

PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática
Lucas Sampaio Leite



Autenticação e autorização

- Autenticação e autorização são dois processos distintos, mas complementares, que garantem a segurança e o acesso controlado a recursos.
- A autenticação verifica a identidade de um usuário ou dispositivo, enquanto a autorização determina quais recursos esse usuário ou dispositivo pode acessar e com quais níveis de permissão.
- Esses conceitos são fundamentais na segurança de APIs.

Autenticação (Authentication)

- Processo de verificar a identidade de um usuário ou sistema.
 - Responde: “Quem é você?”
 - Exemplo: login com usuário e senha, biometria, token.



Autorização (Authorization)

- Processo de definir os privilégios e permissões de um usuário autenticado.
 - Responde: “O que você pode fazer?”
 - Exemplo: um usuário comum não pode acessar funções de administrador.



JWT (JSON Web Token)

- JWT (JSON Web Token) é um padrão para autenticação e troca segura de informações entre sistemas, usando objetos em formato JSON.
- Padrão aberto (RFC 7519).
- Como funciona:
 - O usuário faz login → o servidor gera um JWT e envia para o cliente.
 - O cliente guarda o token (ex.: no navegador, postman, etc).
 - Em cada requisição, o cliente envia o JWT → o servidor valida a assinatura e autoriza ou não a ação.



Características do JWT

- **Compacto:** pode ser transmitido facilmente em URLs, parâmetros POST ou em um cabeçalho HTTP.
- **Autocontido:** o payload armazena todas as informações necessárias sobre o usuário, reduzindo a necessidade de múltiplas consultas ao banco de dados.
- **Padrão em APIs REST:** amplamente utilizado e suportado em diversas linguagens e frameworks.
- **Flexibilidade:** pode carregar tanto dados de autenticação quanto informações adicionais.



Características do JWT

- Um token JWT é dividido em 3 partes, separadas por pontos (.):
 - **Header (Cabeçalho)**: informa o algoritmo de criptografia e o tipo do token.
 - **Payload (Corpo/Dados)**: contém as informações (claims) do usuário, como ID, permissões etc.
 - **Signature (Assinatura)**: garante a integridade do token e que não foi alterado.



xxxxxx.yyyyyy.zzzzzz

Flask-JWT-Extended

- O Flask-JWT-Extended é uma extensão do Flask que facilita a implementação de autenticação e autorização baseada em JWT.
- Ele não serve para armazenar senhas, mas sim para gerar, validar e gerenciar tokens JWT que serão usados para proteger os endpoints da sua API.
- Instalação: `pipenv install flask-jwt-extended`

Flask-JWT-Extended

- Documentação: <https://flask-jwt-extended.readthedocs.io/en/stable/>

Flask-JWT-Extended's Documentation

- [Installation](#)
- [Basic Usage](#)
- [Automatic User Loading](#)
- [Storing Additional Data in JWTs](#)
- [Partially protecting routes](#)
- [JWT Locations](#)
 - [Headers](#)
 - [Cookies](#)
 - [Query String](#)
 - [JSON Body](#)
- [Refreshing Tokens](#)
 - [Implicit Refreshing With Cookies](#)
 - [Explicit Refreshing With Refresh Tokens](#)
 - [Token Freshness Pattern](#)
- [JWT Revoking / Blocklist](#)
 - [Redis](#)
 - [Database](#)
 - [Revoking Refresh Tokens](#)

Flask-JWT-Extended

```
• lucas@lucas-Inspiron-15-3520:~/Dropbox/IF_Baiano/web_II/codes/Projeto_Flask_Blog$ pipenv graph
Flask-JWT-Extended==4.7.1
├── Flask
│   ├── blinker
│   ├── click
│   ├── itsdangerous
│   ├── Jinja2
│   │   └── MarkupSafe
│   ├── MarkupSafe
│   ├── Werkzeug
│   │   └── MarkupSafe
│   └── PyJWT
│       └── Werkzeug
│           └── MarkupSafe
├── Flask-SQLAlchemy==3.1.1
│   ├── Flask
│   │   ├── blinker
│   │   ├── click
│   │   ├── itsdangerous
│   │   ├── Jinja2
│   │   │   └── MarkupSafe
│   │   ├── MarkupSafe
│   │   ├── Werkzeug
│   │   │   └── MarkupSafe
│   └── SQLAlchemy
│       ├── greenlet
│       └── typing_extensions
```

Flask-JWT-Extended's Documentation

- 
- [Installation](#)
 - [Basic Usage](#)
 - [Automatic User Loading](#)
 - [Storing Additional Data in JWTs](#)
 - [Partially protecting routes](#)
 - [JWT Locations](#)
 - [Headers](#)
 - [Cookies](#)
 - [Query String](#)
 - [JSON Body](#)
 - [Refreshing Tokens](#)
 - [Implicit Refreshing With Cookies](#)
 - [Explicit Refreshing With Refresh Tokens](#)
 - [Token Freshness Pattern](#)

Integrando suporte a JWT ao app Flask

```
import click
from flask import Flask, current_app
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
→ from flask_jwt_extended import JWTManager
```

```
from datetime import datetime
from sqlalchemy import func
```

```
✓ class Base(DeclarativeBase):
    pass
```

```
db = SQLAlchemy(model_class=Base)
→ jwt = JWTManager()
```

Integrando suporte a JWT ao app Flask

```
import click
from flask import Flask, current_app
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
from flask_jwt_extended import JWTManager
```

```
from datetime import datetime
from sqlalchemy import func
```

```
class Base(DeclarativeBase):
    pass
```

```
db = SQLAlchemy(model_class=Base)
jwt = JWTManager()
```

A classe `JWTManager()` é a responsável por integrar o suporte a JWT no app Flask.

Integrando suporte a JWT ao app Flask

```
def create_app(test_config=None):  
    # create and configure the app  
    app = Flask(__name__, instance_relative_config=True)  
    app.config.from_mapping(  
        SECRET_KEY='dev',  
        SQLALCHEMY_DATABASE_URI='sqlite:///blog.sqlite',  
        → JWT_SECRET_KEY="super-secret"  
    )
```

```
app.cli.add_command(init_db_command)
```

```
→ db.init_app(app)  
  jwt.init_app(app)
```

```
from controllers import user  
app.register_blueprint(user.app)
```

Integrando suporte a JWT ao app Flask

```
def create_app(test_config=None):  
    # create and configure the app  
    app = Flask(__name__, instance_relative_config=True)  
    app.config.from_mapping(  
        SECRET_KEY='dev',  
        SQLALCHEMY_DATABASE_URI='sqlite:///blog.sqlite',  
        JWT_SECRET_KEY="super-secret"  
    )
```

O JWT_SECRET_KEY é a chave secreta usada para assinar digitalmente os tokens JWT.

Ao gerar um token, ele não é apenas um JSON comum. Ele é assinado usando um algoritmo criptográfico (por padrão HMAC SHA-256) junto com a chave secreta.

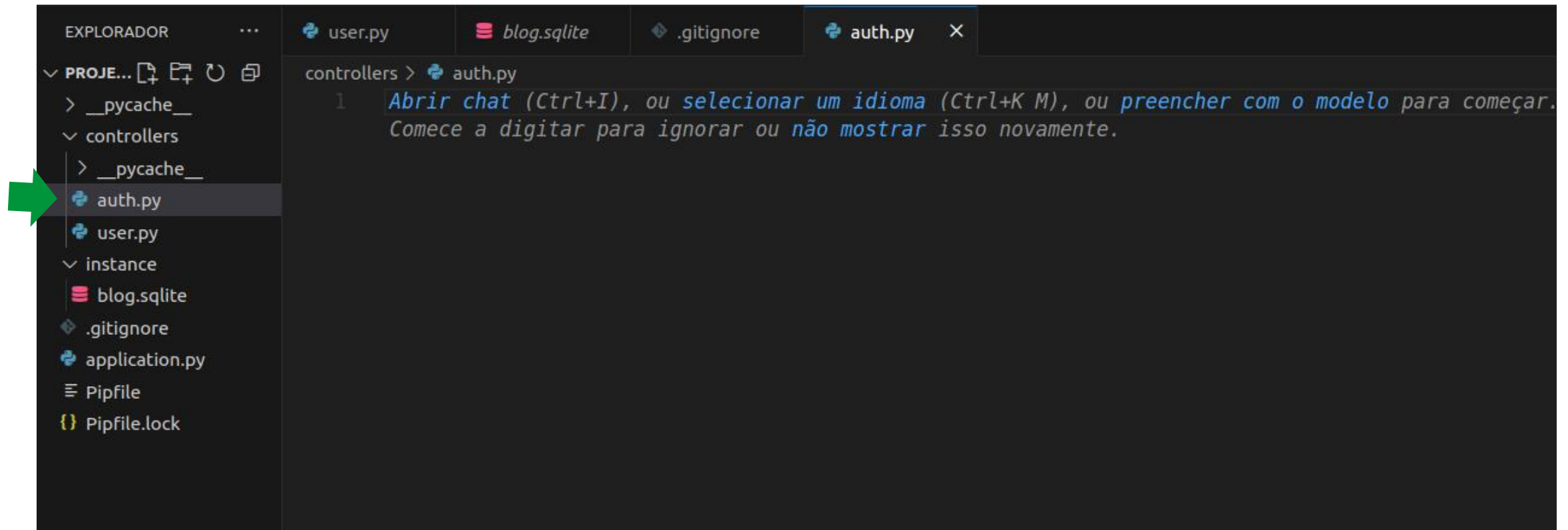
Isso garante que ninguém consiga alterar o conteúdo do token sem que seja detectado,.

```
app.cli.add_command(init_db_command)
```

```
db.init_app(app)  
jwt.init_app(app)
```

```
from controllers import user  
app.register_blueprint(user.app)
```

Criando a rota de login com Blueprint no Flask




Criando a rota de login com Blueprint no Flask


✓ PROJETO_FLASK_BLOG

> __pycache__


✓ controllers

> __pycache__

➡  auth.py

 user.py

✓ instance

 blog.sqlite

controllers >  auth.py > ...

1 ➡ from flask import Blueprint, request

2 from http import HTTPStatus

3

4

5 ➡ app = Blueprint("auth", __name__, url_prefix="/auth")

6

7

8

Criando a rota de login com Blueprint no Flask

```
from flask_jwt_extended import create_access_token
```

```
app = Blueprint("auth", __name__, url_prefix="/auth")
```



```
@app.post("/login")
```

```
def login():
```


```
    data = request.get_json()
```

```
    username = data.get("username")
```


```
    password = data.get("password")
```

```
    if not data or not username or not password:
```

```
        return {"msg": "Username and password required"}, HTTPStatus.BAD_REQUEST
```

```
    if username != "test" or password != "test": 
```

```
        return {"msg": "Bad username or password"}, HTTPStatus.UNAUTHORIZED
```

```
    access_token = create_access_token(identity=username) 
```

```
    return {"access_token": access_token}, HTTPStatus.OK
```

Criando a rota de login com Blueprint no Flask

```
from flask_jwt_extended import create_access_token
```

```
app = Blueprint("auth", __name__, url_prefix="/auth")
```

```
@app.post("/login")
```

```
def login():
```

```
    data = request.get_json()
```

```
    username = data.get("username")
```

```
    password = data.get("password")
```

```
    if not data or not username or not password:
```

```
        return {"msg": "Username and password required"}
```

```
    if username != "test" or password != "test":
```

```
        return {"msg": "Bad username or password"}, HTTPStatus.UNAUTHORIZED
```

```
    access_token = create_access_token(identity=username)
```

```
    return {"access_token": access_token}, HTTPStatus.OK
```

Método do Flask-JWT-Extended que serve para gerar um token JWT de acesso.

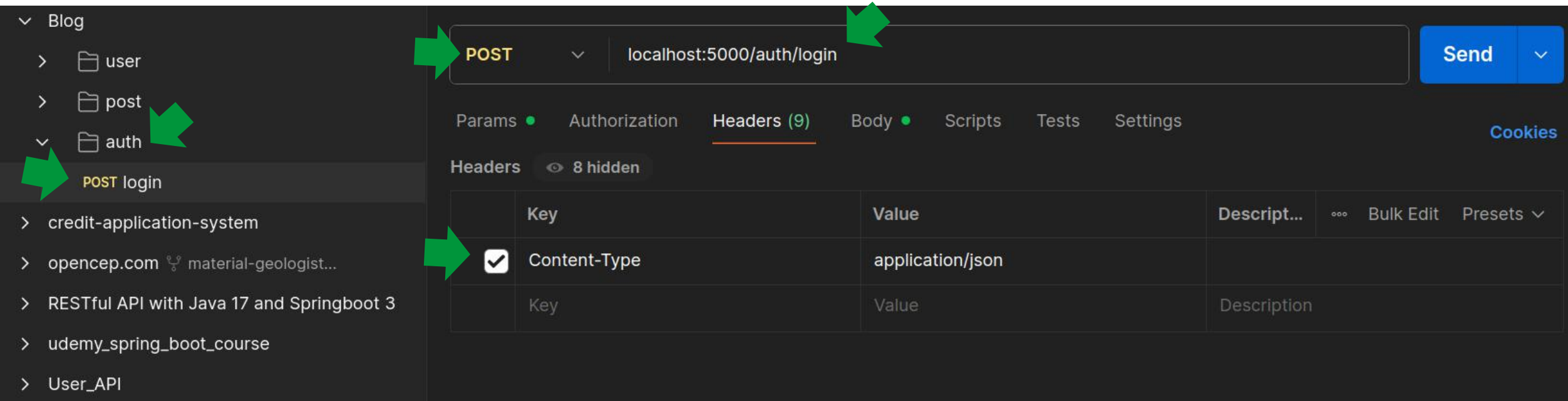
identity pode ser qualquer identificador único do usuário, como id, username ou email.

No Flask-JWT-Extended, a validade padrão do access token é 15 minutos.

Criando a rota de login com Blueprint no Flask

```
from controllers import user  
app.register_blueprint(user.app)  
  
from controllers import auth  
app.register_blueprint(auth.app)
```



Verificando login e tokens no Postman



The screenshot shows the Postman interface with a POST request configured. The left sidebar shows a collection named 'auth' with a sub-collection 'POST login'. The main area shows the request details for 'localhost:5000/auth/login'. The 'Headers' tab is selected, showing a table with headers. The 'Content-Type' header is checked and set to 'application/json'. The 'Send' button is visible in the top right corner.

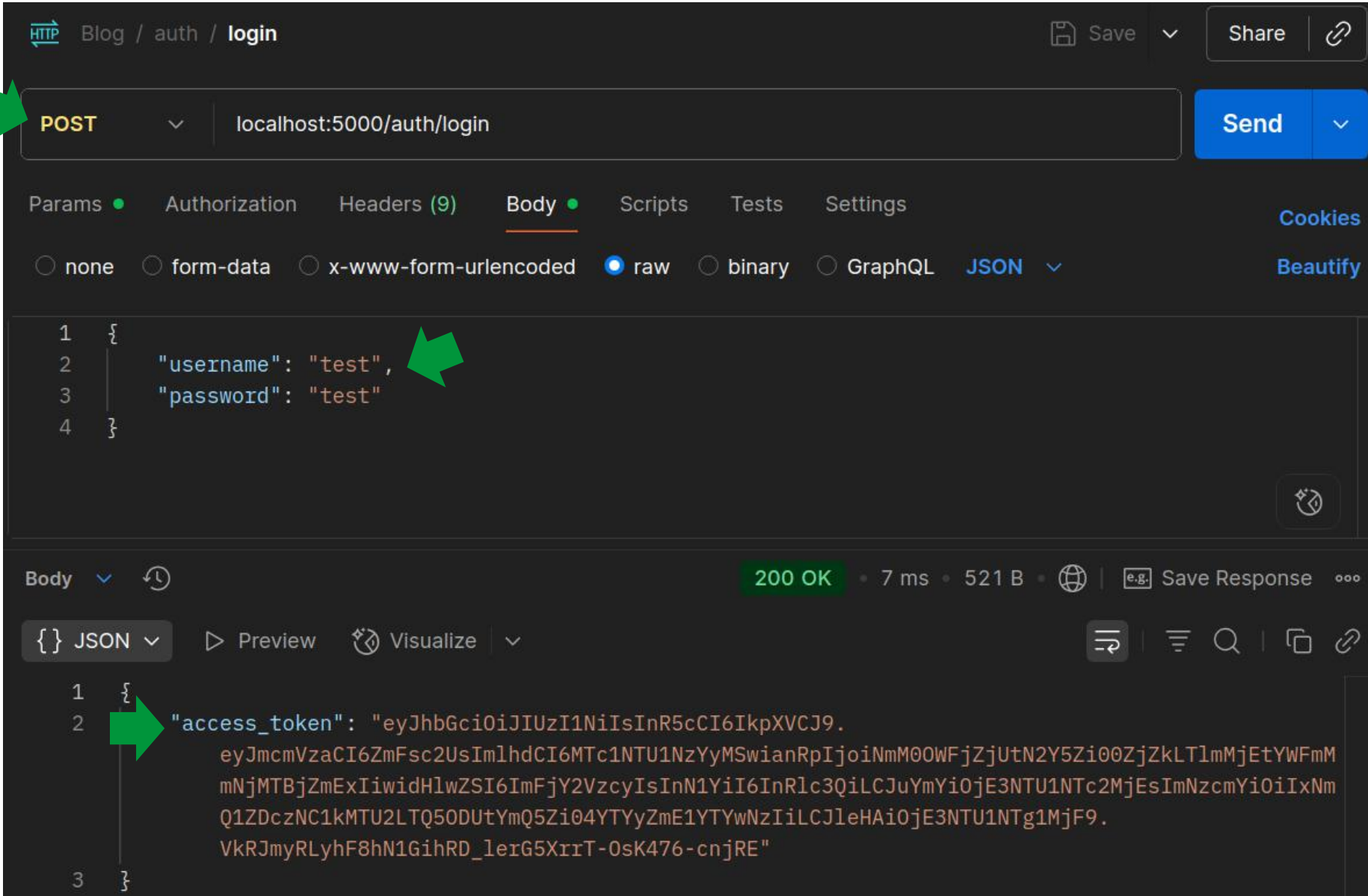
POST `localhost:5000/auth/login` **Send**

Params • Authorization **Headers (9)** Body • Scripts Tests Settings [Cookies](#)

Headers  8 hidden

	Key	Value	Descript...	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Content-Type	application/json				
	Key	Value	Description			

Verificando login e tokens no Postman



The screenshot shows the Postman interface for a POST request to `localhost:5000/auth/login`. The request body is a JSON object with `username: "test"` and `password: "test"`. The response is a 200 OK status with a response time of 7 ms and a size of 521 B. The response body is a JSON object containing an `access_token`.

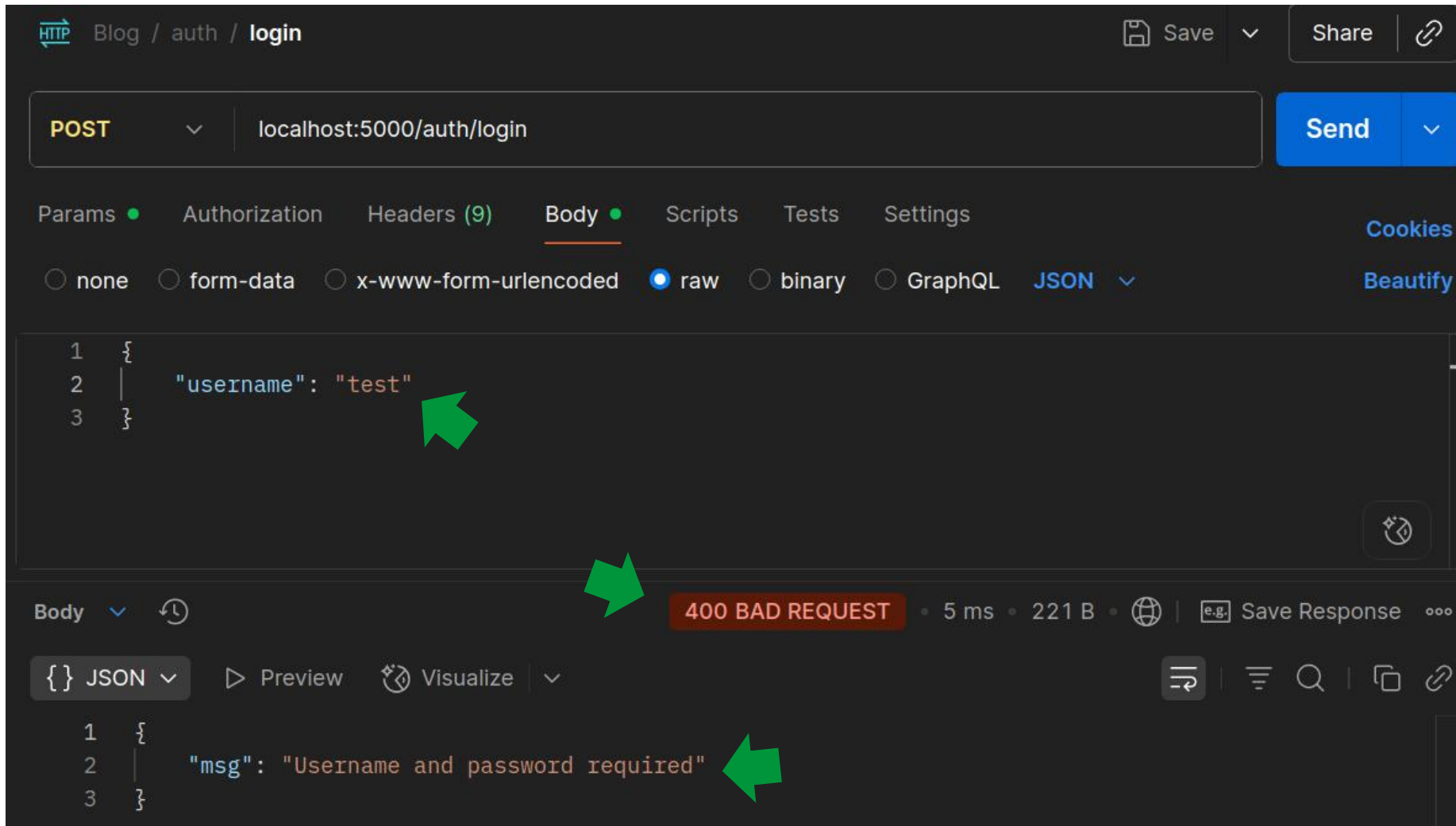
Request:

```
1 {  
2   "username": "test",  
3   "password": "test"  
4 }
```

Response:

```
1 {  
2   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
3     eyJmcmVzaCI6ZmFsc2UsIm1hdCI6MTc1NTU1NzYyMSwianRpIjoibmM0OWFjZjZjUtn2Y5Zi00ZjZkLTlmMjEtYWVmMm  
4     mNjMTBjZmExIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6InRlc3QiLCJuYmYiOiJE3NTU1NTc2MjEsImNzcmYiOiIxNm  
5     Q1ZDczNC1kMTU2LTQ5ODUtYmQ5Zi04YTtyZmE1YTYwNzIiLCJleHAiOiJE3NTU1NTg1MjF9.  
6     VkrJmyRLyhF8hN1GihRD_lerG5XrrT-OsK476-cnJRE"  
7 }
```

Verificando login e tokens no Postman



The screenshot shows the Postman interface for a POST request to `localhost:5000/auth/login`. The request body is a JSON object with a `username` field set to `"test"`. The response is a `400 BAD REQUEST` with a message: `"msg": "Username and password required"`. Green arrows highlight the `username` field in the request and the error message in the response.

Blog / auth / login

POST localhost:5000/auth/login

Send

Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

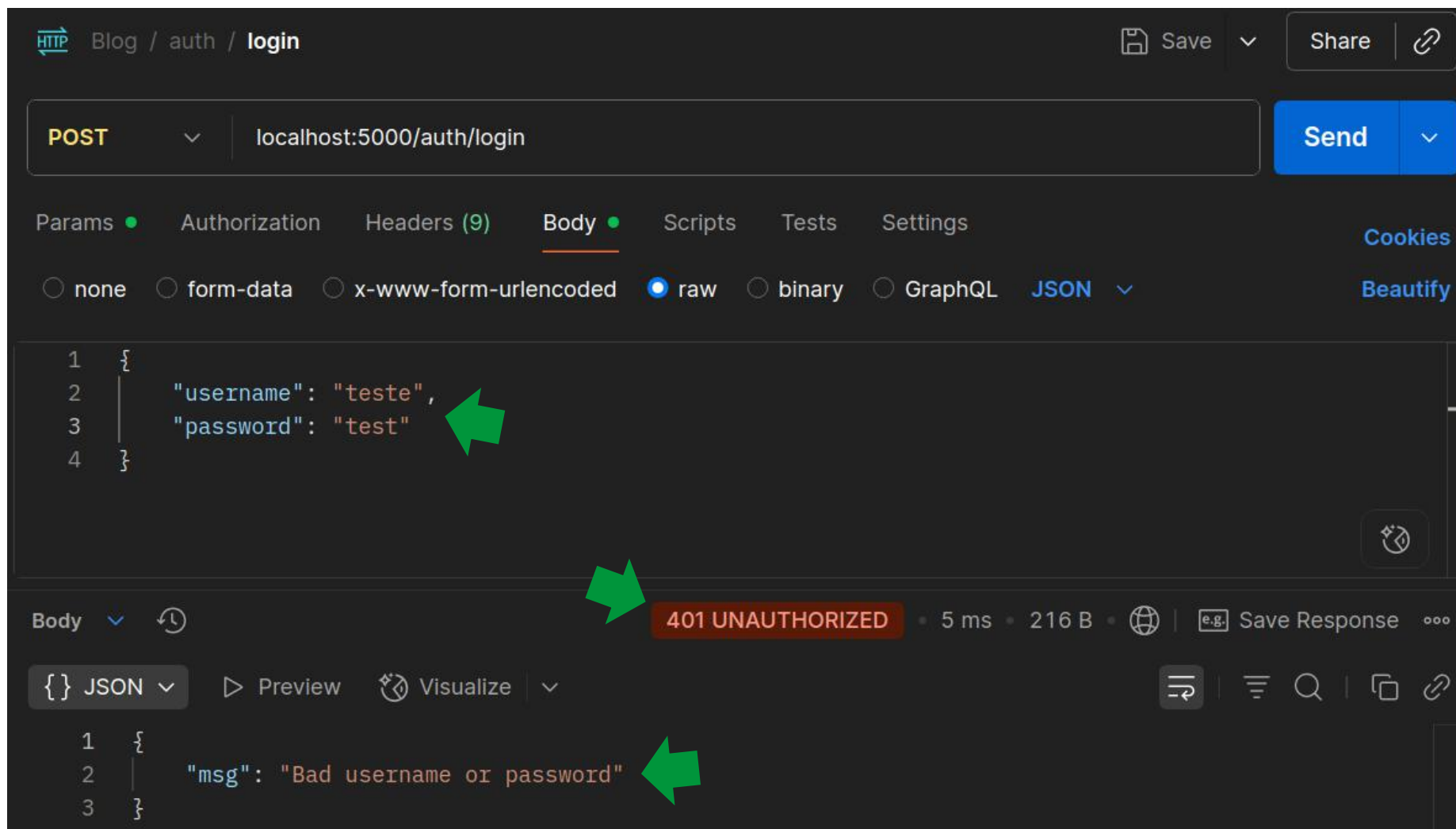
```
1 {
2   "username": "test"
3 }
```

Body 400 BAD REQUEST • 5 ms • 221 B • Save Response

JSON Preview Visualize

```
1 {
2   "msg": "Username and password required"
3 }
```

Verificando login e tokens no Postman



The screenshot shows the Postman interface for a POST request to `localhost:5000/auth/login`. The request body is a JSON object with `username: "teste"` and `password: "test"`. The response is a `401 UNAUTHORIZED` status with a response body of `{ "msg": "Bad username or password" }`. Green arrows highlight the request body and the response status and body.

HTTP Blog / auth / login

POST localhost:5000/auth/login

Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

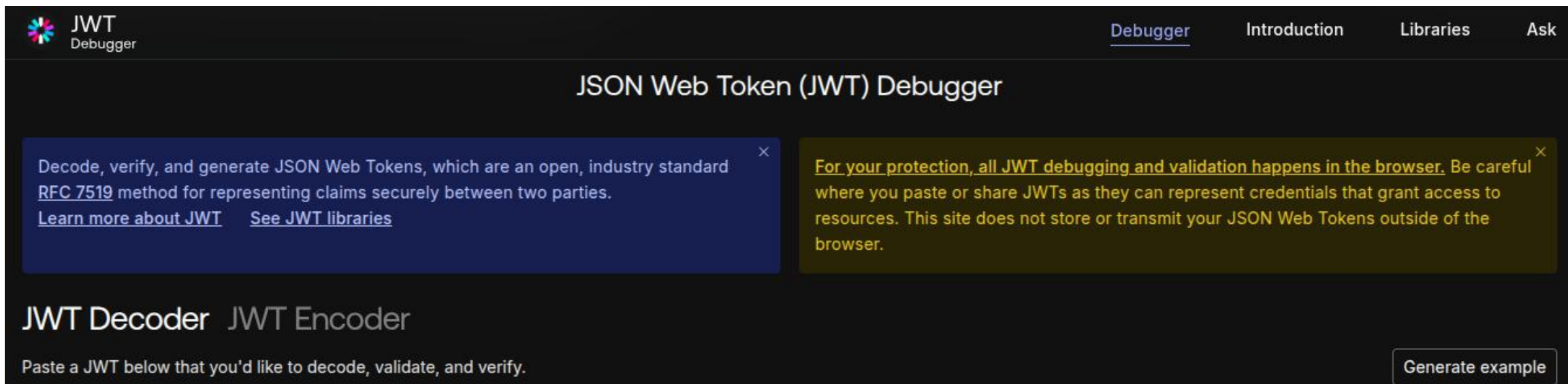
```
1 {
2   "username": "teste",
3   "password": "test"
4 }
```

Body 401 UNAUTHORIZED • 5 ms • 216 B • Save Response

{ JSON Preview Visualize

```
1 {
2   "msg": "Bad username or password"
3 }
```


- O jwt.io é uma ferramenta online para trabalhar com JSON Web Tokens (JWTs) que permite gerar e decodificar tokens para visualizar o header, o payload (com as informações do usuário e claims) e a assinatura, além de verificar se a assinatura é válida.



JWT Debugger

Debugger Introduction Libraries Ask

JSON Web Token (JWT) Debugger

Decode, verify, and generate JSON Web Tokens, which are an open, industry standard [RFC 7519](#) method for representing claims securely between two parties.
[Learn more about JWT](#) [See JWT libraries](#)

For your protection, all JWT debugging and validation happens in the browser. Be careful where you paste or share JWTs as they can represent credentials that grant access to resources. This site does not store or transmit your JSON Web Tokens outside of the browser.

JWT Decoder JWT Encoder

Paste a JWT below that you'd like to decode, validate, and verify.

Generate example

JWT Decoder JWT Encoder

Paste a JWT below that you'd like to decode, validate, and verify.

Generate example

ENABLED

Enable auto-focus

JSON WEB TOKEN (JWT)

COPY

CLEAR

Valid JWT

Invalid Signature

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImhhbmQiOiMTc1NTU1ODAxNCwianRpIjoieWE3YTEyMzItMTZiMi00MTljLW50dQtmZFhmMTZiYTdmYWY4IiwiaWF0IjE6ImFjcyIsInN1YiI6InRlc3QiLCJuYmYiOiJE3NTU1NTgwMTQsImNzcmyoiI3YmU5MmZkMC0yOWVhLTQ1MTEtOWE1Ni0xZWJiMTlkOTlkZGUlLCJleHAiOiJE3NTU1NTg5MTR9.z2WRxmFl0ffewACppyECsa7294iNUd14IR3dNQ5Xfbg

DECODED HEADER

JSONCLAIMS TABLECOPY↗

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

DECODED PAYLOAD

JSONCLAIMS TABLECOPY↗

{
 "fresh": false,
 "iat": 1755558014,
 "jti": "aa7a1232-16b2-419c-a984-01a16ba7faf8",
 "type": "access",
 "sub": "test",
 "nbf": 1755558014,
 "csrf": "7be92fd0-29ea-4511-9a56-1ebb19d98dde",
 "exp": 1755558914
}

JWT SIGNATURE VERIFICATION (OPTIONAL)

Enter the secret used to sign the JWT below:

SECRET

COPY

CLEAR

signature verification failed


a-string-secret-at-least-256-bits-long

Restringindo os endpoints

- Após a criação da autenticação (login + geração de JWT), o próximo passo é a proteção dos endpoints.
- Por que proteger endpoints?
 - **Restringir acesso a recursos sensíveis:** Sem proteção, qualquer pessoa (mesmo sem login) poderia chamar /users, /delete_post, /admin etc.
 - **Garantir identidade do usuário:** O token JWT traz no payload a identidade (identity) do usuário logado.
 - **Controle de autorização:** Não basta saber quem é o usuário (autenticação). Você também precisa decidir o que ele pode fazer (autorização).
 - **Segurança contra ataques:** Sem proteção, endpoints viram alvo fácil para bots ou usuários mal-intencionados. Com JWT, você exige um token válido e assinado, o que dificulta acessos indevidos.

Restringindo os endpoints

```
from flask_jwt_extended import jwt_required
```



```
@app.get("/")
@jwt_required()
def list_users():
    query = db.select(User)
    result = db.session.execute(query)
    users = result.scalars().all()

    return [
        {
            "id": user.id,
            "username": user.username,
            "email": user.email
        }
        for user in users
    ], HTTPStatus.OK
```


Restringindo os endpoints

```
from flask_jwt_extended import jwt_required
```

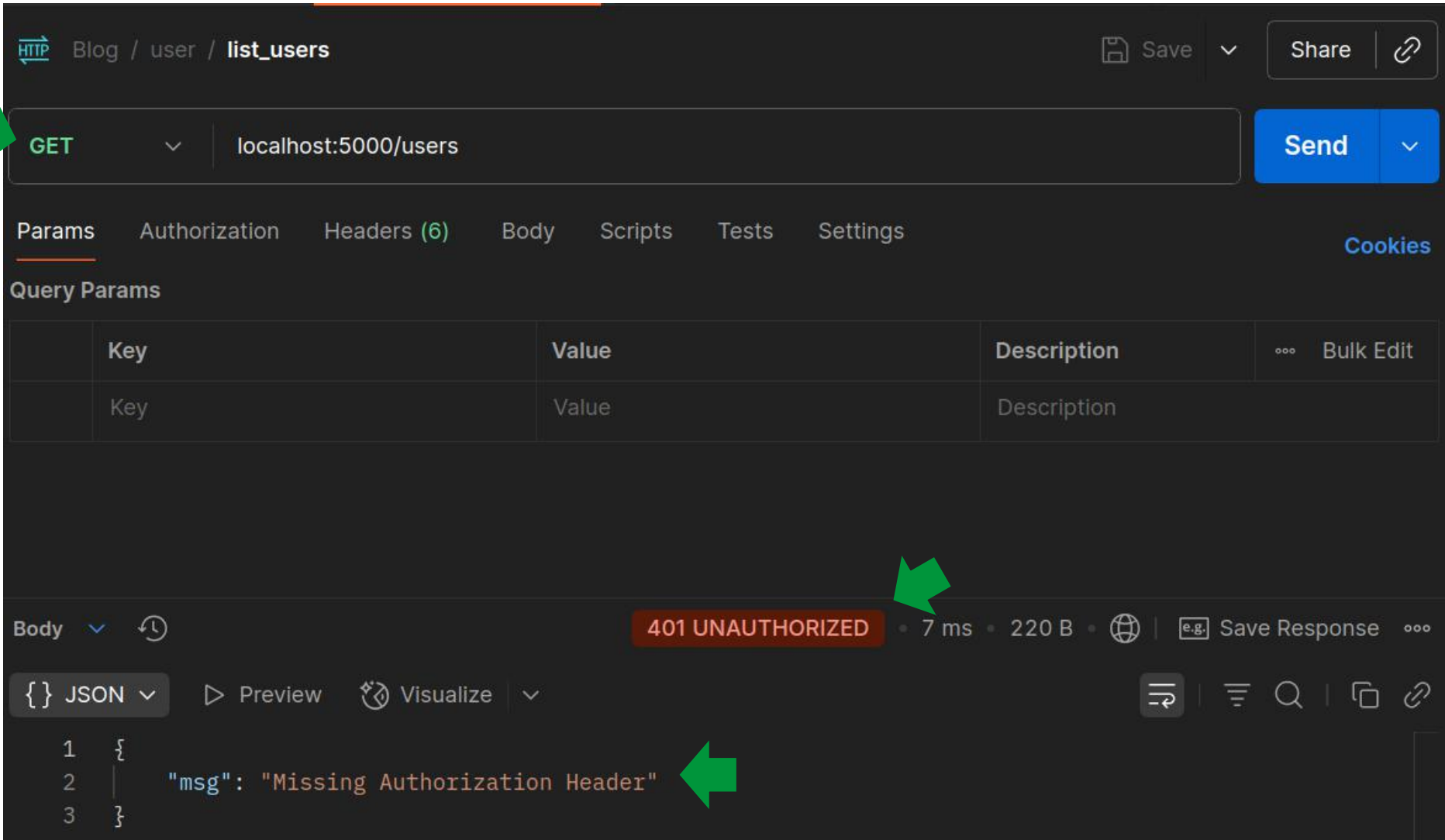
```
@app.get("/")
@jwt_required()
def list_users():
    query = db.select(User)
    result = db.session.execute(query)
    users = result.scalars().all()

    return [
        {
            "id": user.id,
            "username": user.username,
            "email": user.email
        }
        for user in users
    ], HTTPStatus.OK
```

O `@jwt_required()` é um decorator do Flask-JWT-Extended usado para proteger rotas da sua aplicação.

Ele exige que um token JWT válido seja enviado na requisição.

Testando a rota protegida com JWT no Postman



The screenshot shows the Postman interface for a GET request to `localhost:5000/users`. The request is in the `Params` tab. The response is `401 UNAUTHORIZED` with a status bar indicating `7 ms` and `220 B`. The response body is JSON, showing a message: `"msg": "Missing Authorization Header"`.

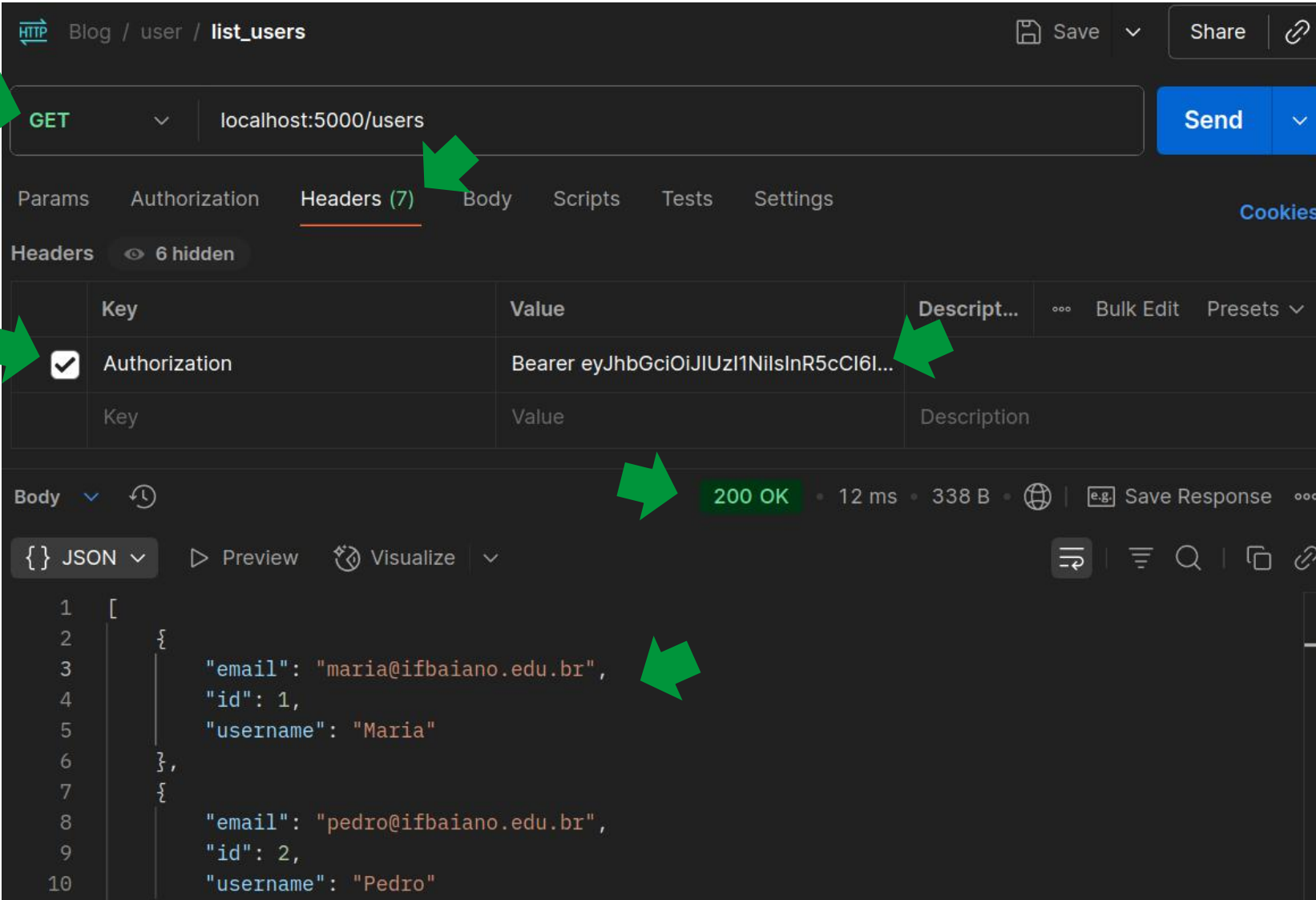
Key elements highlighted with green arrows:

- The `GET` method dropdown.
- The `Send` button.
- The `401 UNAUTHORIZED` status bar.
- The JSON response body.

Key	Value	Description
Key	Value	Description

```
1 {
2   "msg": "Missing Authorization Header"
3 }
```

Testando a rota protegida com JWT no Postman



The screenshot shows the Postman interface for testing a REST API. The request is a GET to `localhost:5000/users`. The **Headers** tab is active, showing an **Authorization** header with a Bearer token. The response is a 200 OK status with a JSON body containing a list of two users.

Request Details:

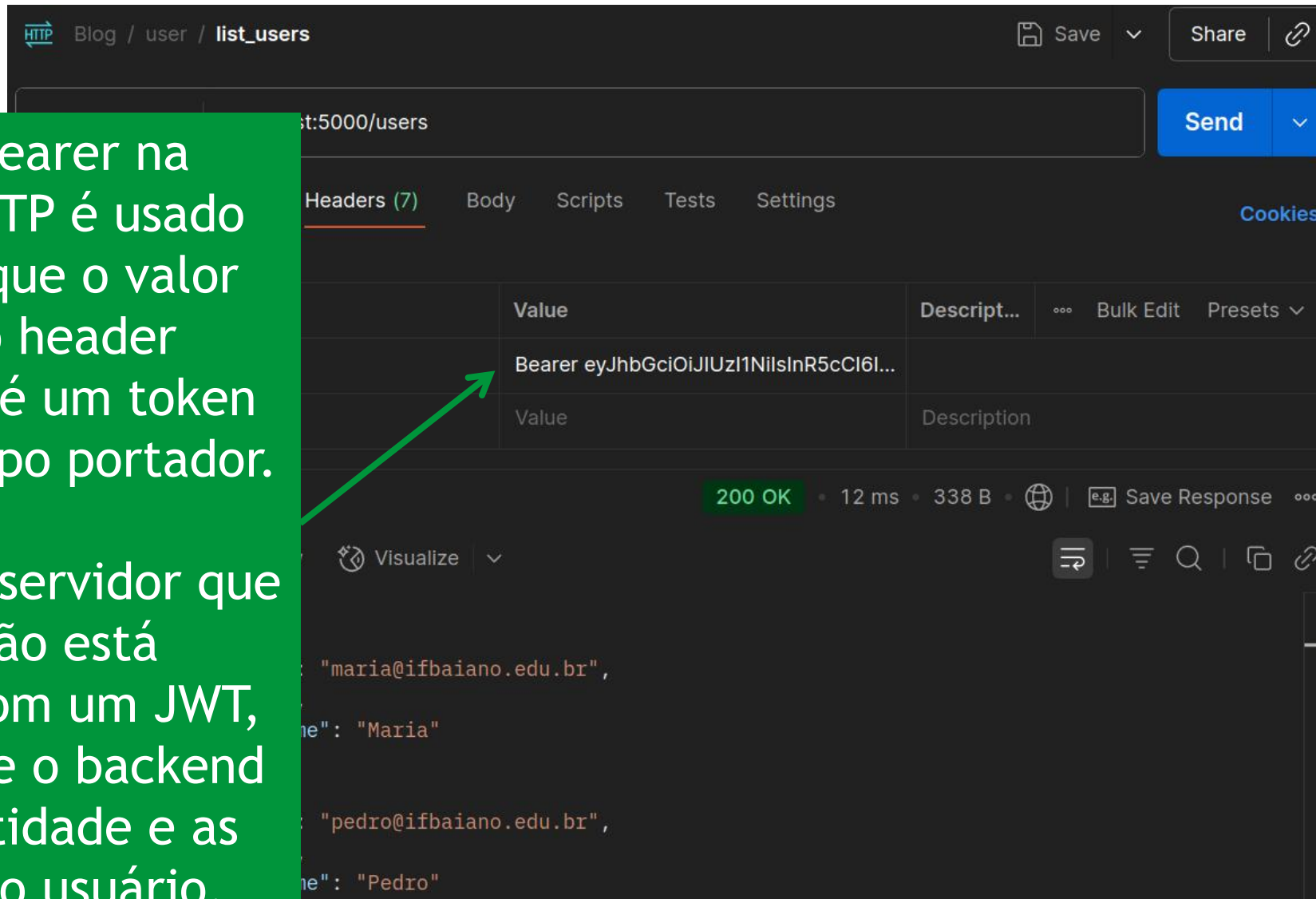
- Method: GET
- URL: `localhost:5000/users`
- Headers (7):
 - ☒ Authorization: Bearer `eyJhbGciOiJIUzI1NiIsInR5cCI6I...`

Response Details:

- Status: 200 OK
- Time: 12 ms
- Size: 338 B
- Body (JSON):

```
[
  {
    "email": "maria@ifbaiano.edu.br",
    "id": 1,
    "username": "Maria"
  },
  {
    "email": "pedro@ifbaiano.edu.br",
    "id": 2,
    "username": "Pedro"
  }
]
```

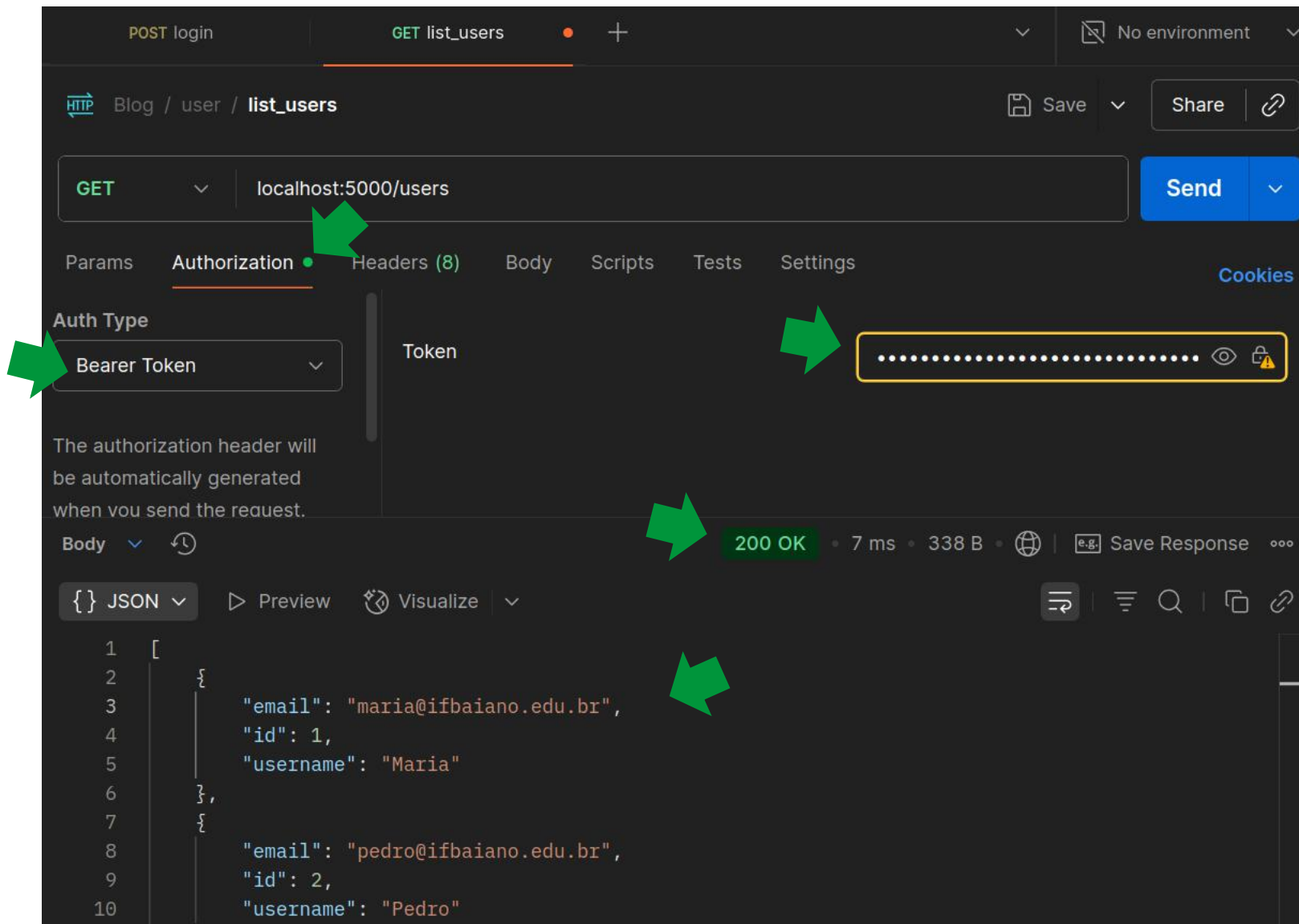

Testando a rota protegida com JWT no Postman



O prefixo Bearer na requisição HTTP é usado para indicar que o valor enviado no header Authorization é um token de acesso do tipo portador.

Ele informa ao servidor que a requisição está autenticada com um JWT, permitindo que o backend valide a identidade e as permissões do usuário.

Testando a rota protegida com JWT no Postman



The screenshot shows the Postman interface for a GET request to `localhost:5000/users`. The **Authorization** tab is selected, showing the **Auth Type** as **Bearer Token** and a **Token** field containing a masked JWT token. The **Body** tab shows the response status as **200 OK** with a response time of 7 ms and a size of 338 B. The response body is displayed in JSON format, showing a list of two users.

Request Details:

- Method: GET
- URL: localhost:5000/users
- Auth Type: Bearer Token
- Token: [Masked JWT Token]

Response Details:

- Status: 200 OK
- Time: 7 ms
- Size: 338 B

Response Body (JSON):

```
[
  {
    "email": "maria@ifbaiano.edu.br",
    "id": 1,
    "username": "Maria"
  },
  {
    "email": "pedro@ifbaiano.edu.br",
    "id": 2,
    "username": "Pedro"
  }
]
```

Armazenando senhas de forma segura usando hash



```
from werkzeug.security import generate_password_hash, check_password_hash
```

```
class User(db.Model):
    id: Mapped[int] = mapped_column(primary_key=True)
    username: Mapped[str] = mapped_column(
        db.String(80), unique=True, nullable=False)
    email: Mapped[str] = mapped_column(db.String(120), nullable=True)
    ➔ password_hash: Mapped[str] = mapped_column(db.String(128), nullable=False)
    ➔ def set_password(self, password: str):
        self.password_hash = generate_password_hash(password)
    ➔ def check_password(self, password: str) -> bool:
        return check_password_hash(self.password_hash, password)

    def __repr__(self) -> str:
        return f"User(id={self.id!r}, email={self.email!r})"
```

Armazenando senhas de forma segura usando hash

- A biblioteca `werkzeug.security` fornece funções simples e seguras para lidar com senhas, como `generate_password_hash`, que transforma a senha em um hash seguro antes de armazená-la no banco de dados, e `check_password_hash`, que compara uma senha fornecida com o hash armazenado para validar autenticações.
- Existem alternativas bastante usadas, como o Bcrypt, o Argon2 e o PBKDF2.

Armazenando senhas de forma segura usando hash

```
@app.post("/login")
def login():
    data = request.get_json()
    username = data.get("username")
    password = data.get("password")

    if not data or not username or not password:
        return {"msg": "Username and password required"}, HTTPStatus.BAD_REQUEST

    ➡ user = User.query.filter_by(username=username).first()

    ➡ if user is None or not user.check_password(password):
        return {"msg": "Bad username or password"}, HTTPStatus.UNAUTHORIZED

    access_token = create_access_token(identity=user.username)
    return {"access_token": access_token}, HTTPStatus.OK
```

Armazenando senhas de forma segura usando hash


```
@click.command("init-db")
def init_db_command():
    with current_app.app_context():
        db.create_all()




        ➡ # cria um usuário "admin" se não existir
        if not User.query.filter_by(username="admin").first():
            user = User(username="admin", email="admin@example.com")
            user.set_password("admin123")
            db.session.add(user)
            db.session.commit()
            click.echo("Usuário admin criado!")
        else:
            click.echo("Usuário admin já existe.")

    click.echo("Inicializando a base de dados...")
```

Removendo e recriando o banco de dados

- `rm instance/blog.sqlite`
- `flask --app application init-db`

instance >  blog.sqlite

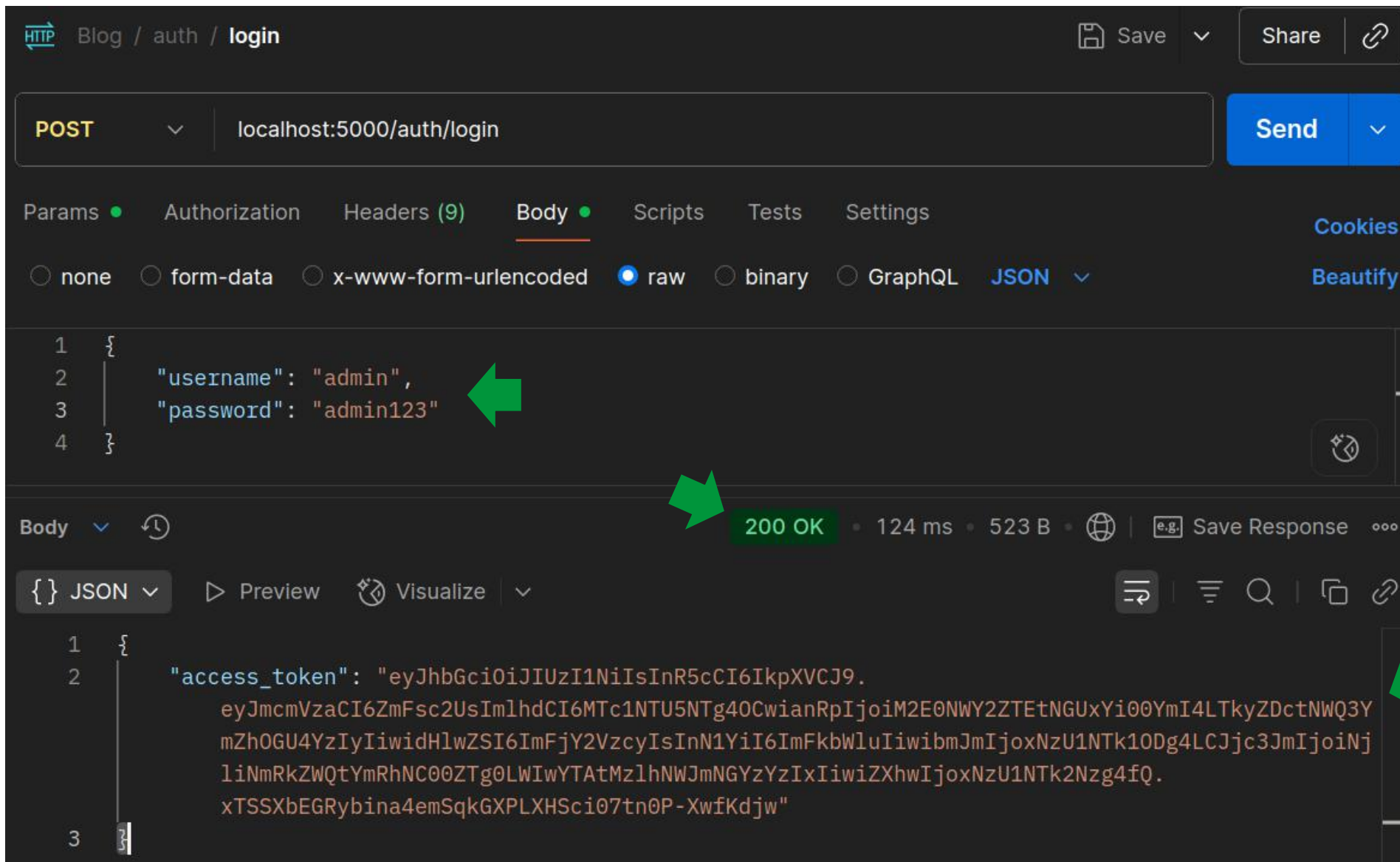
⌂  Filtrar 2 t Linhas: 1 Filtrar 1 linhas... [Atualizar para PRO](#)  

▼ TABELAS

- > post
- > user

	id	username	email	password_hash
1	1	admin	admin@example....	scrypt:32768:8:1\$A5Y1cV5hALfc6hLZ\$
+	2			

Testando a rota protegida com JWT no Postman



The screenshot shows a Postman interface for a POST request to `localhost:5000/auth/login`. The request body is a JSON object with `username: "admin"` and `password: "admin123"`. The response is a 200 OK status with a JSON body containing an `access_token`. Three green arrows highlight the request body, the status bar, and the access token in the response.

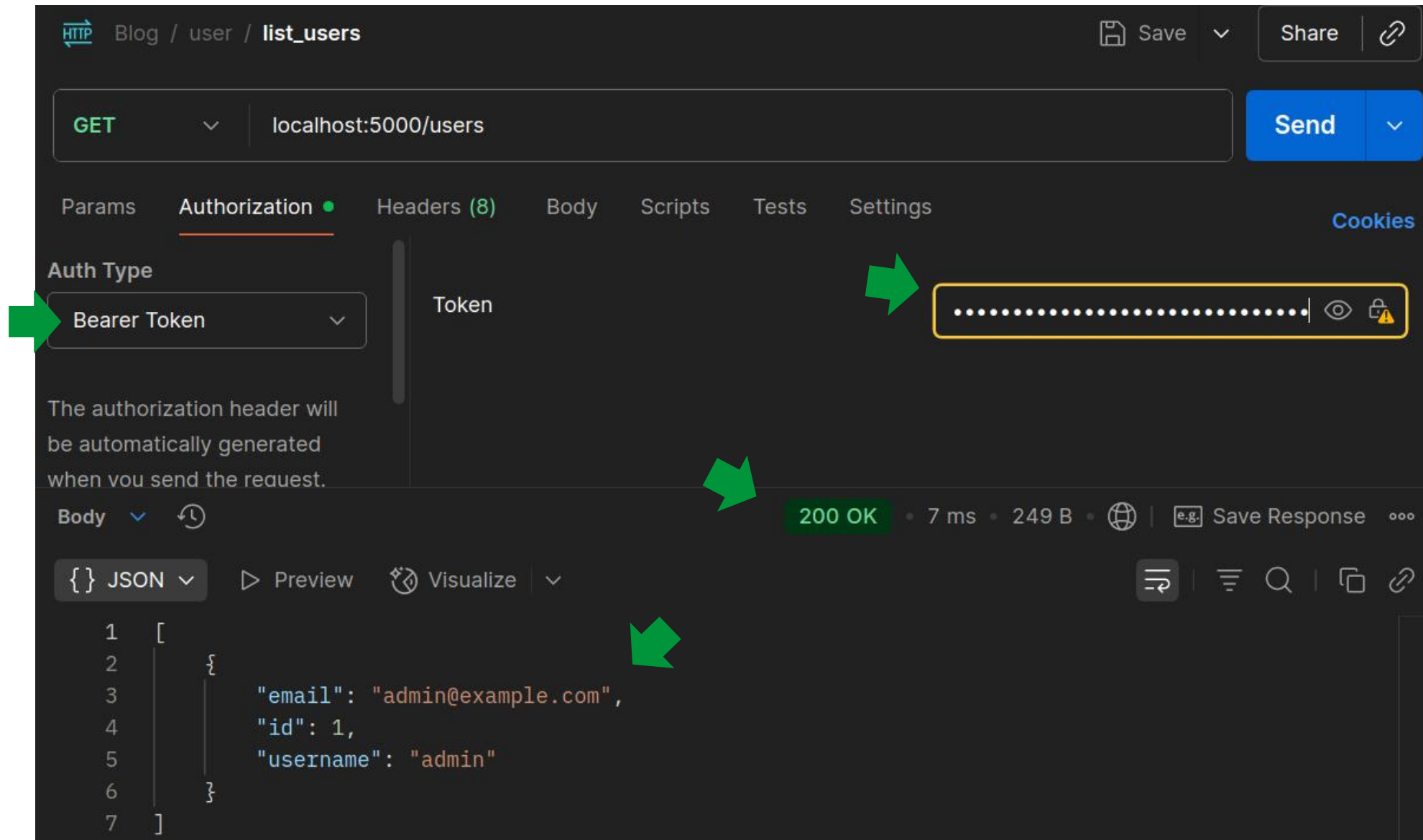
Request:

```
1 {
2   "username": "admin",
3   "password": "admin123"
4 }
```

Response:

```
1 {
2   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsIm1hdCI6MTc1NTU5NTg0CwianRpijoiM2E0NWY2ZTEtNGUxYi00YmI4LTkyZDctNWQ3YmZhOGU4YzIyIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6ImFkbWluIiwibmJmIjoxNzU1NTk1ODg4LCJjc3MjoiNjliNmRkZWQtYmRhNC00ZTg0LWIwYTAtMzlhNWJmNGYzYzIxIiwiaXNjaXhwIjoxNzU1NTk1NTk2Nzg4fQ.xTSSXbEGRybina4emSqkGXPLXHSci07tn0P-XwfKdjw"
3 }
```

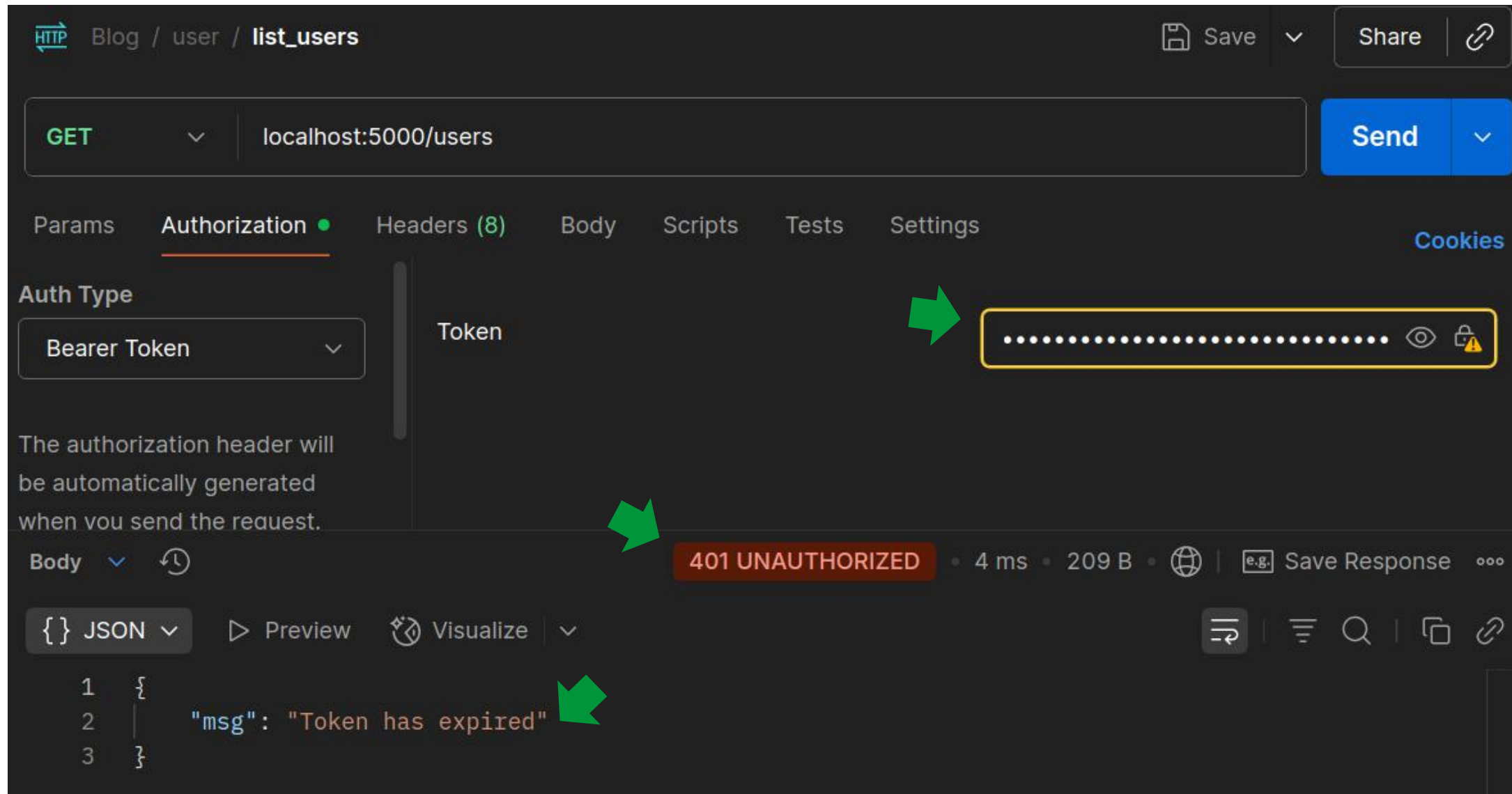

Testando a rota protegida com JWT no Postman



The screenshot shows the Postman interface for testing a REST API. The request is a GET to `localhost:5000/users`. The **Authorization** tab is selected, showing **Bearer Token** as the auth type. A green arrow points to the **Bearer Token** dropdown. Another green arrow points to the token input field, which contains a masked token (dots) and a warning icon. A third green arrow points to the **Body** tab, which shows a JSON response. A fourth green arrow points to the **200 OK** status bar. The response body is a JSON array with one object:

```
1 [
2   {
3     "email": "admin@example.com",
4     "id": 1,
5     "username": "admin"
6   }
7 ]
```

Testando a rota protegida com JWT no Postman



The screenshot shows the Postman interface for testing the `list_users` endpoint. The request is a `GET` to `localhost:5000/users`. The `Authorization` tab is selected, showing a `Bearer Token` type. A green arrow points to the token input field, which contains a masked token (dots) and a warning icon. Another green arrow points to the `401 UNAUTHORIZED` status in the response bar. A third green arrow points to the JSON response body, which contains the message `"msg": "Token has expired"`.

Blog / user / **list_users** Save Share

GET `localhost:5000/users` Send

Params **Authorization** Headers (8) Body Scripts Tests Settings Cookies

Auth Type
Bearer Token

Token

The authorization header will be automatically generated when you send the request.

Body JSON Preview Visualize

401 UNAUTHORIZED • 4 ms • 209 B • Save Response

```
1 {
2   "msg": "Token has expired"
3 }
```

Testando a rota protegida com JWT no Postman

O token gerado possui, por padrão, validade de 15 minutos, mas há a possibilidade de implementar o uso de refresh tokens, permitindo que o usuário obtenha um novo token de acesso sem precisar refazer o login.



The screenshot shows the Postman interface for a GET request to `localhost:5000/users`. The **Authorization** tab is selected, and the **Auth Type** is set to **Bearer Token**. A green arrow points from the text box above to the **Token** input field, which contains a masked token (represented by dots). Another green arrow points from the same text box to the response body, which shows a **401 UNAUTHORIZED** status and a JSON message: `{ "msg": "Token has expired" }`. The response details at the top indicate a status of 401, a time of 4 ms, and a size of 209 B.

Blog / user / list_users

GET localhost:5000/users

Params Authorization Headers (8) Body Scripts Tests Settings Cookies

Auth Type

Bearer Token

Token

The authorization header will be automatically generated when you send the request.

Body

401 UNAUTHORIZED • 4 ms • 209 B • Save Response

JSON Preview Visualize

```
1 {
2   "msg": "Token has expired"
3 }
```

Exercícios

1. Bloqueie a rota de criação de usuários.
2. Garanta que apenas usuários autenticados possam acessar esta rota.
3. Utilize o Postman para autenticar como administrador e cadastrar um novo usuário.
4. Verifique se o acesso aos endpoints protegidos funciona corretamente com o login recém-criado.

Dúvidas



PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática
Lucas Sampaio Leite

