Autenticação e Autorização

> https://fastapidozero.dunossauro.com/05/

Objetivos dessa aula:

- Armazenamento seguro de senhas
- Autenticação
- Autorização
- Testes e fixtures

Armazenamento senhas de forma segura

Nossas senhas estão sendo armazenadas de forma limpa no banco de dados. Isso pode nos trazer diversos problemas:

- Erros eventuais: Uma simples alteração do schema e a senha estará exposta
- Vazamento de banco de dados:
  - Caso alguém consiga acesso ao banco de dados, pode ver as senhas
  - Pessoas costumam usar as mesmas senhas em N lugares

> https://monitor.firefox.com

Armazenas senhas de forma segura

Para isso vamos armazenar somente o hash das senhas e criar duas funções para controlar esse fluxo:

```
poetry add passlib[bcrypt]
```

- `Passlib` é uma biblioteca criada especialmente para manipular hashs de senhas.
- `Bcrypt` é um algorítimo de hash

Funções para gerenciar o hash

Vamos criar um novo arquivo no nosso pacote para gerenciar a parte de segurança.

`security.py`:

```python
# security.py
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=['bcrypt'])


def get_password_hash(password: str):
    return pwd_context.hash(password)


def verify_password(plain_password: str, hashed_password: str):
    return pwd_context.verify(plain_password, hashed_password)
```

Alterando o endpoint de cadastro

Agora precisamos alterar o endpoint de criação de users para sempre armazenar o hash da senha:

```python
# app.py
@app.post('/users/', response_model=UserPublic, status_code=201)
def create_user(user: UserSchema, session: Session = Depends(get_session)):
    # ...

    db_user = User(
        email=user.email,
        username=user.username,
        password=get_password_hash(user.password),
    )

    # ...
```

Alterando o endpoint de Update

Como agora as senhas estão sendo encriptadas durante o cadastro, caso o `User` altere a senha no endpoint de update, a senha precisa ser encriptada também:

```
@app.put('/users/{user_id}', response_model=UserPublic)
def update_user(
    user_id: int,
    user: UserSchema,
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
        # ...
    current_user.username = user.username
    current_user.password = get_password_hash(user.password)
    current_user.email = user.email
    session.commit()
    session.refresh(current_user)
    return current_user
```

Teste

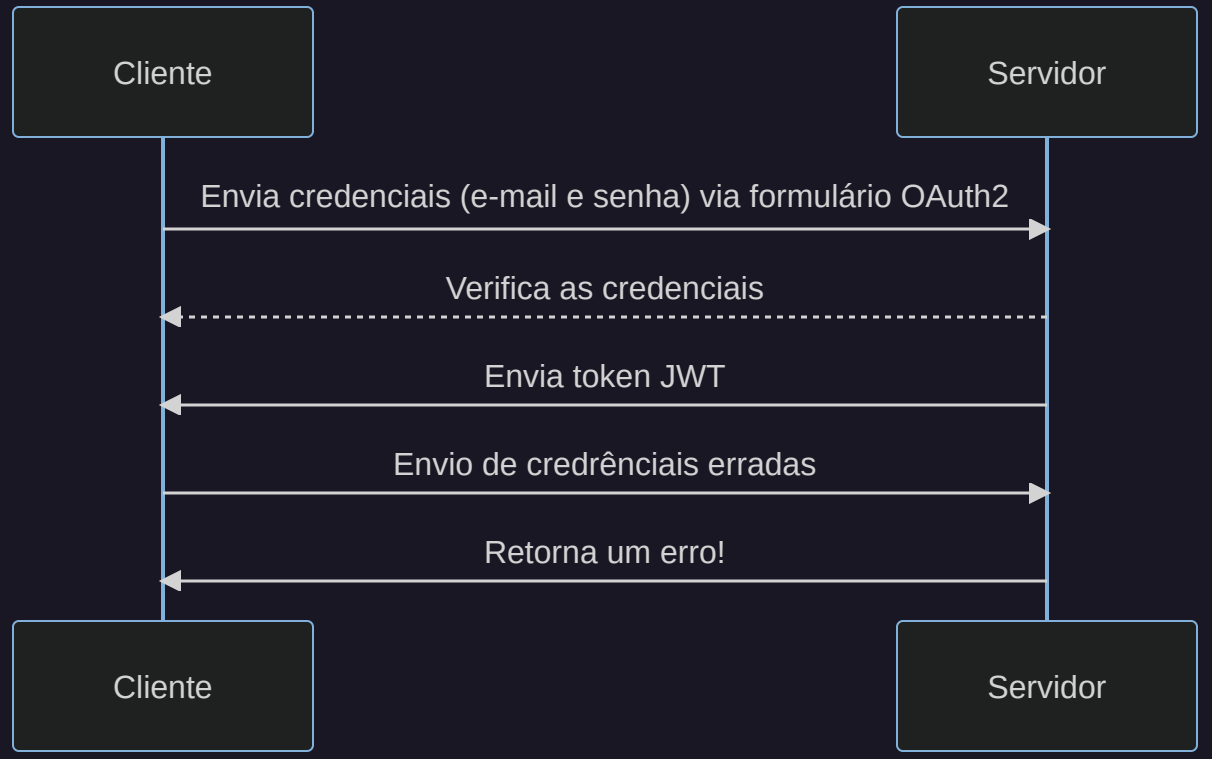
Em teoria, todos os testes devem continuar passando, pois não validamos a senha em nenhum momento:

```
task test
```

Parte 2: Autenticação

Autenticação

A ideia por trás da autenticação é dizer (comprovar) que você é você. No sentido de garantir que para usar a aplicação, você conhece as suas credenciais (email e senha no nosso caso).



Criando o endpoint

Para que os clientes se autentiquem na nossa aplicação, precisamos criar um endpoint que receba as credenciais. Vamos chamá-lo de `/token`.

Alguns pontos:

1. Precisamos de um schema de credenciais e um schema para o token
2. Validar se o email existe e se sua senha bate com o hash
    - Caso não batam, retornar um erro
3. Retornar um Token com uma duração de tempo! (30 minutos?)

Materiais para implementação

1. Precisamos de um schema de credenciais e um schema para o token
    - Para schema de credenciais, o FastAPI conta com o `OAuth2PasswordRequestForm`
    - Para o retorno, vamos criar um novo Schema chamado `Token`
2. Validar se o email existe e se sua senha bate com o hash
    - Para isso podemos injetar a `Session` com `Depends`
3. Retornar um Token com uma duração de tempo! (30 minutos?)
    - Para isso podemos usar o `datetime.timedelta`

OAuth2

OAuth2 É um protocolo aberto para autorização. O FastAPI disponibiliza alguns schemas prontos para usar OAuth2, como o `OAuth2PasswordRequestForm`. Traduzindo de forma literal: "Formulário de Requisição de Senha OAuth2"

```python
# app.py
from fastapi.security import OAuth2PasswordRequestForm

# ...

@app.post('/token')
def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    session: Session = Depends(get_session),
):
    ...
```

> Mostrar como isso ficará no Swagger!

O uso de formulários

Quando usamos formulários no FastAPI, como `OAuth2PasswordRequestForm`, precisamos instalar uma biblioteca para multipart:

```
poetry add python-multipart
```

Validando os dados!

```python
# app.py
@app.post('/token')
def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    session: Session = Depends(get_session),
):
    user = session.scalar(select(User).where(User.email == form_data.username))

    if not user or not verify_passwordform_data.password, user.password):
        raise HTTPException(
            status_code=400, detail='Incorrect email or password'
        )
```

Criando o endpoint

Para que os clientes se autentiquem na nossa aplicação, precisamos criar um endpoint que receba as credenciais. Vamos chamá-lo de `/token`.

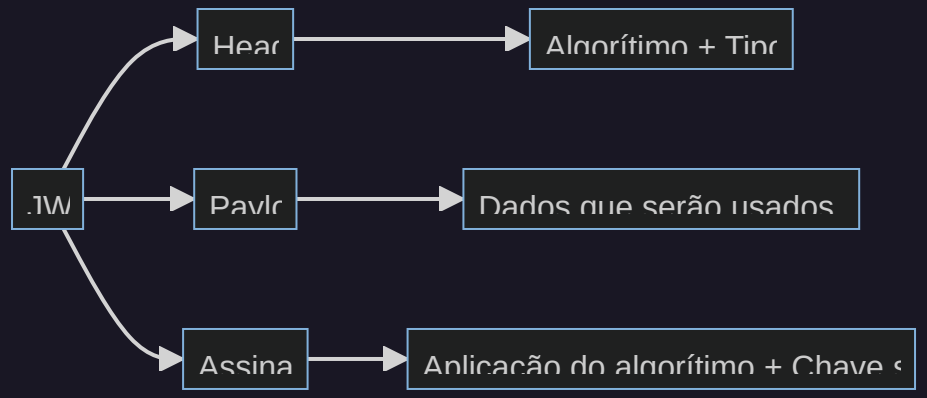
Alguns pontos:

1. ~~Precisamos de um schema de credenciais~~ e um schema para o token
2. ~~Validar se o email existe e se sua senha bate com o hash~~
3. Retornar um Token com uma duração de tempo! (30 minutos?)

O Token JWT

De forma simples, o JWT (Json Web Token) é uma forma de assinatura do servidor.

O token diz que o cliente foi autenticado com a assinatura **desse** servidor. Ele é divido em 3 partes:



Geração de tokens JWT com Python

Existem diversas bibliotecas para geração de tokens, usemos o `python-jose`.

```
poetry add python-jose[cryptography]
```

JOSE: Javascript Object Singin and Encryption

Olhando os tokens mais de perto

```
from jose import jwt

jwt.encode(dados, key)   # Os dados devem ser um dict, retorna o token

jwt.decode(token, key)    # Isso retorna o dict dos dados
```

Sobre a chave

A chave deve ser secreta, ela é o que define em conjunto com o algorítimo que foi assinado pelo nosso servidor. O Python tem uma biblioteca embutida que gera segredos:

```
import secrets

secretes.token_hex()   # Retorna um token randômico
```

investigando o token gerado

> https://jwt.io/#debugger-io

Aqui podemos ver o token e validar a integridade da assinatura.

O schema do token

```python
# schemas.py
class Token(BaseModel):
    access_token: str  # O token JWT que vamos gerar
    token_type: str    # O modelo que o cliente deve usar para Autorização
```

A geração do token

Agora que já temos o schema e o esqueleto do endpoint, podemos criar nossa função de criação de token em `security.py`:

```
from datetime import datetime, timedelta

from jose import jwt

SECRET_KEY = 'your-secret-key'  # Isso é privisório, vamos ajustar!

def create_access_token(data: dict):
    to_encode = data.copy()

    # Adiciona um tempo de 30 minutos para expiração
    expire = datetime.utcnow() + timedelta(minutes=30)
    to_encode.update({'exp': expire})

    encoded_jwt = jwt.encode(to_encode, SECRET_KEY)
    return encoded_jwt
```

Testando a geração de tokens

```python
# tests/test_security.py
from jose import jwt

from fast_zero.security import create_access_token, SECRET_KEY


def test_jwt():
    data = {'test': 'test'}
    token = create_access_token(data)

    decoded = jwt.decode(token, SECRET_KEY)

    assert decoded['test'] == data['test']
    assert decoded['exp']  # Testa se o valor de exp foi adicionado ao token
```

De volta ao endpoint `/token`

```python
# app.py
from fast_zero.schemas import ..., Token, ...
from fast_zero.security import create_access_token, ...

@app.post('/token', response_model=Token)
def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    session: Session = Depends(get_session),
):
    # ...
    access_token = create_access_token(data={'sub': user.email})

    return {'access_token': access_token, 'token_type': 'Bearer'}
```

Testando o endpoint `/token`

```python
# test_app.py
def test_get_token(client, user):
    response = client.post(
        '/token',
        data={'username': user.email, 'password': user.password},
    )
    token = response.json()

    assert response.status_code == 200
    assert token['token_type'] == 'Bearer'
    assert 'access_token' in token
```

Problema!

A fixture de `User` que estamos criando salva a senha limpa. Isso dá erro na hora de comparar se a senha está correta na criação do token.

```python
# conftest.py
from fast_zero.security import get_password_hash
# ...
@pytest.fixture
def user(session):
    user = User(
        username='test',
        email='test@test.com',
        password=get_password_hash('testtest'),  # Criando com a senha suja!
    )
    session.add(user)
    session.commit()
    session.refresh(user)

    return user
```

Problema 2!

Embora a senha agora consiga ser comparada, a senha que enviamos na requisição está indo suja também.

```
# conftest.py
@pytest.fixture
def user(session):
    password = 'testtest'
    user = User(
        username='test',
        email='test@test.com',
        password=get_password_hash(password),
    )
    session.add(user)
    session.commit()
    session.refresh(user)

    user.clean_password = password

    return user
```

Com isso, todos os testes devem voltar a passar:

```
tests/test_app.py::test_get_token PASSED
tests/test_app.py::test_root_deve_retornar_200_e_ola_mundo PASSED
tests/test_app.py::test_create_user PASSED
tests/test_app.py::test_read_users_empty PASSED
tests/test_app.py::test_read_users PASSED
tests/test_app.py::test_update_user PASSED
tests/test_app.py::test_delete_user PASSED
tests/test_models.py::test_create_user PASSED
tests/test_security.py::test_jwt PASSED
```

Parte 3: Autorização

Autorização

A ideia por trás da autorização é garantir que somente pessoas autorizadas possam executar determinadas operações. Como:

- Alterar (PUT): Queremos garantir que o cliente possa alterar somente sua conta
- Deletar: Queremos garantir que o cliente possa deletar somente a sua conta

Autorização

Agora que temos os tokens, podemos garantir que só clientes com uma conta já criada e logada possam ter acesso aos endpoints.

- Listar: Somente se estiver logado
- Deletar: Somente se a conta for sua
- Alterar: Somente se a conta for sua

Assim como nos formulários, o FastAPI também conta com um validador de Tokens passados nos cabeçalhos: `OAuth2PasswordBearer`

```python
# security.py
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session

from fast_zero.database import get_session

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


async def get_current_user(
    session: Session = Depends(get_session),
    token: str = Depends(oauth2_scheme),
):
    ...
```

Só com essa exigência de receber o token, podemos aplicar isso em nosso endpoint de listagem.

```python
# app.py
@app.get('/users/', response_model=UserList)
def list_users(
    session: Session = Depends(get_session),
    current_user=Depends(get_current_user),
):
    database = session.scalars(select(User)).all()
    return {'users': database}
```

> Mostar o cadeado no Swagger!

Porém

Ao rodar os testes...

Precisamos de um token para enviar aos endpoints agora!

```
@pytest.fixture
def token(client, user):
    response = client.post(
        '/token',
        data={'username': user.email, 'password': user.clean_password},
    )
    return response.json()['access_token']
```

Agora podemos enviar o token no cabeçalho da requisição

```python
# test_app.py
def test_read_users(client: TestClient, token):
    response = client.get(
        '/users/', headers={'Authorization': f'Bearer {token}'}
    )

    assert response.status_code == 200
    assert response.json() == {
        'users': [
            {
                'id': 1,
                'username': 'test',
                'email': 'test@test.com',
            },
        ],
    }
```

Funciona,

maaaaaaaaaasssssssssss não validamos o payload do token ainda!

A validação do JWT

```
async def get_current_user(...):
    credentials_exception = HTTPException(
        status_code=HTTPStatus.UNAUTHORIZED,
        detail='Could not validate credentials',
        headers={'WWW-Authenticate': 'Bearer'},
    )

    try:
        payload = jwt.decode(token, SECRET_KEY)
        username: str = payload.get('sub')
        if not username:
            raise credentials_exception
    except JWTError:
        raise credentials_exception

    # ...
```

Caso esteja tudo correto com o token:

```
async def get_current_user(...):
    # ...
    user = session.scalar(
        select(User).where(User.email == username)
    )

    if user is None:
        raise credentials_exception

    return user
```

Com isso podemos alterar os endpoints para depender do usuário corrente:

```python
@app.put('/users/{user_id}', response_model=UserPublic)
def update_user(
    user_id: int,
    user: UserSchema,
    session: Session = Depends(get_session),
    current_user: User = Depends(get_current_user),
):
    if current_user.id != user_id:
        raise HTTPException(status_code=400, detail='Not enough permissions')

    current_user.username = user.username
    current_user.password = user.password
    current_user.email = user.email
    session.commit()
    session.refresh(current_user)

    return current_user
```

Alteração do teste

```python
def test_update_user(client, user, token):
    response = client.put(
        f'/users/{user.id}',
        headers={'Authorization': f'Bearer {token}'},
        json={
            'username': 'bob',
            'email': 'bob@test.com',
            'password': 'mynewpassword',
        },
    )
    assert response.status_code == 200
    assert response.json() == {
        'username': 'bob',
        'email': 'bob@test.com',
        'id': 1,
    }
```

Para terminar...

Precisamos fazer isso no endpoint de DELETE

Commit!

```
git status
git add .
git commit -m "Protege os endpoints GET, PUT e DELETE com autenticação"
```