

.

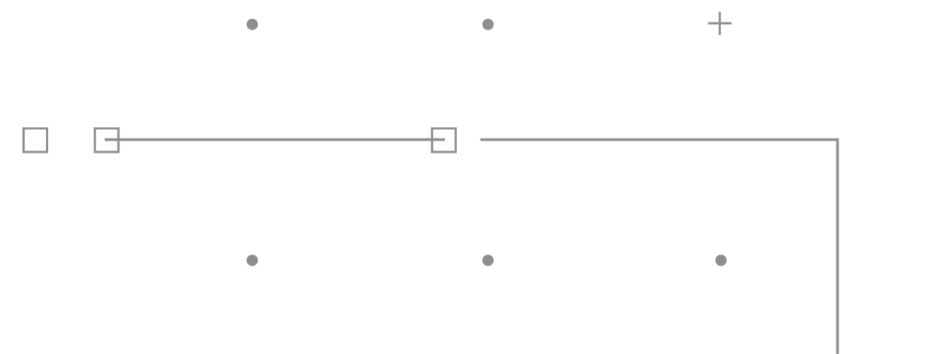
F

-

I

A

P




FastAPI + Autenticação



Introdução

Agora que montamos a estrutura da nossa aplicação, iremos colocar a parte de autenticação. Utilizamos esse recurso para evitar que nossos endpoints fiquem abertos e tentem realizar atividades maliciosas

Iremos montar uma autenticação simples com usuário e senha (credenciais) e uma autenticação utilizando token



Credenciais

Para realizarmos uma autenticação básica, o `FastAPI` já tem classes que fazem esse trabalho. Vamos criar um módulo chamado `auth.py` com o seguinte conteúdo:

```
from database import get_db
from database.crud import get_user_by_email
from fastapi.security import HTTPBasic, HTTPBasicCredentials

security = HTTPBasic()

def validate_user(db, email: str, password: str):
    user = get_user_by_email(db, email)
    if not user:
        return

    if not user.hashed_password == f"{password}notreallyhashed":
        return
    return user

def authenticate_user(credentials: HTTPBasicCredentials = Depends(security), db = Depends(get_db)):
    user = validate_user(db, credentials.username, credentials.password)
    if not user:
        raise HTTPException(
            status_code=HTTPStatus.UNAUTHORIZED,
            detail="Incorrect email or password",
            headers={"WWW-Authenticate": "Basic"},
        )
    return credentials.username
```

Credenciais

Importamos duas classes para fazer o tratamento da requisição: `HTTPBasic` e `HTTPBasicCredentials`. A `HTTPBasic` serve para validar se no cabeçalho da requisição contém os cabeçalhos necessários para tratar uma autenticação, enquanto o `HTTPBasicCredentials` se encarrega de tratar os dados e retornar nos seus respectivos campos.

Ou seja, o `HTTPBasic` vai validar e verificar se no cabeçalho da requisição tem o cabeçalho `Authentication: Basic {valor}` enquanto o `HTTPBasicCredentials` vai pegar o valor do cabeçalho e descompactar nas propriedades `username` e `password`

Credenciais

Vamos criar um endpoint que irá retornar os dados do usuário. E nele, definiremos que necessita de autenticação:

```
from auth import authenticate_user

@app.get("/users/me", response_model=schemas.User)
def read_current_user(user = Depends(authenticate_user)):
    return user
```

⚠ IMPORTANTE Esse endpoint deve ser feito antes do endpoint `/users/{user_id}`, pois o `FastAPI` segue a ordem que é colocado no código

Credenciais

Ao acessarmos a documentação <http://localhost:8000/docs>, podemos notar dois detalhes diferentes. No topo superior direito da página aparecerá o botão **Authorize**

FastAPI 0.1.0 OAS 3.1
[/openapi.json](#)

Authorize 

default ^

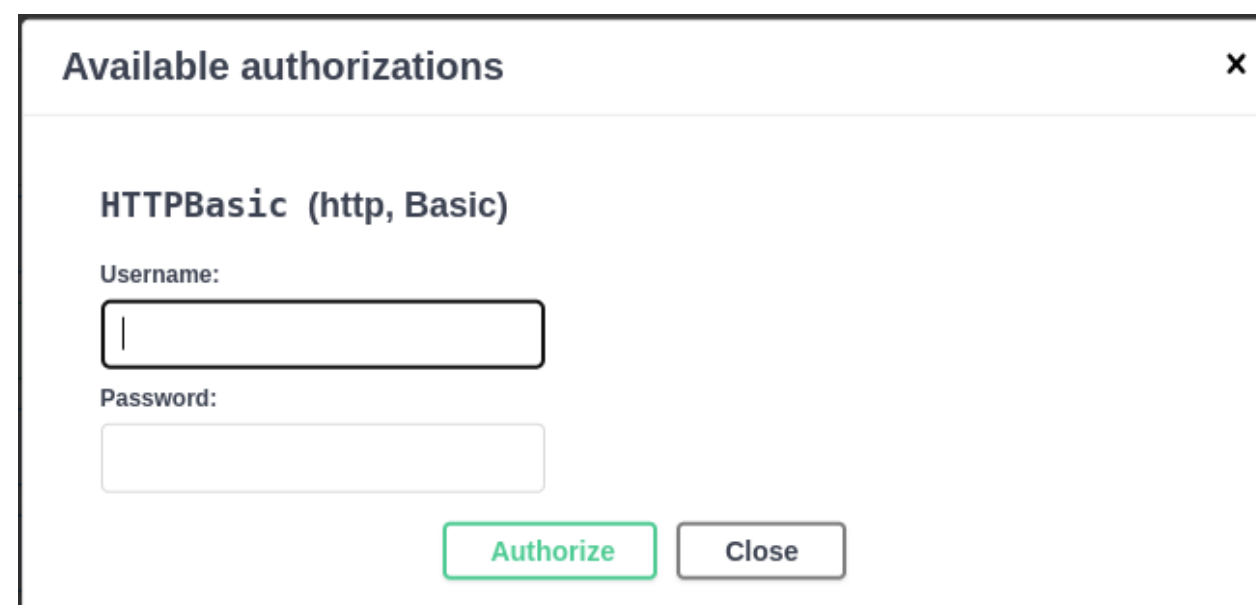
GET /users/ Read Users

POST /users/ Create User

GET /users/me Read Current User

Credenciais

Ao clicar no botão **Authorize**,



The screenshot shows a modal dialog box titled "Available authorizations" with a close button (X) in the top right corner. Inside the dialog, the text "HTTPBasic (http, Basic)" is displayed. Below this, there are two input fields: "Username:" and "Password:". The "Username:" field is currently empty with a cursor. At the bottom right of the dialog, there are two buttons: "Authorize" (highlighted with a green border) and "Close".

Basta entrar com os dados e clicar no botão **Authorize**

Após isso, basta testar o endpoint **/users/me** e ver o seu retorno

Credenciais

Um detalhe interessante desse tipo de autenticação é que podemos passar dois formatos. Note na documentação da API a forma que a requisição está sendo feita:

```
curl -X 'GET' \
  'http://localhost:8000/users/me' \
  -H 'accept: application/json' \
  -H 'Authorization: Basic c3RyaW5nOnN0cm1uZw=='
```

Credenciais

Veja que o valor do cabeçalho tem um monte de caracter que não parece ser o usuário e senha. Porque quando usamos esse tipo de autenticação, o usuário e senha são codificados no formato **Base64** . Implicitamente ele fez o que o código a seguir faz:

```
import base64
credentials = f'{username}:{password}'.encode()
data = base64.b64encode(credentials)
```

No tratamento dos dados, a classe **HTTPBasicCredentials** descodifica esse conteúdo, conforme o código a seguir:

```
import base64
data = b'bgVvbmFyZG86c2VuaGE='
credentials = base64.b64decode(credentials)
username, password = credentials.decode().split(":")
```

PyJWT

Para fazer autenticação com token, utilizamos o conceito de **JWT**, que vem de **JSON Web Token**.
Mais informações podem ser conferidas no [site oficial](#)

No Python, para realizar o processo de criptografia/descriptografia, utilizamos a biblioteca [PyJWT](#)
Para instala-la, execute o seguinte comando:

```
pip install pyjwt
```

PyJWT

Ele é bem simples de se utilizar.

Exemplo:

```
import jwt

key = "my secret key"
token = jwt.encode({'username': 'leonardo'}, key, algorithm='HS256')
"""
A variavel token vai ter algum valor parecido com essa estrutura:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imxlb25hcmRvIn0.0QD1FXlxSj1-4sd02ZkKvA28FKYngCNAHfuvo4L59WA
"""

data = jwt.decode(token, key, algorithms='HS256')

"""
A variavel data vai retornar ao seu conteúdo original
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imxlb25hcmRvIn0.0QD1FXlxSj1-4sd02ZkKvA28FKYngCNAHfuvo4L59WA
"""
```

PyJWT

Como se pode notar, existe o módulo `jwt` que tem as funções `encode` e `decode`.

No `encode` passamos o conteúdo que queremos criptografar, sempre sendo um `dict`, qual chave para usar na criptografia e qual algoritmo de criptografia será utilizado

No `decode` passamos o conteúdo que queremos descriptografar, qual chave será utilizada para realizar o procedimento. Um detalhe curioso é que pode ser passado uma lista de algoritmos que será utilizados para realizar a descriptografia

Caso queira saber quais algoritmos podem ser usados, você pode conferir a lista [aqui](#)

FastAPI

Vamos agora fazer todo o tratamento de criação e validação de um token.
Altere seu o módulo `auth.py`, colocando o seguinte conteúdo:

```
import jwt
from http import HTTPStatus

from pydantic import BaseModel
from fastapi import Depends, HTTPException
from fastapi.security import HTTPAuthorizationCredentials, HTTPBearer

from database import get_db
from database.crud import get_user_by_email

token = HTTPBearer()

SECRET_KEY = "secret"

class EmailAuthentication(BaseModel):
    email: str
    password: str

async def verify_token(token: HTTPAuthorizationCredentials = Depends(token), db=Depends(get_db)):
    try:
        data = jwt.decode(token.credentials, SECRET_KEY, algorithms=["HS256"])
    except jwt.exceptions.DecodeError:
        raise HTTPException(HTTPStatus.BAD_REQUEST, "Invalid token format")

    email = data.get("email")
    user = get_user_by_email(db, email)
    if not user:
        raise HTTPException(HTTPStatus.UNAUTHORIZED, "Invalid credentials")

    return data
```

FastAPI

Continuação do código anterior:

```
async def create_token(credentials: EmailAuthentication, db=Depends(get_db)):
    user = validate_user(db, credentials.email, credentials.password)
    if not user:
        raise HTTPException(HTTPStatus.UNAUTHORIZED, "Invalid user or password")

    token = jwt.encode({"email": credentials.email}, SECRET_KEY, algorithm="HS256")
    return token
```

FastAPI

Vamos as explicações:

Na linha 5, importamos algumas classes do FastAPI que já faz a abstração do cabeçalho de autorização que vem na requisição

A classe `HTTPBearer` define que é esperado na requisição o cabeçalho `Authorization: Bearer {token}`

A classe `HTTPAuthorizationCredentials` faz o trabalho de tratar o cabeçalho `Authorization: Bearer {token}`

Ambas são usadas para a validação de um token válido. Essa validação é feita na função `verify_token`, que como o nome sugere, irá validar os tokens enviados na requisição. Vale colocar que é tratada a exceção `jwt.exceptions.DecodeError` pois o token tem um formato específico.

Já a função `create_token` autentica o usuário e caso esteja correto, gera um token usando apenas o email dele

FastAPI

Agora no seu arquivo `main.py`, onde estão todos os endpoints, primeiro importe as funções do módulo `auth.py`

```
from auth import create_token, verify_token
```

Depois, adicione um endpoint para criar o token:

```
@app.post("/token")
def get_token(data=Depends(create_token)):
    return data
```

FastAPI

Vamos agora, pegar o endpoint `/items/` e definir que ele vai requerer autorização. Deixe o código da seguinte forma:

```
@app.get("/items/", response_model=list[schemas.Item], dependencies=[Depends(verify_token)])
def read_items(skip: int = 0, limit: int = 100, db: Session = Depends(get_db)):
    items = crud.get_items(db, skip=skip, limit=limit)
    return items
```

FastAPI

Pronto! Ao acessar a url <http://localhost:8000/docs>, no topo superior direito irá aparecer o botão **Authorize**. Ao clicar nesse botão, irá pedir um valor.

Esse valor pode ser gerado através do endpoint **/token**, que irá retornar um token no formato JWT e que pode ser utilizado no valor do **Authorize**

FastAPI

É possível gerar token que tem um prazo de validade e isso é bem simples!

Na função `encode` do `jwt` basta passar no dicionário a chave `exp`.

Veja como ficaria a função `create_token`:

```
async def create_token(credentials: EmailAuthentication, db=Depends(get_db)):
    user = authenticate_user(db, credentials.email, credentials.password)
    if not user:
        raise HTTPException(HTTPStatus.UNAUTHORIZED, "Invalid user or password")

    expiration_date = datetime.utcnow() + timedelta(minutes=15)
    token = jwt.encode({"email": credentials.email, "exp": expiration_date}, SECRET_KEY, algorithm="HS256")
    return token
```

⚠ Não esqueça de importar o `datetime` e o `timedelta` do módulo `datetime`

⚠ Nota-se que utiliza o método `utcnow`, que pega a hora atual no formato [UTC](#) (Universal Time Coordinated)

FastAPI

Agora na sua função `verify_token`, não se esqueça de tratar a exceção `jwt.exceptions.ExpiredSignatureError`, que é lançada quando o token expirou

```
async def verify_token(token: HTTPAuthorizationCredentials = Depends(token), db=Depends(get_db)):
    try:
        data = jwt.decode(token.credentials, SECRET_KEY, algorithms=["HS256"])
    except jwt.exceptions.DecodeError:
        raise HTTPException(HTTPStatus.BAD_REQUEST, "Invalid token format")
    except jwt.exceptions.ExpiredSignatureError:
        raise HTTPException(HTTPStatus.BAD_REQUEST, "Token expired")

    email = data.get("email")
    user = get_user_by_email(db, email)
    if not user:
        raise HTTPException(HTTPStatus.UNAUTHORIZED, "Invalid credentials")

    return data
```

Testes automatizados

E quanto aos testes automatizados? Como isso é feito?

Vamos considerar o `unittest` para realizar o testes, conforme falado na parte de testes automatizados.

Vamos criar o teste para obter os items no arquivo `test_main.py`:

```
from main import app, get_db, verify_token

...

class ItemTestCase(TestCase):
    def setUp(self):
        self.items = [{"title": "string", "description": "string", "id": 1, "owner_id": 0}]

    def override_verify_token(self):
        pass

    @mock.patch("database.crud.get_items")
    def test_get_items(self, get_items):
        app.dependency_overrides[verify_token] = self.override_verify_token
        get_items.return_value = self.items

        response = client.get("/items/")
        assert response.status_code == HTTPStatus.OK
        assert response.json() == self.items
```

Dessa forma, você consegue fazer o teste de forma onde a parte de autorização é ignorada

Testes automatizados

Mas é possível testar a autenticação? Com certeza. Vamos criar os testes para validação de autenticação e autorização

```
class AuthTestCase(TestCase):
    def setUp(self):
        self.user = User(email="fakeemail", hashed_password="password")
        self.token = jwt.encode({"email": self.user.email}, SECRET_KEY, algorithm=ALGORITHM)

    @mock.patch("auth.authenticate_user")
    def test_create_token(self, authenticate_user):
        authenticate_user.return_value = self.user
        response = client.post("/token/", json={"email": self.user.email, "password": self.user.hashed_password})
        token = response.json()
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        assert response.status_code == HTTPStatus.OK
        assert payload.get("email") == self.user.email
```

Neste teste, testamos se o recurso de criação de token está se comportando conforme o esperado. Note que a função `authenticate_user` é "mockada" para fazer de conta que o usuário existe

Testes automatizados

```
class AuthTestCase(TestCase):
    ...
    @mock.patch("auth.get_user_by_email")
    @mock.patch("database.crud.get_items")
    def test_valid_token(self, get_items, get_user_by_email):
        get_items.return_value = []
        get_user_by_email.return_value = self.user

        response = client.get("/items/", headers={"Authorization": f"Bearer {self.token}"})
        assert response.status_code == HTTPStatus.OK
```

No `test_valid_token`, validamos como seria uma requisição passando um token válido. As funções `get_user_by_email` e `get_items` são "mockadas" para simular que um usuário existe e que uma lista de items será retornada (apesar que o `return_value` é uma lista vazia)

Testes automatizados

Foram criados testes apenas para o "caminho feliz", mas sempre é bom criar testes para situações inesperadas

```
class AuthTestCase(TestCase):
    ...

    @mock.patch("auth.get_user_by_email")
    @mock.patch("database.crud.get_items")
    def test_invalid_token_format(self, get_items, get_user_by_email):
        get_items.return_value = []
        get_user_by_email.return_value = self.user

        response = client.get("/items/", headers={"Authorization": "Bearer blah"})
        assert response.status_code == HTTPStatus.BAD_REQUEST
        assert response.json() == {"detail": "Invalid token format"}
```

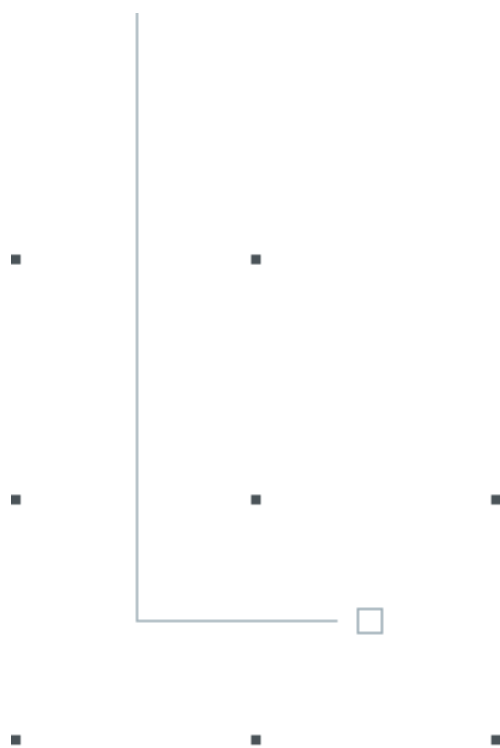
O `test_invalid_token_format` vai validar uma requisição que será feita com um token cujo formato não é o esperado. E caso isso aconteça, deve ser retornado na resposta essa informação

Testes automatizados

```
class AuthTestCase(TestCase):
    ...
    @mock.patch("auth.get_user_by_email")
    @mock.patch("database.crud.get_items")
    def test_invalid_token(self, get_items, get_user_by_email):
        get_items.return_value = []
        get_user_by_email.return_value = self.user
        token = jwt.encode({"email": self.user.email}, "any secret", algorithm=ALGORITHM)

        response = client.get("/items/", headers={"Authorization": f"Bearer {token}"})
        assert response.status_code == HTTPStatus.BAD_REQUEST
        assert response.json() == {"detail": "Invalid token format"}
```

O `test_invalid_token` vai validar uma requisição que será feita com um token cujo formato é o esperado, porém é um token inválido. E caso isso aconteça, deve ser retornado na resposta essa informação



.

F

-

I

A

P

