

# Desenvolvimento Web II

## Aula 05 - Web Service e API

Prof. Fabricio Bizotto

Instituto Federal Catarinense  
*[fabricio.bizotto@ifc.edu.br](mailto:fabricio.bizotto@ifc.edu.br)*

Ciência da Computação  
20 de janeiro de 2024

## 1 Web Service

## 2 SOAP

### ■ Experimentos

## 3 REST

### ■ Experimentos

## 4 GraphQL

## Definição

- É uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes.
- Permite que aplicações se comuniquem independentemente de linguagem, software e hardware utilizados.
- É uma tecnologia utilizada para **padronizar** e **organizar** a comunicação entre aplicações.

## Características

- **Interoperabilidade** - Comunicação entre diferentes plataformas.
- **Independência de Linguagem** - Permite a comunicação entre diferentes linguagens de programação.
- **Formato de Mensagem** - Utiliza XML ou JSON.
- **Padrões Abertos** - Utiliza padrões abertos como SOAP e REST.

OI, EU ESTOU COM PROBLEMAS AQUI PARA FAZER UMA INTEGRAÇÃO COM O SISTEMA DE VOCÊS E ABRI UM CHAMADO AÍ HÁ ALGUNS DIAS. VOCÊ PODE VERIFICAR?

OK, ESPERA UM POUQUINHO...



É ESSE CHAMADO AQUI SOBRE UM WEBSERVICE?

ESSE MESMO!



ENTÃO, FALEI COM MEU CHEFE AQUI DO T.I. E ELE NÃO CONHECE ESSE SERVIÇO CHAMADO WEBSERVICE... VOCÊ TEM MAIS DETALHES DE QUAIS SISTEMAS ESTÃO ENVOLVIDOS?

**PLOFT!**



**VIDA DE PROGRAMADOR**

real historia;  
string sender = "Beatriz";

**#VIDASNEGRASIMPORTAM**



**#2019**



# VIDA DE PROGRAMADOR

.COM.BR

```
real historia;  
string sender;  
sender = "Herbet Mota";
```



#908

AQUI É O PROGRAMADOR,  
ESTOU COM UM PROBLEMA AO  
ACESSAR UM DOS WEB SERVICES DE  
VOCÊS. O RETORNO ESTÁ VINDO  
APENAS COM OS DECIMAIS DO  
VALOR, SEM A PARTE INTEIRA.



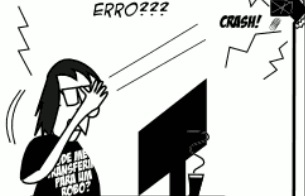
ENTENDO, MAS QUAL FOI A ABA  
DO WEB SERVICE QUE O SENHOR  
ESTÁ ACESSANDO?

NÃO, VEJA BEM... WEB  
SERVICE NÃO TEM ABA.  
É UM SISTEMA MEU QUE  
ACESSA O SEU SERVIÇO...



HMMM... ENTENDO... MAS QUAL  
A PÁGINA DO WEB SERVICE QUE  
O SENHOR ESTÁ NAVEGANDO  
E ENCONTRANDO O  
ERRO???

CRASH!



### API - *Application Programming Interface*

API é um termo bastante amplo, que não necessariamente descreve um web service. API é uma interface que permite a comunicação entre dois componentes de software. Por meio de uma API, um software cliente é capaz de consumir serviços disponibilizados por um software servidor.

### Web Service

Um web service, por sua vez, é um tipo de API que fornece a sua interface de comunicação **via internet**.

**Nem toda API é um web service, mas todos os web services são APIs.**

Web Service

# SOAP

*Simple Object Access Protocol*

### Definição

- Protocolo de comunicação usado para troca de mensagens entre aplicações.
- As mensagens SOAP basicamente são **documentos XML** serializados seguindo o padrão W3C enviados em cima de um protocolo de rede como HTTP.
- Para descrever os serviços SOAP, é comum utilizar o WSDL (*Web Services Description Language*), um documento XML que define a interface, operações, e protocolos de comunicação.

### Estrutura

- *Envelope* - Define o início e o fim da mensagem. É o elemento raiz.
- *Header* - Define informações adicionais sobre a mensagem. Opcional
- *Body* - Define o conteúdo da mensagem. Obrigatório.
- *Fault* - Define informações sobre erros. Opcional



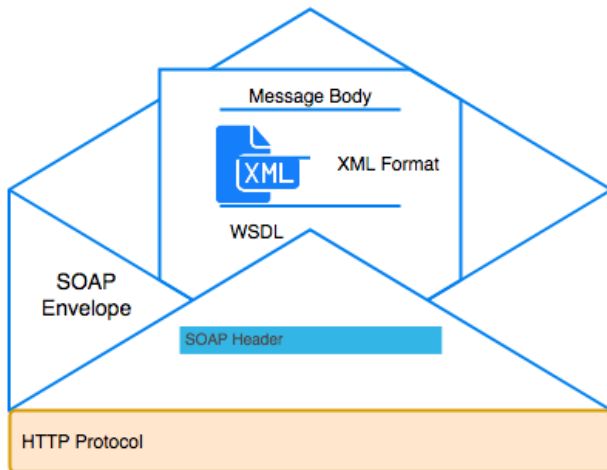


Figura: Estrutura SOAP

Web Service

# SOAP - Exemplo

Requisição e Resposta

```

1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
2    <soapenv:Header/>
3    <soapenv:Body>
4      <sch:UserDetailsRequest>
5        <sch:name>John</sch:name>
6      </sch:UserDetailsRequest>
7    </soapenv:Body>
8  </soapenv:Envelope>

```

```

1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
2    <soapenv:Header/>
3    <soapenv:Body>
4      <ns2:UserDetailsResponse xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/">
5        <ns2:User>
6          <ns2:name>John</ns2:name>
7          <ns2:age>5</ns2:age>
8          <ns2:address>Greenville</ns2:address>
9        </ns2:User>
10     </ns2:UserDetailsResponse>
11   </soapenv:Body>
12 </soapenv:Envelope>

```

Figura: SOAP - Exemplo - Requisição e Resposta

Web Service

# SOAP - Exemplo

Olá Mundo usando protocolo SOAP e Python

```
6 class HelloWorldService(ServiceBase):
7
8     # O decorator @rpc define que o método say_hello é um método remoto
9     @rpc(Unicode, Integer, _returns=Unicode)
10     def say_hello(ctx, name, times):
11         ip_address = ctx.transport.req["REMOTE_ADDR"]
12
13         for i in range(times):
14             print(f"Hello {name} from {ip_address} #{i+1}")
15
16         return f"Hello {name} from {ip_address}!"
17
18 # Criando uma aplicação Spyne com o serviço HelloWorldService
19 soap_app = Application([HelloWorldService], 'spyne.examples.hello.soap',
20                         in_protocol=Soap11(validator='lxml'),
21                         out_protocol=Soap12())
22
23 # Criando um aplicativo WSGI a partir da aplicação SOAP
24 # WSGI: Web Server Gateway Interface é uma especificação padrão para a
25 # interface entre servidores web e aplicações web em Python
26 wsgi_app = WsgiApplication(soap_app)
27
28 if __name__ == '__main__':
29     server = make_server('0.0.0.0', 8000, wsgi_app)
30     server.serve_forever()
```

Figura: SOAP - Servidor

exemplos > soap >  client\_soap.py > ...

```

1  from zeep import Client
2  from zeep.plugins import HistoryPlugin
3  from lxml import etree
4
5  # Criar um cliente Zeep com base no URL do WSDL
6  history = HistoryPlugin()
7  client = Client(f'http://localhost:8000/?wsdl', plugins=[history])
8
9  # Chamar o método do serviço
10 response = client.service.say_hello(name='Professor', times=3)
11
12 # Exibir a resposta
13 for hist in [history.last_sent, history.last_received]:
14     print(etree.tostring(hist["envelope"], encoding="unicode", pretty_print=True))

```

Figura: SOAP - Cliente

```
(.venv) fabricio@DESKTOP-MG3SLC3:~/Projetos/Desenvolvimento-Web-II/exemplos/soap$ python server.py
127.0.0.1 - - [12/Jan/2024 12:39:01] "GET /?wsdl HTTP/1.1" 200 2613
Hello Professor from 127.0.0.1
Hello Professor from 127.0.0.1
Hello Professor from 127.0.0.1
127.0.0.1 - - [12/Jan/2024 12:39:01] "POST / HTTP/1.1" 200 235
[]
(.venv) fabricio@DESKTOP-MG3SLC3:~/Projetos/Desenvolvimento-Web-II/exemplos/soap$ python client.py
None
```

```
<wsdl:definitions
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:plink="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:wsdlsoap11="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdlsoap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap11enc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soap11env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soap12env="http://www.w3.org/2003/05/soap-envelope"
  xmlns:soap12enc="http://www.w3.org/2003/05/soap-encoding"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
  xmlns:xop="http://www.w3.org/2004/08/xop/include"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:tns="spyne.examples.hello.soap" targetNamespace="spyne.examples.hello.soap" name="Application">
  <wsdl:types>
    <xs:schema targetNamespace="spyne.examples.hello.soap" elementFormDefault="qualified">
      <xs:complexType name="say hello">
        <xs:sequence>
          <xs:element name="name" type="xs:string" minOccurs="0" nillable="true"/>
          <xs:element name="times" type="xs:integer" minOccurs="0" nillable="true"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="say_helloResponse">
        <xs:sequence>
          <xs:element name="say_helloResult" type="xs:string" minOccurs="0" nillable="true"/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name="say_hello" type="tns:say_hello"/>
      <xs:element name="say_helloResponse" type="tns:say_helloResponse"/>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="say_hello">
    <wsdl:part name="say_hello" element="tns:say_hello"/>
  </wsdl:message>
```

localhost:8000/?wsdl

Figura: SOAP - Chamada e WSDL

## Web Service

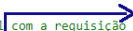
# SOAP - Exemplo com Chamada Direta

Podemos enviar o arquivo XML diretamente para o servidor



exemplos > soap >  client\_soap\_xml.py > ...

```
1 from zeep import Client
2 from zeep.plugins import HistoryPlugin
3 from lxml import etree
4 import http.client
5
6 # ler o arquivo xml com a requisição
7 with open("request.xml", "r") as f:
8     xml_content = f.read()
9
10 # Criar um cliente Zeep com base no XML
11 connection = http.client.HTTPConnection("localhost", 8000)
12 connection.request("POST", "/", xml_content, headers={"Content-Type": "text/xml"})
13
14 # Exibir a resposta
15 response = connection.getresponse()
16 print(response.status, response.reason)
17 print(response.read().decode())
18
19 # Fechar a conexão
20 connection.close()
```



```
<soap-env:Envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/en
  <soap-env:Body>
    <ns0:say_hello xmlns:ns0="spyne.examples.hello.s
      <ns0:name>Professor</ns0:name>
      <ns0:times>10000</ns0:times>
    </ns0:say_hello>
  </soap-env:Body>
</soap-env:Envelope>
```

Figura: SOAP - Chamada Direta - Cliente

# SOAP

## Enviando XML para o Servidor via Postman

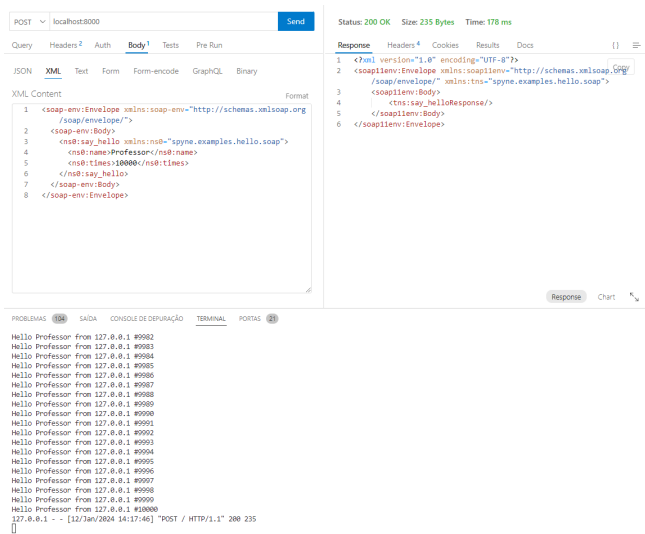


Figura: SOAP - Enviando XML

## Experimento 1

- Consoma o Web Service SOAP disponível em `http://www.dneonline.com/calculator.asmx?WSDL`.
- Crie um cliente para testar o Web Service.

## Experimento 2

- Crie um Web Service SOAP que receba um número inteiro e retorne o dobro do número.
- Crie um cliente para testar o Web Service ou use o Postman/Insomnia.
- Peça para outro colega testar seu Web Service.

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.



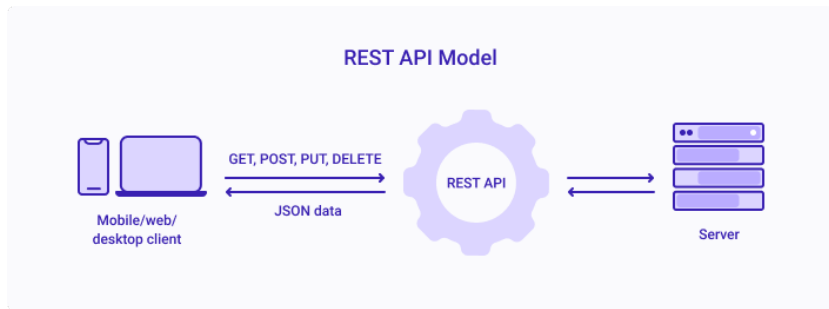


Figura: Estrutura REST

# 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 <sup>1</sup>	HTTPS

Tabela: SOAP vs REST

---

<sup>1</sup>Conhecendo 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.

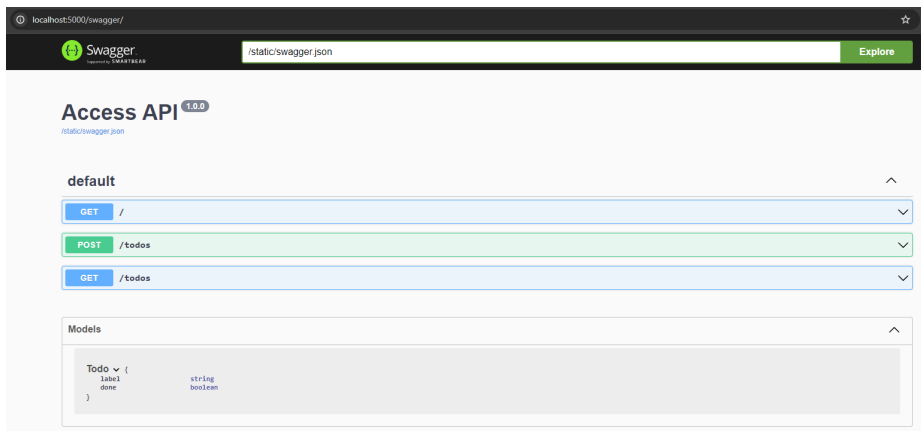


Figura: Swagger para documentar a API REST.

## 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

- |                                    |   |
|------------------------------------|---|
| ■ <b>200</b> - OK                  | ■ <b>404</b> - Não encontrado           |
| ■ <b>201</b> - Criado              | ■ <b>500</b> - Erro interno do servidor |
| ■ <b>400</b> - Requisição inválida | ■ <b>501</b> - Não implementado         |
| ■ <b>401</b> - Não autorizado      | ■ <b>503</b> - 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 shows a REST client interface with two requests. The first request is a POST to localhost:5000/login with a JSON body containing 'username': 'admin' and 'password': 'errada'. It returns a 401 UNAUTHORIZED status with a response body of { "message": "Invalid credentials" }. The second request is a POST to localhost:5000/login with a JSON body containing 'username': 'admin' and 'password': 'admin'. It returns a 200 OK status with a response body of { "token": "xEqxyz0HnVi7btzYkLgVJw" }.

**Request 1:**

POST localhost:5000/login

Query Headers<sup>2</sup> Auth<sup>1</sup> **Body<sup>1</sup>**

JSON XML Text Form Form-enc

JSON Content Format

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

**Response:**

Status: 401 UNAUTHORIZED Size: 39 Bytes

Response Headers<sup>5</sup> Cookies Results

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

**Request 2:**

POST localhost:5000/login

Query Headers<sup>2</sup> Auth<sup>1</sup> **Body<sup>1</sup>**

JSON XML Text Form Form-enc

JSON Content Format

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

**Response:**

Status: 200 OK Size: 40 Bytes Time: 4 ms

Response Headers<sup>5</sup> Cookies Results

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

Figura: Bearer Token - Login

# REST

## Bearer Token - Exemplo Prático

The image shows a REST client interface with two requests and their corresponding responses.

**Request 1:**

- Method: GET
- URL: localhost:5000/products
- Auth: Bearer
- Token: token\_invalido
- Status: 401 UNAUTHORIZED
- Size: 32
- Response Body: 

```
{
  "message": "Unauthorized"
}
```

**Request 2:**

- Method: GET
- URL: localhost:5000/products
- Auth: Bearer
- Token: WVIKbxQsFGMvAUPzwGLlIg
- Status: 200 OK
- Size: 95 Bytes
- Time: 7 ms
- Response Body: 

```
[
  {
    "id": 1,
    "name": "Product A"
  },
  {
    "id": 2,
    "name": "Product B"
  }
]
```

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.

The screenshot shows the Postman interface with a POST request to `http://localhost:5000/login`. The "Body" tab is selected, displaying JSON content:

```
{  
  "username": "admin",  
  "password": "admin"  
}
```

The status bar at the top indicates a successful response: Status: 200 OK, Size: 356 Bytes, Time: 5 ms.

The "Response" tab is also visible, showing the following JSON output:

```
{  
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cGU6IjYmcmVzaCI6ZmFsc2UsIm1hdCI6MnN1YiI6ImFkbWluIiwibmJmIjozeA.YhJ5HFV_i9im5EdV234Kyz6NPSoS..."  
}
```

Figura: JWT - Exemplo (parte 1)

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

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

- Method:** GET
- URL:** http://localhost:5000/protected
- Status:** 200 OK
- Size:** 30 Bytes
- Auth:** Bearer (selected)
- Response:**

```
1 {
2   "logged_in_as": "admin"
3 }
```
- Token:**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImh0bWw6MTcwNTYwNDkwOCwianRpIjoib2M0NjZkZTMtNzI3MC00NjI0LTkyNGQzMjZiMzAxYTE1NzIhliwidHlwZSI6ImFjY2VzcyIsbnN1YiI6ImFkbWw6MTJlMjZDhiMWMtYTk1ZS00MmYzLWI1NWltYzc4YWFiZTlwNWU1IiwiaXNjaA0OTY4fQ.YhJ5HFV_i9im5EdV234Kyz6NPSOSbc67k-sGlyBoliQ
```

Figura: JWT - Exemplo (parte 2)

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

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

- Request:**
  - Method: GET
  - URL: `http://localhost:5000/protected`
  - Auth: Bearer
  - Body: OAuth 2
- Response:**
  - Status: 401 UNAUTHORIZED
  - Size: 33
  - Body: 

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

Below the request tab, the Bearer Token is displayed:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImhhdCI6MTcwNTYwNDkwOCwianRpIjoib2M0NjZkZTMtNzI3MC00NjI0LTkyNGQtMjZiMzAxYTE1NzlhliwidHlwZSI6ImFjY2VzcyIsbnN1Yil6ImFkbWluciwibmJmljoxNzA1NjA0OTA4LCJjc3ImljoiY2E1ZDhiMWMtYTk1ZS00MmYzLWl1NWltYz4YWFZlbnNwE1liwiZXhwIjoxNzA1NjA0OTY4fQ.YhJ5HFV_i9im5EdV234Kyz6NPSOSbc67k-sGlyBolIQ
```

Figura: JWT - Exemplo (parte 3)

Validando o token JWT na página [jwt.io](https://jwt.io).

Encoded

[illegible]

- 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

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

**PAYLOAD:** DATA

```
{
  "fresh": false,
  "iat": 1785604908,
  "jti": "7c466de3-7270-4624-924d-26b301a1579a",
  "typ": "access",
  "sub": "admin",
  "nbf": 1785604908,
  "csrf": "ca5d8b1c-a95e-42f3-b55b-c78aabe205a5",
  "exp": 1785604968
}
```

VERIFY SIGNATURE

```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    minha_chave_secreta_aq
) # secret base64 encoded
```

## Chave Secreta

Signature Verified

[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.


### 11. Segurança - Autenticação com OAuth

O OAuth é um protocolo ou estrutura de autorização de padrão aberto que fornece aos aplicativos a capacidade de “acesso designado seguro”. Você pode, por exemplo, dizer ao Facebook que a ESPN.com pode acessar seu perfil ou postar atualizações em sua linha do tempo sem precisar fornecer à ESPN sua senha do Facebook. Isso minimiza o risco de forma importante: caso a ESPN sofra uma violação, sua senha do Facebook permanece segura.

#### OAuth - Fluxo de Autorização

Google, Facebook, Twitter, GitHub, etc.

- **Passo 1** - O cliente solicita autorização do usuário.
- **Passo 2** - O usuário autoriza o cliente.
- **Passo 3** - O cliente recebe um código de autorização.
- **Passo 4** - O cliente troca o código de autorização por um token de acesso.
- **Passo 5** - O cliente usa o token de acesso para acessar o recurso protegido.

 OAuth - Simulação

## 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
```

### 14. Testes Automatizados

Crie testes automatizados para garantir a estabilidade da API e detectar rapidamente problemas de integração ou regressão.

#### Exemplo

```
# Integration Test
def test_get_all_products(self):
    response = self.client.get("http://localhost:5000/api/v1/produtos")
    self.assertEqual(response.status_code, 200)
```

## 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.

### 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.
- Use uma ferramenta como o Insomnia para testar a API.
- Navegue pela documentação da API e teste os endpoints.

### Experimento 2

Consumir a API REST The Movies DB.

- Utilizar uma biblioteca/framework de sua escolha para o desenvolvimento frontend (por exemplo, React, Vue, Angular, etc.).
- Fazer solicitações HTTP para a API do TMDb (<https://www.themoviedb.org/documentation/api>) para obter informações sobre os filmes.

Web Service

# GraphQL

*Definição*

### GraphQL - *Graph Query Language*

- Linguagem de consulta para APIs.
- Foi criada pelo Facebook em 2012 e tornou-se open-source em 2015.
- É uma alternativa ao REST. Permite que os clientes solicitem dados de forma flexível.

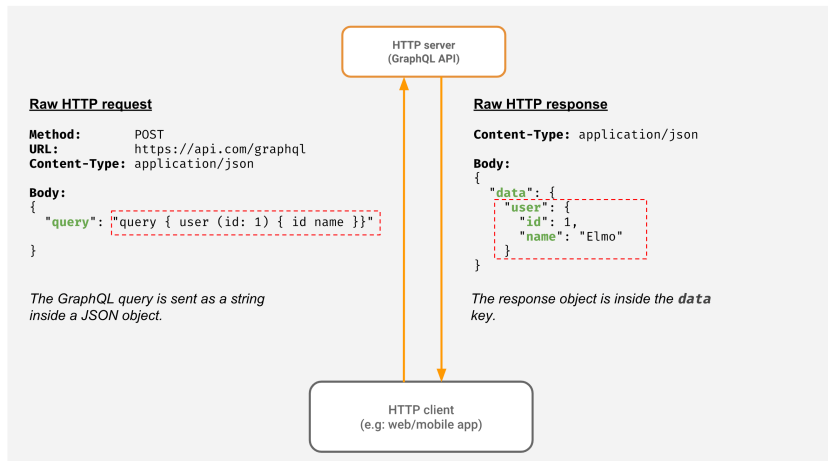


Figura: Estrutura GraphQL

REST	GraphQL
É uma arquitetura	É uma linguagem de consulta
Os dados são expostos como recursos	Os dados são expostos como um grafo
Overfetching ou underfetching são comuns.	Os clientes solicitam apenas os dados necessários, evitando overfetching e underfetching.
Múltiplos endpoints para diferentes recursos.	Um único ponto de entrada para todas as operações.
Operações de escrita (POST, PUT, DELETE) têm endpoints separados.	Mutações são tratadas no mesmo sistema de tipos e no mesmo endpoint que as consultas.
Utiliza diversos métodos HTTP (GET, POST, PUT, DELETE).	Usa o método POST para todas as operações. Pode ser usado com qualquer protocolo de transporte.
Pode exigir versionamento da API para adicionar ou modificar funcionalidades.	Não requer versionamento devido à flexibilidade nas consultas.

Tabela: REST vs GraphQL

### Mutation

- É um tipo de operação que permite criar, atualizar ou excluir dados.
- É semelhante a uma operação de escrita no REST.
- As mutações são tratadas no mesmo sistema de tipos e no mesmo endpoint que as consultas.

```
1 mutation {  
2   createTask(  
3     title: "Estudar",  
4     description: "Estudar Web Service") {  
5     task {  
6       id  
7       title  
8       description  
9     }  
10  }  
11 }
```

```
{  
  "data": {  
    "createTask": {  
      "task": {  
        "id": "3",  
        "title": "Estudar",  
        "description": "Estudar Web Service"  
      }  
    }  
  }  
}
```

Figura: Exemplo de Mutation



### Query

- É um tipo de operação que permite recuperar dados.
- É semelhante a uma operação de leitura no REST.
- As consultas são tratadas no mesmo sistema de tipos e no mesmo endpoint que as mutações.

```
1 query {  
2   tasks {  
3     title  
4   }  
5 }
```

```
{  
  "data": {  
    "tasks": [  
      {  
        "title": "Estudar"  
      }  
    ]  
  }  
}
```

Figura: Exemplo de Query