Refatorando a Estrutura do Projeto

> https://fastapidozero.dunossauro.com/06/

Objetivos da Aula:

- **Reestruturar o projeto para facilitar sua manutenção**
- Mover coisas de altenticação para um arquivo chamado `fast_zero/auth.py`
- Deixando em `fast_zero/secutiry.py` somente as validações de senha
- Remover constantes usados em código (`SECRET_KEY`, `ALGORITHM` e `ACCESS_TOKEN_EXPIRE_MINUTES`) usando a classe Settings
- Criar routers específicos

Parte 1

Routers

Routers

O FastAPI nos fornece um recurso útil chamado routers, que nos permite organizar e agrupar diferentes rotas em nossa aplicação. Em outras palavras, um router é um "subaplicativo" FastAPI que pode ser montado em uma aplicação principal.

Ao usar routers, podemos manter nosso código mais organizado e legível, especialmente à medida que nossa aplicação cresce e adicionamos mais rotas.

Criando um router para Users

A ideia é mover tudo que é referente a users para um arquivo único que vamos chamar de `fast_zero/routes/users.py`

```python
from fastapi import APIRouter

# imports ...

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

Criamos uma instância do APIRouter com o prefixo '/users'. Isso nos permitirá definir as rotas relacionadas aos usuários neste router, em vez de no aplicativo principal. O parâmetro `tags` ajuda na organização e documentação das rotas, associando-as a uma tag específica (neste caso, 'users'), que será exibida na interface de documentação automática gerada pelo FastAPI.

Implementando as rotas

Temos que alterar nossos endpoints. Agora o decorador deixa de ser `@app` e vira `@router`. Como já criamos os prefixos, as URLs não precisam mais iniciar com `/users`

```
@router.post('/', response_model=UserPublic, status_code=201)
@router.get('/', response_model=UserList)
@router.put('/{user_id}', response_model=UserPublic)
@router.delete('/{user_id}', response_model=Message)
```

Um router para Auth

Da mesma forma podemos criar um router para a rota de autenticação em `fast_zero/routers/auth.py`

```python
from fastapi import APIRouter

# outros imports

router = APIRouter(tags=['token'])


@router.post('/token', response_model=Token)
```

Juntando os routers no APP

O FastAPI oferece uma maneira fácil e direta de incluir routers em nossa aplicação principal. Isso nos permite organizar nossos endpoints de maneira eficiente e manter nosso arquivo `app.py` focado apenas em suas responsabilidades principais.

```python
from fastapi import FastAPI

from fast_zero.routes import auth, users

app = FastAPI()

app.include_router(users.router)
app.include_router(auth.router)


@app.get('/')
def read_root():
    return {'message': 'Olá Mundo!'}
```

Uma pausa para acessar o swagger agora!

> **http://localhost:8000**

Outra pausa para rodar os testes

E ver se tudo continua indo bem!

```
task test
```

Parte 2

Reestruturando os testes

Criando novos arquivos

Da mesma forma que dividimos as responsabilidades do app nos routers, também podemos deixar nossos arquivos de teste mais simples.

- `/tests/test_app.py`: Para testes relacionados ao aplicativo em geral
- `/tests/test_auth.py`: Para testes relacionados à autenticação e token
- `/tests/test_users.py`: Para testes relacionados às rotas de usuários

Claro, precisamos executar os testes de novo

```
task test
```

Parte 3

Usando o tipo `Annotated` para simplificar definições

O tipo Annotated

O FastAPI suporta um recurso fascinante da biblioteca nativa `typing`, conhecido como `Annotated`. Esse recurso prova ser especialmente útil quando buscamos simplificar a utilização de dependências.

Ao definir uma anotação de tipo, seguimos a seguinte formatação: `nome_do_argumento: Tipo = Depends(o_que_dependemos)`. Em todos os endpoints, acrescentamos a injeção de dependência da sessão da seguinte forma:

```
session: Session = Depends(get_session)
```

O tipo Annotated

O tipo `Annotated` nos permite combinar um tipo e os metadados associados a ele em uma única definição. Através da aplicação do FastAPI, podemos utilizar o `Depends` no campo dos metadados. Isso nos permite encapsular o tipo da variável e o `Depends` em uma única entidade, facilitando a definição dos endpoints.

Veja o exemplo a seguir:

```
from typing import Annotated

Session = Annotated[Session, Depends(get_session)]
CurrentUser = Annotated[User, Depends(get_current_user)]
```

Simplificando Users

```python
@router.post('/', response_model=UserPublic, status_code=201)
def create_user(user: UserSchema, session: Session):
# ...

@router.get('/', response_model=UserList)
def read_users(session: Session, skip: int = 0, limit: int = 100):
# ...

@router.put('/{user_id}', response_model=UserPublic)
def update_user(
    user_id: int,
    user: UserSchema,
    session: Session,
    current_user: CurrentUser
):
# ...

@router.delete('/{user_id}', response_model=Message)
def delete_user(user_id: int, session: Session, current_user: CurrentUser):
# ...
```

Simplificando Auth

```python
from typing import Annotated

# ...

OAuth2Form = Annotated[OAuth2PasswordRequestForm, Depends()]
Session = Annotated[Session, Depends(get_session)]

@router.post('/token', response_model=Token)
def login_for_access_token(form_data: OAuth2Form, session: Session):
#...
```

Claro, precisamos executar os testes de novo

```
task test
```

Parte 4

Movendo as constantes para variáveis de ambiente

O problema

Conforme mencionamos na aula sobre os 12 fatores, é uma boa prática manter as constantes que podem mudar dependendo do ambiente em variáveis de ambiente. Isso torna o seu projeto mais seguro e modular, pois você pode alterar essas constantes sem ter que modificar o código-fonte.

Por exemplo, temos estas constantes em nosso módulo `security.py`:

```
SECRET_KEY = 'your-secret-key'  # Isso é provisório, vamos ajustar!
ALGORITHM = 'HS256'
ACCESS_TOKEN_EXPIRE_MINUTES = 30
```

Estes valores não devem estar diretamente no código-fonte, então vamos movê-los para nossas variáveis de ambiente e representá-los na nossa classe `Settings`.

Adicionando as constantes a Settings

Já temos uma classe ideal para fazer isso em `zero_app/settings.py`. Vamos alterar essa classe para incluir estas constantes.

```
from pydantic_settings import BaseSettings, SettingsConfigDict


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

    DATABASE_URL: str
    SECRET_KEY: str
    ALGORITHM: str
    ACCESS_TOKEN_EXPIRE_MINUTES: int
```

Adicionando estes valores ao nosso arquivo `.env`.

```
DATABASE_URL="sqlite:///database.db"
SECRET_KEY="your-secret-key"
ALGORITHM="HS256"
ACCESS_TOKEN_EXPIRE_MINUTES=30
```

Com isso, podemos alterar o nosso código em `zero_app/security.py` para ler as constantes a partir da classe `Settings`.

Alterando o arquivo de security

```python
from fast_zero.settings import Settings

settings = Settings()
```

```python
def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(
        minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES
    )
    to_encode.update({'exp': expire})
    encoded_jwt = jwt.encode(
        to_encode, settings.SECRET_KEY, algorithm=settings.ALGORITHM
    )
    return encoded_jwt
```

Claro, precisamos executar os testes de novo

```
task test
```

Commit!

```
git add .
git commit -m "Refatorando estrutura do projeto: Criado routers para Users e Auth;\
movido constantes para variáveis de ambiente."
```