# DOCUMENTAÇÃO DO DESENVOLVIMENTO

**Nome do desenvolvedor:** Alexandre Cabral do Espírito Santo

**Data de início:** 22/11/2024

**Data de término:** 27/11/2024

## 22/11/2024

### TAREFAS REALIZADAS:

- Criação do documento de requisitos.
- Criação do ambiente virtual.
- Instalação do Flask e SQLAlchemy.
- Criação dos primeiros arquivos de código.

### PROBLEMAS ENCONTRADOS:

- Por enquanto, só indecisão de como criar os documentos e organizar o projeto.

### SOLUÇÕES/DECISÕES:

- Decidi fazer os documentos de forma simples.
- Por enquanto planejo deixar todas as rotas em um só arquivo routes.py, como não há muito a ser feito. Talvez eu mude isso se ficar muito grande e ruim de entender.
- O mesmo vai valer para os models, colocarei tudo em um arquivo models.py.

### REFLEXÕES:

- Sendo minha primeira vez organizando e documentando tanto assim, acabei gastei muito tempo pensando no que fazer e como fazer, mas isso me ensinou várias coisas sobre organização de projeto.

## 23/11/2024

### TAREFAS REALIZADAS:

- Criação do requirements.txt.

### PROBLEMAS ENCONTRADOS:

- Precisei passar meu projeto para outro computador em que estaria a base de dados, precisando instalar todas as dependências.

### SOLUÇÕES/DECISÕES:

- A solução/decisão foi justamente a criação do requirements.txt, que facilita nesse trabalho.

### REFLEXÕES:

- Isso ajudou a pensar no passo-a-passo para instalar o projeto.

## 24/11/2024

### TAREFAS REALIZADAS:

- Comecei a fazer o modelo da database.
- Criação do ".env" para guardar variáveis possivelmente sensíveis.
- Criação do passo a passo de como rodar o projeto (README.md).
- Criação da base de dados.

### PROBLEMAS ENCONTRADOS:

- Inicialmente eu queria deixar o projeto com as variáveis vazias para que os usuários preencham com seus dados. Mas não há como eu rodar o projeto com elas vazias, então eu teria que ficar tirando e colocando toda vez que eu precisasse fazer commit, o que seria ineficiente.
- Há muitas formas diferentes de se fazer os modelos, então está sendo complicado escolher o que mais se enquadra na situação.

### SOLUÇÃO/DECISÃO:

- O que decidi fazer sobre o primeiro problema é, pelo menos por enquanto, deixar o ".env" no ".gitignore" e colocar nas instruções para o próprio usuário criar o arquivo e como deveria ser feito. No final do projeto talvez eu faça o commit desse ".env" com variáveis vazias, mas por enquanto vai ficar assim.
- Para o problema número dois, eu decidi apenas ir fazendo, me preocupando com detalhes depois que eu conseguir fazer o programa rodar sem problemas de execução, já que este é o mais importante.
- Decidi deixar as configurações da base de dados junto ao modelo, visto que não há muita coisa a ser configurada e só tem uma tabela que precisa ser feita. Caso fique muito ruim, eu mudo depois.

### REFLEXÕES:

- Aprendi que colocar o arquivo no gitIgnore após ele já estar "commitado" não adianta, então preciso ter ainda mais cuidado com variáveis sensíveis.
- Existem diversas formas de se resolver um mesmo problema, mas não há porque perder muito tempo escolhendo a "melhor", apenas faça e depois adapte para o que você precisa.

## 25/11/2024

### TAREFAS REAIZADAS:

- Criação completa do modelo Partner.
- Conexão com a base de dados feita.
- Primeiro partner adicionado na base de dados com sucesso.
- Organização do projeto.
- Instalei uma biblioteca que me permite mexer com coordenadas na base de dados.

### PROBLEMAS ENCONTRADOS:

- O programa não estava tendo permissão para conectar com a base de dados.

- Estava acontecendo uma importação circular, por causa da importação do app em outras áreas do código.
- Programa não mudava as informações junto da ".env", como se tivesse um arquivo temporários que as guardava.

### SOLUÇÕES/DECISÕES:

- Precisei instalar bibliotecas que facilitavam a conexão com a base de dados.
- Precisei Fazer um arquivo separado apenas para instanciar a base de dados, chamado database.py.
- Modulei o código, colocando tudo dentro de um diretório "app", que tem um arquivo "__init__.py", onde fiz todas as configurações iniciais necessárias.
- Coloquei uma configuração no carregador do ".env" que o faz colocar as informações que estão no arquivo toda ver que o código rodar.

### REFLEXÕES:

- Eu inicialmente Iria deixar o código mais compacto (com só três arquivos), mas vi que isso só complicou mais a minha vida. Imagino que é melhor sempre fazer tudo modulado, separado e organizado mesmo.

## 26/11/2024

### TAREFAS REALIZADAS:

- Criação da rota "/create", que cria um novo Partner seguindo as especificações do json do desafio.

### PROBLEMAS ENCONTRADOS:

- Muitos problemas relacionados ao tipo das informações, muitas vezes por vacilo meu, mas a maioria por ser de um formato que nunca usei (WKTElement).
- Coordenadas não podiam ser salvas diretamente, por ser de um formato diferente do que está na base de dados.
- Erros vinham de uma forma que ficava difícil de entender o motivo o que também dificultava no processo de corrigi-los.

### SOLUÇÕES/DECISÕES:

- Criação de uma função "format_multipolygon(coordinates)" dentro de um arquivo "helpers.py", que recebe as coordenadas passadas como array no json do corpo da requisição. A função formata em um tipo válido para a conversão, que seria uma string com números separados por vírgulas e parênteses. Ex: ((1,2),(1,2)).
- Passei a usar try-except para dividir os tipos de erro, além de criar a função "format_error(message, details=None, code=400)", que retorna o erro de uma forma mais amigável em forma de objeto json no console do lugar em que a requisição foi feita. A função é simples, mas ajudou.

### REFLEXÕES:

- Toda a dificuldade que tive em relação a erros e tipos só me fez entender cada vez mais a importância de modular, dividir, comentar e documentar bem o código, além de fazer o código de forma a facilitar a resolução no caso de algum erro ocorrer.

## 27/11/2024 – DIA FINAL!

### TAREFAS REALIZADAS:

- Criação da rota "/loadById", que recebe um id no corpo da requisição e retorna o Partner correspondente.
- Criação da rota "/search", que recebe coordenadas no corpo da requisição e retorna o Partner mais próximo, se elas estiverem dentro de sua área de atuação.

### PROBLEMAS ENCONTRADOS:

- Carregar o Partner na resposta como um objeto json, pois o mesmo não aceitava conversão, o que dificultava as coisas.
- Mais problemas tentando entender como o WKTElement funciona, pois precisava dele para realizar consultas no banco de dados.

### SOLUÇÕES/DECISÕES:

- Em relação à consulta, precisei realizar vários testes na base de dados para entender que tipo de consulta daria certo, o que não funcionou na maioria dos casos e, o que parecia funcionar inicialmente, revelava ser um erro, pois ele mostrava o mais próximo, mas independia de ser dentro da área de atuação ou não. No final achei a função do próprio SQLAlchemy, ST_Within, que fazia exatamente o que eu precisava, e então eu ordenei pela distância retornada da função ST_Distance.
- No que diz respeito ao WKTElement, foi até bem mais simples que os anteriores, já que agora eu só precisava instancia-lo colocando o POINT, que é representado pela latitude e longitude.
- Em relação à resposta, foi criado a função as_dict() na própria classe Partner, onde fiz as conversões necessárias para retorná-lo como um objeto json em ambas as rotas.

### REFLEXÕES:

- Não vou poder dizer com certeza de que ainda não há erros ou se vai funcionar em todos os casos, mas ele faz o que foi proposto a fazer, então vai funcionar contanto que envie um objeto json correto na rota correta. Talvez eu ainda atualize no futuro melhorando a segurança, ainda mais no que diz respeito à conversão de dados, mas por enquanto é isso.