Event-Driven System

Introdução

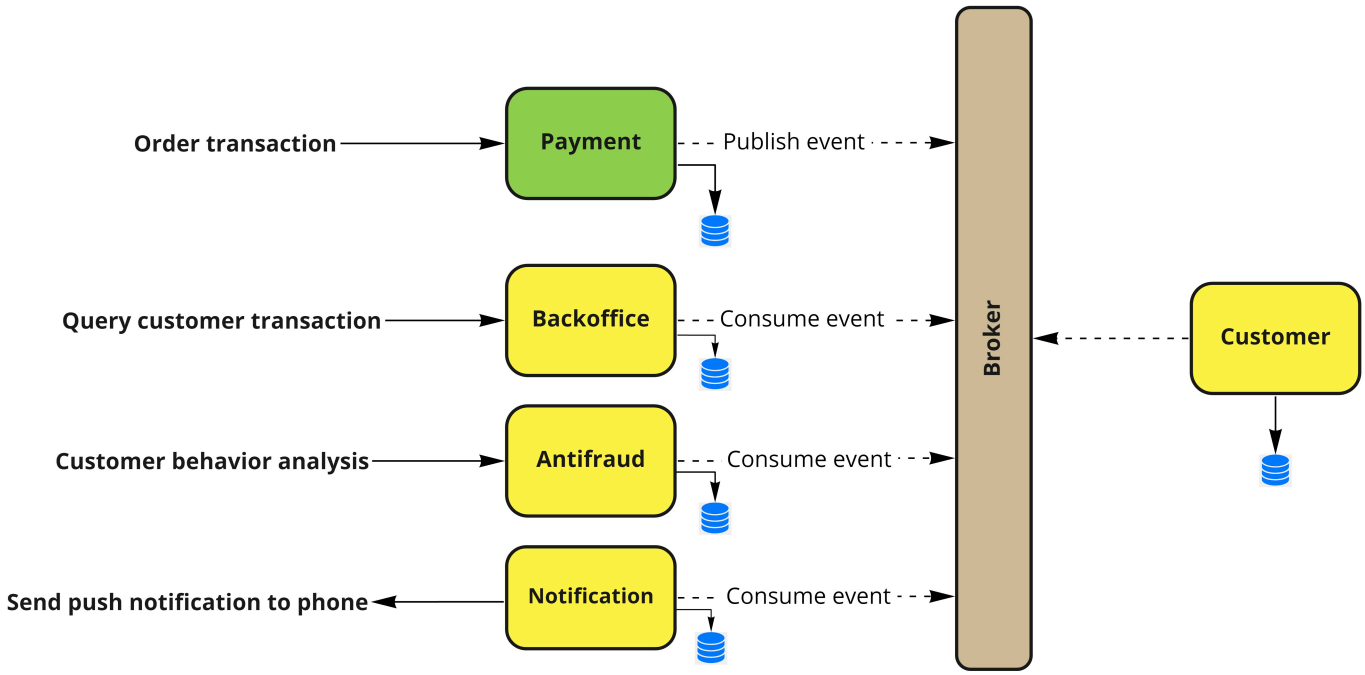
Este artigo vamos discutir sobre Event-Driven e como podemos construir aplicações dirigidas a eventos e como todo um ecosistema pode beneficiar-se desta prática muito difundida entre sistemas críticos e de alta performance.

Definição do site Wikipedia em 26/01/2022 é “Arquitetura orientada a eventos (**EDA** - Event Driven Architecture) é um paradigma de arquitetura de software que promove a produção, detecção, consumo e reação a eventos. Um evento pode ser definido como ***uma mudança significativa no estado***. Uma arquitetura orientada a eventos é extremamente fracamente acoplada e bem distribuída. A grande distribuição desta arquitetura existe porque um evento pode ser quase tudo e existir em quase qualquer lugar. A arquitetura é extremamente fracamente acoplada porque o próprio evento não conhece as consequências de sua causa”. Link original em inglês: https://en.wikipedia.org/wiki/Event-driven_architecture

Conforme descrições acima, sistema que utilizam **EDA** ficam fracamente acoplados, o que possibilita criar novos processos para um evento sem afetar os processos já existentes para este evento. Sendo assim novos processos não prejudicam outros processos.

Aplicações que fazem uso de uma arquitetura orientada a eventos

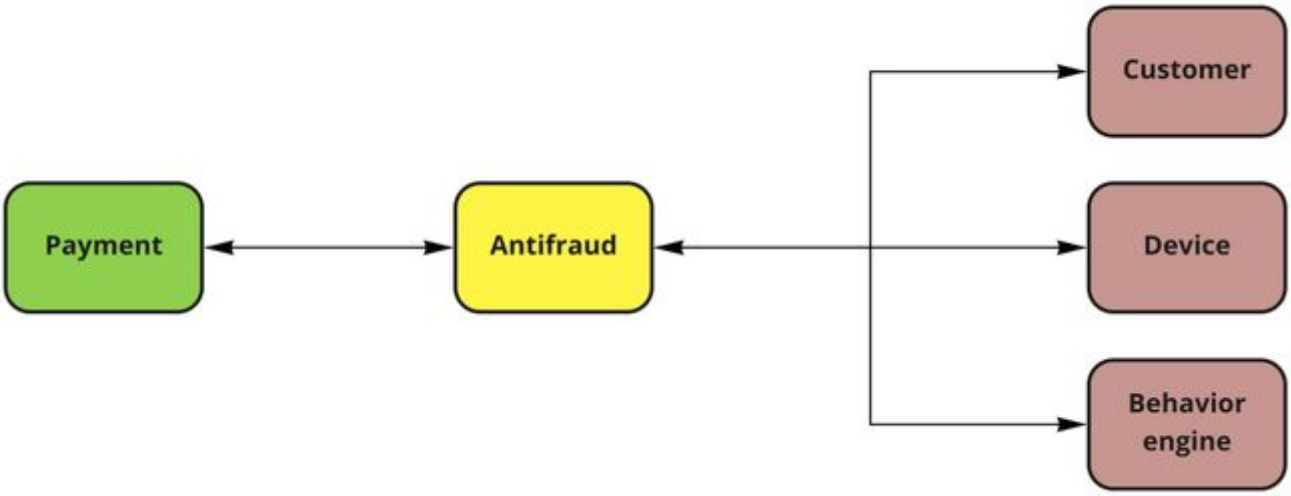
Imagine uma aplicação que realiza transações PIX, nela existe a entidade **payment** que mantém os estados dos pagamentos, tipo estidade **payment X** estado **was accepted**. Notamos que a entidade **payment X** esta em um estado e quando este estado sofre alguma mudança podemos dizer que houve um evento na entidade, tipo a entidade **payment X** mudou de **was accepted** para **was done**. Neste caso aplicações que usam **EDA** publicariam o evento em um broker (kafka, rabbimq, e etc) notificando todo um ecosistema sobre a mudança do estado da entidade desta forma outras aplicações poderam se beneficiar ouvindo este evento publicado para realizar algum processo que teve como origem na publicação do evento.



Neste exemplo já podemos visualizar como ter um arquitetura orientada a eventos teremos aplicações respeitando seu bounded context e desacopladas uma das outras, o que nos da liberdade de evolução e performance.

Modelo tradicional de integração HTTP resquest/response

Comumente aplicações integram-se de forma síncrona utilizando HTTP pois é um formato simples de integração baseado em request/response onde uma aplicação faz uma solicitação a outra aplicação e aguarda o retorno de uma resposta.



No diagrama acima notamos um modelo de integração síncrona usando HTTP. Neste exemplo temos a aplicação que realiza pagamento (**paym ent**) que solicita análise de fraude (aplicação **antifraud**) da transação de pagamento. A aplicação de fraude realizar uma análise de comportamento e precisa obter o cliente (aplicação **customer**), o dispositivo (aplicação **device**) e solicita analisa de comportamento para um motor de análise (aplicação **behavior engine**) para retornar uma a análise de fraude da transação de pagamento.

Neste modelo de integração o acoplamento entre as aplicações esta em um nível muito alto, pois cada aplicação se integra diretamente. Este tipo de integração compromete a disponibilidade das aplicações devido a comunicação direta entre as aplicações, sendo que caso uma das aplicação passe por indisponibilidade ela compromete outras aplicações que dependam dela para realizar seus processamentos. Exemplo de cálculo de disponibilidade para as aplicações**.**

Exemplo de dispobilidade ignorando as integrações entre aplicações.

customer: 98,99%

device: 99,3%

behavior engine: 97,99%

antifraud: 97,55%

payment: 98,77%

Acima esta a disponibilidade de cada aplicação ignorando as integrações entre elas. Agora veja como ficará a disponibilidade de cada aplicação com suas integrações.

customer: 98,99%

device: 99,3%

behavior engine: 97,99%

antifraud: (97,55% x 98,99% x 99,3% x 97,99%) = 93,96%

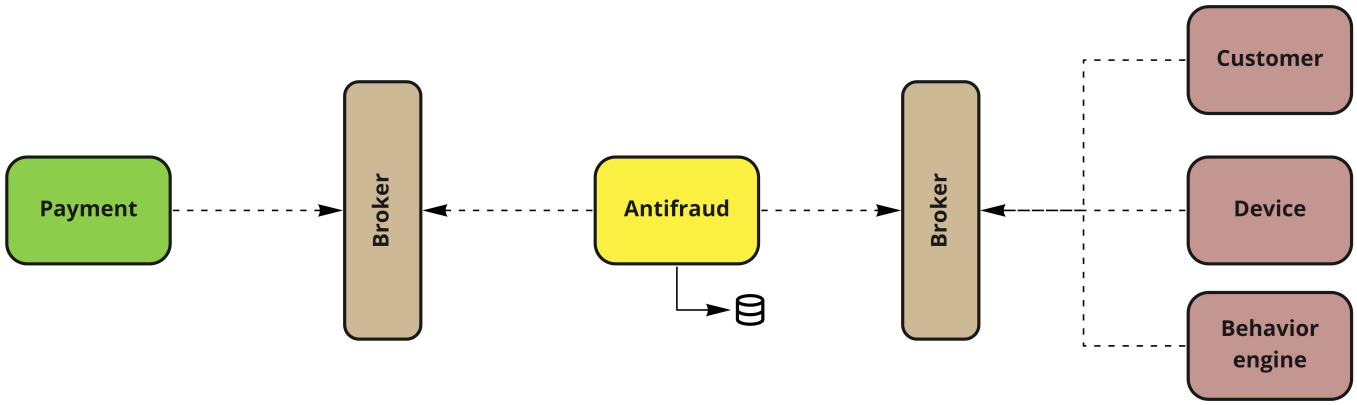
payment: (98,77% x 93,96%) = 92,80%

Notamos que quanto maior for as integrações síncrona entre as aplicações maior será sua indisponibilidade.

Outro problema muito importante é o acoplamento entre as aplicações que dificulta a evolução das features e processos pois os mesmos estão acoplados uns aos outros. Exemplo caso uma das seguintes aplicações **customer**, **device, behavior engine** fiquem indiponível não será possível processar pagamentos, outro exemplo caso a aplicação **customer** precise mudar seu DNS vou precisar atualizar a aplicação **antifraud** também, o que pode levar a deploys travados e processos de deploys dependentes uns dos outros. Um problema muito recorrente neste tipo de integração seria caso um deploy no microservice **customer** gere indiponibilidade o microservice **antifraud** também passará por indisponibilidade o que é um problema muito grave pois são contexto de negócio (**bounded context**) diferente e não deveriam interferir um com o outro.

Modelo de integração assíncrono

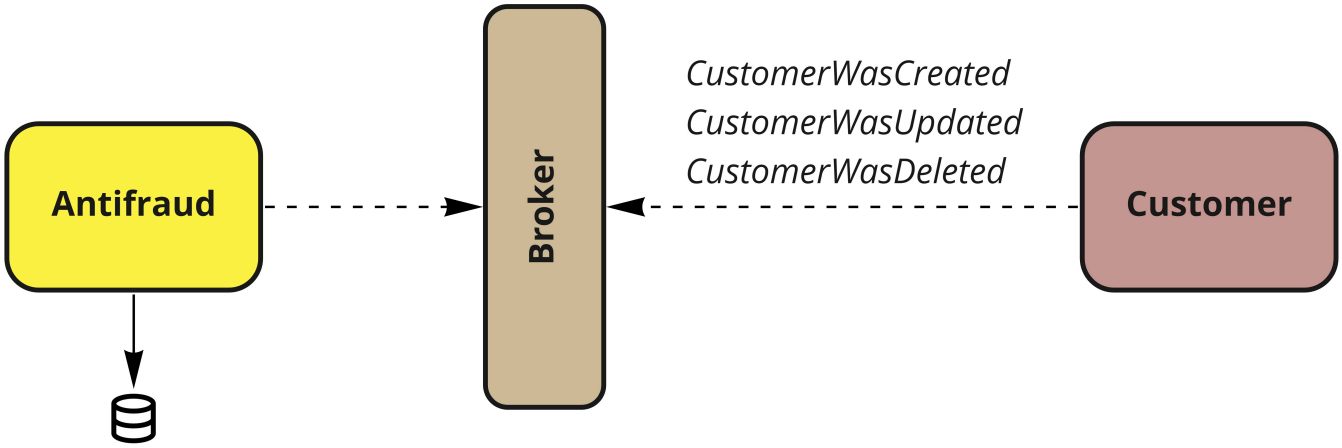
Para resolvermos os problemas levantados no tópico anterior, vamos construir um modelo de comunicação assíncrona onde as aplicações fazem uso de uma arquitetura de eventos diminuindo assim o acoplamento entre as aplicações. Veja diagrama abaixo:



No diagrama acima as aplicações não se comunição diretamente o que diminui o acoplamento entre elas, nesta forma de comunição uma aplicação não vai gerar um impacto direto na disponibilidade da outra aplicação quando esta passar por indisponibilidade. Desta forma as aplicações conseguem manter sua disponibilidade independente de outras aplicações e o acoplamento direto entre aplicações será muito menor. Neste exemplo a aplicação de pagamento (**payment**) não sabe qual será a aplicação que fará a análise de fraude pois para seu fluxo isto é indiferente o que a aplicação de pagamento precisa é que a análise seja realizada, quem fará não importa.

Padrão Event Notification

O pattern event notification consiste na idéia de eventos gerados por um microserviço são ouvidos por outras aplicações que fazem o armazenamento destes dados em uma base local, diminuindo ainda mais a dependência direta entre as aplicações. Exemplo abaixo:



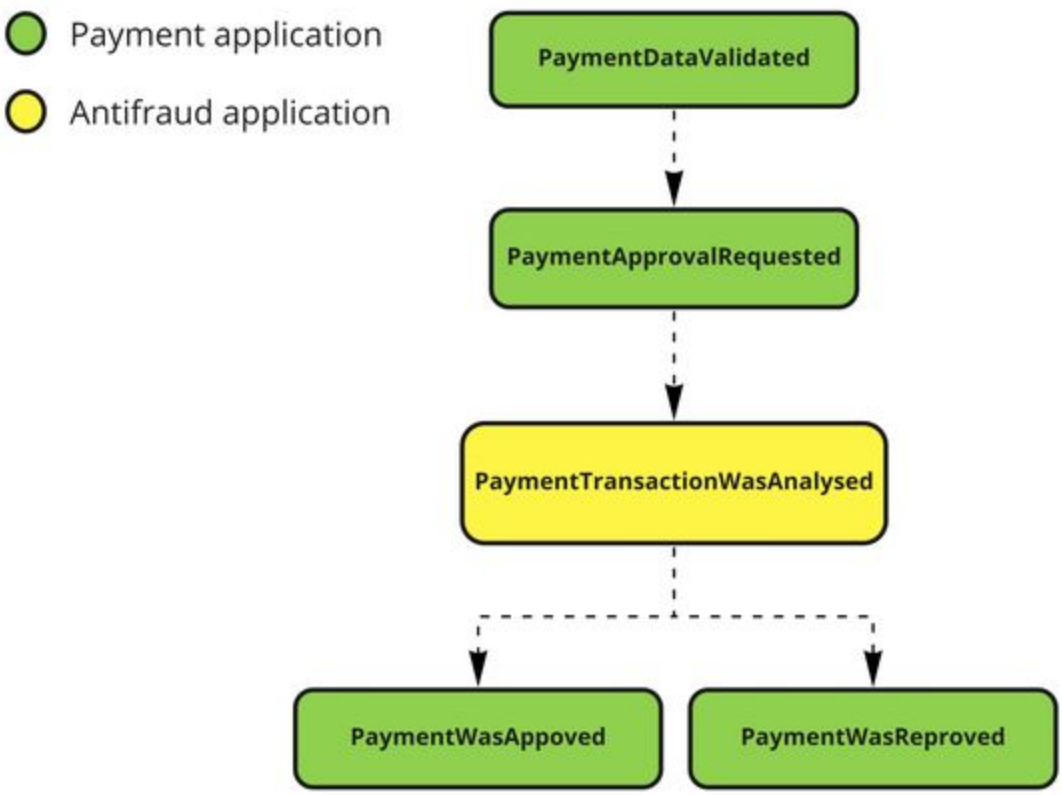
Neste modelo a aplicação **antifraud** processa os eventos ***CustomerWasCreated***, ***CustomerWasUpdated*** e ***CustomerWasDeleted*** gerados pela aplicação **customer**, e armazenar os dados em uma base de dados local, desta forma a aplicação **antifraud** diminui ainda mais seu acoplamento com a aplicação **customer** conseguindo realizar processamento de análise de fraude mesmo quando a aplicação **customer** estiver indisponível.

Referências: https://martinfowler.com/articles/201701-event-driven.html

Livro: **First Edition "Designing Event-Driven Systems" book confluent (O'Reilly - By Ben Stopford) Pg: 34**

Evento de colaboração

Eventos de colaboração basicamente iremos ter mais de uma aplicação envolvida para que um determinado evento ocorra, exemplo abaixo:



No diagrama acima podemos ver que os evento **PaymentWasAppoved** ou **PaymentWasReproved** dependem da um evento publicado pela aplicação de **antifraud** de **PaymentTransactionWasAnalysed,** assim caso este evento não ocorrá não será possível realizar a publicação dos eventos finais **PaymentWasAppoved** ou **PaymentWasReproved**, desta forma temos uma colaboração de eventos que é controlada por uma **S AGA (Saga Long Lived Transactions).**

SAGAs é uma padrão que ajuda no controle da maquina de estado com relação a processos de longo duração, sendo utilizados dois modelos o **coreografado** e **orquestrado** conforma documentação: https://microservices.io/patterns/data/saga.html para maior explicação. Explicação básica:

> **SAGA Orquestrado**: Existe um orquestrador que controla cada estado de maquina da saga, este orquestrador controla cada passo e reação a cada evento gerado no processo.
>
> **SAGA Coreografado:** Aqui os processos ocorrem sem que exista um ponto central controlando a SAGA, os processos vão ocorrendo conforme são gerados os eventos no processo.

"Os sistemas coreografados têm a vantagem de serem plugáveis. Se o pagamento decide criar três novos tipos de eventos para a parte de pagamento do
fluxo de trabalho, desde que o evento Pagamento processado permaneça, ele pode fazê-lo sem
afetando qualquer outro serviço" Livro: **First Edition "Designing Event-Driven Systems" book confluent (O'Reilly - By Ben Stopford)** Pg: 40

Conforme descrição acima, saga que utilizam coreografia de eventos, são mais flexíveis, sendo a melhor opção para implementação de **SAGAs.**