Rest API: Python Flask

Prof. Me. Eng. Hélio Lourenço Esperidião Ferreira

O que é o Flask?

- Framework web minimalista em Python.
- Permite criar aplicações web rapidamente.
- Muito utilizado em APIs, prototipagem rápida e micro serviços.
- Concorrente mais "leve" do Django.

Flask

- Lançado em 2010 e criado por Armin Ronacher, o Flask é um micro-framework voltado para aplicações menores com requisitos simples, como a construção de sites básicos.
- Ele oferece um núcleo leve e expansível, permitindo que a aplicação utilize apenas os recursos essenciais para sua execução.
- Conforme novas necessidades surgem, pacotes adicionais podem ser incorporados para aumentar as funcionalidades.

Características do Flask

- **Simplicidade**: O Flask fornece apenas os recursos essenciais para o desenvolvimento de uma aplicação, o que torna os projetos mais enxutos em comparação com frameworks maiores. Com menos arquivos e uma arquitetura mais simples, o desenvolvimento é facilitado.
- **Desenvolvimento ágil**: No Flask, o foco do desenvolvedor é apenas no que é necessário para o projeto, evitando configurações complexas ou desnecessárias, o que acelera o processo de desenvolvimento.
- **Projetos menores**: Com uma estrutura básica e a possibilidade de iniciar o projeto com um único arquivo, os projetos em Flask tendem a ser mais compactos e leves quando comparados a frameworks mais complexos.

Vantagens do Flask

- **Leve e minimalista**
- Não vem com muitas dependências por padrão, você só adiciona o que precisa.
- **Curva de aprendizado suave**
- Mais fácil de começar do zero comparado a frameworks maiores (como Django).
- **Flexibilidade**
- Você escolhe como organizar o projeto, quais extensões usar e quais bibliotecas integrar.
- **Comunidade e extensões**
- Várias extensões prontas para autenticação, ORM, formulários, etc.
- **Ideal para microserviços e APIs**
- Simples de integrar em arquiteturas modernas.
- **Prototipagem rápida**
- Excelente para criar MVPs, provas de conceito e projetos acadêmicos.

Desvantagens do Flask

Menos estrutura pronta

- Ao contrário do Django, não traz autenticação, administração ou ORM embutido.

Exige mais decisões do desenvolvedor

- Como organizar pastas, qual banco de dados usar, quais libs de segurança integrar etc.

Pode virar um "Frankenstein“

- Projetos grandes podem ficar bagunçados se não houver boas práticas desde o início.

Escalabilidade em grandes equipes

- Times maiores preferem frameworks mais "opinionados" (como Django, FastAPI ou Spring).

Menos pronto para projetos complexos

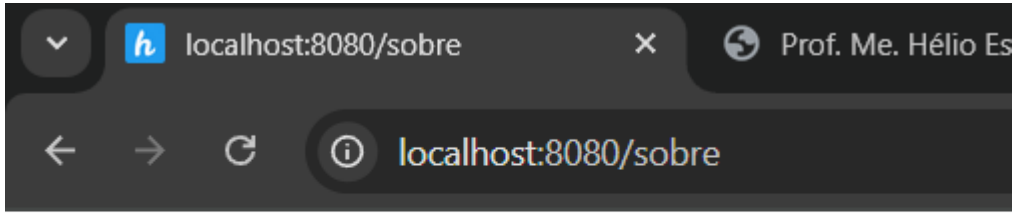
- Precisa integrar manualmente autenticação, segurança avançada e administração.

Preparando o ambiente

- pip install flask

Olá mundo

```python
from flask import Flask


app = Flask(__name__)


@app.route("/")
def home():
    return "Olá, mundo com Flask!"

@app.route("/sobre")
def sobre():
    return "Essa é a página Sobre."

@app.route("/user/<nome>")
def user(nome):
    return f"Olá, {nome.capitalize()}!"

if __name__ == "__main__":
    app.run(debug=True, port=8080)
```



Essa é a página Sobre.

POO

- **POO** é um **paradigma de programação** que organiza o código em **objetos**, ao invés de apenas funções e estruturas de dados.
  - Um **objeto** representa algo do mundo real ou conceitual, com **atributos** (dados) e **métodos** (comportamentos).
  - A ideia é modelar o software de forma **mais próxima da realidade**, promovendo **modularidade, reuso e manutenção mais fácil**.

Nomenclatura de arquivos e classes

- **Arquivos**: usar *snake_case*, tudo minúsculo.
  Ex.: retangulo.py, minha_classe.py
- **Classes**: usar *CamelCase* (também chamado PascalCase).
  Ex.: class Retangulo:, class PessoaFisica:
- **Métodos e funções**: *snake_case*.
  Ex.: def calcular_area():, def get_nome():
- **Constantes**: letras maiúsculas com underscore.
  Ex.: PI = 3.14, TAMANHO_MAX = 100

O que é o construtor em Python?

- Em Python, o **construtor** é o método especial __init__.
  Ele é chamado **automaticamente** sempre que você cria um objeto de uma classe.

```python
class Pessoa2:
    def __init__(self):
        self.__nome = None
        self.__idade = None


    # Getter e setter de nome
    @property
    def nome(self):
        return self.__nome


    @nome.setter
    def nome(self, value):
        self.__nome = value


    # Getter e setter de idade
    @property
    def idade(self):
        return self.__idade


    @idade.setter
    def idade(self, value):
        self.__idade = value
```

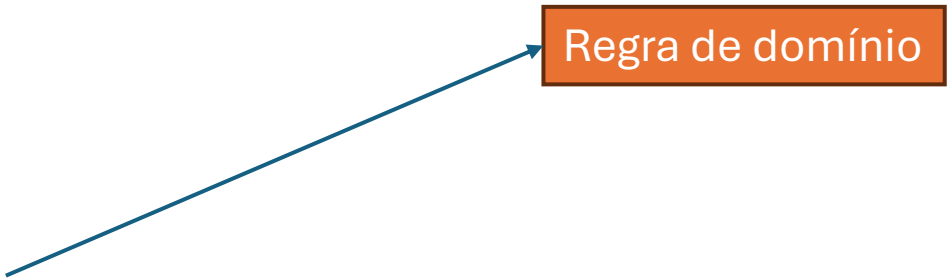
```python
p = Pessoa("Ana", 25)
print(p.nome)   # Ana
print(p.idade)  # 25


p.nome = "João"
p.idade = 30
print(p.nome)   # João
print(p.idade)  # 30
```

```python
class Pessoa:
    def __init__(self, nome, idade):
        self.__nome = nome
        self.__idade = idade

    # Getter e setter de nome
    @property
    def nome(self):
        return self.__nome

    @nome.setter
    def nome(self, value):
        self.__nome = value


    # Getter e setter de idade
    @property
    def idade(self):
        return self.__idade


    @idade.setter
    def idade(self, value):
        self.__idade = value
```

```python
p = Pessoa("Ana", 25)
print(p.nome)   # Ana
print(p.idade)  # 25


p.nome = "João"
p.idade = 30
print(p.nome)   # João
print(p.idade)  # 30
```

Classes

```python
import math
class Retangulo:
    def __init__(self):
        self.__base = None
        self.__altura = None
    @property
    def base(self):
        return self.__base
    @base.setter
    def base(self, value):
        if value <= 0:
            raise ValueError("A base deve ser positiva")
        self.__base = value
    @property
    def altura(self):
        return self.__altura
    @altura.setter
    def altura(self, value):
        if value <= 0:
            raise ValueError("A altura deve ser positiva")
        self.__altura = value
    def calcular_area(self):
        if self.base is None or self.altura is None:
            raise ValueError("Base e altura precisam estar definidas para calcular a área")
        return self.base * self.altura
    def calcular_perimetro(self):
        if self.base is None or self.altura is None:
            raise ValueError("Base e altura precisam estar definidas para calcular o perímetro")
        return 2 * (self.base + self.altura)
    def calcular_diagonal(self):
        if self.base is None or self.altura is None:
            raise ValueError("Base e altura precisam estar definidas para calcular a diagonal")
        return math.sqrt(self.base**2 + self.altura**2)
```



Exemplo Cargo

- O Model em MVC representa os dados da aplicação e também pode conter regras de domínio.
- Regras de domínio são as regras que definem como os dados podem existir ou interagir, independente de banco de dados ou interface.
- O Model não deve se preocupar com persistência nem com apresentação.

Modelo /cargo.py

```python
class Cargo:
    def __init__(self):
        self.__idCargo = None
        self.__nomeCargo = None
    @property
    def idCargo(self):
        return self.__idCargo
    @idCargo.setter
    def idCargo(self, value):
        try:
            parsed = int(value)
        except (ValueError, TypeError):
            raise ValueError("idCargo deve ser um número inteiro.")
        if parsed <= 0:
            raise ValueError("idCargo deve ser maior que zero.")
        self.__idCargo = parsed
    @property
    def nomeCargo(self):
        return self.__nomeCargo
    @nomeCargo.setter
    def nomeCargo(self, value):
        if not isinstance(value, str):
            raise ValueError("nomeCargo deve ser uma string.")
        nome = value.strip()
        if len(nome) < 3:
            raise ValueError("nomeCargo deve ter pelo menos 3 caracteres.")
        self.__nomeCargo = nome
```

DAO

- DAO é um padrão de projeto cuja função principal é isolar o acesso ao banco de dados da lógica de negócio da aplicação.
- Em outras palavras:
  - Ele atua como uma camada de abstração entre seu Model e o banco de dados.
  - Isso significa que o restante da aplicação não precisa saber como os dados são armazenados (SQL, NoSQL, arquivos etc.).

Responsabilidades do DAO

- Um DAO típico faz:
  - CRUD (Create, Read, Update, Delete) de um Model:
  - create(obj) → insere o objeto no banco.
  - read(id) → retorna um objeto a partir do banco.
  - update(obj) → atualiza dados existentes.
  - delete(id) → remove do banco.

dao
/cargoDAO.py

Resumindo o bloco `with`

1)Abre a conexão com o banco (conn).
2)Cria um cursor (cursor) para executar comandos SQL.
3)Executa a query de inserção, passando parâmetros seguros.
4)Faz o commit para salvar no banco.
5)Recupera o ID do registro inserido.
6)Fecha automaticamente o cursor e a conexão ao sair dos blocos with.

```python
from api.model.cargo import Cargo
class CargoDAO:
    def __init__(self, database_dependency):
        print("⬆️  CargoDAO.__init__()")
        self.__database = database_dependency

    def create(self, objCargo: Cargo) -> int:
        SQL = "INSERT INTO cargo (nomeCargo) VALUES (%s);"
        params = (objCargo.nomeCargo,)

        with self.__database.get_connection() as conn:
            with conn.cursor() as cursor:
                cursor.execute(SQL, params)
                conn.commit()
                insert_id = cursor.lastrowid

        if not insert_id:
            raise Exception("Falha ao inserir cargo")
        print("✅ CargoDAO.create()")
        return insert_id
```

dao
/cargoDAO.py

```
    def delete(self, cargo: Cargo) -> bool:
        SQL = "DELETE FROM cargo WHERE idCargo = %s;"
        params = (cargo.idCargo,)
        with self.__database.get_connection() as conn:
            with conn.cursor() as cursor:
                cursor.execute(SQL, params)
                conn.commit()
                affected = cursor.rowcount
        print("✅ CargoDAO.delete()")
        return affected > 0

    def update(self, objCargo: Cargo) -> bool:
        SQL = "UPDATE cargo SET nomeCargo = %s WHERE idCargo = %s;"
        params = (objCargo.nomeCargo, objCargo.idCargo)
        with self.__database.get_connection() as conn:
            with conn.cursor() as cursor:
                cursor.execute(SQL, params)
                conn.commit()
                affected = cursor.rowcount


        print("✅ CargoDAO.update()")
        return affected > 0
```

dao
/cargoDAO.py

```python
def findAll(self) -> list[dict]:
        SQL = "SELECT * FROM cargo;"
        with self.__database.get_connection() as conn:
            with conn.cursor(dictionary=True) as cursor:
                cursor.execute(SQL)
                resultados = cursor.fetchall()
        print(f"✅ CargoDAO.findAll() -> {len(resultados)} registros encontrados")
        return resultados

    def findById(self, idCargo: int) -> dict | None:
        resultados = self.findByField("idCargo", idCargo)
        print("✅ CargoDAO.findById()")
        return resultados[0] if resultados else None

    def findByField(self, field: str, value) -> list[dict]:
        allowed_fields = ["idCargo", "nomeCargo"]
        if field not in allowed_fields:
            raise ValueError(f"Campo inválido para busca: {field}")

        SQL = f"SELECT * FROM cargo WHERE {field} = %s;"
        params = (value,)

        with self.__database.get_connection() as conn:
            with conn.cursor(dictionary=True) as cursor:
                cursor.execute(SQL, params)
                resultados = cursor.fetchall()
        print("✅ CargoDAO.findByField()")
        return resultados
```

Service

- Um Service é uma camada intermediária entre o Controller e o DAO/Model.
  - Sua função principal é: Encapsular regras de negócio complexas que não cabem apenas no Model.
  - Orquestrar operações envolvendo múltiplos DAOs ou Models.
  - Garantir consistência antes de persistir ou retornar dados.

- **Por que usar Service?**
  - Evita que o **Controller fique cheio de lógica**.
  - Evita que o **DAO fique com regras de negócio**.
  - Deixa o código mais modular e fácil de testar.

ps.

- O Service não conversa diretamente com a camada de apresentação (View).
- Ele responde apenas ao Controller, que é o responsável por receber o input do usuário e decidir o que mostrar na interface.
- O Service aplica regras de negócio e orquestra DAOs.

dao
/cargoService.py

```python
# -*- coding: utf-8 -*-
from api.dao.cargo_dao import CargoDAO
from api.model.cargo import Cargo
from api.http.error_response import ErrorResponse

class CargoService:

    def __init__(self, cargo_dao_dependency: CargoDAO):
        print("⬆️  CargoService.__init__()")
        self.__cargoDAO = cargo_dao_dependency  # injeção de dependência

    def createCargo(self, cargoBodyRequest: dict) -> int:
        print("🟣 CargoService.createCargo()")
        cargo = Cargo()
        cargo.nomeCargo = cargoBodyRequest.get("nomeCargo")
        # valida regra de negócio: cargo duplicado
        resultado = self.__cargoDAO.findByField("nomeCargo", cargo.nomeCargo)
        if resultado and len(resultado) > 0:
            raise ErrorResponse(
                400,
                "Cargo já existe",
                {"message": f"O cargo {cargo.nomeCargo} já existe"}
            )
        return  self.__cargoDAO.create(cargo)
```

dao
/cargoService.py

```
    def findAll(self) -> list[dict]:
        print("🔵 CargoService.findAll()")
        return self.__cargoDAO.findAll()

    def findById(self, idCargo: int) -> dict | None:
        print("🔵 CargoService.findById()")
        cargo = Cargo()
        cargo.idCargo = idCargo  # passa pela validação de domínio
        return self.__cargoDAO.findById(cargo.idCargo)

    def updateCargo(self, idCargo: int, jsonCargo: dict) -> bool:
        print("🔵 CargoService.updateCargo()")
        cargo = Cargo()
        cargo.idCargo = idCargo
        cargo.nomeCargo = jsonCargo.get("nomeCargo")
        return self.__cargoDAO.update(cargo)

    def deleteCargo(self, idCargo: int) -> bool:
        print("🔵 CargoService.deleteCargo()")
        cargo = Cargo()
        cargo.idCargo = idCargo  # validação de regra de domínio
        return self.__cargoDAO.delete(cargo)
```

Controle

- O **Controller** é a **camada intermediária entre a View e o Service**.
  Ele é responsável por:
- **Receber input do usuário**
  - requisição HTTP (GET, POST, PUT, DELETE)
- **Interpretar a ação do usuário**
  - Saber qual operação deve ser feita, como criar, atualizar, ou buscar dados.
- **Chamar o Service**
  - Envia os dados recebidos para as regras de negócio (Service)
- **Receber o resultado**
  - O Controller pega o retorno do Service prepara a resposta.
- **Enviar resposta para a View**
  - Traduz o resultado da operação para a interface do usuário: mensagem (JSON).

```python
from flask import request, jsonify
from api.service.cargo_service import CargoService

class CargoControl:
    def __init__(self, cargo_service:CargoService):
        print("⬆️  CargoControl.constructor()")
        self.__cargo_service = cargo_service

    def store(self):
        print("🔵 CargoControle.store()")
        cargo_body_request = request.json.get("cargo")
        novo_id = self.__cargo_service.createCargo(cargo_body_request)
        obj_resposta = {
            "success": True,
            "message": "Cadastro realizado com sucesso",
            "data": {
                "cargos": [
                    {
                        "idCargo": novo_id,
                        "nomeCargo": cargo_body_request.get("nomeCargo")
                    }
                ]
            }
        }

        if novo_id:
            return jsonify(obj_resposta), 200
```

```python
from flask import request, jsonify
from api.service.cargo_service import CargoService

class CargoControl:
    def index(self):
        print("🔵 CargoControle.index()")

        array_cargos = self.__cargo_service.findAll()

        return jsonify({
            "success": True,
            "message": "Busca realizada com sucesso",
            "data": {"cargos": array_cargos}
        }), 200

    def show(self):
          # Pega o idCargo diretamente da URI
        idCargo = request.view_args.get("idCargo")


        cargo = self.__cargo_service.findById(idCargo)
        obj_resposta = {
            "success": True,
            "message": "Executado com sucesso",
            "data": {"cargos": cargo}
        }
        return jsonify(obj_resposta), 200
```

```python
from flask import request, jsonify
from api.service.cargo_service import CargoService

class CargoControl:
    def update(self):
        print("🔵 CargoControle.update()")
        # Pega o idCargo diretamente da URI
        idCargo = request.view_args.get("idCargo")
        # Pega os dados do cargo no corpo da requisição
        json_cargo = request.json.get("cargo")
        resposta = self.__cargo_service.updateCargo(idCargo, json_cargo)
        return jsonify({
            "success": True,
            "message": "Cargo atualizado com sucesso",
            "data": {
                "cargo": {
                    "idCargo": int(idCargo),
                    "nomeCargo": json_cargo.get("nomeCargo")
                }
            }
        }), 200
```

```python
from flask import request, jsonify
from api.service.cargo_service import CargoService

class CargoControl:
    def destroy(self):
        print("🔵 CargoControle.destroy()")
        # Pega o idCargo diretamente da URI
        idCargo = request.view_args.get("idCargo")
        try:
            excluiu = self.__cargo_service.deleteCargo(idCargo)
            if not excluiu:
                return jsonify({
                    "success": False,
                    "message": f"Não existe Cargo com id {idCargo}"
                }), 404


            return jsonify({
                "success": True,
                "message": "Excluído com sucesso"
            }), 204
        except Exception as error:
            return jsonify({"success": False, "message": str(error)}), 500
```

Roteador

- Um roteador (router) é a camada ou componente que direciona requisições de entrada para o Controller correto, com base na URL, método HTTP, ou outro tipo de identificador.
- Ele não processa dados nem aplica regras de negócio.
- Ele apenas determina para qual Controller ou função a requisição deve ir

Roteador

- **Função principal**
- Receber a **requisição do usuário** (HTTP).
- Analisar a **rota ou caminho** da requisição.
- **Encaminhar** a requisição para o **Controller correspondente**.
- Retornar a resposta do Controller para o usuário.

```python
# -*- coding: utf-8 -*-
from flask import Blueprint, request
from api.middleware.jwt_middleware import JwtMiddleware
from api.middleware.cargo_middleware import CargoMiddleware
from api.control.cargo_control import CargoControl

class CargoRoteador:
    def __init__(self, jwt_middleware: JwtMiddleware, cargo_middleware: CargoMiddleware, cargo_control:
CargoControl):
        print("⬆️  CargoRoteador.__init__()")
        self.__jwt_middleware = jwt_middleware
        self.__cargo_middleware = cargo_middleware
        self.__cargo_control = cargo_control
        # Blueprint é a coleção de rotas da entidade Cargo
        self.__blueprint = Blueprint('cargo', __name__)
```

```python
def create_routes(self):

    # POST / -> cria um cargo
    @self.__blueprint.route('/', methods=['POST'])
    @self.__jwt_middleware.validate_token  # valida token JWT antes de executar
    @self.__cargo_middleware.validate_body  # valida corpo da requisição
    def store():
        return self.__cargo_control.store()
    # GET / -> lista todos os cargos
    @self.__blueprint.route('/', methods=['GET'])
    @self.__jwt_middleware.validate_token  # valida token JWT
    def index():
        return self.__cargo_control.index()
    # GET /<idCargo> -> retorna um cargo específico
    @self.__blueprint.route('/<int:idCargo>', methods=['GET'])
    @self.__jwt_middleware.validate_token
    @self.__cargo_middleware.validate_id_param  # valida se o ID é válido
```

```
        def show(idCargo):
            return self.__cargo_control.show(idCargo)
        # PUT /<idCargo> -> atualiza um cargo
        @self.__blueprint.route('/<int:idCargo>', methods=['PUT'])
        @self.__jwt_middleware.validate_token
        @self.__cargo_middleware.validate_id_param
        @self.__cargo_middleware.validate_body
        def update(idCargo):
            return self.__cargo_control.update()


        # DELETE /<idCargo> -> remove um cargo
        @self.__blueprint.route('/<int:idCargo>', methods=['DELETE'])
        @self.__jwt_middleware.validate_token
        @self.__cargo_middleware.validate_id_param
        def destroy(idCargo):
            return self.__cargo_control.destroy()
        # Retorna o Blueprint configurado para registro na aplicação Flask
        return self.__blueprint
```

Middleware

- Um **Middleware** é um **componente intermediário** que **processa requisições ou respostas** antes que elas cheguem ao Controller ou antes de serem enviadas de volta ao usuário.
- Ele **não é a lógica principal da aplicação**.
- Atua como **camada transversal** que pode adicionar funcionalidades ou controlar o fluxo.

Funções típicas de um Middleware

- **Validações simples de estruturas de dados recebidos**
  - Validar estrutura json
- **Autenticação e autorização**
  - Verifica se o usuário está logado ou tem permissão para acessar determinada rota.
- **Logging / Monitoramento**
  - Registra requisições, tempo de execução, erros.
- **Tratamento de erros**
  - Intercepta exceções e retorna mensagens amigáveis ao usuário.
- **Transformação de dados**
  - Pode modificar dados da requisição antes de chegar ao Controller.

```python
from functools import wraps
from flask import request
from api.http.error_response import ErrorResponse
class CargoMiddleware:
    def validate_body(self, f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            print("🔷 CargoMiddleware.validate_body()")
            body = request.get_json()
            if not body or 'cargo' not in body:
                raise ErrorResponse(
                    400, "Erro na validação de dados",
                    {"message": "O campo 'cargo' é obrigatório!"}
                )
            cargo = body['cargo']
            if 'nomeCargo' not in cargo:
                raise ErrorResponse(
                    400, "Erro na validação de dados",
                    {"message": "O campo 'nomeCargo' é obrigatório!"}
                )
            return f(*args, **kwargs)
        return decorated_function
```

Permite chamar de forma elegante o middleware no roteador

```python
    def validate_id_param(self, f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            print("🔷 CargoMiddleware.validate_id_param()")
            if 'idCargo' not in kwargs:
                raise ErrorResponse(
                    400, "Erro na validação de dados",
                    {"message": "O parâmetro 'idCargo' é obrigatório!"}
                )
            return f(*args, **kwargs)
        return decorated_function
```

Roteador

- Um **roteador** é o componente responsável por **direcionar requisições do usuário** para a **função ou Controller correto** que vai processar aquela requisição.
  - Ele **analisa a URL, o método HTTP (GET, POST, PUT, DELETE)** e decide qual função deve ser chamada.
  - O roteador **não processa dados** nem aplica regras de negócio.
  - Em aplicações web, ele é essencial para criar **rotas REST ou endpoints de API**.

Funções principais

- **Mapear URLs para funções**
  - Ex.: /cargo/ → CargoController.index()
  - Ex.: /cargo/<id> → CargoController.show(id)
- **Determinar método HTTP**
  - GET, POST, PUT, DELETE, PATCH, etc.
- **Aplicar middlewares** (opcional)
  - Antes de chegar ao Controller, algumas verificações podem ser aplicadas: autenticação, validação, logging, etc.
- **Passar parâmetros da rota**
  - Parâmetros na URL (/<int:idCargo>) são extraídos e repassados para a função.

```python
# -*- coding: utf-8 -*-
from flask import Blueprint, request
from api.middleware.jwt_middleware import JwtMiddleware
from api.middleware.cargo_middleware import CargoMiddleware
from api.control.cargo_control import CargoControl
class CargoRoteador:

    def __init__(self, jwt_middleware: JwtMiddleware, cargo_middleware: CargoMiddleware, cargo_control: CargoControl):
        print("⬆️  CargoRoteador.__init__()")
        self.__jwt_middleware = jwt_middleware
        self.__cargo_middleware = cargo_middleware
        self.__cargo_control = cargo_control
        self.__blueprint = Blueprint('cargo', __name__)
```

```
def create_routes(self):
@self.__blueprint.route('/', methods=['POST'])
@self.__jwt_middleware.validate_token
@self.__cargo_middleware.validate_body
def store():
    return self.__cargo_control.store()

@self.__blueprint.route('/', methods=['GET'])
@self.__jwt_middleware.validate_token
def index():
    return self.__cargo_control.index()
@self.__blueprint.route('/<int:idCargo>', methods=['GET'])
@self.__jwt_middleware.validate_token
@self.__cargo_middleware.validate_id_param
def show(idCargo):
    return self.__cargo_control.show(idCargo)
```

```
@self.__blueprint.route('/<int:idCargo>', methods=['PUT'])
        @self.__jwt_middleware.validate_token
        @self.__cargo_middleware.validate_id_param
        @self.__cargo_middleware.validate_body
        def update(idCargo):
            return self.__cargo_control.update()
        @self.__blueprint.route('/<int:idCargo>', methods=['DELETE'])
        @self.__jwt_middleware.validate_token
        @self.__cargo_middleware.validate_id_param
        def destroy(idCargo):
            return self.__cargo_control.destroy()


        return self.__blueprint
```