

# Introdução ao FastAPI

Aprendendo a construir APIs REST com Python





# Agenda

→ Contextualização teórica

- ◆ HTTP
- ◆ Cliente x Servidor
- ◆ API
- ◆ API REST
- ◆ CRUD
- ◆ Framework

→ Projeto prático



## Contextualização teórica

O embasamento necessário  
para construir uma API

# O que é HTTP?

O Protocolo HTTP (Hypertext Transfer Protocol) é um protocolo de comunicação utilizado para a transferência de dados entre clientes e servidores.

É a base da comunicação na web, permitindo que navegadores e servidores troquem informações de forma eficiente e padronizada.



# Como funciona o HTTP

**Cliente:** envia uma solicitação HTTP para o servidor - uma requisição.

**Requisição:** é uma mensagem padronizada que contém a descrição do que está sendo pedido pelo cliente para o servidor

**Servidor:** processa a solicitação e envia uma resposta HTTP de volta ao cliente.

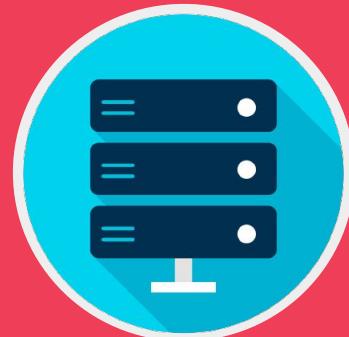
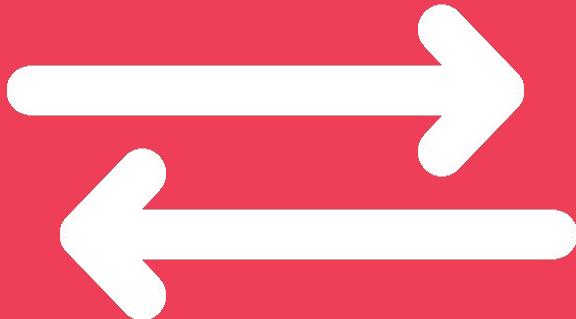
**Resposta:** é também uma mensagem padronizada, que contém:

- Dados solicitados pelo cliente
- ou
- "parecer" do servidor, caso aquele pedido não tenha sido atendido

**Requisição**



**Computador/Celular**



**Servidor**

**Resposta**



# Componentes do HTTP

Toda requisição HTTP possui  
três partes

- **Métodos:** indicam a ação desejada
- **Headers:** contêm informações sobre a solicitação ou resposta
- **Body:** contém os dados a serem enviados ou recebidos;



# Métodos HTTP

Os métodos mais comuns

- **GET**: solicita dados do servidor
- **POST**: envia dados ao servidor
- **PUT**: atualiza dados existentes no servidor
- **DELETE**: remove dados do servidor



# Métodos HTTP

Outros métodos menos utilizados

- **PATCH**: altera um recurso parcialmente
- **HEAD**: requisita apenas os cabeçalhos da resposta, sem o corpo
- **OPTIONS**: requisita os métodos que o recurso aceita
- **TRACE**: ecoa a requisição, para verificar o que cada servidor altera nela



# Códigos de Status HTTP

Os códigos de status são divididos em categorias

- **1xx: Informacional**
- **2xx: Sucesso**
- **3xx: Redirecionamento**
- **4xx: Erro do Cliente**
- **5xx: Erro do Servidor**



# Códigos de Status HTTP

Os códigos mais comuns

- **200: OK**
- **201: Criado**
- **400: Requisição com problema**
- **401: Não autorizado**
- **404: Não encontrado**
- **500: Erro interno do servidor**

## Cliente x Servidor

**Cliente:** é um dispositivo com acesso à internet que faz **solicitações** de informações. Ele vai interpretar a informação recebida e exibir de uma maneira que seja legível para nós.

**Servidor:** é um computador que contém as informações e **responde** essas solicitações do cliente. Esse computador pode estar localizado em qualquer lugar do mundo.

# Aplicação do Minicurso

No contexto de aplicações web, o que queremos construir neste minicurso é uma aplicação tipo servidor, algo que fornecerá informações para serem consumidas por clientes.

## Aviso



Como esse curso não abrange deploy, nossa aplicação não poderá efetivamente servir conteúdos para serem acessados por clientes externos, pois ela estará restrita à nossa rede local.

# API

APIs (Application Programming Interface) são interfaces para que seja possível interagir com recursos.

Uma API possui regras para definir como essa interação será feita, especificando os tipos de requisições e quais dados devem estar presentes para que o servidor consiga retornar os dados necessários.

# API REST

O **REST** (Representational State Transfer) é um estilo de arquitetura amplamente utilizado na modelagem de APIs. Alguns dos **princípios REST** são:

- **Client-server:** separação de papéis
- **Stateless:** cada requisição independe de requisições anteriores
- **Interface uniforme:** a identificação e manipulação de recursos segue um padrão

Quando uma API implementa todos os **princípios REST**, ela é considerada uma **API RESTful**.

# CRUD

É um acrônimo que representa **quatro operações básicas** realizadas sobre informações: Criar (Create), Ler (Read), Atualizar (Update) e Deletar (Delete).

- **C** - **Criar**: adicionar novos dados
- **R** - **Ler**: recuperar e exibir informações
- **U** - **Atualizar**: modificar dados que já estão salvos no servidor
- **D** - **Deletar**: excluir dados



## CRUD x HTTP

Existe uma identificação entre os elementos do CRUD e os verbos HTTP, estabelecida pelo padrão de arquitetura REST.

→ C - Criar: **POST**

→ R - Ler: **GET**

→ U - Atualizar: **PUT**

→ D - Deletar: **DELETE**

# Framework

**É uma estrutura que fornece um conjunto de ferramentas e componentes reutilizáveis para simplificar e acelerar o desenvolvimento de software.**

**Permite que você foque no seu objetivo, abstraindo os procedimentos mais técnicos.**

# FastAPI

**FastAPI é um framework web Python bastante moderno, usado para criação de APIs.**

**Foi criado por Sebastian Ramírez em 2018, e tem se tornado muito popular nos últimos anos graças à sua arquitetura robusta e escalável, e à simplicidade de sua sintaxe.**



Vamos aprender na prática

Construindo uma API de Mulheres Cientistas



# Contexto do Projeto

# Contexto do Projeto

## Catálogo de Mulheres Cientistas:

- Nome
- Descrição
- Data de Nascimento
- Data de Morte (opcional)
- Link para foto
- Link para artigo da Wikipedia
- Frases famosas ditas pela cientista

# API de Mulheres Cientistas 0.1.0 OAS 3.1

/openapi.json

## Cientistas

- `GET /cientistas` Get Cientistas
- `POST /cientistas` Create Cientista
- `GET /cientistas/{cientista_id}` Get Cientista By Id
- `PUT /cientistas/{cientista_id}` Update Cientista
- `DELETE /cientistas/{cientista_id}` Delete Cientista

## Frases

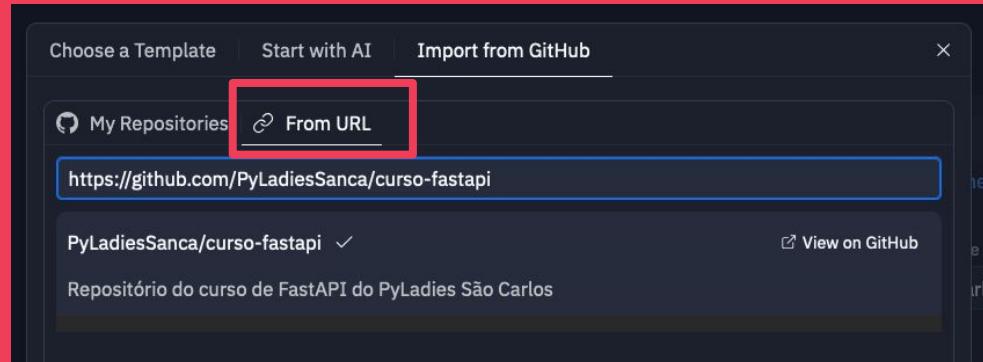
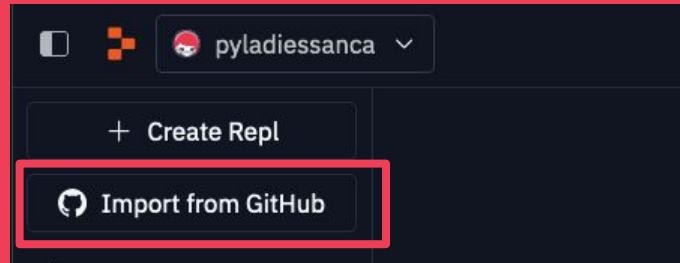
- `GET /frases` Get Frases
- `POST /frases` Create Frase
- `GET /frases/{frase_id}` Get Frase By Id
- `PUT /frases/{frase_id}` Update Frase
- `DELETE /frases/{frase_id}` Delete Frase

## default

- `GET /` Index

# Preparando o Ambiente de Desenvolvimento

- Abrir o Replit (<https://replit.com>) e criar uma conta, caso não tenha
- Escolher o plano gratuito (pode criar até 3 ambientes)
- Clicar em Import from GitHub e ir para a aba From URL





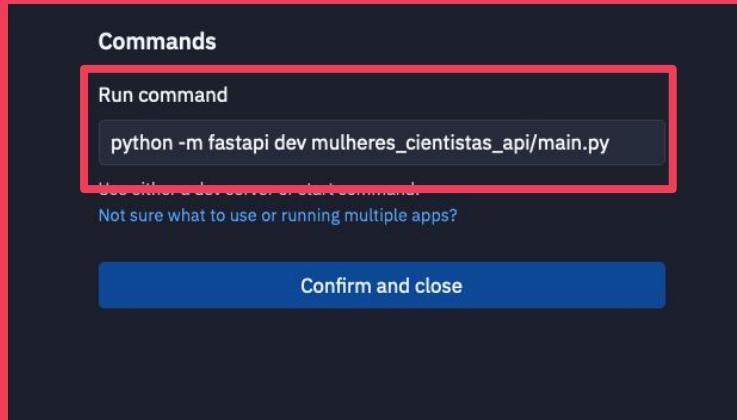
# Repositório:

<https://github.com/PyLadiesSanca/curso-fastapi>

# Preparando o Ambiente de Desenvolvimento

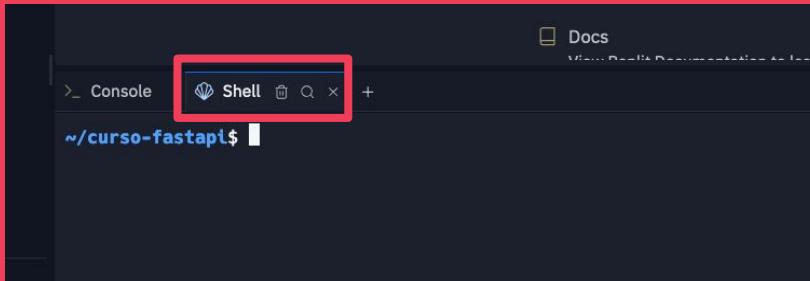
→ Defina o comando para executar a aplicação:

◆ `python -m fastapi dev mulheres_cientistas_api/main.py`



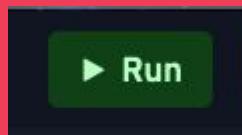
# Preparando o Ambiente de Desenvolvimento

- Abra a guia **Shell**:



- Digite o comando para instalar as dependências do Python:  
◆ **poetry install**

- Clique no botão **Run**:



# Preparando o Ambiente de Desenvolvimento

→ Abra a guia **Console**:

The screenshot shows a terminal window with the following interface elements:

- Tab Bar:** Shows tabs for "Console" (highlighted with a red box), "Shell", and a "+" button.
- Status Bar:** Shows "Show Only Latest" and "Clear History" buttons.
- Run Button:** A small downward arrow icon.
- Message Bar:** Displays the message "Port :8000 opened, but not exposed to the web."
- Toolbar:** Includes "Ask AI", "....replit.dev", port "8000", timestamp "2m on 17:10:53, 09/01", and a gear icon (highlighted with a red box).
- Text Area:** Displays log output from the application.

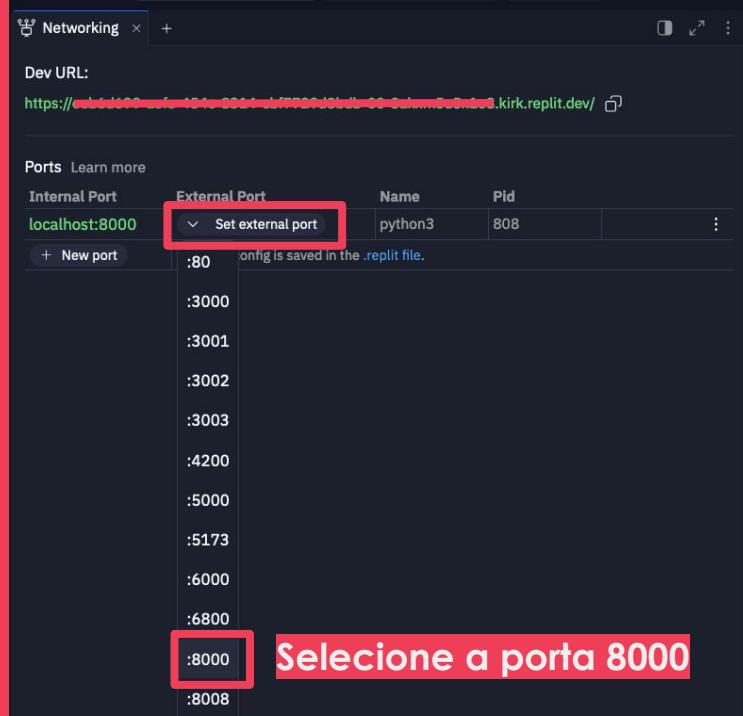
```
INFO    Using path mulheres_cientistas_api/main.py
INFO    Resolved absolute path /home/runner/curso-fastapi/mulheres_cientistas_api/main.py
INFO    Searching for package file structure from directories with __init__.py files
INFO    Importing from /home/runner/curso-fastapi
```
- File Structure Diagram:** A box labeled "Python package file structure" containing:

```
mulheres_cientistas_api
└── __init__.py
└── main.py
```

A large red box highlights the message "Port :8000 opened, but not exposed to the web." and the gear icon in the toolbar. Another red box highlights the text "Clique na engrenagem para expor a porta 8000".



# Preparando o Ambiente de Desenvolvimento

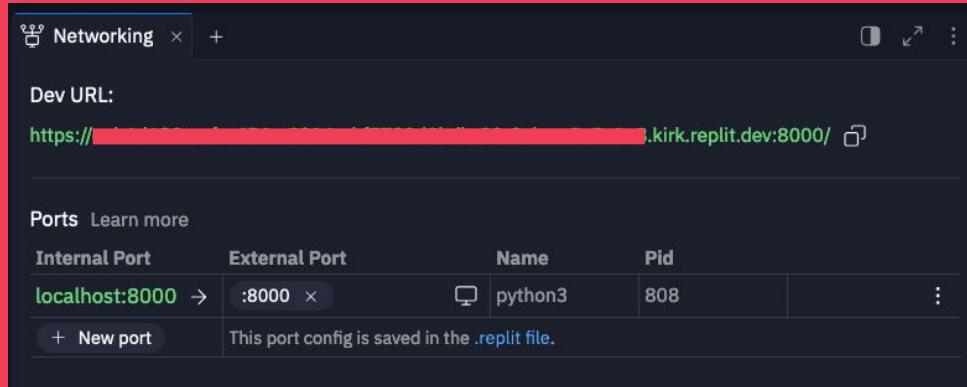


The screenshot shows the Replit Networking interface. At the top, it displays a Dev URL: [https://\[REDACTED\].kirk.replit.dev/](https://[REDACTED].kirk.replit.dev/). Below this, there's a table titled "Ports" with columns: Internal Port, External Port, Name, and Pid. A row for "localhost:8000" is selected, and its "External Port" field is highlighted with a red box. The dropdown menu in this field is open, showing options from ":80" to ":8008". The option ":8000" is also highlighted with a red box. A large red callout box covers the bottom right of the table area, containing the text "Selecione a porta 8000".

Internal Port	External Port	Name	Pid	
localhost:8000	:8000	python3	808	
+ New port	:80	onfig is saved in the .replit file.		
	:3000			
	:3001			
	:3002			
	:3003			
	:4200			
	:5000			
	:5173			
	:6000			
	:6800			
	:8000			
	:8008			

# Preparando o Ambiente de Desenvolvimento

→ Clique na URL de desenvolvimento para abrir a API no navegador:



# Preparando o Ambiente de Desenvolvimento



- O esperado é aparecer uma página como essa:

```
JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON
title: "API de Mulheres Cientistas"
```

- Adicione `/docs` no final da URL para acessar a documentação interativa do **FastAPI**:

API de Mulheres Cientistas 0.1.0 OAS 3.1  
[/openapi.json](#)

default

GET / Index



# Entendendo a Estrutura do Projeto



```
▽ CURSO-FASTAPI
  ▽ mulheres_cientistas_api
    __init__.py ←
    config.py
    database.py
    main.py
    .env
    .gitignore
    exemplos.json
    mulheres_cientistas.db
    poetry.lock
    pyproject.toml
    README.md
```

# Estrutura do Projeto

## `__init__.py`

- **Transforma a pasta `mulheres_cientistas_api` em um módulo**
- **Permite que os arquivos da pasta dentro dela possam ser importados**
- **Não possui nenhum conteúdo**



```
▽ CURSO-FASTAPI
  ▽ mulheres_cientistas_api
    __init__.py
    config.py ←
    database.py
    main.py
    .env
    .gitignore
    exemplos.json
    mulheres_cientistas.db
    poetry.lock
    pyproject.toml
    README.md
```

# Estrutura do Projeto

## config.py

- Módulo para configurações do projeto
- Nesse projeto, a única configuração relevante é o nome do banco de dados
- Importa as configurações do .env
- Utiliza a biblioteca Pydantic para validação dos dados



# config.py

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    DATABASE_NAME: str Variável de ambiente do tipo str

    model_config = SettingsConfigDict(env_file=".env")
                Lê os dados do arquivo .env

settings = Settings()
Permite importar a env var:
from config import settings
```



```
▽ CURSO-FASTAPI
  ▽ mulheres_cientistas_api
    __init__.py
    config.py
    database.py ←
    main.py
    .env
    .gitignore
    exemplos.json
    mulheres_cientistas.db
    poetry.lock
    pyproject.toml
    README.md
```

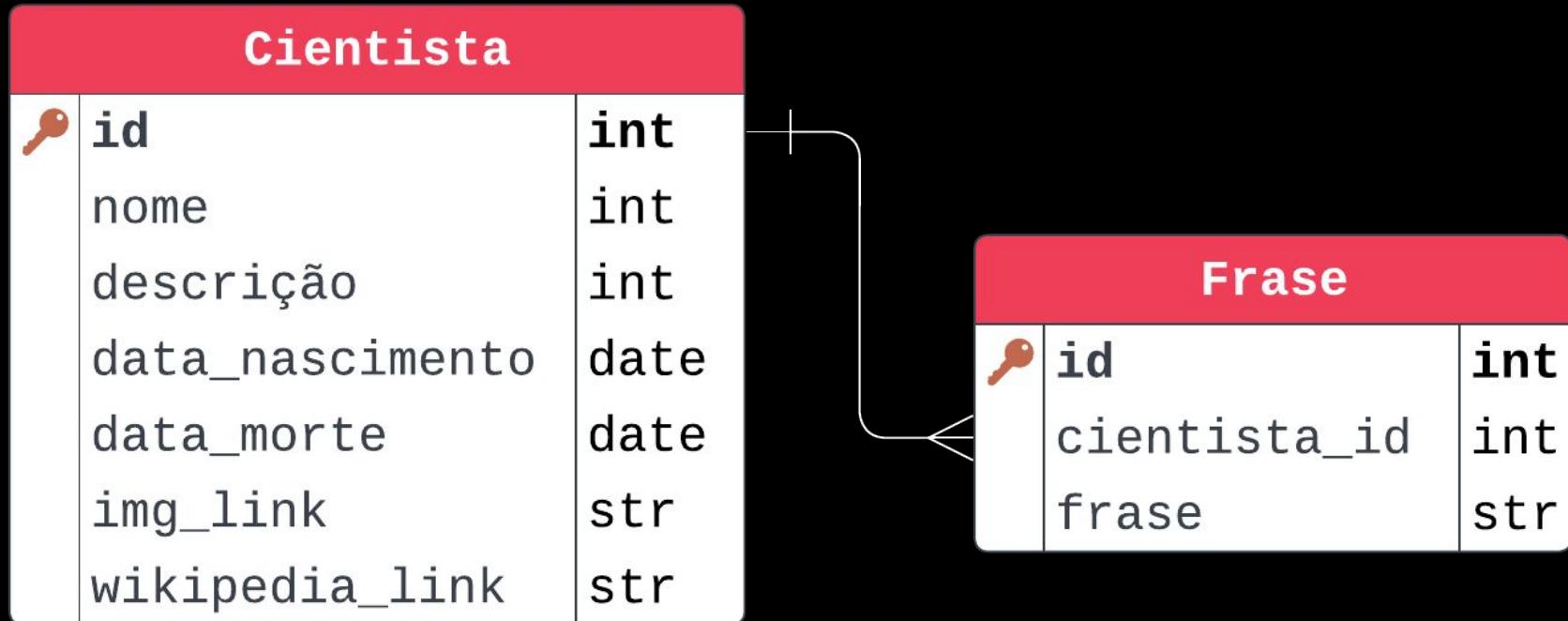
# Estrutura do Projeto

## database.py

- Módulo para gerenciamento do banco de dados
- Responsável por gerir a engine e as sessions
- Utiliza as bibliotecas SQLModel e SQLAlchemy
- Banco de dados em memória (SQLite3)



# Esquema do Banco de Dados



# database.py



```
from sqlalchemy import event
from sqlmodel import Session, SQLModel, create_engine

from mulheres_cientistas_api.config import settings

def _fk_constraint_on(dbapi_con, con_record):
    dbapi_con.execute("PRAGMA foreign_keys = ON")
Permite a utilização de Foreign Keys

engine = create_engine(f"sqlite:///{settings.DATABASE_NAME}.db", echo=True)
event.listen(engine, "connect", _fk_constraint_on)

Cria a engine de conexão com o banco
```



# database.py

```
def init_db():
    SQLModel.metadata.create_all(engine)
```

**Cria todas as tabelas declaradas nos Modelos**

**Gera uma sessão de**

**acesso ao banco de dados**

```
def get_session():
    with Session(engine) as session:
```

**try:**

**yield session**

**session.commit()**

```
except Exception as e:
```

**Salva os dados quando  
o acesso é finalizado**

**session.rollback()**

**raise e**

**finally:**

**session.close()**

**Fecha a conexão**

**Cancela a operação se alguma  
exceção for levantada**



```
✓ CURSO-FASTAPI
  └── mulheres_cientistas_api
      ├── __init__.py
      ├── config.py
      ├── database.py
      └── main.py ←
  ├── .env
  ├── .gitignore
  ├── exemplos.json
  ├── mulheres_cientistas.db
  ├── poetry.lock
  ├── pyproject.toml
  └── README.md
```

# Estrutura do Projeto

## main.py

- Ponto de entrada do projeto
- Define a aplicação do FastAPI
- Define a rota inicial /
- Inicia o banco de dados
- Importa as demais rotas do projeto

# main.py



```
from contextlib import asynccontextmanager
from fastapi import FastAPI
from mulheres_cientistas_api.database import init_db

@asynccontextmanager
async def lifespan(_: FastAPI):
    init_db() Inicia o banco de dados junto com a aplicação
    yield

App do FastAPI
app = FastAPI(
    title="API de Mulheres Cientistas",
    lifespan=lifespan
)

@app.get("/")
Rota inicial da aplicação
def index():
    return {"title": "API de Mulheres Cientistas"}
```



# Implementação: Modelos

# O que são os Modelos da aplicação?

Para interagir com o banco de dados de forma otimizada, utiliza-se um **ORM (Mapeamento Objeto-Relacional)**.

O ORM mapeia uma **tabela** do banco de dados para uma **classe** no Python, onde **cada objeto da classe representa um registro na tabela**.

Os **modelos** são a representação desse mapeamento, e podem ser definidos de diversas formas.

# Diferentes modos de criar modelos no FastAPI

Modo “padrão” – mais comum, porém mais trabalhoso para manter:

- Modelo do SQLAlchemy
  - ◆ Necessário para interação com o banco de dados
- Modelo/Esquema do Pydantic
  - ◆ Necessário para validação dos dados no FastAPI

Os campos de ambos os modelos são os mesmos, gerando duplicidade do código.

Modo SQLModel – biblioteca do criador do FastAPI e feita para facilitar essa etapa:

- Cria modelo do SQLModel
  - ◆ É ao mesmo tempo um modelo do SQLAlchemy e do Pydantic

O mesmo modelo é utilizado para interação com o banco de dados e para validação do FastAPI



# Cientista

Nosso primeiro modelo

Cientista		
🔑	<b>id</b>	<b>int</b>
	nome	int
	descrição	int
	data_nascimento	date
	data_morte	date
	img_link	str
	wikipedia_link	str



```
▽ CURSO-FASTAPI
  ▽ mulheres_cientistas_api
    __init__.py
    config.py
    database.py
    main.py
    models.py ←
    .env
    .gitignore
    exemplos.json
    mulheres_cientistas.db
    poetry.lock
    pyproject.toml
    README.md
```

# Crie o arquivo

**models.py**

# models.py



```
from datetime import date
from sqlmodel import Field, SQLModel
```

```
class CientistaBase(SQLModel): Base comum para os modelos
```

```
    nome: str = Field(index=True, nullable=False)
    descrição: str = Field(nullable=False)
    data_nascimento: date = Field(nullable=False)
    data_morte: date | None = Field(default=None)
    img_link: str = Field(nullable=False)
    wikipedia_link: str = Field(nullable=False)
```

```
class Cientista(CientistaBase, table=True): Define a tabela do banco
    id: int | None = Field(default=None, primary_key=True)
```

# models.py



```
class CientistaCreate(CientistaBase):
    pass
```

**Para criar uma Cientista utiliza-se  
somente os dados do Base**

```
class CientistaPublic(CientistaBase):
    id: int
```

**Para exibir uma Cientista  
acrescenta-se o id**

```
class CientistaUpdate(SQLModel):
    nome: str | None = Field(default=None) Todos os dados são opcionais  
ao atualizar uma Cientista
    descrição: str | None = Field(default=None)
    data_nascimento: date | None = Field(default=None)
    data_morte: date | None = Field(default=None)
    img_link: str | None = Field(default=None)
    wikipedia_link: str | None = Field(default=None)
```



# Implementação: CRUD

# O que é o módulo CRUD?

Para manipular os objetos criados a partir dos nossos modelos, é preciso interagir com o banco de dados através do **ORM**.

O módulo **CRUD** implementa as 4 operações utilizando o **SQLAlchemy**.

É através dos métodos deste módulo que nossa API irá se comunicar com o banco.

Note que nossa aplicação terá **dois modelos** (Cientista e Frase), portanto nosso **CRUD** será **genérico** para ser utilizado por ambos os modelos.



```
✓ CURSO-FASTAPI
  ✓ mulheres_cientistas_api
    __init__.py
    config.py
    crud.py ←
    database.py
    main.py
    models.py
    .env
    .gitignore
    exemplos.json
    poetry.lock
    pyproject.toml
    README.md
```

# Crie o arquivo

**crud.py**



# CRUD – Create

Etapas para a criação de um  
objeto no banco de dados

## Etapas

- Função `create()` recebe os dados no formato do modelo `CientistaCreate`
  - ◆ Possui todos os dados **exceto o id**
- **Converte os dados pro modelo Cientista**
  - ◆ Definição da nossa `tabela`
- **Insere os dados no banco**
- **Atualiza o objeto para pegar o id**
- **Retorna o registro no formato Cientista**
  - ◆ Posteriormente será **convertido no formato CientistaPublic para exibição no FastAPI**)

# crud.py – Create



```
from typing import Optional, Type, TypeVar  
from sqlmodel import Session, SQLModel, select
```

```
T = TypeVar("T", bound=SQLModel)
```

**Define um tipo genérico  
vinculado aos nossos modelos**

```
def create(session: Session, model: Type[T], data: T) -> T:  
    obj = model(**data.model_dump())  
    session.add(obj)  
    session.flush()  
    session.refresh(obj)  
    return obj
```

Classe

Objetos da classe

**Desempacota os dados para criar uma  
instância da classe do modelo**



# crud.py – Create

```
from typing import Optional, Type, TypeVar
from sqlmodel import Session, SQLModel, select

T = TypeVar("T", bound=SQLModel)

def create(session: Session, model: Type[T], data: T) -> T:
    obj = model(**data.model_dump())
    session.add(obj)
    session.flush()
    session.refresh(obj)
    return obj
```

Adiciona o objeto na sessão

Envia para o banco os objetos da sessão

Atualiza o objeto com os dados salvos no banco (id, por exemplo)



# CRUD – Read All

Etapas para a leitura dos registros no banco de dados

## Etapas

- Função `get_all()` constrói uma query para realizar uma operação no banco
  - ◆ Seleciona todos os dados da tabela
- Executa a query para pegar os dados da tabela Cientista
- Retorna todos os registros encontrados pela query

# crud.py – Get All



```
def get_all(session: Session, model: Type[T]) -> list[T]:  
    query = select(model)  
    return session.exec(query).all()
```



# CRUD – Read One

Etapas para a leitura dos registros no banco de dados

## Etapas

- Função `get_one()` constrói uma query para realizar uma operação no banco
  - ◆ Seleciona o registro cujo id é o mesmo do parâmetro recebido
- Executa a query para procurar o dado na tabela `Cientista`
- Retorna no formato `Cientista`:
  - ◆ O único registro com o id especificado
  - ◆ `None` caso não exista

# crud.py – Get One



```
def get_one(session: Session, model: Type[T], id: int) -> Optional[T]:  
    query = select(model).where(model.id == id)  
    return session.exec(query).one_or_none()
```



# CRUD – Update

Etapas para a atualizar um registro no banco de dados

## Etapas

- Função `update()` constrói uma query para realizar uma operação no banco
  - ◆ Seleciona o registro cujo id é o mesmo do parâmetro recebido
- Executa a query para procurar o registro na tabela `Cientista`
- Caso não encontre, retorna `None`
- Chama a função `sqlmodel_update()` com os dados especificados no objeto
  - ◆ Ignora os dados faltantes
- Atualiza o objeto para pegar o id
- Retorna o registro no formato `Cientista`

# crud.py – Update



```
def update(session: Session, model: Type[T], id: int, data: T) -> Optional[T]:  
    query = select(model).where(model.id == id)  
    obj = session.exec(query).one_or_none()  
  
    if not obj:  
        return None  
  
    obj.sqlmodel_update(data.model_dump(exclude_unset=True))  
  
    session.flush()  
    session.refresh(obj)  
    return obj
```

Modifica os campos do objeto do banco de dados com os dados especificados no modelo de update



# CRUD – Delete

Etapas para a remover um registro do banco de dados

## Etapas

- Função `delete()` constrói uma query para realizar uma operação no banco
  - ◆ Seleciona o registro cujo id é o mesmo do parâmetro recebido
- Executa a query para procurar o registro na tabela Cientista
- Caso não encontre, retorna False
- Utiliza o método `delete()` da sessão para marcar o registro como deletado
- Atualiza a sessão para persistir no banco a operação realizada
- Retorna True caso tenha dado certo

# crud.py – Delete



```
def delete(session: Session, model: Type[T], id: int) -> bool:
    query = select(model).where(model.id == id)
    obj = session.exec(query).one_or_none()

    if not obj:
        return False

    session.delete(obj)
    session.flush()
    return True
```



# Implementação: Routers

# O que são Routers?

As rotas da API definem o caminho onde a requisição HTTP é processada.

No FastAPI cada rota é associada com uma função que recebe a requisição, processa os dados, e retorna o Status Code com o corpo da resposta (caso haja).

Será dentro das rotas que iremos utilizar as funções definidas no módulo CRUD.

Para organizar as rotas, iremos criar uma coleção de rotas (Router), para que o código fique organizado e modular.





```
~ CURSO-FASTAPI
  ~ mulheres_cientistas_api
    ~ routers
      __init__.py
      cientista.py ←
      __init__.py
      config.py
      database.py
      main.py
      models.py
    .env
    .gitignore
    exemplos.json
    mulheres_cientistas.db
    poetry.lock
    pyproject.toml
    README.md
```

# Crie a pasta

→ /routers

# Crie os arquivos

→ \_\_init\_\_.py  
→ cientista.py



## Router – Create

Etapas para criar uma cientista através de uma rota

### Etapas

- Função `create_cientista()` recebe os dados no formato do modelo `CientistaCreate` da requisição e a sessão do banco de dados via `injeção de dependência`
- Os dados recebidos e a sessão do banco de dados são passados para a função `create()` do módulo `CRUD`
- Retorna o status code `201 (Created)` e o objeto criado pelo `CRUD` no formato `CientistaPublic`
  - ◆ O FastAPI faz a conversão do modelo `Cientista` para o `CientistaPublic`

# routers/cientista.py – create\_cientista



```
from fastapi import APIRouter, Depends, HTTPException
from sqlmodel import Session

from mulheres_cientistas_api.crud import create, delete, get_all, get_one, update
from mulheres_cientistas_api.database import get_session
from mulheres_cientistas_api.models import (
    Cientista,
    CientistaCreate,
    CientistaPublic,
    CientistaUpdate,
)
router = APIRouter(prefix="/cientistas", tags=["Cientistas"])

@router.post("", response_model=CientistaPublic, status_code=201)
def create_cientista(
    cientista_to_create: CientistaCreate, session: Session = Depends(get_session)
):
    return create(session=session, model=Cientista, data=cientista_to_create)
```

Define o caminho base das rotas do router

Identificação na página /docs



# routers/cientista.py – create\_cientista

```
from fastapi import APIRouter, Depends, HTTPException
from sqlmodel import Session

from mulheres_cientistas_api.crud import create, delete, get_all, get_one, update
from mulheres_cientistas_api.database import get_session
from mulheres_cientistas_api.models import (
    Cientista,
    CientistaCreate,
    CientistaPublic,
    CientistaUpdate,
)
router = APIRouter(prefix="/cientistas", tags=["Cientistas"])

Caminho da rota: POST: /cientistas

@router.post("", response_model=CientistaPublic, status_code=201)
def create_cientista(
    cientista_to_create: CientistaCreate, session: Session = Depends(get_session)
):
    return create(session=session, model=Cientista, data=cientista_to_create)
```

Injeta a sessão em cada  
requisição feita nessa rota





## Router – Get All

Etapas para a ler todos os  
cientistas através de uma rota

### Etapas

- Função `get_cientistas()` recebe a sessão do banco de dados via injeção de dependência
- A função `get_all()` do módulo `CRUD` é chamada
- Retorna o status code `200 (OK)` e uma lista com todos os objetos retornados pelo `CRUD` no formato `CientistaPublic`
  - ◆ O FastAPI faz a conversão do modelo `Cientista` para o `CientistaPublic`



## routers/cientista.py – get\_cientistas

Caminho da rota: GET: /cientistas

```
@router.get("", response_model=list[CientistaPublic], status_code=200)
def get_cientistas(session: Session = Depends(get_session)):
    return get_all(session=session, model=Cientista)
```



## Router – Get One

Etapas para a ler uma cientista específica através de uma rota com parâmetro de query

### Etapas

- Função `get_cientista_by_id()` recebe o `id` da cientista através da `requisição` e a `sessão` do banco de dados via `injeção de dependência`
- A função `get_one()` do módulo `CRUD` é chamada com o `id` e a `sessão`
- Caso a função `get_one()` retorne `None`, a resposta da requisição será `404` (`não encontrado`)
- Caso tenha encontrado, retorna o status code `200` (`OK`) e a cientista no formato `CientistaPublic`
  - ◆ O FastAPI faz a conversão do modelo `Cientista` para o `CientistaPublic`

# routers/cientista.py – get\_cientista\_by\_id



Caminho da rota: GET: /cientistas/1, caso o id seja 1

```
@router.get("/{cientista_id}", response_model=CientistaPublic, status_code=200)
def get_cientista_by_id(cientista_id: int, session: Session = Depends(get_session)):
    cientista = get_one(session=session, model=Cientista, id=cientista_id)
    if cientista is None:
        raise HTTPException(status_code=404, detail="Cientista não encontrada")
    return cientista
```



Exceção tratada pelo FastAPI



# Router – Update

Etapas para a atualizar uma cientista específica através de uma rota com parâmetro de query

## Etapas

- Função `update_cientista()` recebe o `id` da cientista e os dados para serem atualizados através da `requisição`, e a `sessão` do banco de dados via `injeção de dependência`
- A função `update()` do módulo `CRUD` é chamada com o `id`, os dados e a `sessão`
- Caso a função `update()` retorne `None`, a `resposta` da `requisição` será `404` (não encontrado)
- Caso tenha encontrado, os dados são atualizados e retornados no formato `CientistaPublic`
  - ◆ O `FastAPI` faz a conversão do modelo `Cientista` para o `CientistaPublic`

# routers/cientista.py – update\_cientista



Caminho da rota: PUT: /cientistas/1, caso o id seja 1

```
@router.put("/{cientista_id}", response_model=CientistaPublic, status_code=200)
def update_cientista(
    cientista_id: int,
    cientista_update: CientistaUpdate,
    session: Session = Depends(get_session),
):
    cientista = update(session=session, model=Cientista, id=cientista_id, data=cientista_update)
    if cientista is None:
        raise HTTPException(status_code=404, detail="Cientista não encontrada")
    return cientista
```



## Router – Delete

Etapas para a deletar uma cientista específica através de uma rota com parâmetro de query

### Etapas

- Função `delete_cientista()` recebe o `id` da cientista e os dados para serem atualizados através da `requisição`, e a `sessão` do banco de dados via `injeção de dependência`
- A função `delete()` do módulo `CRUD` é chamada com o `id` e a `sessão`
- Caso a função `delete()` retorne `False`, a resposta da requisição será `404 (não encontrado)`
- Caso tenha encontrado, os dados apagados
  - ◆ O retorno será `204 (No Content)` com o `body` vazio (`None`)

# routers/cientista.py – delete\_cientista



Caminho da rota: **DELETE: /cientistas/1, caso o id seja 1**

```
@router.delete("/{cientista_id}", status_code=204)
def delete_cientista(cientista_id: int, session: Session = Depends(get_session)):
    if not delete(session=session, model=Cientista, id=cientista_id):
        raise HTTPException(status_code=404, detail="Cientista não encontrada")
    return None
```



# Acessando a API

# Adicionando o Router no app do FastAPI

No arquivo `main.py` foi definido o `app` da aplicação FastAPI.

Para que as rotas definidas no arquivo `/routers/cientista.py` sejam acessadas, é necessário incluir o router na aplicação.



## Edite no arquivo

→ `/main:`

- ◆ Importe o módulo `cientista` da pasta `/routers`
- ◆ Chame a função `app.include_router()` passando o router do módulo como argumento



# Modificações - main.py

```
from contextlib import asynccontextmanager

from fastapi import FastAPI

from mulheres_cientistas_api.database import init_db
from mulheres_cientistas_api.routers import cientista
#-----#
@asynccontextmanager
async def lifespan(_: FastAPI):
    init_db()
    yield

app = FastAPI(
    title="API de Mulheres Cientistas",
    lifespan=lifespan
)
app.include_router(cientista.router)
#-----#
@app.get("/")
def index():
    return {"title": "API de Mulheres Cientistas"}
```



# Acessando a API

- Clique na URL de desenvolvimento e adicione `/docs` no final

The screenshot shows the Networking tab in the Replit interface. It displays the Dev URL as `https://[REDACTED].kirk.replit.dev:8000/`. Below it, a table lists port mappings:

Internal Port	External Port	Name	Pid	⋮
localhost:8000	:8000	python3	808	⋮
+ New port	This port config is saved in the <code>.replit</code> file.			

→ O esperado é aparecer uma página como essa:

The screenshot shows the API documentation for "API de Mulheres Cientistas" version 0.1.0. The main title is "Cientistas". The available endpoints are:

- `GET /cientistas` Get Cientistas
- `POST /cientistas` Create Cientista
- `GET /cientistas/{cientista_id}` Get Cientista By Id
- `PUT /cientistas/{cientista_id}` Update Cientista
- `DELETE /cientistas/{cientista_id}` Delete Cientista

Below this, there is a "default" section with the endpoint:

- `GET / Index`



**Utilize os dados do arquivo `exemplos.json` para popular a API**

**Após inserir os dados, teste as rotas para leitura, edição e deleção**



# Adicionando as Frases



# Frase

Nosso segundo modelo

Frase		
🔑	<code>id</code>	<code>int</code>
	<code>cientista_id</code>	<code>int</code>
	<code>frase</code>	<code>str</code>

# Modificações – models.py



```
from sqlmodel import Field, Relationship, SQLModel

class Cientista(CientistaBase, table=True):
    id: int | None = Field(default=None, primary_key=True)
    frases: list["Frase"] = Relationship(back_populates="cientista", cascade_delete=True)

class CientistaCreate(CientistaBase):
    frases: list[str] | None = Field(exclude=True)

class CientistaPublic(CientistaBase):
    id: int
    frases: list["FrasePublic"] = []
```

# Modificações – models.py



```
from sqlmodel import Field, Relationship, SQLModel

class Cientista(CientistaBase, table=True):
    id: int | None = Field(default=None, primary_key=True)
    frases: list["Frase"] = Relationship(back_populates="cientista", cascade_delete=True)

        Relaciona a tabela Cientista com a tabela Frase
```

```
class CientistaCreate(CientistaBase):
    frases: list[str] | None = Field(exclude=True)

        Remove o campo na serialização
```

```
class CientistaPublic(CientistaBase):
    id: int
    frases: list["FrasePublic"] = []
```

Inclui a lista de Frases na  
exibição de cada cientista



# Modificações – models.py

```
class FraseBase(SQLModel):
    frase: str = Field(nullable=False)
    cientista_id: int = Field(foreign_key="cientista.id", nullable=False)
        Foreign Key para a tabela Cientista

class Frase(FraseBase, table=True):
    id: int | None = Field(default=None, primary_key=True)
    cientista: Cientista = Relationship(back_populates="frases")
        Relaciona a tabela Frase com a tabela Cientista

class FraseCreate(FraseBase):
    pass

class FrasePublic(FraseBase):
    id: int
```



# Desafio: /routers/frases.py



```
✓ CURSO-FASTAPI
  ✓ mulheres_cientistas_api
    ✓ routers
      __init__.py
      cientista.py
      frase.py ←
      __init__.py
      config.py
      database.py
      main.py
    .env
    .gitignore
    exemplos.json
    mulheres_cientistas.db
    poetry.lock
    pyproject.toml
    README.md
```

**Na pasta**

→ /routers

**Crie o arquivo**

→ frase.py

## Router – Frase

- Crie as rotas CRUD para Frase, seguindo o mesmo padrão utilizado para Cientista
- Teste as novas rotas através da página /docs
  - ◆ Será necessário passar o id de uma Cientista existente para criar uma Frase vinculada à ela

Dica



- Não esqueça de importar o router no módulo main.py

# Template – router/frase.py



```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.exc import IntegrityError
from sqlmodel import Session

from mulheres_cientistas_api.crud import create, delete, get_all, get_one, update
from mulheres_cientistas_api.database import get_session
from mulheres_cientistas_api.models import (
    Frase,
    FraseCreate,
    FrasePublic,
    FraseUpdate,
)
router = APIRouter(prefix="/frases", tags=["Frases"])

@router.post("", response_model=FrasePublic, status_code=201)
def create_frase(frase_to_create: FraseCreate, session: Session = Depends(get_session)):
    pass
```

```
@router.get("", response_model=list[FrasePublic], status_code=200)
def get_frases(session: Session = Depends(get_session)):
    pass

@router.get("/{frase_id}", response_model=FrasePublic, status_code=200)
def get_frase_by_id(frase_id: int, session: Session = Depends(get_session)):
    pass

@router.put("/{frase_id}", response_model=FrasePublic, status_code=200)
def update_frase(
    frase_id: int,
    frase_update: FraseUpdate,
    session: Session = Depends(get_session),
):
    pass

@router.delete("/{frase_id}", status_code=204)
def delete_frase(frase_id: int, session: Session = Depends(get_session)):
    pass
```



# Modificações – main.py

```
from contextlib import asynccontextmanager
from fastapi import FastAPI
from mulheres_cientistas_api.database import init_db
from mulheres_cientistas_api.routers import cientista, frase

@asynccontextmanager
async def lifespan(_: FastAPI):
    init_db()
    yield

app = FastAPI(
    title="API de Mulheres Cientistas",
    lifespan=lifespan
)
app.include_router(cientista.router)
app.include_router(frase.router)

@app.get("/")
def index():
    return {"title": "API de Mulheres Cientistas"}
```



# Lidando com casos especiais



# Lidando com casos especiais

Tratamento de erros

## Caso #1

- O que acontece quando tenta-se criar uma Frase com um `cientista_id` inválido?

# Modificações – routers/frase.py



```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.exc import IntegrityError
from sqlmodel import Session

from mulheres_cientistas_api.crud import create, delete, get_all, get_one, update
from mulheres_cientistas_api.database import get_session
from mulheres_cientistas_api.models import (
    Frase,
    FraseCreate,
    FrasePublic,
    FraseUpdate,
)

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

@router.post("", response_model=FrasePublic, status_code=201)
def create_frase(frase_to_create: FraseCreate, session: Session = Depends(get_session)):
    try:
        return create(session=session, model=Frase, data=frase_to_create)
    except IntegrityError:
        raise HTTPException(status_code=422, detail="Cientista especificada não existe")
```

Tratamento da Exceção IntegrityError do ORM



# Lidando com casos especiais

Registros aninhados

→ É possível criar uma  
**Cientista** e suas **Frases**  
utilizando somente uma  
requisição?

**Caso #2**



# Modificações – routers/cientista.py

```
from fastapi import APIRouter, Depends, HTTPException
from sqlmodel import Session

from mulheres_cientistas_api.crud import create, delete, get_all, get_one, update
from mulheres_cientistas_api.database import get_session
from mulheres_cientistas_api.models import (
    Cientista,
    CientistaCreate,
    CientistaPublic,
    CientistaUpdate,
    Frase,
    FraseCreate,
)
router = APIRouter(prefix="/cientistas", tags=["Cientistas"])

@router.post("", response_model=CientistaPublic, status_code=201)
def create_cientista(
    cientista_to_create: CientistaCreate, session: Session = Depends(get_session)
):
    cientista_id = create(session=session, model=Cientista, data=cientista_to_create).id Cria a Cientista primeiro
    for frase_to_create in cientista_to_create.frases:
        create(session=session, model=Frase, data=FraseCreate(frase=frase_to_create, cientista_id=cientista_id))

    return get_one(session=session, model=Cientista, id=cientista_id) Retorna a Cientista
```

**Depois cria  
as Frases**



# Dúvidas?

# Obrigada!

-  [saocarlos@pyladies.com](mailto:saocarlos@pyladies.com)
-  [instagram.com/pyladiessanca](https://instagram.com/pyladiessanca)
-  [linkedin.com/company/pyladiessanca](https://linkedin.com/company/pyladiessanca)
-  [facebook.com/PyLadiesSaoCarlos](https://facebook.com/PyLadiesSaoCarlos)

