# Apresentação

Este é um programa que simula uma funcionalidade real da Zapay: a busca de débitos veiculares. No fluxo real, um cliente acessa nosso frontend, insere as informações de seu veículo (comumente placa e renavam) e é apresentado para a lista de débitos associados ao mesmo.

Você encontrará os seguintes arquivos anexados a este documento:

- **main.py**: o arquivo principal, que executa o programa. Ele recebe três parâmetros: debt_option, license_plate e renavam. Um exemplo:

  python main.py ticket ABC1234 11111111111

  onde:

  - ticket é o parâmetro “debt_option”, que pode ser "ticket", "ipva" ou “dpvat”, representando multas, IPVA e seguro obrigatório, respectivamente.
  - ABC1234 é o parâmetro “license_plate”. É a placa do veículo.
  - 11111111111 é o parâmetro “renavam”, e é o… Renavam do veículo.

- **api.py**: possui a classe API, que representa uma API externa possuidora de informações sobre débitos veiculares de São Paulo.
- **service.py**: possui a classe SPService, responsável pela comunicação com a API.
- **parser.py**: possui a classe SPParser, responsável pela conversão dos dados enviados pela API (e resgatados pelo SPService) em um formato compreensível pelo frontend.
- **tests.py**: deve possuir os testes unitários dos métodos do programa.

# Instruções

1. Sinta-se livre para modificar o que quiser em todos os arquivos, exceto o api.py. Por exemplo, se achar que um dado método pode ser refatorado, ou completamente reescrito, fique à vontade.
2. Dito isso, não modifique o arquivo api.py! Ele representa uma API externa no mundo real, então não teríamos controle sobre ele.
3. Para criar e executar os testes, você precisará da biblioteca pytest. Você pode instalá-la com o comando pip install pytest, e executá-la com o comando pytest tests.py.
4. Caso não queira usar o pytest, sinta-se livre para escrever seus testes com a tecnologia de sua preferência.

5. A API trabalha com valores inteiros em centavos. Então R$ 121,22 são representados como 12122. O SPParser converte este valor para float em reais.

6. Utilize Git, assim facilita o seu trabalho de enviar o projeto e o nosso de revisar! Pode subir o projeto na sua plataforma de preferência, seja ela github, gitlab ou qualquer outra!

# Atividades

1. Repare que a classe SPService recebe um parâmetro chamado “debt_option”, que pode ser “ticket”, “ipva” ou “dpvat”. Na prática, isto significa que o usuário precisa fazer uma pesquisa para cada tipo de débito que quer pagar - e se ele quer pagar todos os débitos de seu veículo, precisará fazer três pesquisas! Não seria melhor se ele pudesse fazer tudo de uma vez? Para isso, atualize a implementação do SPService e do SPParser (bem como quaisquer modificações no arquivo main.py que julgar necessárias) de modo que haja um jeito de retornar todos os débitos. Mas lembre-se, o envio do parâmetro “debt_option” deve continuar funcionando, mesmo que ele passe a ser não obrigatório!

2. A API possui ainda um outro tipo de débito, ainda não implementado, chamado Licenciamento. Trata-se do pagamento da taxa de licenciamento do veículo, muito requisitado pelos clientes. Você pode consultá-lo no arquivo api.py. Seu formato é um pouco diferente dos demais. Implemente um parser para ele, de modo que retorne as chaves “amount”, “title”, “description”, “year" e “type”. Seu “debt_option” e seu “type" devem ser “licensing”.

3. Um problema começou a ocorrer com a chegada das novas placas do Mercosul: a API de São Paulo não havia se atualizado ainda para o formato novo, e apenas compreendia a versão antiga daquela placa. Assim, um usuário que fizesse uma pesquisa com a placa ABC1C34 receberia a mensagem “Veículo não encontrado”, simplesmente porque a API não era capaz de converter esta informação para ABC1234. Implemente um método no service que faça esta conversão pela API, para que os clientes com placas atualizadas possam pesquisar seus débitos. Você pode testar usa implementação pesquisando com a placa ABC1C34.


Caso não saiba, a conversão das placas Mercosul é feita desta maneira.

4. Faça testes! Não precisa testar todos os métodos, apenas dois ou três já bastam. Caso queira fazer mais testes do que isso, no intuito de mostrar seus conhecimentos, tanto melhor.

5. Transforme esse script numa api simples, pode utilizar Flask, FastAPI ou qualquer outro framework de sua preferência, você só precisa demonstrar sua familiaridade com a ferramenta. Deve conter somente 1 endpoint que retorne a busca dos débitos, você é livre para escolher o verbo http, não existe resposta certa somente a escolha

mais adequada na sua visão, mesma coisa para o formato de recebimento das informações organize como preferir.

6. BÔNUS: use mocks! Quando estamos testando o SPService, que está muito atrelado à API, o ideal é que não façamos testes que de fato acessam esta API. Isto pode gerar vários problemas, como testes falhando aleatoriamente em caso de indisponibilidade, ou valores imprevisíveis. Ao invés disso, podemos mockar os resultados da API. Caso queira fazer testes com mocks, utilize a tecnologia de sua preferência.

7. BÔNUS: crie um Dockerfile para rodar sua mais nova API! Ele não precisa ter nada além disso, somente conseguir rodar seu mais novo sistema!

8. BÔNUS: sabe trabalhar com ReactJS e quer mostrar para a gente? Maravilha! Implemente um frontend simples que se alimente das informações geradas pelo SPParser. Não precisa fazer uma API se não quiser: contanto que o formato dos dados usados no frontend seja o mesmo, não faz mal se estiverem “hardcoded".