Event-Sourcing | CQRS

Event-Sourcing

Quando trabalhamos com **EDA (Event-Driven Architecture)** um assunto que comumente aparece é sobre **event sourcing**, pois é uma das práticas difundidas sobre como trabalhar com eventos e que possibilita ter uma visão de tudo o que foi aconteceu com determinada entidade de uma aplicação durante todo seu ciclo de vida.

“Em um nível alto, Event Sourcing é apenas a observação de que eventos (ou seja, estado mudanças) são um elemento central de qualquer sistema. Então, se eles são armazenados, imutavelmente, na ordem em que foram criados, o registros de eventos resultante fornece um passo a passo de exatamente o que o sistema fez.” Livro: ***First Edition “Designing Event-Driven Systems” book confluent (O’Reilly - By Ben Stopford) Pg: 55***

Como o próprio nome sugere **eventos** são a fonte para compormos uma entidade. Desta forma conseguimos não somente ver o estado final da entidade mas podemos analisar todas mudanças que foram geradas ao longo do ciclo de vida da entidade. Abaixo segui um exemplo de um entidade de agregação e os eventos gerados conforme seu estado vai mudando ao longo do tempo.

entidade: **payment** (aggregate root)

```
{
   "aggregateId":"string",
   "version":"number",
   "customerNumber":"string",
   "amount":"number",
   "status":"string",
   "receive":{
      "bank":"string",
      "agency":"string",
      "account":"string"
   },
   "transactionCode":"string"
}
```

evento: **paymentWasReceived**

```
{
   "aggregateId":"12345",
   "version":1,
   "customerNumber":"5",
   "amount":5,
   "receive":{
      "bank":"0005",
      "agency":"0001",
      "account":"0001"
   }
}
```

aplicando o evento **paymentWasReceived** a entidade ficará desta forma:

```
{
   "aggregateId":"12345",
   "version":1,
   "customerNumber":"5",
   "amount":5,
   "status":"wasReceived",
   "receive":{
      "bank":"0005",
      "agency":"0001",
      "account":"0001"
   },
   "transactionCode":null
}
```

evento: **paymentTransactionCodeWasGenerated**

```
{
        aggregateId: "12345",
        version: 2,
        transactionCode: "asdf12"
}
```

aplicando o evento **paymentTransactionCodeWasGenerated** a entidade ficará desta forma:

```
{
        aggregateId: "12345",
        version: 2,
        customerNumber: "5",
        amount: 5,
        status: "transactionCodeWasGenerated",
        receive: {
                bank: "0005",
                agency: "0001",
                account: "0001",
        },
        transactionCode: "asdf12",
}
```

evento: **paymentWasAppoved**

```
{
        aggregateId: "12345",
        version: 3
}
```

aplicando o evento **paymentWasAppoved** a entidade ficará desta forma:

```
{
        aggregateId: "12345",
        version: 3,
        customerNumber: "5",
        amount: 5,
        status: "wasAppoved",
        receive: {
                bank: "0005",
                agency: "0001",
                account: "0001",
        },
        transactionCode: "asdf12",
}
```

evento: **paymentWasDone**

```
{
        aggregateId: "12345",
        version: 4
}
```

aplicando o evento **paymentWasDone** a entidade ficará desta forma:

```
{
        aggregateId: "12345",
        version: 4,
        customerNumber: "5",
        amount: 5,
        status: "wasDone",
        receive: {
                bank: "0005",
                agency: "0001",
                account: "0001",
        },
        transactionCode: "asdf12",
}
```

Imutabilidade de eventos

Eventos sempre são acontecimento no passado desta forma um evento nunca pode ser modificado, caso sejá necessário desfazer um evento deverá ser lançado um evento de compensação ao evento que deseja ser anulado. Exemplo a transação de pagamento no qual solicita a conta uma reserva do valor para a transação, sendo lançado um evento **amountWasReserved** pela entidade **account** com o seguinte formato:

Evento: **amountWasReserved**

```
{
        aggregateId: "987654",
        version: 2,
        amount: 5
}
```

deverá ter um evento de compensação desta forma:

Evento: **amountReservedWasCancelled**

```
{
        aggregateId: "987654",
        version: 3
}
```

Quando usamos event-sourcing é muito importante nunca mudar um evento e sim lançar eventos de compensação quando for necessário.

Armazenando eventos

Quando estamos utilizando event-sourcing sempre vamos trabalhar com o princípio da imutabilidade de cada evento, desta forma não devemos editar um evento. Abaixo segui um exemplo de como devemos armazenar estes eventos isoladamente em um banco de dados o formato poderá ser utilizado tanto em banco de dados não relacional como relacional. Segui exemplo:

```
{
        id: "1",
        aggregateId: "12345",
        version: 1,
        name: "paymentWasReceived",
        data: "{ aggregateId: "12345", version: 1, customerNumber: "5",
amount: 5, receive: { bank: "0005", agency: "0001", account: "0001" } }"
}

{
        id: "55",
        aggregateId: "12345",
        version: 2,
        name: "paymentTransactionCodeWasGenerated",
        data: "{ aggregateId: "12345", version: 2, transactionCode:
"asdf12" }"
}

{
        id: "64",
        aggregateId: "12345",
        version: 3,
        name: "paymentWasAppoved",
        data: "{ aggregateId: "12345", version: 3 }"
}

{
        id: "97",
        aggregateId: "12345",
        version: 4,
        name: "paymentWasDone",
        data: "{ aggregateId: "12345", version: 4 }"
}
```

Quando olhamos o modelo de dados notamos que o aggregateId é o id referente ao aggregate root da entidade sequenciado pelo versionamento que é sequêncial, este campo de versionamento nunca poderá se repetir para o mesmo aggregateId.

Quando olhamos cada evento isoladamente ele por si só não diz muita coisa, sendo necessário fazer a redução de todos os eventos para termos o estado final da entidade, sendo assim isto pode ser um problema de performance quanto precisamos realizar esta operação repetidades vezes compromentendo o tempo de resposta o **SLA** (**Service Level Agreement**) das apis. Para conseguirmos contornar este problema podemos utilizar uma técnica chamada de **CQRS** (**Command Query Responsibility Segregation**) que apresenta uma estratégia de **s napshot** dos registros o que elimina a necessidade de ficar reduzindo os eventos da entidade agregadora.

Livro: ***First Edition "Designing Event-Driven Systems" book confluent (O'Reilly - By Ben Stopford) Chapter: 7***

Artigo: https://martinfowler.com/eaaDev/EventSourcing.html

CQRS - **Command Query Responsibility Segregation**

Como o próprio nome já menciona **CQRS** é a separação entre a intensão de cirar ou modificar da intensão de consulta, sendo que **command** são intensão de criar ou modificar e que geram eventos e **query** são consultas que retornam algo. Iremos abordar neste artigo uma visão superficial do que é **CQRS**, sendo possível aprofundar mais no assunto lendo livros como ***Designing Event-Driven Systems (By Ben Stopford), Implementing Domain-Driven Design (By Vaughn Vernon) entre outros.*** Seguindo com o modelo de dados dos eventos acima,

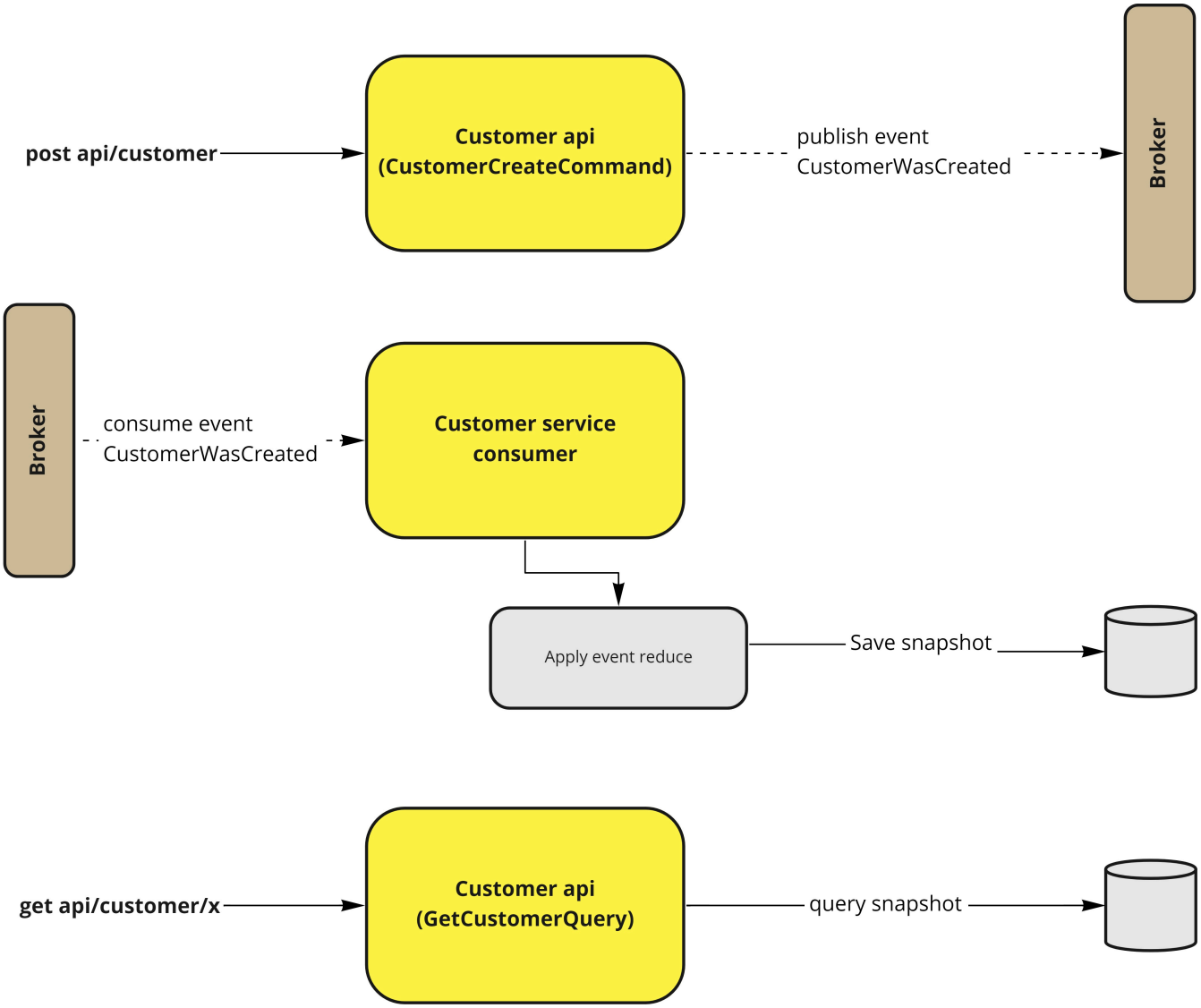
conseguimos resolver o problema de não ficar reduzindo todos os eventos de uma entidade salvando um snapshot da entidade sendo assim otimizando performance nas consulta destes dados**,** exemplo de modelo de snapshot da entitade **payment**:

```
{
        aggregateId: "12345",
        version: 4,
        customerNumber: "5",
        amount: 5,
        status: "wasDone",
        receive: {
                bank: "0005",
                agency: "0001",
                account: "0001",
        },
        transactionCode: "asdf12",
}
```

Este modelo acima é comumente chamado de snapshot da entidade pois é um retrato do estado final da entidade **payment**, e com este snapshot conseguimos eliminar processamento desnecessário no momento da consulta dos dados. O que fizemos foi isolar a forma de consultar dados da forma de criar ou atualizar o estado da entidade.

Atualizar snapshot

Um passo importante neste modelo é decidir como o snapshot será atualizado. Existem muitos formatos para se realizar esta operação, vou apresenta um formato que é bem aceito quanto utilizamos **event-driven** mas não é o único formato.



Neste modelo todas as consultas são realizadas em um modelo de dados do snapshot e caso haja a necessidade de implementar outras técnicas para se obter performance nas consultas isto não afetará o modelo de dados de eventos que ocorre em decorrência de **commands**, isto ocorre pois cada estrutura está isolada.