Arquivo Main,py

Este código é a configuração básica para iniciar uma aplicação web usando o framework Flask em Python.

#### Importações

- \*\*Flask\*\*: Importa a classe `Flask` do módulo `flask`, que é usada para criar a aplicação web.

```python

from flask import Flask

```

#### Inicialização da Aplicação

- \*\*app = Flask(\_\_name\_\_, static\_folder='static')\*\*:

- Cria uma instância da aplicação Flask.

- O primeiro argumento, `\_\_name\_\_`, é o nome do módulo atual. Flask usa isso para determinar a raiz do caminho da aplicação e localizar recursos relacionados, como arquivos estáticos e templates.

- O parâmetro `static\_folder='static'` especifica o diretório onde os arquivos estáticos (CSS, JavaScript, imagens, etc.) serão armazenados. Por padrão, o Flask procura uma pasta chamada "static" no diretório raiz do projeto. Este argumento pode ser usado para definir um caminho diferente, se necessário.

```python

app = Flask(\_\_name\_\_, static\_folder='static')

```

#### Execução da Aplicação

- \*\*if \_\_name\_\_ == "\_\_main\_\_":\*\*

- Este bloco de código garante que a aplicação só seja executada se este arquivo for executado diretamente. Isso evita que o código seja executado inadvertidamente se o script for importado como um módulo em outro lugar.

- Dentro deste bloco, o código importa tudo (`\*`) do módulo `views`, onde são definidos os endpoints e rotas da aplicação.

- \*\*app.run(debug=True, port=5000, host='0.0.0.0')\*\*:

- Inicia o servidor de desenvolvimento do Flask.

- \*\*debug=True\*\*: Ativa o modo de depuração, que permite ver os erros em tempo real e reiniciar automaticamente o servidor sempre que o código é alterado.

- \*\*port=5000\*\*: Define a porta na qual o servidor vai escutar. O padrão do Flask é a porta 5000.

- \*\*host='0.0.0.0'\*\*: Define o host como '0.0.0.0', o que permite que o servidor seja acessado externamente, não apenas localmente.

```python

if \_\_name\_\_ == "\_\_main\_\_":

from views import \*

app.run(debug=True, port=5000, host='0.0.0.0')

```

#### Resumo

Este código inicializa uma aplicação Flask básica e define as configurações para executar o servidor localmente, permitindo que a aplicação web seja acessada externamente através do IP e da porta especificados. As rotas e outras funcionalidades da aplicação estão configuradas no módulo `views`, que é importado no momento da execução do script.

---

Arquivo Views.py

Aqui está a documentação detalhada para o código fornecido:

---

### Documentação do Código Flask para Geração de Roteiros

Este código é uma aplicação Flask que permite aos usuários calcular a melhor rota entre dois endereços e gerar um roteiro detalhado, humanizado e com coordenadas geográficas, integrando a API do Google Maps.

#### Importações

- \*\*render\_template\*\*: Renderiza templates HTML com Flask.

- \*\*request\*\*: Manipula requisições HTTP (por exemplo, capturar dados de formulários).

- \*\*flash\*\*: Exibe mensagens de feedback para o usuário.

- \*\*send\_from\_directory\*\*: Serve arquivos estáticos de um diretório específico.

- \*\*os\*\*: Interage com o sistema operacional, como manipulação de variáveis de ambiente.

- \*\*json\*\*: Manipula dados em formato JSON.

- \*\*load\_dotenv\*\*: Carrega variáveis de ambiente de um arquivo `.env`.

- \*\*CalcularRota, DescreveRoteiroRota, HumanizaResposta\*\*: Importa classes customizadas para calcular rotas, descrever o roteiro, e humanizar a resposta.

- \*\*app\*\*: Importa a aplicação Flask a partir do módulo `main`.

```python

from flask import render\_template, request, flash, send\_from\_directory

import os

import json

from dotenv import load\_dotenv

from calcula\_rota import CalcularRota

from descrever\_roteiro import DescreveRoteiroRota

from humanizar\_resposta import HumanizaResposta

from main import app

```

#### Carregamento das Variáveis de Ambiente

- \*\*load\_dotenv()\*\*: Carrega as variáveis de ambiente a partir do arquivo `.env`.

- \*\*google\_maps\_api\_key\*\*: Pega a chave da API do Google Maps armazenada no arquivo `.env`.

```python

load\_dotenv()

google\_maps\_api\_key = os.getenv('MAPS\_API\_KEY')

```

#### Rotas da Aplicação

1. \*\*Rota Inicial ("/")\*\*:

- \*\*@app.route("/")\*\*: Define a rota para a página inicial da aplicação.

- Cria um arquivo `pontos.json` vazio no diretório `files` para armazenar os dados de pontos geográficos.

- Renderiza o template `index.html`, passando a chave da API do Google Maps como variável.

```python

@app.route("/")

def index():

dados\_vazios = {}

with open('files/pontos.json', 'w') as arquivo:

json.dump(dados\_vazios, arquivo, indent=4)

return render\_template('index.html', api\_key=google\_maps\_api\_key)

```

2. \*\*Serviço de Arquivos Estáticos\*\*:

- \*\*@app.route('/files/<path:path>')\*\*: Serve arquivos da pasta `files`, acessíveis via URL.

- \*\*@app.route('/icon\_map/<path:path>')\*\*: Serve imagens de ícones da pasta `icon\_map`.

```python

@app.route('/files/<path:path>')

def send\_static(path):

return send\_from\_directory('files', path)

@app.route('/icon\_map/<path:path>')

def send\_img(path):

return send\_from\_directory('icon\_map', path)

```

3. \*\*Rota para Geração de Roteiro ("/gerar\_roteiro")\*\*:

- \*\*@app.route("/gerar\_roteiro", methods=['POST'])\*\*: Define a rota para processar o formulário enviado via método POST.

- Captura os endereços de origem (`endOrigim`) e destino (`endDestino`) do formulário.

- Exibe mensagens de erro (usando `flash`) se algum dos campos estiver vazio e renderiza novamente a página inicial.

- Cria uma instância de `CalcularRota`, chama o método para calcular a melhor rota, e processa o resultado.

- Utiliza a classe `DescreveRoteiroRota` para gerar a descrição detalhada do roteiro e a classe `HumanizaResposta` para melhorar a apresentação final.

- Gera as coordenadas de latitude e longitude dos pontos da rota.

- Renderiza o template `index.html` com o roteiro final, coordenadas e informações de entrada, além da chave da API do Google Maps.

```python

@app.route("/gerar\_roteiro", methods=['POST'])

def gerar\_roteiro():

endOrigim = request.form.get('endOrigem')

endDestino = request.form.get('endDestino')

if not endOrigim:

flash('Necessário informar o Endereço Origem!')

return render\_template('index.html')

if not endDestino:

flash('Necessário informar o Endereço Destino!')

return render\_template('index.html')

calcula\_rota = CalcularRota(endOrigim, endDestino)

rota = calcula\_rota.calcular\_melhor\_rota()

roteiro = DescreveRoteiroRota(rota)

resumo\_rota, descricao\_rota = roteiro.passo\_a\_passo\_rota()

humaniza = HumanizaResposta()

roteiro\_final = humaniza.humanizar\_resposta\_rota(resumo\_rota, descricao\_rota)

coordenadas = calcula\_rota.gerar\_lat\_e\_long\_enderecos(rota)

return render\_template('index.html', roteiro=roteiro\_final, coordenadas=coordenadas, endOrigem=endOrigim, endDestino=endDestino, api\_key=google\_maps\_api\_key)

```

#### Resumo

Este código é responsável por gerenciar uma aplicação Flask que calcula rotas entre dois endereços e gera um roteiro detalhado e humanizado, utilizando a API do Google Maps. O código inclui a configuração das rotas para a página inicial e a geração do roteiro, bem como o tratamento de dados e renderização do resultado para o usuário. A integração com a API do Google Maps é feita através de uma chave armazenada em um arquivo `.env` para garantir a segurança.

---

Arquivo calcula\_rota.py

Aqui está a documentação detalhada para o código fornecido:

---

### Documentação da Classe `CalcularRota`

A classe `CalcularRota` é responsável por calcular a melhor rota entre dois endereços utilizando a API de direções do Google Maps, além de gerar as coordenadas geográficas (latitude e longitude) dos pontos ao longo da rota e armazená-las em um arquivo JSON.

#### Importações

- \*\*load\_dotenv\*\*: Carrega variáveis de ambiente de um arquivo `.env`.

- \*\*os\*\*: Permite interagir com o sistema operacional, como acessar variáveis de ambiente.

- \*\*requests\*\*: Biblioteca utilizada para fazer requisições HTTP, como chamadas para a API do Google Maps.

- \*\*json\*\*: Manipula dados em formato JSON, incluindo leitura e escrita em arquivos.

```python

from dotenv import load\_dotenv

import os

import requests

import json

```

#### Carregamento de Variáveis de Ambiente

- \*\*load\_dotenv()\*\*: Carrega as variáveis de ambiente a partir de um arquivo `.env`, como a chave da API do Google Maps (`MAPS\_API\_KEY`).

```python

load\_dotenv() # Carregar variáveis de ambiente do arquivo .env

```

#### Classe `CalcularRota`

A classe `CalcularRota` possui dois métodos principais:

1. \*\*Método Construtor (`\_\_init\_\_`)\*\*:

- Inicializa a classe com os endereços de origem e destino.

- Define a URL base da API de direções do Google Maps.

- Obtém a chave da API do Google Maps das variáveis de ambiente.

```python

class CalcularRota:

def \_\_init\_\_(self, endOrigin, endDestination):

self.api\_key = os.getenv('MAPS\_API\_KEY')

self.api\_url = f'https://maps.googleapis.com/maps/api/directions/json'

self.endOrigin = endOrigin

self.endDest = endDestination

```

2. \*\*Método `gerar\_lat\_e\_long\_enderecos`\*\*:

- \*\*Objetivo\*\*: Extrair as coordenadas de latitude e longitude de cada etapa da rota e salvá-las em um arquivo JSON.

- \*\*Parâmetros\*\*: Recebe os dados da rota em formato JSON (resultado da API do Google Maps).

- \*\*Processo\*\*:

- Percorre as etapas da rota (`steps`) para capturar as coordenadas de início e fim de cada etapa.

- Armazena as coordenadas em uma lista.

- Escreve a lista em um arquivo `pontos.json` no diretório `files`.

- \*\*Retorno\*\*: Retorna a lista de coordenadas geradas.

```python

def gerar\_lat\_e\_long\_enderecos(self, data):

json\_final = []

for leg in data[0]['legs']:

for step in leg['steps']:

start\_coords = {"lat": step['start\_location']['lat'], "lng": step['start\_location']['lng']}

json\_final.append(start\_coords)

end\_coords = {"lat": step['end\_location']['lat'], "lng": step['end\_location']['lng']}

json\_final.append(end\_coords)

with open('files/pontos.json', 'w') as f:

json.dump(json\_final, f)

return json\_final

```

3. \*\*Método `calcular\_melhor\_rota`\*\*:

- \*\*Objetivo\*\*: Fazer uma requisição à API de direções do Google Maps para calcular a melhor rota entre os endereços de origem e destino especificados.

- \*\*Parâmetros\*\*: Nenhum (usa os endereços e a chave da API definidos no construtor).

- \*\*Processo\*\*:

- Configura os parâmetros da requisição, incluindo:

- `origin` e `destination`: Endereços de origem e destino.

- `mode`: Define o modo de transporte (neste caso, `transit` para transporte público).

- `avoid`: Evita caminhadas dentro de grandes terminais.

- `traffic\_model`: Usa um modelo pessimista para estimativas de tráfego.

- `transit\_mode`: Especifica o uso de trens como modo de transporte.

- `unit`: Define a unidade de medida para métrica.

- `language`: Define o idioma da resposta como português do Brasil (`pt-BR`).

- `key`: A chave da API do Google Maps.

- Faz uma requisição GET à API do Google Maps com os parâmetros configurados.

- Verifica se a requisição foi bem-sucedida (`status\_code == 200`).

- Verifica se a resposta da API contém rotas (`status == 'OK'`).

- Retorna a lista de rotas caso tudo esteja correto; caso contrário, imprime uma mensagem de erro.

- \*\*Retorno\*\*: Retorna a lista de rotas ou `None` em caso de erro.

```python

def calcular\_melhor\_rota(self):

params = {

'origin': self.endOrigin,

'destination': self.endDest,

'mode': 'transit',

'avoid': 'indoor',

'traffic\_model': 'pessimistic',

'transit\_mode': 'train',

'unit': 'metric',

'language': 'pt-BR',

'key': self.api\_key

}

response = requests.get(self.api\_url, params=params)

if response.status\_code == 200:

data = response.json()

if data['status'] == 'OK':

return data['routes']

else:

print(f"Erro na resposta da API: {data['status']}")

else:

print(f"Erro na requisição: {response.status\_code}")

return None

```

#### Resumo

A classe `CalcularRota` permite calcular a melhor rota entre dois endereços usando a API do Google Maps, especialmente focando em transporte público. Ela também gera um arquivo JSON contendo as coordenadas de latitude e longitude dos pontos ao longo da rota. A classe trata possíveis erros na resposta da API e inclui opções para ajustar o modelo de tráfego, modo de transporte, e outras configurações relevantes para a busca de rotas.

---

Classe `DescreveRoteiroRota`

A classe `DescreveRoteiroRota` é responsável por fornecer uma descrição detalhada das rotas calculadas, incluindo o status das estações de trem e metrô ao longo do percurso. Ela integra funcionalidades como o cálculo de similaridade entre nomes de estações e a obtenção de informações sobre o status das linhas de transporte público.

#### Importações

- \*\*fuzz\*\*: Parte da biblioteca `fuzzywuzzy`, usada para calcular a similaridade entre strings.

- \*\*EstacoesLista\*\*: Classe importada de `lista\_estacoes`, responsável por fornecer informações sobre as estações de trem e metrô.

- \*\*LinhasStatus\*\*: Classe importada de `linhas\_trem\_e\_metro\_status`, usada para obter o status atual das linhas de trem e metrô.

```python

from fuzzywuzzy import fuzz

from lista\_estacoes import EstacoesLista

from linhas\_trem\_e\_metro\_status import LinhasStatus

```

#### Classe `DescreveRoteiroRota`

A classe possui quatro métodos principais:

1. \*\*Método Construtor (`\_\_init\_\_`)\*\*:

- Inicializa a classe com as rotas calculadas que serão usadas para descrever o roteiro.

- \*\*Parâmetro\*\*: `rotas` - Lista de rotas obtidas da API de direções (como a classe `CalcularRota`).

```python

class DescreveRoteiroRota:

def \_\_init\_\_(self, rotas):

self.rotas = rotas

```

2. \*\*Método `conecta\_estacao\_com\_linha`\*\*:

- \*\*Objetivo\*\*: Identificar quais linhas de transporte público passam por uma determinada estação.

- \*\*Parâmetros\*\*:

- `estacao\_rota`: Nome da estação para a qual se deseja encontrar as linhas.

- `limiar\_similaridade`: Valor de corte para considerar duas strings semelhantes (default: 80).

- \*\*Processo\*\*:

- Obtém a lista de estações da classe `EstacoesLista`.

- Gera um DataFrame contendo as informações das estações.

- Compara a `estacao\_rota` com os nomes das estações usando a função `fuzz.ratio`.

- Retorna uma lista das linhas que passam pela estação correspondente.

```python

def conecta\_estacao\_com\_linha(self, estacao\_rota, limiar\_similaridade=80):

try:

estacoes = EstacoesLista()

lista\_estacoes = estacoes.get\_estacoes

df\_estacoes = estacoes.gera\_dataframe\_lista\_estacoes(lista\_estacoes)

colunas\_encontradas = []

for coluna in df\_estacoes.columns:

for valor in df\_estacoes[coluna]:

similaridade = fuzz.ratio(valor, estacao\_rota)

if similaridade >= limiar\_similaridade:

colunas\_encontradas.append(coluna)

break

return colunas\_encontradas

except Exception as e:

print(f"Error encontrado {e}")

return None

```

3. \*\*Método `captura\_status\_estacao`\*\*:

- \*\*Objetivo\*\*: Capturar o status das linhas de trem e metrô a partir de um arquivo CSV.

- \*\*Parâmetros\*\*:

- `colunas\_encontradas`: Lista das linhas que passam pela estação, obtida do método `conecta\_estacao\_com\_linha`.

- `limiar\_similaridade`: Valor de corte para considerar duas strings semelhantes (default: 80).

- \*\*Processo\*\*:

- Carrega o status das linhas usando a classe `LinhasStatus`.

- Filtra as linhas no DataFrame com base no nome da estação.

- Retorna uma lista de valores que descrevem o status da estação.

```python

def captura\_status\_estacao(self, colunas\_encontradas, limiar\_similaridade=80):

linhas\_status = LinhasStatus()

file = linhas\_status.get\_status\_linha()

df = linhas\_status.generate\_data\_frame(file)

status\_linha = []

linhas\_filtro = []

try:

for valor in df['Título']:

similaridade = fuzz.ratio(valor, colunas\_encontradas)

if similaridade >= limiar\_similaridade:

linhas\_filtro.append(valor)

break

df\_filtrado = df[df['Título'].isin(linhas\_filtro)]

df\_filtrado = df\_filtrado.drop("Linha Número", axis=1)

except KeyError:

print("Valores não encontrados na base")

return []

status\_linha = df\_filtrado.values.tolist()

return status\_linha

```

4. \*\*Método `gerar\_texto\_referente\_estacoes`\*\*:

- \*\*Objetivo\*\*: Gerar uma descrição textual detalhada para as estações encontradas, incluindo as linhas e o status das estações.

- \*\*Parâmetros\*\*:

- `colunas\_encontradas`: Lista de linhas associadas à estação.

- `status`: Lista de status das linhas.

- \*\*Processo\*\*:

- Se houver mais de uma linha na estação, concatena os nomes e gera um texto detalhado.

- Se houver apenas uma linha, gera um texto simples com o status.

- Retorna o texto gerado.

```python

def gerar\_texto\_referente\_estacoes(self, colunas\_encontradas, status):

texto = ""

linhas = ""

if colunas\_encontradas == None:

return None

if len(colunas\_encontradas) > 1:

linhas = ",".join(colunas\_encontradas)

stat = ""

for i in status:

stat += f"\n {i[0]} - {i[1]} - {i[2]} "

texto = (f"Estação Baldeio das linhas :{linhas} \n" +

f"Status Estação: {stat}")

else:

linhas = " ".join(colunas\_encontradas)

texto = f"Estação: {linhas} \n Status Estação: {status}"

return texto

```

5. \*\*Método `passo\_a\_passo\_rota`\*\*:

- \*\*Objetivo\*\*: Gerar um resumo e uma descrição detalhada de cada etapa da rota.

- \*\*Processo\*\*:

- Se as rotas estiverem disponíveis, inicia a descrição com o ponto de partida e chegada, além da distância e tempo total.

- Para cada etapa da rota, verifica se há detalhes de trânsito, como informações de embarque e desembarque, e coleta o status das estações usando os métodos anteriores.

- Gera um texto contendo as instruções e os detalhes das estações, incluindo status.

- Retorna o resumo e a descrição detalhada como strings codificadas em `utf-8`.

```python

def passo\_a\_passo\_rota(self):

if self.rotas:

resumo\_rota = ""

descricao\_rota = {}

for rota in self.rotas:

resumo\_rota = (

f"Ponto de Partida: {rota['legs'][0]['start\_address']} \n"

f"Ponto de Chegada: {rota['legs'][0]['end\_address']} \n"

f"Distância Total: {rota['legs'][0]['distance']['text']} \n"

f"Tempo Total: {rota['legs'][0]['duration']['text']} \n"

)

for leg in rota['legs']:

for index, step in enumerate(leg['steps']):

texto\_instrucoes = f"{step['html\_instructions']} ({step['distance']['text']} - {step['duration']['text']})"

texto\_passos = ""

if 'transit\_details' in step:

transit\_details = step['transit\_details']

estacao\_embarque = self.conecta\_estacao\_com\_linha(transit\_details['departure\_stop']['name'])

status\_estacao\_embarque = self.captura\_status\_estacao(estacao\_embarque)

informacao\_estacao\_embarque = self.gerar\_texto\_referente\_estacoes(estacao\_embarque,status\_estacao\_embarque) if self.gerar\_texto\_referente\_estacoes(estacao\_embarque,status\_estacao\_embarque) else None

estacao\_desembarque = self.conecta\_estacao\_com\_linha(transit\_details['arrival\_stop']['name'])

status\_estacao\_desembarque = self.captura\_status\_estacao(estacao\_desembarque)

informacao\_estacao\_desembarque = self.gerar\_texto\_referente\_estacoes(estacao\_desembarque,status\_estacao\_desembarque) if self.gerar\_texto\_referente\_estacoes(estacao\_desembarque,status\_estacao\_desembarque) else None

texto\_passos = (

f"Embarque em: {transit\_details['departure\_stop']['name']} \n"

f"{informacao\_estacao\_embarque} \n"

f"Desembarque em: {transit\_details['arrival\_stop']['name']} \n"

f"{informacao\_estacao\_desembarque} \n"

f"Em direção a: {transit\_details['headsign']} \n"

f"Linha: {transit\_details['line']['name']} \n"

f"Número de estações até o destino: {transit\_details['num\_stops']}"

)

descricao\_rota[f"step: {index}"] = [texto\_instrucoes, texto\_passos]

resumo\_rota\_str = str(resumo\_rota)

descricao\_rota\_str = str(descricao\_rota)

return resumo\_rota\_str.encode("utf-8"), descricao\_rota\_str.encode("utf-8")

else:

print("Não foi possível calcular a rota.")

return None, None

```

---

### Documentação da Classe `HumanizaResposta`

A classe `HumanizaResposta` é responsável por gerar instruções detalhadas e humanizadas para usuários que desejam realizar um trajeto de metrô. Ela utiliza a API do Google Generative AI para processar e personalizar as respostas com base em resumos e descrições detalhadas das rotas.

#### Importações

- \*\*`google.generativeai`\*\*: Biblioteca para interagir com os modelos de inteligência artificial generativa do Google.

- \*\*`os`\*\*: Biblioteca padrão do Python usada para interagir com variáveis de ambiente.

- \*\*`re`\*\*: Biblioteca para manipulação e busca de padrões em strings usando expressões regulares.

- \*\*`dotenv`\*\*: Biblioteca usada para carregar variáveis de ambiente de um arquivo `.env`.

```python

import google.generativeai as genai

import os

import re

from dotenv import load\_dotenv

```

#### Inicialização do Ambiente

O método `load\_dotenv()` é utilizado para carregar as variáveis de ambiente de um arquivo `.env`, permitindo que a chave da API seja acessada pelo programa.

```python

load\_dotenv()

```

#### Classe `HumanizaResposta`

A classe possui três métodos principais:

1. \*\*Método Construtor (`\_\_init\_\_`)\*\*:

- \*\*Objetivo\*\*: Inicializar a classe com a configuração do modelo generativo do Google (neste caso, o modelo `gemini-1.5-flash`).

- \*\*Processo\*\*:

- A chave da API é configurada usando a variável de ambiente `AI\_API\_KEY`.

- O modelo `gemini-1.5-flash` é selecionado e inicializado como `self.model`.

```python

class HumanizaResposta:

def \_\_init\_\_(self):

self.gemini\_model = 'gemini-1.5-flash'

genai.configure(api\_key=os.environ["AI\_API\_KEY"])

self.model = genai.GenerativeModel(self.gemini\_model)

```

2. \*\*Método `aplicar\_negrito`\*\*:

- \*\*Objetivo\*\*: Substituir textos delimitados por `\*\*` por elementos HTML `<strong>`, que aplicam negrito ao texto.

- \*\*Parâmetros\*\*:

- `texto`: String que contém o texto onde as alterações serão feitas.

- \*\*Processo\*\*:

- Usa expressões regulares para encontrar e substituir todos os textos entre `\*\*` por `<strong>texto</strong>`.

```python

def aplicar\_negrito(self, texto):

return re.sub(r'\\*\\*(.\*?)\\*\\*', r'<strong>\1</strong>', texto)

```

3. \*\*Método `humanizar\_resposta\_rota`\*\*:

- \*\*Objetivo\*\*: Gerar uma resposta humanizada e formatada com base em um resumo de rota e sua descrição detalhada.

- \*\*Parâmetros\*\*:

- `resumo\_roteiro`: String contendo um resumo do trajeto, incluindo pontos de partida e chegada.

- `descricao\_roteiro`: Dicionário contendo a descrição detalhada de cada etapa do trajeto.

- \*\*Processo\*\*:

- Um `prompt` é construído com as instruções detalhadas para gerar as respostas.

- A API do Google Generative AI é chamada para processar o `prompt` e gerar o conteúdo humanizado.

- A resposta gerada é formatada para incluir negrito, utilizando o método `aplicar\_negrito`.

- O resultado final é retornado como uma string formatada.

```python

def humanizar\_resposta\_rota(self, resumo\_roteiro, descricao\_roteiro):

prompt = f"""

Sua função é gerar instruções detalhadas para um usuário que deseja realizar um trajeto de metrô. Você receberá um resumo do trajeto e um dicionário com a descrição detalhada de cada etapa.

\*\*Se os endereços de origem e destino for diferente da região metropolitana de São Paulo, informar que você não poderá efetuar esse trajeto\*\*

\*\*Para parte do roteiro que precise pegar linha de ônibus, não informar o status da linha e nem informações como se existe escadas rolantes, elevadores, etc.\*\*

\*\*Adicionar tag HTML para fazer a divisão do roteiro entre parágrafos\*\*

\*\*Não adicione título do roteiro\*\*

\*\*Dados de Entrada:\*\*

\* \*\*Resumo do trajeto:\*\* Uma breve descrição do ponto de partida, destino e principais pontos de interesse.

\* \*\*Descrição detalhada:\*\* Um dicionário onde cada chave representa uma etapa do trajeto e o valor associado contém informações como:

\* Estação de origem e destino

\* Linhas envolvidas (se houver baldeação)

\* Tempo estimado de viagem

\* Observações relevantes (por exemplo, escadas rolantes, elevadores, pontos de interesse próximos, status da operação da linha)

\*\*Saída Esperada:\*\*

\* \*\*Instruções claras e concisas:\*\* Explique cada etapa do trajeto de forma simples, utilizando linguagem natural e evitando termos técnicos.

\* \*\*Informações sobre as linhas:\*\* Indique as linhas envolvidas em cada etapa, alertando o usuário sobre a necessidade de baldeação quando necessário.

\* \*\*Informações dos status de operação das linhas:\*\* Indique o status da operação da linha em questão, se está em Operação Normal ou outro status, e adicione os comentários a respeito do mesmo.

\* \*\*Tempo e distância total:\*\* Informe o tempo estimado de viagem e a distância total do trajeto.

\* \*\*Observações:\*\* Inclua qualquer observação relevante para o usuário, como a existência de escadas rolantes, elevadores ou pontos de interesse próximos.

\* \*\*Mensagem positiva:\*\* Finalize com uma mensagem encorajadora, como "Boa viagem!"

\*\*Informações geradas para criar o Roteiro:\*\*

Resumo do trajeto = {resumo\_roteiro}

Descrição detalhada = {descricao\_roteiro}

"""

resposta = self.model.generate\_content(

[prompt],

stream=True

)

resposta.resolve()

resultado = ""

try:

resultado = resposta.text

except ValueError:

return "Não foi possível gerar o roteiro"

if resultado:

resultado = self.aplicar\_negrito(resultado)

return resultado

```

---

### Documentação da Classe `LinhasStatus`

A classe `LinhasStatus` é projetada para coletar e organizar informações sobre o status das linhas de metrô e trem a partir do site da Viamobilidade. Ela faz uso de web scraping para extrair os dados relevantes e organiza essas informações em um formato tabular, que pode ser exportado como um arquivo CSV.

#### Importações

- \*\*`BeautifulSoup`\*\*: Biblioteca do pacote `bs4`, usada para analisar (parse) documentos HTML e XML.

- \*\*`requests`\*\*: Biblioteca para fazer requisições HTTP, usada para obter o conteúdo da página da Viamobilidade.

- \*\*`pandas`\*\*: Biblioteca usada para manipulação e análise de dados, permitindo a criação e manipulação de DataFrames.

```python

from bs4 import BeautifulSoup

import requests

import pandas as pd

```

#### Classe `LinhasStatus`

A classe possui dois métodos principais:

1. \*\*Método Construtor (`\_\_init\_\_`)\*\*:

- \*\*Objetivo\*\*: Inicializar a classe com a URL da página da Viamobilidade.

- \*\*Processo\*\*:

- A URL da página é armazenada na variável de instância `self.url`.

```python

class LinhasStatus:

def \_\_init\_\_(self):

self.url = 'https://www.viamobilidade.com.br/'

```

2. \*\*Método `get\_status\_linha`\*\*:

- \*\*Objetivo\*\*: Fazer uma requisição HTTP à página da Viamobilidade e extrair informações sobre o status das linhas de metrô e trem.

- \*\*Processo\*\*:

- Uma requisição GET é feita à URL da Viamobilidade.

- O conteúdo HTML da página é analisado usando o BeautifulSoup.

- O método busca por elementos HTML específicos que contêm as informações das linhas, como número da linha, título, situação e detalhes adicionais.

- Esses dados são organizados em uma lista de listas, onde cada sublista contém informações sobre uma linha específica.

- \*\*Tratamento de Erros\*\*:

- Verifica se a requisição foi bem-sucedida (código de status 200).

- Em caso de falha na requisição ou outro erro durante a análise, uma mensagem de erro é exibida.

```python

def get\_status\_linha(self):

response = requests.get(self.url)

response.encoding = 'utf-8' # Garantir que o encoding está correto

try:

if response.status\_code == 200:

soup = BeautifulSoup(response.text, 'html.parser')

linhas = soup.find\_all('div', class\_='line-wrapper')

data = []

for itens in linhas:

line\_items = itens.find\_all('li')

for line\_item in line\_items:

linha\_numero = line\_item.find('span').text.strip() if line\_item.find('span') else ''

linha\_title = line\_item.find('span')['title'] if line\_item.find('span') and 'title' in line\_item.find('span').attrs else ''

linha\_situacao = line\_item.find('div', class\_='status').text.strip() if line\_item.find('div', class\_='status') else ''

wrapper = line\_item.find('div', class\_='wrapper')

if wrapper:

detalhe = wrapper.find('p').text.strip() if wrapper.find('p') else ''

else:

detalhe = ''

data.append([linha\_numero, linha\_title, linha\_situacao, detalhe])

return data

else:

print(f'Erro ao acessar a página. Status code: {response.status\_code}')

except Exception as e:

print(f"Ocorreu um erro de exceção: {e}")

```

3. \*\*Método `generate\_data\_frame`\*\*:

- \*\*Objetivo\*\*: Converter os dados extraídos em um DataFrame do Pandas e salvar esses dados em um arquivo CSV.

- \*\*Parâmetros\*\*:

- `data`: Uma lista de listas contendo as informações sobre as linhas (número, título, situação e detalhes).

- \*\*Processo\*\*:

- Um DataFrame é criado a partir dos dados.

- O DataFrame é salvo em um arquivo CSV no diretório `files` com o nome `status\_linhas.csv`.

- \*\*Tratamento de Erros\*\*:

- Verifica possíveis erros de mapeamento, como colunas não encontradas no DataFrame.

```python

def generate\_data\_frame(self, data):

# Criar um DataFrame

try:

df = pd.DataFrame(data, columns=['Linha Número', 'Título', 'Situação', 'Detalhe'])

df.to\_csv('files/status\_linhas.csv', index=False, encoding='utf-8')

return df

except KeyError as e:

print(f"Erro de mapeamento: {e}")

```

---

### Resumo

A classe `LinhasStatus` é uma ferramenta útil para coletar e organizar informações sobre o status das linhas de metrô e trem da Viamobilidade. Ela realiza isso ao:

1. Fazer web scraping para obter os dados diretamente do site.

2. Converter os dados em um formato tabular usando Pandas.

3. Exportar os dados para um arquivo CSV para fácil acesso e análise.

A implementação inclui tratamento de erros para garantir a robustez da aplicação em diferentes cenários, como falhas de conexão ou alterações na estrutura da página web.

### Documentação da Classe `EstacoesLista`

A classe `EstacoesLista` é responsável por coletar e organizar informações sobre as estações de metrô e trem a partir do site da Viamobilidade. Ela realiza web scraping para extrair os dados e os organiza em um formato tabular, que é exportado para um arquivo CSV.

#### Importações

- \*\*`pandas`\*\*: Biblioteca para manipulação e análise de dados, usada para criar e manipular DataFrames.

- \*\*`BeautifulSoup`\*\*: Biblioteca do pacote `bs4`, utilizada para analisar e extrair informações de documentos HTML e XML.

- \*\*`requests`\*\*: Biblioteca para fazer requisições HTTP, utilizada para acessar o conteúdo da página da Viamobilidade.

- \*\*`re`\*\*: Módulo para operações de expressões regulares, usado para corrigir tags HTML malformadas.

```python

import pandas as pd

from bs4 import BeautifulSoup

import requests

import re

```

#### Classe `EstacoesLista`

A classe possui três métodos principais:

1. \*\*Método Construtor (`\_\_init\_\_`)\*\*:

- \*\*Objetivo\*\*: Inicializar a classe com a URL da página que contém o mapa das estações da Viamobilidade.

- \*\*Processo\*\*:

- A URL da página é armazenada na variável de instância `self.url`.

```python

class EstacoesLista:

def \_\_init\_\_(self):

self.url = 'https://www.viamobilidade.com.br/voce/mapa'

```

2. \*\*Método `corrigir\_html\_tags`\*\*:

- \*\*Objetivo\*\*: Corrigir o HTML malformado adicionando as tags de fechamento `</li>` onde estão faltando.

- \*\*Processo\*\*:

- Usa expressões regulares para corrigir o HTML, garantindo que todas as tags `<li>` estejam corretamente fechadas com `</li>`.

- \*\*Parâmetros\*\*:

- `html\_content`: O conteúdo HTML que precisa ser corrigido.

- \*\*Retorno\*\*:

- O HTML corrigido.

```python

def corrigir\_html\_tags(self, html\_content):

# Corrigir o HTML malformado adicionando as tags de fechamento </li> onde faltam

corrected\_html\_content = re.sub(r'(?<=</li>)(<li>)', r'\1', html\_content) # Corrige fechamento múltiplo </li>

corrected\_html\_content = re.sub(r'<li>', r'</li><li>', corrected\_html\_content) # Adiciona </li> antes de cada <li> novo

return corrected\_html\_content

```

3. \*\*Método `get\_estacoes`\*\*:

- \*\*Objetivo\*\*: Coletar informações sobre as estações de metrô e trem a partir do site da Viamobilidade.

- \*\*Processo\*\*:

- Faz uma requisição GET para a URL da página.

- Analisa o conteúdo HTML da página com BeautifulSoup.

- Encontra os elementos HTML que contêm informações sobre as linhas e suas estações.

- Corrige possíveis erros no HTML usando o método `corrigir\_html\_tags`.

- Extrai os nomes das estações e organiza essas informações em um dicionário, onde cada chave é o nome da linha e o valor é uma lista das estações dessa linha.

- \*\*Retorno\*\*:

- Um dicionário com as linhas como chaves e listas de estações como valores.

```python

@property

def get\_estacoes(self):

response = requests.get(self.url)

response.encoding = 'utf-8' # Garantir que o encoding está correto

try:

if response.status\_code == 200:

soup = BeautifulSoup(response.text, 'html.parser')

linhas = soup.find\_all('div', class\_='row line-list')

data = {}

for linha in linhas:

colunas = linha.find\_all("div", class\_="col-4")

if colunas:

for coluna in colunas:

nome\_linha\_tag = coluna.find("h4")

if nome\_linha\_tag: # Verifica se o nome da linha foi encontrado

nome\_linha = nome\_linha\_tag.text.strip()

estacoes\_tag = coluna.find("ul")

if estacoes\_tag:

estacoes\_html = str(estacoes\_tag)

estacoes\_corrigidas\_html = self.corrigir\_html\_tags(estacoes\_html)

estacoes\_soup = BeautifulSoup(estacoes\_corrigidas\_html, 'html.parser')

estacoes\_list = estacoes\_soup.find\_all('li')

lista\_estacoes = [estacao.get\_text() for estacao in estacoes\_list]

# Adiciona as estações à linha correspondente

if nome\_linha in data:

data[nome\_linha].extend(lista\_estacoes)

else:

data[nome\_linha] = lista\_estacoes

return data

else:

print('Não foi possível obter a lista de linhas')

except Exception as e:

print(f"Ocorreu um erro de exceção: {e}")

```

4. \*\*Método `gera\_dataframe\_lista\_estacoes`\*\*:

- \*\*Objetivo\*\*: Converter o dicionário de estações em um DataFrame e salvar os dados em um arquivo CSV.

- \*\*Processo\*\*:

- Calcula o comprimento máximo das listas de estações para garantir que todas as listas tenham o mesmo comprimento.

- Ajusta o comprimento das listas de estações para o comprimento máximo, adicionando valores `None` onde necessário.

- Cria um DataFrame a partir do dicionário.

- Salva o DataFrame em um arquivo CSV no diretório `files` com o nome `lista\_estacoes.csv`.

- \*\*Parâmetros\*\*:

- `data`: Um dicionário onde as chaves são os nomes das linhas e os valores são listas de estações.

- \*\*Retorno\*\*:

- O DataFrame criado.

```python

def gera\_dataframe\_lista\_estacoes(self, data):

max\_length = max(len(lst) for lst in data.values())

for key in data:

if len(data[key]) < max\_length:

data[key].extend([None] \* (max\_length - len(data[key])))

df = pd.DataFrame(data, columns=data.keys())

df.to\_csv('files/lista\_estacoes.csv', index=False)

return df

```

---

### Resumo

A classe `EstacoesLista` é uma ferramenta para coletar e organizar informações sobre estações de metrô e trem a partir do site da Viamobilidade. Ela realiza isso ao:

1. Corrigir HTML malformado para garantir a integridade dos dados extraídos.

2. Coletar informações de linhas e suas estações através de web scraping.

3. Organizar esses dados em um formato tabular usando Pandas.

4. Exportar os dados para um arquivo CSV para fácil acesso e análise.

A implementação inclui tratamento de erros e correção de tags HTML para garantir que os dados sejam coletados e apresentados corretamente.

### Documentação do Arquivo `index.html`

O arquivo `index.html` é uma página web projetada para fornecer um formulário de busca para roteiros de transporte público e exibir um mapa interativo com rotas e marcadores baseados nos pontos fornecidos. Este arquivo inclui integração com a API do Google Maps para visualização de rotas e pontos no mapa. Abaixo está uma documentação detalhada sobre o conteúdo e funcionamento deste arquivo.

#### Estrutura do Documento HTML

```html

<!DOCTYPE html>

<html lang="pt-BR">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Roteiro de Viagem - Transporte Público</title>

<link rel="stylesheet" href="static/css/styles.css">

<script src="https://maps.googleapis.com/maps/api/js?key={{api\_key}}&libraries=places&callback=initMap&loading=async" async defer></script>

<style>

#map {

width: 600px;

height: 400px;

}

</style>

</head>

<body>

<div class="container">

<form action="/gerar\_roteiro" method="POST">

<div class="form-field">

<label for="endOrigem">Endereço Origem</label>

<input type="text" name="endOrigem" id="endOrigem" value="{{ endOrigem|safe }}" onfocus="this.value == '{{ endOrigem }}' ? this.value = '' : null;" onblur="this.value == '' ? this.value = 'Endereço Origem' : null;" required>

</div>

<div class="form-field">

<label for="endDestino">Endereço Destino</label>

<input type="text" name="endDestino" id="endDestino" value="{{ endDestino | safe }}" onfocus="this.value == '{{ endDestino }}' ? this.value = '' : null;" onblur="this.value == '' ? this.value = 'Endereço Destino' : null;" required />

</div>

<div>

<button type="submit" class="solid" id="busca-roteiro">Buscar Roteiro</button>

</div>

</form>

<div class="content">

<div id="roteiro">

<h3>Roteiro Gerado:</h3>

{% if roteiro %}

<div>{{ roteiro | safe }}</div>

{% endif %}

</div>

<div id="map"></div>

<script>

// Função para criar linhas de rotas no mapa

function createLines(map, pontos) {

const directionsService = new google.maps.DirectionsService();

for (let i = 0; i < pontos.length - 1; i++) {

const start = pontos[i];

const end = pontos[i + 1];

const request = {

origin: start,

destination: end,

travelMode: google.maps.TravelMode.TRANSIT

};

directionsService.route(request, (response, status) => {

if (status === 'OK') {

const path = response.routes[0].overview\_path;

const line = new google.maps.Polyline({

path: path,

strokeColor: '#1e88e5',

strokeOpacity: 1.0,

strokeWeight: 4

});

line.setMap(map);

}

});

}

}

// Função para inicializar o mapa

async function initMap() {

const mapElement = document.getElementById('map');

if (!mapElement) {

console.error('Elemento com ID "map" não encontrado.');

return;

}

const map = new google.maps.Map(mapElement, {

center: { lat: -23.550520, lng: -46.633308 }, // Centro de São Paulo

zoom: 12

});

try {

const response = await fetch('../files/pontos.json');

const data = await response.json();

const pontos = data;

const firstIcon = {url:'../icon\_map/start.png',

size: new google.maps.Size(32, 32),

scaledSize: new google.maps.Size(24, 24)}; // URL do ícone para o primeiro marcador

const lastIcon = {url:'../icon\_map/end.png',

size: new google.maps.Size(32, 32),

scaledSize: new google.maps.Size(24, 24)}; // URL do ícone para o último marcador

const defaultIcon = {url:'../icon\_map/default.png',

size: new google.maps.Size(32, 32),

scaledSize: new google.maps.Size(24, 24)}; // URL do ícone para os demais marcadores

// Adiciona marcadores ao mapa

for (let i = 0; i < pontos.length; i++) {

const icon = i === 0 ? firstIcon : i === pontos.length - 1 ? lastIcon : defaultIcon;

if (!map) {

console.error('Map is undefined');

return;

}

const marker = new google.maps.Marker({

map: map,

position: { lat: pontos[i].lat, lng: pontos[i].lng },

icon: icon

});

createLines(map, pontos);

// Calcular e ajustar o zoom

const bounds = new google.maps.LatLngBounds();

pontos.forEach(ponto => {

bounds.extend({ lat: ponto.lat, lng: ponto.lng });

});

map.fitBounds(bounds);

}

} catch (error) {

console.error('Erro ao carregar ou processar o arquivo JSON:', error);

}

}

window.onload = initMap;

</script>

</div>

</div>

</body>

</html>

```

#### Descrição dos Componentes

1. \*\*Cabeçalho (`<head>`)\*\*:

- \*\*Meta Tags\*\*:

- `charset="UTF-8"`: Define a codificação de caracteres para UTF-8.

- `viewport`: Configura a visualização responsiva para dispositivos móveis.

- \*\*Título\*\*:

- Define o título da página como "Roteiro de Viagem - Transporte Público".

- \*\*Link para CSS\*\*:

- Inclui um arquivo CSS externo localizado em `static/css/styles.css` para estilização da página.

- \*\*Script do Google Maps\*\*:

- Inclui a API do Google Maps com a chave de API substituída pela variável `{{api\_key}}`. O script carrega a biblioteca `places` e define `initMap` como a função de callback quando o mapa é carregado.

2. \*\*Corpo (`<body>`)\*\*:

- \*\*Div Container\*\*:

- Contém todo o conteúdo da página, incluindo o formulário e a área de conteúdo.

- \*\*Formulário (`<form>`)\*\*:

- Permite ao usuário inserir endereços de origem e destino e enviar uma solicitação para gerar um roteiro.

- Inclui dois campos de entrada de texto para o endereço de origem e destino, ambos com atributos `required`.

- Um botão de envio para buscar o roteiro.

- \*\*Div Content\*\*:

- Contém o conteúdo gerado e o mapa.

- \*\*Div Roteiro (`<div id="roteiro">`)\*\*:

- Exibe o roteiro gerado, se disponível.

- \*\*Div Map (`<div id="map">`)\*\*:

- Área onde o mapa será renderizado.

- \*\*Script\*\*:

- \*\*Função `createLines`\*\*:

- Desenha rotas no mapa entre os pontos fornecidos.

- \*\*Função `initMap`\*\*:

- Inicializa o mapa, adiciona marcadores para os pontos carregados a partir de um arquivo JSON e desenha as linhas entre os pontos.

- Ajusta o zoom do mapa para incluir todos os pontos.

- \*\*Carregamento da Função `initMap`\*\*:

- Define `initMap` para ser executada quando a página é carregada.

#### Observações

- \*\*Placeholder para API Key\*\*: A chave da API do Google Maps deve ser substituída pelo valor real em `{{api\_key}}` durante a renderização da página.

- \*\*Ícones de Marcadores\*\*: Os ícones para os marcadores são especificados com URLs relativas e devem estar localizados no diretório `icon\_map`.

- \*\*Arquivo JSON\*\*: O script espera um arquivo JSON chamado `pontos.json` na pasta `files` com as coordenadas dos pontos para exibição no mapa.

Essa estrutura permite que a página HTML forneça uma interface interativa para os usuários buscarem roteiros de transporte e visualizar essas rotas no mapa do Google Maps.

### Documentação do Arquivo `styles.css`

O arquivo `styles.css` define os estilos visuais para a página HTML do projeto de roteiros de transporte público. Ele é responsável por garantir que a página tenha uma aparência limpa, moderna e responsiva. Abaixo está a documentação detalhada sobre o conteúdo e a finalidade de cada regra de estilo definida no arquivo.

#### Regras de Estilo

```css

/\* Estilização global para o corpo da página \*/

body {

font-family: Arial, sans-serif;

background-color: #f4f4f4;

color: #333;

margin: 0;

padding: 0;

}

/\* Container principal da página \*/

.container {

z-index: 0;

max-width: 1000px;

margin: 50px auto;

padding: 20px;

background-color: #fff;

border-radius: 8px;

box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

}

/\* Estilo para cabeçalhos <h3> \*/

h3 {

color: #1e88e5;

padding-bottom: 5px;

}

/\* Estilo para parágrafos \*/

p {

line-height: 1.6;

}

/\* Estilização dos campos de formulário \*/

.form-field {

margin-bottom: 15px;

}

/\* Estilo para rótulos de formulário \*/

label {

display: block;

margin-bottom: 5px;

font-weight: bold;

color: #555;

}

/\* Estilo para campos de entrada de texto \*/

input[type="text"] {

width: 100%;

padding: 10px;

border: 1px solid #ccc;

border-radius: 4px;

font-size: 16px;

box-sizing: border-box;

}

/\* Estilo para o campo de entrada de texto em foco \*/

input[type="text"]:focus {

border-color: #1e88e5;

outline: none;

}

/\* Estilo para botões \*/

button {

background-color: #1e88e5;

color: #fff;

padding: 10px 20px;

border: none;

border-radius: 4px;

font-size: 16px;

cursor: pointer;

transition: background-color 0.3s;

}

/\* Estilo para o botão ao passar o cursor sobre ele \*/

button:hover {

background-color: #1565c0;

}

/\* Estilo para o botão quando pressionado \*/

button:active {

background-color: #0d47a1;

}

/\* Estilo para a área de conteúdo \*/

.content {

display: flex;

justify-content: space-between;

margin-top: 20px;

}

/\* Estilo para a seção de roteiro \*/

#roteiro {

width: 48%;

}

/\* Estilo para a área do mapa \*/

#map {

z-index: 0;

width: 48%;

height: 400px;

background-color: #e3f2fd;

border-radius: 8px;

}

```

#### Descrição dos Estilos

1. \*\*Estilos Globais (`body`)\*\*:

- Define a fonte padrão como Arial, com uma cor de fundo clara (#f4f4f4) e texto escuro (#333).

- Remove margens e preenchimentos padrão para uma página limpa.

2. \*\*Container (`.container`)\*\*:

- Define a largura máxima do container como 1000px e centraliza-o horizontalmente.

- Aplica um fundo branco, bordas arredondadas, e uma leve sombra para um efeito de elevação.

3. \*\*Cabeçalhos (`h3`)\*\*:

- Define a cor dos cabeçalhos <h3> como azul (#1e88e5) e adiciona um espaçamento inferior.

4. \*\*Parágrafos (`p`)\*\*:

- Ajusta a altura da linha para melhorar a legibilidade.

5. \*\*Campos de Formulário (`.form-field`)\*\*:

- Adiciona uma margem inferior para separar os campos de entrada.

6. \*\*Rótulos de Formulário (`label`)\*\*:

- Exibe os rótulos de forma destacada, com margens e texto em negrito.

7. \*\*Campos de Entrada de Texto (`input[type="text"]`)\*\*:

- Estiliza os campos de entrada com preenchimento, bordas arredondadas, e cor de borda padrão.

- Ajusta a largura para 100% do contêiner e garante que o box-sizing esteja configurado para border-box.

8. \*\*Campo de Entrada de Texto em Foco (`input[type="text"]:focus`)\*\*:

- Altera a cor da borda para azul (#1e88e5) quando o campo está em foco e remove a borda padrão de foco.

9. \*\*Botões (`button`)\*\*:

- Define um fundo azul (#1e88e5) e cor do texto branco.

- Adiciona preenchimento, bordas arredondadas, e um efeito de transição para mudanças na cor do fundo.

10. \*\*Estado Hover e Ativo do Botão (`button:hover` e `button:active`)\*\*:

- Ajusta a cor de fundo do botão quando o usuário passa o cursor sobre ele ou clica nele.

11. \*\*Área de Conteúdo (`.content`)\*\*:

- Exibe o conteúdo em um layout flexível, distribuindo os itens igualmente e adicionando um espaçamento superior.

12. \*\*Seção de Roteiro (`#roteiro`)\*\*:

- Define a largura da seção de roteiro para 48% do contêiner pai.

13. \*\*Área do Mapa (`#map`)\*\*:

- Define a largura e altura do mapa, aplica um fundo azul claro, e arredonda os cantos.

Este arquivo de estilos garante que a interface do usuário seja visualmente atraente e funcional, com um layout responsivo e fácil de usar para interações com o formulário e visualização do mapa.