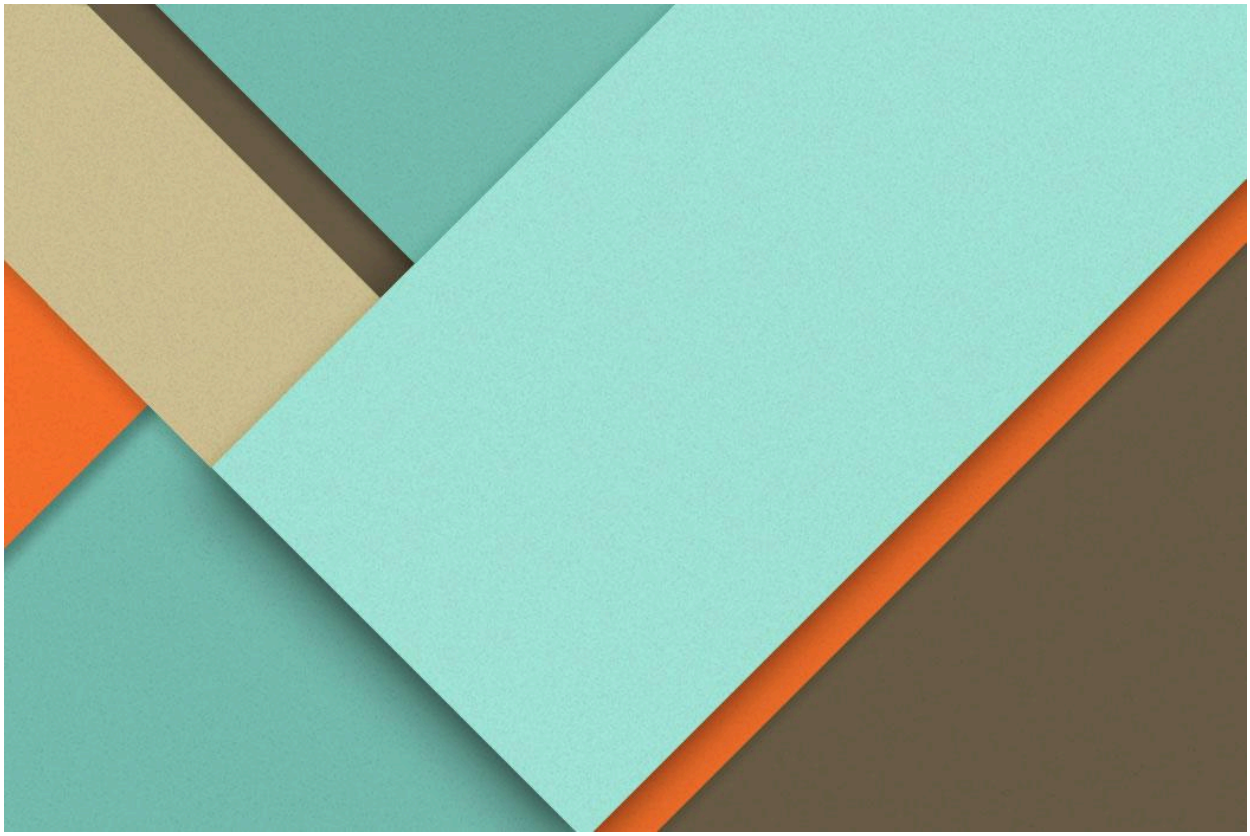


API REST com Python e Flask

—

Mariana dos Santos Costa Lima

Visão geral

Projeto para entender como funcionam as rotas e os métodos HTTP, usando API REST com Flask.

Objetivos

1. Compreender o conceito de arquitetura REST e sua importância em sistemas distribuídos.
2. Implementar uma API REST básica utilizando Python e Flask.
3. Experimentar operações CRUD (Create, Read, Update, Delete) em recursos distribuídos.
4. Testar endpoints utilizando ferramentas como Postman ou cURL.
5. Refletir sobre segurança, escalabilidade e papel das APIs em aplicações distribuídas.

Introdução Conceitual

REST (Representational State Transfer) é um **estilo arquitetural** usado para criar APIs que sejam **simples, escaláveis e interoperáveis**.

Ele se baseia em **recursos, verbos HTTP e códigos de status**, entre outros princípios.

Recursos

Tudo no REST é um **recurso**, que representa algo concreto ou abstrato do sistema.

Cada recurso tem um **identificador único**, normalmente uma **URL**.

Exemplos de recursos na nossa API de alunos:

/alunos

Representa todos os alunos

/alunos/1

Representa o aluno com ID = 1

Verbos

O REST usa **verbos HTTP para indicar a ação a ser realizada no recurso**.

GET: Lê dados

POST: Cria novo recurso

PUT: Atualiza recurso

DELETE: Deleta recurso

PATCH: Atualiza recurso parcialmente

Códigos de status HTTP

Os **códigos de status** informam ao cliente o resultado da requisição.

Códigos:

200 OK: Sucesso

201 Created: Recurso criado

204 No Content: Sucesso sem corpo

400 Bad Request: Requisição inválida

404 Not Found: Recurso não encontrado

500 Internal Server Error: Erro no servidor

Papel das APIs em sistemas distribuídos modernos

Hoje, sistemas modernos quase sempre são **distribuídos** e as **APIs são o "elo" entre componentes**, permitindo escalabilidade, manutenibilidade e interoperabilidade.

Testes

Código Python da API implementada.

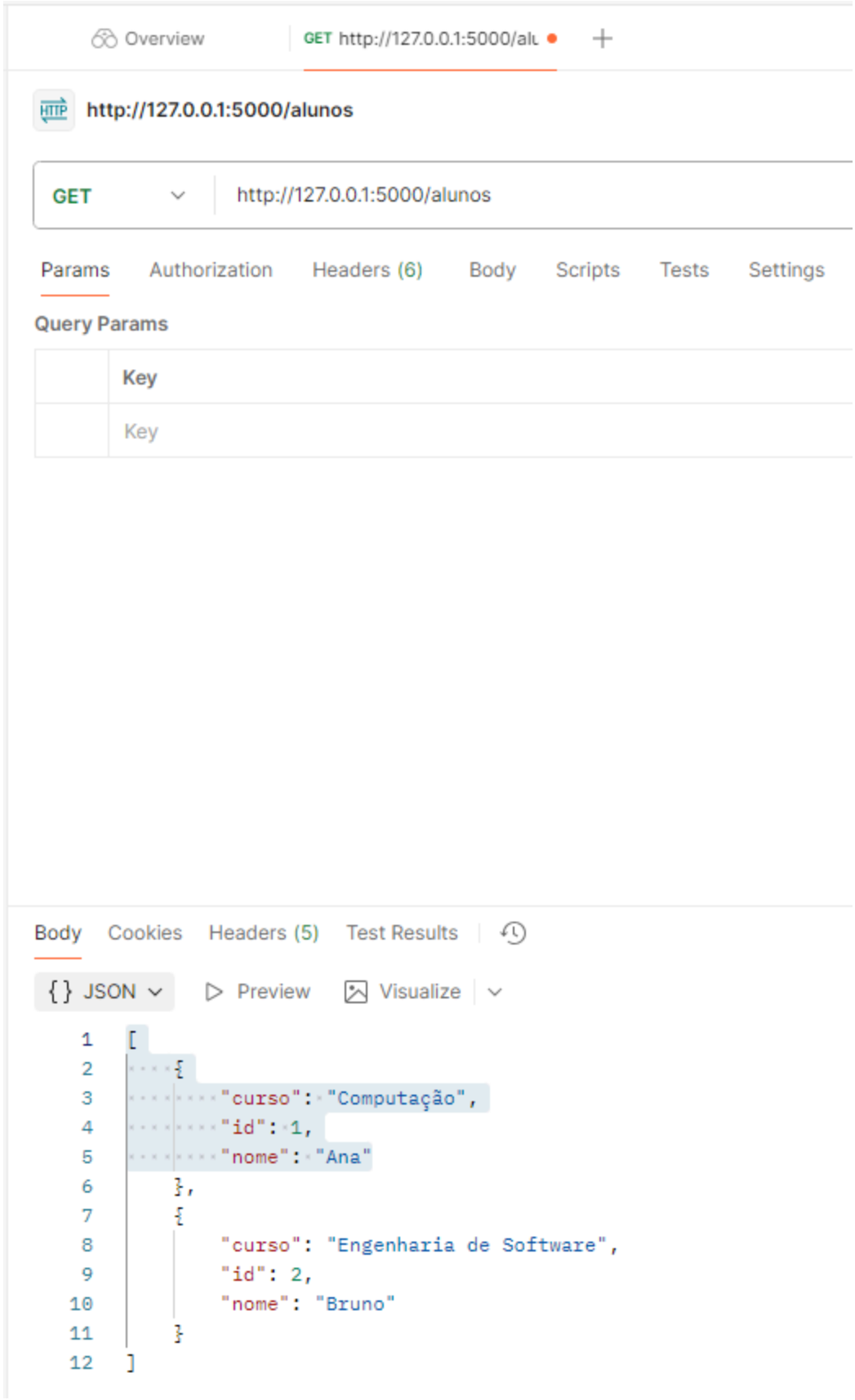
app.py

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

# Dados iniciais
alunos = [
    {"id": 1, "nome": "Ana", "curso": "Computação"},
    {"id": 2, "nome": "Bruno", "curso": "Engenharia de Software"}
]

# Rota GET - lista todos os alunos
@app.route('/alunos', methods=['GET'])
def get_alunos():
    return jsonify(alunos)

# Rota POST - adiciona um novo aluno
@app.route('/alunos', methods=['POST'])
def add_aluno():
    novo = {"id": len(alunos) + 1, **request.json}
    alunos.append(novo)
    return jsonify(novo), 201

# Rota PUT - atualiza um aluno existente
@app.route('/alunos/<int:id>', methods=['PUT'])
def update_aluno(id):
    for aluno in alunos:
        if aluno["id"] == id:
            aluno.update(request.json)
            return jsonify(aluno)
    return jsonify({"mensagem": "Aluno não encontrado"}), 404

# Rota DELETE - remove um aluno pelo ID
@app.route('/alunos/<int:id>', methods=['DELETE'])
def delete_aluno(id):
    global alunos
    alunos = [a for a in alunos if a["id"] != id]
    return jsonify({"mensagem": "Aluno removido com sucesso"})

# Executa o servidor Flask
if __name__ == '__main__':
    app.run(debug=True, port=5000)

```

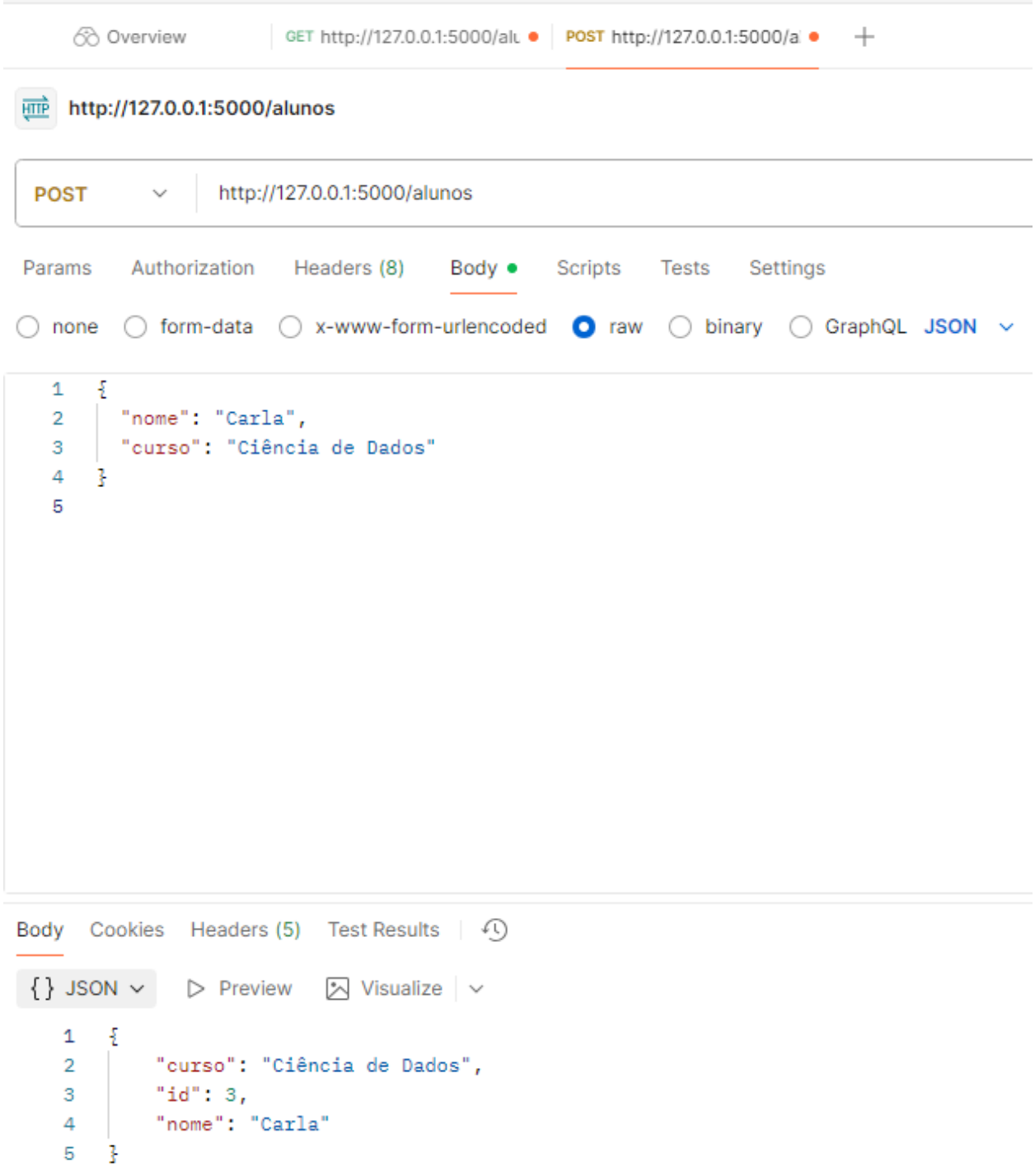
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Requisições e respostas no Postman

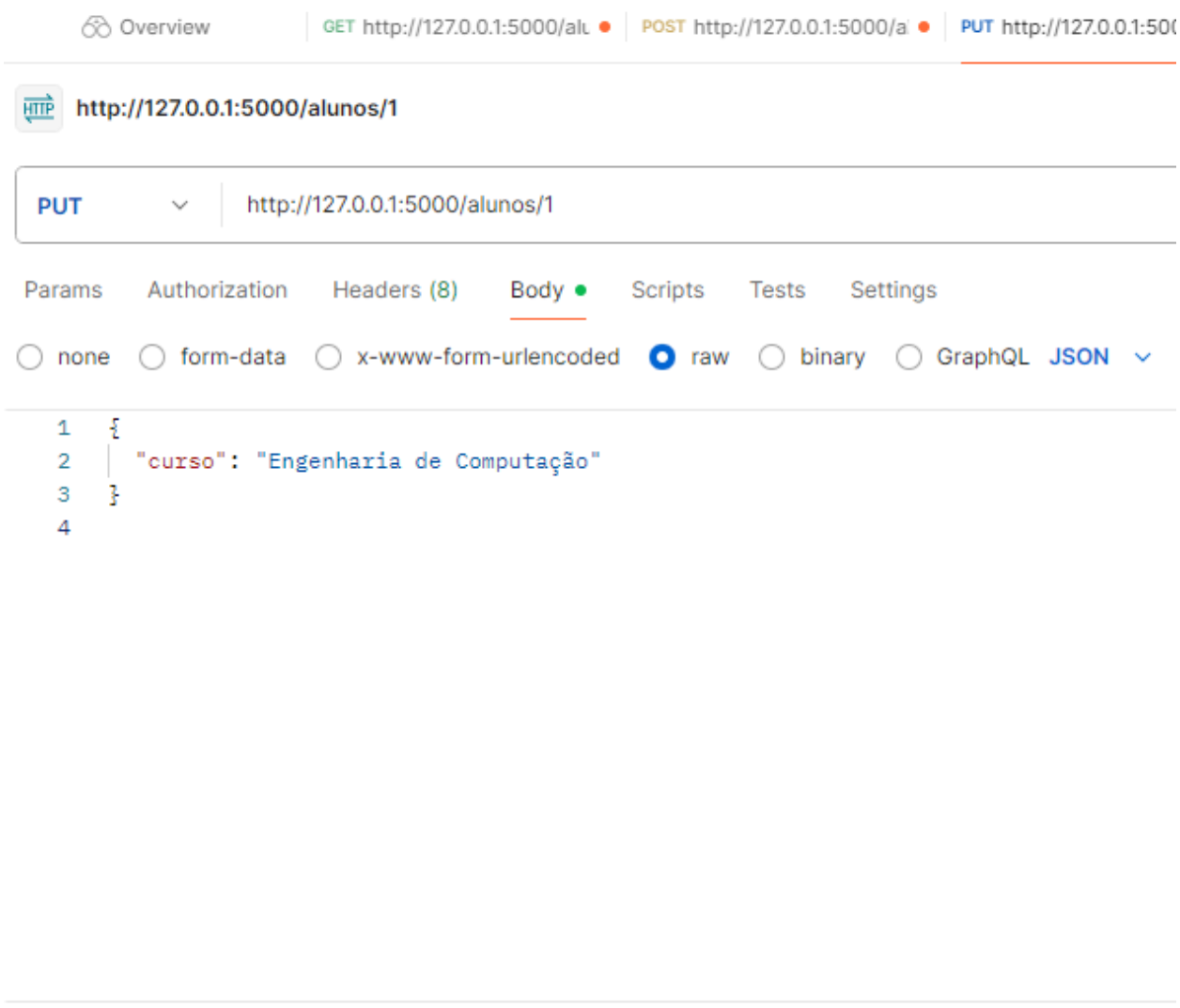
GET:

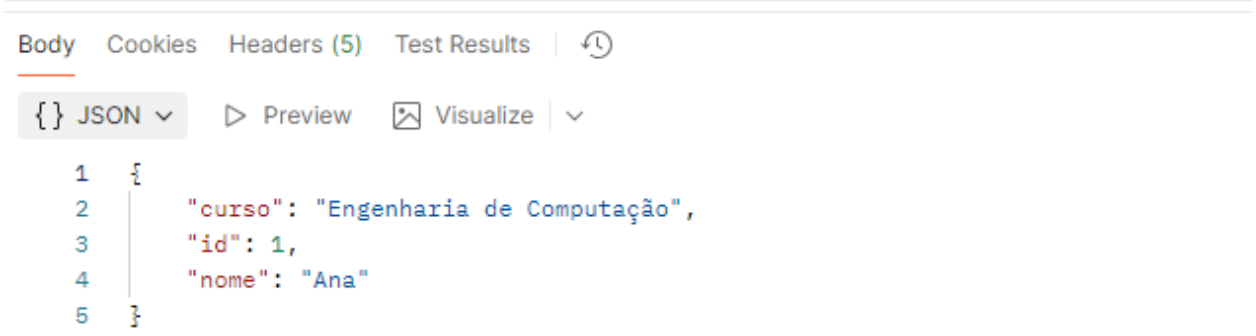

Overview
GET http://127.0.0.1:5000/alu
http://127.0.0.1:5000/alunos
GET
http://127.0.0.1:5000/alunos
Params
Authorization
Headers (6)
Body
Scripts
Tests
Settings
Query Params
Key
Key
Body
Cookies
Headers (5)
Test Results
{ } JSON
Preview
Visualize
1 [
2 {
3 "curso": "Computação",
4 "id": 1,
5 "nome": "Ana"
6 },
7 {
8 "curso": "Engenharia de Software",
9 "id": 2,
10 "nome": "Bruno"
11 }
12]

POST:

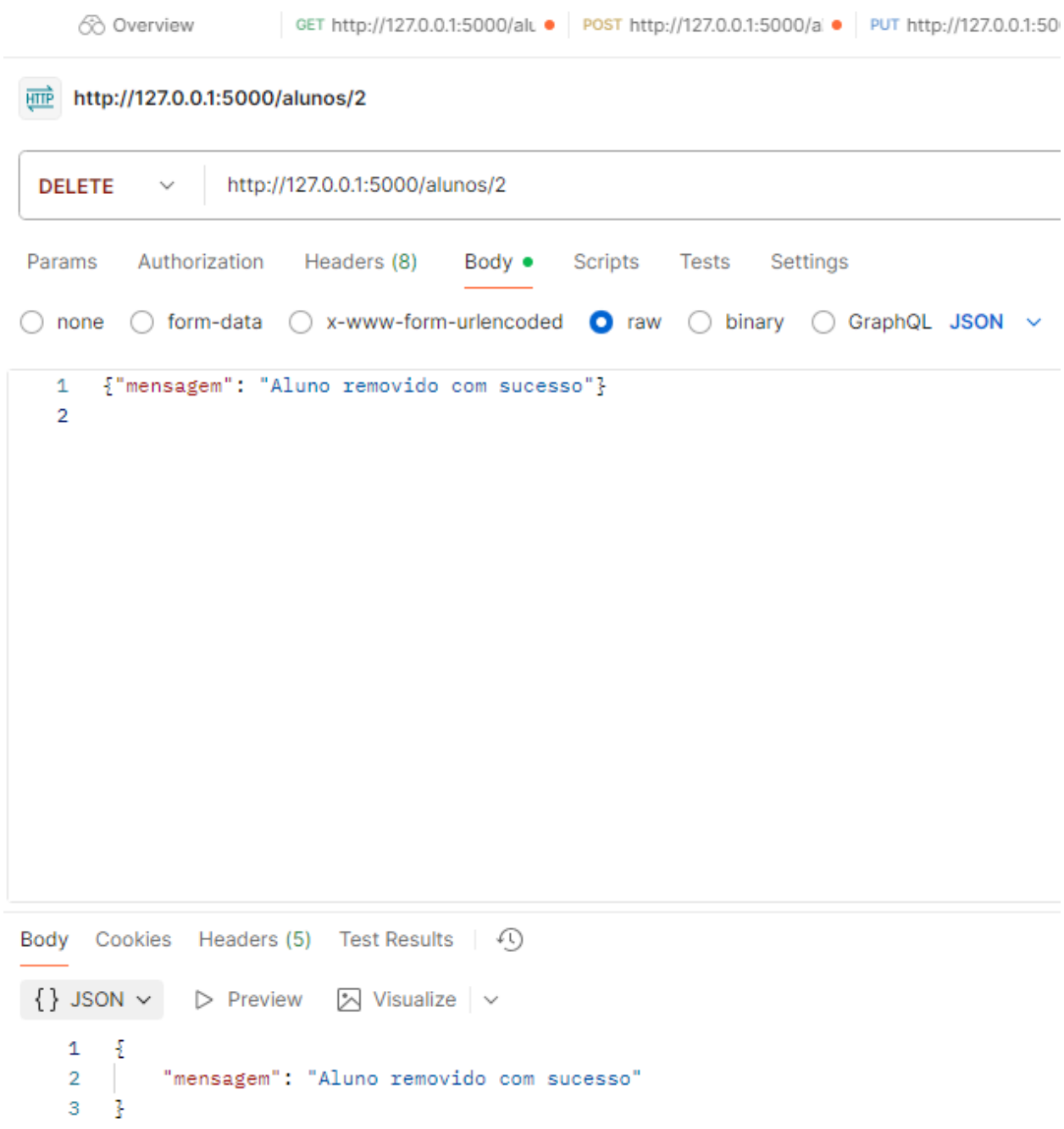

Overview
GET http://127.0.0.1:5000/alu
POST http://127.0.0.1:5000/a
http://127.0.0.1:5000/alunos
POST
http://127.0.0.1:5000/alunos
Params
Authorization
Headers (8)
Body
Scripts
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"nome": "Carla",
"curso": "Ciência de Dados"
}
Body
Cookies
Headers (5)
Test Results
{} JSON
Preview
Visualize
{
"curso": "Ciência de Dados",
"id": 3,
"nome": "Carla"
}

PUT:

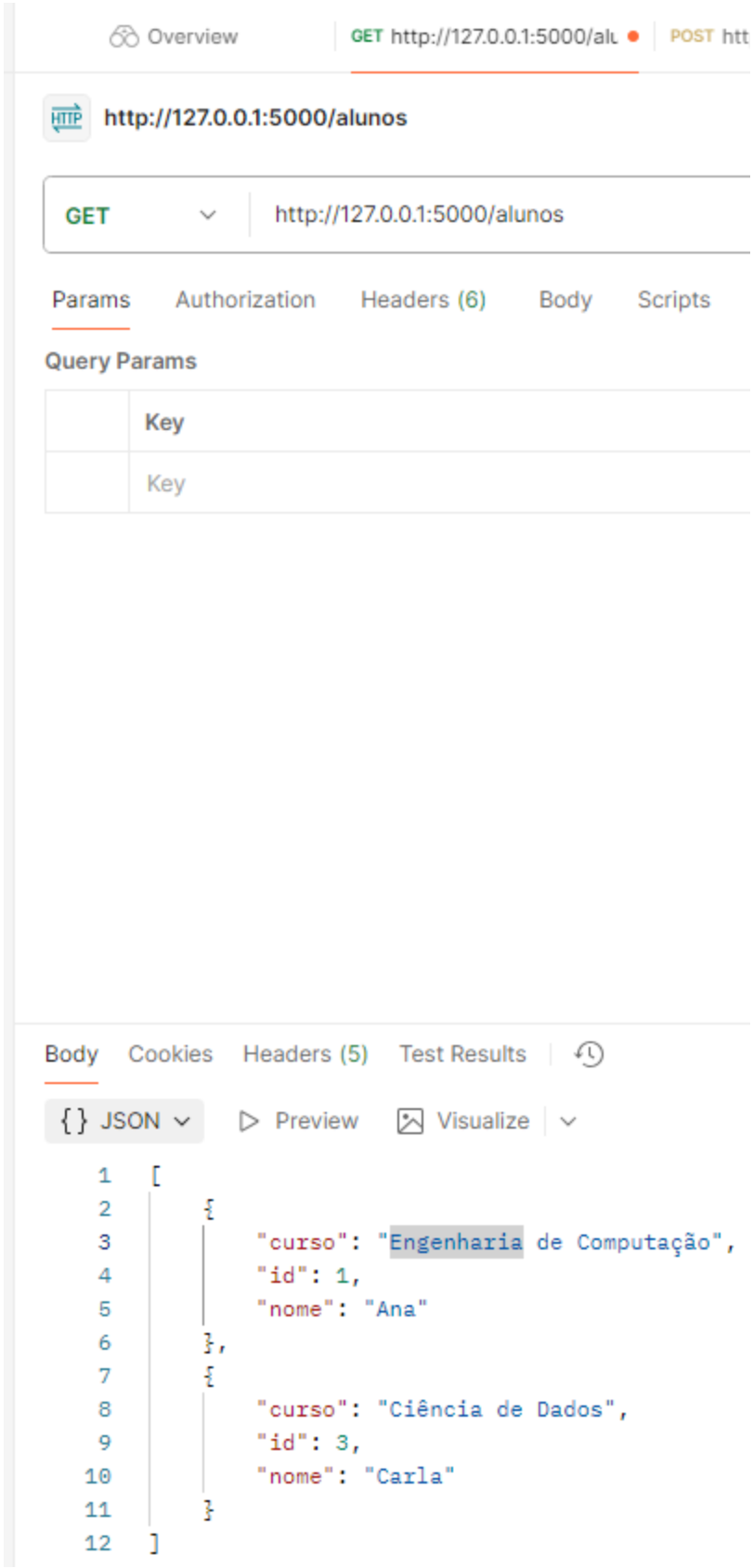

Overview
GET http://127.0.0.1:5000/alu
POST http://127.0.0.1:5000/a
PUT http://127.0.0.1:500
http://127.0.0.1:5000/alunos/1
PUT
http://127.0.0.1:5000/alunos/1
Params
Authorization
Headers (8)
Body
Scripts
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"curso": "Engenharia de Computação"
}


Body
Cookies
Headers (5)
Test Results
{ } JSON
Preview
Visualize
{
"curso": "Engenharia de Computação",
"id": 1,
"nome": "Ana"
}

DELETE:



GET (após todas as alterações):


Overview
GET http://127.0.0.1:5000/alu
POST htt
http://127.0.0.1:5000/alunos
GET
http://127.0.0.1:5000/alunos
Params
Authorization
Headers (6)
Body
Scripts
Query Params
Key
Key
Body
Cookies
Headers (5)
Test Results
{} JSON
Preview
Visualize
1 [
2 {
3 "curso": "Engenharia de Computação",
4 "id": 1,
5 "nome": "Ana"
6 },
7 {
8 "curso": "Ciência de Dados",
9 "id": 3,
10 "nome": "Carla"
11 }
12]

Reflexão e Discussão

1. Como essa API poderia ser expandida para rodar em múltiplos servidores em um ambiente distribuído?

Armazenando os dados em um **banco de dados centralizado**. Assim, cada instância do Flask acessa o mesmo banco, garantindo a consistência;

Executando **múltiplas instâncias do Flask** em servidores diferentes ou em containers Docker;

Cada instância pode rodar na mesma porta internamente, mas externamente você usa **balanceamento de carga**, o qual distribui requisições entre as instâncias disponíveis;

E **gerenciando sessões**.

2. Quais problemas podem surgir com concorrência no acesso ao recurso "alunos"?

Race Condition: Duas requisições tentam modificar o mesmo dado simultaneamente e apenas uma atualização é aplicada corretamente;

Inconsistência: Leituras e escritas não refletem o estado real;

Perda de dados: Atualizações podem sobrescrever alterações.

3. Como autenticação e autorização poderiam ser incorporadas?

Para a autenticação, um dos métodos comuns em APIs é o JWT (JSON Web Tokens)

1. O servidor gera um token após o login do usuário.
2. O cliente envia esse token em cada requisição (via cabeçalho `Authorization`).
3. O servidor valida o token antes de processar a requisição.

Para a autorização, pode ser feita a implementação simples de “roles”, em que cada role define quais ações o usuário pode executar na API.

4. Qual seria o impacto de integrar essa API a um banco de dados distribuído?

Escalabilidade (múltiplas instâncias do Flask podem acessar o mesmo banco);

Alta disponibilidade (se um nó falhar, outros ainda mantêm os dados);

Persistência (dados não se perdem ao reiniciar o servidor).

Mas requer atenção à

Consistência (alguns bancos distribuídos não garantem que todas as réplicas estejam imediatamente sincronizadas);

Latência (Operações podem ser mais lentas devido à replicação entre nós);

Gerenciamento de concorrência (O banco precisa lidar com múltiplas atualizações simultâneas, e conflitos podem ocorrer).