Autenticação e Autorização com JWT

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

Objetivos dessa aula:

- Um entendimento básico sobre JWT
- Implementar autenticação de usuários com JWT.
- Adicionar lógica de autorização aos endpoints de atualização e deleção.
- Utilizar a biblioteca Bcrypt para encriptar as senhas dos usuários.

JWT

O JWT é um padrão (RFC 7519) que define uma maneira compacta e autônoma de transmitir informações entre as partes de maneira segura. Essas informações são transmitidas como um objeto JSON que é digitalmente assinado usando um segredo (com o algoritmo HMAC) ou um par de chaves pública/privada usando RSA ou ECDSA.

JWT

A ideia por trás do JWT não é criptografar as mensagens, mas prover um tipo de "assinatura" de forma de a mensagem valide que ela foi gerada por aquele mesmo serviço.

O token é composto por 3 partes:

- Header: tipicamente consiste em dois componentes: o tipo de token, que é JWT neste caso, e o algoritmo de assinatura.
- Payload: é onde as reivindicações (os dados) são armazenadas. As reivindicações são informações que queremos transmitir e que são relevantes para a interação entre o cliente e o servidor.
- Assinatura: é utilizada para verificar que o remetente do JWT é quem afirma ser e para garantir que a mensagem não foi alterada ao longo do caminho.

A cara dos componentes

O header: Com o tipo de token e o algorítimo de assinatura

```
{"alg": "HS256", "typ": "JWT"}
```

O payload: com as informações que o tokem transmite

```
{"sub": "teste@test.com", "exp": 1690258153}
```

A assinatura: Com as informações em base64 + um segredo

```
HMACSHA256(
        base64UrlEncode(header) + "." +
        base64UrlEncode(payload),
        nosso-segredo
)
```

JWT

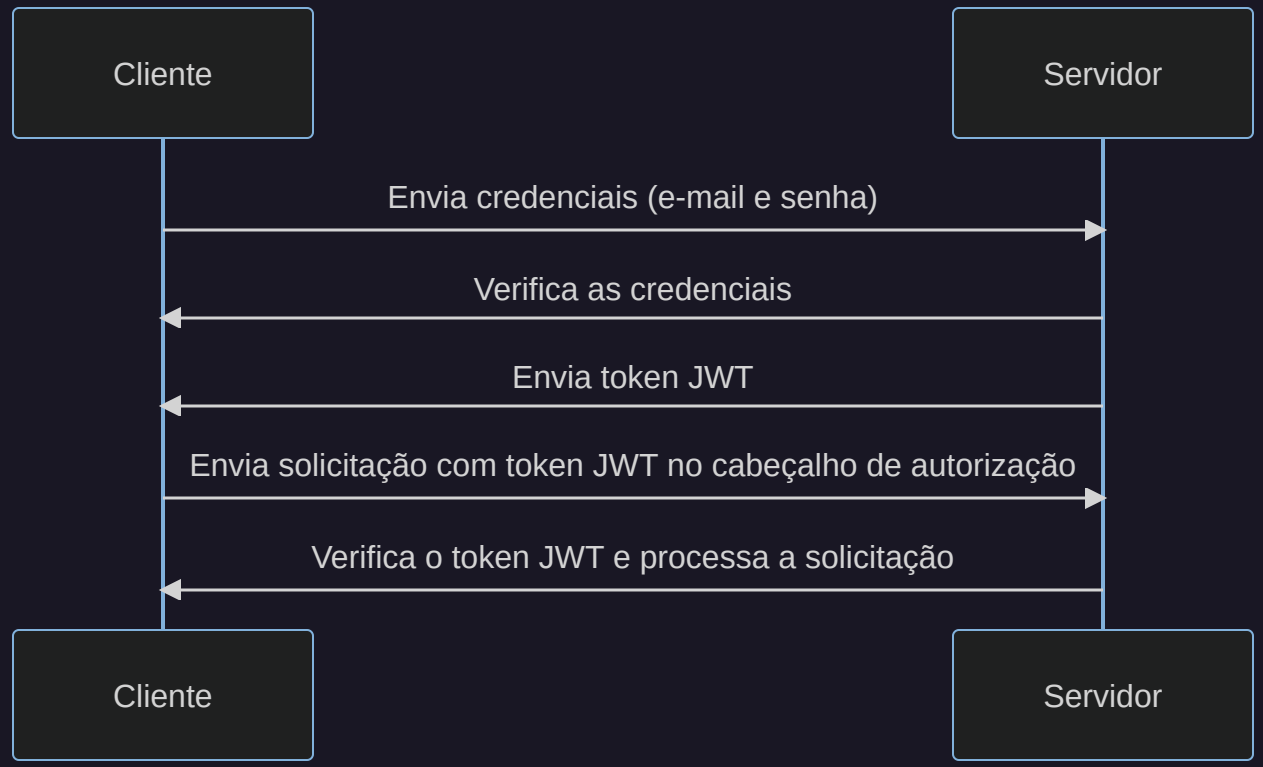
Essas três partes são separadas por pontos (.) e juntas formam um token JWT. Formando a estrutura: `HEADER.PAYLOAD.SIGNATURE` que formam um token parecido com

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0ZUB0ZXN0LmNvbSIsImV4cCI6MTY5MDI1ODE1M30.Nx0P_ornVwJBH_LLLVrlJoh6RmJeXR-Nr7YJ_mlGY04
```

Podemos ver as informações desse token no debugger no jwt.

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

Geração dos tokens



Geração dos tokens

1. O usuário envia suas credenciais (e-mail e senha) para o servidor em um endpoint de geração de token (`/token` por exemplo);
2. O servidor verifica as credenciais e, se estiverem corretas, gera um token JWT e o envia de volta ao cliente;
3. Nas solicitações subsequentes, o cliente deve incluir esse token no cabeçalho de autorização de suas solicitações. Como por exemplo: `Authorization: Bearer <token>`;
4. Quando o servidor recebe uma solicitação com um token JWT, ele pode verificar a assinatura e se o token é válido e não expirou, ele processa a solicitação.

Instalando o que vamos precisar para gerar os tokens

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

> **Curiosidade: JavaScript Object Signing and Encryption**

Criando uma função para gerar tokens

```python
from datetime import datetime, timedelta

from jose import jwt

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


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

Os tokens gerados pela nossa função terão a duração de 30 minutos

Testando!!!

vamos criar um arquivo chamado `tests/test_security.py` para efetuar esse teste:

```python
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, algorithms=['HS256'])

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

Parte 2

Criptografando a senha no banco de dados

Citografando a senha

Aproveitando que estamos trabalhando com uma parte que envolve segurança. Agora que vamos implementar a geração do token, é um bom momento para salvarmos nossa senha de forma "não limpa" no banco de dados.

Para isso vamos usar a passlib

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

As funções

No mesmo arquivo que criamos para segurança, vamos adicionar funções para criar o hash da senha e para verificar a validade do hash. Vamos usar o esquema `bcrypt` de hash:

```
from passlib.context import CryptContext

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


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)
```

Com isso podemos alterar nosso endpoint POST para criação usando o hash na senha:

```python
from fast_zero.security import get_password_hash

# ...

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

    hashed_password = get_password_hash(user.password)

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

    # ...
```

Endpoint para geração do token

O objetivo desse endpoint é fazer a validação dos dados de login. No nosso caso e-mail e senha.

Para isso precisamos de:

- Um formulário para o login
- A validação da senha no hash
- Um schema para a resposta do token

Endpoint para geração do token

- Um formulário para o login
  - O FastAPI tem um formulário pronto para isso

    `OAuth2PasswordRequestForm`
  - Para usar formulários precisamos instalar `python-multipart`
- A validação da senha no hash

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

- Um schema para a resposta do token

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

Implementação do endpoint

```python
from fastapi.security import OAuth2PasswordRequestForm
from fast_zero.schemas import Message, Token, UserList, UserPublic, UserSchema
from fast_zero.security import create_access_token, get_password_hash, verify_password

@app.post('/token', response_model=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:
        raise HTTPException(
            status_code=400, detail='Incorrect email or password'
        )

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

    access_token = create_access_token(data={'sub': user.email})

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

TESTANDO!!!

```
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 'access_token' in token
    assert 'token_type' in token
```

Problema!

A fixture de user tem que ser ajustada para funcionar em todos os contextos...

```python
from fast_zero.security import get_password_hash

# ...

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

    return user
```

Problema 2!

Isso vai fazer os outros testes falharem!

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

    user.clean_password = 'testtest'

    return user
```

Parte 3

Protegendo os endpoints!

Controle de acesso (autorização)

Agora com os tokens sendo gerados, precisamos criar limitações nos endpoints para:

- Só o usuário logado pode alterar sua própria conta no PUT
- Só o usuário logado pode remover sua própria conta no DELETE

Para isso vamos criar um controle com token que valide quem está usando o endpoint com base no payload do JWT

Chamaremos essa função de `get_current_user` em `security.py`:

```python
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")


async def get_current_user(
    session: Session = Depends(get_session),
    token: str = Depends(oauth2_scheme),
):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail='Could not validate credentials',
        headers={'WWW-Authenticate': 'Bearer'},
    )

    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str = payload.get('sub')
        if not username:
            raise credentials_exception
        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception

    user = session.scalar(
        select(User).where(User.email == token_data.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
```

Testes

Para simplificar, vamos criar uma nova fixture para gerar um token

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

Alteração do teste

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

Commit!

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