Documentação do Projeto

Este projeto consiste em disponilizar endpoints para os dados obtidos através de web scrapping do site http://books.toscrape.com.

Este documento tem como objetivo documentar o funcionamento técnico e detalhado deste projeto.

Arquitetura

A arquitetura do projeto segue a arquitetura de camada, ou seja, cada camada é separada e escala independente das demais, podendo escalar o projeto sem ter que alterar muito o código pelo baixo acoplamento do mesmo.

A camada de api é onde se encontra os endpoints disponibilizados. Nela temos também um arquivo de dependencias, onde criamos os objetos de serviço necessários para os endpoints, criando uma espécie de singleton para o projeto. Esta prática possibilita utilizar somente uma instância, garantindo assim um menor uso de memória RAM, ainda mais pensando em múltiplos usos da API.

A camada do core é onde se encontra os arquivos de configuração e o bootstrap. Os arquivos de configuração vão trazer os valores das variáveis de ambiente e configurar conexões com bancos (postgresql e redis) .

A camada de models é onde se encontram as classes de modelo do projeto, se dividindo em entidades e dtos. As entidades são as classes que refletimos no banco de dados, e os dtos são as classes que trafegamos na API.

A camada de repositories é onde se encontra as chamadas do banco de dados relacional. Cada classe que vai para o banco de dados relacional (no caso desse projeto, é apenas a entidade usuários) tem um repository para fazer as queries sql.

A camada de security é onde definimos o núcleo da segurança, ou seja, criação e validação do token jwt caso as credenciais inseridas sejam validas.

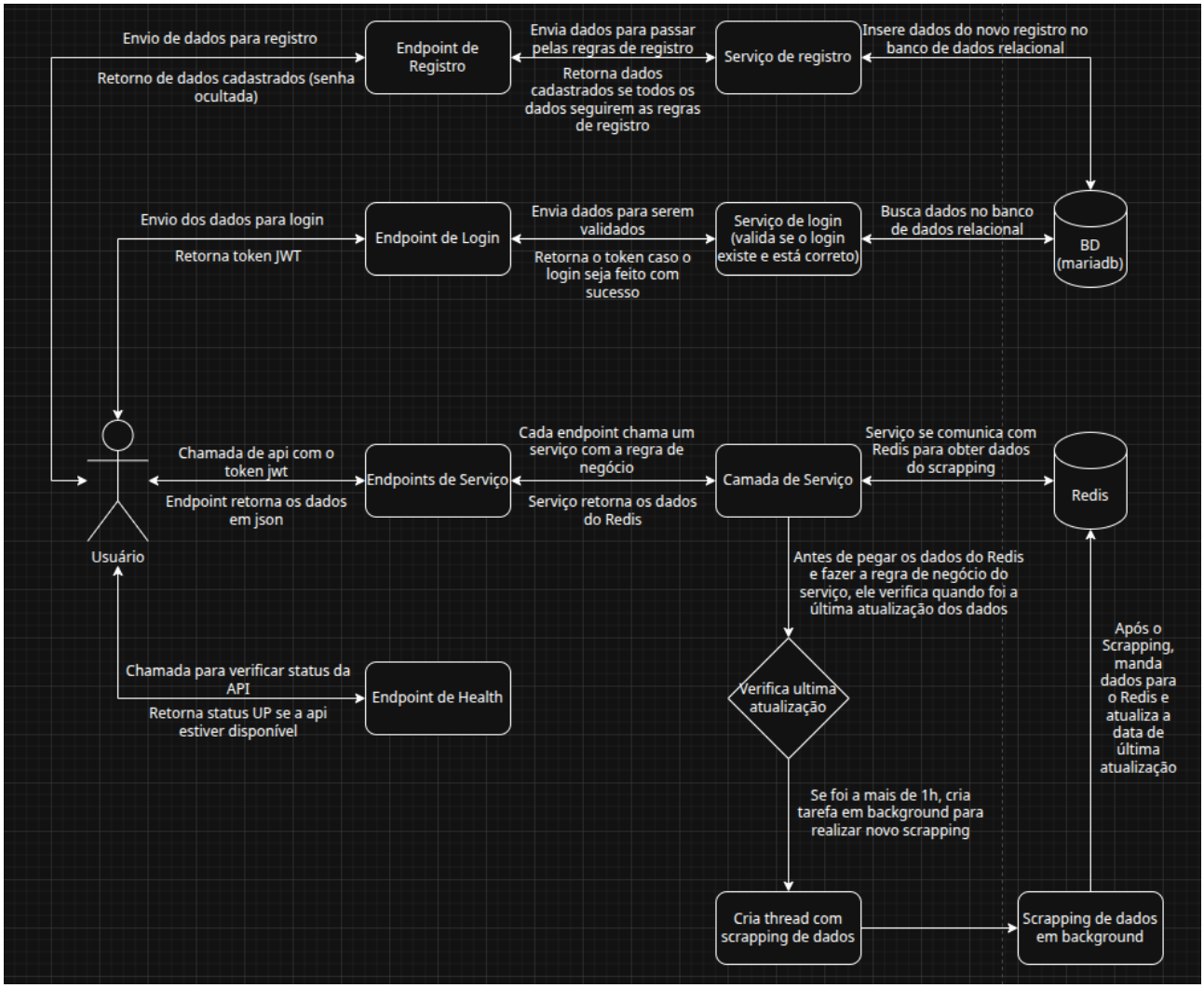
A camada de services é onde ficam as classes de serviço, ou seja, toda lógica de negócio do nosso projeto, incluindo a lógica de web scrapping, se encontra nessa camada. Aqui também fazemos uma classe de serviço para o redis e para a autenticação.

Por fim, temos a camada de util, onde guardamos métodos de utilidade para o projeto, para auxiliar outras classes em suas tarefas.

Fluxo da API

Sempre que a API é iniciada, ela vai rodar o web scrapping, para que os dados fiquem o mais atualizados possíveis. Após isso, sempre que for chamada qualquer rota, é feita a verificação dos dados, se a ultima alteração ter sido feita a mais de 1 hora, é criada uma thread em paralelo que realiza um novo web scrapping, para atualizar os dados.

A Imagem abaixo mostra o fluxo geral da API em cada chamada.



Com este fluxo, garantimos a integridade e atualização dos dados sempre para os usuários que chamam a API.

As rotas das API's são protegidas, exigindo um registro e um login para acessar os serviços (exceto o endpoint de health, que verifica se a API está em funcionamento).

Lógica

Como dito no fluxo geral, temos dois fluxos, um fluxo ao subir o projeto, e um sempre que é chamado um endpoint de serviço.

O arquivo main.py, após as conexões com os bancos de dados feita, ele vai executar o método extract_and_save_books() da classe ExtractService (em app.services.extract). Este método vai realizar o web scrapping e salvar eles no Redis, para maior velocidade de leitura dos dados.

Após isso, ele inicia o FastAPI normalmente incluindo as rodas na camada de api.

Cada chamada de serviço exige um token JWT, obtido a partir da rota de login após o registro feito pelo endpoint de registro.

Ao chamar um serviço, ele vai chamar os métodos de alguma classe de serviço que realiza a lógica de negócio para trazer os dados requisitados. Dentro desta classe de serviço, existe a lógica que verifica a data de ultima atualização. Sempre que salvamos os dados do scrapping, nós salvamos um registro no Redis com a chave LAST_UPDATE, contendo a data exata de quando salvamos os registros no Redis. Neste método que verifica, utiliza desse registro para descobrir a última vez que o web scrapping foi executado com sucesso.

Após essa verificação, se a data está dentro de 1 hora, então ele não faz nada, mas se passou de 1 hora, então ele executa novamente o web scrapping, e depois salva os dados obtidos no Redis, atualizando também o registro que contém a data da última atualização, garantindo assim sempre os dados atualizados para os usuários.

Esta lógica possui um lock, justamente para mais de uma chamada lançar mais de um web scrapping simultâneo, assim garantimos que a API não se sobrecarregue ou caia por conta de diversas threads.

Sempre que essa lógica de verificação e possível web scrapping é feita, é realizada em uma thread em paralelo, ou seja, o cliente recebe os dados de qualquer maneira, garantindo velocidade também na resposta da API.

Esta lógica foi escolhida justamente para que os dados sempre fiquem atualizados independente de um agente externo, e também garanta a velocidade de entrega dos dados.

Tecnologias escolhidas

As tecnologias e os motivos delas terem sido escolhidas são:

Banco de Dados Relacional: PostgreSQL
O motivo pela escolha deste banco de dados foi sua alta popularidade, fazendo com que exista em praticamente todo host e máquina, aumentando a portabilidade do projeto sem a necessidade de ficar instalando um em toda máquina que o projeto for instalado.
Além disso, fizemos também a portabilidade de usar MariaDB no nosso projeto, para assim ter um segundo tipo de BD para caso não possa usar o PostgreSQL. Para usar o MariaDB, basta apenas modificar as variáveis de ambiente.

Banco de Dados Não Relacional: Redis
O motivo pela escolhia deste banco de dados foi sua alta velocidade. O Redis é um banco de dados não relacional em memória, ou seja, ele salva os dados na memória RAM, e por isso ele é tão rápido. Isso faz com que a disponibilidade de dados sempre sejas rápida, além da facilidade de inserção de dados, ou seja, caso os tipos de dados mude, a lógica com o Redis não vai mudar, pois ele salva o json independente do que está dentro dele.

Framework: FastAPI
O motivo pela escolha do FastAPi foi sua velocidade e facilidade. O FastAPI faz api's rápidas e faceis de disponibilizar endpoints, e como queremos escalabilidade e velocidade, esta foi nossa escolha.

Linguagem: Python
O motivo pela escolha do Python foi justamente o seu poder para machine learning, facilitando ainda mais a escalabilidade para integração com modelos de Machine Learning.

Deploy: Docker
O motibo pela escolha do docker é sua portabilidade, basicamente, qualquer sistema que tiver o docker consegue rodar nossa aplicação sem a necessidade de mais nenhuma instalação, sendo perfeito para rodar em qualquer host ou máquina.