FastAPI:

Um guia feito com uso da Gemini, ferramenta de IA generativa da Google.

Autores:

Aman Modesto

E Gemini

Versão: 1.0

Data: 15 de Julho de 2024

Imagem da capa gerada também com uso da Gemini.

Sumário

Introdução

O FastAPI é um framework Python moderno e de alto desempenho para construir APIs RESTful. Ele oferece uma sintaxe simples e expressiva, recursos poderosos e uma comunidade vibrante, tornando-o uma escolha popular para desenvolvedores que desejam criar APIs robustas, escaláveis e fáceis de manter.

Neste ebook, você aprenderá tudo o que precisa saber para construir APIs RESTful poderosas com o FastAPI. Abordaremos desde os conceitos básicos do FastAPI até técnicas avançadas de desenvolvimento, como modelagem de dados com Pydantic, integração com bancos de dados e testes com TDD.

Ao final deste ebook, você estará pronto para:

- Criar APIs RESTful completas com endpoints, validação de dados, manipulação de requisições e respostas e gerenciamento de erros.
- Modelar dados de forma robusta e eficiente utilizando o Pydantic.
- Integrar suas APIs com bancos de dados como o MongoDB para armazenar e recuperar dados.
- Escrever testes unitários e de integração para garantir a qualidade e confiabilidade do seu código.
- Implantar e monitorar suas APIs em produção utilizando ferramentas como o Uvicorn e o Sentry.

Este ebook é ideal para:

- Desenvolvedores Python que desejam aprender a construir APIs RESTful.
- Desenvolvedores experientes em APIs que desejam explorar o FastAPI e seus recursos poderosos.
- Estudantes de ciência da computação e engenharia de software que desejam se aprofundar no desenvolvimento de APIs modernas.

Comece sua jornada agora mesmo e desvende o poder do FastAPI!

Capítulo 1: Mergulhando no FastAPI

O FastAPI se consolidou como a estrela do universo do desenvolvimento web Python, conquistando os corações de programadores com sua simplicidade, velocidade e elegância. Este ebook te guiará por uma jornada de aprendizado abrangente, desde os fundamentos até as técnicas mais avançadas, transformando você em um mestre do FastAPI.

1.1 Conceitos Básicos

O que é FastAPI?

O FastAPI é um framework web moderno para Python, baseado em ASGI e Starlette. Ele combina a simplicidade do Flask com o poder do OpenAPI e do Pydantic, resultando em um framework altamente performante, seguro e fácil de usar.

Principais Funcionalidades:

- Roteamento automático e intuitivo
- Validação de dados robusta e automática
- Documentação automática e amigável
- Suporte completo para APIs RESTful
- Integração com bancos de dados, autenticação e muito mais

Benefícios do FastAPI:

- Desenvolvimento rápido e eficiente: Com menos código, você cria APIs mais rapidamente.
- Código limpo e elegante: A sintaxe intuitiva e expressiva torna o código fácil de ler e manter.
- Altamente performante: O FastAPI é um dos frameworks Python mais rápidos disponíveis.
- Seguro e confiável: Possui recursos de segurança integrados para proteger sua API.
- Fácil de aprender: A curva de aprendizado é suave, ideal para iniciantes e experientes.

1.2 Primeiros Passos

Criando sua primeira API:

- Instale o FastAPI e seus dependentes:

Bash

```
pip install fastapi
```

Use code with caution.

Defina um endpoint para gerenciar tarefas:

Python

```
from fastapi import FastAPI

app = FastAPI()
```

Use code with caution.

Receba requisições HTTP e processe os dados:

Python

```
@app.get("/tarefas/{tarefa_id}")
def get_tarefa(tarefa_id: int):
    # Simula a busca da tarefa no banco de dados
    tarefa = {"id": tarefa_id, "titulo": "Fazer compras", "descricao": "Comprar leite e pão"}
    return tarefa
```

Use code with caution.

Retorne respostas JSON com as informações da tarefa:

```
{
  "id": 1,
  "titulo": "Fazer compras",
```

```
  "descricao": "Comprar leite e pão"
}
```

Exemplo completo:

Python

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/tarefas/{tarefa_id}")
def get_tarefa(tarefa_id: int):
    # Simula a busca da tarefa no banco de dados
    tarefa = {"id": tarefa_id, "titulo": "Fazer compras", "descricao": "Comprar leite e pão"}
    return tarefa
```

Use code with caution.

1.3 Roteamento e Requisições

Crie endpoints para diferentes operações:

- GET /tarefas: Obter lista de tarefas
- GET /tarefas/{tarefa_id}: Obter tarefa específica
- POST /tarefas: Criar nova tarefa
- PUT /tarefas/{tarefa_id}: Atualizar tarefa existente
- DELETE /tarefas/{tarefa_id}: Deletar tarefa

Defina os métodos HTTP para cada endpoint:

GET: Obter informações

POST: Criar novos recursos

PUT: Atualizar recursos existentes

DELETE: Remover recursos

Processe os dados das requisições:

Parâmetros: Valores passados na URL (ex: /tarefas?filtro=pendente)

Cabeçalhos: Informações adicionais sobre a requisição (ex: Authorization: Bearer token)

Corpo da requisição: Dados em formato JSON, XML ou outros

Exemplo de código:

Python

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/tarefas")
def get_tarefas():
    # Simula a busca da lista de tarefas no banco de dados
    tarefas = [
        {"id": 1, "titulo": "Fazer compras", "descricao": "Comprar leite e pão"},
        {"id": 2, "titulo": "Estudar para prova", "descricao": "Revisar conceitos de matemática"},
    ]
    return tarefas

@app.post("/tarefas")
```

Capítulo 2: Habilidades Avançadas

Neste capítulo, você mergulhará em técnicas avançadas do FastAPI, explorando recursos que o tornarão um mestre na construção de APIs RESTful robustas e escaláveis.

2.1 Modelagem de Dados com Pydantic

Modelos Robustos:

Defina modelos de dados que garantem a confiabilidade e consistência:

Utilize o Pydantic para criar modelos com tipos de dados, regras de validação e documentação automática.

Garanta que os dados recebidos nas requisições estejam sempre no formato esperado.

Evite erros e inconsistências que podem comprometer a integridade da sua API.

Exemplo de código:

Python

```
from fastapi import FastAPI
from pydantic import BaseModel

class Tarefa(BaseModel):
    titulo: str
    descricao: str
    status: str = "pendente"  # Valor padrão

app = FastAPI()

@app.post("/tarefas")
```

```
def criar_tarefa(tarefa: Tarefa):
    # Simula a criação da tarefa no banco de dados
    nova_tarefa = {"id": 10, **tarefa.dict()}
    return nova_tarefa
```

Use code with caution.

Validação Automática:

Valide automaticamente os dados de entrada, evitando erros e protegendo seu sistema:

Defina regras de validação para cada campo do modelo (tipos de dados, tamanhos, valores permitidos, etc.).

O FastAPI verifica automaticamente se os dados recebidos nas requisições atendem às regras definidas.

Erros de validação são tratados de forma elegante, retornando respostas informativas para o cliente.

Exemplo de código:

Python

```
from fastapi import FastAPI
from pydantic import BaseModel

class Tarefa(BaseModel):
    titulo: str
    descricao: str
    status: str = "pendente"  # Valor padrão

    @validator("titulo")
    def validar_titulo(cls, v):
        if len(v) < 5:
```

```
        raise ValueError("O título deve ter pelo menos 5 caracteres")
    return v

app = FastAPI()

@app.post("/tarefas")
def criar_tarefa(tarefa: Tarefa):
  # Simula a criação da tarefa no banco de dados
  nova_tarefa = {"id": 10, **tarefa.dict()}
  return nova_tarefa
```

Use code with caution.

2.2 Banco de Dados com SQLAlchemy

Conectando ao Banco de Dados:

Conecte-se a bancos de dados de forma eficiente e segura:

Utilize o SQLAlchemy para criar conexões com diferentes tipos de bancos de dados (MySQL, PostgreSQL, SQLite, etc.).

Defina modelos de banco de dados que mapeiam as tabelas do banco para objetos Python.

Realize operações CRUD (Criar, Ler, Atualizar, Deletar) de forma intuitiva e eficiente.

Exemplo de código:

Python

```
from fastapi import FastAPI
from pydantic import BaseModel
from sqlalchemy import create_engine
```

```
engine = create_engine("postgresql://user:password@host:port/database")

class Tarefa(BaseModel):
    id: int
    titulo: str
    descricao: str
    status: str

class TarefaDB(BaseModel):
    __tablename__ = "tarefas"

    id: int
    titulo: str
    descricao: str
    status: str

    class Config:
        orm_mode = True

TarefaDB.metadata.create_all(engine)

app = FastAPI()

@app.get("/tarefas")
def get_tarefas():
    # Simula a busca da lista de tarefas no banco de dados
    with engine.connect() as conn:
        tarefas = conn.execute("SELECT * FROM tarefas").fetchall()
        return [Tarefa.from_orm(tarefa) for tarefa in tarefas]
```

Use code with caution.

Consultas e Manipulações:

Execute consultas SQL complexas e manipule dados com o poder do SQLAlchemy:

Utilize a sintaxe SQL familiar para realizar pesquisas, filtros, ordenações e muito mais.

Crie, atualize e exclua registros no banco de dados de forma eficiente.

Combine o poder do SQLAlchemy com o FastAPI para construir APIs RESTful poderosas.

Exemplo de código:

Python

from fastapi import FastAPI

from pydantic import

Capítulo 3: Aprimorando Habilidades e Explorando Ferramentas

Neste capítulo, você elevará suas habilidades com o FastAPI ao explorar ferramentas e práticas avançadas que tornarão suas APIs ainda mais robustas, seguras e escaláveis.

3.1 Documentação Automática com FastAPI Docs

Documentação Completa:

Gere documentação detalhada e amigável para sua API, facilitando o uso por desenvolvedores e clientes:

O FastAPI Docs gera automaticamente documentação em formato OpenAPI, Swagger UI e Markdown.

A documentação inclui informações sobre endpoints, modelos de dados, exemplos de uso e muito mais.

Facilite a integração da sua API com diferentes ferramentas e clientes.

Exemplo de código:

Python

```
from fastapi import FastAPI

app = FastAPI(docs_url="/api/docs")

@app.get("/tarefas")
def get_tarefas():
  # ...

@app.post("/tarefas")
def criar_tarefa(tarefa: Tarefa):
  # ...
```

Use code with caution.

Exemplos e Tutoriais:

Inclua exemplos de código e tutoriais na documentação para auxiliar os usuários:

Demonstre como utilizar os endpoints da sua API com exemplos práticos.

Crie tutoriais passo a passo para guiar os usuários na integração com sua API.

Aumente a usabilidade e a adoção da sua API pela comunidade.

Exemplo:

Markdown

Criar uma Tarefa

Este endpoint permite a criação de novas tarefas na sua lista.

Exemplo de requisição

```bash
curl -X POST http://localhost:8000/api/tarefas \
-H "Content-Type: application/json" \
-d '{
  "titulo": "Fazer compras",
  "descricao": "Comprar leite e pão"
}'
```

Use code with caution.

Resposta

JSON

```
{
  "id": 10,
  "titulo": "Fazer compras",
  "descricao": "Comprar leite e pão"
}
```

Use code with caution.

3.2 Monitoramento com Sentry

Monitorando Erros:

* **Monitore erros e exceções em tempo real para identificar problemas rapidamente e garantir a disponibilidade da sua API:**
    * Utilize o Sentry para registrar e rastrear erros que ocorrem na sua API.
    * Receba alertas instantâneos sobre erros críticos e tome medidas corretivas antes que impactem seus usuários.
    * Mantenha sua API sempre funcionando de forma estável e confiável.

Exemplo de configuração:

```python
from fastapi import FastAPI
from sentry import SentryClient

sentry_client = SentryClient("https://your-sentry-project.com")

app = FastAPI(sentry=sentry_client)

@app.get("/tarefas")
def get_tarefas():
    # ...

    try:
        # ...
    except Exception as e:
        sentry_client.capture_exception(e)
        raise e
```

Notificações e Alertas:

Configure notificações e alertas para ser avisado sobre erros críticos e tomar medidas corretivas:

Defina regras para receber notificações por email, SMS ou outros canais quando erros específicos ocorrerem.

Agende alertas para serem enviados em horários específicos ou quando a quantidade de erros exceder um limite definido.

Mantenha-se sempre informado sobre o estado da sua API e tome medidas proativas para garantir a qualidade do serviço.

3.3 Implementação de Boas Práticas

Arquitetura em Camadas:

Mantenha seu código organizado e modularizado, separando responsabilidades em camadas distintas:

Defina camadas para modelos de dados, lógica de negócios, controle de acesso e acesso a banco de dados.

Facilite a reutilização de código, a manutenção e a testabilidade da sua API.

Promova a escalabilidade e a sustentabilidade do seu projeto à medida que ele cresce.

Exemplo:

Python

```
# models.py
from pydantic import BaseModel

class Tarefa(BaseModel):
    # ...

# business_logic.py
from models import Tarefa
```

```
def criar_tarefa(tarefa: Tarefa):
    # ...

def atualizar_tarefa(tarefa_id: int, tarefa: Tarefa):
    # ...

def deletar_tarefa(tarefa_id: int):
    #
```

Capítulo 4: Ferramentas Essenciais para o Sucesso com o FastAPI

Neste capítulo, você conhecerá ferramentas essenciais que complementam o FastAPI e o elevam a um novo patamar de robustez, segurança e escalabilidade. Domine estas ferramentas e torne-se um mestre na construção de APIs RESTful que impressionam!

4.1 Uvicorn: Servidor ASGI para APIs High Performance

O que é Uvicorn?

O Uvicorn é um servidor ASGI (Asynchronous Server Gateway Interface) leve e de alto desempenho, ideal para executar APIs FastAPI em produção. Ele combina velocidade, confiabilidade e facilidade de uso, tornando-se a escolha perfeita para seus projetos.

Principais Características:

Altamente Performante: O Uvicorn utiliza técnicas avançadas de processamento assíncrono para lidar com um grande volume de requisições com rapidez e eficiência.

Fácil de Usar: A configuração e o uso do Uvicorn são extremamente simples, permitindo que você coloque sua API em funcionamento rapidamente.

Robusto e Confiável: O Uvicorn é testado rigorosamente e oferece recursos de segurança integrados para proteger sua API contra ataques e falhas.

Suporte para HTTP/2 e WebSockets: O Uvicorn suporta os protocolos HTTP/2 e WebSockets, permitindo comunicação mais rápida e eficiente com seus clientes.

Executando sua API com Uvicorn:

Para executar sua API com Uvicorn, basta seguir estes passos simples:

Instale o Uvicorn: pip install uvicorn

Execute o seguinte comando no terminal: uvicorn main:app --host 0.0.0.0 --port 8000

Onde:

main é o nome do seu módulo principal (ex: main.py)

app é o nome da instância do FastAPI

0.0.0.0 é o endereço IP que sua API irá escutar (todas as interfaces)

8000 é a porta que sua API irá utilizar

4.2 Alembic: Migrações de Banco de Dados Simples e Eficientes

O que é Alembic?

O Alembic é uma ferramenta poderosa para gerenciar migrações de banco de dados em projetos Python. Ele permite que você faça alterações na estrutura do seu banco de dados de forma segura e controlada, sem comprometer os dados existentes.

Principais Características:

Migrações Automáticas: O Alembic gera scripts automáticos para realizar as alterações necessárias no seu banco de dados, minimizando o trabalho manual.

Controle de Versão: O Alembic integra-se perfeitamente com sistemas de controle de versão como o Git, permitindo que você acompanhe o histórico de alterações do seu banco de dados.

Reversibilidade: O Alembic permite reverter migrações caso seja necessário, garantindo flexibilidade e segurança durante o desenvolvimento.

Suporte para Diversos Bancos de Dados: O Alembic funciona com diversos bancos de dados populares, como MySQL, PostgreSQL, SQLite e Oracle.

Utilizando o Alembic em seu projeto:

Instale o Alembic: pip install alembic

Crie um diretório para armazenar os scripts de migração: alembic init

Execute o comando alembic revision --message "Primeira migração" para criar a primeira migração.

Faça as alterações necessárias no seu modelo de banco de dados.

Execute o comando alembic upgrade head para aplicar a migração ao seu banco de dados.

4.3 Pydantic: Validação de Dados Robusta e Modelagem de Objetos

O que é Pydantic?

O Pydantic é uma biblioteca Python poderosa para validação de dados e modelagem de objetos. Ele permite que você defina modelos de dados com tipos de dados rigorosos, regras de validação e documentação automática.

Principais Características:

Validação Robusta: O Pydantic garante que os dados recebidos nas suas APIs estejam sempre no formato esperado, evitando erros e inconsistências.

Modelagem de Objetos: O Pydantic permite que você crie modelos de objetos que representam as entidades do seu sistema, facilitando o trabalho com dados.

Documentação Automática: O Pydantic gera automaticamente documentação detalhada dos seus modelos de dados, tornando

Capítulo 5: Integrando o FastAPI com o MongoDB

Neste capítulo, você explorará a integração do FastAPI com o MongoDB, um banco de dados NoSQL de alta performance e escalabilidade. Aprenda a realizar operações CRUD (Criar, Ler, Atualizar, Deletar) em documentos JSON de forma eficiente e segura, construindo APIs RESTful robustas e dinâmicas.

5.1 Introdução ao MongoDB

O que é o MongoDB?

O MongoDB é um banco de dados NoSQL que armazena dados em formato JSON, oferecendo flexibilidade, escalabilidade e performance para aplicações modernas. Ele é ideal para armazenar grandes volumes de dados não estruturados ou semiestruturados, como documentos, logs e dados de sensores.

Principais Características:

Armazenamento de Documentos JSON: O MongoDB armazena dados em documentos JSON, permitindo estruturas flexíveis e dinâmicas.

Esquema Semiestruturado: O MongoDB não exige um esquema rígido para os dados, permitindo que você adicione novos campos e atributos sem precisar alterar a estrutura do banco de dados.

Escalabilidade Horizontal: O MongoDB é facilmente escalável horizontalmente, permitindo que você distribua seus dados em vários servidores para atender a demandas crescentes.

Consultas Poderosas: O MongoDB oferece uma linguagem de consulta poderosa (MongoDB Query Language) para realizar pesquisas complexas em seus dados.

5.2 Conectando o FastAPI ao MongoDB

Instalando as Bibliotecas Necessárias:

Para conectar o FastAPI ao MongoDB, você precisará instalar as seguintes bibliotecas:

motor: pip install motor

pymongo: pip install pymongo

Estabelecendo a Conexão:

Para estabelecer a conexão com o MongoDB, siga estes passos:

Python

```
from motor.motor_asyncio import AsyncIOMotorClient

async def connect_db():
    client = AsyncIOMotorClient("mongodb://localhost:27017/")
    db = client["sua_base_de_dados"]
    return db

app.db = await connect_db()
```

Use code with caution.

5.3 Operações CRUD no MongoDB

Criando Documentos (Create):

Python

```
from app.db import app

@app.post("/tarefas")
async def criar_tarefa(tarefa: Tarefa):
    db = app.db
    tarefas_collection = db["tarefas"]
    await tarefas_collection.insert_one(tarefa.dict())
    return tarefa
```

Use code with caution.

Lendo Documentos (Read):

Python

```
from app.db import app

@app.get("/tarefas/{tarefa_id}")
async def get_tarefa(tarefa_id: int):
    db = app.db
    tarefas_collection = db["tarefas"]
    tarefa = await tarefas_collection.find_one({"_id": tarefa_id})
    if tarefa:
        return Tarefa.from_orm(tarefa)
    else:
        return {"mensagem": "Tarefa não encontrada"}
```

Use code with caution.

Atualizando Documentos (Update):

Python

```
from app.db import app

@app.put("/tarefas/{tarefa_id}")
async def atualizar_tarefa(tarefa_id: int, tarefa: Tarefa):
    db = app.db
    tarefas_collection = db["tarefas"]
    await tarefas_collection.update_one({"_id": tarefa_id}, {"$set": tarefa.dict()})
    return {"mensagem": "Tarefa atualizada"}
```

Use code with caution.

Excluindo Documentos (Delete):

Python

```
from app.db import app

@app.delete("/tarefas/{tarefa_id}")
async def deletar_tarefa(tarefa_id: int):
    db = app.db
    tarefas_collection = db["tarefas"]
    await tarefas_collection.delete_one({"_id": tarefa_id})
    return {"mensagem": "Tarefa deletada"}
```

Use code with caution.

5.4 Considerações Importantes

Gerenciamento de Erros: Implemente mecanismos robustos de gerenciamento de erros para lidar com exceções e garantir a confiabilidade da sua

Capítulo 6: TDD e Pytest: Testando com Responsabilidade

Neste capítulo, você mergulhará no mundo do TDD (Test Driven Development) e do Pytest, aprendendo a escrever testes robustos e eficientes para suas APIs FastAPI. Domine as melhores práticas de TDD e garanta que seu código esteja sempre livre de bugs e pronto para atender às demandas do seu projeto.

6.1 O que é TDD?

O TDD (Test Driven Development) é uma metodologia de desenvolvimento de software que propõe a escrita de testes antes da implementação do código. Essa abordagem garante que cada parte do código seja testada e validada antes de ser escrita, prevenindo a introdução de bugs e aumentando a qualidade do software.

6.2 Benefícios do TDD:

Código Mais Robusto: O TDD garante que seu código seja robusto e confiável, pois cada parte é testada e validada antes de ser escrita.

Menos Bugs: A escrita de testes antes da implementação do código ajuda a identificar e corrigir bugs mais cedo no processo de desenvolvimento.

Design Mais Claro: O TDD ajuda a pensar no design do seu código antes de escrevê-lo, resultando em um código mais organizado e fácil de manter.

Maior Confiança: O TDD garante que seu código esteja sempre funcionando corretamente, aumentando sua confiança no software desenvolvido.

6.3 Introduzindo o Pytest

O Pytest é uma biblioteca de testes poderosa e popular para Python. Ela oferece uma sintaxe intuitiva e expressiva, facilitando a escrita de testes robustos e eficientes. O Pytest também possui diversos recursos avançados, como fixtures, parametrização de testes e integração com ferramentas de cobertura de código.

6.4 Escrevendo Testes com Pytest

Para escrever testes com Pytest, siga estes passos:

Crie um diretório para armazenar seus testes: mkdir tests

Crie um arquivo de teste para cada módulo que você deseja testar: touch tests/test_modulo.py

Importe o módulo que você deseja testar e o Pytest:

Python

```
from modulo import funcao_a_testar

import pytest
```

Use code with caution.

Utilize a função @pytest.mark.parametrize para parametrizar seus testes:

Python

```
@pytest.mark.parametrize("valor_de_entrada,valor_esperado", [
    (1, 2),
    (2, 4),
    (3, 6),
])
def test_funcao_a_testar(valor_de_entrada, valor_esperado):
    # ...
```

Use code with caution.

Utilize as funções assert do Pytest para verificar os resultados dos seus testes:

Python

```
assert funcao_a_testar(valor_de_entrada) == valor_esperado
```

Use code with caution.

Execute os testes usando o comando pytest:

Bash

```
pytest tests
```

Use code with caution.

6.5 Testando APIs FastAPI com Pytest

Para testar APIs FastAPI com Pytest, você pode utilizar a biblioteca pytest-fastapi. Essa biblioteca oferece recursos específicos para testar endpoints, modelos de dados e outras funcionalidades do FastAPI.

Exemplo de teste de um endpoint:

Python

```
from app import app
from pytest_fastapi import test_client

client = test_client(app)

def test_get_tarefas():
    response = client.get("/tarefas")
    assert response.status_code == 200
    assert len(response.json()) > 0
```

Use code with caution.

6.6 Integração Contínua e Ferramentas de Automação

Para automatizar a execução dos seus testes e garantir que seu código esteja sempre funcionando corretamente, você pode utilizar ferramentas de integração contínua (CI) como o Travis CI ou o CircleCI.

Lembre-se: O TDD é uma prática que exige disciplina e dedicação. No entanto, os benefícios que ele traz para a qualidade e confiabilidade do seu código são imensos.

Recursos Adicionais:

- Documentação do Pytest
- Documentação do pytest-fastapi [URL inválido removido]
- Curso de TDD com Python [URL inválido removido]

Capítulo 7: Documentação Oficial das Ferramentas

Neste capítulo, você encontrará links para a documentação oficial das ferramentas citadas ao longo do ebook. Explore a documentação completa e aprofunde seus conhecimentos em cada uma delas para se tornar um mestre na construção de APIs RESTful com FastAPI.

7.1 FastAPI:

- Documentação Oficial: https://devdocs.io/fastapi/
- Tutoriais: https://fastapi.tiangolo.com/tutorial/
- Repositório GitHub: https://github.com/tiangolo/fastapi

7.2 Pydantic:

- Documentação Oficial: https://docs.pydantic.dev/latest/
- Tutoriais: https://m.youtube.com/watch?v=XIdQ6gO3Anc
- Repositório GitHub: https://github.com/pydantic/pydantic

7.3 SQLAlchemy:

- Documentação Oficial: https://docs.sqlalchemy.org/

Tutoriais: https://www.tutorialspoint.com/sqlalchemy/index.htm

Repositório GitHub: https://github.com/sqlalchemy/sqlalchemy

7.4 Sentry:

Documentação Oficial: https://docs.sentry.io/

Tutoriais: https://sentry.io/_/tutorials/

Repositório GitHub: https://github.com/getsentry/sentry

7.5 Uvicorn:

Documentação Oficial: https://www.uvicorn.org/

Tutoriais: https://www.uvicorn.org/

Repositório GitHub: https://github.com/encode/uvicorn

7.6 Alembic:

Documentação Oficial: https://alembic.sqlalchemy.org/

Tutoriais: https://alembic.sqlalchemy.org/en/latest/tutorial.html

Repositório GitHub: https://github.com/alembic/alembic

7.7 MongoDB:

Documentação Oficial: https://www.mongodb.com/docs/

Tutoriais: https://www.w3schools.com/mongodb/

Repositório GitHub: https://github.com/mongodb/mongo

7.8 Pytest:

Documentação Oficial: https://docs.pytest.org/en/7.1.x/contents.html

Tutoriais: https://realpython.com/pytest-python-testing/

Repositório GitHub: https://github.com/pytest-dev/pytest

7.9 Pytest-FastAPI:

Documentação Oficial: https://fastapi.tiangolo.com/tutorial/testing/

Tutoriais: https://fastapi.tiangolo.com/tutorial/testing/

Repositório GitHub: https://github.com/Pytest-with-Eric/pytest-fastapi-crud-example

Observações:

A documentação oficial das ferramentas citadas oferece informações detalhadas sobre instalação, configuração, uso e melhores práticas.

Explore os tutoriais e exemplos disponíveis para aprimorar seus conhecimentos e habilidades práticas.

Consulte o repositório GitHub de cada ferramenta para obter informações sobre atualizações, correções de bugs e roadmap de desenvolvimento.

Lembre-se: A documentação oficial é um recurso valioso para qualquer desenvolvedor. Utilize-a para se manter atualizado e aprimorar suas habilidades nas ferramentas que você utiliza diariamente.

Com este capítulo final, você finaliza sua jornada pelo ebook "Desvendando o FastAPI". Continue explorando as ferramentas citadas, aprofundando seus conhecimentos e construindo APIs RESTful cada vez mais robustas, seguras e escaláveis.