# Introdução ao FastAPI

# Ana Dulce

- Desenvolvedora Python desde 2018 - atualmente engenheira de software no KaBuM!;
- Cofundadora do PyLadies São Carlos;
- Organizadora de eventos da comunidade Python no tempo livre;
- Atual Conselheira e ex-presidente da Associação Python Brasil;
- Python Fellow Member;

# Renan

- Bacharel em Física
- Trabalho com engenharia de software no Serasa
- Gosto de vôlei, jogos de tabuleiro e tenho uma tatuagem do desenho Avatar
- Ajudo na organização de eventos Python Brasil

# Alinhando expectativas - o que não vamos abordar

- Deploy
- Testes
- Observabilidade e monitoramento

Spoilers: Testes, observabilidade e muito mais sobre desenvolvimento de sistemas web, quem quiser, tem no minicurso de amanhã :D

# Objetivos

Criar uma aplicação web simples usando o framework FastAPI, usando o banco de dados SQLite3, o ORM SQLALchemy e Pydantic, onde possamos explorar essencialmente a **criação de endpoints** usando o framework, e maneiras de tornar o desenvolvimento mais ágil e menos propenso a erros.

# Instalação de bibliotecas

python 3.11.10

poetry 1.8.2

sqlite3 3.37.2

# Clonar o projeto do GitHub

https://github.com/anadulce/fastapi_demo

# Conhecendo o que já está pré-definido

- as bibliotecas necessárias para executar o projeto;
- formatadores e linter;
- alguns testes;
- automações de comandos no taskpy;

# Estrutura de arquivos inicial

```
fastapi_demo
    src
        __init__.py
        app.py
        database.py
        config.py
        models.py
    .env
    .gitignore
    pyproject.toml
    poetry.lock
    README.md
```

# Hello, world!

```python
# src/app.py

from fastapi import FastAPI

app = FastAPI()


@app.get('/')
def index():
    return "Hello, world!"
```

# Executando

```
task run
```

# Configurações

Criando o .env

```
# .env
DATABASE_URL="sqlite:///fastapi_demo.db"
```

# Configurações

```
# src/config.py

from pydantic_settings import BaseSettings, SettingsConfigDict


class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file='.env', env_file_encoding='utf-8'
    )
    DATABASE_URL: str


settings = Settings()
```

# Criando a conexão com o banco

```python
# src/database.py

from sqlalchemy import create_engine
from sqlalchemy.orm import Session

from src.config import Settings

engine = create_engine(Settings().DATABASE_URL)


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

# Criando as models

```
# src/models.py

from datetime import datetime

from sqlalchemy import ForeignKey, func
from sqlalchemy.orm import Mapped, mapped_column, registry, relationship

table_registry = registry()
```

# Criando as models

```
    # src/models.py

    @table_registry.mapped_as_dataclass
class Genre:
    __tablename__ = 'genre'

    id: Mapped[int] = mapped_column(init=False, primary_key=True)
    name: Mapped[str] = mapped_column(unique=True)

    created_at: Mapped[datetime] = mapped_column(default=func.now())
    updated_at: Mapped[datetime] = mapped_column(
        default=func.now(), onupdate=func.now()
    )
    movies: Mapped[list['Movie']] = relationship(
        init=False, back_populates='genre', cascade='all, delete-orphan'
    )
```

# Criando as models

```
    # src/models.py
    @table_registry.mapped_as_dataclass
class Movie:
    __tablename__ = 'movie'

    id: Mapped[int] = mapped_column(init=False, primary_key=True)
    title: Mapped[str]
    director: Mapped[str]
    year: Mapped[int]

    genre_id: Mapped[int] = mapped_column(ForeignKey('genre.id'))

    genre: Mapped[Genre] = relationship(init=False, back_populates='movies')

    created_at: Mapped[datetime] = mapped_column(default=func.now())
    updated_at: Mapped[datetime] = mapped_column(
        default=func.now(), onupdate=func.now()
    )
```

# Comandos para criação do banco

Inicializa migrações:

```
alembic init migrations
```

# Configurando o banco

No arquivo `migrations/env.py`, adicionar depois do import:

```
from src.models import table_registry
from src.config import Settings
```

Mudar a configuração principal:

```
config.set_main_option('sqlalchemy.url', Settings().DATABASE_URL)
```

Setar o target_metadata:

```
target_metadata = table_registry.metadata
```

# Cria arquivos de migrações

```
alembic revision --autogenerate -m "mensagem de criação"
```

# Conferindo o banco

Abrir terminal interativo do banco:

```
sqlite3 fastapi_demo.db
```

Aqui você pode conferir que ainda não há nenhuma tabela

```
sqlite> .schema
sqlite> .quit
```

# Aplicando as migrações

```
alembic upgrade head
```

E se voltarmos novamente no banco, elas estarão lá!

```
sqlite> .schema
sqlite> .quit
```

# Criando Schemas

```python
from datetime import datetime

from pydantic import BaseModel


class Message(BaseModel):
    message: str


class GenreInSchema(BaseModel):
    name: str
```

# Criando Schemas

```
class GenreOutSchema(GenreInSchema):
    id: int
    created_at: datetime
    updated_at: datetime

    class Config:
        from_attributes = True


class PageGenreSchema(BaseModel):
    page: int = 1
    limit: int = 100
    genres: list[GenreOutSchema]
```

# CRUD - Criando os endpoints

Agora chegamos ao ponto importante da nossa API, vamos criar nossos endpoints!

# Configs

```python
# src/routers/genre.py

from http import HTTPStatus
from fastapi import APIRouter, Depends

from sqlalchemy.orm import Session

from src.database import get_session
from src.models import Genre
from src.routers.schema import GenreInSchema, GenreOutSchema

router = APIRouter(prefix='/genre', tags=['genres'])
```

# CREATE

```
# src/routers/genre.py

@router.post(
    '/',
    status_code=HTTPStatus.CREATED,
    response_model=GenreOutSchema
    )
def create_genre(
    genre: GenreInSchema,
    session: Session = Depends(get_session)
    ):
    db_genre = Genre(**genre.model_dump())
    session.add(db_genre)
    session.commit()
    session.refresh(db_genre)

    return db_genre
```

# Executando o endpoint

```
task run
```

# Documentação automática

http://127.0.0.1:8000/docs

# Refatorando CREATE

```python
@router.post(
    '/',
    status_code=HTTPStatus.CREATED,
    response_model=GenreOutSchema,
)
def create_genre(
    genre: GenreInSchema,
    session: Session = Depends(get_session),
):
    try:
        db_genre = Genre(**genre.model_dump())
        session.add(db_genre)
        session.commit()
        session.refresh(db_genre)
        return db_genre
    except IntegrityError:
        raise HTTPException(
            HTTPStatus.BAD_REQUEST,
            detail='Genre already exists',
        )
```

# READ - Todos os registros

```python
# src/routers/genre.py

@router.get(
    '/',
    status_code=HTTPStatus.OK,
    response_model=list[GenreOutSchema],
)
def read_genre(
    session: Session = Depends(get_session),
):
    genres = session.execute(select(Genre)).scalars().all()
    return genres
```

# READ - Paginação

```
# src/routers/genre.py

@router.get('/', response_model=PageGenreSchema)
def read_genre(
    page: int = 1,
    limit: int = 100,
    session: Session = Depends(get_session),
):
    genres = session.scalars(
        select(Genre)
        .offset((page - 1) * limit)
        .limit(limit)
    ).all()

    return {
        'page': page,
        'limit': limit,
        'genres': genres,
    }
```

# READ - by ID

```python
# src/routers/genre.py
@router.get('/{id}/', response_model=GenreOutSchema)
def get_genre(
    id: int,
    session: Session = Depends(get_session),
):
    if genre := session.scalars(
        select(Genre).where(Genre.id == id)
    ).one_or_none():
        return genre

    raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Genre not found',
    )
```

# UPDATE - PUT

```
@router.put(
    '/{id}/',
    status_code=HTTPStatus.OK,
    response_model=GenreOutSchema,
)
def update_genre(
    id: int,
    genre_to_update: GenreInSchema,
    session: Session = Depends(get_session),
):
    if db_genre := session.scalars(
        select(Genre).where(Genre.id == id)
    ).one_or_none():
        try:
            for (
                attr, value,
            ) in genre_to_update.dict(
                exclude_unset=True
            ).items():
                setattr(db_genre, attr, value)
            session.add(db_genre)
```

# UPDATE - PUT

```
            session.commit()
            session.refresh(db_genre)
            return db_genre
        except IntegrityError:
            raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Genre already exists',
    )
    raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Genre not found',
    )
```

# UPDATE - PATCH

```python
@router.patch(
    '/{id}/',
    status_code=HTTPStatus.OK,
    response_model=GenreOutSchema,
)
def partial_update_genre(
    id: int,
    genre_to_update: GenreInSchema,
    session: Session = Depends(get_session),
):
    if db_genre := session.scalars(
        select(Genre).where(Genre.id == id)
    ).one_or_none():
        try:
            for (
                attr,
                value,
            ) in genre_to_update.dict(
                exclude_unset=True
            ).items():
                setattr(db_genre, attr, value)
            session.add(db_genre)
```

# UPDATE - PATCH

```
# src/routers/genre.py
            session.commit()
            session.refresh(db_genre)
            return db_genre
        except IntegrityError:
            raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Genre already exists',
    )
    raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Genre not found',
    )
```

# DELETE

```python
# src/routers/genre.py
@router.delete(
    '/{id}/',
    status_code=HTTPStatus.OK,
    response_model=Message,
)
def delete_genre(
    id: int,
    session: Session = Depends(get_session),
):
    if db_genre := session.scalars(
        select(Genre).where(Genre.id == id)
    ).one_or_none():
        session.delete(db_genre)
        session.commit()
        return {'message': 'Genre deleted'}
    raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Genre not found',
    )
```

# Separando o CRUD da view

# CREATE - BD

```
# src/crud.py

def create(session: Session, model, data):
    try:
        obj = model(**data.model_dump())
        session.add(obj)
        session.commit()
        session.refresh(obj)
        return obj
    except IntegrityError:
        None
```

# CREATE - VIEW

```python
# src/routers/genre.py
@router.post(
    '/',
    status_code=HTTPStatus.CREATED,
    response_model=GenreOutSchema,
)
def create_genre(
    genre: GenreInSchema,
    session: Session = Depends(get_session),
):
    if genre := create(session, Genre, genre):
        return genre
    raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Genre already exists',
    )
```

# READ ALL - BD

```python
# src/crud.py

def get_all(session: Session, model):
    query = select(model)
    return session.scalars(query).all()
```

# READ ALL - VIEW

```python
# src/routers/genre.py

@router.get('/', response_model=list[GenreOutSchema])
def read_genres(session: Session = Depends(get_session)):
    return get_all(session, Genre)
```

# READ PAGINATED - BD

```python
# src/crud.py

def get_offset(session: Session, model, offset, limit):
    query = (
        select(model).offset(offset * limit).limit(limit)
    )
    return session.scalars(query).all()
```

# READ PAGINATED - VIEW

```python
# src/routers/genre.py

@router.get('/pages/', response_model=PageGenreSchema)
def read_genres_by_page(
    page: int = 1,
    limit: int = 100,
    session: Session = Depends(get_session),
):
    genres = get_offset(session, Genre, page - 1, limit)

    return {'page': page, 'limit': limit, 'genres': genres}
```

# READ ONE - BD

```python
# src/crud.py

def get_one(session: Session, model, id: int):
    query = select(model).where(model.id == id)
    return session.scalars(query).one_or_none()
```

# READ ONE - VIEW

```python
# src/routers/genre.py

@router.get('/{id}/', response_model=GenreOutSchema)
def read_genres(
    id: int, session: Session = Depends(get_session)
):
    if genre := get_one(session, Genre, id):
        return genre

    raise HTTPException(
        HTTPStatus.NOT_FOUND, detail='Genre not found'
    )
```

# UPDATE ALL TYPES - DB

```python
# src/crud.py

def update(session: Session, model, id: int, data):
    query = select(model).where(model.id == id)
    obj = session.scalars(query).one_or_none()
    if obj is None:
        return None, 'not found'
    for attr, value in data.dict(
        exclude_unset=True
    ).items():
        setattr(obj, attr, value)
    try:
        session.commit()
        session.refresh(obj)
        return obj, None
    except IntegrityError:
        return None, 'already exists'
```

# TOTAL UPDATE (PUT) - VIEW

```python
# src/routers/genre.py

@router.put('/{id}/', response_model=GenreOutSchema)
def update_genre(
    id: int,
    genre_to_update: GenreInSchema,
    session: Session = Depends(get_session),
):
    genre, message = update(
        session, Genre, id, genre_to_update
    )
    if genre:
        return genre
    raise HTTPException(
        HTTPStatus.NOT_FOUND, detail=f'Genre {message}'
    )
```

# PARTIAL UPDATE (PATCH) - VIEW

```python
# src/routers/genre.py
@router.patch('/{id}/', response_model=GenreOutSchema)
def partial_update_genre(
    id: int,
    genre_to_update: GenreInSchema,
    session: Session = Depends(get_session),
):
    genre, message = update(
        session, Genre, id, genre_to_update
    )
    if genre:
        return genre
    raise HTTPException(
        HTTPStatus.NOT_FOUND, detail=f'Genre {message}'
    )
```

# DELETE - BD

```python
# src/crud.py

def delete(session: Session, model, id: int) -> bool:
    query = select(model).where(model.id == id)
    obj = session.scalars(query).one_or_none()

    if obj is None:
        return False

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

# DELETE - VIEW

```python
# src/routers/genre.py
@router.delete(
    '/{id}/',
    status_code=HTTPStatus.OK,
    response_model=Message,
)
def delete_genre(
    id: int, session: Session = Depends(get_session)
):
    if delete(session, Genre, id):
        return {'message': 'Genre deleted'}

    raise HTTPException(
        HTTPStatus.NOT_FOUND, detail='Genre not found'
    )
```

# Criar rotas para Movie

# Importando dependências

```python
# src/routers/movie.py

from http import HTTPStatus

from fastapi import (
    APIRouter,
    Depends,
    HTTPException,
)
from sqlalchemy.orm import Session

from src.crud import (
    create,
    delete,
    get_all,
    get_offset,
    get_one,
    update,
)
```

# Importando dependências

```python
# src/routers/movie.py

from src.database import get_session
from src.models import Movie
from src.routers.schema import (
    Message,
    MovieInSchema,
    MovieOutSchema,
    MoviePartialUpdateSchema,
    PageMovieSchema,
)

router = APIRouter(prefix='/movie', tags=['movies'])
```

# CREATE - VIEW

```
# src/routers/movie.py

@router.post(
    '/',
    status_code=HTTPStatus.CREATED,
    response_model=MovieOutSchema,
)
def create_movie(
    movie: MovieInSchema,
    session: Session = Depends(get_session),
):
    if movie := create(session, Movie, movie):
        return movie
    raise HTTPException(
        HTTPStatus.BAD_REQUEST,
        detail='Movie already exists',
    )
```

# READ ALL - VIEW

```python
# src/routers/movie.py

@router.get(
    '/',
    response_model=list[MovieOutSchema],
)
def read_movies(
    session: Session = Depends(get_session),
):
    return get_all(session, Movie)
```

# PAGINATED READ - VIEW

```python
# src/routers/movie.py

@router.get(
    '/pages/',
    response_model=PageMovieSchema,
)
def read_movies_by_page(
    page: int = 1,
    limit: int = 100,
    session: Session = Depends(get_session),
):
    movies = get_offset(session, Movie, page - 1, limit)

    return {
        'page': page,
        'limit': limit,
        'movies': movies,
    }
```

# READ ONE - VIEW

```
# src/routers/movie.py

@router.get(
    '/{id}/',
    response_model=MovieOutSchema,
)
def read_movies(
    id: int,
    session: Session = Depends(get_session),
):
    if movie := get_one(session, Movie, id):
        return movie

    raise HTTPException(
        HTTPStatus.NOT_FOUND,
        detail='Movie not found',
    )
```

# TOTAL UPDATE (PUT) - VIEW

```python
# src/routers/movie.py

@router.put(
    '/{id}/',
    response_model=MovieOutSchema,
)
def update_movie(
    id: int,
    movie_to_update: MovieInSchema,
    session: Session = Depends(get_session),
):
    movie, message = update(
        session,
        Movie,
        id,
        movie_to_update,
    )
    if movie:
        return movie
    raise HTTPException(
        HTTPStatus.NOT_FOUND,
        detail=f'Movie {message}',
    )
```

# PARTIAL UPDATE (PATCH) - VIEW

```python
# src/routers/movie.py

@router.patch(
    '/{id}/',
    response_model=MovieOutSchema,
)
def partial_update_movie(
    id: int,
    movie_to_update: MoviePartialUpdateSchema,
    session: Session = Depends(get_session),
):
    movie, message = update(
        session,
        Movie,
        id,
        movie_to_update,
    )
    if movie:
        return movie
    raise HTTPException(
        HTTPStatus.NOT_FOUND,
        detail=f'Movie {message}',
    )
```

# DELETE - VIEW

```python
# src/routers/movie.py

@router.delete(
    '/{id}/',
    status_code=HTTPStatus.OK,
    response_model=Message,
)
def delete_movie(
    id: int,
    session: Session = Depends(get_session),
):
    if delete(session, Movie, id):
        return {'message': 'Movie deleted'}

    raise HTTPException(
        HTTPStatus.NOT_FOUND,
        detail='Movie not found',
    )
```