

# PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática  
Lucas Sampaio Leite



## Recapitulando...

- 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.

## Recapitulando...


- 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.



## 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.

## Exercícios (correção)



```
@app.post("/")
@jwt_required()
def create_user():
    data = request.get_json()

    if not data or "username" not in data:
        return {"error": "username é obrigatório"}, HTTPStatus.BAD_REQUEST

    email = data.get("email")

    user = User(username=data["username"], email=email)

    db.session.add(user)
    db.session.commit()

    return {
        "id": user.id,
        "username": user.username,
        "email": user.email
    }, HTTPStatus.CREATED
```

## Exercícios (correção)

```
@app.post("/")
@jwt_required() ←
def create_user():
    data = request.get_json()

    if not data or "username" not
        | return {"error": "username

    email = data.get("email")

    user = User(username=data["username"], email=email)

    db.session.add(user)
    db.session.commit()

    return {
        | "id": user.id,
        | "username": user.username,
        | "email": user.email
    }, HTTPStatus.CREATED
```

O decorator `@jwt_required()` é usado para proteger rotas que exigem autenticação via JWT (JSON Web Token).

Ao marcar uma função de rota com `@jwt_required()`, significa que o acesso a essa rota só será permitido se o cliente enviar um JWT válido no cabeçalho da requisição.

## Exercícios (correção)

- Recapitulando a criação do nosso usuário administrador:

```
@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...")
```



## Exercícios (correção)

- Recapitulando o nosso modelo de User:

```
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})"
```



## Exercícios (correção)

- Modificando o método `create_user()`:

```
@app.post("/")
@jwt_required()
def create_user():
    data = request.get_json()

    if not data or "username" not in data or "password" not in data:
        return {"error": "username e password é obrigatório"}, HTTPStatus.BAD_REQUEST

    username = data["username"]
    email = data.get("email")
    password = data["password"]

    if User.query.filter_by(username=username).first():
        return {"error": "username já existe"}, HTTPStatus.CONFLICT

    user = User(username=username, email=email)
    user.set_password(password)

    db.session.add(user)
    db.session.commit()

    return {
        "id": user.id,
        "username": user.username,
        "email": user.email
    }, HTTPStatus.CREATED
```

- [illegible]

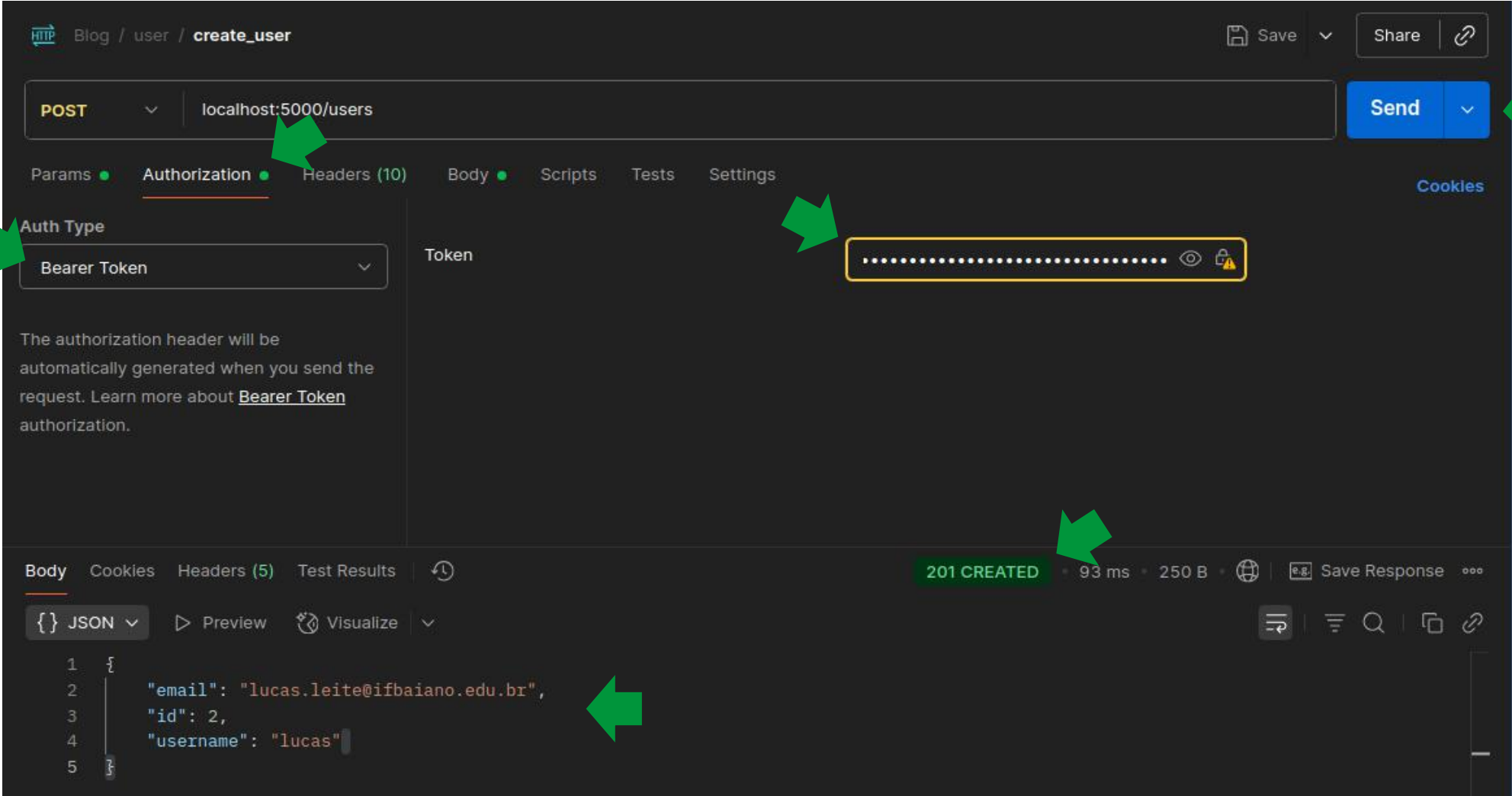
## Exercícios (correção)

- Criando um novo usuário a partir do usuário admin:



# Exercícios (correção)

- Criando um novo usuário a partir do usuário admin:



Blog / user / create\_user

POST localhost:5000/users

Send

Params Authorization Headers (10) Body Scripts Tests Settings

Auth Type

Bearer Token

Token

The authorization header will be automatically generated when you send the request. Learn more about [Bearer Token](#) authorization.

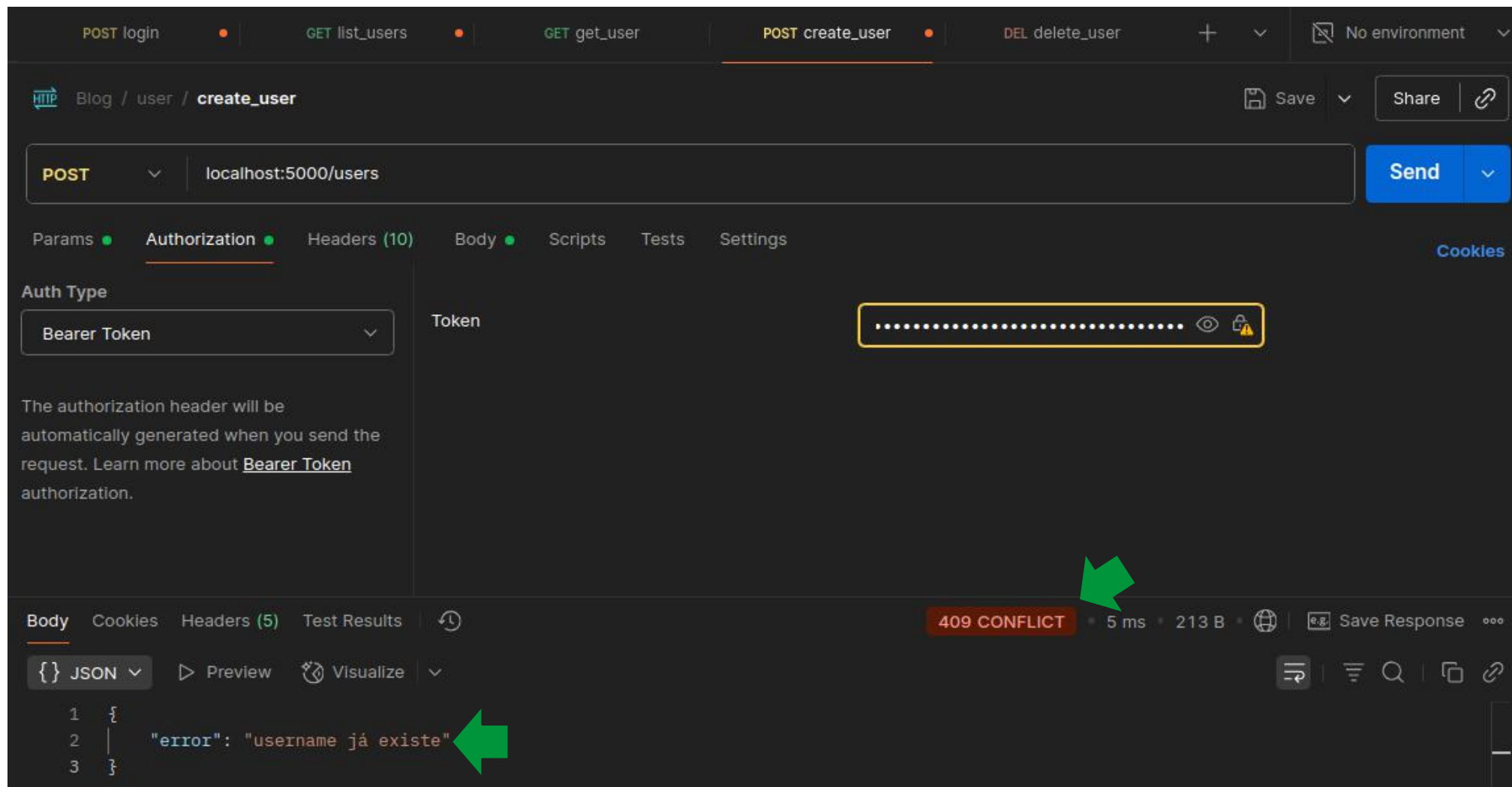
Body Cookies Headers (5) Test Results

201 CREATED • 93 ms • 250 B • Save Response

```
{
  "email": "lucas.leite@ifbaiano.edu.br",
  "id": 2,
  "username": "lucas"
}
```

## Exercícios (correção)

- Criando um novo usuário a partir do usuário admin (em caso de username existente):



The screenshot shows a REST client interface with the following details:

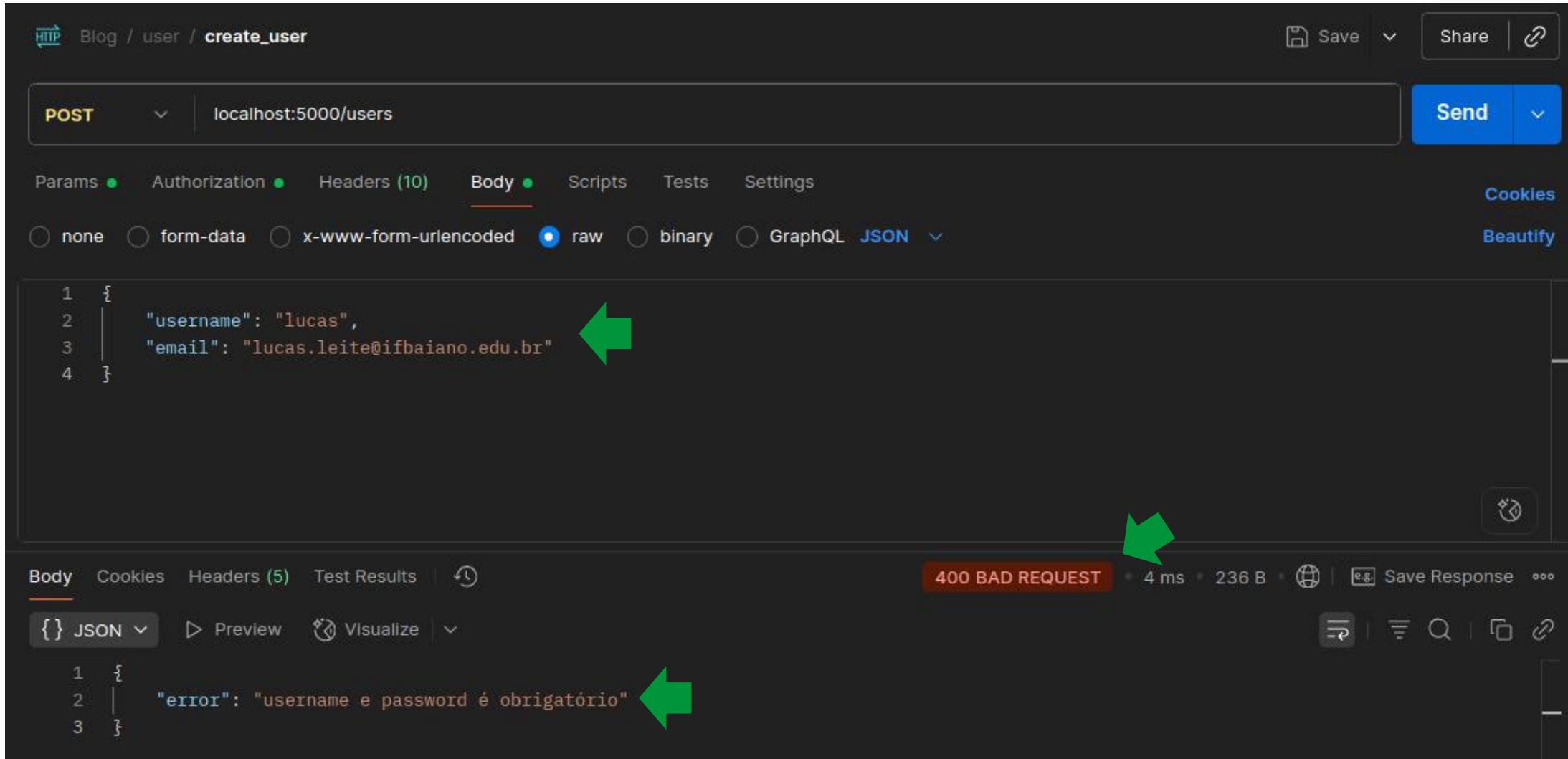
- Request:** POST to `localhost:5000/users`.
- Authorization:** Bearer Token. The token field is highlighted with a yellow box.
- Response:** 409 CONFLICT (5 ms, 213 B).
- Response Body (JSON):**

```
1 {  
2   "error": "username já existe"  
3 }
```

Two green arrows highlight the error response: one points to the **409 CONFLICT** status and the other points to the **"error": "username já existe"** message in the JSON body.

## Exercícios (correção)

- Criando um novo usuário a partir do usuário admin (sem password):



The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** localhost:5000/users
- Body Type:** raw (selected), JSON (dropdown)
- Request Body:**

```
1 {  
2   "username": "lucas",  
3   "email": "lucas.leite@ifbaiano.edu.br"  
4 }
```
- Response:** 400 BAD REQUEST (4 ms, 236 B)
- Response Body:**

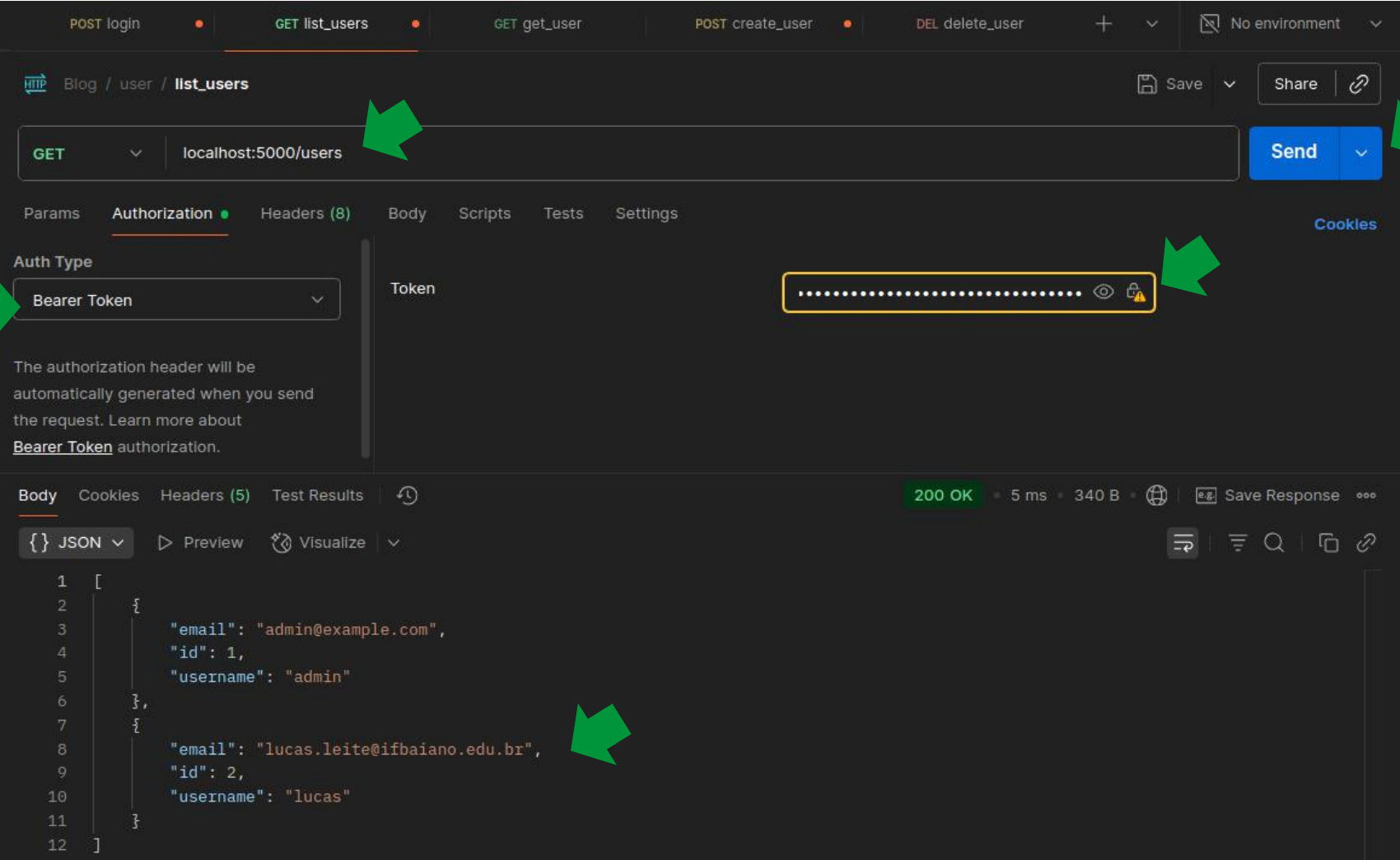
```
1 {  
2   "error": "username e password é obrigatório"  
3 }
```

Green arrows point to the request body, the response status, and the error message in the response body.



# Exercícios (correção)

- Listando os usuários criados:



The screenshot shows a REST client interface with the following elements:

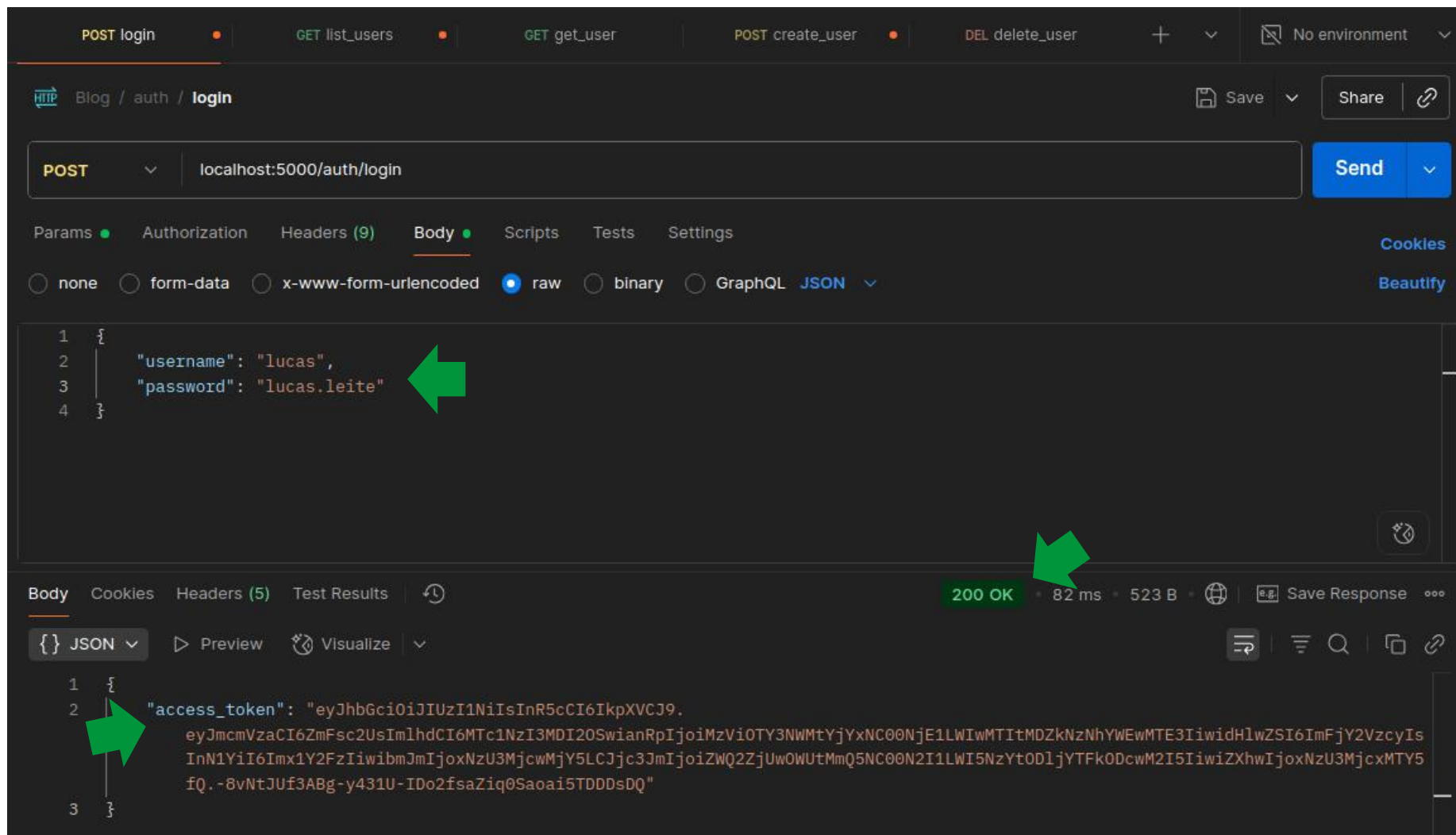
- Request Bar:** Method `GET`, URL `localhost:5000/users`, and a `Send` button.
- Authorization Tab:** Auth Type is `Bearer Token`. The Token field contains a masked token (dots) with an eye icon to toggle visibility.
- Response Section:** Status `200 OK`, Time `5 ms`, Size `340 B`. The response body is a JSON array of two users.

The JSON response is as follows:

```
1 [
2   {
3     "email": "admin@example.com",
4     "id": 1,
5     "username": "admin"
6   },
7   {
8     "email": "lucas.leite@ifbaiano.edu.br",
9     "id": 2,
10    "username": "lucas"
11  }
12 ]
```

# Exercícios (correção)

- Autenticando com o novo usuário:



The screenshot shows a REST client interface with the following details:

- Request:**
  - Method: POST
  - URL: localhost:5000/auth/login
  - Body (JSON):

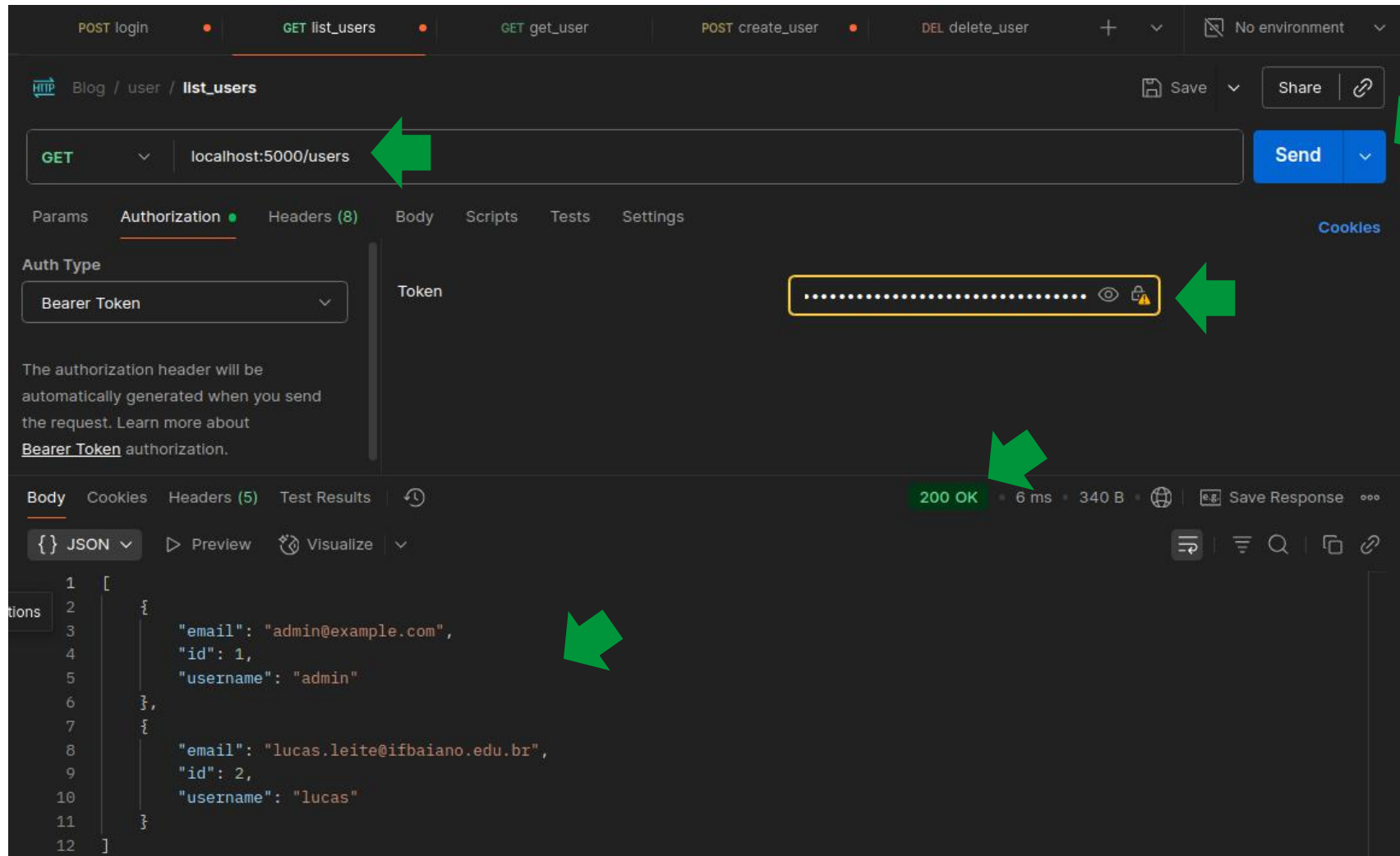
```
{ 1: { 2: "username": "lucas", 3: "password": "lucas.leite" 4: }
```
- Response:**
  - Status: 200 OK
  - Body (JSON):

```
{ 1: { 2: "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsIm1hdCI6MTc1NzI3MDI2OSwianRpIjoimzViOTY3NWMTYjYxNC00NjE1LWIwMTItMDZkNzNhYWEmMTE3IiwidHlwZSI6ImFjY2VzcyIsInN1YiI6Imx1Y2FzIiwibmJmIjoxNzU3MjcwMjY5LCJpc3MjIjoiaWZlZjUwOWUtMmQ5NC00N2I1LWI5NzYtODIjYTFkODcwM2I5IiwiaXhwIjoxNzU3MjcwMjY5fQ.-8vNtJUf3ABg-y431U-IDo2fsaZiq0Saoai5TDDdsDQ" 3: }
```

Green arrows highlight the request body, the response status, and the access token in the response body.

# Exercícios (correção)

- Acessando endpoints protegidos com o novo usuário:



The screenshot displays the Postman API client interface. At the top, a tab for the `GET list_users` endpoint is selected, with the URL `localhost:5000/users` entered in the address bar. A green arrow points to the `GET` method dropdown and another to the `Send` button. Below the address bar, the `Authorization` tab is active, showing `Bearer Token` as the auth type. A text box for the token contains a masked value (dots) and a lock icon, with a green arrow pointing to it. The `Body` tab is selected at the bottom, showing a JSON response with a status of `200 OK`, a response time of `6 ms`, and a size of `340 B`. A green arrow points to the `200 OK` status. The JSON response is as follows:

```
1 [
2   {
3     "email": "admin@example.com",
4     "id": 1,
5     "username": "admin"
6   },
7   {
8     "email": "lucas.leite@ifbaiano.edu.br",
9     "id": 2,
10    "username": "lucas"
11  }
12 ]
```

A green arrow points to the first user object in the JSON array.

## Motivação...

- Criação de usuários atual:
  - Todo usuário novo recebe acesso completo ao sistema, incluindo endpoints restritos.
  - Não há diferenciação entre administrador e usuário comum.
- Riscos e consequências:
  - Acesso indevido a informações sensíveis.
  - Possibilidade de alterações críticas sem controle.
  - Comprometimento da segurança e integridade do sistema.
- Autorização baseada em papéis (roles) é essencial para sistemas multiusuário.

## Relembrando...

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



## Relembrando...

- 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.






## Autorização baseada em roles

- Autorização baseada em roles (ou Role-Based Access Control - RBAC) é um modelo de controle de acesso em que as permissões de um sistema não são atribuídas diretamente a cada usuário, mas sim a papéis (roles) que representam funções ou cargos dentro da organização.
- Usuário → Role → Permissões:
  - Cada usuário recebe um ou mais roles (ex.: "Administrador", "Professor", "Aluno").
  - Cada role tem associadas permissões específicas (ex.: "criar usuário", "editar nota", "visualizar conteúdo").
  - O usuário herda as permissões do seu papel.


# Autorização baseada em roles

1. Definir o modelo de roles - Criar a tabela/entidade responsável por armazenar os diferentes papéis de usuários (roles) no sistema.
2. Estabelecer o relacionamento User ↔ Role - Utilizar o SQLAlchemy ORM para vincular cada usuário a uma role, permitindo o gerenciamento de permissões.
3. Cadastrar roles no sistema - Popular a base de dados com os papéis necessários (ex.: admin, editor, viewer).
4. Atribuir roles aos usuários - Associar cada usuário a um papel específico, garantindo que suas permissões estejam alinhadas com sua função.
5. Restringir o acesso a endpoints - Proteger as rotas da API com base nas roles definidas, limitando funcionalidades conforme o papel do usuário.
6. Validar a identidade via JWT - Extrair a identidade do token JWT, recuperar o usuário associado e verificar sua role para autorizar ou negar o acesso ao endpoint.

# Definindo o modelo de roles



```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship
```



```
class Role(db.Model):  
    id: Mapped[int] = mapped_column(primary_key=True)  
    name: Mapped[str] = mapped_column(db.String(100), nullable=False)  
    users: Mapped[list["User"]] = relationship("User", back_populates="role")  
  
    def __repr__(self) -> str:  
        return f"User(id={self.id!r}, name={self.name!r})"
```

## Estabelecendo o relacionamento User ↔ Role

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship
```

```
class Role(db.Model):  
    id: Mapped[int] = mapped_column(primary_key=True)  
    name: Mapped[str] = mapped_column(db.String(100), nullable=False)  
    users: Mapped[list["User"]] = relationship("User", back_populates="role")  
  
    def __repr__(self) -> str:  
        return f"User(id={self.id!r}, name={self.name!r})"
```

Define o relacionamento entre a classe Role e a classe User no SQLAlchemy ORM. Isso significa que cada papel (role) pode estar associado a vários usuários, criando uma relação de um-para-muitos.

O tipo `Mapped[list["User"]]` indica que o atributo `users` conterá uma lista de objetos da classe User que possuem essa role.

## Estabelecendo o relacionamento User ↔ Role

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship
```

```
class Role(db.Model):  
    id: Mapped[int] = mapped_column(primary_key=True)  
    name: Mapped[str] = mapped_column(db.String(100), nullable=False)  
    users: Mapped[list["User"]] = relationship("User", back_populates="role")  
  
    def __repr__(self) -> str:  
        return f"User(id={self.id!r}, name={self.name!r})"
```

O parâmetro `back_populates="role"` torna o relacionamento bidirecional, ou seja, além de acessar todos os usuários de uma determinada role (`admin.users`), também é possível, a partir de um usuário, acessar a role à qual ele pertence (`user.role`). Esse mapeamento facilita a navegação e manipulação dos dados entre as tabelas de usuários e de papéis no banco.



## Estabelecendo o relacionamento User ↔ Role



```
import sqlalchemy
```

```
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)

    role_id: Mapped[int] = mapped_column(sqlalchemy.ForeignKey("role.id"))
    role: Mapped["Role"] = relationship("Role", back_populates="users")

    def set_password(self, password: str):
        self.password_hash = generate_password_hash(password)

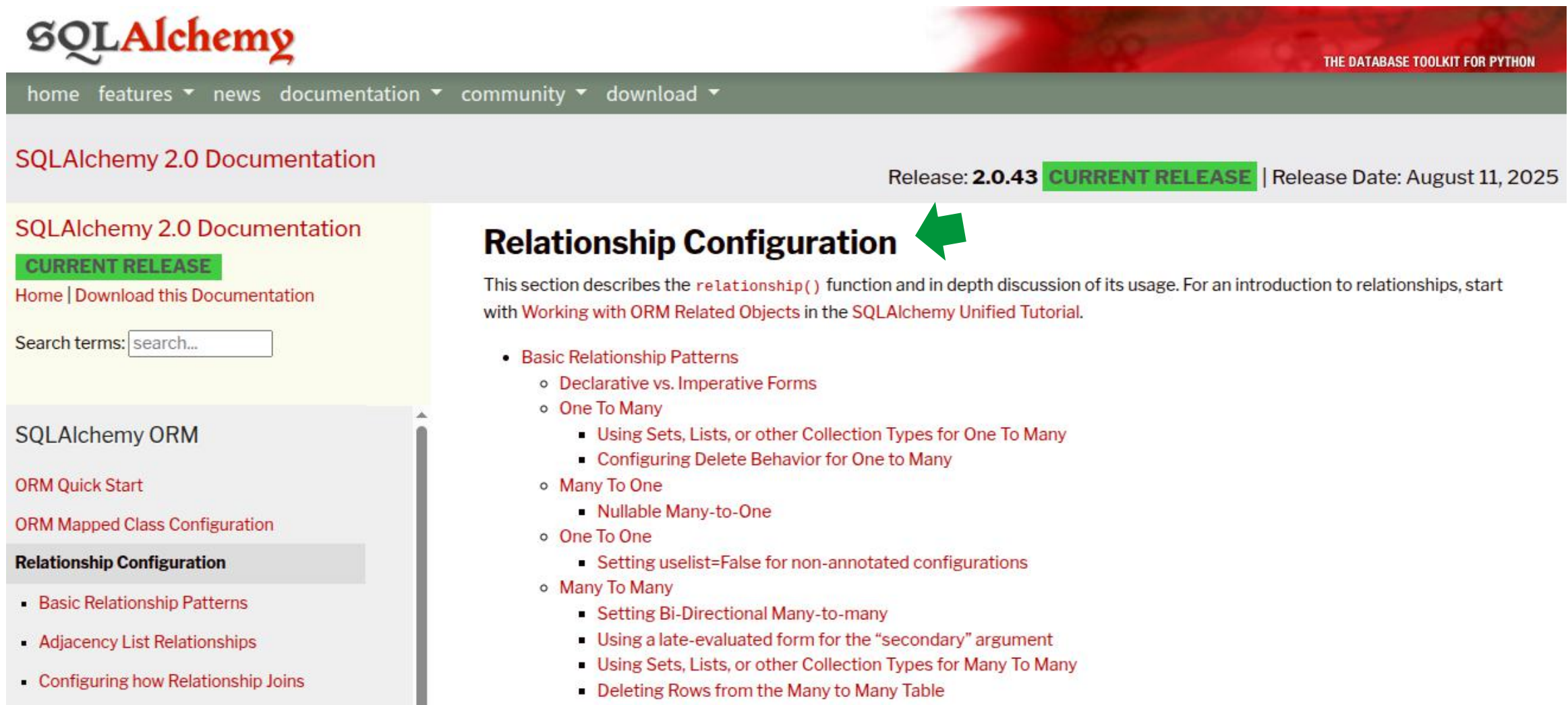
    def check_password(self, password: str) -> bool:
        return check_password_hash(self.password_hash, password)

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



# Estabelecer o relacionamento User ↔ Role

- <https://docs.sqlalchemy.org/en/20/orm/relationships.html>



SQLAlchemy

THE DATABASE TOOLKIT FOR PYTHON

home features ▾ news documentation ▾ community ▾ download ▾

SQLAlchemy 2.0 Documentation

Release: **2.0.43** **CURRENT RELEASE** | Release Date: August 11, 2025

SQLAlchemy 2.0 Documentation

**CURRENT RELEASE**

Home | [Download this Documentation](#)

Search terms:

SQLAlchemy ORM

ORM Quick Start

ORM Mapped Class Configuration

**Relationship Configuration**

- Basic Relationship Patterns
- Adjacency List Relationships
- Configuring how Relationship Joins

## Relationship Configuration

This section describes the `relationship()` function and in depth discussion of its usage. For an introduction to relationships, start with [Working with ORM Related Objects](#) in the [SQLAlchemy Unified Tutorial](#).

- Basic Relationship Patterns
  - Declarative vs. Imperative Forms
  - One To Many
    - Using Sets, Lists, or other Collection Types for One To Many
    - Configuring Delete Behavior for One to Many
  - Many To One
    - Nullable Many-to-One
  - One To One
    - Setting `uselist=False` for non-annotated configurations
  - Many To Many
    - Setting Bi-Directional Many-to-many
    - Using a late-evaluated form for the “secondary” argument
    - Using Sets, Lists, or other Collection Types for Many To Many
    - Deleting Rows from the Many to Many Table

# Cadastrando roles no sistema e atribuindo a usuários

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


        # cria a role "admin" se não existir
        role_admin = Role.query.filter_by(name="admin").first()
        if not role_admin:
            role_admin = Role(name="admin")
            db.session.add(role_admin)
            db.session.commit()
            click.echo("Role 'admin' criada!")

        # cria um usuário "admin" se não existir
        if not User.query.filter_by(username="admin").first():
            user = User(username="admin", email="admin@example.com", role=role_admin)
            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...")
```


## Cadastrando roles no sistema e atribuindo a usuários



- Ao alterar o modelo de dados, a estrutura do banco existente pode ficar incompatível com o novo modelo. Por isso, é necessário remover o banco antigo e recriá-lo, garantindo que ele siga corretamente as regras atuais.
  - `rm instance/blog.sqlite`
  - `flask --app application init-db`



```
(Projeto_Flask_Blog) lucas@lucas-Inspiron-15-3520:~/Dropbox/IF_Baiano/web_II/codes/Projeto_Flask_Blog$ rm instance/blog.sqlite
(Projeto_Flask_Blog) lucas@lucas-Inspiron-15-3520:~/Dropbox/IF_Baiano/web_II/codes/Projeto_Flask_Blog$ flask --app application init-db
Role 'admin' criada!
Usuário admin criado!
Iniciando a base de dados...
```

# Cadastrando roles no sistema e atribuindo a usuários

instance >  blog.sqlite

  Filtrar 3 tab Linhas: 1

TABELAS				id		name	
>	post						
→	role						
	ROWID						
	# id						
	abc name						
>	user						

	1	1	admin
+	2		



# Cadastrando roles no sistema e atribuindo a usuários

Atualizar para PRO

TABELAS		id	username	email	password_hash	role_id
> post						
> role						
> user		1	admin	admin@example.com	scrypt:32768:8:1\$4JdiEnZVDWuUD4bZ\$33feb2ec25f...	1
ROWID	+	2				
id						
username						
email						
password_hash						
role_id						

# Cadastrando roles no sistema e atribuindo a usuários

```
@app.post("/")
@jwt_required()
def create_user():
    data = request.get_json()

    if not data or "username" not in data or "password" not in data:
        return {"error": "username e password é obrigatório"}, HTTPStatus.BAD_REQUEST

    username = data["username"]
    email = data.get("email")
    password = data["password"]
    role_id = data["role_id"]

    if User.query.filter_by(username=username).first():
        return {"error": "username já existe"}, HTTPStatus.CONFLICT

    user = User(username=username, email=email, role_id=role_id)
    user.set_password(password)

    db.session.add(user)
    db.session.commit()

    return {
        "id": user.id,
        "username": user.username,
        "email": user.email
    }, HTTPStatus.CREATED
```



# Cadastrando roles no sistema e atribuindo a usuários

```
@app.post("/")
@jwt_required()
def create_user():
    data = request.get_json()

    if not data or "username" not in data or "password" not in data:
        return {"error": "username e password é obrigatório"}, HTTPStatus.BAD_REQUEST

    username = data["username"]
    email = data.get("email")
    password = data["password"]
    role_id = data["role_id"]

    if User.query.filter_by(username=username).first():
        return {"error": "username já existe"}, HTTPStatus.CONFLICT

    user = User(username=username, email=email, role_id=role_id)
    user.set_password(password)

    db.session.add(user)
    db.session.commit()

    return {
        "id": user.id,
        "username": user.username,
        "email": user.email
    }, HTTPStatus.CREATED
```

Para criar um novo usuário com a função `create_user()`, é necessário atribuir um papel (Role) ao usuário.

## Cadastrando roles no sistema e atribuindo a usuários

- É necessário criar uma rota específica para cadastrar novos papéis no sistema.
- Essa rota permitirá que administradores adicionem roles dinamicamente, garantindo que novos tipos de usuários possam ser configurados com permissões adequadas.
- Sem essa funcionalidade, seria preciso modificar o código ou o banco manualmente sempre que um novo role fosse necessário, tornando a gestão de permissões menos flexível e mais propensa a erros.

# Cadastrando roles no sistema e atribuindo a usuários

```
from flask import Blueprint, request
from application import Role, db
from http import HTTPStatus

from flask_jwt_extended import jwt_required

→ app = Blueprint("roles", __name__, url_prefix="/roles")

→ @app.post("/")
  @jwt_required() ←
  def create_role():
      data = request.get_json()
      role = Role(name=data["name"])
      db.session.add(role)
      db.session.commit()

      return {
          "msg": "Role criada!"
      }, HTTPStatus.CREATED
```

# Cadastrando roles no sistema e atribuindo a usuários

- Registrando o Blueprint role:

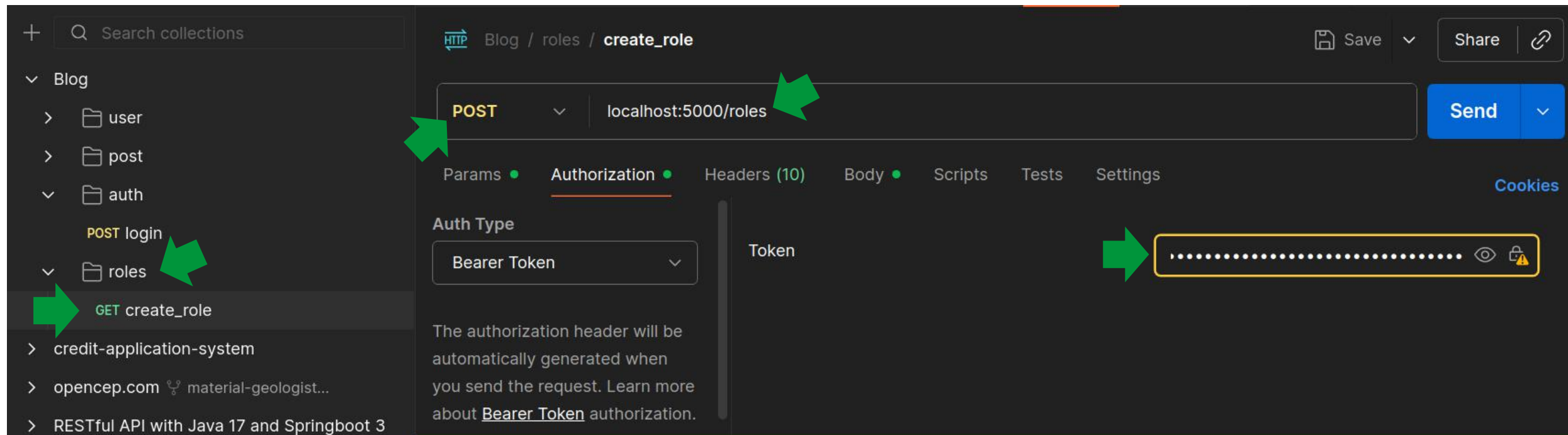
```
from controllers import user, auth, roles
app.register_blueprint(user.app)
app.register_blueprint(auth.app)
app.register_blueprint(roles.app)
```





# Cadastrando roles no sistema e atribuindo a usuários

- Usando o token de admin para cadastrar uma nova role :



The screenshot shows the Postman interface for a REST client. The left sidebar displays a collection named 'Blog' with folders 'user', 'post', 'auth', and 'roles'. The 'roles' folder is expanded, showing a 'GET create\_role' endpoint. The main panel shows a POST request to 'localhost:5000/roles'. The 'Authorization' tab is selected, showing 'Auth Type' as 'Bearer Token' and a 'Token' field with a masked value. A green arrow points to the 'Token' field. Another green arrow points to the 'POST' method dropdown. A third green arrow points to the 'roles' folder in the sidebar. A fourth green arrow points to the 'GET create\_role' endpoint. The 'Send' button is visible on the right.

Search collections

Blog / roles / create\_role

POST localhost:5000/roles

Params Authorization Headers (10) Body Scripts Tests Settings

Auth Type

Bearer Token

Token

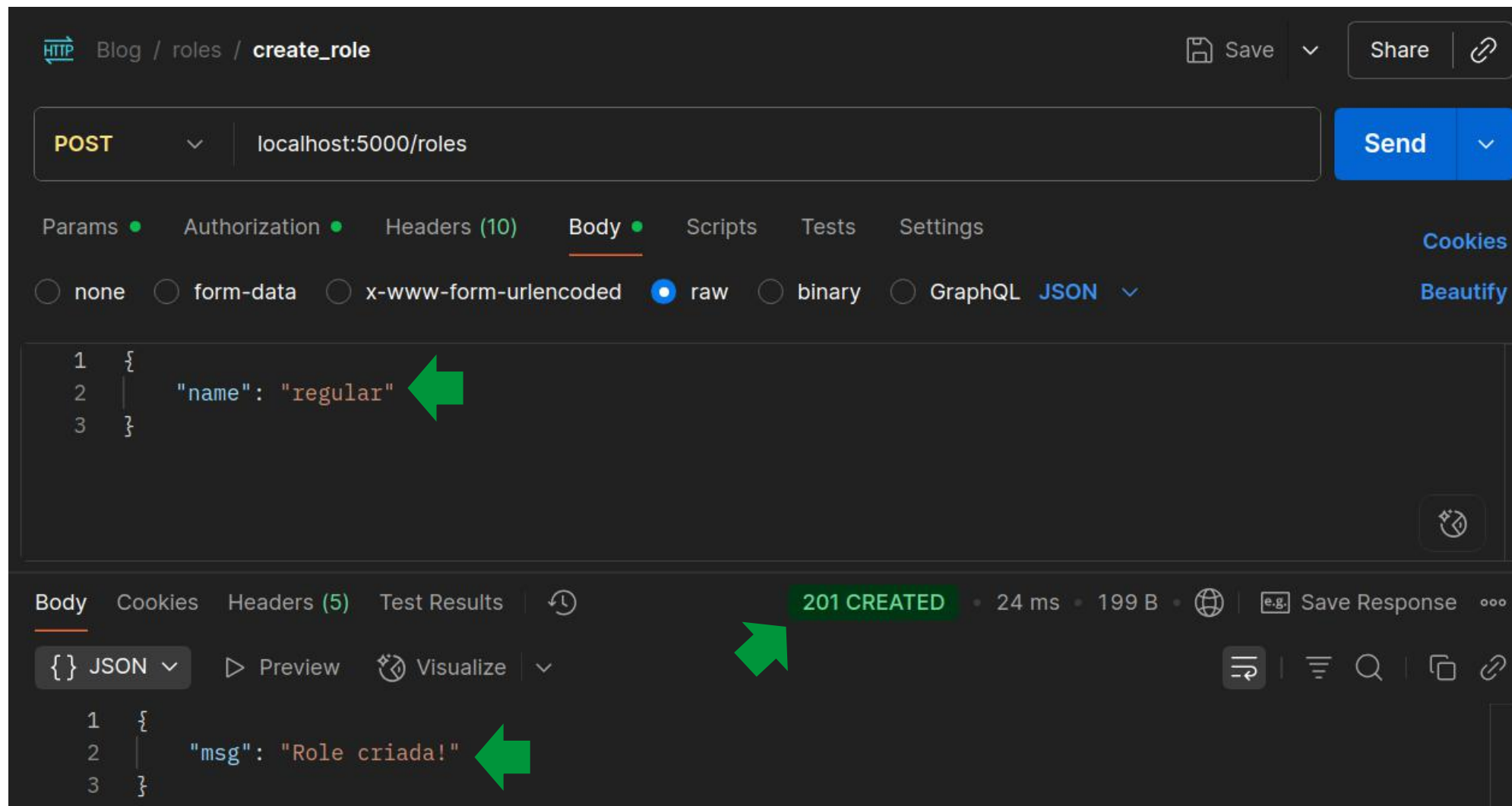
The authorization header will be automatically generated when you send the request. Learn more about [Bearer Token](#) authorization.

Send



# Cadastrando roles no sistema e atribuindo a usuários

- Usando o token de admin para cadastrar uma nova role:



The screenshot shows a REST client interface with the following details:


- URL:** `localhost:5000/roles`
- Method:** `POST`
- Body (raw):**



```
1 {  
2   "name": "regular"  
3 }
```
- Status:** `201 CREATED`
- Response Body (JSON):**







```
1 {  
2   "msg": "Role criada!"  
3 }
```

Green arrows highlight the `Send` button, the `"name": "regular"` field in the request body, the `201 CREATED` status, and the `"msg": "Role criada!"` field in the response body.

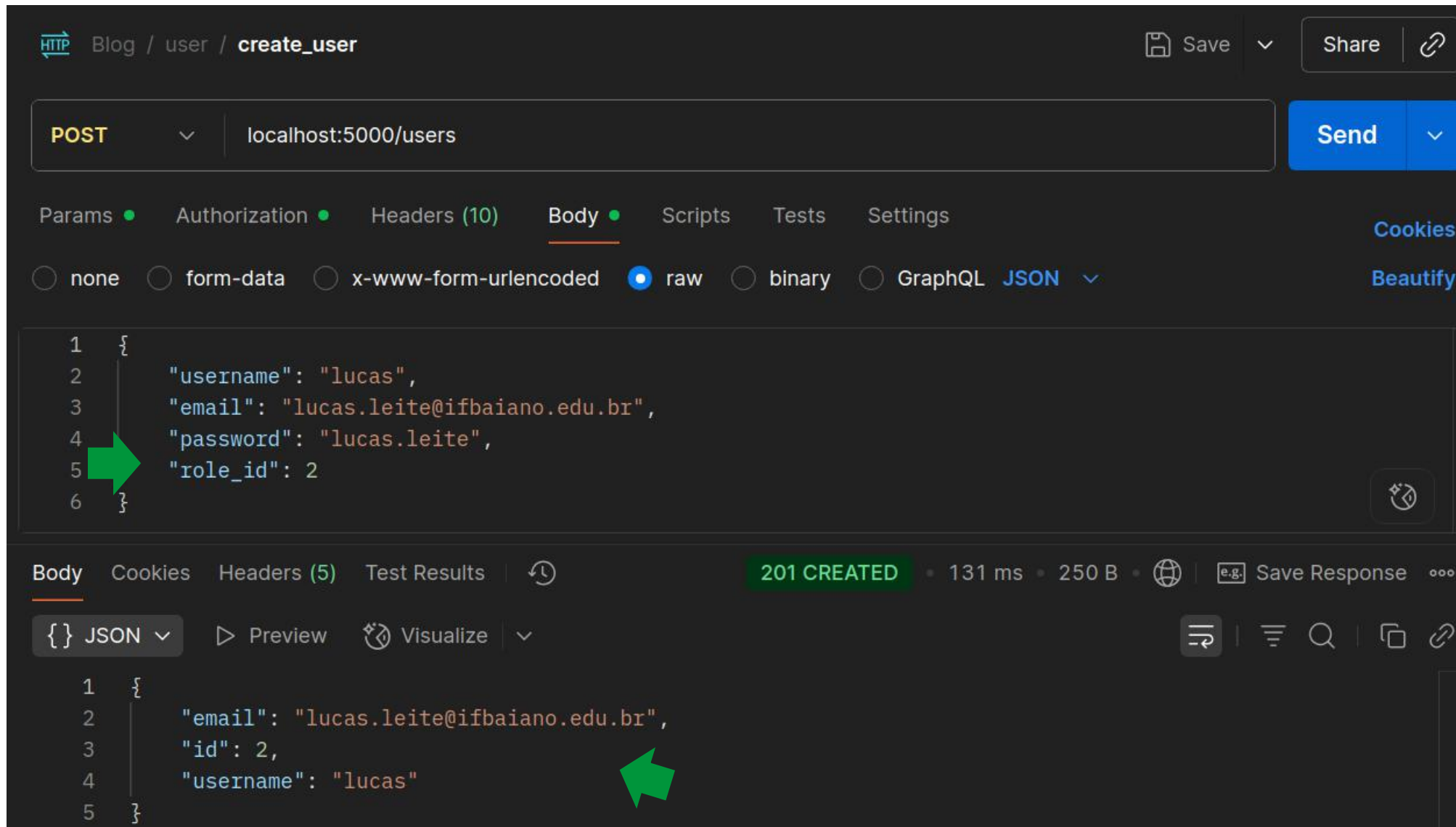
# Cadastrando roles no sistema e atribuindo a usuários

instance >  blog.sqlite

  Filtrar 3 l Linhas: 2 Filtrar 2 linh

TABELAS				id		name
>	post					
>	role					
	ROWID			1	1	admin
	id			2	2	regular
	name		+	3		
>	user					

# Cadastrando roles no sistema e atribuindo a usuários



The screenshot displays a REST client interface with the following details:

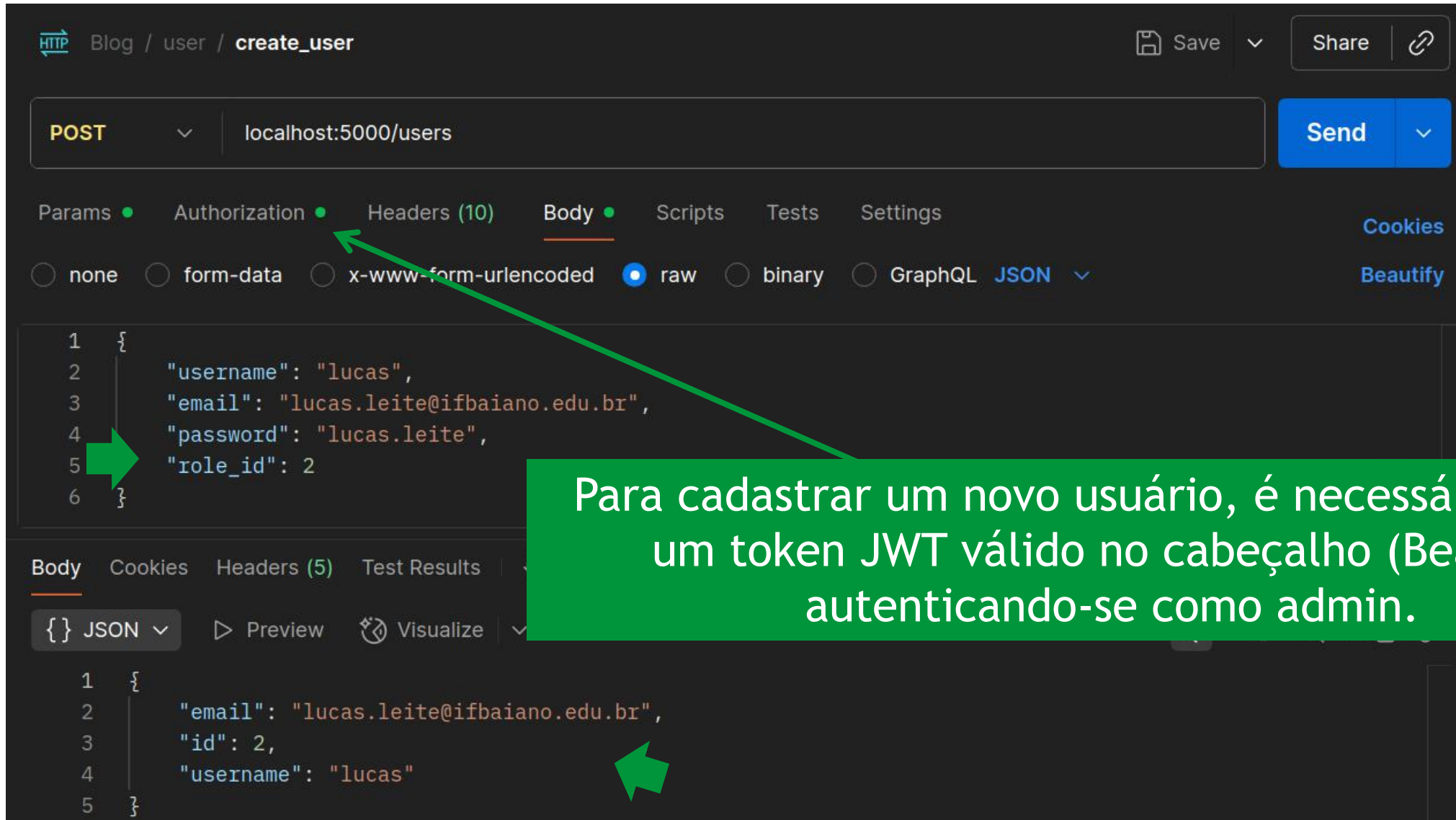
- URL:** `localhost:5000/users`
- Method:** `POST`
- Body (raw):**

```
1 {
2   "username": "lucas",
3   "email": "lucas.leite@ifbaiano.edu.br",
4   "password": "lucas.leite",
5   "role_id": 2
6 }
```
- Status:** `201 CREATED` (131 ms, 250 B)
- Response Body (JSON):**

```
1 {
2   "email": "lucas.leite@ifbaiano.edu.br",
3   "id": 2,
4   "username": "lucas"
5 }
```

Green arrows highlight the `Send` button and the `password` field in the request body, and the `username` field in the response body.

# Cadastrando roles no sistema e atribuindo a usuários



The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** localhost:5000/users
- Authorization:** none
- Headers:** 10 (selected)
- Body:** raw (selected)
- JSON Body:**

```
1 {  
2   "username": "lucas",  
3   "email": "lucas.leite@ifbaiano.edu.br",  
4   "password": "lucas.leite",  
5   "role_id": 2  
6 }
```
- Response:** JSON (selected)  

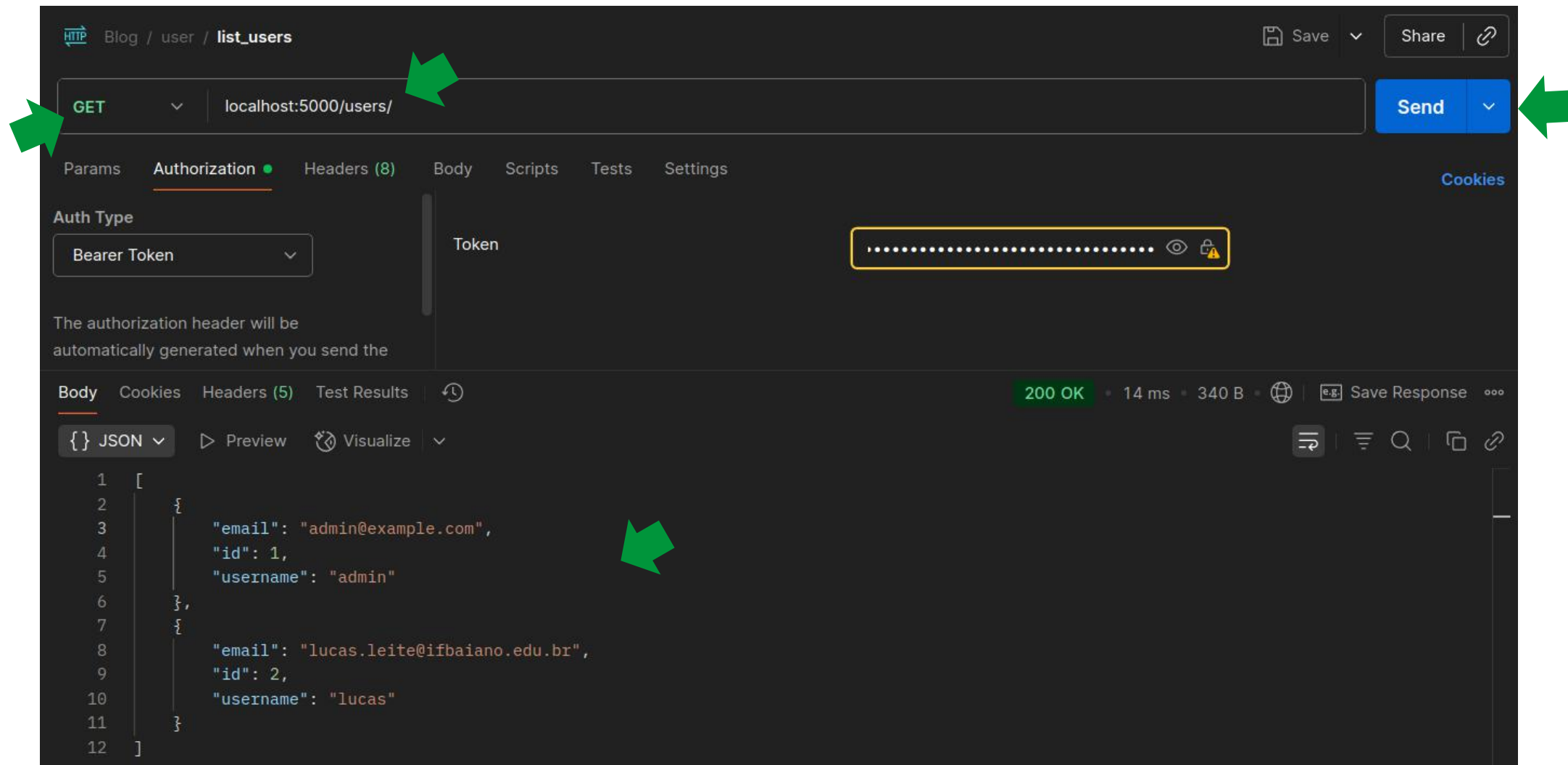
```
1 {  
2   "email": "lucas.leite@ifbaiano.edu.br",  
3   "id": 2,  
4   "username": "lucas"  
5 }
```

Green arrows point to the **Send** button, the **Authorization** tab, and the **role\_id** field in the request body, and the **id** field in the response body.

Para cadastrar um novo usuário, é necessário enviar um token JWT válido no cabeçalho (Bearer), autenticando-se como admin.

# Cadastrando roles no sistema e atribuindo a usuários

- Consultando os usuários cadastrados:



The screenshot displays a REST client interface with the following components:

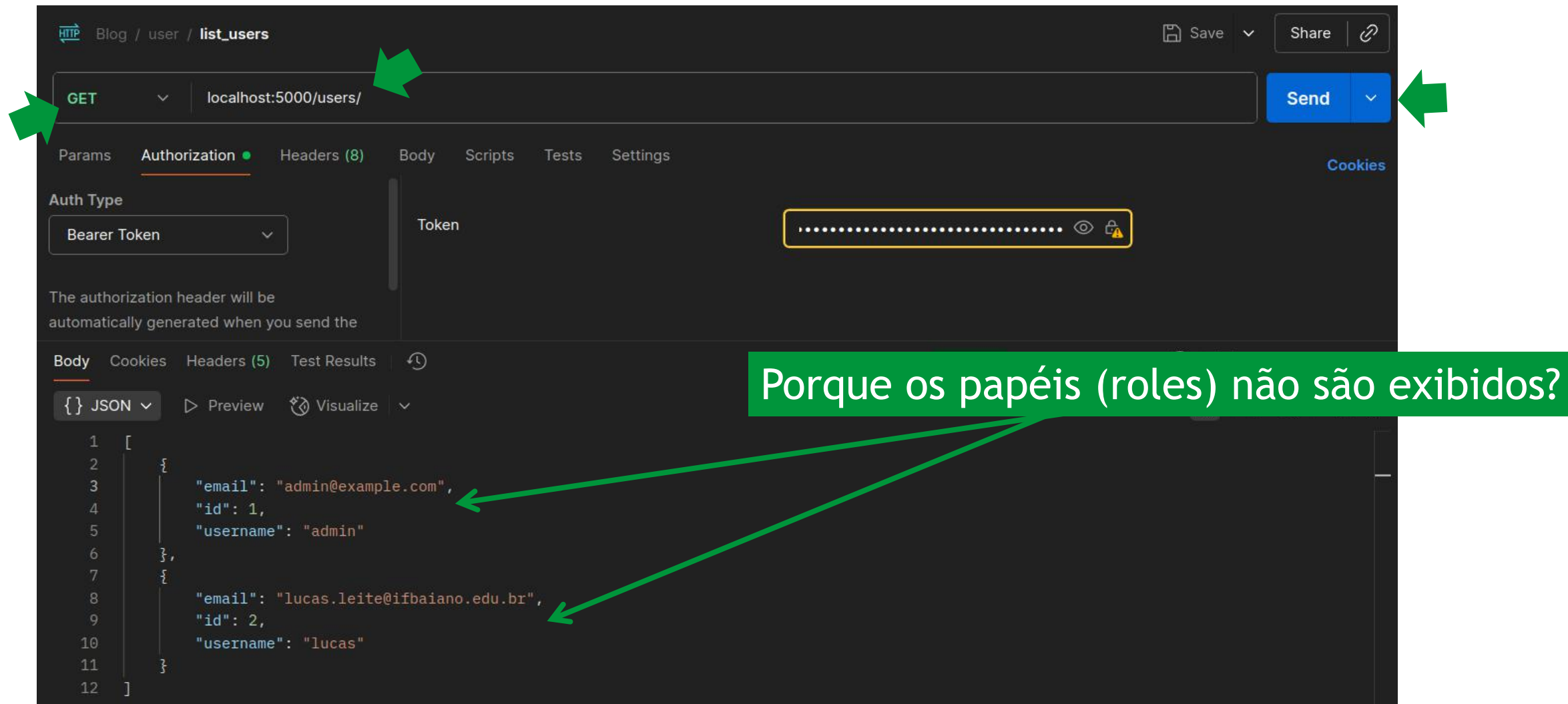
- Request Bar:** Method **GET** and URL **localhost:5000/users/**. A **Send** button is on the right.
- Authorization Tab:** Shows **Auth Type** as **Bearer Token**. The **Token** field contains a masked value (dots) and is highlighted with a yellow box.
- Response Section:** Shows a **200 OK** status with **14 ms** and **340 B**. The response body is in **JSON** format.
- Response Body:** A JSON array of two user objects:

```
1 [
2   {
3     "email": "admin@example.com",
4     "id": 1,
5     "username": "admin"
6   },
7   {
8     "email": "lucas.leite@ifbaiano.edu.br",
9     "id": 2,
10    "username": "lucas"
11  }
12 ]
```

Green arrows highlight the **GET** method, the **localhost:5000/users/** URL, the **Send** button, and the first user object in the response array.

# Cadastrando roles no sistema e atribuindo a usuários

- Consultando os usuários cadastrados:



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

Body Cookies Headers (5) Test Results

{ } JSON Preview Visualize

```
1 [
2   {
3     "email": "admin@example.com",
4     "id": 1,
5     "username": "admin"
6   },
7   {
8     "email": "lucas.leite@ifbaiano.edu.br",
9     "id": 2,
10    "username": "lucas"
11  }
12 ]
```

Porque os papéis (roles) não são exibidos?



# Cadastrando roles no sistema e atribuindo a usuários

```
@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
```



```
{ } JSON v Preview Visualize v
1  [
2  |
3  |   {
4  |       "email": "admin@example.com",
5  |       "id": 1,
6  |       "username": "admin"
7  |   }
8  ]
```

# Cadastrando roles no sistema e atribuindo a usuários

```
@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,
            "role_id": user.role_id
        }
        for user in users
    ], HTTPStatus.OK
```



```
{ } JSON v Preview Visualize v

1  [
2    {
3      "email": "admin@example.com",
4      "id": 1,
5      "role_id": 1,
6      "username": "admin"
7    }
8  ]
```

# Cadastrando roles no sistema e atribuindo a usuários

```
@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,
            "role": {
                "id": user.role_id,
                "name": user.role.name
            }
        }
        for user in users
    ], HTTPStatus.OK
```



```
{ } JSON v Preview Visualize v
1  [
2      {
3          "email": "admin@example.com",
4          "id": 1,
5          "role": {
6              "id": 1,
7              "name": "admin"
8          },
9          "username": "admin"
10     }
11  ]
```

# Restringir o acesso a endpoints e validar a identidade via JWT

- Para validar a identidade via JWT, é necessário extrair a identidade do token JWT, recuperar o usuário associado e verificar sua role para autorizar ou negar o acesso ao endpoint.
  - Consultar as roles no banco a cada requisição ou;
  - Incluir as roles diretamente no token.





# Restringir o acesso a endpoints e validar a identidade via JWT

```
@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=str(user.id))
    return {"access_token": access_token}, HTTPStatus.OK
```

# Restringir o acesso a endpoints e validar a identidade via JWT

```
@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"}

    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=str(user.id))
    return {"access_token": access_token}, HTTPStatus.OK
```

É necessário mudar o username como identidade para o id. O username é único mas pode ser alterado em contextos futuros, enquanto o id é a primary key do usuário e nunca muda.



# Restringir o acesso a endpoints e validar a identidade via JWT

```
@app.get("/")
@jwt_required()
def list_users():

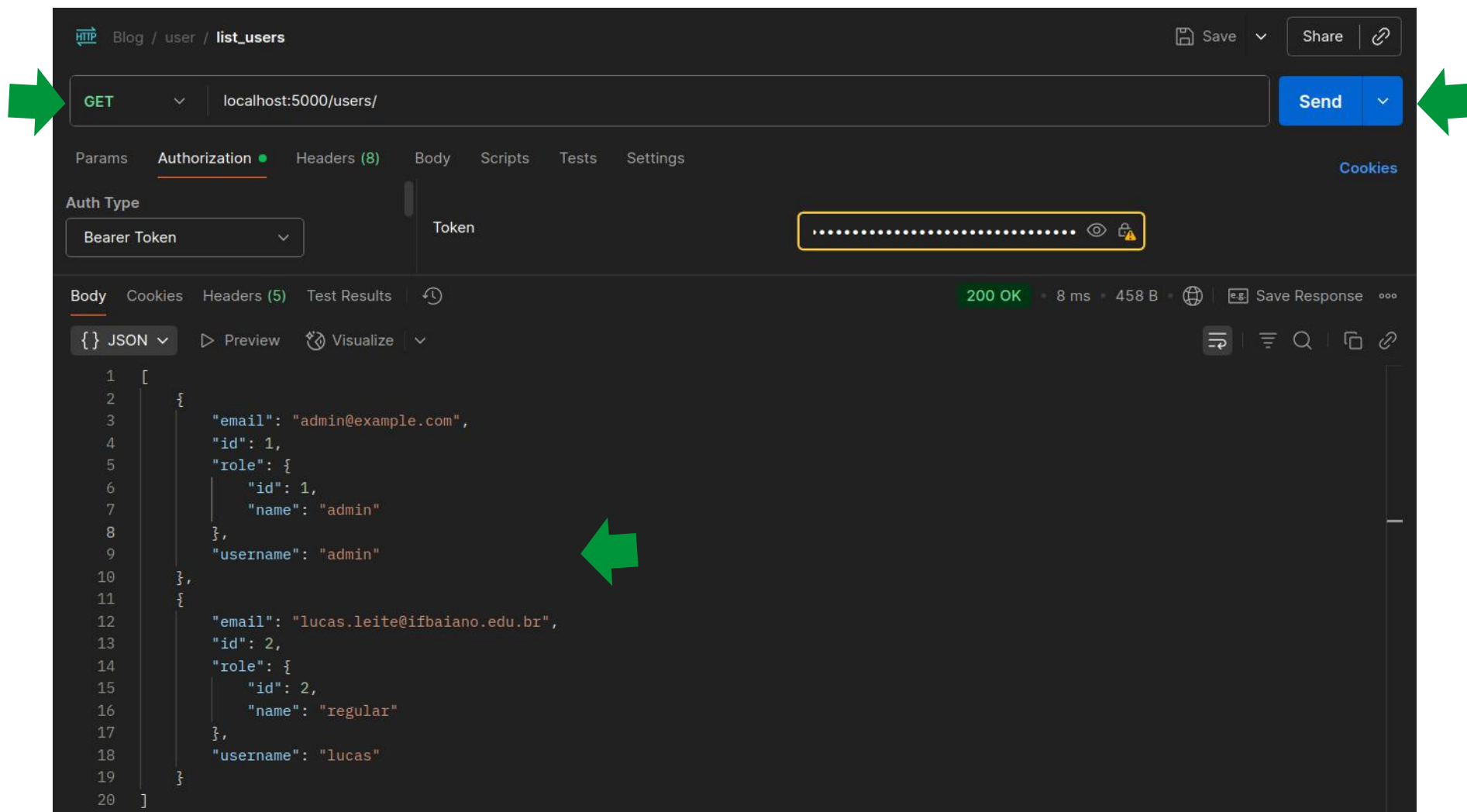
    user_id = int(get_jwt_identity())
    user = db.get_or_404(User, user_id)

    if user.role.name != "admin":
        return {"msg": "Usuário não possui acesso."}, HTTPStatus.FORBIDDEN

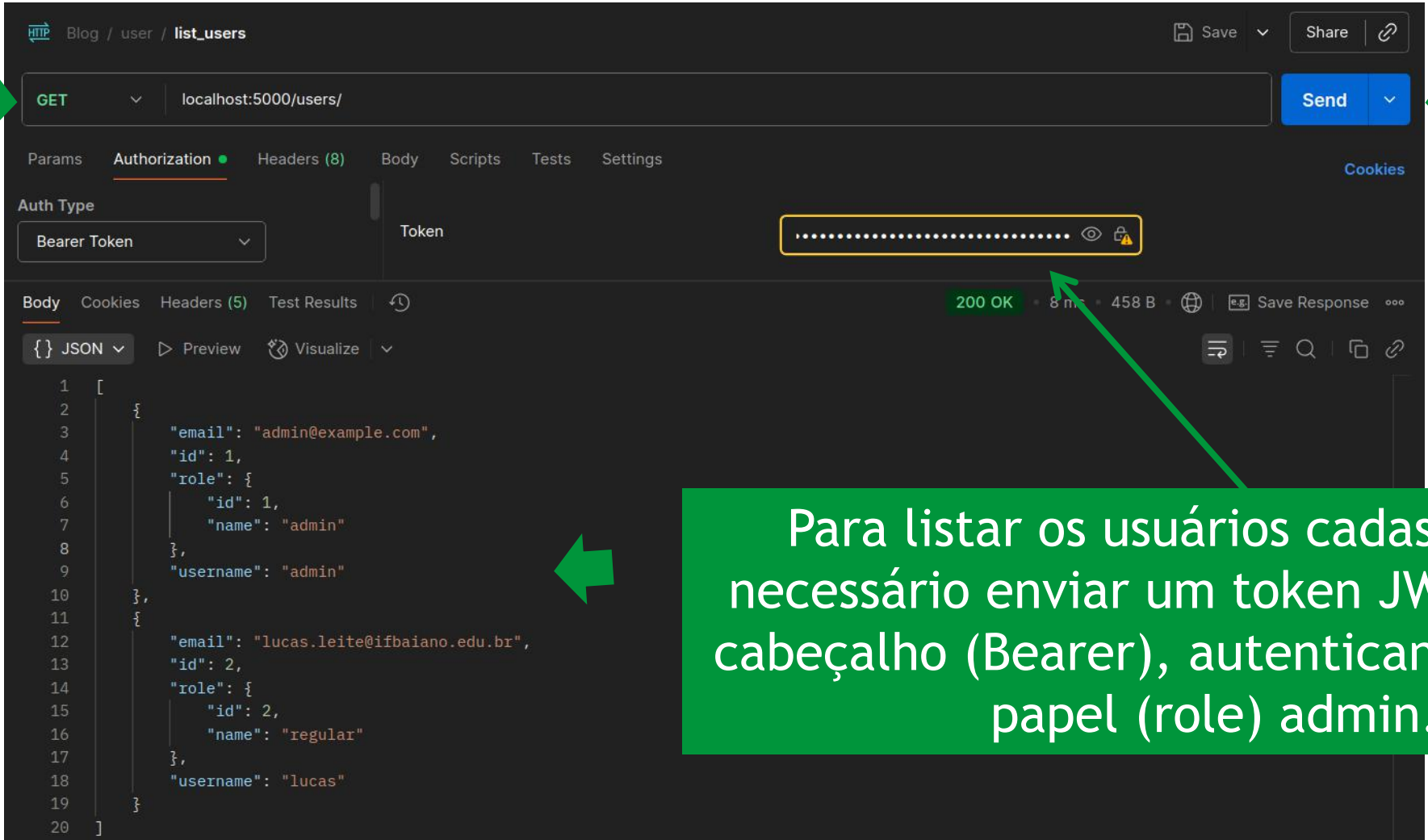
    query = db.select(User)
    result = db.session.execute(query)
    users = result.scalars().all()

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

# Restringir o acesso a endpoints e validar a identidade via JWT



# Restringir o acesso a endpoints e validar a identidade via JWT



The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** localhost:5000/users/
- Auth Type:** Bearer Token
- Token:** A masked token (indicated by dots and a lock icon).
- Status:** 200 OK
- Body:** JSON response showing two users:

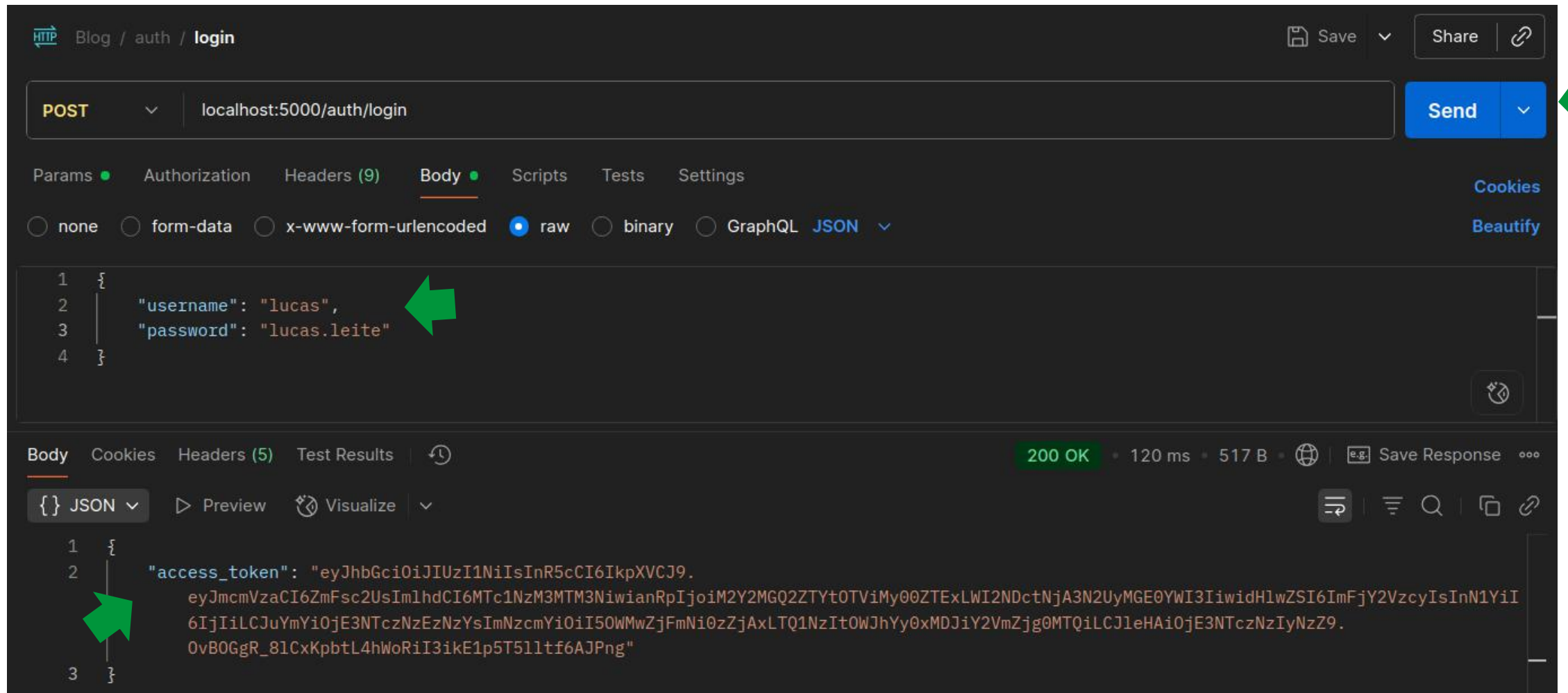
```
[
  {
    "email": "admin@example.com",
    "id": 1,
    "role": {
      "id": 1,
      "name": "admin"
    },
    "username": "admin"
  },
  {
    "email": "lucas.leite@ifbaiano.edu.br",
    "id": 2,
    "role": {
      "id": 2,
      "name": "regular"
    },
    "username": "lucas"
  }
]
```

Green arrows highlight the method, the token field, and the JSON response body.

Para listar os usuários cadastrados, é necessário enviar um token JWT válido no cabeçalho (Bearer), autenticando-se com o papel (role) admin.

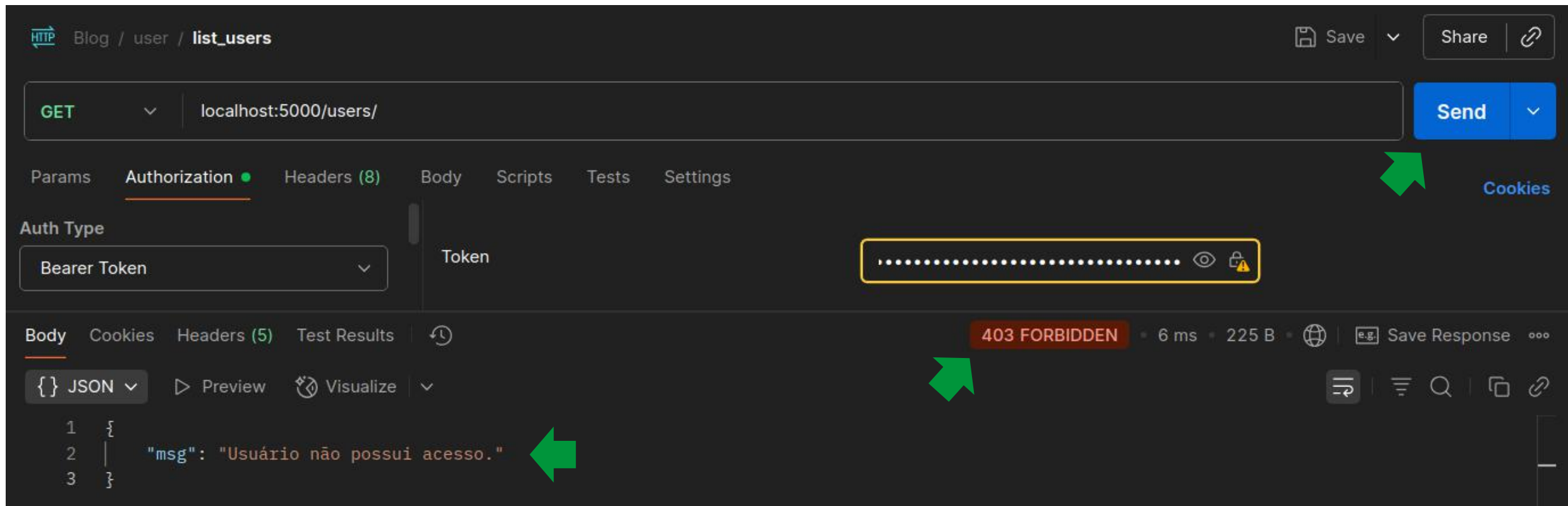
## Restringir o acesso a endpoints e validar a identidade via JWT

- Fazendo login com um usuário sem o papel (role) de admin:



# Restringir o acesso a endpoints e validar a identidade via JWT

- Consultando os usuários cadastrados sem autorização:



The screenshot shows a REST client interface with the following details:

- URL:** `localhost:5000/users/`
- Method:** `GET`
- Authorization:** Set to `Bearer Token`. The token field is highlighted with a yellow box and contains a masked token (dots).
- Response:** The status is `403 FORBIDDEN` (highlighted in red). The response body is JSON: 

```
{  "msg": "Usuário não possui acesso."}
```

Three green arrows highlight key elements: one points to the `Send` button, another points to the `403 FORBIDDEN` status, and a third points to the error message in the response body.

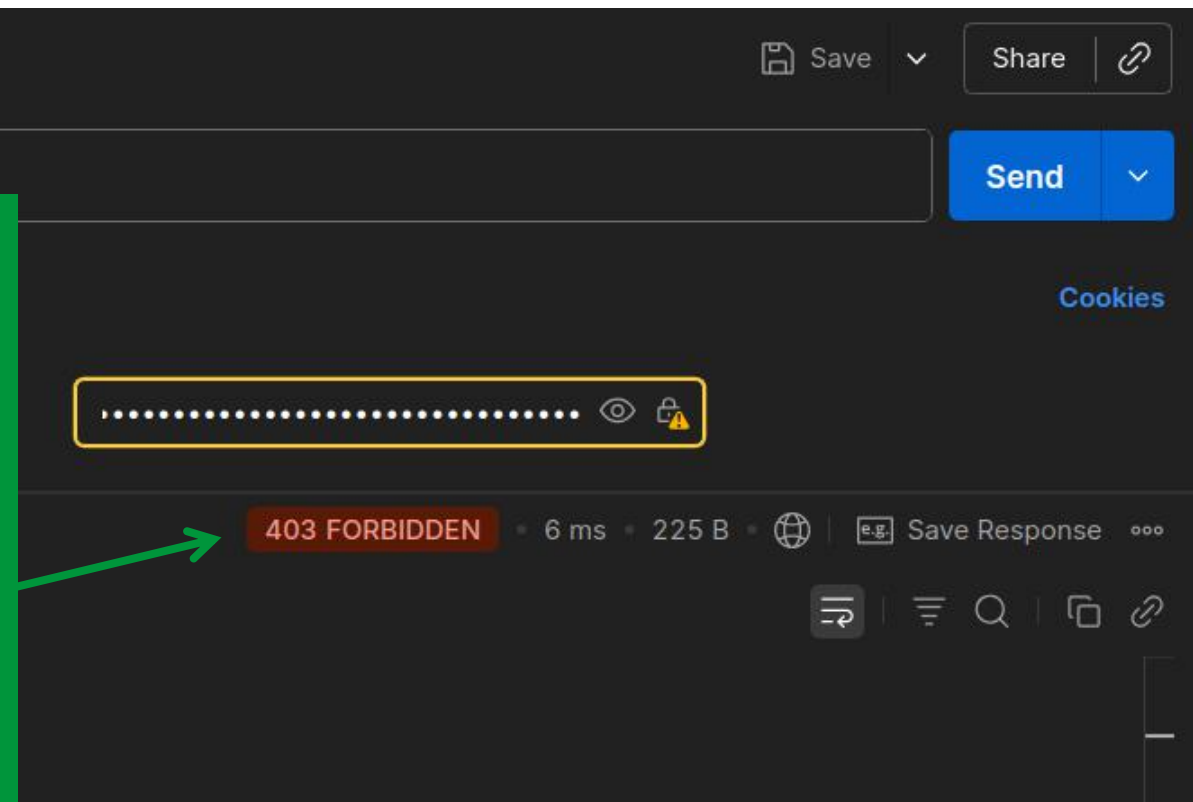


# Restringir o acesso a endpoints e validar a identidade via JWT

- Consultando os usuários cadastrados sem autorização:

O status code 403 Forbidden indica que o servidor entendeu a requisição feita pelo cliente, mas se recusa a autorizá-la.

Diferente do 401 Unauthorized, que normalmente significa falta de autenticação ou token inválido, o 403 é usado quando o cliente até está autenticado corretamente, mas não possui as permissões necessárias para acessar o recurso.



## Discussão:

- Um ponto crítico na implementação de autorização baseada em roles (RBAC) com JWT é decidir como verificar as permissões do usuário: consultar as roles no banco a cada requisição ou incluir as roles diretamente no token.



## Discussão:

- **Consultar as roles no banco a cada requisição:**
  - Sempre reflete o estado atual do banco: se a role do usuário mudar, a alteração é imediata. ✓
  - Mais seguro em sistemas onde roles podem ser alteradas frequentemente. ✓
  - Requer uma consulta ao banco em cada requisição protegida. ✗
  - Pode impactar performance se houver muitas requisições. ✗
- **Incluir as roles diretamente no token:**
  - Sem consultas ao banco → mais rápido. ✓
  - Simples de implementar para sistemas pequenos. ✓
  - Se a role do usuário mudar no banco, o token antigo ainda terá a role antiga até expirar (menor controle de revogação de permissões). ✗

# Flask-Security

- A extensão Flask-Security fornece uma implementação pronta de autenticação, autorização e gerenciamento de roles, integrando-se com o SQLAlchemy, sem a necessidade de implementar manualmente toda a lógica de segurança. <https://flask-security-too.readthedocs.io/en/stable/>

## Project Links

[PyPI releases](#)  
[Source Code](#)  
[Issue Tracker](#)  
[Changes](#)

## Contents

[Welcome to Flask-Security](#)  
[Getting Started](#)  
[Customizing and Usage](#)  
[Patterns](#)  
[API](#)  
[Additional Notes](#)

## Quick search



# Flask-Security

Flask-Security allows you to quickly add common security mechanisms to your Flask application. They include:

1. Authentication (via session, Basic HTTP, or token)
2. User registration (optional)
3. Role and Permission management
4. Account activation (via email confirmation) (optional)
5. Password management (recovery and resetting) (optional)
6. Username management (configuration, recovery, change) (optional)
7. Two-factor authentication via email, SMS, authenticator (optional)
8. WebAuthn Support (optional)
9. 'social'/OAuth for authentication (e.g. google, github, ..) (optional)
10. Change email (optional)
11. Login tracking (optional)
12. JSON/Ajax Support

Many of these features are made possible by integrating various Flask extensions and libraries. They include:

# Exercícios

1. Analise as respostas das requisições e identifique pontos de melhoria e implemente-as, como por exemplo, o fato de `get_user(user_id)` não retornar o papel do usuário.
2. Implemente restrições de acesso adequadas: rotas de escrita (POST, PUT, PATCH, DELETE) devem ser acessíveis apenas a usuários com papel de admin, enquanto rotas de consulta (GET) devem permitir acesso tanto a usuários com papel admin quanto aos usuários com papel regular.



# Dúvidas



# PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática  
Lucas Sampaio Leite



## 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.