14.

LINKEDIN. *Sobre o LinkedIn*. 2022. Disponível em: <https://about.linkedin.com/pt-br>. Acesso em: 26 de maio de 2022. Citado na página 15.

LUCKHAM, D. *The Future of Event Stream Analytics and CEP*. 2020. Disponível em: <https://complexevents.com/2020/06/17/the-future-of-event-stream-analytics-and-cep/>. Acesso em: 30 de jun. de 2022. Citado na página 13.

LUCKHAM, D. *Abstraction Hierarchies in Event Stream Processing*. 2021. Disponível em: <https://complexevents.com/2021/01/02/abstraction-hierarchies-in-event-stream-processing/>. Acesso em: 28 de maio de 2022. Citado na página 13.

MAGIC, K. *Kafka Magic Apache Kafka® Topic Explorer, Manager, and Automation Tool*. 2019. Disponível em: <https://www.kafkamagic.com/>. Acesso em: 17 de jun. de 2022. Citado 2 vezes nas páginas 21 e 22.

NARKHEDE, N.; SHAPIRA, G.; PALINO, T. *Kafka: the definitive guide: real-time data and stream processing at scale*. [S.l.]: "O'Reilly Media, Inc.", 2017. Citado 5 vezes nas páginas 13, 15, 16, 19 e 20.

PAULA, I. R. de. *Projeto de automação de testes em tópicos do Kafka*. 2022. Disponível em: <https://github.com/IsadoraRocha/kafka_case>. Acesso em: 25 de jul. de 2022. Citado na página 26.

POSTMAN. *Postman*. 2022. Disponível em: <https://www.postman.com/>. Acesso em: 29 de jul. de 2022. Citado na página 24.

RABELO, E. *Arquitetura de Software: Explicando Stream Processing, Event Source e Data Streaming*. 2020. Disponível em:<https://oieduardorabelo.medium.com/arquitetura-de-software-explicando-stream-processing-event-source-e-data-streaming-b0c479101e75?a391>. Acesso em: 09 de maio de 2022. Citado na página 14.

RUBY. *Ruby: O melhor amigo do programador*. s.d. Disponível em: <http://www.ruby-lang.org/pt/about/>. Acesso em: 29 de maio de 2022. Citado na página 25.

SOUZA, A. A. *Projeto Kafka-events*. 2022. Disponível em: <https://github.com/alvesdesouzaalex/kafka-events>. Acesso em: 25 de jul. de 2022. Citado na página 24.

TIBCO. *O que é Event Stream Processing?* 2022. Disponível em: <www.tibco.com/pt-br/reference-center/what-is-event-stream-processing>. Acesso em: 09 de maio de 2022. Citado na página 13.

ZENDESK. *zendesk /ruby-kafka*. 2015. Disponível em: <https://github.com/zendesk/ruby-kafka>. Acesso em: maio, jun., jul. de 2022. Citado na página 26.