{reprograma}

ONG Reprograma - Processo Seletivo
Estágio: Engenharia de dados
Faculdade: Engenharia de Computação - UFU

DESAFIO 2

Planejamento de um Pipeline

Isabella Vecchi Ferreira

Uberlândia-MG, 28 de julho de 2023

Questão 1.

1.1. Planejamento

“*Como você planejaria as etapas desta migração? Discorra um pouco sobre cada etapa, se possível.*“

Primeiramente pensaria **Camadas de Aplicação** necessárias para programar um fluxo de dados de um banco a outro.

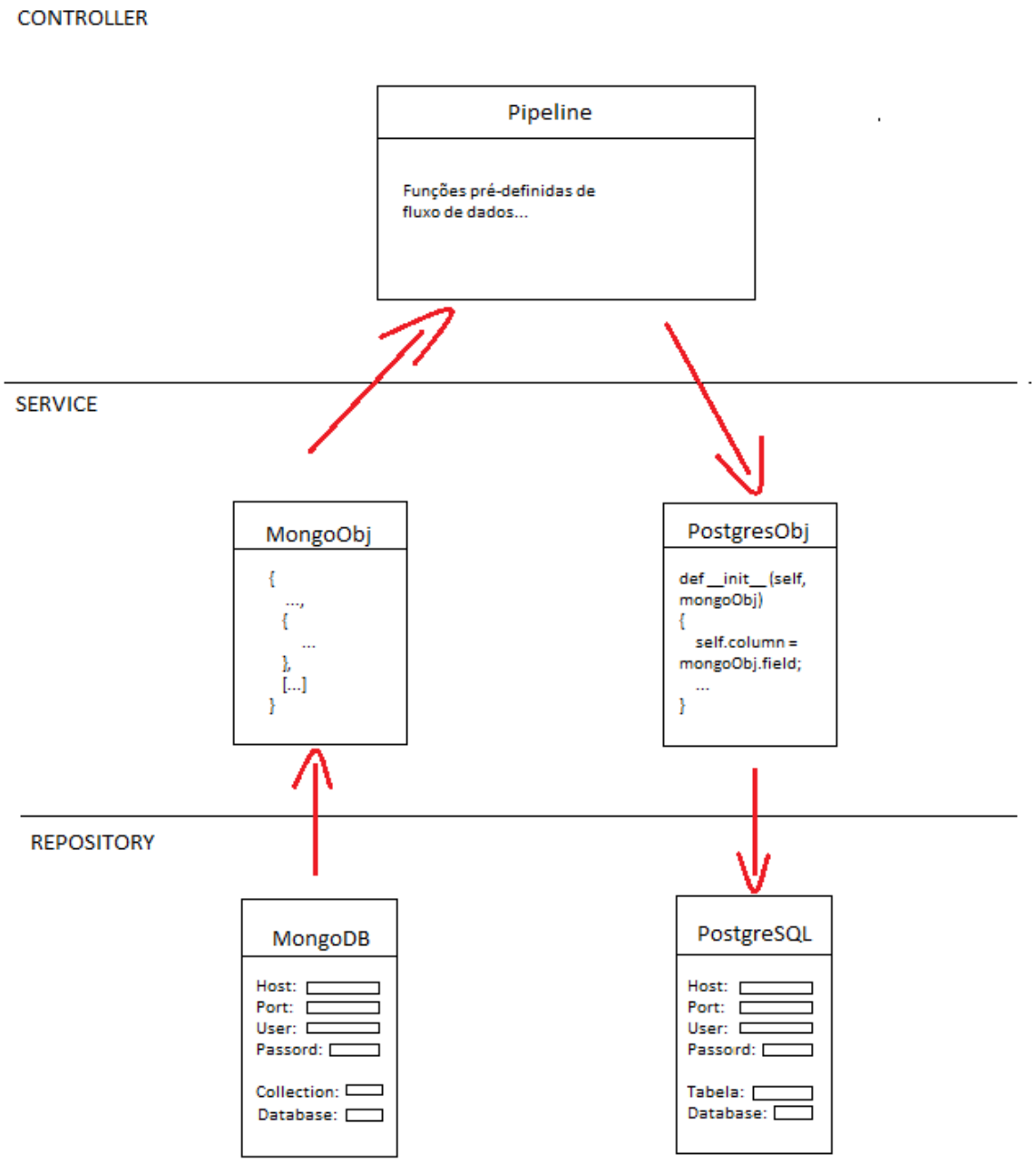

Figura 1.

Em programação orientada a objetos, é imprescindível que as funções das camadas estejam bem separadas e definidas, de forma com que sub-camadas consigam prover serviço a um ou mais entes de uma camada superior.

Desta forma, uma **camada de conexão a banco** (*repository*), com uma classe voltada para o **MongoDB** e uma outra classe para a base de dados **(BD) relacional**. Faria-se necessária para organizar as ações de **CRUD** que estas instâncias precisariam.

Recebendo apenas um tipo Objeto e um tipo string - que seria o nome da collection/tabela a ser alterada/lida - sem que a classe ficasse presa a apenas um modelo de dados.

Assim, independentemente de quando e quantos formatos de dados deseja-se migrar, bastaria criar os **modelos de classes**: transcrevendo o modelo existente **do documento da collection do mongo** e criando o **da linha da tabela SQL** desejado.

Onde também poderia-se criar queries mais personalizadas do que as gerais, caso necessário. É a esta camada de *serviço* que adicionaria-se novos objetos a cada novo pipeline a ser criado, além dos modelos criados.

E então uma **camada de controle do fluxo de dados** (*controller*), sendo construída por objetos e strings para nomes de tabela e collection, automatizando o fluxo de uma para outra.

A BD relacional de minha escolha seria o open source PostgreSQL. Se configurado adequadamente, é um ótimo banco de produção com várias funcionalidades bem versáteis.

Estes seriam os primeiros passos da construção da aplicação em si. Para cada pipeline que formos criar, é necessário um estudo de caso para **modelar os dados** para o banco relacional da melhor forma possível. Validar se é necessário **queries específicas** e se o modelo do mongo está igual os da tabela, caso não, fazer as **alterações de dataframe** necessárias para aquele modelo.

Obs.: Vale lembrar que o fluxo de dados poderia ser desenhado nas duas direções. As funções implementadas variam de acordo com a necessidade da aplicação.

1.2. Tecnologias/Ferramentas

Sabe-se que atualmente o mercado está repleto de ferramentas para automatizar pipelines. Alguns exemplos são: Google Cloud Composer, AWS Step Functions, várias da fundação sem fins lucrativos Apache Software Foundation, como o Apache Airflow, entre outras.

No entanto, deve-se atentar ao recente problema, advindo junto com a tecnologia de nuvens, chamado de “Vendor Lock-in”. No texto deste link, descorro um pouco mais sobre o assunto.

É muito comum que grandes empresas comprem serviços de fonte aberta e eles então passem a ser pagos.

Mas sem dúvidas, são ferramentas que aceleram bastante o processo de desenvolvimento.

Portanto, para que a aplicação possa rodar tanto em nuvem, quanto On Premise, nesta tarefa aberta, foi decidido trabalhar com containerização através de docker.

Com 1 container para o banco mongodb, outro para o postgresql, outro para a api e mais um para o sistema de mensageria. Todos separados, no entanto na mesma network, definida pelo docker-compose.

1.3. Riscos da migração

E se o sistema ficar fora do ar?

Teria como armazenar as requisições que foram feitas, antes de dar erro?

E se o erro ocorrer durante uma sequência de CRUDS? Como eu saberia quais alterações foram bem sucedidas ou não?

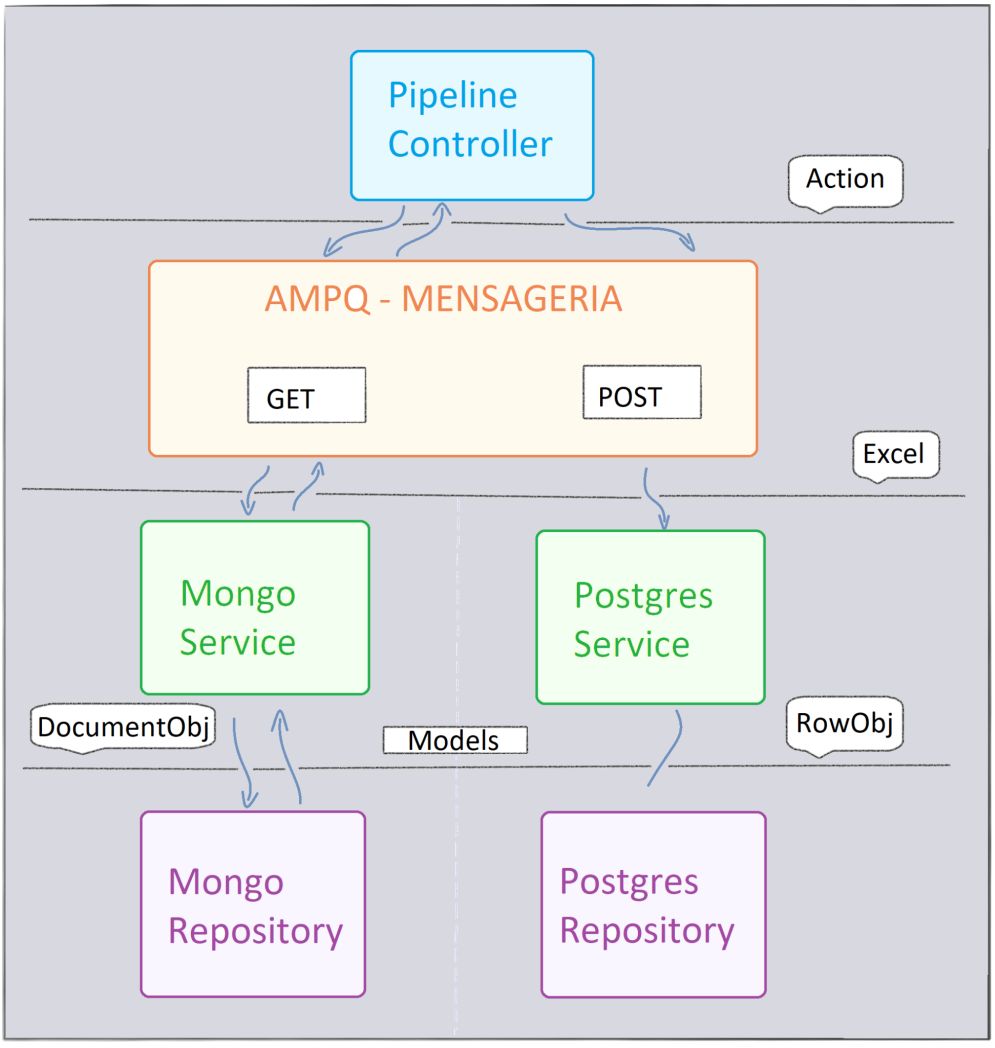

Figura 2.

Bom, vamos por partes. Respondendo as duas primeiras perguntas: o sistema de mensageria "**RabbitMQ**", que é compatível com python, resolveria estas questões.

Toda e qualquer requisição feita na aplicação, que precisa acessar uma BD, teria a **requisição armazenada na fila** do RabbitMQ. De forma, que caso o banco esteja fora do

ar, as requisições **aguardam** na fila **até que o serviço do banco esteja ativo** e pronto para consumir as requisições enfileiradas por ordem de chegada.

A terceira pergunta já seria mais trabalhosa de se resolver pelo sistema de mensageria. Uma forma mais prática de se lidar com isto é através das **sessions** de **conexão do banco**.

Onde as alterações só são escritas na memória do banco, após um **commit**. Todas as alterações podem ser desfeitas através de um **rollback** (caso não tenham sido commitadas) caso ocorra alguma falha na execução do código.

Desta forma, precisaria-se apenas executar a tarefa novamente, sem fazer-se necessária uma curadoria dos dados.

Para o acréscimo desta camada, o desenho da aplicação ficaria parecido com o esquemático da figura 2. Leia a descrição da figura a seguir:

A controller, responsável pelo pipeline, se comunica com a camada de mensageria através de Ações com ou sem parâmetros. No caso desta aplicação, requisitou uma ação de GET, obtendo o resultado e depois executou uma ação de POST.

O serviço de Mensageria insere dados diretamente através da requisição de inserção. No entanto, quando recebe uma requisição de GET, consegue enviar apenas um ACK ou NACK confirmando ou não a busca. Os dados buscados são então escritos num documento excel. Que a controller lerá depois de receber a confirmação de que o acesso ao banco foi executado com sucesso.

Como no caso desta aplicação, os dados inseridos no Mongo vêm de tabelas excel, e os dados inseridos no Postgres, vêem do Mongo, que se transformam em Excel, para então serem enviados para serem inseridos no Postgres… Podemos dizer que o Protocolo de Comunicação entre a camada *AMPQ* e a de *serviço*, são os arquivos excel.

E então, para finalmente tentar se conectar ao banco, a camada de serviço passa os dados de Classe do Objeto (que descreve o dado da collection/tabela a ser alterada/lida) e o nome da collection/tabela. Estes podem ser considerados os protocolos de comunicação entre *serviço* e *repositório*, os arquivos que se encontram dentro do diretório *models*.

E para manter os dados atualizados nas 2 bases?

Para esta questão, adota-se a adição de três campos em cada tabela SQL, denominados por exemplo: “dt_start”, “dt_end” e “fg_status”. Onde ficam armazenados os dados de data de criação daquela tupla, qual a última vez que foi modificada, e se aquela tupla ainda está ativa ou não.

Isto serve apenas para ajudar na governança de dados. Para realmente manter os 2 bancos sincronizados e atualizados entre si, é necessário construir fluxos de pipeline, garantindo que toda ação (com exceção de queries) feita em um, também seja feita no outro.

Questão 2.
Link para a apresentação Power Point:
https://docs.google.com/presentation/d/1iuJgEiKxNRuKjYlAWON0BTtang03VjB6Icp0u54TgdY/edit?usp=sharing

Extra.
Link para repositório do GitHub com o desenvolvimento da aplicação com RabbitMQ; o script para a geração de gráficos e também a versão pdf da apresentação dos dados:
https://github.com/isabellavecchi/data-engineer