WOMAKERS CODE

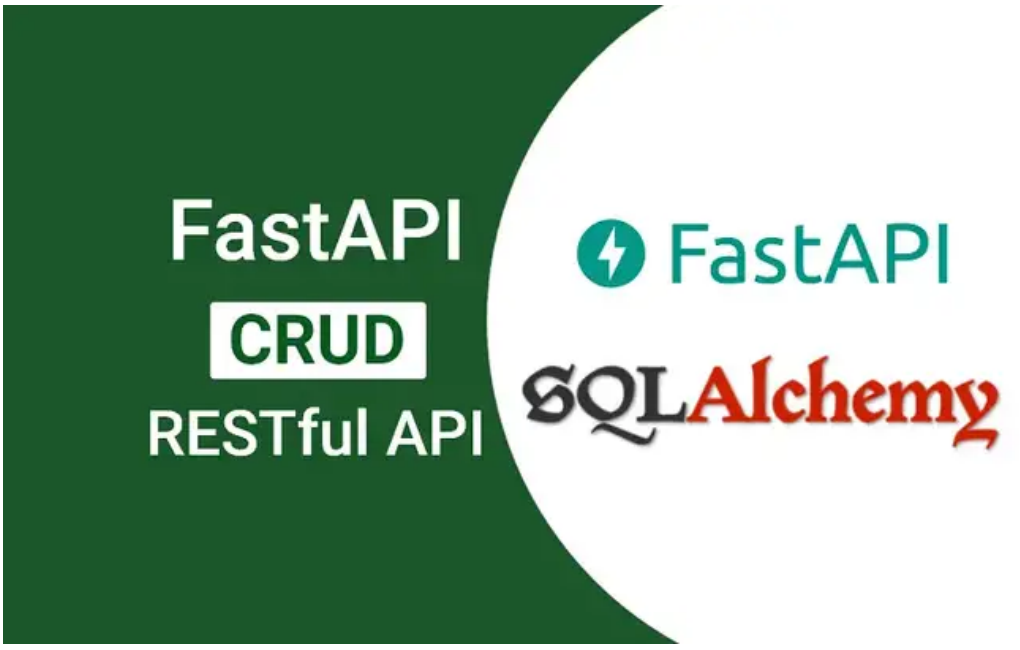
## BOOTCAMP BACK-END PYTHON E DJANGO:

# CONSUMO DE API COM FASTAPI

# Crie um aplicativo CRUD com FastAPI e SQLAlchemy

Neste artigo, fornecerei um guia simples e direto sobre como você pode construir um aplicativo CRUD com FastAPI e SQLAlchemy. O aplicativo FastAPI será executado em um servidor web Starlette, usará Pydantic para validação de dados e armazenará dados em um banco de dados SQLite.

O banco de dados padrão para este artigo é SQLite, mas colocarei uma seção neste tutorial para orientá-lo sobre como configurar um servidor Postgres com Docker e ajustar o código para funcionar com o banco de dados PostgreSQL em execução.



# Pré-requisitos

Antes de prosseguir com este tutorial, estes pré-requisitos são necessários para aproveitar ao máximo o curso.

- Tenha a versão mais recente do Python instalada em sua máquina. Para que FastAPI funcione corretamente, sua versão do Python deve ser **3.7+** .
- Algum conhecimento básico de Python será benéfico
- Ter conhecimento básico de designs de API

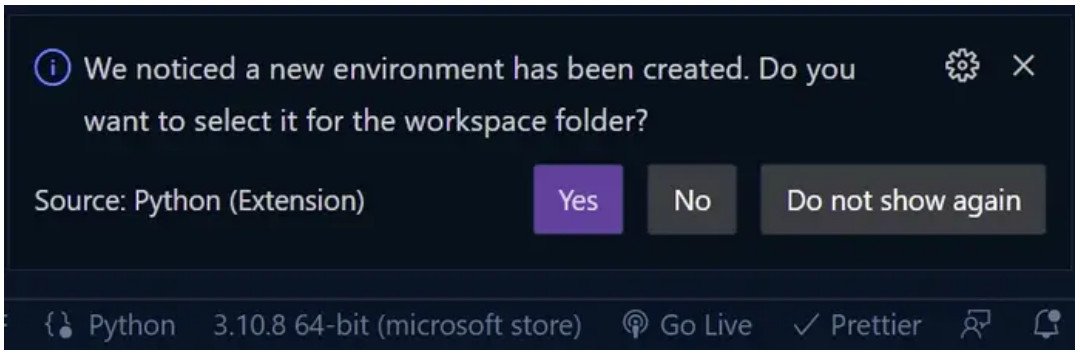
# Execute o aplicativo SQLAlchemy FastAPI localmente

- Baixe ou clone o código-fonte SQLAlchemy CRUD em **https://github.com/wpcodevo/fastapi_sqlalchemy** e abra o projeto com um IDE.
    - Abra o terminal integrado em seu IDE ou editor de texto e execute o seguinte comando para criar um ambiente virtual:
        - **Sistema operacional Windows**

          `python -m venv venv`
        - **Sistema operacional Mac ou Linux**

          `python3 -m venv venv`

- Se solicitado pelo seu IDE, clique em “ **Sim** ” para ativar o ambiente virtual para o espaço de trabalho.




  Se por acaso seu IDE ou editor de texto não solicitar a ativação do ambiente virtual, execute o comando abaixo para ativá-lo manualmente na área de trabalho.

  - Sistema operacional Windows ( **prompt de comando** ) `venv\Scripts\activate.bat`.
  - Sistema operacional Windows ( **Git Bash** ) `venv/Scripts/activate.bat`.
  - Sistema operacional Mac ou Linux `source venv/bin/activate`

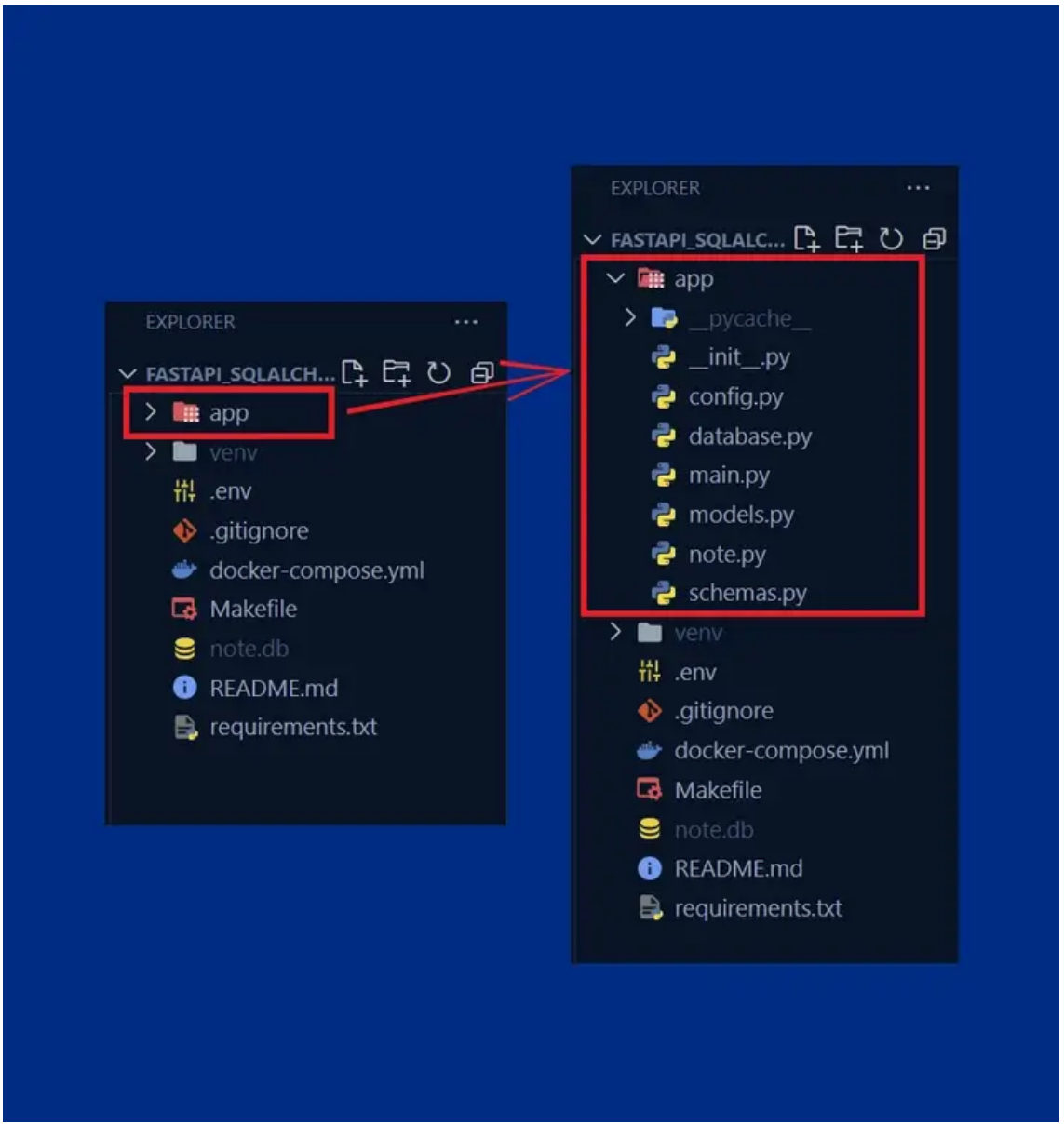
- Instale todos os módulos necessários para o projeto executando `pip install -r requirements.txt`
- Execute `uvicorn app.main:app --reload` para iniciar o servidor HTTP FastAPI na porta **8000** .
- Configure o aplicativo frontend para testar a API CRUD ou faça as solicitações de uma ferramenta de teste de API.

# Execute o aplicativo Frontend localmente

- Baixe ou clone o código-fonte do aplicativo React.js CRUD em **https://github.com/wpcodevo/reactjs-crud-note-app** e abra o projeto com um IDE ou editor de texto.
- Abra o terminal integrado em seu editor de texto e execute `yarn` ou `yarn install` instale todas as dependências necessárias.
- Inicie o servidor de desenvolvimento Vite na porta **3000** com `yarn dev`
- Abra uma nova guia em seu navegador e visite **http://localhost:3000/** para testar o aplicativo CRUD na API de back-end.

# Configure FastAPI e execute o servidor HTTP

No final deste guia completo, sua estrutura de pastas ficará assim:




Usarei **o VS Code** durante todo o tutorial, pois ele possui todos os recursos necessários para desenvolver uma API Python. No entanto, fique à vontade para usar qualquer IDE ou editor de texto.

Primeiramente, crie um novo diretório de projeto no local onde deseja que o código-fonte do projeto resida e abra-o com seu IDE preferido. Você pode nomear o projeto `fastapi_sqlalchemy`.

```
$ mkdir fastapi_sqlalchemy
$ cd fastapi_sqlalchemy
```

```
$ code . # opens the project with VS Code
```

Antes de podermos instalar as dependências do projeto, precisamos criar e ativar um ambiente virtual no espaço de trabalho atual. Para criar o ambiente virtual, execute o comando abaixo com base no seu sistema operacional.

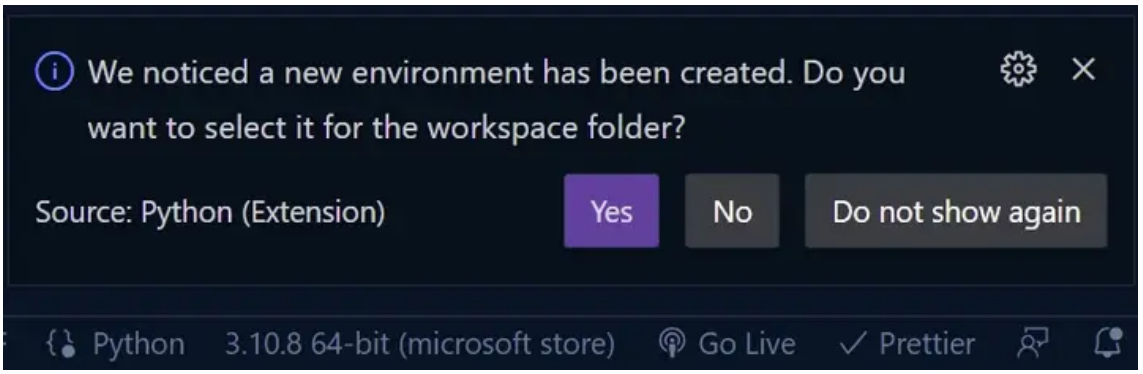
**Sistema Windows:**

```
$ py -3 -m venv venv
```

**Sistema  macOS:**

```
$ python3 -m venv venv
```

Quando seu IDE ou editor de texto solicitar que você ative o ambiente virtual na área de trabalho, clique no botão “ **Sim** ”.




Alternativamente, crie um `app/main.py` arquivo para fazer seu IDE ou editor de texto preparar o ambiente de desenvolvimento Python e execute o seguinte comando para ativar manualmente o ambiente virtual.

- Sistema operacional Windows ( **prompt de comando** )

  `venv\Scripts\activate.bat`.

- Sistema operacional Windows ( **Git Bash** ) `venv/Scripts/activate.bat`.
- Sistema operacional Mac ou Linux

  `source venv/bin/activate`

Se por acaso você também estiver usando o VS Code, poderá fechar e reabrir seu terminal integrado para ativar o ambiente virtual.

Agora crie um `app/__init__.py` arquivo vazio para transformar o diretório do aplicativo em um módulo Python.

Use este comando para instalar FastAPI e suas dependências de pares:

```
pip install fastapi[all]
```

Em seguida, abra o `app/main.py` arquivo e adicione este código:

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/api/healthchecker")
def root():
    return {"message": "Welcome to FastAPI with SQLAlchemy"}
```

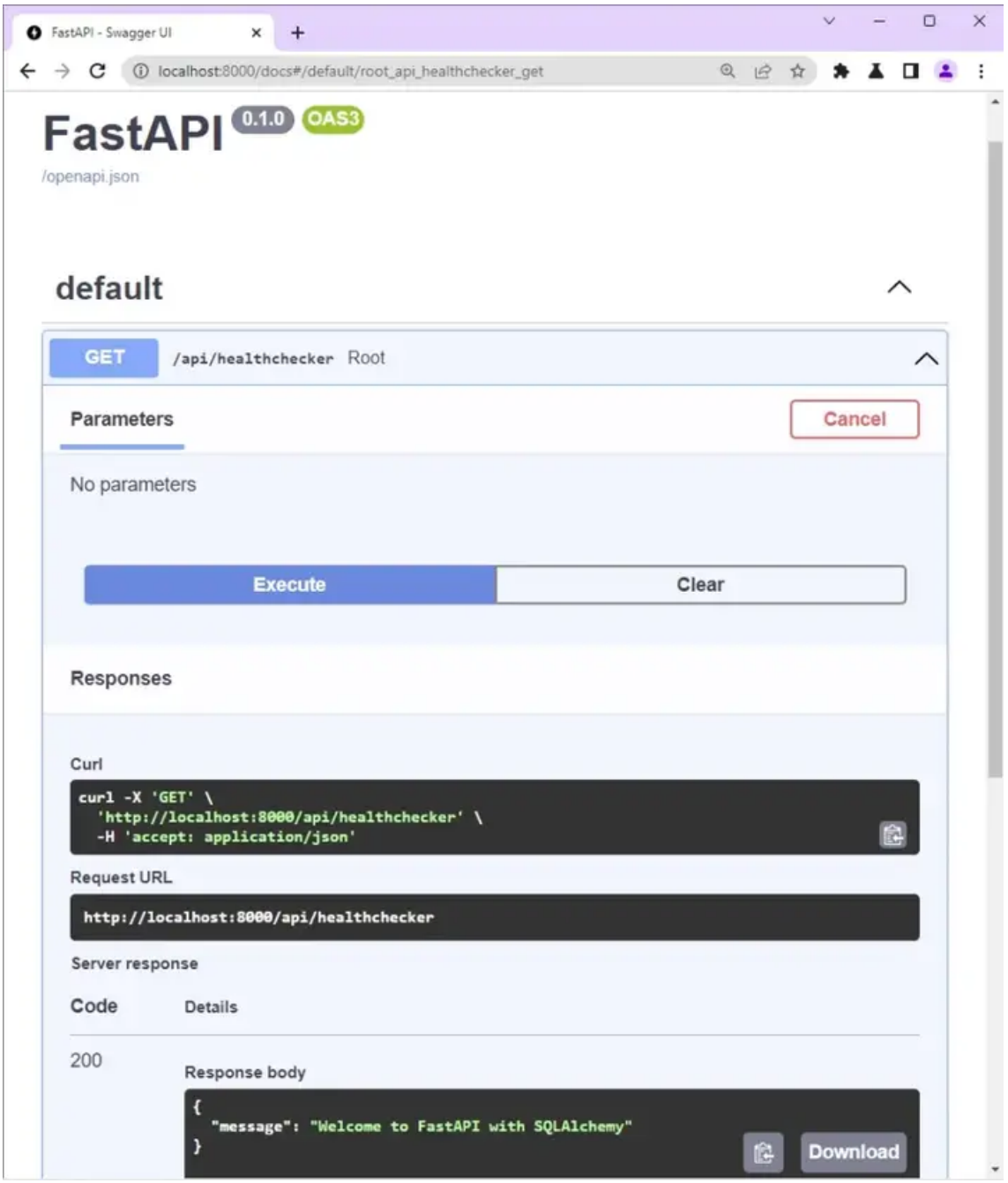
Acima, importamos a classe FastAPI, inicializamos o aplicativo evocando o método `FastAPI()` e adicionamos uma `/api/healthchecker` rota ao pipeline de middleware.

Execute este comando para iniciar o servidor FastAPI HTTP com **Uvicorn** .

```
uvicorn app.main:app --host localhost --port 8000 --reload
```

Assim que o servidor FastAPI estiver escutando na porta **8000** , abra uma nova guia em seu navegador e visite **http://localhost:8000/api/healthchecker** para ver o objeto JSON enviado pelo servidor.

Alternativamente, você pode fazer a mesma solicitação nos documentos do Swagger gerados pelo FastAPI. Navegue até **http://localhost:8000/docs** para ver a documentação da API.




# Projetando a API CRUD

O objetivo principal deste artigo é projetar uma API RESTful que possua funcionalidades CRUD. SQLAchemy já nos fornece funções CRUD de baixo nível que podemos usar para recuperar e alterar dados no banco de dados, mas criaremos nossas próprias funções CRUD de nível superior, chamadas

funções de operação de caminho, para evocar os métodos CRUD de nível inferior.

Abaixo está uma tabela contendo as rotas que definiremos na API e seus verbos HTTP correspondentes.

| MÉTODO HTTP | ROTA | DESCRIÇÃO |
|---|---|---|
| GET | /api/notes | Recuperar todas as notas |
| POST | /api/notes | Adicionar uma nova nota |
| GET | /api/notas/{noteId} | Obtenha uma única nota |
| PATCH | /api/notas/{noteId} | Editar uma nota |
| DELETE | /api/notas/{noteId} | Remover uma nota |

Como você pode ver, o verbo HTTP determina o tipo de operação que será realizada no banco de dados. Por exemplo, o endpoint responsável pela edição de um registro possui o método **PATCH** HTTP.

O `{noteId}` é simplesmente um espaço reservado que será substituído pelo ID de um registro.

Para começar, abra o `app/main.py` arquivo e substitua seu conteúdo pelas seguintes funções de operação do caminho CRUD.

**app/main.py**

```
from fastapi import FastAPI, APIRouter, status

app = FastAPI()
router = APIRouter()
```

```python
@router.get('/')
def get_notes():
    return "return a list of note items"


@router.post('/', status_code=status.HTTP_201_CREATED)
def create_note():
    return "create note item"


@router.patch('/{noteId}')
def update_note(noteId: str):
    return f"update note item with id {noteId}"


@router.get('/{noteId}')
def get_note(noteId: str):
    return f"get note item with id {noteId}"


@router.delete('/{noteId}')
def delete_note(noteId: str):
    return f"delete note item with id {noteId}"


app.include_router(router, tags=['Notes'], prefix='/api/notes')


@app.get("/api/healthchecker")
def root():
    return {"message": "Welcome to FastAPI with SQLAlchemy"}
```

Muita coisa está acontecendo acima, vamos decompô-lo:

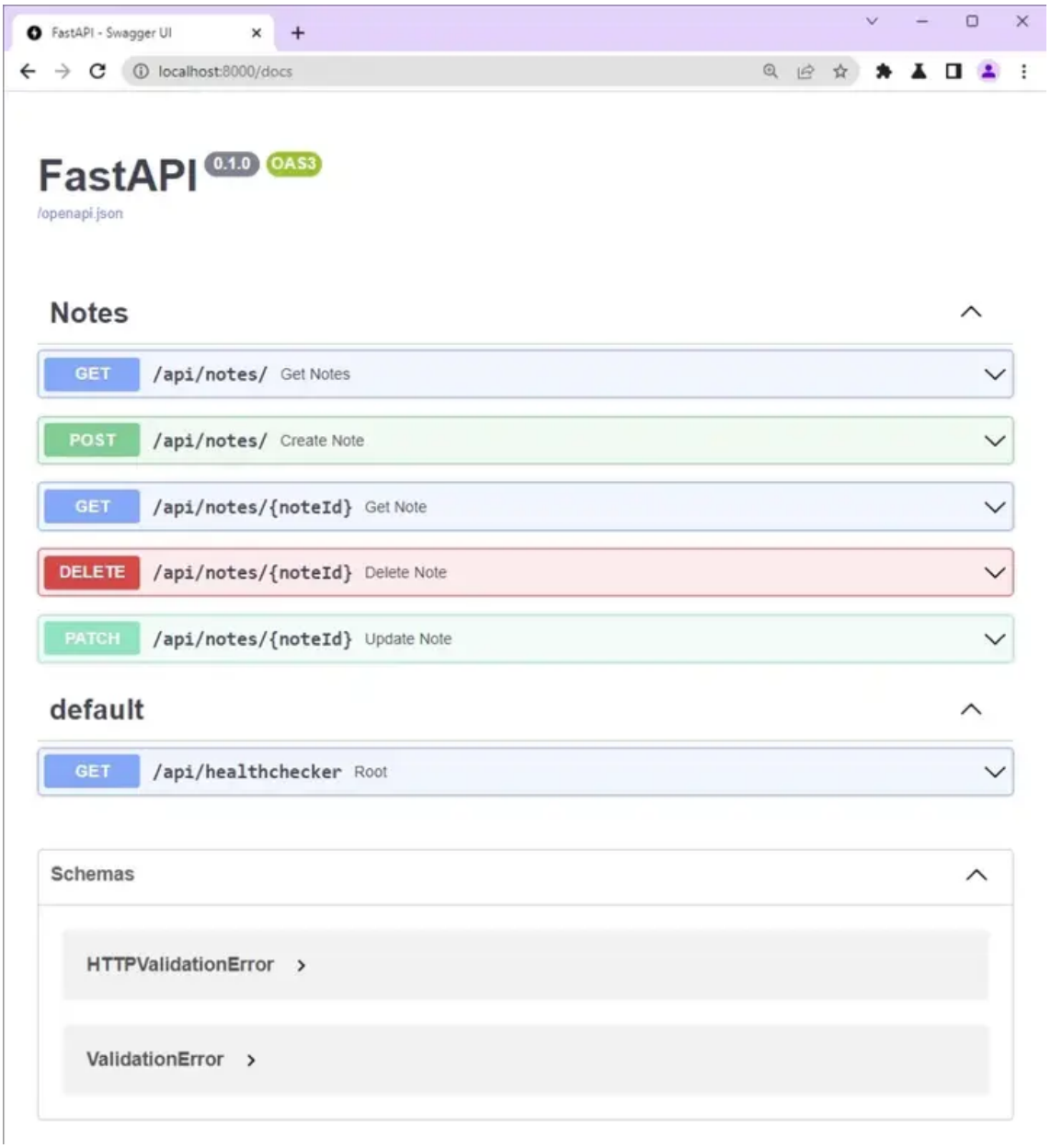
- Primeiro, importamos os módulos necessários no nível superior do arquivo.
- Em seguida, instanciamos a classe **FastAPI** e a atribuímos a uma variável de aplicativo. Além disso, criamos uma instância da classe **APIRouter** e a atribuímos a uma variável de roteador.

- Depois disso, criamos as funções de operação do caminho CRUD no roteador e registramos o roteador no aplicativo com o `app.include_router( )` método.

Agora inicie o servidor FastAPI HTTP executando este comando no terminal do diretório raiz.

```
uvicorn app.main:app --host localhost --port 8000 --reload
```

Quando o servidor estiver pronto para aceitar solicitações na porta **8000** , navegue até **http://localhost:8000/docs** para ver os documentos do Swagger gerados automaticamente. A documentação da API é baseada nos endpoints que definimos no servidor FastAPI.




Para testar cada rota, clique no menu suspenso e clique no botão **“Try it out”** (**Experimentar)**.

# Configurar SQLAlchemy com SQLite

Nesta seção, você criará um mecanismo SQLAlchemy e conectará o aplicativo a um banco de dados SQLite. Antes disso, abra seu terminal e instale essas dependências.

```
pip install SQLAlchemy fastapi-utils
```

- **SQLAlchemy**– Um ORM para Python
- **fastapi-utils**– Este pacote inclui vários utilitários, incluindo a função que usaremos para gerar UUID para cada registro.

Como você deve ter notado, não instalamos nenhum driver SQLite, pois todas as versões modernas do Python vêm com o módulo **sqlite3** .

Agora crie um `app/database.py` arquivo e adicione o seguinte código.

**app/banco de dados.py**

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLITE_DATABASE_URL = "sqlite:///./note.db"

engine = create_engine(
    SQLITE_DATABASE_URL, echo=True, connect_args={"check_same_thread": False}
)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Vamos decompor o código acima:

- Primeiro, importamos as partes do SQLAlchemy e criamos a URL do banco de dados SQLite.
- Em seguida, criamos o mecanismo SQLAlchemy e fornecemos a URL do banco de dados. Por padrão, o SQLite permite apenas um thread para comunicação, a fim de evitar que diferentes solicitações compartilhem a mesma conexão.

  Como estamos usando FastAPI, precisamos fornecer a `"check_same_thread": False` opção nos argumentos de conexão porque cada função de operação de caminho obterá sua própria sessão de banco de dados em uma dependência, portanto não há necessidade desse mecanismo padrão.
- A seguir, criamos as classes **SessionLocal** e **Base** evocando `sessionmaker()` e `declarative_base()` respectivamente.
- Por fim, criamos uma função **get_db** que criará uma nova sessão de banco de dados e fechará a sessão após o término da operação.

# Configure SQLAlchemy com PostgreSQL

Siga as etapas desta seção se preferir um banco de dados PostgreSQL em vez de SQLite. No entanto, se você decidiu usar o SQLite, pule esta seção.

O objetivo desta seção é ajudá-lo a configurar uma instância PostgreSQL com Docker e conectar a instância ao aplicativo FastAPI usando SQLAlchemy. Para fazer isso, abra seu terminal e instale estes módulos.

```
pip install SQLAlchemy fastapi-utils psycopg2
```

- **SQLAlchemy**– Um ORM para Python
- **fastapi-utils**– Um pacote de utilitários que nos ajudará a gerar GUIDs para as colunas de ID.
- **psycopg2**– Um adaptador de banco de dados PostgreSQL para SQLAlchemy

Com isso resolvido, crie um `docker-compose.yml` arquivo no diretório raiz e adicione as seguintes configurações do Docker Compose.

**docker-compose.yml**

```
version: '3'
services:
  postgres:
    image: postgres
    container_name: postgres
    ports:
      - '6500:5432'
    restart: always
    env_file:
      - ./.env
    volumes:
      - postgres-db:/var/lib/postgresql/data
volumes:
  postgres-db:
```

Quando o código acima for executado, o Docker Compose extrairá a imagem Postgres mais recente do Docker-Hub, construirá a imagem com as credenciais fornecidas em um arquivo **.env** e mapeará a porta **6500** para a instância do Postgres em execução.

Para disponibilizar as credenciais para o Docker-compose, crie um arquivo **.env** e adicione as seguintes variáveis de ambiente.

**.env**

```
DATABASE_PORT=6500
POSTGRES_PASSWORD=password123
POSTGRES_USER=postgres
POSTGRES_DB=fastapi
POSTGRES_HOST=postgres
POSTGRES_HOSTNAME=127.0.0.1
```

Execute este comando para iniciar o servidor Postgres:

```
docker-compose up -d
```

Agora que temos o servidor Postgres em execução no container Docker, vamos escrever algum código para disponibilizar as variáveis de ambiente para o aplicativo FastAPI. Felizmente, o Pydantic possui um recurso para fazer isso, então crie um `app/config.py` arquivo e adicione o código a seguir.

**app/config.py**

```
from pydantic import BaseSettings
```

```
class Settings(BaseSettings):
    DATABASE_PORT: int
    POSTGRES_PASSWORD: str
    POSTGRES_USER: str
    POSTGRES_DB: str
    POSTGRES_HOST: str
    POSTGRES_HOSTNAME: str

    class Config:
        env_file = './.env'


settings = Settings()
```

Para conectar o aplicativo FastAPI à instância do Postgres em execução, crie um `app/database.py` arquivo e adicione este código.

**app/banco de dados.py**

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from .config import settings
from fastapi_utils.guid_type import setup_guids_postgresql

POSTGRES_URL =
f"postgresql://{settings.POSTGRES_USER}:{settings.POSTGRES_PASSWORD}@{settings.POS
TGRES_HOSTNAME}:{settings.DATABASE_PORT}/{settings.POSTGRES_DB}"

engine = create_engine(
    POSTGRES_URL, echo=True
)
setup_guids_postgresql(engine)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()


def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Construímos a URL de conexão do banco de dados PostgreSQL com as variáveis armazenadas no arquivo **.env** `create_engine()` e passamos para ele o método SQLAlchemy .

Os UUIDs só podem ser gerados se a `pgcrypto` extensão estiver instalada na instância do Postgres, portanto a `setup_guids_postgresql()` função dirá ao Postgres para instalar a extensão se ela não existir.

# Crie modelo de banco de dados com SQLAlchemy

Agora estamos prontos para criar os modelos de banco de dados que o SQLAlchemy usará para gerar as tabelas SQL no banco de dados. Para evitar confusão entre os modelos SQLAlchemy e Pydantic, usaremos o arquivo `models.py` para os modelos SQLAlchemy e `schemas.py` para os modelos Pydantic.

## Modelo de banco de dados para banco de dados SQLite

Se você optou por um banco de dados SQLite, crie um `app/models.py` arquivo e adicione o código abaixo.

**app/models.py**

```
from .database import Base
from sqlalchemy import TIMESTAMP, Column, String, Boolean
from sqlalchemy.sql import func
from fastapi_utils.guid_type import GUID, GUID_DEFAULT_SQLITE


class Note(Base):
    __tablename__ = 'notes'
    id = Column(GUID, primary_key=True, default=GUID_DEFAULT_SQLITE)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    category = Column(String, nullable=True)
```

```
    published = Column(Boolean, nullable=False, default=True)
    createdAt = Column(TIMESTAMP(timezone=True),
                       nullable=False, server_default=func.now())
    updatedAt = Column(TIMESTAMP(timezone=True),
                       default=None, onupdate=func.now())
```

Importamos a classe **Base** do app/database.py arquivo e criamos um modelo de banco de dados que possui os campos necessários para adicionar um novo registro ao banco de dados. O __tablename__ atributo diz ao SQLAlchemy para usar a string fornecida como nome da tabela no banco de dados.

## Modelo de banco de dados para banco de dados Postgres

Se você escolher um banco de dados PostgreSQL, crie um app/models.py arquivo e adicione o código abaixo.

**app/models.py**

```
from .database import Base
from sqlalchemy import TIMESTAMP, Column, String, Boolean
from sqlalchemy.sql import func
from fastapi_utils.guid_type import GUID, GUID_SERVER_DEFAULT_POSTGRESQL


class Note(Base):
    __tablename__ = 'notes'
    id = Column(GUID, primary_key=True,
                server_default=GUID_SERVER_DEFAULT_POSTGRESQL)
    title = Column(String, nullable=False)
    content = Column(String, nullable=False)
    category = Column(String, nullable=True)
    published = Column(Boolean, nullable=False, server_default='True')
    createdAt = Column(TIMESTAMP(timezone=True),
                       nullable=False, server_default=func.now())
    updatedAt = Column(TIMESTAMP(timezone=True),
                       default=None, onupdate=func.now())
```

# Crie esquemas de validação com Pydantic

Vamos criar os esquemas Pydantic que FastAPI usará para analisar e validar as cargas de solicitação recebidas.

**app/schemas.py**

```
from datetime import datetime
from typing import List
from pydantic import BaseModel


class NoteBaseSchema(BaseModel):
    id: str | None = None
    title: str
    content: str
    category: str | None = None
    published: bool = False
    createdAt: datetime | None = None
    updatedAt: datetime | None = None

    class Config:
        orm_mode = True
        allow_population_by_field_name = True
        arbitrary_types_allowed = True


class ListNote Response(BaseModel):
    status: str
    results: int
    notes: List[NoteBaseSchema]
```

A configuração `orm_mode = True` na `Config` classe diz ao Pydantic para mapear os modelos para objetos ORM. Isso é necessário porque estamos usando SQLAlchemy ORM no projeto.

# Defina as funções de operação do caminho

Nesta seção, você criará cinco funções de operação de caminho que FastAPI usará para executar as operações CRUD no banco de dados. Esses manipuladores de rota são considerados de nível superior e terão a capacidade de chamar as funções CRUD de nível inferior fornecidas pelo SQLAlchemy.

- `get_notes`– Esta função de operação de caminho recuperará um número selecionado de itens de notas do banco de dados.
- `create_note`– Esta função de operação de caminho adicionará um novo item de nota ao banco de dados.
- `update_note`– Esta função de operação de caminho editará um item de nota existente no banco de dados.
- `get_post`– Esta função de operação de caminho recuperará um único item de nota do banco de dados.
- `delete_post`– Esta função de operação de caminho excluirá um item de nota do banco de dados.

Para começar, crie um `app/note.py` arquivo e adicione as seguintes dependências. Além disso, crie um novo roteador evocando a classe **APIRouter** .

**app/note.py**

```
from . import schemas, models
from sqlalchemy.orm import Session
from fastapi import Depends, HTTPException, status, APIRouter, Response
from .database import get_db
```

```
router = APIRouter()
```

## Obtenha todos os registros

A primeira função de operação de caminho executará uma operação **READ** e será chamada para recuperar uma lista paginada dos registros no banco de dados. Quando uma solicitação **GET**`/api/notes` atinge o endpoint, FastAPI evocará este controlador para lidar com a solicitação.

Os recursos incluídos nesta função de operação de caminho incluem:

- Recurso de pesquisa
- Recurso de limitação de resultados
- Recurso de paginação

**app/note.py**

```
# [...] get all records
@router.get('/')
def get_notes(db: Session = Depends(get_db), limit: int = 10, page: int =
1, search: str = ''):
    skip = (page - 1) * limit

    notes = db.query(models.Note).filter(

models.Note.title.contains(search)).limit(limit).offset(skip).all()
    return {'status': 'success', 'results': len(notes), 'notes': notes}
```

Vamos avaliar o código acima:

- Primeiro, criamos uma instrução que calcula o número de registros a serem ignorados no banco de dados usando os parâmetros **de página** e **limite** fornecidos na URL da solicitação.
- Em seguida, construímos a consulta ao banco de dados encadeando os métodos `.query()`, `.filter()`, `.limit()` e do SQLAlchemy `.offset()`.
- Depois disso, executamos a consulta chamando o `.all()`método para retornar todos os registros que correspondem à nossa consulta como uma lista de objetos.

## Crie um registro

A segunda função de operação de caminho executará uma operação **CREATE** e será chamada para adicionar um novo registro ao banco de dados. Quando uma solicitação **POST** é feita ao `/api/notes` endpoint, FastAPI chamará esse manipulador de rota para adicionar os dados fornecidos no corpo da solicitação ao banco de dados.

**app/note.py**

```
# [...] get all records

# [...] create record
@router.post('/', status_code=status.HTTP_201_CREATED)
def create_note(payload: schemas.NoteBaseSchema, db: Session = Depends(get_db)):
    new_note = models.Note(**payload.dict())
    db.add(new_note)
    db.commit()
    db.refresh(new_note)
    return {"status": "success", "note": new_note}
```

Criamos uma instância do modelo SQLAlchemy e passamos a instância para o `.add()` método. O método **add()** apenas registra uma operação de transação sem comunicá-la ao banco de dados.

Uma vez criada a transação, o `.commit()` método será chamado para persistir os dados permanentemente no banco de dados. O método de atualização obterá então uma versão atualizada do registro recém-criado.

## Atualizar um registro

A função de operação do terceiro caminho tratará da operação **UPDATE** e será chamada para editar um registro no banco de dados. Quando uma solicitação **PATCH** chega ao servidor, FastAPI chamará esse controlador de rota para tratar a solicitação.

**app/note.py**

```
# [...] get all records

# [...] create record

# [...] edit record
@router.patch('/{noteId}')
def update_note(noteId: str, payload: schemas.NoteBaseSchema, db: Session =
Depends(get_db)):
    note_query = db.query(models.Note).filter(models.Note.id == noteId)
    db_note = note_query.first()

    if not db_note:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'No note with this id: {noteId} found')
    update_data = payload.dict(exclude_unset=True)
    note_query.filter(models.Note.id == noteId).update(update_data,
                                                       synchronize_session=False)
    db.commit()
    db.refresh(db_note)
    return {"status": "success", "note": db_note}
```

No código acima, criamos a consulta ao banco de dados encadeando os métodos `.query()` e `.filter()`. Depois disso, evocamos o método `.first()` para retornar o primeiro registro que corresponda à nossa consulta.

Em seguida, chamamos o método `.dict()` no objeto JSON de carga útil e atribuímos o resultado à variável **update_data** antes de passá-lo para o método `.update()`.

Por fim, chamamos o método `.commit()` para persistir os dados no banco de dados e retornar o registro atualizado ao cliente.

## Recuperar um único registro

A quarta função de operação de caminho também executará uma operação **READ** para retornar um único registro do banco de dados. Quando uma solicitação **GET** é feita ao `/api/notes/{noteId}` endpoint, FastAPI evocará esse controlador de rota para recuperar o registro que corresponde à consulta do banco de dados e retornar o resultado ao cliente.

**app/note.py**

```
# [...] get all records

# [...] create record

# [...] edit record

# [...] get single record
@router.get('/{noteId}')
def get_post(noteId: str, db: Session = Depends(get_db)):
    note = db.query(models.Note).filter(models.Note.id == noteId).first()
    if not note:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"No note with this id: {id} found")
    return {"status": "success", "note": note}
```

## Excluir um único registro

A quinta e última função de operação de caminho executará uma operação **DELETE** para remover um único registro do banco de dados. Quando uma solicitação **DELETE** é feita ao `/api/notes/{noteId}` endpoint, FastAPI evocará esse controlador de rota para remover o registro que corresponde à consulta.

**app/note.py**

```
# [...] get all records

# [...] create record

# [...] edit record

# [...] get single record

# [...] delete record
@router.delete('/{noteId}')
def delete_post(noteId: str, db: Session = Depends(get_db)):
    note_query = db.query(models.Note).filter(models.Note.id == noteId)
    note = note_query.first()
    if not note:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'No note with this id: {id} found')
    note_query.delete(synchronize_session=False)
    db.commit()
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

O manipulador de rota primeiro consultará o banco de dados para verificar se existe um registro com esse ID antes que o método `.delete()` seja chamado para remover esse registro do banco de dados.

## Funções completas de operação de caminho

**app/note.py**

```
from datetime import datetime
from . import schemas, models
from sqlalchemy.orm import Session
from fastapi import Depends, HTTPException, status, APIRouter, Response
from .database import get_db

router = APIRouter()


@router.get('/')
def get_notes(db: Session = Depends(get_db), limit: int = 10, page: int =
1, search: str = ''):
    skip = (page - 1) * limit

    notes = db.query(models.Note).filter(

models.Note.title.contains(search)).limit(limit).offset(skip).all()
    return {'status': 'success', 'results': len(notes), 'notes': notes}


@router.post('/', status_code=status.HTTP_201_CREATED)
def create_note(payload: schemas.NoteBaseSchema, db: Session =
Depends(get_db)):
    new_note = models.Note(**payload.dict())
    db.add(new_note)
    db.commit()
    db.refresh(new_note)
    return {"status": "success", "note": new_note}


@router.patch('/{noteId}')
def update_note(noteId: str, payload: schemas.NoteBaseSchema, db: Session
= Depends(get_db)):
    note_query = db.query(models.Note).filter(models.Note.id == noteId)
    db_note = note_query.first()

    if not db_note:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'No note with this id: {noteId}
found')
    update_data = payload.dict(exclude_unset=True)
    note_query.filter(models.Note.id == noteId).update(update_data,

synchronize_session=False)
    db.commit()
    db.refresh(db_note)
    return {"status": "success", "note": db_note}


@router.get('/{noteId}')
def get_post(noteId: str, db: Session = Depends(get_db)):
    note = db.query(models.Note).filter(models.Note.id == noteId).first()
    if not note:
```

```
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f"No note with this id: {id} found")
    return {"status": "success", "note": note}


@router.delete('/{noteId}')
def delete_post(noteId: str, db: Session = Depends(get_db)):
    note_query = db.query(models.Note).filter(models.Note.id == noteId)
    note = note_query.first()
    if not note:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=f'No note with this id: {id} found')
    note_query.delete(synchronize_session=False)
    db.commit()
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

# Conecte o API Router ao aplicativo

Agora que definimos todas as funções de operação do caminho CRUD, vamos adicionar o roteador ao aplicativo FastAPI, invocar o método `.metadata.create_all()` para criar a tabela SQL no banco de dados SQLite e configurar o aplicativo para aceitar domínios de origem cruzada. Para fazer isso, abra o `app/main.py` arquivo e substitua seu conteúdo pelo código a seguir.

**app/main.py**

```
from app import models, note
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from .database import engine

models.Base.metadata.create_all(bind=engine)

app = FastAPI()

origins = [
    "http://localhost:3000",
]

app.add_middleware(
```

```
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


app.include_router(note.router, tags=['Notes'], prefix='/api/notes')


@app.get("/api/healthchecker")
def root():
    return {"message": "Welcome to FastAPI with SQLAlchemy"}
```

Muita coisa está acontecendo acima, vamos detalhar:

- No nível superior do arquivo, importamos os módulos necessários, incluindo a classe **CORSMiddleware** e o mecanismo que definimos no `app/database.py` arquivo.
- Evocamos o método `.Base.metadata.create_all()` disponível no modelo SQLAlchemy para criar a tabela SQL armazenada nos metadados.
- Em seguida, criamos uma lista das origens permitidas e evocamos o método `.add_middleware()` com as configurações CORS necessárias.
- Finalmente, usamos o método `.include_router()` para adicionar o roteador à pilha de middleware.

# Conclusão

Neste tutorial, você aprendeu como criar um aplicativo CRUD com FastAPI, SQLAlchemy ORM e um banco de dados SQLite. Além disso, você aprendeu

como criar esquemas de validação com Pydantic e modelos de banco de dados com SQLAlchemy.