Criando uma Aplicação de To-Do’s usando o Python com a FastAPI e MongoDB

Compreendendo a FastAPI com o Python, o MongoDB e como criar aplicação simples de TO-DO.

Uma aplicação de tarefas usando a FastAPI - Um ebook feito com carinho e IA

Sobre este livro: Uma Jornada de Aprendizado com a Força da IA

Este livro é o resultado de uma paixão por programação e da busca por tornar o aprendizado mais acessível e divertido. Criado por Roberto Costa, como um projeto para o curso Python Backend AI Developer da DIO, este guia tem como objetivo descomplicar o desenvolvimento de aplicações com Python, FastAPI e MongoDB, utilizando a inteligência artificial para facilitar o processo.

A construção deste livro contou com a colaboração de duas poderosas ferramentas de IA: o Gemini do Google e a Leonard.ai. O Gemini, com sua capacidade de gerar textos e códigos, foi o responsável por elaborar os conteúdos, enquanto a Leonard.ai, com sua criatividade artística, criou imagens que ilustram e complementam os conceitos, tornando a experiência de aprendizado mais imersiva.

Esperamos que este livro seja um companheiro fiel em sua jornada de aprendizado, ajudando você a dominar as ferramentas necessárias para construir aplicações web incríveis e dar seus primeiros passos no universo da programação com Python!

Python: A Serpente que Conquistou o Mundo da Programação

Uma aplicação de tarefas usando a FastAPI - Um ebook feito com carinho e IA

A história do Python começa em 1989, nascido da mente do holandês Guido van Rossum. Guido buscava uma linguagem de programação mais simples e divertida, inspirada em sua paixão pela comédia britânica Monty Python. Assim, a linguagem Python foi batizada em homenagem ao grupo humorístico e, desde então, tem conquistado o mundo da programação com sua versatilidade e elegância.

Python se destaca por sua sintaxe clara e concisa, que se assemelha à linguagem natural, tornando-o ideal tanto para iniciantes quanto para programadores experientes. Sua natureza multiparadigma, permitindo a programação estruturada, orientada a objetos e funcional, garante que ele se adapte a diversos tipos de projetos.

O Poder da Serpente:

Python se tornou uma linguagem de programação poderosa e versátil, capaz de realizar tarefas complexas em diversos campos, como:

- **Desenvolvimento Web:** Frameworks como Django e Flask permitem a criação de aplicações web robustas e escaláveis.
- **Ciência de Dados:** Python é a linguagem de escolha para análise de dados, Machine Learning e Inteligência Artificial, graças a bibliotecas como Pandas, NumPy e Scikit-learn.
- **Automação:** Python é uma ferramenta poderosa para automatizar tarefas repetitivas, desde scripts simples até robôs que interagem com sistemas complexos.
- **Desenvolvimento de Jogos:** Bibliotecas como Pygame facilitam a criação de jogos 2D e 3D.
- **Aplicações de Escritório:** Python pode ser utilizado para automatizar tarefas em softwares como Excel e Word.

Python no Mercado:

Hoje, Python é uma das linguagens de programação mais populares do mundo, utilizada por empresas de todos os tamanhos, em diversos setores. Sua versatilidade e comunidade ativa o tornam uma escolha atraente para uma variedade de projetos, desde startups até grandes corporações.

A Jornada Continua:

Com sua comunidade vibrante e constante desenvolvimento, Python continua a evoluir, conquistando novos terrenos e oferecendo novas possibilidades para o futuro da programação. Se você está começando sua jornada no mundo da programação, ou já é um programador experiente,

Python é uma linguagem que vale a pena explorar, pois oferece um universo de oportunidades e desafios para quem busca construir soluções inovadoras e eficientes.

Programação Orientada a Objetos: Montando um Móvel com Blocos de Construção

Imagine que você está construindo um móvel, como uma mesa. Em vez de usar tábuas de madeira soltas, você utiliza blocos de construção pré-definidos. Cada bloco representa um "objeto" com características específicas: tamanho, cor, material, formato, etc.

A programação orientada a objetos (POO) funciona de maneira similar. Ela utiliza "objetos" como blocos de construção para criar programas complexos. Cada objeto possui características (atributos) e comportamentos (métodos).

Vamos voltar à mesa:

- **Objeto:** Mesa
- **Atributos:** Tamanho, cor, material, formato, número de pernas
- **Métodos:** Sentar, colocar objetos, mover, limpar

Criando Objetos em Python:

Em Python, você define objetos usando "classes". Uma classe é como um modelo ou um blueprint para criar objetos.

```
class Mesa:
  """Classe para representar uma mesa."""
  def __init__(self, tamanho, cor, material, formato, numero_pernas):
    """Inicializa os atributos da mesa."""
    self.tamanho = tamanho
    self.cor = cor
    self.material = material
    self.formato = formato
    self.numero_pernas = numero_pernas
  def sentar(self):
    """Método para simular o ato de sentar na mesa."""
    print("Você está sentado na mesa.")
  def colocar_objeto(self, objeto):
    """Método para colocar um objeto sobre a mesa."""
```

```
    print(f"Você colocou {objeto} sobre a mesa.")
```

Neste código, definimos a classe `Mesa`. O método `__init__` é chamado automaticamente ao criar um novo objeto da classe. Ele define os valores iniciais dos atributos da mesa.

Criando um objeto da classe Mesa:

```
mesa1 = Mesa("pequena", "marrom", "madeira", "quadrada", 4)
```

Criamos um objeto chamado `mesa1` da classe `Mesa` com os atributos especificados. Podemos então usar os métodos definidos na classe:

```
mesa1.sentar() # Imprime "Você está sentado na mesa."
mesa1.colocar_objeto("livro") # Imprime "Você colocou livro sobre a mesa."
```

Benefícios da POO:

- **Reutilização de código:** Você pode usar a mesma classe para criar diversos objetos, economizando tempo e esforço.
- **Organização:** A POO facilita a organização de código, tornando-o mais legível e fácil de manter.
- **Modularidade:** A POO permite que você divida um programa grande em partes menores (módulos) que podem ser desenvolvidas e testadas independentemente.
- **Abstração:** A POO permite que você oculte os detalhes internos de um objeto, mostrando apenas a informação necessária para o usuário.

A POO é um conceito fundamental na programação moderna, permitindo a criação de softwares complexos e eficientes. Com este guia simples, você deu o primeiro passo para entender este paradigma de programação. Explore mais e descubra todo o potencial da POO!

Bancos de Dados: Organizando a Informação como um Bibliotecário

Imagine uma biblioteca. Ela guarda milhares de livros, organizados por assunto, autor, título e outras informações. Para encontrar um livro específico, você precisa de um sistema que organize e categorize todas essas informações.

Um banco de dados é como um bibliotecário digital. Ele armazena, organiza e gerencia informações de forma estruturada, permitindo que você acesse, atualize e compartilhe dados de maneira rápida e eficiente.

Bancos de Dados do Mundo Real:

- **Biblioteca:** A biblioteca é um exemplo clássico de banco de dados. Cada livro é um registro, com informações como título, autor, data de publicação, assunto, etc. O catálogo da biblioteca é o sistema de gerenciamento de dados, permitindo que você encontre o livro que procura.
- **Consultório Médico:** Um consultório médico também utiliza um banco de dados para armazenar informações dos pacientes: nome, data de nascimento, histórico médico, exames, medicamentos, etc. O sistema de agendamento de consultas também é gerenciado por um banco de dados.

Aplicações Reais de Bancos de Dados:

- **E-commerce:** Lojas online utilizam bancos de dados para armazenar informações sobre produtos, clientes, pedidos, estoque, etc.
- **Redes Sociais:** Redes sociais usam bancos de dados para armazenar perfis de usuários, fotos, vídeos, posts, etc.
- **Serviços Financeiros:** Bancos e instituições financeiras utilizam bancos de dados para gerenciar contas, transações, investimentos, etc.
- **Sistemas de Gestão Empresarial (ERP):** Empresas utilizam bancos de dados para gerenciar suas operações, como estoque, finanças, recursos humanos, etc.

Tipos de Bancos de Dados:

Existem diversos tipos de bancos de dados, cada um com características e aplicações específicas. Os dois tipos mais comuns são:

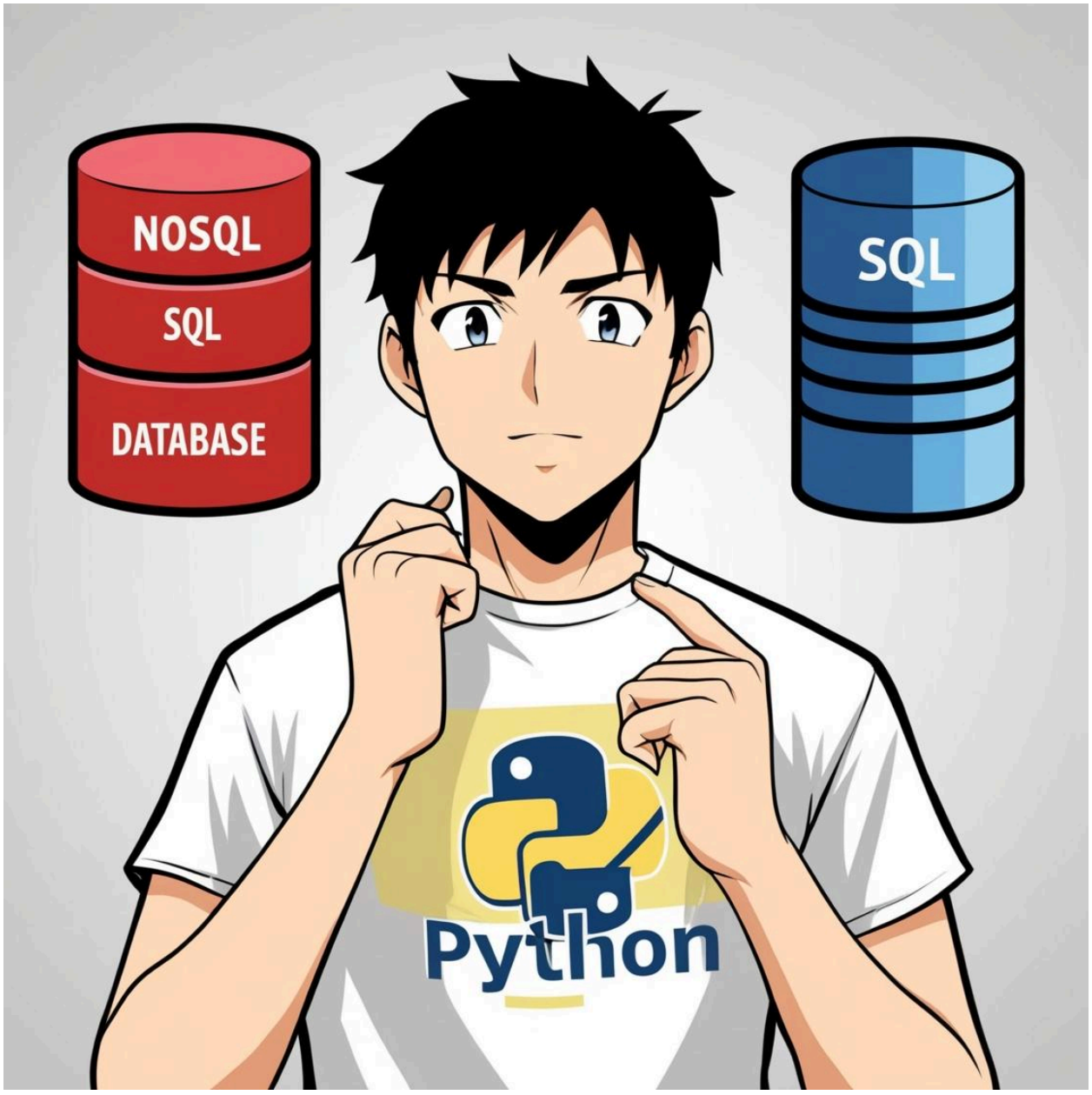
- **Bancos de Dados Relacionais (SQL):** Utilizam tabelas para organizar dados em colunas e linhas, como uma planilha.
- **Bancos de Dados Não Relacionais (NoSQL):** Oferecem mais flexibilidade na estrutura dos dados, ideal para lidar com grandes volumes de informações.

Explorando os Tipos de Bancos de Dados:

No próximos capítulo, vamos entender um pouco mais sobre a diferença dos bancos de dados SQL e NoSQL, explorando suas características, vantagens e desvantagens. Você irá entender como

escolher o tipo de banco de dados ideal para seus projetos e como utilizá-lo para construir aplicações robustas e eficientes.

Explorando os Tipos de Bancos de Dados: SQL vs. NoSQL



Agora que você já entende o que são bancos de dados e como eles são usados, vamos explorar dois tipos principais: SQL e NoSQL.

SQL: A Biblioteca Organizada

Imagine uma biblioteca com prateleiras organizadas e etiquetas bem definidas. Cada prateleira representa uma "tabela", e cada livro, um "registro" com informações como título, autor, data de publicação e gênero. Essa organização rígida facilita a busca por um livro específico.

O SQL (Structured Query Language - Linguagem de Consulta Estruturada) é como um sistema de organização para bancos de dados. Ele utiliza tabelas para armazenar dados em colunas e linhas, com relacionamentos bem definidos entre as tabelas.

Vantagens do SQL:

- **Estrutura:** As tabelas e colunas definem a estrutura dos dados de forma rígida, garantindo consistência e integridade.
- **Consultas:** As consultas SQL são poderosas e eficientes para recuperar informações específicas.
- **Transações:** As transações garantem a consistência dos dados, evitando erros e garantindo que as operações sejam completadas com sucesso.

Desvantagens do SQL:

- **Rigidez:** A estrutura rígida pode dificultar a adaptação a novos tipos de dados ou mudanças na estrutura.
- **Escalabilidade:** A escalabilidade pode ser um desafio, especialmente para lidar com grandes volumes de dados.
- **Complexidade:** A linguagem SQL pode ser complexa para iniciantes.

NoSQL: O Depósito Flexível

Imagine um depósito cheio de caixas, cada uma com diferentes tipos de objetos. Você pode organizar as caixas da maneira que preferir, sem necessidade de uma estrutura rígida.

O NoSQL (Not Only SQL) oferece mais flexibilidade na estrutura dos dados. Ele pode lidar com dados de diferentes tipos, como texto, imagens, vídeos, etc., e se adapta a estruturas complexas.

Vantagens do NoSQL:

- **Flexibilidade:** Permite armazenar dados sem necessidade de uma estrutura rígida.
- **Escalabilidade:** Ideal para lidar com grandes volumes de dados e alta demanda.
- **Performance:** Oferece alta performance para operações de leitura e gravação.

Desvantagens do NoSQL:

- **Estrutura:** A falta de estrutura pode complicar a gestão de dados complexos.
- **Consultas:** As consultas podem ser menos eficientes do que em SQL.
- **Transações:** As transações podem ser mais complexas e menos eficientes.

A Escolha Ideal:

A escolha entre SQL e NoSQL depende das necessidades do seu projeto. Se você precisa de uma estrutura rígida e controle sobre os dados, o SQL é a melhor opção. Se você precisa de flexibilidade, escalabilidade e performance para lidar com dados não estruturados, o NoSQL é a escolha ideal.

O MongoDB: Nosso Parceiro NoSQL

Para a nossa aplicação de tarefas, vamos utilizar o MongoDB, um banco de dados NoSQL popular e versátil. Ele é perfeito para lidar com dados não estruturados, como as tarefas do nosso app, e oferece alta performance e escalabilidade, além da possibilidade de rodar na nuvem de forma gratuita.

MongoDB: O Banco de Dados que Se Adapta à Nuvem como um Camaleão

Imagine um banco de dados que se transforma para se adaptar ao ambiente em que está. Ele pode ser pequeno e ágil, como um camaleão em um galho de árvore, ou grande e poderoso, como um gigante em uma selva. Esse é o MongoDB, um banco de dados NoSQL que se ajusta às necessidades do seu projeto, sem perder a flexibilidade e a performance.

MongoDB é um banco de dados de documentos, ou seja, ele armazena dados em formato JSON (JavaScript Object Notation), como se fossem objetos com diferentes tipos de informações, desde textos até imagens e vídeos. Essa flexibilidade o torna ideal para lidar com dados não estruturados, como os que encontramos em aplicações web modernas, como redes sociais e e-commerce.

MongoDB na Nuvem: Gratuito e Escalável

Mas o MongoDB vai além da flexibilidade. Ele também se destaca por sua capacidade de se integrar perfeitamente à nuvem, oferecendo uma solução completa para seus projetos. Você pode instalar o MongoDB em seu próprio servidor, mas a opção mais popular e vantajosa é utilizar

serviços como o Atlas, da MongoDB, que oferece um ambiente de banco de dados na nuvem de forma gratuita, com recursos como backup, replicação e segurança.

MongoDB como um Depósito Flexível:

Imagine um depósito cheio de caixas, cada uma com diferentes tipos de objetos. O MongoDB é como um sistema de organização que permite acessar, atualizar e compartilhar esses objetos de maneira eficiente. Você pode encontrar um objeto específico por meio de etiquetas (chaves) ou por meio de buscas por características (atributos).

MongoDB: A Escolha Ideal para a Nossa Aplicação:

Em nossa aplicação de tarefas, o MongoDB se torna o parceiro ideal para armazenar e gerenciar nossas informações. Sua flexibilidade nos permite armazenar dados complexos, como descrições de tarefas, datas de entrega e status, e sua integração com a nuvem garante um ambiente seguro e escalável para nossa aplicação, sem custos adicionais.

Frameworks e Bibliotecas: Construindo um Carro com Peças Pré-fabricadas

Imagine que você quer construir um carro. Você poderia começar do zero, fundindo aço, fabricando peças e montando tudo manualmente. Mas imagine que você encontra um kit de peças prontas, como motor, rodas, chassi e painel, que já funcionam perfeitamente juntas. Você economiza tempo e esforço, e ainda tem um carro pronto para funcionar!

Na programação, frameworks e bibliotecas são como esses kits de peças prontas. Eles oferecem componentes e ferramentas pré-fabricadas que facilitam a construção de softwares complexos.

Frameworks: O Chassi do Seu Software

Um framework é como o chassi de um carro, a estrutura básica que define como o software funciona. Ele oferece um conjunto de componentes e ferramentas para lidar com tarefas comuns, como conectar-se a um banco de dados, gerenciar usuários e processar requisições web.

Bibliotecas: Ferramentas para Cada Função

As bibliotecas são como as ferramentas do seu carro: cada uma possui uma função específica, como um motor, uma caixa de câmbio ou um sistema de freios. Você pode usar bibliotecas para realizar tarefas como manipular imagens, processar dados, enviar emails ou integrar seu software a outros sistemas.

FastAPI: Acelere o Desenvolvimento de APIs com Python

FastAPI é um framework moderno e poderoso para construir APIs web em Python. Ele é como um carro de corrida para o desenvolvimento de APIs, oferecendo alta performance, segurança e facilidade de uso.

FastAPI se destaca por sua velocidade impressionante, graças ao uso de ASGI (Asynchronous Server Gateway Interface), que permite que ele processe múltiplas requisições simultaneamente. É como ter um motor potente que garante agilidade e eficiência.

A sintaxe do FastAPI é intuitiva e concisa, facilitando a escrita e a leitura do código, como se você estivesse dirigindo um carro com um volante e pedais fáceis de usar. Ele também oferece recursos de segurança para proteger suas APIs de ataques, como um sistema de freios confiável que garante a segurança da viagem.

Além de todas essas vantagens, o FastAPI facilita a documentação de suas APIs. Ele gera automaticamente uma interface interativa que permite a outros desenvolvedores testar e entender como usar suas APIs, como se você tivesse um manual completo e intuitivo para usar seu carro.

FastAPI: A Escolha Ideal para APIs Modernas

FastAPI é uma excelente escolha para construir APIs web robustas, eficientes e modernas. Ele oferece a combinação perfeita de velocidade, segurança, facilidade de uso e documentação, tornando o desenvolvimento de APIs em Python uma experiência mais rápida, eficiente e prazerosa.

A Nossa Aventura: Construindo um App de Tarefas

Uma aplicação de tarefas usando a FastAPI - Um ebook feito com carinho e IA

com Python, FastAPI e MongoDB

Agora que já exploramos o mundo da programação com Python, aprendemos os conceitos da Orientação a Objetos, desvendamos os segredos dos bancos de dados e descobrimos a magia dos frameworks e bibliotecas, é hora de colocar a mão na massa!

Vamos construir um aplicativo simples, mas poderoso, para gerenciar suas tarefas: um app TO-DO. Imagine um sistema que te ajuda a organizar sua vida, com a possibilidade de criar, editar, apagar e visualizar suas tarefas de forma organizada.

Um CRUD Completo para Suas Tarefas:

Nosso app TO-DO será baseado em um CRUD (Create, Read, Update, Delete - Criar, Ler, Atualizar, Deletar). Utilizando FastAPI, o framework que torna o desenvolvimento de APIs rápido e fácil, e MongoDB, o banco de dados flexível e escalável que aprendemos, construiremos um sistema onde você poderá:

- **Criar novas tarefas:** Defina o título, a descrição, a data de entrega, subtarefas e o status da sua tarefa.
- **Atualizar tarefas:** Editar o título, a descrição, a data de entrega ou o status da tarefa conforme necessário.
- **Apagar tarefas:** Remova as tarefas que já foram concluídas ou que não são mais relevantes.
- **Visualizar tarefas:** Veja todas as suas tarefas, com filtros para:
    - **Tarefas concluídas:** Visualize apenas as tarefas que você já finalizou.
    - **Tarefas não concluídas:** Veja quais tarefas ainda precisam ser realizadas.
    - **Tarefas com subtarefas:** Organize tarefas complexas dividindo-as em subtarefas.
    - **Título da tarefa:** Busque rapidamente por uma tarefa específica pelo seu título.

O Próximo Passo: Seu Frontend Personalizado

Após construirmos a base do nosso app TO-DO, você poderá dar asas à sua criatividade e adicionar um frontend, a interface visual do aplicativo, para que ele seja ainda mais intuitivo e fácil de usar.

Começando a Aventura:

Este livro te guiará passo a passo na construção do seu app TO-DO, desde a configuração inicial até os testes finais. Ao final, você terá um aplicativo funcional e pronto para ser personalizado e

adaptado às suas necessidades. Prepare-se para uma jornada emocionante no mundo da programação e da criação de aplicativos com Python, FastAPI e MongoDB!

Iniciando a Aventura: Criando o Nosso Projeto com Poetry

Chegou a hora de colocar a mão na massa! Vamos iniciar a construção do nosso projeto, utilizando Python, FastAPI e MongoDB para dar vida ao nosso app TO-DO.

Para gerenciar os pacotes do nosso projeto, vamos utilizar o Poetry, uma ferramenta poderosa que facilita a instalação, atualização e organização das bibliotecas que usaremos.

1. Instalando o Poetry:

Se você ainda não tem o Poetry instalado em sua máquina, abra o terminal e execute o seguinte comando:

```
pip install poetry
```

2. Criando o Diretório do Projeto:

Navegue até o diretório onde você deseja criar o projeto. Para este exemplo, vamos criar uma pasta chamada "todo_project":

```
mkdir todo_project
cd todo_project
```

3. Inicializando o Projeto com Poetry:

Dentro do diretório do projeto, execute o comando `poetry init` para iniciar um novo projeto com o Poetry. Responda às perguntas do Poetry, como o nome do projeto ("todo_project"), a versão (ex: "0.1.0"), a descrição e a linguagem de programação (Python).

4. Instalando as Bibliotecas Essenciais:

Agora que o projeto foi criado, vamos instalar as bibliotecas necessárias:

```
poetry add fastapi uvicorn pydantic pydantic-settings motor pytest
pytest-asyncio pre-commit httpx
```

5. Criando o Arquivo `pyproject.toml`:

O Poetry cria um arquivo chamado `pyproject.toml` que define as configurações do seu projeto, incluindo as bibliotecas instaladas, as dependências e outras informações importantes.

Pronto para Ação!

Agora você possui um projeto Python com o Poetry configurado e as bibliotecas essenciais instaladas. É hora de começar a construir o nosso app TO-DO! Em breve, vamos explorar os arquivos e pastas que compõem a estrutura do projeto e dar os primeiros passos na construção da nossa aplicação.

Lembre-se:

- Você pode visualizar a lista de bibliotecas instaladas em seu projeto acessando o arquivo `pyproject.toml`.
- Para atualizar as bibliotecas instaladas, basta usar o comando `poetry update` no terminal.

Configurando o Cenário: Preparando o Terreno para o Nosso App TO-DO

Antes de começarmos a construir as funcionalidades do nosso app TO-DO, precisamos configurar o ambiente e preparar o terreno para que nossa aplicação funcione perfeitamente.

1. Sua Conta no MongoDB Atlas:

Para utilizar o MongoDB, você precisará criar uma conta gratuita no MongoDB Atlas. O MongoDB Atlas oferece um ambiente de banco de dados na nuvem, ideal para projetos como o nosso.

- Acesse o site do MongoDB Atlas (https://www.mongodb.com/cloud/atlas) e crie uma conta gratuita.
- Crie um novo cluster, definindo as configurações de acordo com as suas necessidades.
- Obtenha a URL de conexão do seu cluster. Essa URL será utilizada para conectar a sua aplicação ao banco de dados.

2. Variáveis de Ambiente e o Arquivo `.env`:

A URL de conexão do seu banco de dados é uma informação sensível que não deve ser exposta diretamente no código. Para proteger essa informação, usaremos um arquivo `.env` para armazenar as variáveis de ambiente.

- Crie um arquivo chamado `.env` na raiz do seu projeto (na mesma pasta do arquivo `pyproject.toml`).
- Adicione a seguinte linha ao arquivo `.env`, substituindo "url do mongodb do usuário" pela URL de conexão do seu cluster:

```
DATABASE_URL="sua url para conexão com o MongoDB"
```

3. O Arquivo `main.py`: A Porta de Entrada da Nossa Aplicação

Dentro da pasta "todo_project", crie um arquivo chamado `main.py`. Este será o arquivo de entrada da nossa aplicação, utilizando a FastAPI. Por enquanto, ele estará vazio:

```
# todo_project/main.py

# Este arquivo será preenchido com o código da aplicação.
```

4. Criando a Pasta `core` e Configurando o `config.py`

Dentro da pasta "todo_project", crie uma pasta chamada "core".

- Adicione um arquivo vazio chamado `__init__.py` dentro da pasta "core".
- Crie um arquivo chamado `config.py` dentro da pasta "core".

```
# todo_project/core/config.py

from pydantic import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
PROJECT_NAME: str = "TODO API"
   ROOT_PATH: str = "/"
   DATABASE_URL: str
   model_config = SettingsConfigDict(env_file=".env")
settings = Settings()
```

Este arquivo define as configurações do projeto utilizando a biblioteca `pydantic_settings`. A classe `Settings` carrega as variáveis de ambiente do arquivo `.env`, garantindo que a URL do banco de dados seja carregada de forma segura.

5. Criando a Pasta `db` e Configurando o `mongo.py`:

Dentro da pasta "todo_project", crie uma pasta chamada "db".

- Adicione um arquivo vazio chamado `__init__.py` dentro da pasta "db".
- Crie um arquivo chamado `mongo.py` dentro da pasta "db".

```python
# todo_project/db/mongo.py

from motor.motor_asyncio import AsyncIOMotorClient

from todo_project.core.config import settings

class MongoClient:
    def __init__(self) -> None:
        self.client: AsyncIOMotorClient = AsyncIOMotorClient(settings.DATABASE_URL)
        self.db = self.client["todo-project"]

    def get(self) -> AsyncIOMotorClient:
        return self.client


db_client = MongoClient()
```

Este arquivo define a classe `MongoClient` para conectar ao banco de dados MongoDB, utilizando a URL de conexão armazenada em `settings.DATABASE_URL`.

Próximos Passos:

Com a estrutura do projeto configurada, estamos prontos para começar a construir as funcionalidades do nosso app TO-DO. Nos próximos capítulos, vamos explorar os conceitos de schemas, models, controllers e usecases, que serão fundamentais para a construção da nossa aplicação.

Construindo o Edifício do Nosso App: Schemas, Models, Controllers e Use Cases

Agora que a base do nosso projeto está pronta, vamos organizar a estrutura do nosso app TO-DO de forma eficiente e modular. Para isso, vamos utilizar quatro componentes importantes: schemas, models, controllers e usecases.

1. Schemas: Definindo a Estrutura dos Dados

Os schemas, ou esquemas, são como os "planos" da nossa aplicação. Eles definem a estrutura dos

dados que serão armazenados no banco de dados, como o tipo de informação, os campos obrigatórios e os formatos válidos.

Exemplo: No nosso app TO-DO, o schema da tarefa pode definir campos como título, descrição, data de entrega, status e outros detalhes importantes.

2. Models: Representando os Objetos do Nosso App

Os models são a representação dos objetos do nosso app. Eles são como as "classes" que definimos na programação orientada a objetos, representando as entidades do nosso sistema, como tarefas, usuários, etc.

Exemplo: No nosso app TO-DO, teremos um model "Tarefa" que irá conter os atributos definidos no schema, como título, descrição, data de entrega, status, etc.

3. Controllers: Gerenciando as Interações do Usuário

Os controllers são os "gerentes" do nosso app. Eles recebem as requisições do usuário, como criar uma nova tarefa, atualizar uma tarefa existente ou visualizar todas as tarefas. Os controllers são responsáveis por processar essas requisições, chamar as funções de negócio (usecases) e retornar a resposta ao usuário.

4. Use Cases: As Regras de Negócio da Nossa Aplicação

Os usecases, ou casos de uso, são as regras de negócio do nosso aplicativo. Eles definem as ações que podem ser realizadas com os objetos do sistema, como criar uma nova tarefa, marcar uma tarefa como concluída, filtrar tarefas por status ou buscar tarefas por título.

Exemplo: No nosso app TO-DO, podemos ter um usecase "Criar Tarefa" que recebe os dados da nova tarefa, valida os dados e salva a tarefa no banco de dados.

Trabalhando em Sintonia:

Schemas, models, controllers e usecases trabalham em conjunto para garantir a organização e o funcionamento eficiente do nosso app. Nos próximos capítulos, vamos explorar cada um desses componentes em detalhes, construindo nosso app TO-DO passo a passo.

Criando os Schemas: Definindo a Estrutura dos Dados do Nosso App TO-DO

Para organizar as informações do nosso app TO-DO de forma consistente, vamos definir schemas que irão determinar a estrutura dos dados que serão armazenados no banco de dados. Os schemas são como os "planos" que garantem que os dados sejam armazenados e recuperados de forma correta.

1. Criando a Pasta `schemas` e o arquivo `__init__.py`

Primeiramente, crie uma pasta chamada `schemas` dentro da pasta principal do seu projeto. Dentro da pasta `schemas`, crie um arquivo vazio chamado `__init__.py`. Esse arquivo indica que a pasta `schemas` é um pacote Python, permitindo que os arquivos dentro dela sejam importados de forma organizada.

2. Criando o Arquivo `base.py`:

Crie um arquivo chamado `base.py` dentro da pasta `schemas`:

```python
# todo_project/schemas/base.py

from datetime import datetime
from pydantic import UUID4, BaseModel, Field


class BaseSchemaMixin(BaseModel):
    class Config:
        from_attributes = True


class OutSchema(BaseModel):
    id: UUID4 = Field()
    completed: bool = Field(default=False)
    created_at: datetime = Field()
    updated_at: datetime = Field()
```

Neste arquivo, definimos a classe `BaseSchemaMixin` que será utilizada como base para outros schemas. A classe `OutSchema` define os campos comuns a todos os tipos de tarefas:

- **`id`:** Um identificador único para cada tarefa, utilizando UUID4.
- **`completed`:** Um valor booleano (True ou False) que indica se a tarefa está concluída.
- **`created_at`:** A data e hora em que a tarefa foi criada, no formato padrão `datetime`.
- **`updated_at`:** A data e hora em que a tarefa foi atualizada pela última vez, no formato

padrão `datetime`.

3. Criando o Arquivo `subtodo.py`:

Crie um arquivo chamado `subtodo.py` dentro da pasta `schemas`:

```
# todo_project/schemas/subtodo.py

from pydantic import Field

from todo_project.schemas.base import BaseSchemaMixin, OutSchema


class SubTodoSchema(OutSchema, BaseSchemaMixin):
    title: str = Field(max_length=256)
```

Neste arquivo, definimos a classe `SubTodoSchema` que herda os campos da classe `OutSchema` e adiciona um novo campo `title` para o nome da subtarefa.

4. Criando o Arquivo `todo.py`:

Crie um arquivo chamado `todo.py` dentro da pasta `schemas`:

```
# todo_project/schemas/todo.py

from datetime import time, date
from pydantic import Field

from todo_project.schemas.base import BaseSchemaMixin, OutSchema
from todo_project.schemas.subtodo import SubTodoSchema


class TodoSchema(OutSchema, BaseSchemaMixin):
   title: str = Field(max_length=256)
   note: str = Field(max_length=1024, default="")
   url: str = Field(max_length=256, default="")
   due_date: date = Field()
   due_time: time = Field(..., alias="time")
   priority: str = Field(default="none", max_length=5,
pattern=r"none|low|mid|high")
   is_flagged: bool = Field(default=False)
   sub_tasks: list[SubTodoSchema] = Field(default_factory=list)
```

Neste arquivo, definimos a classe `TodoSchema` que também herda os campos da classe `OutSchema` e adiciona campos específicos para tarefas:

- **`title`:** O título da tarefa, com um limite de 256 caracteres.
- **`note`:** Uma nota para a tarefa, com um limite de 1024 caracteres.
- **`url`:** Uma URL relacionada à tarefa, com um limite de 256 caracteres.
- **`due_date`:** A data de vencimento da tarefa, no formato padrão `date`.
- **`due_time`:** O horário de vencimento da tarefa, no formato padrão `time`.
- **`priority`:** A prioridade da tarefa, com opções "none", "low", "mid" ou "high".
- **`is_flagged`:** Indica se a tarefa está marcada como importante.
- **`sub_tasks`:** Um array de subtarefas.

Schemas: A Base da Estrutura do Nosso App

Com esses schemas, já temos a estrutura básica para organizar as informações do nosso app TO-DO de forma consistente. No próximo capítulo, vamos explorar como esses schemas serão utilizados para criar os models que representam as entidades do nosso aplicativo.

Criando os Models: Representando as Entidades do Nosso App TO-DO

Agora que definimos a estrutura dos dados com os schemas, vamos dar vida a essas informações criando os models. Os models representam as entidades do nosso aplicativo, como tarefas, subtarefas, etc., e são a ponte entre os schemas e o banco de dados.

1. Criando o Arquivo `base.py`:

Crie um arquivo chamado `base.py` dentro da pasta `models`:

```
# todo_project/models/base.py

from typing import Optional

from pydantic import BaseModel, Field

from todo_project.schemas.base import OutSchema


class BaseSchemaMixin(BaseModel):
    class Config:
        from_attributes = True


class MongoModel(OutSchema, BaseSchemaMixin):
    id: Optional[str] = Field(alias="_id")
```

Neste arquivo, definimos a classe `MongoModel` que serve como base para outros models. Ela herda

os campos da classe `OutSchema` e adiciona um campo `id` que será usado para armazenar o ID da tarefa no banco de dados MongoDB.

2. Criando o Arquivo `subtodo.py`:

Crie um arquivo chamado `subtodo.py` dentro da pasta `models`:

```
# todo_project/models/subtodo.py

from todo_project.schemas.subtodo import SubTodoSchema

class SubTodoModel(SubTodoSchema):
    pass
```

Neste arquivo, definimos a classe `SubTodoModel` que herda diretamente do `SubTodoSchema`, evitando repetição de código.

3. Criando o Arquivo `todo.py`:

Crie um arquivo chamado `todo.py` dentro da pasta `models`:

```
# todo_project/models/todo.py

from todo_project.schemas.todo import TodoSchema

class TodoModel(TodoSchema):
    pass
```

Neste arquivo, definimos a classe `TodoModel`, que representa uma tarefa no nosso aplicativo. Ela herda diretamente do `TodoSchema`.

Models: Conectando Schemas e Banco de Dados

Com esses models, estamos construindo a ponte entre os schemas, que definem a estrutura dos dados, e o banco de dados MongoDB, onde esses dados serão armazenados. Os models garantem que as informações sejam organizadas de forma consistente e que os dados sejam validados antes de serem armazenados.

No próximo capítulo, vamos explorar como os controllers serão usados para interagir com os models e gerenciar as ações do usuário.

Criando os Use Cases: Definindo as Regras de Negócio do Nosso App TO-DO

Os usecases, ou casos de uso, são como os "cérebros" do nosso aplicativo. Eles definem as ações que podem ser realizadas com os objetos do sistema, como criar uma nova tarefa, marcar uma tarefa como concluída, filtrar tarefas por status ou buscar tarefas por título.

1. Criando a Pasta `usecases` e o Arquivo `__init__.py`

Crie uma pasta chamada `usecases` dentro da pasta principal do seu projeto. Dentro da pasta `usecases`, crie um arquivo vazio chamado `__init__.py`. Esse arquivo indica que a pasta `usecases` é um pacote Python, permitindo que os arquivos dentro dela sejam importados de forma organizada.

2. Criando o Arquivo `todo.py`:

Crie um arquivo chamado `todo.py` dentro da pasta `usecases`:

```python
# todo_project/usecases/todo.py

from typing import List, Optional
from datetime import date

from fastapi import Depends
from motor.motor_asyncio import AsyncIOMotorClient

from todo_project.models.todo import TodoModel
from todo_project.schemas.todo import TodoSchema
from todo_project.db.mongo import db_client

class TodoUseCases:
    def __init__(self, db: AsyncIOMotorClient = Depends(db_client.get)) -> None:
        self.db = db.db["todos"]

    async def create_todo(self, todo: TodoModel) -> TodoSchema:
        result = await self.db.insert_one(todo.dict())
        new_todo = await self.db.find_one({"_id": result.inserted_id})
        return TodoSchema(**new_todo)

    async def get_all_todos(self) -> List[TodoSchema]:
        todos = await self.db.find().to_list(length=None)
        return [TodoSchema(**todo) for todo in todos]

    async def get_todo(self, todo_id: str) -> TodoSchema:
        todo = await self.db.find_one({"_id": todo_id})
        return TodoSchema(**todo)

    async def update_todo(self, todo_id: str, todo: TodoSchema) -> TodoSchema:
        await self.db.update_one({"_id": todo_id}, {"$set": todo.dict(exclude={"id"})})
        updated_todo = await self.db.find_one({"_id": todo_id})
        return TodoSchema(**updated_todo)
```

```
    async def delete_todo(self, todo_id: str) -> None:
        await self.db.delete_one({"_id": todo_id})

    async def get_todos_by_status(self, completed: bool) -> List[TodoSchema]:
        todos = await self.db.find({"completed": completed}).to_list(length=None)
        return [TodoSchema(**todo) for todo in todos]

    async def get_todos_by_title(self, title: str) -> List[TodoSchema]:
        todos = await self.db.find({"title": {"$regex": title, "$options": "i"}}).to_list(length=None)
        return [TodoSchema(**todo) for todo in todos]

    async def get_todos_by_is_flagged(self, is_flagged: bool) -> List[TodoSchema]:
        todos = await self.db.find({"is_flagged": is_flagged}).to_list(length=None)
        return [TodoSchema(**todo) for todo in todos]

    async def get_todos_by_due_date(self, due_date: date) -> List[TodoSchema]:
        todos = await self.db.find({"due_date": due_date}).to_list(length=None)
        return [TodoSchema(**todo) for todo in todos]
```

Neste arquivo, definimos a classe `TodoUseCases` que é responsável por executar as regras de negócio do nosso app TO-DO, como:

- **`create_todo`:** Cria uma nova tarefa.
- **`get_all_todos`:** Lista todas as tarefas.
- **`get_todo`:** Busca uma tarefa específica pelo seu ID.
- **`update_todo`:** Atualiza uma tarefa específica pelo seu ID.
- **`delete_todo`:** Deleta uma tarefa específica pelo seu ID.
- **`get_todos_by_status`:** Filtra tarefas por status (concluídas ou não concluídas).
- **`get_todos_by_title`:** Filtra tarefas pelo título.
- **`get_todos_by_is_flagged`:** Filtra tarefas por importância (marcadas ou não como importantes).
- **`get_todos_by_due_date`:** Filtra tarefas por data de vencimento.

Use Cases: O Cérebro do Nosso App

Os usecases são os responsáveis por implementar as regras de negócio do nosso app TO-DO. Eles interagem com o banco de dados MongoDB para realizar as operações necessárias e garantem que os dados sejam manipulados de forma consistente e segura.

No próximo capítulo, vamos finalmente implementar os controllers, que irão conectar os usecases aos endpoints da API e permitir que os usuários interajam com o nosso app TO-DO.

Controlando a Ação: Gerenciando as Interações do Usuário com os Controllers

Agora que definimos as regras de negócio do nosso app TO-DO com os usecases, vamos implementar os controllers, que são os "gerentes" da nossa aplicação. Os controllers são responsáveis por receber as requisições do usuário, como criar uma nova tarefa, atualizar uma tarefa existente ou visualizar todas as tarefas. Eles processam essas requisições, chamam os usecases para executar as regras de negócio e retornam a resposta ao usuário.

1. Criando o Arquivo `todo.py`:

Crie um arquivo chamado `todo.py` dentro da pasta `controllers`:

```python
# todo_project/controllers/todo.py

from fastapi import APIRouter, Depends, status
from fastapi.responses import JSONResponse
from fastapi.exceptions import HTTPException
from fastapi.param_functions import Query
from datetime import date
from typing import Optional

from todo_project.schemas.todo import TodoSchema, SubTodoSchema
from todo_project.models.todo import TodoModel
from todo_project.usecases.todo import TodoUseCases


router = APIRouter(prefix="/todos", tags=["todos"])


@router.post("/", status_code=status.HTTP_201_CREATED, response_model=TodoSchema)
async def create_todo(todo: TodoSchema, usecases: TodoUseCases = Depends()):
    todo_model = TodoModel(**todo.dict())
    new_todo = await usecases.create_todo(todo_model)
    return new_todo


@router.get("/", status_code=status.HTTP_200_OK, response_model=list[TodoSchema])
async def get_all_todos(
    usecases: TodoUseCases = Depends(),
    completed: Optional[bool] = Query(None, alias="completed"),
    title: Optional[str] = Query(None, alias="title"),
    is_flagged: Optional[bool] = Query(None, alias="is_flagged"),
    due_date: Optional[date] = Query(None, alias="due_date"),
):
    if completed is not None:
        todos = await usecases.get_todos_by_status(completed)
    elif title is not None:
        todos = await usecases.get_todos_by_title(title)
    elif is_flagged is not None:
        todos = await usecases.get_todos_by_is_flagged(is_flagged)
    elif due_date is not None:
        todos = await usecases.get_todos_by_due_date(due_date)
    else:
```

```
        todos = await usecases.get_all_todos()
    return todos


@router.get("/{todo_id}", status_code=status.HTTP_200_OK, response_model=TodoSchema)
async def get_todo(todo_id: str, usecases: TodoUseCases = Depends()):
    todo = await usecases.get_todo(todo_id)
    if todo is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Tarefa não encontrada")
    return todo


@router.put("/{todo_id}", status_code=status.HTTP_200_OK, response_model=TodoSchema)
async def update_todo(todo_id: str, todo: TodoSchema, usecases: TodoUseCases = Depends()):
    updated_todo = await usecases.update_todo(todo_id, todo)
    return updated_todo


@router.delete("/{todo_id}", status_code=status.HTTP_204_NO_CONTENT)
async def delete_todo(todo_id: str, usecases: TodoUseCases = Depends()):
    await usecases.delete_todo(todo_id)
    return JSONResponse(status_code=status.HTTP_204_NO_CONTENT, content="Tarefa deletada")


@router.post("/{todo_id}/sub-tasks", status_code=status.HTTP_201_CREATED, response_model=SubTodoSchema)
async def create_sub_task(todo_id: str, sub_task: SubTodoSchema, usecases: TodoUseCases = Depends()):
    new_sub_task = await usecases.create_sub_task(todo_id, sub_task)
    return new_sub_task
```

Neste arquivo, definimos os endpoints da API utilizando o FastAPI para gerenciar as tarefas. Cada endpoint corresponde a uma ação específica do usuário, como criar, ler, atualizar ou deletar uma tarefa:

- **`POST /todos`:** Cria uma nova tarefa.
- **`GET /todos`:** Lista todas as tarefas, com filtros opcionais: completed, title, is_flagged, due_date.
- **`GET /todos/{todo_id}`:** Busca uma tarefa específica pelo seu ID.
- **`PUT /todos/{todo_id}`:** Atualiza uma tarefa específica pelo seu ID.
- **`DELETE /todos/{todo_id}`:** Deleta uma tarefa específica pelo seu ID.
- **`POST /todos/{todo_id}/sub-tasks`:** Cria uma subtarefa para uma tarefa específica pelo seu ID.

Utilizando os Use Cases:

Observe que os controllers utilizam a dependência `usecases: TodoUseCases = Depends()`, que é uma instância da classe `TodoUseCases`, responsável por executar as regras de negócio (usecases). Essa abordagem garante a separação de responsabilidades entre controllers e usecases, tornando o código mais organizado e fácil de manter.

Gerenciando as Interações do Usuário:

Os controllers são a interface entre a API e o usuário, traduzindo as requisições do usuário para as ações correspondentes no backend do nosso aplicativo. Eles garantem que as interações do usuário sejam processadas de forma eficiente e segura.

No próximo capítulo, vamos conectar os controllers à nossa aplicação principal e finalmente executar o nosso app TO-DO!

Iniciando a Aventura: Executando o Nosso App TO-DO

Agora que construímos todos os componentes do nosso app TO-DO, é hora de conectá-los e finalmente executar a nossa aplicação!

1. Criando o Arquivo `routers.py`:

Crie um arquivo chamado `routers.py` na raiz do seu projeto:

```
# todo_project/routers.py

from fastapi import APIRouter

from todo_project.controllers.todo import router as todo_router

api_router = APIRouter()
api_router.include_router(todo_router)
```

Neste arquivo, definimos um router principal (`api_router`) que irá incluir o router do controller de tarefas (`todo_router`). Essa organização facilita a manutenção e a expansão da nossa aplicação, permitindo que adicionemos novos controllers de forma modular.

2. Configurando o Arquivo `main.py`:

Agora, vamos configurar o arquivo `main.py` para iniciar a nossa aplicação:

```
# todo_project/main.py

from fastapi import FastAPI

from todo_project.routers import api_router

app = FastAPI(title="Todo Project", version="0.1.0")
app.include_router(api_router)
```

Neste arquivo, definimos a aplicação FastAPI (`app`) e incluímos o router principal (`api_router`),

conectando os controllers à nossa aplicação.

3. Executando a Aplicação:

Para executar o nosso app TO-DO, abra o terminal, navegue até a pasta do seu projeto e execute o seguinte comando:

```
poetry run uvicorn main:app --reload
```

Use code with caution.Bash

O comando `uvicorn` inicia um servidor web que irá executar a nossa aplicação FastAPI. A opção `--reload` garante que o servidor seja reiniciado automaticamente quando você salvar alterações no código, facilitando o desenvolvimento.

Pronto para Testar!

Agora, abra seu navegador e acesse a URL `http://127.0.0.1:8000/docs`. Você verá a documentação da sua API, com informações sobre os endpoints disponíveis, os parâmetros que podem ser utilizados e os exemplos de requisições e respostas.

Próximos Passos:

Com a aplicação em execução, você pode utilizar ferramentas como o Postman para realizar testes manuais e verificar o funcionamento de cada endpoint. No próximo capítulo, vamos explorar técnicas de testes automatizados para garantir a qualidade do código e a funcionalidade da nossa aplicação.

Parabéns por ter chegado até aqui! Você construiu um app TO-DO completo e funcional utilizando Python, FastAPI e MongoDB. Agora, explore as funcionalidades da sua aplicação, adicione novos recursos e personalize-o para atender às suas necessidades.

Testando em Campo: Utilizando o Postman para Criar uma Tarefa

Agora que o nosso app TO-DO está funcionando, vamos colocar ele à prova! Para testar os endpoints da nossa API, vamos utilizar o Postman, uma ferramenta poderosa e intuitiva para realizar requisições HTTP.

1. Baixando e Instalando o Postman:

- Acesse o site oficial do Postman (https://www.postman.com/) e baixe a versão do Postman para o seu sistema operacional.
- Instale o Postman e abra a aplicação.

2. Criando uma Nova Requisição:

- Na tela principal do Postman, clique no botão "New" e selecione "Request".
- No campo "Enter request URL", digite a URL do endpoint para criar uma nova tarefa. No nosso caso, a URL é: `http://127.0.0.1:8000/todos`.

3. Definindo o Método HTTP:

- Selecione o método HTTP "POST", pois queremos enviar uma requisição para criar uma nova tarefa.

4. Adicionando o Corpo da Requisição:

- Clique na aba "Body" e selecione o tipo de corpo da requisição como "raw".
- Selecione o tipo de conteúdo como "JSON".
- Preencha o corpo da requisição com os dados da nova tarefa, no formato JSON. Por exemplo:

```
{
 "title": "Criar um novo capítulo para o livro",
 "note": "",
 "url": "",
 "due_date": "",
 "due_time": "",
 "priority": "none",
 "is_flagged": false,
 "sub_tasks": []
}
```

5. Enviando a Requisição:

- Clique no botão "Send" para enviar a requisição para o servidor.

6. Verificando a Resposta:

- Se a requisição for bem-sucedida, você verá a resposta do servidor na aba "Body". A resposta deve incluir os dados da nova tarefa que foi criada, incluindo o ID da tarefa.
- Se a requisição falhar, você verá uma mensagem de erro na aba "Body".

Testando Outros Endpoints:

Você pode seguir os mesmos passos para testar os outros endpoints da API, como:

- **`GET /todos`:** Para listar todas as tarefas.
- **`GET /todos/{todo_id}`:** Para buscar uma tarefa específica pelo seu ID.
- **`PUT /todos/{todo_id}`:** Para atualizar uma tarefa específica pelo seu ID.
- **`DELETE /todos/{todo_id}`:** Para deletar uma tarefa específica pelo seu ID.
- **`POST /todos/{todo_id}/sub-tasks`:** Para criar uma subtarefa para uma tarefa específica pelo seu ID.

O Postman é uma ferramenta essencial para testar APIs

O Postman oferece uma interface amigável e intuitiva para enviar requisições HTTP, visualizar as respostas do servidor e realizar testes manuais de forma eficiente. Utilizando o Postman, você pode garantir que os endpoints da sua API estejam funcionando corretamente e que a aplicação esteja respondendo como esperado.

O Fim da Nossa Aventura (Por Enquanto...): Um Novo Começo para Sua Jornada!

Chegamos ao fim da nossa jornada na construção do app TO-DO! Esperamos que você tenha se divertido e aprendido muito durante essa aventura. Apesar dos desafios que encontramos pelo caminho, acreditamos que você conquistou uma nova habilidade: desenvolver aplicações web com Python, FastAPI e MongoDB.

Seu App TO-DO Funcional:

Com o código que desenvolvemos, você tem um app TO-DO completo e funcional, pronto para te ajudar a organizar suas tarefas e aumentar sua produtividade.

O Começo de Uma Nova História:

Este livro te deu as ferramentas e os conhecimentos básicos para dar seus primeiros passos na criação de aplicações web. Agora, você está pronto para explorar novas ideias, construir novos projetos e aprimorar a sua aplicação TO-DO com recursos extras, como um frontend para tornar sua interface mais intuitiva e visualmente atraente.

Desafios e Soluções:

Caso você encontre algum erro ou dificuldade ao construir a aplicação, lembre-se que a internet é um oceano de conhecimento! Utilize o Google Gemini, uma poderosa ferramenta de inteligência artificial, para pesquisar soluções, entender conceitos complexos e encontrar respostas para suas dúvidas.

A Jornada Continua:

O mundo da programação é vasto e cheio de oportunidades. Explore novas tecnologias, experimente novos frameworks e bibliotecas e continue aprendendo. A cada desafio que você superar, você estará construindo um futuro brilhante como desenvolvedor.

Obrigado por embarcar nessa jornada conosco!