# FLOW

Experimentação técnica

**Olá,**

**Segue resposta sobre o teste.**

**Fiz o máximo dentro do tempo estabelecido, mas escopo realmente carece de cuidado devido à sua complexidade.**

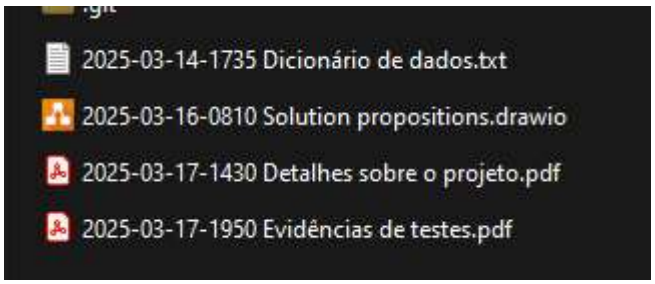
**Repositórios:**

https://github.com/llisboa/FlowApi -> código c# da API

https://github.com/llisboa/FlowDb -> scripts de criação da base de dados

https://github.com/llisboa/FlowDoc -> documentos

Documentos(FlowDoc):




**Programas Utilizados:**

* AWS SAM, validação dos CF para criação de objetos na AWS.

* VisualStudio 2022, programação C#

* VSCode, edição de scripts CF entre outros.

* DBeaver, acesso ao banco de dados

* DrawIO, para desenhos de arquitetura.

* Postman, testes e geração do token.

# Breve Resumo

O exercício não indicou questões de custo. Então estão sendo submetidos três desenhos, cada um variando o custo conforme a tecnologia empregada.

Para efeito de referência, serão mencionadas soluções disponíveis na núvem AWS, que possuem similares em outros provedores de nuvem. A escolha da AWS é mais pela familiaridade.

Um controle de caixa de auto desempenho precisa tratar de modo assíncrono seus lançamentos bem como permitir que o atendimento às requisições escale conforme a necessidade, seja o tratador das requisições ou até mesmo o banco de dados.

O tempo não foi suficiente para desenvolvimento de todos os módulo(sprint de um homem só), até pelo grau de complexidade que existe em um sistema de fluxo estilo bancário, mas aproveitei para tentar demonstrar não só conhecimento mas também experiência adquirida com instituições bancárias.

Então, a parte usada para exemplificar programação foi apenas a API, que trata dos impactos e permite consultas. No desenho constam mais elementos (API Gateway, SNS, SQS e Lambda) servindo como uma proposta futura de expansão.

O código da API por sí só já exercita muios conceitos que também seriam empregados na construção dos demais módulos.

# Segurança

Senha do banco foi gravada em SecretsManager na AWS e está sendo lida através de

AWSSDK. Tanto a senha quanto o servidor e usuário, ambos estão sendo carregados via SSO e atualizados em tempo de execução para evitar riscos com segurança.

Token BEARER também está sendo exigido para evitar acesso sem sessão válida. Um aplicativo foi criado no AZURE apenas para emitir tokens válidos. Detalho mais a seguir.

# Sequência de Tópicos Apresentados

Na tentativa de organizar os esclarecimentos seguei conforme as etapas abaixo:

* Arquitetura

* Banco de Dados

* Experiência Anterior(com instutição financeira)

* Estrutura de Tabelas

* Processo de Desenvolvimento e Implantação

* Motor Transacional

* Monitoramento Integrado

* Etapas Críticas

* Log de Erros (DATADOG)

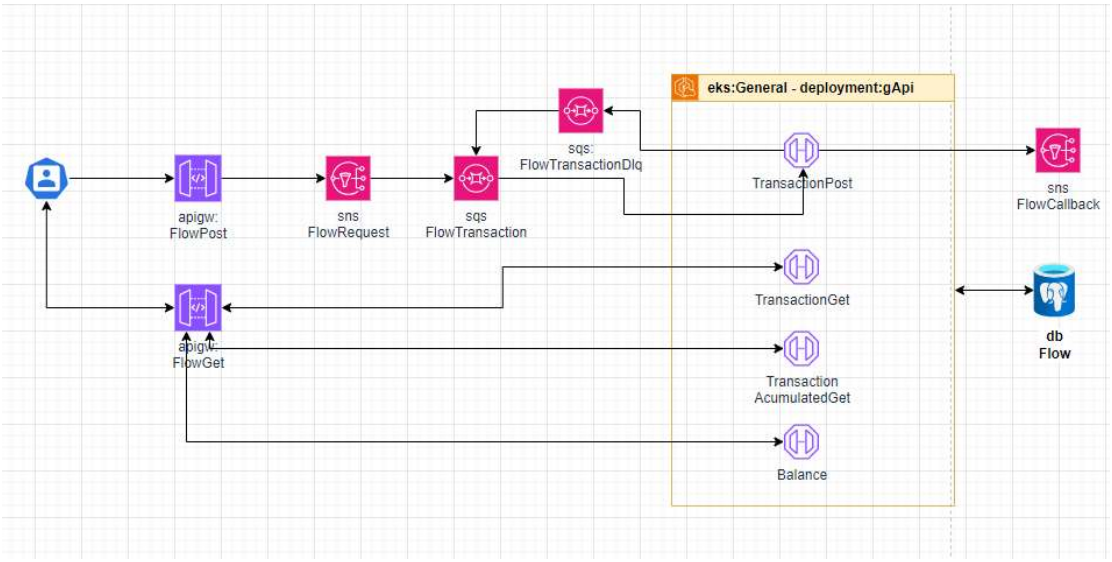
* Sobre Testes

* Programas Utilizados

# Arquitetura

Como não foi tratado neste momento o assunto de custo de projeto, apresento três esboços hipotéticos, cada um exigindo um nível de investimento.

**Primeiro Esboço**(mais próximo da implementação)



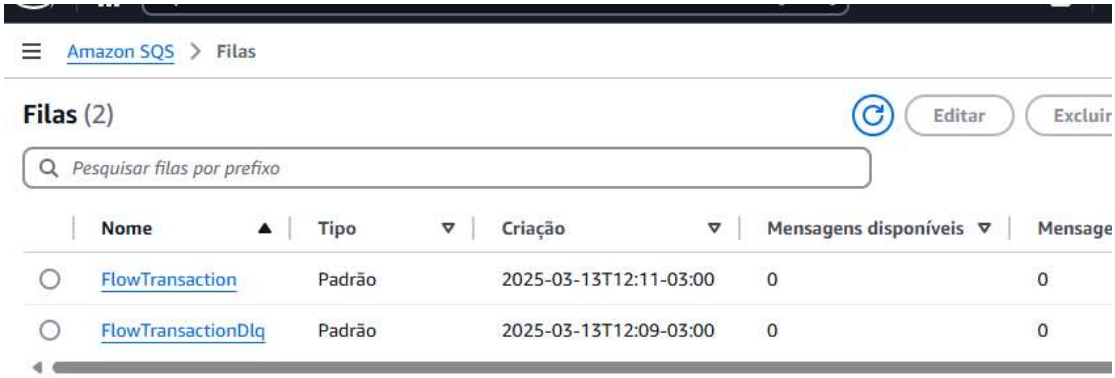

Algumas explicações deste contexto também já servem para esclarecer os próximos também. Da esquerda para a direita:

Na linha de cima, usuário utiliza API Gateway FLOWPOST para introduzir IMPACTO, que passa por um SNS FLOWREQUEST sendo possível, a qualquer momento acoplar qualquer outro serviço para poderem acompanhar as requisições.


APIGW filtra mensagens incorretas oferecendo um nível de validação antes que a mensagem seja introduzida no sistema. Também oferece a oportunidade de registro da requisição para rastreamento.

Acoplada agora no SNS está o SQS FLOWTRANSACTION que recebe todos os impactos até que a API trate e persista no banco de dados.

Caso ocorra algum problema durante o tratamento da mensagem e retetativas posteriores essa mensagem vai para a DLQ FLOWTRANSACTIONDLQ.



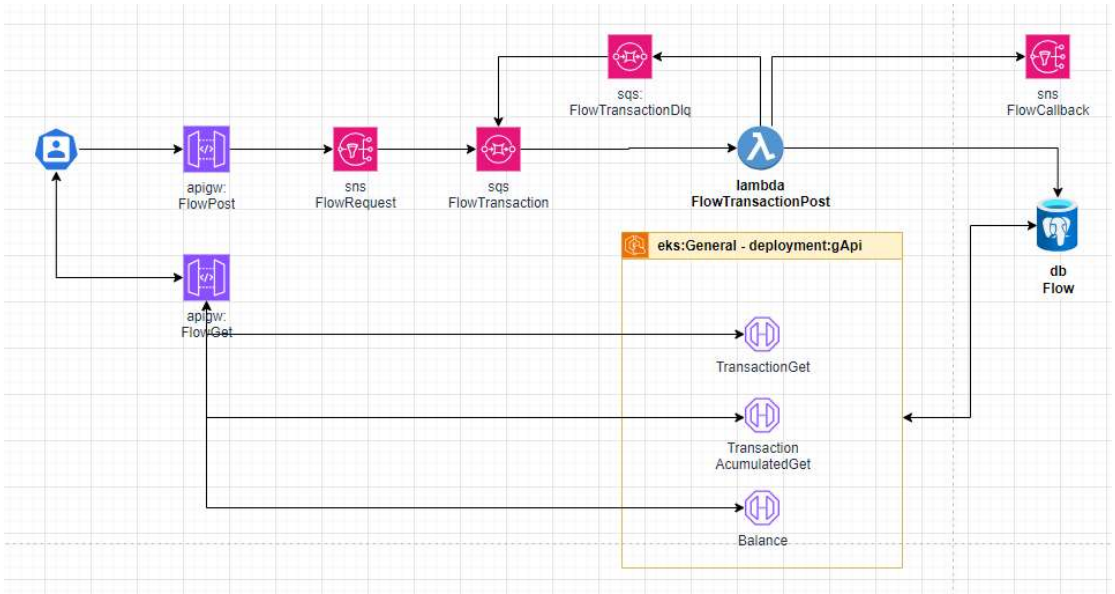

Com custo mais em conta, API GAPI rodando em EKS GENERAL lê as mensagens em lote persiste no POSTGRESQL DB FLOW e comunica resultado pelo tópico SNS FLOWCALLBACK.

Todo o processo, tanto gravação no banco quanto envio no tópico recebe tratamento de POLLY, que realiza várias tentativas para resolução de falhas eventuais.

Já na linha de baixo é só o uusário efetuando suas consultas, detalhes de transação, extrato e detalhes de saldo de conta.

Custo do EKS é mais em conta que o da LAMBDA, utilizada no próximo modelo, que também não possui ainda base espelhada.

**Segundo Esboço**




A diferença deste está no uso de uma LAMBDA FLOWTRANSACTIONPOST para registro do impacto no banco e encaminhamento do resultado via SNS FLOWCALLBACK.

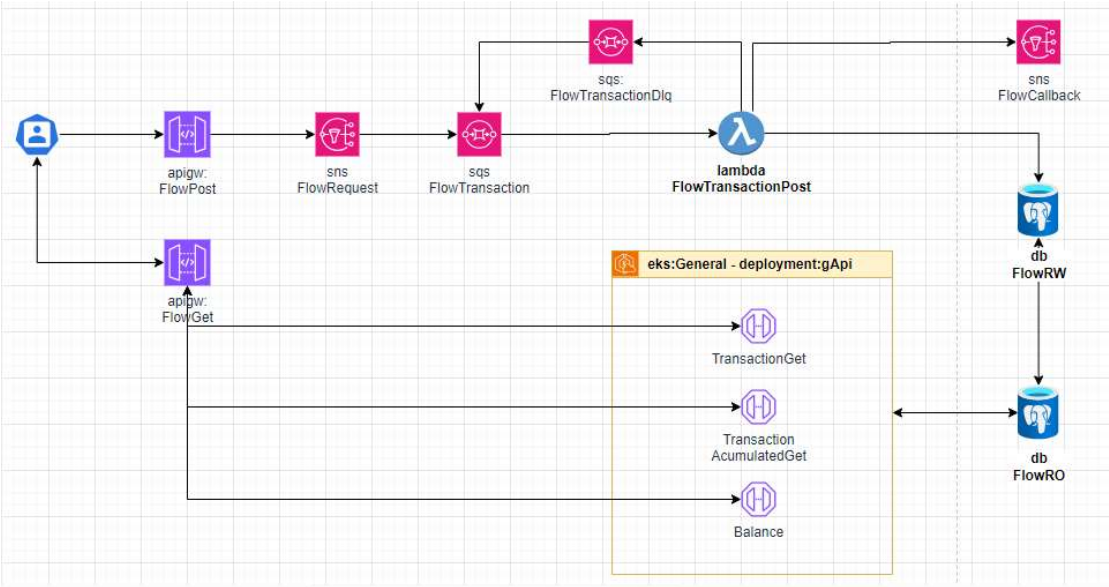
A Lambda permite escala facilitada embora seu custo seja alto visto que é cobrada por memória utilizada x tempo de utilização.

Mas o banco de dados, apesar de possuir configuração de IOPs de leitura/escrita, ainda

pode comprometer leitura com tanta escrita.

Então temos a opção do terceiro esboço.

**Terceiro Esboço**




Única diferença do modelo anterior é que utiliza réplicas apenas para leitura, sendo possível existirem várias até. Trata-se de uma sugestão para tornar ainda mais escalável a solução, antes que possamos pensar em outros tipos de banco, como é o caso do DynamoDB.

# Banco de Dados

Grande parte do custo do projeto é o banco de dados e depende diretamente dele questões como escalabilidade e resiliência. Portanto, é uma das decisões mais importantes do projeto.

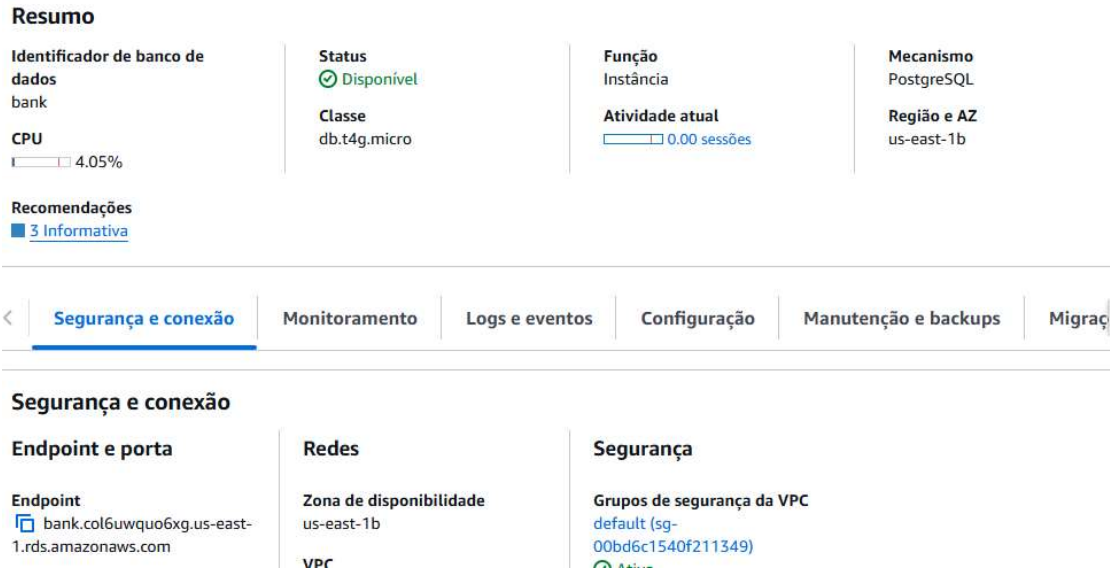
Existem opções como DynamoDB com excelente performance(não relacional), velocidade de totalmente configurável no que diz respeito à capacidade de leitura e escrita, mas seu custo também é elevado.

Por isso optei por melhor custo/belefício, que é o POSTGRESQL. Levando em consideração o objetivo do projeto que é exercitar conceitos, creio que funcione bem para o propósito.

Apesar de ser relacional possui boa performance e não deixa a desejar em confiabilidade.

Para laboratório utilizei conta criada especificamente com essa finalidade na AWS, onde

criei uma base RDS Postgresql(plano gratuito).




Antes mesmo de criar a conta eu já tinha feito em máquina local testes com as tabelas para saber o que fazer após a base criada no RDS. Utilizando DOCKER facilmente temos o postgresql acessível em nossa máquina para testes locais.

```
docker run --name my-postgres -e POSTGRES_PASSWORD=beleza123 -d -p 5432:5432 postgres
```

AWS cobra também por fluxo na rede, a base local também torna-se uma grande opção para testes de massa.

# Experiência Anterior (com instituição financeira)

Alguns detalhes que foram obtidos após muita experiência com problemas reais em instituições financeiras.

TRACEKEY (um guid) facilita bastante a depuração. Permite relacionar registros de uma mesma execução durante todo o fluxo. Isso auxilia muito no entendimento de contextos complexos e, principalmente, agiliza na resolução de problemas.

DATA DE CRIAÇÃO em todos os registros. Trata-se de ótima forma para montar linha do tempo, também bem útil no entendimento de problemas complexos.

Campos de MOMENTO (data/hora) sempre considerando horário GMT. Em soluções distribuídas é importante que utilizemos qualquer campo do tipo data/hora armazenando

no horário GMT.

Uso de CTEs em consultas no POSTGRESQL. É mais confiável e simples de se entender, pois um contexto de retorno serve de entrada para o outro.

...e por aí vai. São 4 anos quase de experiência prática em PRODUÇÃO, tendo que entender e resolvendo problemas.

# Estrutura de Tabelas

Outor ponto que pude aprender com a experiência é que temos sim a possibilidade de agência e conta serem campos ALFANUMÉRICOS. Claro que depende do banco, mas optei por essa abordagem.

Faz parte até do modelo de segurança sendo atribuídos textos para contas virtuais, por exemplo, que desta forma a instituição financeira teria total certeza de que aquela conta não migraria para o sistema finaceiro em hipótese alguma.

IMPORTANTE:

Uma possível evolução do sistema deveria contemplar arquivo morto para evitar crescimento do banco indiscriminado.

Postgresql também simplifica esse processo pelo uso de tabelas particionadas (com PART_MAN) sendo possível "desatachar" com o tempo as mais antigas e salvá-las via CRONJOB no LAKE.

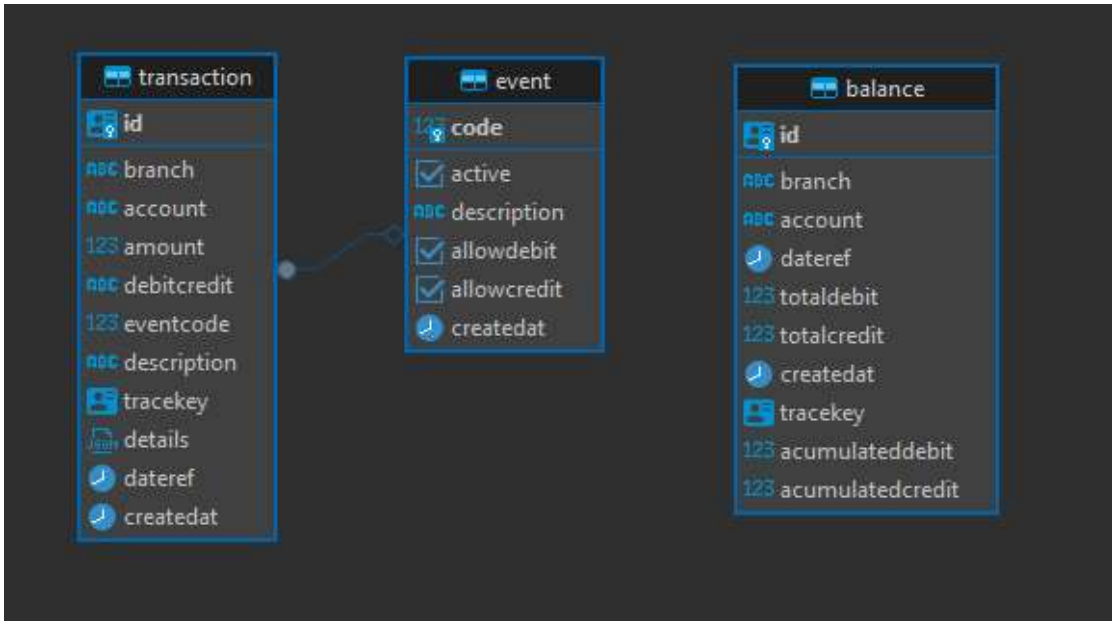
Optei por criar poucas tabelas até por entender que o objetivo maior seria avaliar as aptidões. Evitei também criar as tabelas de cadastro, mas seriam essenciais como tabelas de BRANCH e ACCOUNT, por exemplo.

TRANSACTION para armazenamento das transações.

EVENT exemplificando um dos cadastros.

BALANCE, para armazenamento dos saldos diários.

Esse é o ponto principal de qualquer fluxo de conta.




Índices e validações também são incluídas após o ambiente minimamente operacional para os testes. Entrariam em uma possível revalidação e não foram adicionadas no momento. Apenas algumas poucas validações como é o caso do campo DEBITCREDIT que aceita só 'C'=crédito ou 'D'=débito.

**Dicionário de dados**, TABELAS:

event

code serial4 -> Código de evento sequencial.

active bool -> Se evento está ativo ou não. Desabilitado o banco não mais receberia impacto com esse evento.

description text -> Texto descritivo do evento. Utilizado para agrupar os lançamentos.

allowdebit bool -> Se evento pode ser utilizado para impactar como débito.

allowcredit bool -> Se evento pode ser utilizado para impactar como crédito.

createdat timestamptz DEFAULT now() -> Data e hora da criação.

transaction

id uuid DEFAULT gen_random_uuid() -> GUID como chave para a transação, utilizado como "confirmação" de gravação.

branch text -> Agência, formato desejável "0000". Permite string podendo ser uma agência

virtual, utilizada por questões de segurança.

account text -> Conta, formato desejável "000000000". Permite também texto, utilizada por questão de segurança.

amount numeric(15 2) -> Valor, sempre positivo pois existe o campo de tipo de lançamento: débito ou crédito.

debitcredit varchar(1) -> Tipo de lançamento, valores válidos 'D'=Débito ou 'C'=Crédito.

eventcode int4 -> Código do evento, campo relacionado com a tabela EVENT.

description text -> Descrição adicional na transação.

tracekey uuid -> Tracekey fornecido pelo usuário e registrado durante todo o fluxo. Apenas para fins de depuração.

details json -> Campo que registra mais dados relacionados à transação, que podem ser detalhes de origem do impacto. Exclusivamente de responsabilidade da ORIGEM.

dateref date -> Data de referência do impacto. Prudente a base limita em dias no máximo, exemplo 7 dias para receber impactos retroativos apenas. Após isso não seriam mais aceitos.

createdat timestamptz DEFAULT now() -> Momento de criação da transação.

balance

id uuid DEFAULT gen_random_uuid() -> GUID para referência daquele saldo (agência/conta/dateref).

branch text -> Agência.

account text -> Conta.

dateref date -> Data de referência para cálculo do saldo.

totaldebit numeric(15 2) -> Total de débitos do dia.

totalcredit numeric(15 2) -> Total de créditos do dia.

createdat timestamptz DEFAULT now() -> Momento de criação desse saldo (TODO: Atualização também seria importante).

tracekey uuid -> Tracekey utilizado na criação (TODO: Tracekey de atualização também seria importante).

acumulateddebit numeric(15 2) -> Acumulado considerando todos os dias até esse de total

de débitos.

acumulatedcredit numeric(15 2) -> Acumulado considerando todos os dias até esse de total de créditos.

# Processo de Desenvolvimento e Implantação

Também não foram informadas as condições de desenvolvimento exceto uso do GIT. Tomei a liberdade então para imaginar como seriam as etapas para um controle adequado de resultados das sprints.

Inicialmente seriam montados os ambientes. Idealmente estamos falando de 3:

* Desenvolvimento

* Homologação

* Produção

Desenvolvimento para qualquer teste e determinação de estratégia.

Homologação, validação final da versão antes da subida definitiva e, em alguns casos, produção paralela quando precisamos "replicar" condições de produção para uma espécie de validação real, que não deixa de ser homologação.

Utilizaríamos CodePipelines (ou similar) para realização dos Deploys, mas que no momento optei por não realizar a configuração devido ao tempo e custos envolvidos.

Tenho grande experiência também com as PIPES do Azure, que considero ótima ferramenta.

Neste momento também optei por criar objetos diretamente na AWS, apenas como experimento. O Ideal seria uso de CloudFormation para garantir controle via STACK e versionamento da infraestrutura, que também seria armazenado em REPOSITÓRIO.

OS TESTES também estariam integrados nas automações de pipe sendo exigidos 100% de aproveitamento para realização do deploy do próximo nível. Homologação exigiria 100% dos testes com sucesso de DEV e PROD 100% dos testes com sucesso em Homologação. A maioria das plataformas de automação de pipes possui essa validação.

Importante também em cada deploy de PROD existir uma documentação capaz de evidenciar testes feitos. Essa "formalidade" blinda o processo exigindo o mínimo de análise antes de qualquer subida, sendo também um amparo LEGAL. É excelente prática.

**Componente de PostgreDbContext**

Em toda minha esperiência a possibilidade de manter o banco escamotiável, facilitando troca do tipo de banco de dados nunca funcionou bem. Injetar interface de banco acaba comprometendo e limitando muito pela necessidade de compatibilidade além de nunca ter sido realmente uma opção trocar de banco durante o projeto.

Em um cenário em que essa troca é necessária, inevitavelmente teremos que refazer consultas e repensar todo o sistema, pois o que pode ser uma boa solução para um banco nem sempre é para outro.

Por isso optei pela abordagem de DbContext, assumindo o banco FlowDb como Postgresql.

É muito comum que uma mesma solução tenha bancos de tipos diferentes, como DynamoDb ou SqlServer. Neste caso seriam novos contextos no repositório, fácil de organizar.

Preferi também não me extender muito nos relacionamentos pois distanciaria demais da linha principal de análise. Criei cadastros apenas para EVENTO, mas a mesma solução deveria ser aplicada as cadastros AGÊNCIA e CONTA, por exemplo.

Outro ponto que deixei de fora é o Entity Framework, pois acaba sendo anti-produtivo. Hoje em dia o SQL é tão vasto e complexo que acaba sendo uma grande limitação o uso de qualquer framework para manipulação do banco de dados.

O Data Migrations também vai pelo menos caminho com outro agravante. Em bancos o banco de dados recebe tanta preocupação que, normalmente, ocorre auditorias específicas relacionadas a eles sendo bem mais clara a análise em codificação DDL. Devido a isso optei por enviar os scripts como executados na base.

# Motor Transacional

Sem dúvida é preciso decidir o que é melhor. Se montar um esquema de triggers ou atualizações transacionais para manter os saldos atualizados ou desacoplá-los para uma consolidação eventual.

Em minha experiência o desacoplamento tem se mostrado excelente solução pois se tudo bem o saldo aparece rapidamente, mas caso ocorra algum problema, as mensagens se acumulam para tratativa posterior, o que é bem tolerado por processos mais críticos.

Mas de qualquer modo sempre existirá um MOTOR TRANSACIONAL que obtém as mudanças de conta corrente e totaliza para que não tenhamos que "calcular" saldos durante o processo de consulta, o que não seria possível.

Neste nosso experimento o motor transacional nada mais é que uma sequência de CTEs que executam várias atividades explorando a capacidade transacional da base para garantia de saldo da forma mais performática possível.

Como o tempo de experimento é limitado optei por fazer desta forma, até por ser uma vontade minha desde muito tempo realizar esse teste, mas preocupa-me a escalabilidade, se o banco apresentaria problemas de bloqueio e performance no geral, mesmo com a criação de índices.

Segue o código:

```
        var sql = """
            with ins_trans as (
                insert into flow.transaction (branch,account,
                    amount,debitcredit,
                    eventcode,description,
                    tracekey, details, dateref)
                    select @Branch, @Account, @Amount,
                    @DebitCredit, @EventCode, @Description,
                    @TraceKey::uuid, @Details::json, @DateRef
                RETURNING *
            )
            , block_day as (
                select * from flow.balance where (branch,account,dateref)
in
                    (select balance.branch,balance.account,balance.dateref
from
                    ins_trans inner join flow.balance on
                    ins_trans.branch = balance.branch and
                    ins_trans.account = balance.account and
                    ins_trans.dateref <= balance.dateref) for update
            )
            , acum as (
```

```
                select balance.branch, balance.account, balance.dateref,
                    balance.acumulateddebit, balance.acumulatedcredit
                from flow.balance inner join ins_trans
                    on balance.branch=ins_trans.branch and
balance.account=ins_trans.account
                    and balance.dateref<ins_trans.dateref
                order by balance.dateref desc limit 1
            )
            , total as (
                select branch, account,
                    case when debitcredit='D' then amount else 0 end
totaldebit,
                    case when debitcredit='C' then amount else 0 end
totalcredit
                from ins_trans
            )
            , ins_balance as (
                insert into flow.balance(branch,account,dateref,

totaldebit,totalcredit,acumulateddebit,acumulatedcredit,tracekey)
                select ins_trans.branch, ins_trans.account,
                    ins_trans.dateref,
                    total.totaldebit,
                    total.totalcredit,
                    coalesce(acum.acumulateddebit,0)+total.totaldebit,
                    coalesce(acum.acumulatedcredit,0)+total.totalcredit,
                    tracekey
                from
                    ins_trans
                    left join acum on ins_trans.branch=acum.branch and
ins_trans.account=acum.account
                    join total on ins_trans.branch=total.branch and
```

```
ins_trans.account=total.account

                on conflict (branch,account,dateref)

                do update set

                    totaldebit = balance.totaldebit +
coalesce(EXCLUDED.totaldebit,0),

                    totalcredit = balance.totalcredit +
coalesce(EXCLUDED.totalcredit,0),

                    acumulateddebit = balance.acumulateddebit +
coalesce(EXCLUDED.totaldebit,0),

                    acumulatedcredit = balance.acumulatedcredit +
coalesce(EXCLUDED.totalcredit,0)

                returning *

            ), update_next_balances as (

                update flow.balance

                set

                    acumulateddebit = acumulateddebit + total.totaldebit,

                    acumulatedcredit = acumulatedcredit + total.totalcredit

                from ins_trans,total

                where balance.dateref > ins_trans.dateref and

                    balance.branch = ins_trans.branch and

                    balance.account = ins_trans.account

            )

            select id::text, tracekey::text from ins_trans
```

Explicando rapidamente:

Foi feito para incluir transação uma por uma.

Explicando cada trecho(CTE):

```
ins_trans -> insert da linha na transação

block_day -> para bloqueio das linhas no balance que serão ajustadas

acum -> acumulado do dia anterior daquela conta

total -> melhor nome seria "resume", serve para evitar repetição deb/cred
```

```
ins_balance -> atualiza balance do dia

update_next_balances -> atualiza balance dos dias para a frente
```

Por final retornamos indicando operação bem sucedida:

```
id::text, tracekey::text from ins_trans
```

# Monitoramento Integrado

Processo de análise da solução arquitetural envolve sempre monitoramento, que é idealizado no momento elaboração de todo o projeto, ou seja, assim que um dado elemento é criado busca-se formas de garantir seu funcionamento.

Essa é exigência é fundamental para qualidade de software.

O monitoramento se dá essencialmente por métricas, que podem ser utilizadas em dashs de acompanhamento ou nos alertas.

Os logs também devem ser analisados para gerar alertas.

Considerando o ecosistema da AWS, já é possível criar vários alertas capazes de sinalizar por mensagem seja para TEAMS ou para um celular qualquer eventualidade.

Um dash também auxiliar no processo de esclarecimento e ajuda a consolidar informações importantes capazes de auxiliar na solução de problemas.

Atualmente utilizo o DATADOG, que diferente do CloudWatch, não cobra por alarme criado. Também oferece inúmeras formas de realizar a busca e tratar métricas, sendo bastante versátil e eficaz.

Mesmo que não seja o DATADOG, a base da estratégia de monitoramento é a consolidação de logs e existem várias ferramentas capazes de captura da console dos sistemas os logs gerados e apresentá-los em um painel (ferramenta LOKI por exemplo).

# Etapas Críticas

Em todo projeto é importante definir ponto críticos para se atacar com mais testes

Neste experimento considero dois pontos mais crítico:

MOTOR TRANSACIONAL:

* Várias atualizações ao mesmo tempo (necessidade de limitar impacto retroativo);

* Integridade transacional com vários containers rodando em paralelo, executando atualizações ao mesmo tempo;

* Possibilidade de bloqueio, também pelo motivo da concomitância principalmente caso seja utilizada a solução de LAMBDA;

* Gravação adequada no TÓPICO DE CALLBACK, fundamental para que o usuário possa acompanhar o que acontece com seus pedidos. A sinalização no tópico de callback em particular precisa de uma atenção extra, pois ocorre a gravação no banco primeiro e logo após é enviada a mensagem no tópico. Importante garantir transacionalmente. Caso a mensagem não seja gravada por algum motivo será essencial efetuar rollback da transação.

OBSERVAÇÃO:

Outra opção para o rollback da transação no caso de falha no envio do tópico seria desacoplar ainda mais criando uma etapa extra de análise (raw_data), mas deixaria a solução bem mais complexa além de aumentar o custo.

É importante levantar esses pontos critíticos de funcionamento e tratar como casos de teste.

CONFERÊNCIA DE SALDOS:

* O saldo acumulado é sempre um desafio. Existem vários fatores que influenciam no saldo como impactos retroativos, ordem judicial ou ajustes de saldo inicial.

* Também existe a possibilidade de alguma falha acontecer, ainda mais em ambientes de concorrência extrema.

A conferência de saldos deverá periodicamente, principalmente durante os testes.

# Log de Erros (DATADOG)

Erros NÃO DEVEM ser apresentados para usuários sem o devido tratamento.

Por isso existe tratamento em cada controle, mas uma melhoria desejada seria criar o contexto genérico, embora agora não tenhamos todos os cenários de erro.

De qualquer modo o detalhamento do erro que não aparece para o usuário precisa ser armazenado para a equipe técnica.

Por isso a opção por utilizar o SERILOG, que já ajusta em formato adequado para o entendimento além de ser possível com extrema facilitade adatá-lo para o DATADOG ou para qualquer ferramenta similar de consolidação de logs.

Vou para meu 4 ano utilizando o DATADOG para consolidação de mensagens com ótimos resultados e por isso preferi montar a ferramenta como se fosse enviar logs para ela.

Uma das preocupações quando se usa DATADOG, para evitar questões com o custo é formatar os logs em JSON. Assim temos a garantia de que todo o log é apenas um evento. Enviando no modo texto existe sempre a chance da indexação de vários eventos para uma mesma mensagem de log.

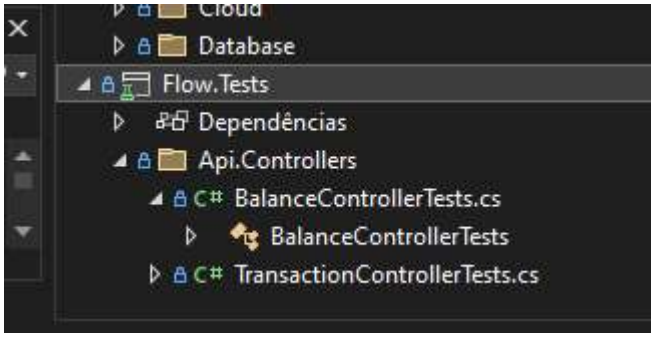
SERILOG facilita esse procedimento bastando mudar uma linha para formatar o log como JSON:

Deixei sinalizado no código:

```
/// <returns></returns>
1 referência
public static IHostBuilder ConfigureSerilog(this IHostBuilder host)
{
    host.UseSerilog((ctx, config) => config
        .Enrich.FromLogContext()
        .Enrich.WithProperty("hostname",
            Environment.GetEnvironmentVariable("HOSTNAME") ??
            Environment.GetEnvironmentVariable("COMPUTERNAME") ??
            "undefined"
        )
        .WriteTo.Console()
        // TODO: Enable for DATADOG -> .WriteTo.Console(new JsonFormatter())
    );
    return host;
}
```

# Sobre Testes

Incluí apenas alguns exemplos.




Programação dos testes precisa ser bem minuciosa o que exigiria substituição profunda até mesmo na camada de banco para evitar que parem de funcionar devido a falta de dados ou troca de banco, o que levaria muito tempo.

Mantive para esse experimento a camada de banco ativa, acessando a base.

Vai depender de como é montada a pipe, mas normalmente se utiliza mesmo o ambiente DEV para os testes unitários, que são realizados antes do deploy onde é criada a imagem(docker) que seguirá para UAT e depois para PROD. Pelo menos assim é o modo que até hoje vivenciei mais seguro e produtivo.

Testes de integração também não foram incluídos pelo mesmo motivo, o tempo, mas devem ser pensados desde o início do projeto sendo complementados a medida que novos recursos são incluídos.

Exemplo de caso de teste de integração:

* Salvamento de saldo das contas/datas selecionadas para o teste (sugestão: 1000 transações)

* Inserção na fila das transações.

* Aguardar execução de todos os tracekeys.

* Conferir extrato das contas e datas em questão.

Inclusive, o armazenando do tempo de realização do teste de integração oferece a oportunidade de acompanhar performance básica conforme as mudanças vão acontecendo sendo possível identificar pontos de atenção na codificação, além das falhas.

Como o teste de integração confere saldo de datas/contas e não somente os tracekeys envolvidos, deverá ser feito de modo "exclusivo". Portanto precisaríamos de uma flag de suspensão cautelar para que fossem feitos impactos durante o processo.

# Melhorias Propostas

Algumas ideias de melhorias propostas:

* Padronização de mensagems no sistema incluindo numeração R001-Mensagem...Algo assim. Auxilia na montagem do sistema de monitoramento.

* Classe centralizada para padronização de tratamento de erros.

* Otimização pela criação de índices baseados nas consultas e no motor transacional.

* Ajuste em AddTransactions para persistir em lote retornando TraceKey de todos e resultado.

* Implementação de paginação das buscas.

* Montar procedimento periódico de conferência de saldo acumulado apenas sinalizando diferenças.

* Ativação de ferramenta de consolidação de logs.