README.md

FastAPI Workshop

Criando uma aplicação web com API usando FastAPI

Requisitos

- Computador com Python 3.10
- Docker & docker-compose
- Ou https://gitpod.io para um ambiente online
- Um editor de códigos como VSCode, Sublime, Vim, Micro (Nota: Eu usarei o Micro-Editor)

> **importante**: Os comandos apresentados serão executados em um terminal Linux, se estiver no Windows recomendo usar o WSL, uma máquina virtual ou um container Linux, ou por conta própria adaptar os comandos necessários.

Ambiente

Primeiro precisamos de um ambiente virtual para instalar as dependencias do projeto.

```
python -m venv .venv
```

E ativaremos a virtualenv

```
# Linux
source .venv/bin/activate
# Windows Power Shell
.\venv\Scripts\activate.ps1
```

Vamos instalar ferramentas de produtividade neste ambiente e para isso vamos criar um arquivo chamado requirements-dev.txt

```
ipython         # terminal
ipdb            # debugger
sdb             # debugger remoto
pip-tools       # lock de dependencias
pytest          # execução de testes
pytest-order    # ordenação de testes
httpx           # requests async para testes
```

```
black           # auto formatação
flake8          # linter
```

Instalamos as dependencias iniciais.

```
pip install --upgrade pip
pip install -r requirements-dev.txt
```

O Projeto

Nosso projeto será um microblog estilo twitter, é um projeto simples porém com funcionalidade suficientes para exercitar as principais features de uma API.

Vamos focar no backend, ou seja, na API apenas, o nome do projeto é "PAMPS" um nome aleatório que encontrei para uma rede social ficticia.

Funcionalidades

Usuários

- Registro de novos usuários
- Autenticação de usuários
- Seguir outros usuários
- Perfil com bio e listagem de posts, seguidores e seguidos

Postagens

- Criação de novo post
- Edição de post
- Remoção de post
- Listagem de posts geral (home)
- Listagem de posts seguidos (timeline)
- Likes em postagens
- Postagem pode ser resposta a outra postagem

Estrutura de pastas e arquivos

Script para criar os arquivos do projeto.

```
# Arquivos na raiz
touch setup.py
touch {settings,.secrets}.toml
touch {requirements,MANIFEST}.in
touch Dockerfile.dev docker-compose.yaml
```

```
# Imagem do banco de dados
mkdir postgres
touch postgres/{Dockerfile,create-databases.sh}

# Aplicação
mkdir -p pamps/{models,routes}
touch pamps/default.toml
touch pamps/{__init__,cli,app,auth,db,security,config}.py
touch pamps/models/{__init__,post,user}.py
touch pamps/routes/{__init__,auth,post,user}.py

# Testes
touch test.sh
mkdir tests
touch tests/{__init__,conftest,test_api}.py
```

Esta será a estrutura final (se preferir criar manualmente)

```
❯ tree --filesfirst -L 3 -I docs
.
├── docker-compose.yaml        # Orquestração de containers
├── Dockerfile.dev             # Imagem principal
├── MANIFEST.in                # Arquivos incluidos na aplicação
├── requirements-dev.txt       # Dependencias de ambiente dev
├── requirements.in            # Dependencias de produção
├── .secrets.toml              # Senhas locais
├── settings.toml              # Configurações locais
├── setup.py                   # Instalação do projeto
├── test.sh                    # Pipeline de CI em ambiente dev
├── pamps
│   ├── __init__.py
│   ├── app.py                 # FastAPI app
│   ├── auth.py                # Autenticação via token
│   ├── cli.py                 # Aplicação CLI `$ pamps adduser` etc
│   ├── config.py              # Inicialização da config
│   ├── db.py                  # Conexão com o banco de dados
│   ├── default.toml           # Config default
│   ├── security.py            # Password Validation
│   ├── models
│   │   ├── __init__.py
│   │   ├── post.py            # ORM e Serializers de posts
│   │   └── user.py            # ORM e Serialziers de users
│   └── routes
│       ├── __init__.py
│       ├── auth.py            # Rotas de autenticação via JWT
│       ├── post.py            # CRUD de posts e likes
│       └── user.py            # CRUD de user e follows
├── postgres
│   ├── create-databases.sh    # Script de criação do DB
│   └── Dockerfile             # Imagem do SGBD
└── tests
    ├── conftest.py            # Config do Pytest
    ├── __init__.py
    └── test_api.py            # Tests da API
```

Adicionando as dependencias

Editaremos o arquivo `requirements.in` e adicionaremos

```
fastapi
uvicorn
sqlmodel
typer
dynaconf
jinja2
python-jose[cryptography]
passlib[bcrypt]
python-multipart
psycopg2-binary
alembic
rich
```

A partir deste arquivo vamos gerar um `requirements.txt` com os locks das versões.

```
pip-compile requirements.in
```

E este comando irá gerar o arquivo `requirements.txt` organizado e com as versões pinadas.

Criando a API base

Vamos editar o arquico `pamps/app.py`

```
from fastapi import FastAPI

app = FastAPI(
    title="Pamps",
    version="0.1.0",
    description="Pamps is a posting app",
)
```

Tornando a aplicação instalável

`MANIFEST.in`

```
graft pamps
```

`setup.py`

```
import io
import os
```

```
from setuptools import find_packages, setup


def read(*paths, **kwargs):
    content = ""
    with io.open(
        os.path.join(os.path.dirname(__file__), *paths),
        encoding=kwargs.get("encoding", "utf8"),
    ) as open_file:
        content = open_file.read().strip()
    return content


def read_requirements(path):
    return [
        line.strip()
        for line in read(path).split("\n")
        if not line.startswith(('"', "#", "-", "git+"))
    ]


setup(
    name="pamps",
    version="0.1.0",
    description="Pamps is a social posting app",
    url="pamps.io",
    python_requires=">=3.8",
    long_description="Pamps is a social posting app",
    long_description_content_type="text/markdown",
    author="Melon Husky",
    packages=find_packages(exclude=["tests"]),
    include_package_data=True,
    install_requires=read_requirements("requirements.txt"),
    entry_points={
        "console_scripts": ["pamps = pamps.cli:main"]
    }
)
```

Instalação

O nosso objetivo é instalar a aplicação dentro do container, porém é recomendável que instale também no ambiente local pois desta maneira auto complete do editor irá funcionar.

```
pip install -e .
```

Containers

Vamos agora escrever o Dockerfile.dev responsável por executar nossa api

`Dockerfile.dev`

```
# Build the app image
FROM python:3.10

# Create directory for the app user
RUN mkdir -p /home/app

# Create the app user
RUN groupadd app && useradd -g app app

# Create the home directory
ENV APP_HOME=/home/app/api
RUN mkdir -p $APP_HOME
WORKDIR $APP_HOME

# install
COPY . $APP_HOME
RUN pip install -r requirements-dev.txt
RUN pip install -e .

RUN chown -R app:app $APP_HOME
USER app

CMD ["uvicorn","pamps.app:app","--host=0.0.0.0","--port=8000","--reload"]
```

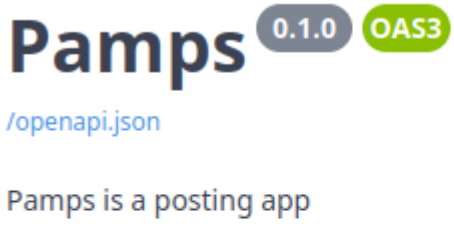
Build the container

```
docker build -f Dockerfile.dev -t pamps:latest .
```

Execute o container para testar

```
$ docker run --rm -it -v $(pwd):/home/app/api -p 8000:8000 pamps
INFO:     Will watch for changes in these directories: ['/home/app/api']
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [1] using StatReload
INFO:     Started server process [8]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

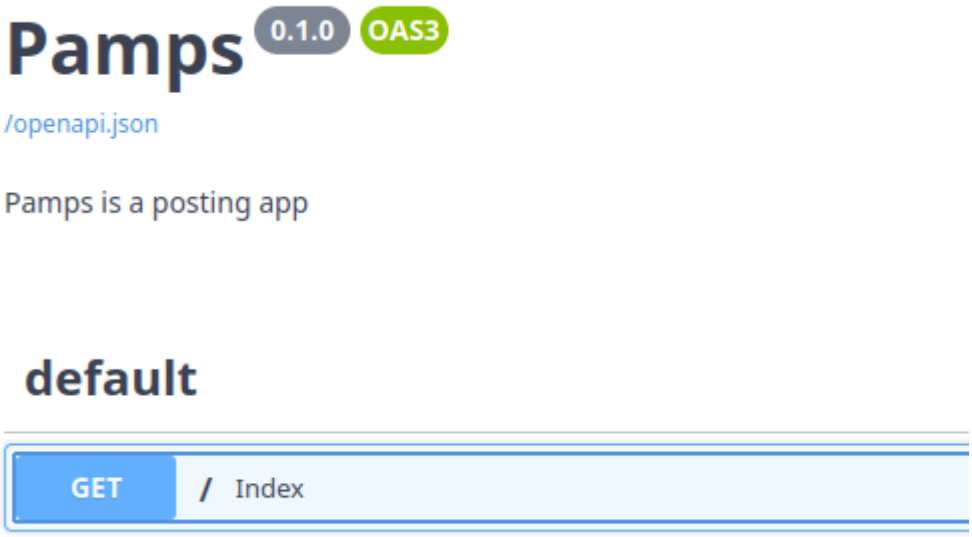
Acesse: http://0.0.0.0:8000/docs



A API vai ser atualizada automaticamente quando detectar mudanças no código, somente para teste edite `pamps/app.py` e adicione

```
@app.get("/")
async def index():
    return {"hello": "world"}
```

Agora acesse novamente http://0.0.0.0:8000/docs



> **NOTA**: pode remover a rota `index()` pois foi apenas para testar, vamos agora adicionar rotas de maneira mais organizada.

Rodando um banco de dados em container

Agora precisaremos de um banco de dados e vamos usar o PostgreSQL dentro de um container.

Edite `postgres/create-databases.sh`

```
#!/bin/bash
```

```
set -e
set -u

function create_user_and_database() {
        local database=$1
        echo "Creating user and database '$database'"
        psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL
            CREATE USER $database PASSWORD '$database';
            CREATE DATABASE $database;
            GRANT ALL PRIVILEGES ON DATABASE $database TO $database;
EOSQL
}

if [ -n "$POSTGRES_DBS" ]; then
        echo "Creating DB(s): $POSTGRES_DBS"
        for db in $(echo $POSTGRES_DBS | tr ',' ' '); do
                create_user_and_database $db
        done
        echo "Multiple databases created"
fi
```

O script acima vai ser executado no inicio da execução do Postgres de forma que quando a aplicação iniciar teremos certeza de que o banco de dados está criado.

Agora precisamos do Sistema de BD rodando e vamos criar uma imagem com Postgres.

Edite `postres/Dockerfile`

```
FROM postgres:alpine3.14
COPY create-databases.sh /docker-entrypoint-initdb.d/
```

Docker compose

Agora para iniciar a nossa API + o Banco de dados vamos precisar de um orquestrador de containers, em produção isso será feito com Kubernetes mas no ambiente de desenvolvimento podemos usar o docker-compose.

Edite o arquivo `docker-compose.yaml`

- Definimos 2 serviços `api` e `db`
- Informamos os parametros de build com os dockerfiles
- Na `api` abrimos a porta `8000`
- Na `api` passamos 2 variáveis de ambiente `PAMPS_DB__uri` e `PAMPS_DB_connect_args` para usarmos na conexão com o DB
- Marcamos que a `api` depende do `db` para iniciar.
- No `db` informamos o setup básico do postgres e pedimos para criar 2 bancos de dados, um para a app e um para testes.

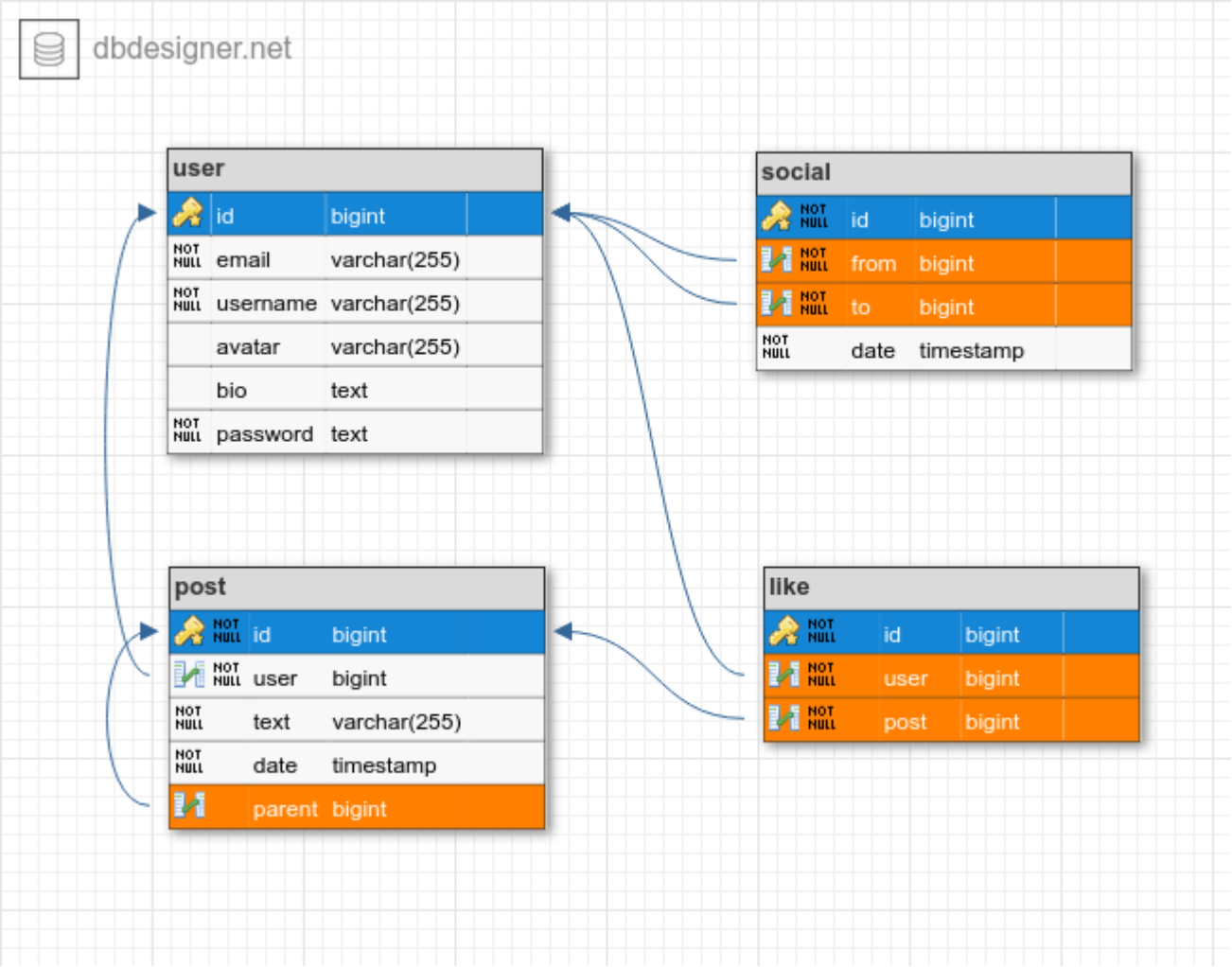
```
version: '3.9'

services:
  api:
    build:
      context: .
      dockerfile: Dockerfile.dev
    ports:
      - "8000:8000"
    environment:
      PAMPS_DB__uri: "postgresql://postgres:postgres@db:5432/${PAMPS_DB:-pamps}"
      PAMPS_DB__connect_args: "{}"
    volumes:
      - .:/home/app/api
    depends_on:
      - db
    stdin_open: true
    tty: true
  db:
    build: postgres
    image: pamps_postgres-13-alpine-multi-user
    volumes:
      - $HOME/.postgres/pamps_db/data/postgresql:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DBS=pamps, pamps_test
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

O próximo passo é executar com

```
docker-compose up
```

Definindo os models com Pydantic



https://dbdesigner.page.link/qQHdqeYRTqKUfmrt7

Vamos modelar o banco de dados definido acima usando o SQLModel, que é uma biblioteca que integra o SQLAlchemy e o Pydantic e funciona muito bem com o FastAPI.

Vamos começar a estruturar os model principal para armazenar os usuários

edite o arquivo `pamps/models/user.py`

```
"""User related data models"""
from typing import Optional
from sqlmodel import Field, SQLModel


class User(SQLModel, table=True):
    """Represents the User Model"""

    id: Optional[int] = Field(default=None, primary_key=True)
    email: str = Field(unique=True, nullable=False)
    username: str = Field(unique=True, nullable=False)
    avatar: Optional[str] = None
    bio: Optional[str] = None
    password: str = Field(nullable=False)
```

No arquivo `pamps/models/__init__.py` adicione

```
from sqlmodel import SQLModel
from .user import User

__all__ = ["User", "SQLModel"]
```

Settings

Agora que temos pelo menos uma tabela mapeada para uma classe precisamos estabelecer conexão com o banco de dados e para isso precisamos carregar configurações

Edite o arquivo `pamps/default.toml`

```
[default]

[default.db]
uri = ""
connect_args = {check_same_thread=false}
echo = false
```

Lembra que no `docker-compose.yaml` passamos as variáveis `PAMPS_DB...` aquelas variáveis vão sobrescrever os valores definidos no default settings.

Vamos agora inicializar a biblioteca de configurações:

Edite `pamps/config.py`

```
"""Settings module"""
import os

from dynaconf import Dynaconf

HERE = os.path.dirname(os.path.abspath(__file__))

settings = Dynaconf(
    envvar_prefix="pamps",
    preload=[os.path.join(HERE, "default.toml")],
    settings_files=["settings.toml", ".secrets.toml"],
    environments=["development", "production", "testing"],
    env_switcher="pamps_env",
    load_dotenv=False,
)
```

No arquivo acima estamos definindo que o objeto `settings` irá carregar variáveis do arquivo `default.toml` e em seguida dos arquivos `settings.toml` e `.secrets.toml` e que será possivel usar `PAMPS_` como prefixo nas variáveis de ambiente para sobrescrever os valores.

Conexão com o banco de dados

Edite `pamps/db.py`

```
"""Database connection"""
from sqlmodel import create_engine
from .config import settings

engine = create_engine(
    settings.db.uri,
    echo=settings.db.echo,
    connect_args=settings.db.connect_args,
)
```

Criamos um objeto `engine` que aponta para uma conexão com o banco de dados e para isso usamos as variáveis que lemos do `settings`.

Database Migrations

Portanto agora já temos uma tabela mapeada e um conexão com o banco de dados precisamos agora garantir que a estrutura da tabela existe dentro do banco de dados.

Para isso vamos usar a biblioteca `alembic` que gerencia migrações, ou seja, alterações na estrutura das tabelas.

Começamos na raiz do repositório e rodando:

```
alembic init migrations
```

O alembic irá criar um arquivo chamado `alembic.ini` e uma pasta chamada `migrations` que servirá para armazenar o histórico de alterações do banco de dados.

Começaremos editando o arquivo `migrations/env.py`

```
# No topo do arquivo adicionamos
from pamps import models
from pamps.db import engine
from pamps.config import settings


# Perto da linha 23 mudamos de
# target_metadata = None
# para
target_metadata = models.SQLModel.metadata

# Na função `run_migrations_offline()` mudamos
#  url = config.get_main_option("sqlalchemy.url")
# para
url = settings.db.uri

# Na função `run_migration_online` mudamos
# connectable = engine_from_config...
```

```
#para
connectable = engine
```

Agora precisamos fazer só mais um ajuste edite `migrations/script.py.mako` e em torno da linha 10 adicione

```
#from alembic import op
#import sqlalchemy as sa
import sqlmodel  # linha NOVA
```

Agora sim podemos começar a usar o **alembic** para gerenciar as migrations, precisamos executar este comando dentro do container portando execute

```
$ docker-compose exec api /bin/bash
app@c5dd026e8f92:~/api$ # este é o shell dentro do container
```

> IMPORTANTE!!!: todos os comandos serão executados no shell dentro do container!!!

E dentro do prompt do container rode:

```
$ alembic revision --autogenerate -m "initial"
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
  Generating /home/app/api/migrations/versions/ee59b23815d3_initial.py ...  done
```

Repare que o alembic identificou o nosso model `User` e gerou uma migration inicial que fará a criação desta tabela no banco de dados.

Podemos aplicar a migration rodando dentro do container:

```
$ alembic upgrade head
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> ee59b23815d3, initial
```

E neste momento a tabela será criada no Postgres, podemos verificar se está funcionando ainda dentro do container:

> **DICA** pode usar um client como https://antares-sql.app para se conectar ao banco de dados.

```
$ ipython
>>>
```

Digite

```
from sqlmodel import Session, select
from pamps.db import engine
from pamps.models import User

with Session(engine) as session:
    print(list(session.exec(select(User))))
```

O resultado será uma lista vazia `[]` indicando que ainda não temos nenhum usuário no banco de dados.

Foi preciso muito **boilerplate** para conseguir se conectar ao banco de dados para facilitar a nossa vida vamos adicionar uma aplicação `cli` onde vamos poder executar tarefas administrativas no shell.

Criando a CLI base

Edite `pamps/cli.py`

```
import typer
from rich.console import Console
from rich.table import Table
from sqlmodel import Session, select

from .config import settings
from .db import engine
from .models import User

main = typer.Typer(name="Pamps CLI")


@main.command()
def shell():
    """Opens interactive shell"""
    _vars = {
        "settings": settings,
        "engine": engine,
        "select": select,
        "session": Session(engine),
        "User": User,
    }
    typer.echo(f"Auto imports: {list(_vars.keys())}")
    try:
        from IPython import start_ipython

        start_ipython(
            argv=["--ipython-dir=/tmp", "--no-banner"], user_ns=_vars
        )
    except ImportError:
        import code

        code.InteractiveConsole(_vars).interact()
```

```python
@main.command()
def user_list():
    """Lists all users"""
    table = Table(title="Pamps users")
    fields = ["username", "email"]
    for header in fields:
        table.add_column(header, style="magenta")

    with Session(engine) as session:
        users = session.exec(select(User))
        for user in users:
            table.add_row(user.username, user.email)

    Console().print(table)
```

E agora no shell do container podemos executar

```
$ pamps --help

 Usage: pamps [OPTIONS] COMMAND [ARGS]...

╭─ Options ──────────────────────────────────────────────────────────────────────
│ --install-completion        [bash|zsh|fish|powershell|pwsh]  Install completion f
│                                                              specified shell.
│                                                              [default: None]
│ --show-completion           [bash|zsh|fish|powershell|pwsh]  Show completion for
│                                                              specified shell, to
│                                                              customize the instal
│                                                              [default: None]
│ --help                                                       Show this message an
╰────────────────────────────────────────────────────────────────────────────────
╭─ Commands ─────────────────────────────────────────────────────────────────────
│ shell                         Opens interactive shell
│ user-list                     Lists all users
╰────────────────────────────────────────────────────────────────────────────────
```

E cada um dos comandos:

```
$ pamps user-list
    Pamps users
┏━━━━━━━━━━┳━━━━━━━┓
┃ username ┃ email ┃
┡━━━━━━━━━━╇━━━━━━━┩
└──────────┴───────┘
```

e

```
$ pamps shell
Auto imports: ['settings', 'engine', 'select', 'session', 'User']
```

```
In [1]: session.exec(select(User))
Out[1]: <sqlalchemy.engine.result.ScalarResult at 0x7fb1aa275ea0>

In [2]: settings.db
Out[2]: <Box: {'uri': 'postgresql://postgres:postgres@db:5432/pamps', 'connect_args
```

Ainda não temos usuários cadastrados pois ainda está faltando uma parte importante **criptografar as senhas** para os usuários.

Hash passwords

Precisamos ser capazes de encryptar as senhas dos usuários e para isso tem alguns requisitos, primeiro precisamos de uma chave em nosso arquivo de settings:

Edite `pamps/default.toml` e adicione ao final

```
[default.security]
# Set secret key in .secrets.toml
# SECRET_KEY = ""
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
REFRESH_TOKEN_EXPIRE_MINUTES = 600
```

Como o próprio comentário acima indica, vamos colocar uma secret key no arquivo `.secrets.toml` na raiz do repositório.

```
[development]
dynaconf_merge = true

[development.security]
# openssl rand -hex 32
SECRET_KEY = "ONLYFORDEVELOPMENT"
```

> NOTA: repare que estamos agora usando a seção `environment` e isso tem a ver com o modo como o dynaconf gerencia os settings, esses valores serão carregados apenas durante a execução em desenvolvimento.

Você pode gerar uma secret key mais segura se quiser usando

```
$ python -c "print(__import__('secrets').token_hex(32))"
b9483cc8a0bad1c2fe31e6d9d6a36c4a96ac23859a264b69a0badb4b32c538f8

# OU

$ openssl rand -hex 32
b9483cc8a0bad1c2fe31e6d9d6a36c4a96ac23859a264b69a0badb4b32c538f8
```

Agora vamos editar `pamps/security.py` e adicionar alguns elementos

```python
"""Security utilities"""
from passlib.context import CryptContext

from pamps.config import settings

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")


SECRET_KEY = settings.security.secret_key
ALGORITHM = settings.security.algorithm


def verify_password(plain_password, hashed_password) -> bool:
    """Verifies a hash against a password"""
    return pwd_context.verify(plain_password, hashed_password)


def get_password_hash(password) -> str:
    """Generates a hash from plain text"""
    return pwd_context.hash(password)


class HashedPassword(str):
    """Takes a plain text password and hashes it.
    use this as a field in your SQLModel
    class User(SQLModel, table=True):
        username: str
        password: HashedPassword
    """

    @classmethod
    def __get_validators__(cls):
        # one or more validators may be yielded which will be called in the
        # order to validate the input, each validator will receive as an input
        # the value returned from the previous validator
        yield cls.validate

    @classmethod
    def validate(cls, v):
        """Accepts a plain text password and returns a hashed password."""
        if not isinstance(v, str):
            raise TypeError("string required")

        hashed_password = get_password_hash(v)
        # you could also return a string here which would mean model.password
        # would be a string, pydantic won't care but you could end up with some
        # confusion since the value's type won't match the type annotation
        # exactly
        return cls(hashed_password)
```

E agora editaremos o arquivo `pamps/models/user.py`

No topo na linha 7

```
from pamps.security import HashedPassword
```

E no model mudamos o campo `password` na linha 18 para

```
password: HashedPassword
```

Adicionando usuários pelo cli

Agora sim podemos criar usuários via CLI, edite `pamps/cli.py`

No final adicione

```
@main.command()
def create_user(email: str, username: str, password: str):
    """Create user"""
    with Session(engine) as session:
        user = User(email=email, username=username, password=password)
        session.add(user)
        session.commit()
        session.refresh(user)
        typer.echo(f"created {username} user")
        return user
```

E no terminal do container execute

```
$ pamps create-user --help

 Usage: pamps create-user [OPTIONS] EMAIL USERNAME PASSWORD

 Create user

╭─ Arguments ──────────────────────────────────────────────────────────────────╮
│ *    email         TEXT  [default: None] [required]                          │
│ *    username      TEXT  [default: None] [required]                          │
│ *    password      TEXT  [default: None] [required]                          │
╰──────────────────────────────────────────────────────────────────────────────╯
```

E então

```
$ pamps create-user admin@admin.com admin 1234
created admin user

$ pamps user-list
         Pamps users
┏━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━┓
┃ username ┃ email                ┃
┡━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━┩
```

```
| admin    | admin@admin.com |
```

Agora vamos para a API

Adicionando rotas de usuários

Agora vamos criar endpoints na API para efetuar as operações que fizemos através da CLI, teremos as seguintes rotas:

- `GET /user/` - Lista todos os usuários
- `POST /user/` - Cadastro de novo usuário
- `GET /user/{username}/` - Detalhe de um usuário

> **TODO:** A exclusão de usuários por enquanto não será permitida mas no futuro você pode implementar um comando no CLI para fazer isso e também um endpoint privado para um admin fazer isso.

Serializers

A primeira coisa que precisamos é definir serializers, que são models intermediários usados para serializar e de-serializar dados de entrada e saída da API e eles são necessários pois não queremos export o model do banco de dados diretamente na API.

Em `pamps/models/user.py`

No topo na linha 4

```
from pydantic import BaseModel
```

No final após a linha 20

```
class UserResponse(BaseModel):
    """Serializer for User Response"""

    username: str
    avatar: Optional[str] = None
    bio: Optional[str] = None


class UserRequest(BaseModel):
    """Serializer for User request payload"""

    email: str
    username: str
    password: str
    avatar: Optional[str] = None
    bio: Optional[str] = None
```

E agora criaremos as URLS para expor esses serializers com os usuários edite `pamps/routes/user.py`

```
from typing import List

from fastapi import APIRouter
from fastapi.exceptions import HTTPException
from sqlmodel import Session, select

from pamps.db import ActiveSession
from pamps.models.user import User, UserRequest, UserResponse

router = APIRouter()


@router.get("/", response_model=List[UserResponse])
async def list_users(*, session: Session = ActiveSession):
    """List all users."""
    users = session.exec(select(User)).all()
    return users


@router.get("/{username}/", response_model=UserResponse)
async def get_user_by_username(
    *, session: Session = ActiveSession, username: str
):
    """Get user by username"""
    query = select(User).where(User.username == username)
    user = session.exec(query).first()
    if not user:
        raise HTTPException(status_code=404, detail="User not found")
    return user


@router.post("/", response_model=UserResponse, status_code=201)
async def create_user(*, session: Session = ActiveSession, user: UserRequest):
    """Creates new user"""
    db_user = User.from_orm(user)  # transform UserRequest in User
    session.add(db_user)
    session.commit()
    session.refresh(db_user)
    return db_user
```

Agora repare que estamos importando `ActiveSession` mas este objeto não existe em `pamps/db.py` então vamos criar

No topo de `pamps/db.py` nas linhas 2 e 3

```
from fastapi import Depends
from sqlmodel import Session, create_engine
```

No final de `pamps/db.py` após a linha 13

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


ActiveSession = Depends(get_session)
```

O objeto que `ActiveSession` é uma dependência para rotas do FastAPI quando usarmos este objeto como parâmetro de uma view o FastAPI vai executar de forma **lazy** este objeto e passar o retorno da função atrelada a ele como argumento da nossa view.

Neste caso teremos sempre uma conexão com o banco de dados dentro de cada view que marcarmos com `session: Session = ActiveSession`.

Agora podemos mapear as rotas na aplicação principal primeiro criamos um router principal que serve para agregar todas as rotas:

em `pamps/router/__init__.py`

```
from fastapi import APIRouter

from .user import router as user_router

main_router = APIRouter()

main_router.include_router(user_router, prefix="/user", tags=["user"])
```

E agora em `pamps/app.py`

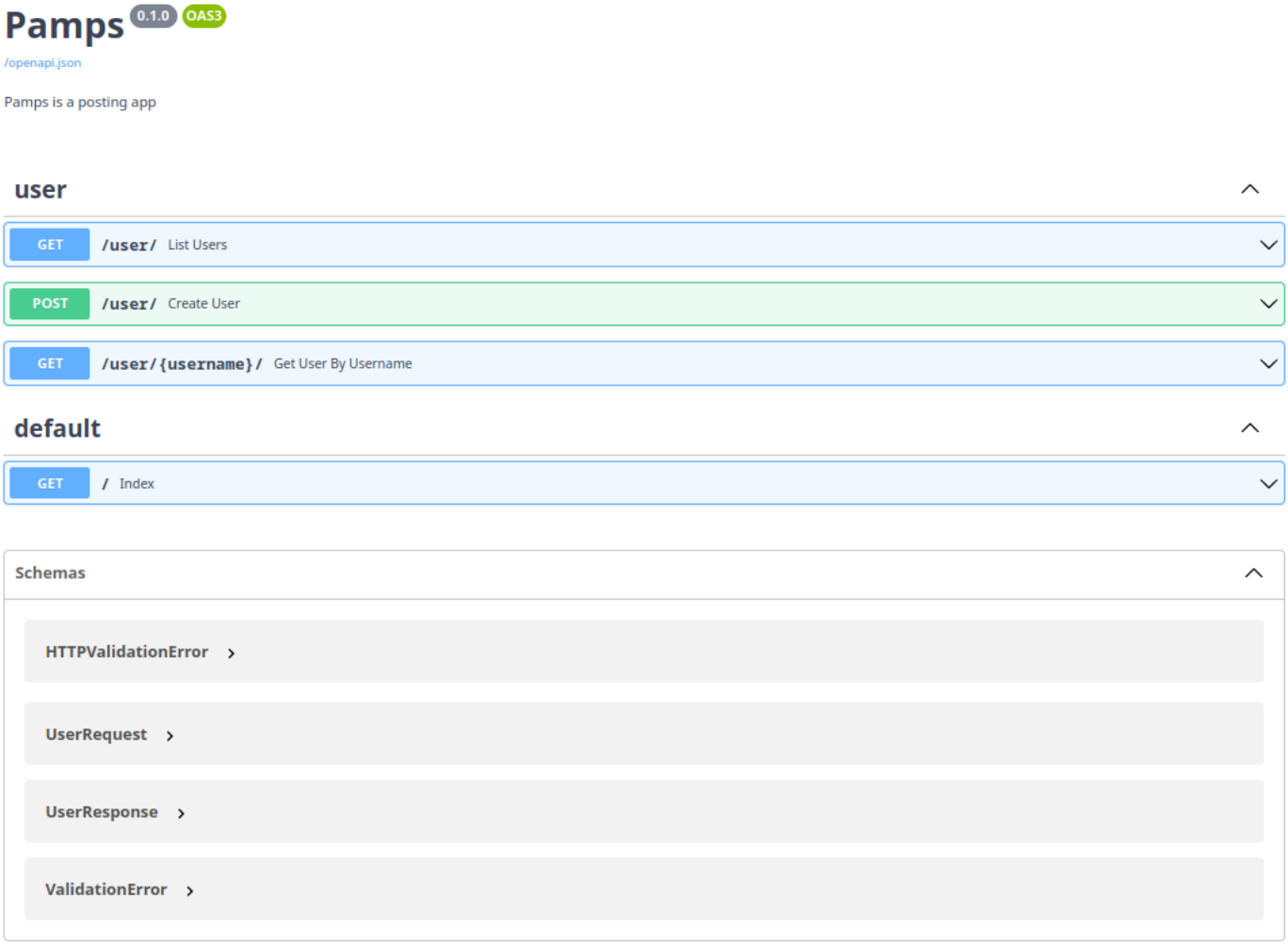
NO topo na linha 4

```
from .routes import main_router
```

Logo depois de `app = FastAPI(...` após a linha 11

```
app.include_router(main_router)
```

E agora sim pode acessar a API e verá as novas rotas prontas para serem usadas, http://0.0.0.0:8000/docs/



Pode tentar pela web interface ou via curl

Criar um usuário

```
curl -X 'POST' \
  'http://0.0.0.0:8000/user/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "email": "rochacbruno@gmail.com",
  "username": "rochacbruno",
  "password": "lalala",
  "avatar": "https://github.com/rochacbruno.png",
  "bio": "Programador"
}'
```

Pegar um usuário pelo ID

```
curl -X 'GET' \
  'http://0.0.0.0:8000/user/rochacbruno/' \
  -H 'accept: application/json'
```

```
{
  "username": "rochacbruno",
  "avatar": "https://github.com/rochacbruno.png",
```

```
  "bio": "Programador"
}
```

Listar todos

```
curl -X 'GET' \
  'http://0.0.0.0:8000/user/' \
  -H 'accept: application/json'
```

```
[
  {
    "username": "admin",
    "avatar": null,
    "bio": null
  },
  {
    "username": "rochacbruno",
    "avatar": "https://github.com/rochacbruno.png",
    "bio": "Programador"
  }
]
```

Autenticação

Agora que já podemos criar usuários é importante conseguirmos autenticar os usuários pois desta forma podemos começar a criar postagens via API

Esse será arquivo com a maior quantidade de código **boilerplate**.

No arquivo `pamps/auth.py` vamos criar as classes e funções necessárias para a implementação de JWT que é a autenticação baseada em token e vamos usar o algoritmo selecionado no arquivo de configuração.

pamps/auth.py

```
"""Token absed auth"""
from datetime import datetime, timedelta
from typing import Callable, Optional, Union

from fastapi import Depends, HTTPException, Request, status
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
from pydantic import BaseModel
from sqlmodel import Session, select

from pamps.config import settings
from pamps.db import engine
from pamps.models.user import User
from pamps.security import verify_password
```

```python
SECRET_KEY = settings.security.secret_key
ALGORITHM = settings.security.algorithm


oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


class Token(BaseModel):
    access_token: str
    refresh_token: str
    token_type: str


class RefreshToken(BaseModel):
    refresh_token: str


class TokenData(BaseModel):
    username: Optional[str] = None


def create_access_token(
    data: dict, expires_delta: Optional[timedelta] = None
) -> str:
    """Creates a JWT Token from user data"""
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire, "scope": "access_token"})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt


def create_refresh_token(
    data: dict, expires_delta: Optional[timedelta] = None
) -> str:
    """Refresh an expired token"""
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(minutes=15)
    to_encode.update({"exp": expire, "scope": "refresh_token"})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt


def authenticate_user(
    get_user: Callable, username: str, password: str
) -> Union[User, bool]:
    """Authenticate the user"""
    user = get_user(username)
    if not user:
        return False
```

```
    if not verify_password(password, user.password):
        return False
    return user


def get_user(username) -> Optional[User]:
    """Get user from database"""
    query = select(User).where(User.username == username)
    with Session(engine) as session:
        return session.exec(query).first()


def get_current_user(
    token: str = Depends(oauth2_scheme), request: Request = None, fresh=False
) -> User:
    """Get current user authenticated"""
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )

    if request:
        if authorization := request.headers.get("authorization"):
            try:
                token = authorization.split(" ")[1]
            except IndexError:
                raise credentials_exception

    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get("sub")

        if username is None:
            raise credentials_exception
        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception
    user = get_user(username=token_data.username)
    if user is None:
        raise credentials_exception
    if fresh and (not payload["fresh"] and not user.superuser):
        raise credentials_exception

    return user


async def get_current_active_user(
    current_user: User = Depends(get_current_user),
) -> User:
    """Wraps the sync get_active_user for sync calls"""
    return current_user


AuthenticatedUser = Depends(get_current_active_user)
```

```
async def validate_token(token: str = Depends(oauth2_scheme)) -> User:
    """Validates user token"""
    user = get_current_user(token=token)
    return user
```

> **NOTA**: O objeto `AuthenticatedUser` é uma dependência do FastAPI e é através dele que iremos garantir que nossas rotas estejas protegidas com token.

A simples presença das urls `/token` e `/refresh_token` fará o FastAPI incluir autenticação na API portanto vamos definir essas urls:

`pamps/routes/auth.py`

```
from datetime import timedelta

from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm

from pamps.auth import (
    RefreshToken,
    Token,
    User,
    authenticate_user,
    create_access_token,
    create_refresh_token,
    get_user,
    validate_token,
)
from pamps.config import settings

ACCESS_TOKEN_EXPIRE_MINUTES = settings.security.access_token_expire_minutes
REFRESH_TOKEN_EXPIRE_MINUTES = settings.security.refresh_token_expire_minutes

router = APIRouter()


@router.post("/token", response_model=Token)
async def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
):
    user = authenticate_user(get_user, form_data.username, form_data.password)
    if not user or not isinstance(user, User):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username, "fresh": True},
        expires_delta=access_token_expires,
    )
```

```
    refresh_token_expires = timedelta(minutes=REFRESH_TOKEN_EXPIRE_MINUTES)
    refresh_token = create_refresh_token(
        data={"sub": user.username}, expires_delta=refresh_token_expires
    )

    return {
        "access_token": access_token,
        "refresh_token": refresh_token,
        "token_type": "bearer",
    }


@router.post("/refresh_token", response_model=Token)
async def refresh_token(form_data: RefreshToken):
    user = await validate_token(token=form_data.refresh_token)

    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username, "fresh": False},
        expires_delta=access_token_expires,
    )

    refresh_token_expires = timedelta(minutes=REFRESH_TOKEN_EXPIRE_MINUTES)
    refresh_token = create_refresh_token(
        data={"sub": user.username}, expires_delta=refresh_token_expires
    )

    return {
        "access_token": access_token,
        "refresh_token": refresh_token,
        "token_type": "bearer",
    }
```

E agora vamos adicionar essas URLS ao router principal

`pamps/routes/__init__.py`

No topo na linha 3

```
from .auth import router as auth_router
```

E depois na linha 9

```
main_router.include_router(auth_router, tags=["auth"])
```

Vamos testar a aquisição de um token via curl ou através da UI.

```
curl -X 'POST' \
  'http://0.0.0.0:8000/token' \
  -H 'accept: application/json' \
```

```
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d 'grant_type=&username=admin&password=1234&scope=&client_id=&client_secret='
```

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbiIsImZyZXN
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbiIsImV4cC
  "token_type": "bearer"
}
```

Adicionando Models de conteúdo

Vamos definir a tabela e serializers para posts.

pamps/models/post.py

```
"""Post related data models"""

from datetime import datetime
from typing import TYPE_CHECKING, Optional

from pydantic import BaseModel, Extra
from sqlmodel import Field, Relationship, SQLModel

if TYPE_CHECKING:
    from pamps.models.user import User


class Post(SQLModel, table=True):
    """Represents the Post Model"""

    id: Optional[int] = Field(default=None, primary_key=True)
    text: str
    date: datetime = Field(default_factory=datetime.utcnow, nullable=False)

    user_id: Optional[int] = Field(foreign_key="user.id")
    parent_id: Optional[int] = Field(foreign_key="post.id")

    # It populates a `.posts` attribute to the `User` model.
    user: Optional["User"] = Relationship(back_populates="posts")

    # It populates `.replies` on this model
    parent: Optional["Post"] = Relationship(
        back_populates="replies",
        sa_relationship_kwargs=dict(remote_side="Post.id"),
    )
    # This lists all children to this post
    replies: list["Post"] = Relationship(back_populates="parent")

    def __lt__(self, other):
        """This enables post.replies.sort() to sort by date"""
        return self.date < other.date
```

```python
class PostResponse(BaseModel):
    """Serializer for Post Response"""

    id: int
    text: str
    date: datetime
    user_id: int
    parent_id: Optional[int]


class PostResponseWithReplies(PostResponse):
    replies: Optional[list["PostResponse"]] = None

    class Config:
        orm_mode = True


class PostRequest(BaseModel):
    """Serializer for Post request payload"""

    parent_id: Optional[int]
    text: str

    class Config:
        extra = Extra.allow
        arbitrary_types_allowed = True
```

Vamos adicionar uma back-reference em `User` para ser mais fácil obter todos os seus posts.

pamps/models/user.py

```python
# No topo do arquivo
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from pamps.models.post import Post


class User...
    ...
    # it populates the .user attribute on the Content Model
    posts: List["Post"] = Relationship(back_populates="user")
```

E agora vamos colocar o model Post na raiz do módulo models.

pamps/models/__init__.py

```python
from sqlmodel import SQLModel

from .post import Post
from .user import User
```

```
__all__ = ["User", "SQLModel", "Post"]
```

E para facilitar a vida vamos adicionar também ao `cli.py` dentro do comando shell no dict `_vars` adicione o model `Post`.

pamps/cli.py

```
from .models import Post, User
...
_vars = {
    ...
    "Post": Post,
}
```

Database Migration

Agora precisamos chamar o **alembic** para gerar a database migration relativa a nova tabela `post`.

Dentro do container shell

```
$ alembic revision --autogenerate -m "post"
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'post'
INFO  [alembic.ddl.postgresql] Detected sequence named 'user_id_seq' as owned by in
  Generating /home/app/api/migrations/versions/f9b269f8d5f8_post.py ...  done
```

e aplicamos com

```
$ alembic upgrade head
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade 4634e842ac70 -> f9b269f8d5f8, pos
```

Pode testar no cli dentro do container

```
$ pamps shell
Auto imports: ['settings', 'engine', 'select', 'session', 'User', 'Post']

In [1]: session.exec(select(Post)).all()
Out[1]: []
```

Adicionando rotas de conteúdo

Agora os endpoints para listar e adicionar posts

- `GET /post/` lista todos os posts
- `POST /post/` cria um novo post (exige auth)
- `GET /post/{id}` pega um post pelo ID com suas respostas
- `GET /post/user/{username}` Lista posts de um usuário especifico

`pamps/routes/post.py`

```python
from typing import List

from fastapi import APIRouter
from fastapi.exceptions import HTTPException
from sqlmodel import Session, select

from pamps.auth import AuthenticatedUser
from pamps.db import ActiveSession
from pamps.models.post import (
    Post,
    PostRequest,
    PostResponse,
    PostResponseWithReplies,
)
from pamps.models.user import User

router = APIRouter()


@router.get("/", response_model=List[PostResponse])
async def list_posts(*, session: Session = ActiveSession):
    """List all posts without replies"""
    query = select(Post).where(Post.parent == None)
    posts = session.exec(query).all()
    return posts


@router.get("/{post_id}/", response_model=PostResponseWithReplies)
async def get_post_by_post_id(
    *,
    session: Session = ActiveSession,
    post_id: int,
):
    """Get post by post_id"""
    query = select(Post).where(Post.id == post_id)
    post = session.exec(query).first()
    if not post:
        raise HTTPException(status_code=404, detail="Post not found")
    return post


@router.get("/user/{username}/", response_model=List[PostResponse])
```

```
async def get_posts_by_username(
    *,
    session: Session = ActiveSession,
    username: str,
    include_replies: bool = False,
):
    """Get posts by username"""
    filters = [User.username == username]
    if not include_replies:
        filters.append(Post.parent == None)
    query = select(Post).join(User).where(*filters)
    posts = session.exec(query).all()
    return posts


@router.post("/", response_model=PostResponse, status_code=201)
async def create_post(
    *,
    session: Session = ActiveSession,
    user: User = AuthenticatedUser,
    post: PostRequest,
):
    """Creates new post"""

    post.user_id = user.id

    db_post = Post.from_orm(post)  # transform PostRequest in Post
    session.add(db_post)
    session.commit()
    session.refresh(db_post)
    return db_post
```

Adicionamos as rotas de `post` em nosso router principal.

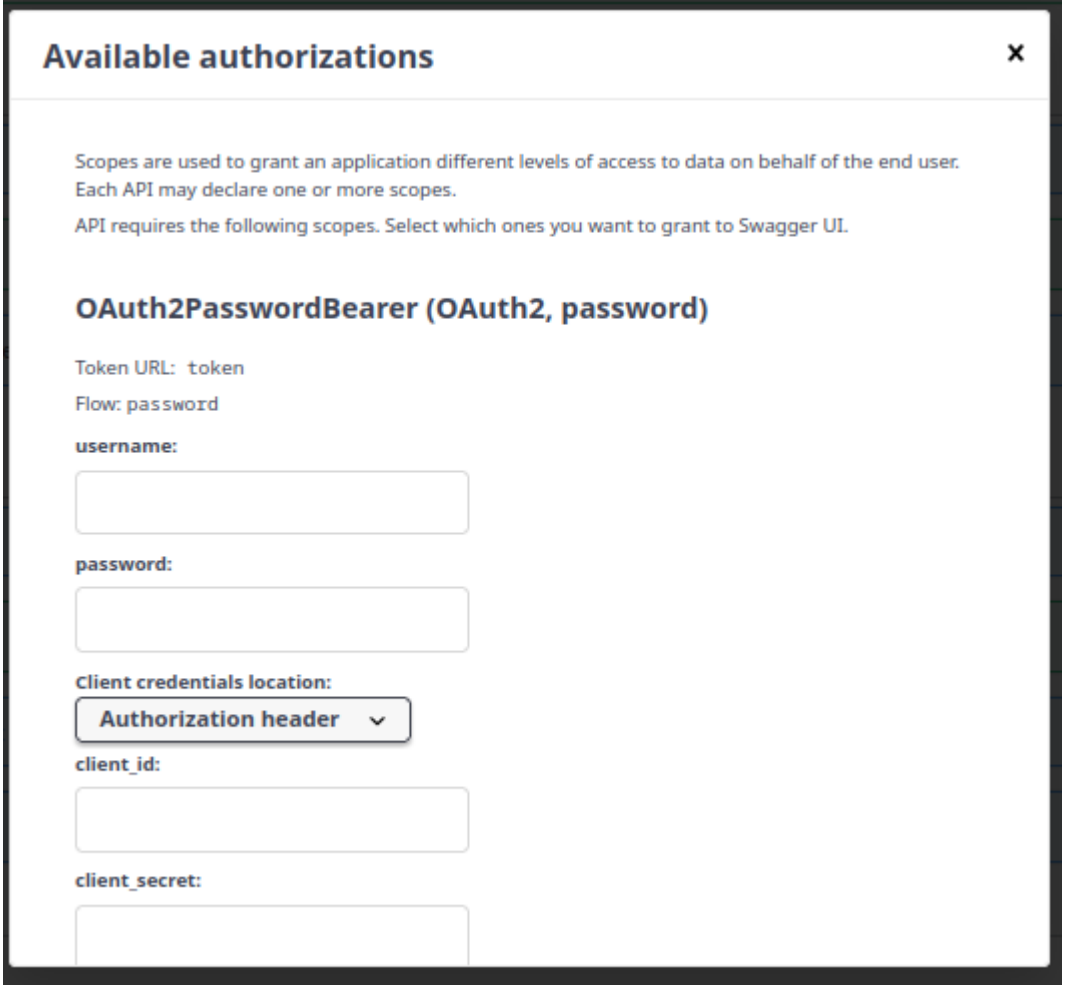
`pamps/routes/__init__.py` No topo linha 4

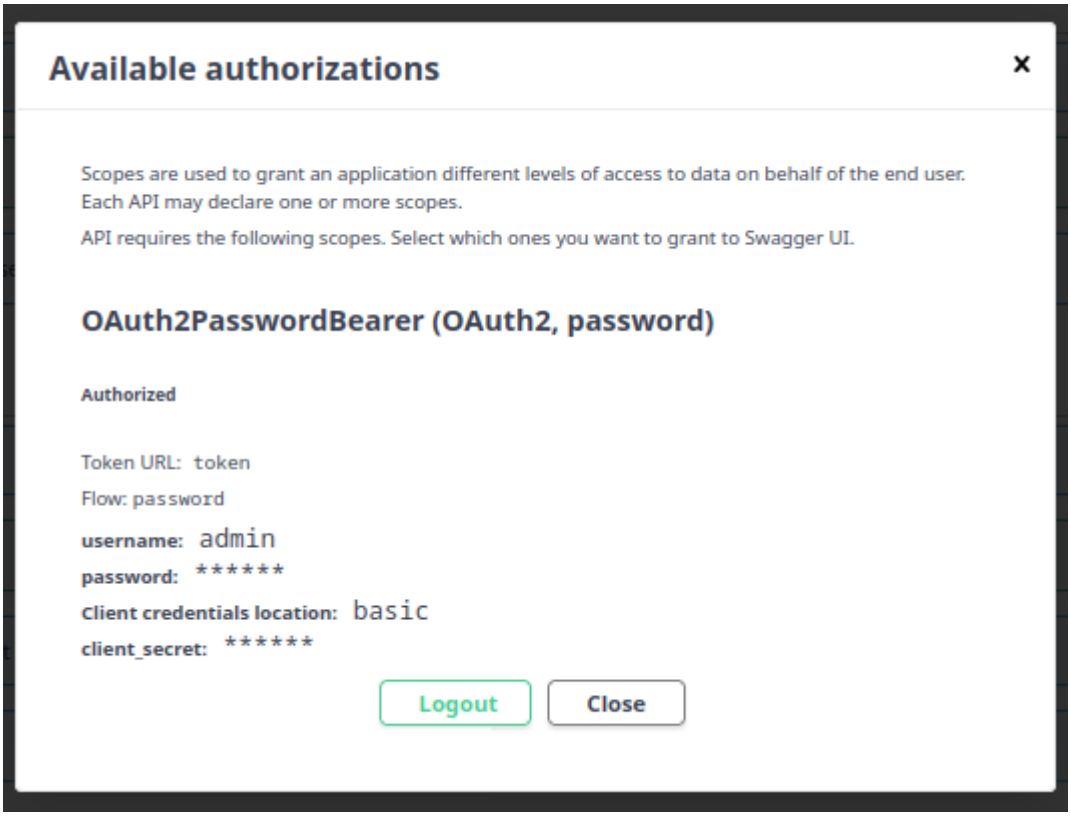
```
from .post import router as post_router
```

E no final na linha 11

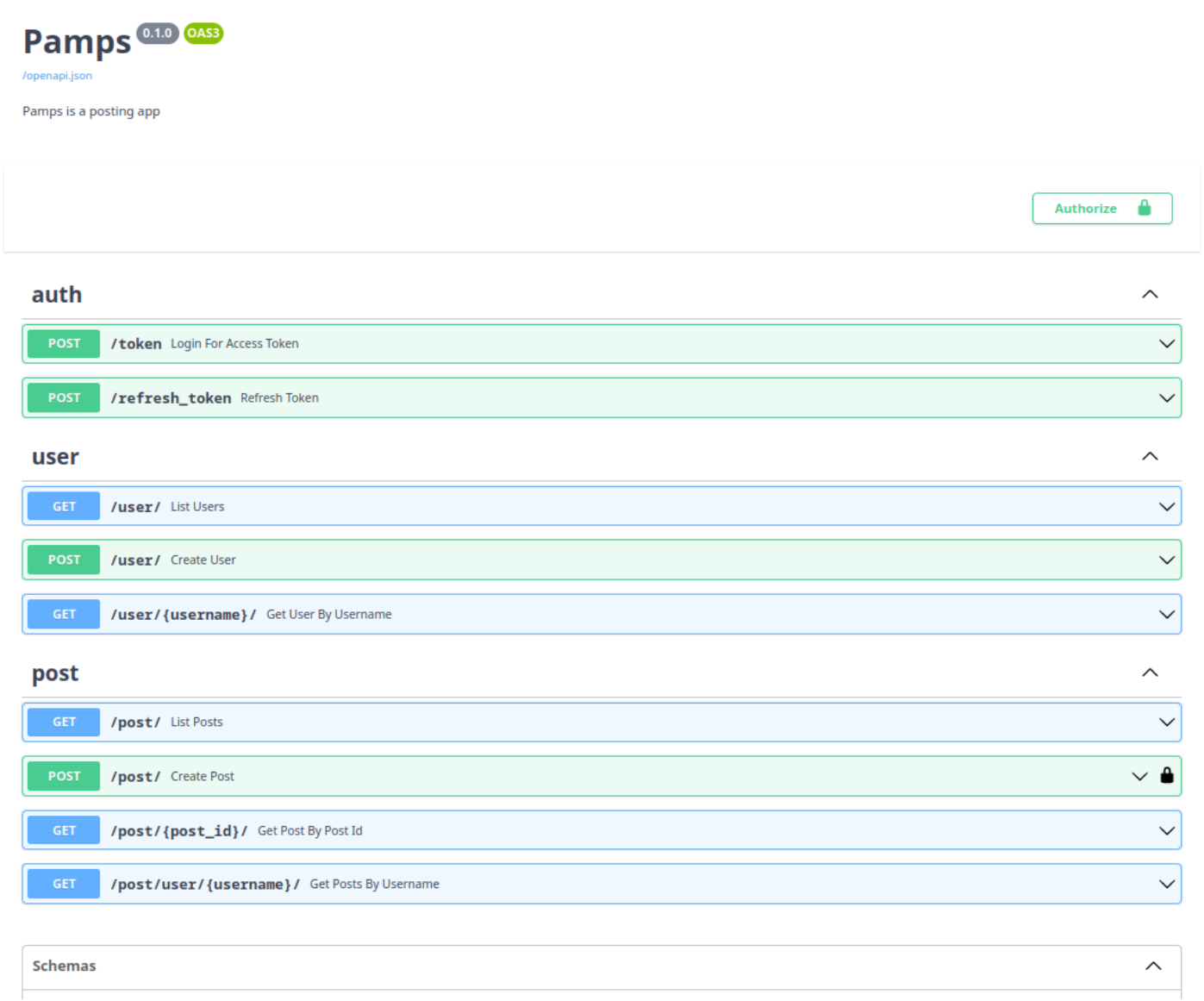
```
main_router.include_router(post_router, prefix="/post", tags=["post"])
```

Agora temos uma API quase toda funcional e pode testar clicando em `Authorize` usando as senhas criadas pelo CLI ou então crie um novo user antes de postar.


Available authorizations
Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.
API requires the following scopes. Select which ones you want to grant to Swagger UI.
OAuth2PasswordBearer (OAuth2, password)
Token URL: token
Flow: password
username:
password:
Client credentials location:
Authorization header
client_id:
client_secret:


Available authorizations
Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.
API requires the following scopes. Select which ones you want to grant to Swagger UI.
OAuth2PasswordBearer (OAuth2, password)
Authorized
Token URL: token
Flow: password
username: admin
password: ******
Client credentials location: basic
client_secret: ******
Logout
Close

A API final



> **NOTA** Ainda está faltando adicionar models e rotas para seguir usuários e para dar like em post.

Testando

O Pipeline de testes será

0. Garantir que o ambiente está em execução com o docker-compose
1. Garantir que existe um banco de dados `pamps_test` e que este banco está vazio.
2. Executar as migrations com alembic e garantir que funcionou
3. Executar os testes com Pytest
4. Apagar o banco de dados de testes

Vamos adicionar um comando `reset_db` no cli

> **NOTA** muito cuidado com esse comando!!!

edite `pamps/cli.py` e adicione ao final

```
@main.command()
def reset_db(
    force: bool = typer.Option(
```

```
        False, "--force", "-f", help="Run with no confirmation"
    )
):
    """Resets the database tables"""
    force = force or typer.confirm("Are you sure?")
    if force:
        SQLModel.metadata.drop_all(engine)
```

Em um ambiente de CI geralmente usamos `Github Actions` ou `Jenkins` para executar esses passos, em nosso caso vamos criar um script em bash para executar essas tarefas.

`test.sh`

```
#!/usr/bin/bash

# Start environment with docker-compose
PAMPS_DB=pamps_test docker-compose up -d

# wait 5 seconds
sleep 5

# Ensure database is clean
docker-compose exec api pamps reset-db -f
docker-compose exec api alembic stamp base

# run migrations
docker-compose exec api alembic upgrade head

# run tests
docker-compose exec api pytest -v -l --tb=short --maxfail=1 tests/

# Stop environment
docker-compose down
```

Para os tests vamos utilizar o Pytest para testar algumas rotas da API, com o seguinte fluxo

1. Criar usuário1
2. Obter um token para o usuário1
3. Criar um post1 com o usuário1
4. Criar usuario2
5. Obter um token para o usuario2
6. Responder o post1 com o usuario2
7. Consultar `/post` e garantir que apareçam os posts
8. COnsultar `/post/id` e garantir que apareça o post com a resposta
9. Consultar `/post/user/usuario1` e garantir que os posts são listados

Começamos configurando o Pytest

`tests/conftest.py`

```python
import os

import pytest
from fastapi.testclient import TestClient
from sqlalchemy.exc import IntegrityError

from pamps.app import app
from pamps.cli import create_user

os.environ["PAMPS_DB__uri"] = "postgresql://postgres:postgres@db:5432/pamps_test"


@pytest.fixture(scope="function")
def api_client():
    return TestClient(app)


def create_api_client_authenticated(username):

    try:
        create_user(f"{username}@pamps.com", username, username)
    except IntegrityError:
        pass

    client = TestClient(app)
    token = client.post(
        "/token",
        data={"username": username, "password": username},
        headers={"Content-Type": "application/x-www-form-urlencoded"},
    ).json()["access_token"]
    client.headers["Authorization"] = f"Bearer {token}"
    return client


@pytest.fixture(scope="function")
def api_client_user1():
    return create_api_client_authenticated("user1")


@pytest.fixture(scope="function")
def api_client_user2():
    return create_api_client_authenticated("user2")
```

E agora adicionamos os testes

```python
import pytest


@pytest.mark.order(1)
def test_post_create_user1(api_client_user1):
    """Create 2 posts with user 1"""
    for n in (1, 2):
        response = api_client_user1.post(
            "/post/",
```

```
            json={
                "text": f"hello test {n}",
            },
        )
        assert response.status_code == 201
        result = response.json()
        assert result["text"] == f"hello test {n}"
        assert result["parent_id"] is None


@pytest.mark.order(2)
def test_reply_on_post_1(api_client, api_client_user1, api_client_user2):
    """each user will add a reply to the first post"""
    posts = api_client.get("/post/user/user1/").json()
    first_post = posts[0]
    for n, client in enumerate((api_client_user1, api_client_user2), 1):
        response = client.post(
            "/post/",
            json={
                "text": f"reply from user{n}",
                "parent_id": first_post["id"],
            },
        )
        assert response.status_code == 201
        result = response.json()
        assert result["text"] == f"reply from user{n}"
        assert result["parent_id"] == first_post["id"]


@pytest.mark.order(3)
def test_post_list_without_replies(api_client):
    response = api_client.get("/post/")
    assert response.status_code == 200
    results = response.json()
    assert len(results) == 2
    for result in results:
        assert result["parent_id"] is None
        assert "hello test" in result["text"]


@pytest.mark.order(3)
def test_post1_detail(api_client):
    posts = api_client.get("/post/user/user1/").json()
    first_post = posts[0]
    first_post_id = first_post["id"]

    response = api_client.get(f"/post/{first_post_id}/")
    assert response.status_code == 200
    result = response.json()
    assert result["id"] == first_post_id
    assert result["user_id"] == first_post["user_id"]
    assert result["text"] == "hello test 1"
    assert result["parent_id"] is None
    replies = result["replies"]
    assert len(replies) == 2
    for reply in replies:
```

```
        assert reply["parent_id"] == first_post_id
        assert "reply from user" in reply["text"]


@pytest.mark.order(3)
def test_all_posts_from_user1(api_client):
    response = api_client.get("/post/user/user1/")
    assert response.status_code == 200
    results = response.json()
    assert len(results) == 2
    for result in results:
        assert result["parent_id"] is None
        assert "hello test" in result["text"]


@pytest.mark.order(3)
def test_all_posts_from_user1_with_replies(api_client):
    response = api_client.get(
        "/post/user/user1/", params={"include_replies": True}
    )
    assert response.status_code == 200
    results = response.json()
    assert len(results) == 3
```

E para executar os tests podemos ir na raiz do projeto **FORA DO CONTAINER**

```
$ chmod +x test.sh
```

e

```
$ ./test.sh
[+] Running 3/3
 ⠿ Network fastapi-workshop_default  Created                    0.0s
 ⠿ Container fastapi-workshop-db-1   Started                    0.5s
 ⠿ Container fastapi-workshop-api-1  Started                    1.4s

INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running stamp_revision f432efb19d1a ->
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> ee59b23815d3, initial
INFO  [alembic.runtime.migration] Running upgrade 4634e842ac70 -> f9b269f8d5f8, pos

========================= test session starts ==========================
platform linux -- Python 3.10.8, pytest-7.2.0, pluggy-1.0.0 -- /usr/local/bin/pytho
cachedir: .pytest_cache
rootdir: /home/app/api
plugins: order-1.0.1, anyio-3.6.2
collected 6 items

tests/test_api.py::test_post_create_user1 PASSED                [ 16%]
tests/test_api.py::test_reply_on_post_1 PASSED                  [ 33%]
tests/test_api.py::test_post_list_without_replies PASSED        [ 50%]
```

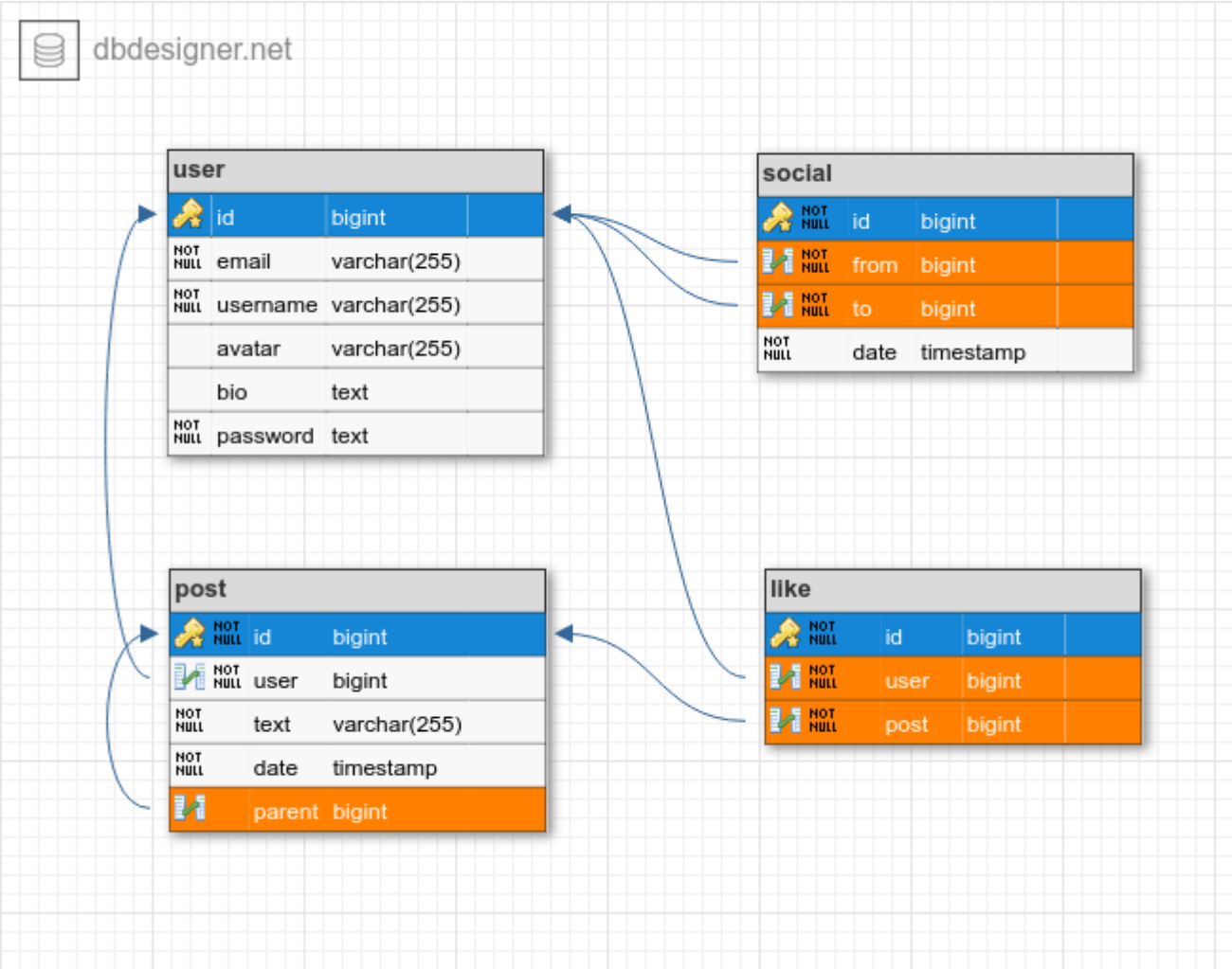
```
tests/test_api.py::test_post1_detail PASSED                          [ 66%]
tests/test_api.py::test_all_posts_from_user1 PASSED                  [ 83%]
tests/test_api.py::test_all_posts_from_user1_with_replies PASSED [100%]

========================== 6 passed in 1.58s ==========================

[+] Running 3/3
 ⠿ Container fastapi-workshop-api-1  Removed                          0.8s
 ⠿ Container fastapi-workshop-db-1   Removed                          0.6s
 ⠿ Network fastapi-workshop_default  Removed                          0.5s
```

Desafios finais

Lembra-se do nosso database?



Em nosso projeto está faltando adicionar os models para `Social` e `Like`

Social

O objetivo é que um usuário possa seguir outro usuário, para isso o usuário precisará estar autenticado e fazer um `post` request em `POST /user/follow/{id}` e sua tarefa é implementar esse endpoint armazenando o resultado na tabela `Social`.

- Passo 1 Edite `pamps/models/user.py` e adicione a tabela `Social` com toda a especificação e relacionamentos necessários. (adicione esse model ao `__init__.py`

- Passo 2 Execute dentro do shell do container `alembic revision --autogenerate -m 'social'` para criar a migração
- Passo 3 Aplique as migrations de tabela com `alembic upgrade head`
- Passo 4 Crie o Endpoint em `pamps/routes/user.py` com a lógica necessária e adicione ao router `__init__.py`
- Passo 5 Escreva um teste em `tests_user.py` para testar a funcionalidade de um usuário seguir outro usuário
- Passo 6 Em `pamps/routes/user.py` cria uma rota `/timeline` que ao acessar `/user/timeline` irá listar todos os posts de todos os usuários que o user autenticado segue.

Like

O objetivo é que um usuário possa enviar um like em um post e para isso precisará estar autenticado e fazer um `post` em `/post/{post_id}/like/` e a sua tarefa é implementar esse endpoint salvando o resultado na tabela `Like` .

- Passo1 Edite `pamps/model/post.py` e adicione a tabela `Like` com toda a especificação necessária com relacionamentos e adicione ao model `__init__.py`
- Passo 2 Execute dentro do shell do container `alembic revision --autogenerate -m 'like'` para criar a migração
- Passo 3 Aplique as migrations de tabela com `alembic upgrade head`
- Passo 4 Crie o endpoint em `pamps/routes/post.py` com a lógica necessária e adicione ao routes `__init__.py`
- Passo 5 Escreva um teste onde um user pode deixar um like em um post
- Passo 6 Em `pamps/routes/post.py` crie uma rota `/likes/{username}/` que retorne todos os posts que um user curtiu.

Desafio extra opcional

Use React ou VueJS para criar um front-end para esta aplicação :)