Stacks:

- python
- flask para HTTP
- grpc
- postgres

- A funcionalidade foi desenvolvida seguindo arquitetura de micro-serviços com um BFF(*back-for-front*) para centralizar a comunicação.
- Temos um serviço focado nos dados de cliente (service-client), um focado em autenticação (service-authorization) e fora a parte, fiz uma implementação bem simples para mockar os endpoints de produto, uma vez que a api não estava funcionando.
- Escolhi a arquitetura de micro-serviços porque pensei em como o produto se desenvolveria em larga escala e como poderia se encaixar com outras funcionalidades que possam surgir
- Como até o momento só guardamos nome e email dos clientes para dados de credenciamento, optei pela autenticação OTP (*one time password*), onde o usuário entra com seu email e um código é gerado, o usuário retorna esse dado e um token será retornado para permitir acesso aos recursos que necessitam autenticação
- Os serviços foram implementados seguindo a arquitetura Clean Architecture (sim, é o mesmo nome do livro porque vem do autor do livro hehe😅). Optei por essa arquitetura por alguns motivos:
    - É algo que não vi muita gente usando e eu gostaria de adquirir o conhecimento
    - A teoria salta muito aos olhos: apesar de parecer um pouco difícil de entender no começo, quando a gente pensa em sistemas grandes, serviços/produtos com muitas regras de negócio e casos de uso, essa arquitetura torna mais fácil criar funcionalidades novas, comunicá-las entre si e talvez um dos maiores atrativos na minha opinião: permite uma atualização mais fácil e desacoplada de tecnologias como bancos de dados, ORMs, e até mesmo a linguagem em que foi implementado. Com o Clean Architecture podemos facilmente separar regras de negócio e o que são necessidades específicas de código da tecnologia aplicada
    - Pensei em várias experiências onde uma simples atualização de versão de alguma tecnologia virava uma task extensa e com risco de causar muitos prejuízos, técnicos, de produto, mão de obra dos times e muito mais para decidir aplicar essa arquitetura, também pensei no tamanho das aplicações que poderia encontrar na Luiza Labs e como essa forma de codificar poderia ser útil

---

Algumas considerações e débitos técnicos:

O plano inicial para esta implementação era usar o redis e cache para otimizar as listas de produtos e produtos favoritos, infelizmente essa parte fica como débito técnico 😓

Devido ao meu problema com meu computador e para não atrasar ainda mais, estou entregando aqui as funcionalidades:

- Criar usuário
- GET usuário (com autenticação)
- Update do usuário (com autenticação)
- Criar código OTP para autenticação
- Validar código OTP para autenticação
- Listar produtos
- Criar lista de favoritos (com autenticação)

Coisas que pretendo continuar evoluindo:

- Adicionar os testes unitários
- Tratamentos de erros mais robustos
- Paginação mais eficiente das listas
- Listagem de favoritos
- Mudanças na lista de favoritos com o redis e acessar as imagens mockando o s3 da AWS
- Otimização com cache
- Autenticação entre os serviços por meio de certificados

(OBS: Caso seus comandos make não funcionem, você pode substituir nos arquivos Makefile o comando **docker compose** por **docker-compose,** isso acontece devido a divergência de versões do docker compose)

Como levantar o ambiente:

- service-client:

execute:

- Entre no diretório do service-client e execute:

$make init

$make build

$make grpc

$make up

$make db-create

$make up (sim, de novo 😅)

- service-authorization

Execute:

- Entre no repositório do service-authorization e execute:

$make build

$make grpc

$make up

$make db-create

$make up

- service-product:

- Execute dentro do repositório:

$docker compose up --build

- bff-app

- Execute no repositório:

$make build
$make grpc

$make up

Executando as rotas:

- Para criar um usuário:

    - Acesse em um client como postman a url: POST http://0.0.0.0:5025/client, passe no body um payload no formato:
      {
      “name”: <string>,
      “email”: <string>
      }

      A resposta será semelhante:

```
{
   "data": {
       "client_id": 13
   },
   "message": "client"
}
```

- Criar codigo otp:

- POST http://0.0.0.0:5025/client/login/otp e passe no body um payload semelhante ao abaixo:
  {
  "email": "let@let.let.com"
  }

  A resposta retornada:

```
{
   "code": "310952"
}
```

- Para validar o código acesse POST http://0.0.0.0:5025/client/login/otp/validate com um body semelhante:

```
{
      "email": str,
      "code": str
}
```

O retorno:


```
{
   "auth_id": 14,
   "code": "310952",
   "id": 18,
   "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb2RlIjoiMzEwOTUyIiwiYXV0aF9pZCI6MTQsImNsaWVudF9pZCI6MTMsImNyZWRlbnRpYWwiOiJsZXRAbGV0LmxldC5jb20iLCJleHAiOjE3MzclODU3MTB9.M5ZJnz5-U9rqfRDRsdwppGEoYufHW2iJkugzC8M41LM"
}
```


- Para adicionar um produto na sua lista: acesse POST http://0.0.0.0:5025/product/favorite/<product_id> passando no bearer token o token retornado na rota de validação

O retorno:


```
{
   "favorite_product_id": 9
}
```


- Para atualizar um cliente PUT http://0.0.0.0:5025/client/update junto com o token no campo bearer token. No body passe todos os campos de um cliente, incluindo a atualização que está fazendo:

EX: Antes

```
{
```

```
      “nome”: “Let Let Lelelelet”,
      “email”: “let@let.com”
}
```

O body passado para atualizar o email:

```
{
      “nome”: “Let Let Lelelelet”,
      “email”: “let@luizalabs.com”
}
```

Você também pode acessar os dados de autenticação do cliente por meio do bff-app.

EX: GET http://0.0.0.0:5025/auth/<str:email> retorna os dados referentes a autenticação do cliente