

Treinamento Associates – Projeto final Swagger

–

Patrick Luiz de Araújo

Conteúdo

APIs – REST e RESTful	03
Conceitos	04
Aplicações	05
 Swagger	 06
 Implementação	 08
Link	09
Serviço a ser oferecido	10
Estrutura do projeto	11
Forma de funcionamento	12

APIs – REST e RESTful

Conceitos

API REST (Representational State Transfer) é um conjunto de boas práticas e padrões para o desenvolvimento de uma API.

Os padrões abarcam:

- **Endpoint:** base url, recursos, queries...
- **Métodos:** Get, Post, Put/Patch, Delete...
- **Headers:** informações enviadas
- **Body:** dados passados como argumento
- **Status Codes:** logs, códigos de erros...
- **Autenticação:** segurança

Aplicações

Base URL: <https://api.minhagastronomia.com>

Recursos: <https://api.minhagastronomia.com/vinhos>

Query: <https://api.minhagastronomia.com/vinhos?pais=brasil®iao=sul>

Método Get:

POST "<https://api.minhagastronomia.com/vinhos?pais=brasil®iao=sul>" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"tipo\": [\"Cabernet Sauvignon\"] }"

Resposta:

```
{"resultado": [ "Vinho Nacional Casa Perini", "Vinho Salton Classic" ], "messages": "Itens encontrados.", "status": "200" }
```

Swagger

Swagger

- Interface padronizada
- Permite o teste dos endpoints sem domínio em programação
- Versionamento de APIs

Para ver mais: <https://petstore.swagger.io/>

Swagger Petstore ^{1.0.6}

[Base URL: petstore.swagger.io/v2]

<https://petstore.swagger.io/v2/swagger.json>

This is a sample server Petstore server. You can find out more about Swagger at <http://swagger.io> or on [#irc.freenode.net](https://irc.freenode.net), [#swagger](https://twitter.com/swagger). For this sample, you can use the api key **special-key** to the authorization filters.

[Terms of service](#)

[Contact the developer](#)

[Apache 2.0](#)

[Find out more about Swagger](#)

Schemes

HTTPS

Authorize

pet Everything about your Pets

Find out more: <http://swagger.io>

POST **/pet/{petId}/uploadImage** uploads an image

POST **/pet** Add a new pet to the store

PUT **/pet** Update an existing pet

GET **/pet/findByStatus** Finds Pets by status

GET **/pet/findByTags** Finds Pets by tags

GET **/pet/{petId}** Find pet by ID

POST **/pet/{petId}** Updates a pet in the store with form data

Implementação

Link



https://github.ibm.com/patrick-ibm/swagger_api

Serviço a ser oferecido

Modelo **VADER** que prevê o **sentimento** de uma frase em inglês por meio de uma **requisição POST**

Para ver mais: <https://github.com/cjhutto/vaderSentiment>

The screenshot shows a REST client interface with a light green background. At the top, there's a header bar with a green button labeled 'POST' and a text label '/main'. Below this is a 'Parameters' section with a red 'Cancel' button in the top right corner. The main area is divided into two columns: 'Name' and 'Description'. Under 'Name', there's a field labeled 'payload' with a red asterisk and the word 'required' next to it, and a sub-label '(body)' below it. Under 'Description', there's a text area containing a JSON object:

```
{  "textoMensagem": [    "fuck you"  ]}
```

. Below the text area is a red 'Cancel' button. Further down, there's a 'Parameter content type' dropdown menu with 'application/json' selected. At the bottom of the configuration section, there are two buttons: a blue 'Execute' button and a 'Clear' button. Below the configuration section is a 'Responses' section with a 'Response content type' dropdown menu set to 'application/json'. At the very bottom, there's a 'Curl' section with a dark background and a text area containing the following curl command:

```
curl -X POST "http://localhost:9000/main" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"textoMensagem\": [ \"fuck you\" ]}"
```

Estrutura do projeto

service: pasta onde o código principal fica

config: arquivo de configurações gerais

constants: salva os códigos de retorno HTTP e o texto das principais respostas

controller: declara os endpoints servidos pela aplicação

responses: código para a construção da resposta

service: código do serviço oferecido

util: códigos diversos

__init__.py: configurações e início do aplicativo

__main__.py: início do aplicativo

app.py: setup da API e inicialização dos endpoints

logging.conf: configuração do logger

restplus.py: configuração da API

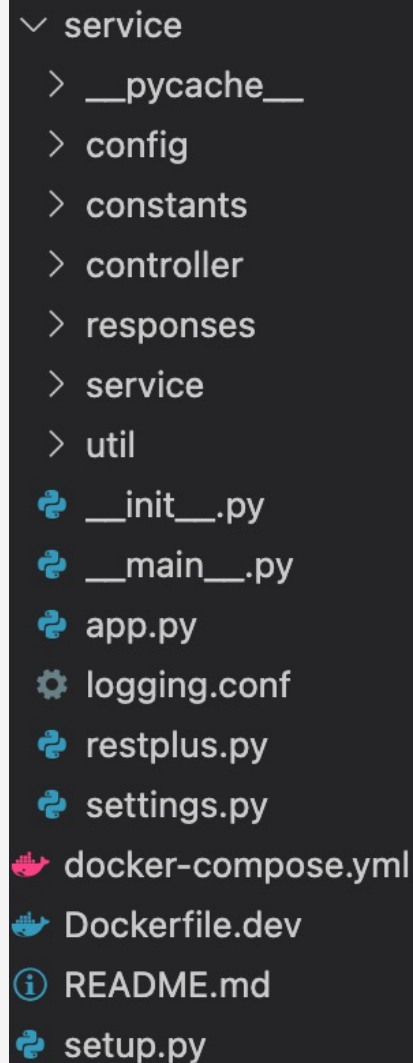
settings.py: variáveis e configurações

docker-compose.yml: setup do container

Dockerfile.dev: códigos do container

README.md: readme

setup.py: pré-requisitos da aplicação



Forma de funcionamento

1. Construção do container e inicialização

O comando *docker-compose build* faz a construção do container com as especificações corretas;

O comando *docker-compose up* sobe o serviço.

```
1 FROM centos/python-38-centos7
2 USER root
3
4 RUN pip install --no-cache-dir --upgrade pip==20.2.3
5 RUN pip install --no-cache-dir --upgrade setuptools==50.3.0
6
7 ENV VERSAO=1.0.1 \
8     CMAKE_C_COMPILER=/usr/bin/gcc \
9     CMAKE_CXX_COMPILER=/usr/bin/g++
10
11 COPY service /code/service
12 COPY setup.py /code/
13 WORKDIR /code
14
15 RUN pip3 install pandas
16 RUN pip3 install -e .[dev]
17
18 COPY Dockerfile.dev ${BOM_PATH}/
19
20 COPY service/config ./config
21
22 RUN chmod -R +x /code
23 You, a week ago • Mudança no padrão
24 CMD ["python3", "-m", "service"]
```

Forma de funcionamento

2. Inicialização do swagger

O código *service/app.py* inicializa os endpoints da aplicação listados em:

- */docs* com a página principal
- *service/controller/start_controller/*
- *service/controller/main_controller/*

```
def configure_app(flask_app):
    flask_app.config['SERVER_NAME'] = settings.FLASK_SERVER_NAME
    flask_app.config['SWAGGER_UI_DOC_EXPANSION'] = settings.RESTPLUS_SWAGGER_UI_DOC_EXPANSION
    flask_app.config['RESTPLUS_VALIDATE'] = settings.RESTPLUS_VALIDATE
    flask_app.config['RESTPLUS_MASK_SWAGGER'] = settings.RESTPLUS_MASK_SWAGGER
    flask_app.config['ERROR_404_HELP'] = settings.RESTPLUS_ERROR_404_HELP

def initialize_app(flask_app):
    configure_app(flask_app)
    CORS(flask_app)

    blueprint = Blueprint('api', __name__, url_prefix=settings.URL_PREFIX)
    api.init_app(blueprint)

    api.add_namespace(operation_basic_start)
    api.add_namespace(operation_basic_padroes)
    flask_app.register_blueprint(blueprint)

def main(app):
    initialize_app(app)
    CORS(app)
    logger.debug("[+] --- starting at: {}:{}".format(settings.FLASK_HOST, settings.FLASK_PORT, settings.URL_PREFIX))
    return app

app = main(app)
```

Forma de funcionamento

3. Ao receber uma requisição em /main

- I. *service/controller/main_controller.py* carrega o endpoint e faz alguns testes de conformidade
- II. O objeto *service.service.main_service* *SentimentosService* carrega o modelo
- III. O parâmetro passado é processado pelo modelo
- IV. A resposta é enviada

```
@pa.route('/main', methods=['POST'])
class MainService(Resource):

    @api.expect(doc_swagger.INPUT_MAIN_SERVICE)
    def post(self) -> dict:
        try:
            dados_request = request.get_json()
            main_service = SentimentosService()
            resp = main_service.executar_rest(dados_request)
            response = objResponse.send_success(data=resp, messages=mensagens.SUCESSO_PREDICT, status=codeHttp.SUCCESS_200)

        except OSError as error:
            response = objResponse.send_exception(objError=error, messages=mensagens.ERROR_OS, status=codeHttp.ERROR_500)
            logger.error(mensagens.ERROR_NONE_TYPE)

        except TypeError as error:
            response = objResponse.send_exception(objError=error, messages=mensagens.ERROR_NONE_TYPE, status=codeHttp.ERROR_500)
            logger.error(mensagens.ERROR_NONE_TYPE)

        except Exception as error:
            response = objResponse.send_exception(objError=error, messages=mensagens.ERROR_GENERIC, status=codeHttp.ERROR_500)
            logger.error(error)

        return response
```

Forma de funcionamento

3. Ao receber uma requisição em /main

- I. `service/controller/main_controller.py` carrega o endpoint e faz alguns testes de conformidade
- II. O objeto `service.service.main_service.SentimentosService` carrega o modelo
- III. O parâmetro passado é processado pelo modelo
- IV. A resposta é enviada

```
class SentimentosService():  
    You, a week ago • Mudança no padrão  
  
    def __init__(self):  
        logger.debug(mensagens.INICIO_LOAD_MODEL)  
        self.load_model()  
  
    def load_model(self):  
        """  
        Carrega o modelo VADER a ser usado  
        """  
  
        self.model = SentimentIntensityAnalyzer()  
  
        logger.debug(mensagens.FIM_LOAD_MODEL)  
  
    def executar_rest(self, texts):  
        response = {}  
  
        logger.debug(mensagens.INICIO_PREDICT)  
        start_time = time.time()  
  
        response_predicts = self.buscar_predicao(texts['textoMensagem'])  
  
        logger.debug(mensagens.FIM_PREDICT)  
        logger.debug(f"Fim de todas as predições em {time.time()-start_time}")  
  
        df_response = pd.DataFrame(texts, columns=['textoMensagem'])  
        df_response['predict'] = response_predicts  
  
        df_response = df_response.drop(columns=['textoMensagem'])  
  
        response = {  
            "listaClassificacoes": json.loads(df_response.to_json(  
                orient='records', force_ascii=False)))  
        }  
  
        return response
```

Forma de funcionamento

3. Ao receber uma requisição em `/main`

- I. `service/controller/main_controller.py` carrega o endpoint e faz alguns testes de conformidade
- II. O objeto `service.service.main_service.SentimentosService` carrega o modelo
- III. O parâmetro passado é processado pelo modelo
- IV. A resposta é enviada

```
def buscar_predicao(self, texts):  
    """  
    Pega o modelo carregado e aplica em texts  
    """  
    logger.debug('Iniciando o predict...')  
  
    response = []  
  
    for text in texts:  
        sentiment_dict = self.model.polarity_scores(text)  
  
        # decide sentiment as positive, negative and neutral  
        if sentiment_dict['compound'] >= 0.05:  
            response.append("Positive")  
  
        elif sentiment_dict['compound'] <= - 0.05:  
            response.append("Negative")  
  
        else:  
            response.append("Neutral")  
  
    return response
```


Forma de funcionamento

1. Construção do container e inicialização

O comando *docker-compose build* faz a construção do container com as especificações corretas;

O comando *docker-compose up* sobe o serviço.

2. Inicialização do swagger

O código *service/app.py* inicializa os endpoints da aplicação listados em:

- */docs* com a página principal
- *service/controller/start_controller: /*
- *service/controller/main_controller: /main*

3. Ao receber uma requisição em */main*

- I. *service/controller/main_controller.py* carrega o endpoint e faz alguns testes de conformidade
- II. O objeto *service.service.main_service SentimentosService* carrega o modelo
- III. O parâmetro passado é processado pelo modelo
- IV. A resposta é enviada