

Desenvolvimento Web II

Aula 05 - Web Service e API

Prof. Fabricio Bizotto

Instituto Federal Catarinense

fabricio.bizotto@ifc.edu.br

Ciência da Computação
23 de abril de 2025

1 REST

■ Experimentos

Web Service

REST

Representational State Transfer

REST

Como surgiu?

A arquitetura de sistema REST foi criada pelo cientista da computação **Roy Fielding em 2000**.

Anteriormente ele já havia trabalhado na criação do **protocolo HTTP e do URI**, um conjunto de elementos que identifica recursos nas aplicações web.

Buscando padronizar e organizar os protocolos de comunicação e desenvolvimento na internet, Fielding se uniu a um time de especialistas para desenvolver, **durante 6 anos**, as características da **REST**, que foi definida em sua tese de PhD.



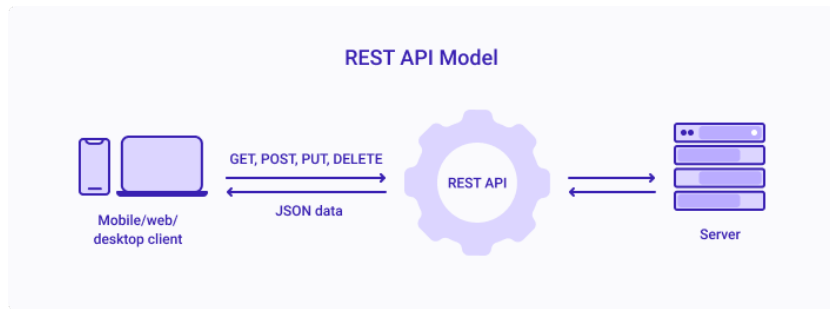


Figura: Estrutura REST

REST

Qual a diferença entre REST e RESTful?

REST

É uma **arquitetura** que define um conjunto de princípios para projetar aplicações web. Os critérios que devem ser cumpridos são:

- **Cliente-Servidor** - Separação entre o cliente e o servidor.
- **Stateless** - O servidor não armazena informações sobre o cliente. Cada requisição é independente.
- **Cache** - O servidor deve informar se a resposta pode ser armazenada em cache.
- **Interface Uniforme** - O cliente só precisa saber a URL do recurso e o servidor deve retornar os dados no formato apropriado.
- **Sistema em camadas** - O cliente não precisa saber se está se comunicando diretamente com o servidor ou com um intermediário.

RESTful

É uma **implementação** dos princípios REST.

REST

REST vs SOAP

SOAP	REST
SOAP é um protocolo	REST é uma arquitetura
Geralmente usa HTTP/HTTPS, mas pode usar outros	Usa apenas HTTP/HTTPS
XML	XML, JSON, HTML, etc
SOAP usa WSDL	Rest usa apenas a URL
É mais pesado	É mais leve
Não usa cache	Pode usar cache
WS-Security ¹	HTTPS

Tabela: SOAP vs REST

¹Conhecendo o WS-Security

Web Service

REST

Boas práticas

1. Documentação Clara

- Documente sua API para que os desenvolvedores possam entender facilmente como usá-la.
- Use uma ferramenta como o Swagger para documentar sua API.
- Descreva os recursos, parâmetros, cabeçalhos, corpo da solicitação, corpo da resposta, códigos de status, etc.

Exemplo

- **Descrição** - Retorna uma lista de produtos.
- **Método** - GET
- **URL** - /api/v1/produtos
- **Parâmetros** - page, limit, sort, order, ...
- **Cabeçalhos** - Authorization, Content-Type, ...
- **Corpo** - JSON
- **Resposta** - JSON

1. Documentação clara - *Exemplo* - Swagger

- O Swagger é uma ferramenta para documentar APIs REST.
- O Swagger permite que você descreva a estrutura da sua API para que os desenvolvedores possam entender como interagir com ela sem precisar ler o código-fonte.
- O Swagger gera automaticamente uma documentação interativa da API, que permite aos desenvolvedores enviar solicitações que chamam os endpoints da API.
- O Swagger pode ser usado com a maioria das linguagens de programação modernas e frameworks da web.

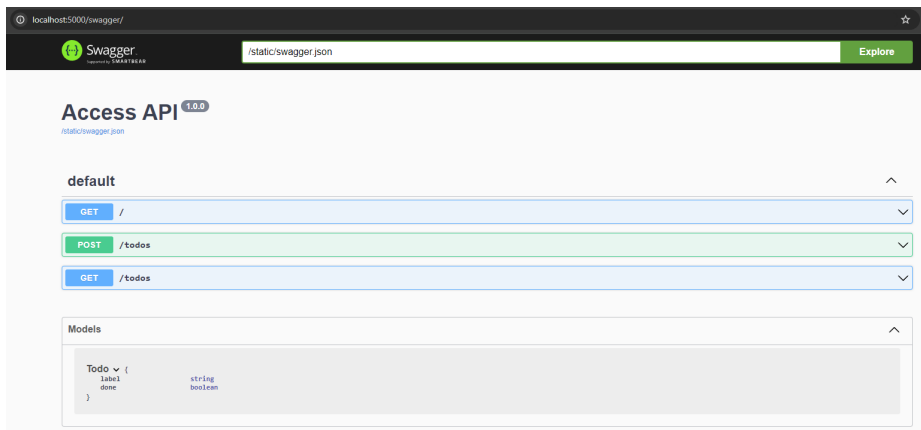


Figura: Swagger para documentar a API REST.

2. JSON

JSON é o formato de dados mais utilizado, embora você possa enviar dados em outros formatos como CSV, XML e HTML. A sintaxe JSON pode tornar os dados fáceis de ler para humanos.

```
"produto": {  
  "id": 1,  
  "nome": "Produto 1",  
  "descricao": "Descricao do produto 1",  
  "preco": 100.00,  
  "categorias": [  
    {  
      "id": 1,  
      "nome": "Categoria 1"  
    }  
  ]  
}
```

3. Versionamento da API

Inclua versões na sua API para garantir a compatibilidade com versões anteriores e permitir evolução controlada. Pode ser feito por meio de versões na URI ou por meio de cabeçalhos. O mais comum é usar a versão na URI.

Exemplo

- **URI** - `/api/v1/produtos` ou `/api/v2/produtos`
- **Cabeçalho** - `Accept: application/vnd.company.app-v1+json`

4. Nomes de Recursos Descritivos

- Use substantivos para nomear recursos.
- Use o plural para nomear coleções.
- Use o singular para nomear itens individuais.

Certo

- `/api/v1/produtos`
- `/api/v1/produtos/1`
- `/api/v1/produtos/1/categorias`

Errado

- `/api/v1/criarProduto`
- `/api/v1/obterProduto/1`
- `/api/v1/prodCat/1`

5. Verbos HTTP

Use métodos HTTP para operações CRUD. Por exemplo: GET, POST, PUT e DELETE .

Exemplo

GET	/api/v1/produtos
POST	/api/v1/produtos
GET	/api/v1/produtos/1
PUT	/api/v1/produtos/1
DELETE	/api/v1/produtos/1
PATCH	/api/v1/produtos/1 <i>(atualiza apenas alguns campos)</i>

6. Códigos de Status HTTP

- **1xx** - Informação
- **2xx** - Sucesso
- **3xx** - Redirecionamento
- **4xx** - Erro do cliente
- **5xx** - Erro do servidor

Exemplo

- | | |
|------------------------------------|---|
| ■ 200 - OK | ■ 404 - Não encontrado |
| ■ 201 - Criado | ■ 500 - Erro interno do servidor |
| ■ 400 - Requisição inválida | ■ 501 - Não implementado |
| ■ 401 - Não autorizado | ■ 503 - Serviço indisponível |

7. Paginação

Para coleções muito grandes, use paginação para limitar o número de itens retornados.

Exemplo

- `/api/v1/produtos?page=1&limit=10`

- `/api/v1/produtos?page=2&limit=10`

8. Filtros

Para coleções muito grandes, use filtros para limitar os itens retornados.

<https://www.netshoes.com.br/busca?q=chuteira&tamanho=40>

Exemplo

GET /api/v1/produtos?type=eletronicos

GET /api/v1/produtos?price_min=100&price_max=200

GET /api/v1/produtos?search=smartphone

9. Ordenação

Para coleções muito grandes, use ordenação para classificar os itens retornados.

Exemplo

GET `/api/v1/produtos?sort=nome`

GET `/api/v1/produtos?sort=nome&asc=false`

GET `/api/v1/produtos?sort=preco,vendas&ordem=desc,desc`

10. HATEOAS - Hypermedia As The Engine Of Application State

- Se possível, adote o HATEOAS para permitir que os clientes naveguem pela API dinamicamente usando links nos recursos para descrever as ações disponíveis a seguir.
- Pode não ser viável fora do escopo de CRUD.
- *Keep it simple and stupid (KISS)*. Nem sempre é necessário adicionar mais complexidade ao projeto.

```
"account": {  
  "account_number": 12345,  
  "balance": {  
    "currency": "usd",  
    "value": 100.00  
  },  
  "links": {  
    "deposit": "/accounts/12345/deposit",  
    "withdraw": "/accounts/12345/withdraw",  
    "transfer": "/accounts/12345/transfer",  
    "close": "/accounts/12345/close"  
  }  
}
```

11. Segurança

- Utilize sempre HTTPS para garantir a criptografia dos dados durante a transmissão. Isso protege contra ataques de interceptação (man-in-the-middle) e assegura a confidencialidade das informações.
- Evite chave primária incremental. Use UUIDs ou chaves primárias aleatórias para evitar a adivinhação de IDs. Isso evita escavação de dados .
 - Ex: /api/v1/users/a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11

11. Segurança - Autenticação com Basic Auth

Nesse método, o nome de usuário e a senha são codificados e incluídos no cabeçalho da solicitação HTTP usando a sintaxe **Authorization: Basic**. Embora seja simples, não é a opção mais segura, especialmente se a conexão não for protegida por SSL/TLS.

Basic Auth

```
GET /api/resource HTTP/1.1
```

```
Host: example.com
```

```
Authorization: Basic base64(username:password)
```

11. Segurança - Autenticação com Bearer Token

- Um token de acesso (Bearer Token) é incluído no cabeçalho da solicitação HTTP para autenticação.
- O cliente deve incluir o token de acesso em cada solicitação.
- O servidor valida o token de acesso e, se for válido, processa a solicitação.
- O esquema de autenticação Bearer foi originalmente criado como parte do OAuth 2.0 na RFC 6750 , mas às vezes também é usado sozinho. Da mesma forma que a autenticação Básica, a autenticação Bearer só deve ser usada via HTTPS (SSL).

Bearer Token

GET /api/resource HTTP/1.1

Host: example.com

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

REST

Bearer Token - Exemplo Prático

The image displays two sequential screenshots of a REST client interface, illustrating a login process for a Bearer Token.

Top Screenshot (Failed Login):

- Method:** POST
- URL:** localhost:5000/login
- Body:** JSON format with the following content:

```
1 {
2   "username": "admin",
3   "password": "errada"
4 }
```
- Status:** 401 UNAUTHORIZED
- Size:** 39 Bytes
- Response:** JSON format with the following content:

```
1 {
2   "message": "Invalid credentials"
3 }
```

Bottom Screenshot (Successful Login):

- Method:** POST
- URL:** localhost:5000/login
- Body:** JSON format with the following content:

```
1 {
2   "username": "admin",
3   "password": "admin"
4 }
```
- Status:** 200 OK
- Size:** 40 Bytes
- Time:** 4 ms
- Response:** JSON format with the following content:

```
1 {
2   "token": "xEqxyz0HnVi7btzYkLgVJw"
3 }
```

Figura: Bearer Token - Login

REST

Bearer Token - Exemplo Prático

The image shows a REST client interface with two requests. The first request is a GET to localhost:5000/products with an invalid Bearer token, resulting in a 401 Unauthorized status and a JSON response: {"message": "Unauthorized"}. The second request is a GET to localhost:5000/products with a valid Bearer token, resulting in a 200 OK status and a JSON array of two products: [{"id": 1, "name": "Product A"}, {"id": 2, "name": "Product B"}].

Query	Headers ²	Auth ¹	Body ¹	Tests
GET	localhost:5000/products	Send		
None	Basic	<u>Bearer</u>	OAuth 2	NTLM
Bearer Token				
token_invalido				

Status: 401 UNAUTHORIZED Size: 32

Response	Headers ⁵	Cookies
1 {		
2 "message": "Unauthorized"		
3 }		

Query	Headers ²	Auth ¹	Body ¹	Tests
GET	localhost:5000/products	Send		
None	Basic	<u>Bearer</u>	OAuth 2	NTLM
Bearer Token				
WVlKbxQsFGMvAUPzwGLlIg				

Status: 200 OK Size: 95 Bytes Time: 7 ms

Response	Headers ⁵	Cookies	Results
1 [
2 {			
3 "id": 1,			
4 "name": "Product A"			
5 },			
6 {			
7 "id": 2,			
8 "name": "Product B"			
9 }			
10]			

Figura: Bearer Token - Usando o Token de Acesso

JWT - JSON Web Token

- É um padrão aberto definido pela RFC 7519 que define um método compacto e autocontido para transmitir com segurança informações entre partes como um objeto JSON.
- As informações podem ser verificadas e confiadas porque são assinadas digitalmente.
- Os JWTs podem ser assinados usando um segredo (com o algoritmo HS256) ou um par de chaves pública/privada usando RSA ou ECDSA.
- Um JWT consiste em três partes separadas por pontos (.), que são:
 - **Cabeçalho** - Contém o tipo de token e o algoritmo de assinatura.
 - **Corpo** - Contém as informações.
 - **Assinatura** - Usada para verificar se o remetente do JWT é confiável.

JWT - Exemplo Prático

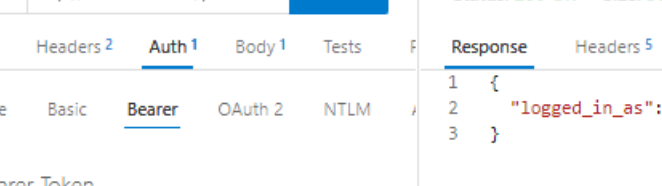
Efetuoando login com JWT. A resposta contém o token de acesso.

[illegible]

Figura: JWT - Exemplo (parte 1)

JWT - Exemplo Prático

Usando o token de acesso para acessar um recurso protegido da API.



GET Send

Status: 200 OK Size: 30 Bytes

Query Headers 2 **Auth 1** Body 1 Tests

None Basic **Bearer** OAuth 2 NTLM

Bearer Token

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImIhdCI6MTcwNTYwNDkwOCwianRpIjoib2M0NjZkZTMtNzI3MC00NjI0LTkyNGQtMjZiMzAxYTE1NzIhliwidHlwZSI6ImFjY2VzcyIsbnN1Yil6ImFkbWluciwibmJmljozNzA1NjA0OTA4LCJjc3JmljoiY2E1ZDhiMWMtYTk1ZS00MmYzLWI1NWltYzc4YWFiZTlwNWU1IiwiaXhwaWljoxNzA1NjA0OTY4fQ.YhJ5HFV_i9im5EdV234Kyz6NPSOSbc67k-sGlyBollQ
```

Response Headers 5 Cookies

```
1 {
2   "logged_in_as": "admin"
3 }
```

Figura: IWT - Exemplo (parte 2)

JWT - Exemplo Prático

O token expirou após 1 minuto, conforme definido no servidor.

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

- Method:** GET
- URL:** http://localhost:5000/protected
- Auth:** Bearer
- Token:** eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImh0bCI6MTcwNTYwNDkwOCwianRpIjoib2M0NjZkZTMtNzI3MC00NjI0LTkyNGQzMjZiMzAxYTE1NzlhliwidHlwZSI6ImFjY2VzcyIsbnN1Yil6ImFkbWluliwibmJmljozNzA1NjA0OTA4LCJpc3JmljoieY2E1ZDhiMWMtYTk1ZS00MmYzLWI1NWltYzc4YWFiZTlwNWU1liwiZXhwIjozNzA1NjA0OTY4fQ.YhJ5HFV_i9im5EdV234Kyz6NPSOSbc67k-sGlyBollQ
- Status:** 401 UNAUTHORIZED
- Size:** 33
- Response:**

```
{
  "msg": "Token has expired"
}
```

Figura: JWT - Exemplo (parte 3)

Validando o token JWT na página jwt.io.

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImhhdCI6MTcwNTYwNDkwOiwianRpIjoib2M0bjZkZmTmNzI3MC00bjI0LTkyNGQzMjZiMzAxYTE1NzlhIiwidHlwZSI6ImFjY2VzcyIsIn11YiI6ImFkbWluIiwibmJmIjoxNzA1NjA0OTA4LCJpc3JmIjoieY2E1ZDhiMWMtYTk1ZS00MmYzLWI1NWItYzc4YWFiZTIwNWE1Iiw1ZXhwIjoxNzA1NjA0OTY4fQ.YhJ5HFV_i9im5EdV234Kyz6NPS0Sbc67k-sGlyBoIIQ
```

- fresh(false): ainda não foi renovado
- iat: quando o token foi emitido
- jti: identificador único
- type: tipo do token
- sub: username
- nbf: pode ser usado depois desse tempo
- csrf: proteção contra ataque deste tipo
- exp: data de expiração

✓ Signature Verified

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{  "fresh": false,  "iat": 1705604908,  "jti": "7c466de3-7270-4624-924d-26b301a1579a",  "type": "access",  "sub": "admin",  "nbf": 1705604908,  "csrf": "ca5d8b1c-a95e-42f3-b55b-c78aabe205a5",  "exp": 1705604968}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  minha_chave_secreta_aq )
```

Chave Secreta☐ secret base64 encoded

SHARE JWT

Figura: JWT - Exemplo (parte 4)

Estratégia de Renovação do Token JWT

- O cliente envia o token de acesso para o servidor.
- O servidor verifica se o token de acesso é válido.
- Se o token de acesso for válido, o servidor retorna um novo token de acesso.
- Se o token de acesso for inválido, o servidor retorna um erro.

Como armazenar o token JWT no cliente?

- **Cookies** - O token de acesso é armazenado em um cookie. O cookie é enviado automaticamente pelo navegador para o servidor em cada solicitação.
- **LocalStorage** ou **SessionStorage** - O token de acesso é armazenado no armazenamento local ou de sessão do navegador.
- **Banco de Dados** - O token de acesso pode ser armazenado no IndexedDB ou WebSQL do navegador.

Importante

- Evite armazenar tokens em LocalStorage ou SessionStorage se sua aplicação for vulnerável a ataques XSS.
- Considere configurar o token como um cookie seguro com HttpOnly para mitigar alguns riscos.
- Mantenha o tempo de expiração (exp) do token curto para reduzir o impacto de um possível vazamento.
- Use HTTPS para proteger a transmissão do token entre o cliente e o servidor.

12. CORS (Cross Origin Resource Sharing)

Permite que os clientes acessem a API de um domínio diferente .

Cabeçalho

Access-Control-Allow-Origin

http://localhost:3000, *, ...

Access-Control-Allow-Methods

Métodos HTTP permitidos (GET, POST, PUT, DELETE, ...)

Access-Control-Allow-Headers

Indica quais cabeçalhos podem ser expostos como parte da resposta (Content-Type, Authorization, ...)

Access-Control-Allow-Credentials

Indica se o navegador deve incluir credenciais (como cookies ou cabeçalhos de autenticação) na solicitação.

REST

Boas Práticas - CORS - Exemplo Prático

```
exemplos > cors > app.py > ...
1  # --- Servidor rodando na porta 5000
2  from flask import Flask, jsonify
3  from flask_cors import CORS
4
5  app = Flask(__name__)
6  # --- Cliente rodando em http://localhost:5500 terá acesso a API
7  CORS(app, origins=['http://localhost:5500'])
8
9  @app.route('/api/data', methods=['GET'])
10 def get_data():
11     data = {'message': 'Dados da API acessados com sucesso!'}
12     return jsonify(data)
13
14 if __name__ == '__main__':
15     app.run(debug=True)
```

servidor

```

8  <body>
9  <h1>CORS Test</h1>
10 <button onclick="getData()">Obter Dados da API</button>
11 <p id="result"></p>
12
13 <script>
14     function getData() {
15         fetch('http://localhost:5000/api/data')
16         .then(response => response.json())
17         .then(data => {
18             document.getElementById('result').innerText =
19                 data.message;
20         })
21         .catch(error => {
22             document.getElementById('result').innerText =
23                 'Erro ao obter dados da API'
24         });
25     }
26 </script>
27 </body>
```

cliente
(servidor web)

Figura: CORS - Exemplo - Servidor e Cliente

← → ↺ 🏠 ⓘ localhost:5500/exemplos/cors/

CORS Test

Obter Dados da API

Dados da API acessados com sucesso!

Aceitando requisições em:
CORS(app, origins=['http://localhost:5000'])

❌ Access to fetch at 'http://localhost:5000/api/data' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

Aceitando requisições em:
CORS(app, origins=['http://localhost:3000'])

Figura: CORS - Exemplo - Simulação

13. Monitoramento e Logs

- Monitore a API para garantir que ela esteja sempre disponível.
- Registre todas as solicitações e respostas para fins de auditoria e depuração.

Exemplo

```
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Middleware para monitoramento do tempo de resposta
@app.after_request
def after_request(response):
    duration = time.time() - request.start_time
    logger.info(f"{request.method} {request.path} - Tempo de Resposta: {duration}")
    return response
```

15. Proteja contra ataques

SQL Injection: Use prepared statements ou ORM - Object Relational Mapping .

Exemplo - SQL Injection

exemplos > seg > sql.py > ...

```
1 @app.route('/login', methods=['POST'])
2 def login():
3     query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'" # Errado
4     query = f"SELECT * FROM users WHERE username = ? AND password = ?" # Correto
5     result = cursor.execute(query, (username, password)).fetchone()
6
7 # Simulation of SQL Injection
8 # SELECT * FROM users WHERE username = ' OR 'a'='a';-- AND password = '';
```

15. Proteja contra ataques (cont.)

Cross-Site Scripting (XSS): Use `escape` ou `sanitize` para evitar que os usuários insiram código HTML ou JavaScript nos dados.

Exemplo - XSS

```
@app.route("/")
def index():
    return """
    <form action="/search">
        <input name="query" type="text" />
        <input type="submit" value="Buscar" />
    </form>
    """

# <script>alert("XSS");</script>
# <style>body { background: red; }</style>
@app.route("/search")
def search():
    nome = request.args.get("query", "")
    nome_escapedo = escape(nome) # usar escape para evitar XSS
    return f"<h3>Buscando por: {nome_escapedo}</h3>"
```

15. Proteja contra ataques (cont.)

Cross-Site Request Forgery (CSRF): Use **tokens** para evitar que os usuários sejam enganados para executar ações indesejadas em nome deles. O token CSRF é um valor aleatório que é gerado pelo servidor web e enviado ao cliente. O cliente deve enviar o token CSRF de volta ao servidor web ao enviar um formulário. A requisição é rejeitada se o token CSRF não corresponder ao token esperado.

Exemplo - CSRF

```
<form action="/login" method="post">
  <input type="text" name="email" value="admin@example.com">
  <input type="password" name="senha" value="admin">
  <input type="hidden" name="csrf_token" value="[CSRF_TOKEN]">
  <input type="submit" value="Enviar">
</form>
```

Experimento 1

- Consumir a API REST PokéAPI, a Brasil API ou outra API pública.
- Navegue pela documentação da API e teste os endpoints.
- Use o **Postman** ou **Thunder Client** para testar a API.
- Faça requisições GET, POST, PUT e DELETE.
- Crie um cliente em (Python, JavaScript, outro) para consumir a API.

Experimento 2

Temas Sugeridos: Receitas Caseiras, Cardápio, Pizzaria, Diário Alimentar, Usuários, Produtos, etc.

- Criar uma API REST seguindo os princípios RESTful:
 - Uso correto dos verbos HTTP (GET, POST, PUT, DELETE, PATCH).
 - Recursos identificados por URLs significativas (ex: `/usuarios`, `/produtos/123`).
 - Uso adequado de códigos de status HTTP (200, 201, 400, 404, 500, etc.).
 - HATEOAS (opcional): forneça links para ações relacionadas nos retornos da API.
- A API deve expor pelo menos dois recursos (ex: usuários e produtos), com operações CRUD completas.
- Os dados devem ser retornados em formato JSON, com estrutura consistente e clara.
- A API deve ter paginação, ordenação e filtros nos endpoints de listagem.