



AULA 03

FASTAPI

PYDANTIC E VALIDAÇÃO DE DADOS

O QUE VEREMOS HOJE

- 01 MÉTODO POST
- 02 REQUEST BODY
- 03 PYDANTIC
- 04 VALIDAÇÃO DE DADOS
- 05 TRATAMENTO DE ERROS

MÉTODO POST

O método POST é utilizado para enviar dados para o servidor. Esses dados são geralmente enviados em formato JSON ou XML.

É o método HTTP mais comum para enviar dados, como formulários de contato, informações de registro ou outros dados que precisam ser processados pelo servidor.


O método POST envia os dados no corpo da requisição HTTP, ao invés da URL.



MÉTODO POST NO FASTAPI

No FastAPI utilizamos o `@app.post` para definir uma rota que responde a requisições do tipo POST.

E na função, definimos os parâmetros que serão recebidos no corpo da requisição. O FastAPI mapeia os dados enviados no corpo da requisição diretamente para os parâmetros da função.



```
1  from fastapi import FastAPI
2
3  app = FastAPI()
4
5  @app.post("/user/")
6  def create_user(nome: str, email: str, idade: int):
7      return {
8          "nome": nome,
9          "email": email,
10         "idade": idade
11     }
12
```

REQUEST BODY

O request body é a parte de uma requisição HTTP que contém os dados que você deseja enviar ao servidor.

Esses dados podem ser em formato JSON, XML, texto simples ou outros formatos, dependendo da API que você está utilizando.

O request body é geralmente usado com métodos HTTP como POST e PUT, que são usados para enviar dados ao servidor para criar, atualizar ou modificar recursos.

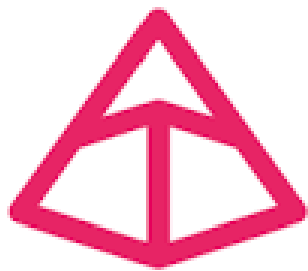


PYDANTIC

O Pydantic é uma biblioteca Python que fornece validação de dados e tratamento de erros.

Ele define modelos de dados usando anotações de tipo Python, que permitem que você defina a estrutura e o tipo dos dados esperados.

O Pydantic fornece validação automática de dados, o que significa que ele verifica se os dados fornecidos correspondem ao modelo de dados definido.

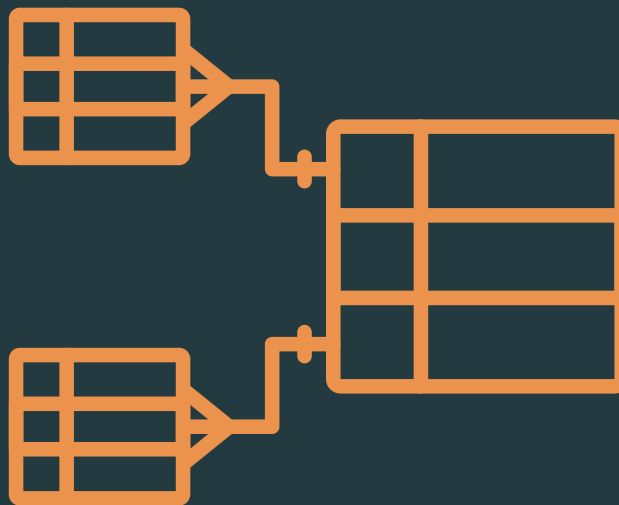


Pydantic

MODELO DE DADOS

Modelos de dados são estruturas que definem a forma e o tipo de dados que um programa espera receber ou gerar. Eles são um molde para garantir que seus dados sejam consistentes e confiáveis.

Isso permite a validação dos dados recebidos com base no modelo de dados, ajudando a prevenir erros e a garantir a integridade dos dados.



COMO DEFINIR MODELOS COM PYDANTIC

Para criar um modelo Pydantic, é preciso definir uma **classe Python que herda de BaseModel**. Dentro da classe, defina os atributos que você deseja validar como campos da classe. Cada campo pode ter um tipo de dado específico, como int, str, float, bool, ou até mesmo outros modelos Pydantic.



```
1  from fastapi import FastAPI
2  from pydantic import BaseModel, EmailStr
3  from typing import Optional
4
5  class User(BaseModel):
6      name: str
7      email: EmailStr
8      age: Optional[int] = None
```


VALIDAÇÃO DE DADOS DE REQUEST BODY

Ao definir o modelo Pydantic com os campos e tipos de dados corretos, você garante que os dados recebidos no request body estejam na estrutura e formato esperados.

Isso permite que o FastAPI valide automaticamente os dados recebidos e gere erros informativos caso haja algum problema.

```
1 from fastapi import FastAPI
2 from pydantic import BaseModel, EmailStr
3
4 class User(BaseModel):
5     name: str
6     email: EmailStr
7     age: int
8
9 app = FastAPI()
10
11 @app.post('/create-user')
12 def create_user(user: User):
13     return {
14         'message': f'User with {user.email} has been created successfully',
15     }
```

VALIDATOR

Para os casos em que seja necessário adicionar validações personalizadas, é possível usar o validator.

O validator é uma funcionalidade do Pydantic usada para adicionar validações personalizadas em campos de um modelo de dados. Ele permite criar regras de validação adicionais além das tipagens definidas nos campos.

```
1 class User(BaseModel):
2     name: str
3     email: EmailStr
4     age: Optional[int] = None
5
6     @field_validator('name')
7     def check_name(cls, value):
8         if len(value) < 3:
9             raise ValueError('Name must be at least 3 characters')
10        return value
11
12    @field_validator('age')
13    def check_age(cls, value):
14        if value is not None and value < 18:
15            raise ValueError('User must be at least 18 years old')
16        return value
```

ATIVIDADE PRÁTICA

Crie um endpoint que receba as informações de um produto e adicione a uma lista de produtos validando as seguintes regras:

1. **Nome do produto:** Deve ser uma string e não pode ter menos de 3 caracteres.
2. **Preço:** O valor deve ser do tipo decimal e não pode ser negativo.
3. **Estoque:** O estoque deve ser um número positivo (maior que zero).

O endpoint deve retornar uma mensagem indicando se os dados são válidos ou não.

- Caso os dados sejam inválidos, retorne uma mensagem clara indicando o erro.
- Caso as informações sejam válidas, adicione o produto à lista e retorne uma mensagem de sucesso.

ATIVIDADE PRÁTICA

Crie uma lista de produtos que contenha alguns produtos e crie um endpoint que receba as informações de um pedido e valide as seguintes regras:

1. **Nome do produto:** Tem que ser o nome de um produto que contenha na lista de produtos.
2. **Preço unitário:** deve ser do tipo decimal e não pode ser negativo
3. **Quantidade solicitada:** deve ser maior que 0.

O endpoint deve incluir o pedido em uma lista de pedidos e **retornar a mensagem** que o pedido foi recebido e o **valor total do pedido**.

TRATAMENTO DE ERROS (HTTPException)

`HTTPException` é uma classe especial em FastAPI usada para gerar respostas de erro com status HTTP apropriados. É um tipo de exceção que, quando lançada, é interceptada pelo FastAPI e retorna uma resposta HTTP ao cliente, sinalizando um erro.

É possível usar `HTTPException` para lidar com erros de validação de dados, erros de banco de dados, erros de autorização e outros erros que ocorrem em sua API.



TRATAMENTO DE ERROS (HTTPException)

Para usar o HTTPException, é necessário importar a classe.

A resposta de erro deve conter um código de status HTTP apropriado, como 400 (Bad Request) ou 422 (Unprocessable Entity), e uma mensagem de erro descritiva que ajude o cliente a entender o que deu errado. Por fim, você deve retornar essa resposta de erro para o cliente, garantindo que ele seja informado sobre o problema e possa corrigir seus dados.

```
1 from fastapi import FastAPI, HTTPException
2 from pydantic import BaseModel, EmailStr, field_validator
3 from typing import Optional
4 from data import users
5
6 class User(BaseModel):
7     name: str
8     email: EmailStr
9     age: Optional[int] = None
10
11     @field_validator('name')
12     def check_name(cls, value):
13         if len(value) < 3:
14             raise HTTPException(status_code=400, detail='Name must have at least 3 characters')
15         return value
16
17     @field_validator('age')
18     def check_age(cls, value):
19         if value is not None and value < 18:
20             raise HTTPException(status_code=400, detail='Age must be greater than or equal to 18')
21         return value
22
23
```

ATIVIDADE PRÁTICA

No código do exercício anterior, adicione o tratamento de erro, adicionando o código de status e a mensagem detalhada e informativa do erro.