

.

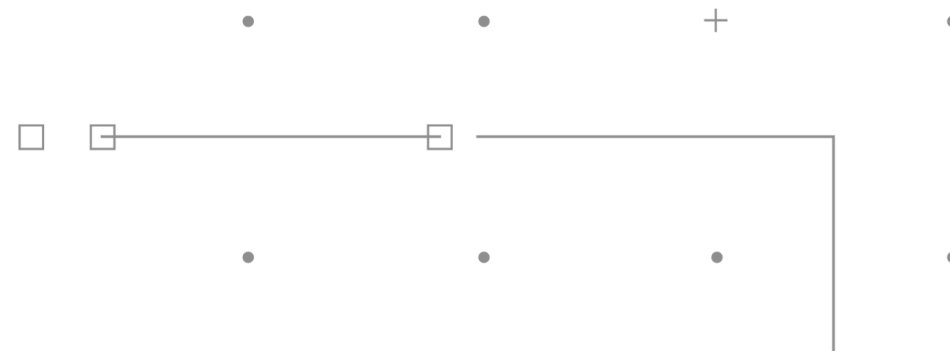
F

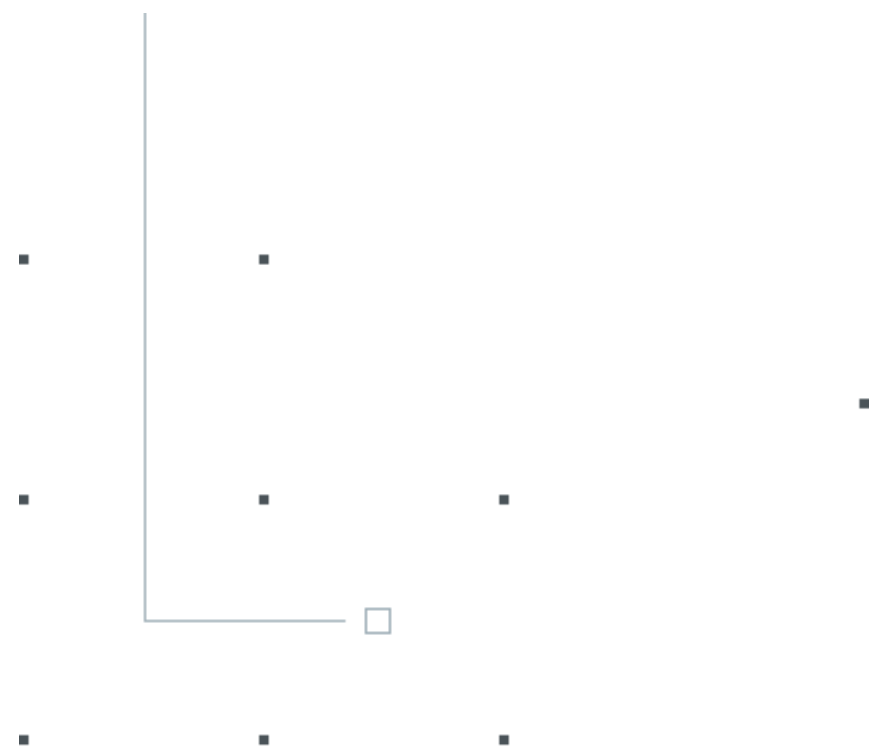
-

I

A

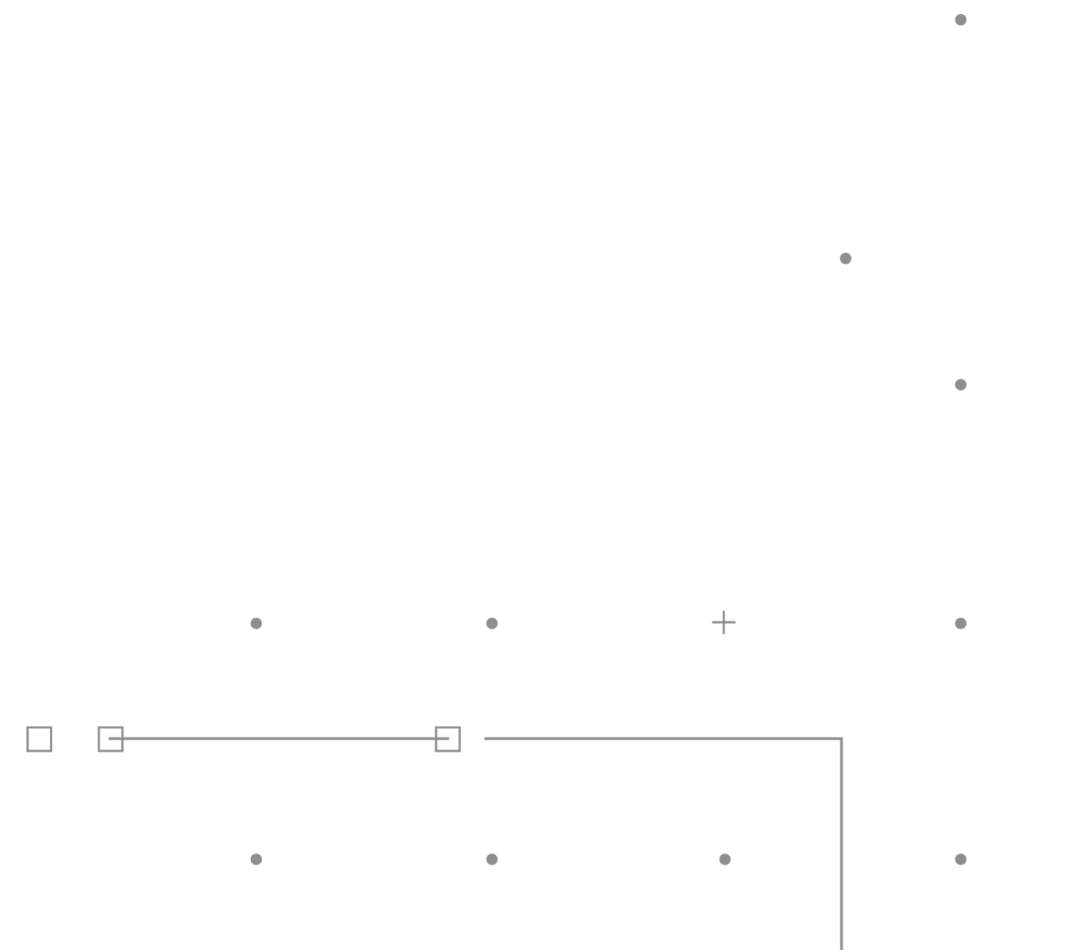
P







FastAPI

Parte II





Introdução



Sua aplicação precisa ser robusta e estável de tal forma que sua aplicação fique rodando 24/7. Muitas aplicações tem bem definida uma estrutura de backend e de frontend. Ao utilizar o FastAPI, criamos uma estrutura de backend onde não compromete o frontend e vice-versa. Sua aplicação vai precisar armazenar e manipular dados. E quando precisamos desse tipo de recurso, utilizamos banco de dados, seja ele SQL ou NoSQL, vai depender muito da sua necessidade.



SQLAlchemy + Pydantic

O **Pydantic** é uma biblioteca que faz validação dos tipos de dados que estão sendo manipulados. Não confundir a validação de modelos (banco de dados) com validação de schemas (**Pydantic**). Se você notar, quando instalamos o FastAPI ele já instala essa biblioteca junto no ambiente. Se quiser instalar manualmente, basta rodar no ambiente:

```
pip install pydantic
```

SQLAlchemy + Pydantic

Agora iremos definir nossos schemas, para serem utilizados durante o recebimento de uma requisição

```
from typing import Union
from pydantic import BaseModel

class ItemBase(BaseModel):
    title: str
    description: Union[str, None] = None

class ItemCreate(ItemBase):
    pass

class Item(ItemBase):
    id: int
    owner_id: int

    class Config:
        orm_mode = True
```

SQLAlchemy + Pydantic

```
class UserBase(BaseModel):  
    email: str  
  
class UserCreate(UserBase):  
    password: str  
  
class User(UserBase):  
    id: int  
    is_active: bool  
    items: list[Item] = []  
  
class Config:  
    orm_mode = True
```

SQLAlchemy + Pydantic

Importante: Não confunda a sintaxe utilizada na definição dos modelos com a sintaxe utilizada na definição dos schemas.

Como se pode notar, nos models um atributo é definido da seguinte forma:

coluna = Column(String)

Enquanto no schema, fica:

atributo: str

E por que isso? O **SQLAlchemy** existe bem antes do recurso de anotação de tipos do Python.

Então por esse motivo ele é declarado de outra forma

SQLAlchemy + Pydantic

Um outro fator importante de se ressaltar, é que em alguns schemas você vai notar que tem a linha:

```
orm_mode = True
```

Quando se configura essa opção, o Pydantic vai tentar ler dados mesmo se não for um dicionário de dados. Ou seja, na sua abstração ele lê um dicionário de dados e pega um valor dele da seguinte forma:

```
id = data["id"]
```

Com a configuração acima, ele também vai tentar acessar como atributo, ou seja:

```
id = data.id
```


SQLAlchemy + Pydantic

Depois disso tudo, crie um arquivo dentro da pasta database chamado `crud.py`.
Dentro dele, iremos colocar algumas operações que faremos no banco de dados.

```
from sqlalchemy.orm import Session
from database import models, schemas

def get_user(db: Session, user_id: int):
    return db.query(models.User).filter(models.User.id == user_id).first()

def get_user_by_email(db: Session, email: str):
    return db.query(models.User).filter(models.User.email == email).first()

def get_users(db: Session, skip: int = 0, limit: int = 100):
    return db.query(models.User).offset(skip).limit(limit).all()

def create_user(db: Session, user: schemas.UserCreate):
    fake_hashed_password = user.password + "notreallyhased"
    db_user = models.User(email=user.email, hashed_password=fake_hashed_password)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
```

SQLAlchemy + Pydantic

```
def get_items(db: Session, skip: int = 0, limit: int = 100):  
    return db.query(models.Item).offset(skip).limit(limit).all()  
  
def create_user_item(db: Session, item: schemas.ItemCreate, user_id: int):  
    db_item = models.Item(**item.dict(), owner_id=user_id)  
    db.add(db_item)  
    db.commit()  
    db.refresh(db_item)  
    return db_item
```

SQLAlchemy + Pydantic

Uma breve explicação das funções:

- `get_user` -> Irá buscar os dados do usuário de acordo com seu `id`
- `get_user_by_email` -> Irá buscar os dados do usuario usando seu `email` como filtro para a busca
- `get_users` -> Busca a lista de usuários de forma paginada. O parametro `skip` define quantos registros serão pulados enquanto o parâmetro `limit` define quantos registros serão retornados. É o equivalente a seguinte query: `select * from user LIMIT 10 OFFSET 0`
- `create_user` -> Irá criar um usuário na tabela `User`, recebendo como argumento o `schema` que será tratado na requisição recebida no seu endpoint

SQLAlchemy + Pydantic

- `get_items` -> Busca a lista de usuárioritems de forma paginada. O parametro `skip` define quantos registros serão pulados enquanto o parâmetro `limit` define quantos registros serão retornados. É o equivalente a seguinte query: `select * from item LIMIT 10 OFFSET 0`
- `create_user` -> Irá criar um item na tabela `Item`, recebendo como argumento o `schema` que será tratado na requisição recebida no seu endpoint e para qual usuário será adicionado o item. Para isso, utilizamos o `user_id` para fazer esse vínculo

FastAPI

Ufa! Depois disso tudo, que tal criar os endpoints da minha aplicação?
Coloque o seguinte código no arquivo `main.py`

```
from sqlalchemy.orm import Session

from database import get_db, crud, schemas
from fastapi import Depends, FastAPI, HTTPException

app = FastAPI()

from http import HTTPStatus

@app.post("/users/", response_model=schemas.User)
def create_user(user: schemas.UserCreate, db: Session = Depends(get_db)):
    db_user = crud.get_user_by_email(db, email=user.email)
    if db_user:
        raise HTTPException(status_code=400, detail="Email already registered")
    return crud.create_user(db=db, user=user)
```

FastAPI

O primeiro endpoint que criaremos é o endpoint responsável para criar um usuários

Na definição dele, receberá uma requisição do tipo **POST**, no endereço **/users** e devolverá uma resposta do tipo **schemas.User**.

Para criar um usuário, o corpo da requisição deve ser no formato do **schema.UserCreate**, ou seja, um **JSON** que contenha todos os campos definidos neste esquema.

O parâmetro **db** que vemos na função é automaticamente resolvido pela função **Depends**, que recebe como argumento a função **get_db**, que retorna a conexão com o banco de dados

FastAPI

```
@app.get("/users/", response_model=list[schemas.User])
def read_users(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    users = crud.get_users(db, skip, limit)
    return users
```

Tratará uma requisição **GET** feita no endpoint **/users** e irá retornar uma lista de usuários, conforme especificada no parâmetro **response_model**.

Os parâmetros **skip** e **limit** são opcionais. Caso sejam informados, irá paginar os resultados conforme fizemos na função **get_users**

FastAPI

```
@app.get("/users/{user_id}", response_model=schemas.User)
def read_user(user_id: int, db: Session = Depends(get_db)):
    db_user = crud.get_user(db, user_id)
    if db_user is None:
        raise HTTPException(status_code=HTTPStatus.NOT_FOUND, detail="User not found")
    return db_user
```

Tratará uma requisição **GET** feita no endpoint `/users/{user_id}` e irá retornar os dados do usuário informado no parâmetro `user_id`.

Irá fazer uma busca no banco de dados para localizar o usuário informado e caso não encontre, retornar um status 404

FastAPI

```
@app.post("/users/{user_id}/items/", response_model=schemas.Item)
def create_item_for_user(user_id: int, item: schemas.ItemCreate, db: Session = Depends(get_db)):
    return crud.create_user_item(db, item, user_id)
```

Recebe uma requisição do tipo **POST**, no endereço **/users/{user_id}/items** e devolverá uma resposta do tipo **schemas.Item**.

Note que para criar um item, um usuário deve ser informado, para vincular o item ao respectivo usuário

FastAPI

```
@app.get("/items/", response_model=list[schemas.Item])
def read_items(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    items = crud.get_items(db, skip, limit)
    return items
```

Tratará uma requisição **GET** feita no endpoint **/items** e irá retornar uma lista de items, conforme especificada no parâmetro **response_model**.

Os parâmetros **skip** e **limit** são opcionais. Caso sejam informados, irá paginar os resultados conforme fizemos na função **get_items**

CORS

CORS ou **Cross-Origin Resource Sharing** se refere a situações quando existe um outro domínio tentando acessar sua aplicação. Isso é bem comum quando falamos de arquiteturas onde tem um frontend se comunicando com um backend. Isso acontece porque geralmente o frontend está hospedado em um domínio diferente do backend. Por exemplo:

- Frontend -> <https://app.aplicacao.com.br>
- Backend -> <https://api.aplicacao.com.br>

Apesar de estarem no mesmo domínio, são sub-dominios diferentes e com isso, todas as requisições feitas no backend serão bloqueadas caso não seja feitas

CORS

Para isso, no **FastAPI** podemos utilizar um **middleware** para configurar quais domínios poderão acessar os endpoints definidos na sua aplicação.

Primeiro, importe o **middleware** para configurar o **CORS**

```
from fastapi.middleware.cors import CORSMiddleware
```

Agora, adicione no seu **app** a configuração

```
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

CORS

Vamos as opções:

- `allow_origins` -> Define a lista de quais domínios podem acessar a aplicação. É possível definir `["*"]`, que vai aceitar requisições de qualquer origem
- `allow_credentials` -> Define que cookies podem ser utilizados durante as requisições. Porém o `allow_origins` não pode ser igual a `["*"]`
- `allow_methods` -> É possível definir uma lista de quais métodos `HTTP` sua aplicação aceitará. Por padrão, apenas o método `GET` é aceito.
- `allow_headers` -> Quais os cabeçalhos que sua aplicação aceitará. Os cabeçalhos `Accept - Language`, `Content-Language` e `Content-Type` sempre são aceitos, independente da configuração feita

CORS

Pronto! Basta rodar sua aplicação e acessar sua documentação através da URL <http://localhost:8000/docs>

Aparecerá a seguinte página:

FastAPI 0.1.0 OAS 3.1
[/openapi.json](#)

default

GET	/users/	Read Users	⌵
POST	/users/	Create User	⌵
GET	/users/{user_id}	Read User	⌵
POST	/users/{user_id}/items/	Create Item For User	⌵
GET	/items/	Read Items	⌵

CORS

Dentro dessa página é possível testar todas as requisições. Por exemplo, podemos testar a requisição para criar um usuário. Ao clicar no endpoint `/users` que irá fazer um `POST`, aparece um botão escrito `Try it out`. Ao clicar nele, é possível definir qual será o corpo da sua requisição

POST /users/ Create User

Parameters

No parameters

Request body required

application/json

```
{
  "email": "email@email.com",
  "password": "123456"
}
```

Execute

CORS

Ao clicar no **execute** a requisição será feita e você poderá ver o que resultou a requisição. É possível até ver como executar a requisição utilizando o **curl**

The screenshot displays a web application interface for testing HTTP requests. It features a 'Responses' tab at the top. Below this, there are sections for 'Curl' (showing a curl command for a POST request to localhost:8000/users/ with JSON body), 'Request URL' (http://localhost:8000/users/), and 'Server response'. The 'Server response' section includes a table with 'Code' (200) and 'Details'. The 'Details' section shows the 'Response body' as a JSON object: { "email": "email@email.com", "id": 1, "is_active": true, "items": [] }. Below the response body, the 'Response headers' are listed: access-control-allow-credentials: true, content-length: 62, content-type: application/json, date: Tue, 12 Sep 2023 20:44:15 GMT, and server: uvicorn. A 'Download' button is visible next to the response body.

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8000/users/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "email": "email@email.com",
    "password": "123456"
  }'
```

Request URL

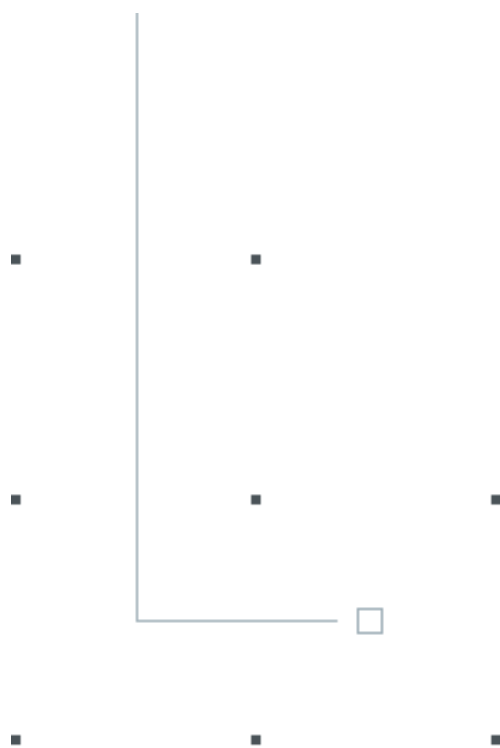
http://localhost:8000/users/

Server response

Code	Details
200	<p>Response body</p> <pre>{ "email": "email@email.com", "id": 1, "is_active": true, "items": [] }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true content-length: 62 content-type: application/json date: Tue, 12 Sep 2023 20:44:15 GMT server: uvicorn</pre>

Responses

Divirta-se e faça algumas requisições e veja como a aplicação se comporta!



.

F

-

I

A

P

