Desafio Técnico para fazer em casa

Introdução

Este teste visa dar uma olhada nas diferentes estratégias que os candidatos podem ter ao implementar este recurso crucial.

Resolução e entrega [leia com atenção!]

O que será avaliado?

Além de avaliar a correção da sua solução, temos interesse em ver como você modela o domínio, organiza seu código e implementa seus testes.

Agora, vamos guiá-lo através de alguns conceitos básicos.

Transaction

Uma versão simplificada de um transaction payload de cartão de crédito é o seguinte:

```
{
        "account": "123",
        "totalAmount": 100.00,
        "mcc": "5811",
        "merchant": "PADARIA DO ZE            SAO PAULO BR"
}
```

Atributos

- **id** - Um identificador único para esta transação.
- **accountId** - Um identificador para a conta.
- **amount** - O valor a ser debitado de um saldo.
- **merchant** - O nome do estabelecimento.
- **mcc** - Um código numérico de 4 dígitos que classifica os estabelecimentos comerciais de acordo com o tipo de produto vendido ou serviço prestado.

  O MCC contém a classificação do estabelecimento. Baseado no seu valor, deve-se decidir qual o saldo será utilizado (na totalidade do valor da transação). Por simplicidade, vamos usar a seguinte regra:
  - Se o mcc for "5411" ou "5412", deve-se utilizar o saldo de FOOD.
  - Se o mcc for "5811" ou "5812", deve-se utilizar o saldo de MEAL.
  - Para quaisquer outros valores do mcc, deve-se utilizar o saldo de CASH.

Desafios (o que você deve fazer)

Cada um dos desafios a seguir são etapas na criação de um **autorizador completo**. Seu autorizador deve ser um servidor HTTP que processe a transaction payload JSON usando as regras a seguir.
As possíveis respostas são:

- { "code": "00" } se a transação é **aprovada**
- { "code": "51" } se a transação é **rejeitada**, porque não tem saldo suficiente
- { "code": "07" } se acontecer qualquer outro problema que impeça a transação de ser processada

O HTTP Status Code é sempre 200

L1. Autorizador simples

O **autorizador simples** deve funcionar da seguinte forma:

- Recebe a transação
- Usa **apenas** a MCC para mapear a transação para uma categoria de benefícios
- Aprova ou rejeita a transação
- Caso a transação seja aprovada, o saldo da categoria mapeada deverá ser diminuído em **totalAmount**.

L2. Autorizador com fallback

Para despesas não relacionadas a benefícios, criamos outra categoria, chamada **CASH**. O autorizador com fallback deve funcionar como o autorizador simples, com a seguinte diferença:

- Se a MCC não puder ser mapeado para uma categoria de benefícios ou se o saldo da categoria fornecida não for suficiente para pagar a transação inteira, verifica o saldo de **CASH** e, se for suficiente, debita esse saldo.

L3.Dependente do comerciante

As vezes, os MCCs estão incorretos e uma transação deve ser processada levando em consideração também os dados do comerciante. Crie um mecanismo para substituir MCCs com base no nome do comerciante. O nome do comerciante tem maior precedência sobre as MCCs.
Exemplos:

- UBER TRIP SAO PAULO BR
- UBER EATS SAO PAULO BR
- PAG*JoseDaSilva RIO DE JANEI BR

- PICPAY*BILHETEUNICO GOIANIA BR

L4. Questão aberta

A seguir está uma questão aberta sobre um recurso importante de um autorizador completo (que você não precisa implementar, apenas discuta da maneira que achar adequada, como texto, diagramas, etc.).

- Transações simultâneas: dado que o mesmo cartão de crédito pode ser utilizado em diferentes serviços online, existe uma pequena mas existente probabilidade de ocorrerem duas transações ao mesmo tempo. O que você faria para garantir que apenas uma transação por conta fosse processada em um determinado momento? Esteja ciente do fato de que todas as solicitações de transação são síncronas e devem ser processadas rapidamente (menos de 100 ms), ou a transação atingirá o timeout.

Para este teste, tente ao máximo implementar um sistema de autorização de transações considerando todos os desafios apresentados (L1 a L4) e conceitos básicos.

L1 - L3:

L4:

Para garantir que apenas uma transação é processada em dado momento, é necessário a adição *locks*, isto previne que duas ou mais transações simultâneas consigam escrever ao mesmo tempo no mesmo registro, causando inconsistências.

Por exemplo, sem lock, se duas transações, T1 e T2, debitando ao mesmo tempo R$ 900 e R$ 800 reais, respectivamente, em uma conta com R$ 1000. A aplicação pode processar essas duas transações concurrentemente, e ao buscar o saldo no banco o valor retornado em ambas as operações é R$ 1000. Como ambas transações são válidas (não superam o valor saldo), a aplicação debita do saldo, resultando no valor final de R$ 100 e R$ 200 para as transações T1 e T2, respectivamente. Após o débito é submetido o salvamento no banco de dados. Dependente na ordem que o banco processou as requisições, o saldo salvo será R$ 200 ou R$ 100. Ao final do processo o usuário conseguiu efetuar um debito de R$ 1700 e ainda sobrou R$ 100 ou R$ 200 reais na conta.

Com *locks* a aplicação pode se salvaguardar do problema descrito acima, pois mesmo na presença transações simultâneas, a aplicação pode garantir que somente uma transação poderá ler e/ou escrever no banco de dados em um dado momento. Isso acontece, pois cada transação usando *locks* pode obter exclusividade nos acessos de leitura e escrita, e mesmo na presença de transações simultâneas, somente uma transação pode ter o lock por vez. Dessa forma é garantido que apenas uma transação possa ser executada por vez.

Existem dois tipos de *locks*, os *locks* otimistas (exclusividade na escrita) e os pessimistas (exclusividade na leitura e escrita). Contudo para manter a consistência é necessário escolher o lock pessimista, pois o problema citado acima também pode ocorrer com o lock otimista. Por exemplo, se duas transações simultâneas obtêm o mesmo estado do saldo na leitura, e no momento da escrita uma das transações obteve o *lock*, efetuou o salvamento e liberou o lock antes que a outra transação tente escrever obtendo o *lock*, podemos cair na mesma situação citada anteriormente.

Para garantir o processamento mais rápido das transações, e garantir que elas não superem o limite de 100ms, podem ser tomadas algumas medidas:

- **Arquitetura Master-Slave com consistência sequencial**

Uma forma de reduzir o tempo de resposta das transações é a adoção arquitetura master-slave com a consistência sequencial, onde haverão instâncias master (uma ou mais, de preferência um número pequeno para não impactar a performance) que atenderão apenas operações que requeiram consistência forte, enquanto slaves serão responsáveis pelas as demais operações. Com isso as operações que dependem de consistência forte se beneficiarão pois não precisarão disputar recursos com outras operações, o que traz ganhos na performance, por evitar situações de *throttiling* (sufocamento) e enfileiramento de operações. No caso das operações que não necessitem de consistência forte, elas se beneficiarão pois não precisarão mais ficar esperando um lock ser liberado, para que possam efetuar suas operações, trazendo também ganhos no tempo de resposta.

- **Particionamento da base de dados**

Com o aumento no número de requisições simultâneas (mesmo na ausência de transações conflitantes) e/ou no número de informações armazenadas, a performance no banco de dados tende a degradar. Nesse ponto, o particionamento do banco se torna indispensável para manter a performance geral das operações. Com o particionamento, podemos dividir todo o conjunto de dados em subconjuntos menores (partições) que são distribuídos entre instâncias do banco. Dessa forma é possível atender mais clientes ao mesmo tempo, com o aumento de hardware, e ter operações de leitura e escrita mais rápidas pois apenas lidarão com um subconjunto menor dos dados tornando mais efetivas as leituras e escritas das informações.

- **Escolher hardware adequado para o workload**

De acordo com o workload pode ser necessário escolher hardware específico para que o atenda da melhor forma. Por exemplo se seu workload realiza um grande número de operações de leitura e escrita, pode ser necessário investir em dispositivos de armazenamento com um bom número de IOPS (operações de escrita e leitura por segundo). Além disso, como é necessário transferir os dados tanto para servir as aplicações quanto para o processo de replicação, é necessário escolher máquinas que tenham uma taxa de transferência adequada.

- **Manter o banco próximo as aplicações**

Mesmo com os hardwares modernos se duas instâncias, banco ou aplicação, estão geograficamente distantes uma da outra, o tempo de comunicação entre elas (latência) demorará mais. Logo, mesmo parecendo bem óbvio mencionar isso, é importante considerar também esse aspecto na modelagem da arquitetura.