

# Desenvolvimento de APIs REST

## 01 - Introdução

- REST
- HTTP
- Servidores Web
- Spring vs JEE

# APIs Rest - Definição

## **API - Application Program Interface (interface de programação de aplicações)**

- Conjunto de definições e protocolos usado no desenvolvimento e na integração de aplicações, podendo ser descrita como um contrato
- Estabelece conteúdo exigido pelo consumidor (a chamada) e o conteúdo exigido pelo produtor (a resposta)

Exemplo: um serviço meteorológico espera um CEP e o produz como resposta em duas partes: a primeira contendo a temperatura mais elevada e a segunda com a temperatura mais baixa.

(Semelhante a assinatura de um método ou conjunto de métodos em java)

## **REST - Representational State Transfer (Transferência Representacional de Estado)**

- Estilo de arquitetura de software que define um conjunto de restrições a serem usadas para a criação de web services (serviços Web), também chamados de Web services RESTful
- Fornecem interoperabilidade entre sistemas de computadores na Internet.
- Permitem que os sistemas solicitantes acessem e manipulem representações textuais de recursos da Web

# Introdução - Protocolo HTTP Rest



O Hypertext Transfer Protocol, sigla HTTP (em português **Protocolo de Transferência de Hipertexto**) é um protocolo de comunicação (...) utilizado para sistemas de informação de hipermídia distribuídos e colaborativos, sendo a base para a comunicação de dados da World Wide Web.

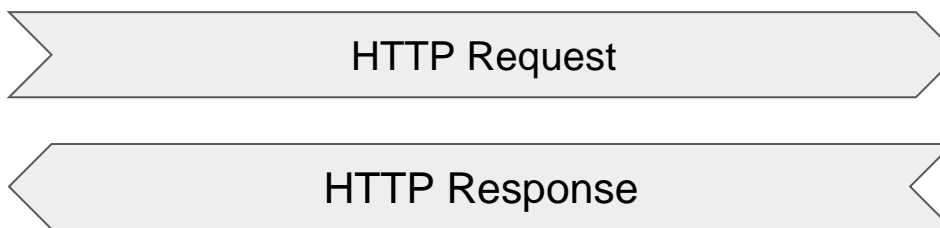
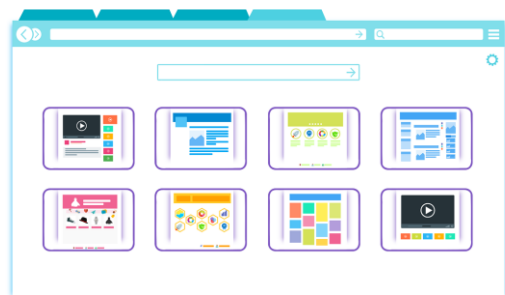
Wikipedia

- É um protocolo que permite a obtenção de recursos, como documentos HTML, imagens, arquivos css, arquivos js, etc.
- É a base de qualquer troca de dados na Web
- É um protocolo cliente-servidor, o que significa que as requisições são iniciadas pelo destinatário, geralmente um navegador da Web.

# Introdução - Protocolo HTTP Rest



Clientes e servidores se comunicam trocando mensagens individuais (ao contrário de um fluxo de dados). As mensagens enviadas pelo cliente, geralmente um navegador da Web, são chamadas de solicitações (requests), ou também requisições, e as mensagens enviadas pelo servidor como resposta são chamadas de respostas (responses).



# Introdução - Protocolo HTTP Rest



Mensagens HTTP são feitas a partir de um header e um body:

- O body contém informações que desejamos transmitir, de acordo com as instruções presentes no header.
- O header guarda as informações metadata, no caso de requisições, também pode conter informações, como métodos HTTP, content-type e outros.

```
GET /index.html HTTP/1.1
Host: www.exemplo.com
```

HTTP Request

HTTP Response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2020 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2020 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
<!DOCTYPE html >
<html>
<head>
  <title>Titulo da página</title>
  <meta charset="utf-8"/>
</head>
<body>
  <h1>Curso de java backend</h1>
...

```

# Introdução - Protocolo HTTP Rest



1. Browser abre uma conexão com o servidor (semelhante a abrir arquivo)
2. Browser envia uma requisição ao servidor (semelhante a gravar dados no arquivo):
  - a. Método utilizado (GET - recuperar informação)
  - b. Qual informação desejada (/index.html) e qual a versão do protocolo (HTTP/1.1)
  - c. Endereço do servidor "host" (www.exemplo.com)
3. Servidor responde a requisição (semelhante a ler dados do arquivo):
  - a. Versão do protocolo e código de status (200 ok)
  - b. Data da resposta
  - c. Servidor web
  - d. Data de última atualização
  - e. Tamanho em bytes (Content-length)
  - f. Tipo de conteúdo (imagem, texto, vídeo, pdf, etc.)
  - g. Conteúdo propriamente dito (html da página, por exemplo)
4. Browser fecha a conexão (semelhante a fechar arquivo)

```
GET /index.html HTTP/1.1
Host: www.exemplo.com
```

HTTP Request

HTTP Response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2020 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2020 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
<!DOCTYPE html>
<html>
<head>
  <title>Titulo da página</title>
  <meta charset="utf-8"/>
</head>
<body>
  <h1>Curso de APIs</h1>
...

```

# REST

**Representational State Transfer (REST)**, que traduzindo para o português significa **Transferência Representacional de Estado**. Trata-se de um estilo de arquitetura de software que define um conjunto de restrições a serem usadas para a criação de **web services (serviços Web)**. Os Web services que estão em conformidade com o estilo arquitetural REST, denominados Web services RESTful, fornecem interoperabilidade entre sistemas de computadores na Internet. Os Web services RESTful **permitem que os sistemas solicitantes acessem e manipulem representações textuais de recursos da Web** usando um conjunto uniforme e predefinido de operações sem estado.

Wikipedia

# Introdução - REST



- Maneira simples de realizar interações entre sistemas
- “Resource-based”: baseado no conceito de recursos (coisas e não ações, como produto, pessoa, email, pedido)
- Utiliza o protocolo HTTP e seus *verbos* para as operações
- Recursos são identificados por URIs (Uniform Resource Identifier)
- Podem ser representados em *JSON* ou *XML*
- Podem ser armazenados no cache

**REST:** conjunto de princípios de arquitetura

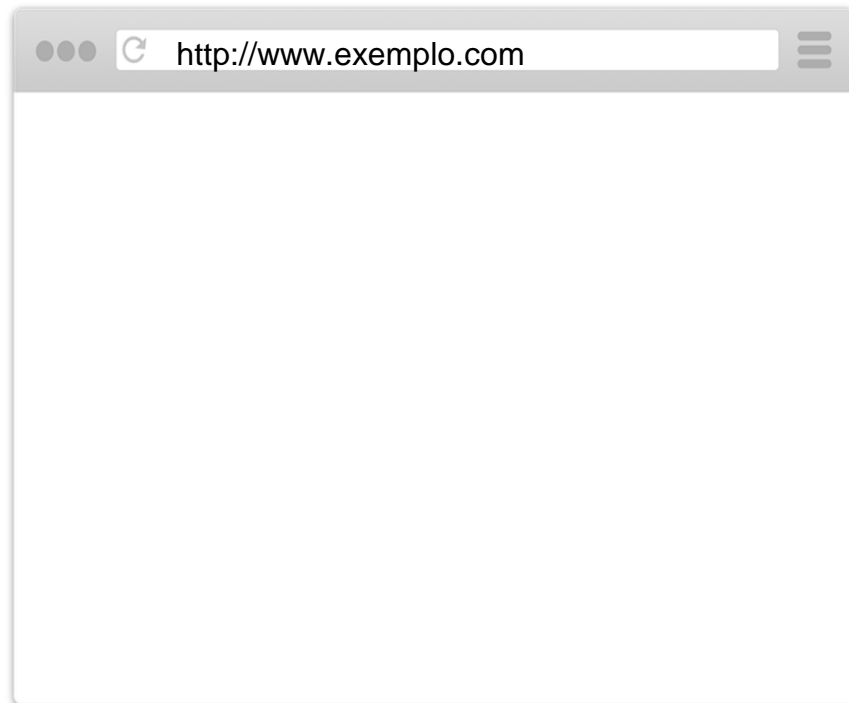
**RESTful:** capacidade de determinado sistema aplicar os princípios de REST



# REST: Exemplo Simples



Acessar no browser um endereço web



HTTP Request

```
GET /index.html HTTP/1.1  
Host: www.exemplo.com
```



# REST: Exemplo Simples



Servidor web retorna o conteúdo da página.  
Browser interpreta o conteúdo.



## HTTP Response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2020 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2020 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
<!DOCTYPE html>
<html>
<head> <title>Rest exemplo</title> </head>
<body>
  <h1>Lista de e-mail</h1>
  <table border=1>
    <tr><td>Nome</td><td>E-mail</td></tr>
    <tr><td></td><td></td></tr>
  </table>
</body>
<script>
  function carregaDados() {
    .....
  }
</script>
</html>
```



# REST: Exemplo Simples



Browser solicita dados da página ao servidor



HTTP Request

```
GET /emails HTTP/1.1  
Host: www.exemplo.com
```



# REST: Exemplo Simples



Informações são “mostradas” na página



## HTTP Response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2020 22:38:34 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2020 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/json; charset=UTF-8
[
  {
    "nome": "antonino",
    "email": "antonio@mail.com"
  },
  {
    "nome": "joaquim",
    "email": "joaquim@mail.com"
  },
  {
    "nome": "maria",
    "email": "maria@mail.com"
  }
]
```



# Introdução - REST



REST não é uma biblioteca ou framework. É simplesmente um modelo utilizado para projetar a arquitetura de softwares distribuídos que fazem a comunicação pela rede.

Nele, o cliente (front-end) faz uma requisição para o servidor (back-end) para enviar, modificar ou consultar dados.

Um requisição consiste em:

- Um **verbo** ou método tipo de operação o back-end vai realizar: **POST**
- Um **header**, ou seja, que passa informações sobre a **Content-Type**
- Um **path** (caminho ou rota) para o back-end: **/cliente**
- informação no **body**, ou seja, { "nome" .... }

Exemplo:



```
POST /cliente HTTP/1.1
Host: www.exemplo.com
Content-Length: 68
Content-Type: application/json

{
  "nome": "José da Silva",
  "endereco": {
    "logradouro": "rua dos bobos",
    "numero": "0"
  },
  "estadoCivil": "complicado"
}
```

# Introdução - REST



## Ausência de Estado

A comunicação não deve ter um contexto de cliente armazenado no servidor. Isso quer dizer que, cada solicitação ao servidor deve ser feita com todos os dados solicitados e que não se deve presumir que o servidor possua dados de solicitações anteriores.

Por exemplo, o cliente pode enviar no cabeçalho cookies ou tokens de autenticação para que o servidor identifique o usuário que está realizando a requisição:

```
POST /cliente HTTP/1.1
Host: www.exemplo.com
Content-Length: 68
Content-Type: application/json
Authorization: Basic NlhvWU1EUjIybWhrMWJlcV9kcVczMVR4NlhzYTpQSWk5YmdwVzJWczhUTkdLcFowTzZ4cHBGOEFh

{
  "nome": "José da Silva",
  "endereco": {
    "logradouro": "rua dos bobos",
    "numero": "0":
  },
  "estadoCivil": "complicado"
}
```

# Introdução - REST: Identificação de Recursos



**Recurso:** É um conteúdo disponibilizado na web, pode ser um arquivo, texto, imagem página e até mesmo um serviço disponibilizado pela API. Todo recurso é identificado através de uma **URI**.

## URI(Uniform Resource Identifier)

É um identificador único de um recurso. Ele é composto pela URL e pela URN.

- <http://www.serratec.org/index.html>
- <http://serratec.org/desenvolvedor-backend>

## URL(Uniform Resource Locator)

É um endereço que estamos acessando um recurso.

Exemplos de URL:

- <http://www.serratec.org/index.html>
- <http://serratec.org/desenvolvedor-backend>

## URN(Uniform Resource Name)

É o nome do recurso que será acessado.

- <http://www.serratec.org/index.html>
- <http://serratec.org/desenvolvedor-backend>

<a href="http://www.serratec.org/index.html">http://www.serratec.org/index.html</a>	
URL	URN
URI	

Resumindo a URI é um identificador geral, em que um URL especifica um local e um URN especifica um nome

# Introdução - REST: Identificação de Recursos



## Exemplo de Identificação de Recursos

Ao definir uma URI devemos utilizar nomes que sejam fáceis de entender e que tenham relação com a aplicação. Vejamos o exemplo nas URIs apresentadas abaixo de um sistema acadêmico:

### **Identificar todos os alunos**

`http://universidade.com.br/alunos`

### **Identificar aluno com matricula 100**

`http://universidade.com.br/alunos/100`

**Não é necessário incluir o nome que identifica a informação do recurso na requisição/request, como excluir, alterar e etc.**

`http://universidade.com.br/alunos/cadastrar`

`http://universidade.com.br/alunos/excluir`

Mas como especificamos uma ação? Como podemos saber se vamos criar um novo aluno ou se queremos listar os alunos? Para isso existem os verbos HTTP.



# Introdução - REST: curl



O comando curl é uma abreviação de **client URL** derivado da família UNIX é uma forma de verificar informações da url e também uma ferramenta de transferência de dados.

## Verificar o header

```
curl -I http://www.serratec.org  
curl -I https://www.google.com
```

## Informações detalhadas

```
curl -v https://www.google.com  
curl -v http://www.serratec.org
```

## Fazer download de arquivos

```
curl -O http://serratec.org/wp-content/uploads/2018/01/logo-neq.png
```

- Para quem usa Windows e não tem o curl instalado, é possível usá-lo pelo site <https://reqbin.com/curl> :

```
bulinha@bula-idk-notebook:~$ curl -I https://www.google.com  
HTTP/2 200  
content-type: text/html; charset=ISO-8859-1  
p3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."  
date: Wed, 31 Aug 2022 17:40:24 GMT  
server: gws  
x-xss-protection: 0  
x-frame-options: SAMEORIGIN  
expires: Wed, 31 Aug 2022 17:40:24 GMT  
cache-control: private  
...
```

# REST: Verbos HTTP



Para cada requisição criada temos um verbo associado no início do cabeçalho:

**GET / HTTP/1.1**

## **GET**

Serve para obtermos dados de um recurso. As informações passadas via GET não são alteradas no servidor.

[/alunos](#)

[/alunos/id](#)

## **POST**

É utilizado quando queremos inserir uma nova informação no servidor.

[/alunos](#)

## **DELETE**

É utilizado quando queremos remover um recurso. Utilizamos um parâmetro para remoção.

[/alunos/id](#)

## **PUT**

É utilizado quando queremos atualizar um recurso. Utilizamos um parâmetro para atualização.

[/alunos/id](#)

# REST: Verbos HTTP



## PATCH

Usado para editar o recurso sem a necessidade de enviar todos os atributos. E enviado apenas o que foi alterado.

[/alunos/id](#)

## OPTIONS

Usado para saber quais recursos estão disponíveis em um servidor.

```
bulinha@bula-idk-notebook:~$ curl -X OPTIONS http://www.google.com -I
HTTP/1.1 405 Method Not Allowed
Allow: GET, HEAD
Date: Wed, 31 Aug 2022 17:34:39 GMT
Content-Type: text/html; charset=UTF-8
Server: gws
Content-Length: 1592
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
```

## HEAD

O método HEAD é muito semelhante ao GET, a diferença é que retorna somente o cabeçalho da requisição.

```
bulinha@bula-idk-notebook:~$ curl -X HEAD http://www.google.com -I
HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Date: Wed, 31 Aug 2022 17:36:20 GMT
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Transfer-Encoding: chunked
Expires: Wed, 31 Aug 2022 17:36:20 GMT
Cache-Control: private
...
```

# REST: Verbos HTTP



## Classificação dos Métodos:

### Métodos Idempotentes

Um método é idempotente quando retorna o mesmo resultado independente do número de vezes que é chamado.

**GET, HEAD, PUT, DELETE** são idempotentes.

**POST** não é idempotente. A função do POST é criar um recurso, então a cada requisição realizada, um recurso será criado e assim alterando o estado da aplicação e não mantendo o atual.

### Métodos Seguros

Os métodos seguros são aqueles que não alteram o estado da aplicação. O único método seguro é o GET, os outros modificam os recursos.

# REST: Verbos HTTP -> SQL -> CRUD



As mais importantes e utilizadas são: **POST, GET, PUT e DELETE**. Frequentemente, estas operações são combinadas com operações C.R.U.D. para a persistência de dados.

Operação CRUD	Método Rest	Comando SQL
Create	POST	INSERT
Recovery	GET	SELECT
Update	PUT	UPDATE
Delete	DELETE	DELETE

# REST: JSON / XML?



## Representação de Recursos

Os dois padrões mais utilizados são JSON e XML.

### JSON

- Fácil e simples de usar
- Consome menos memória, menor banda para transferência de dados
- Não é necessário criar “Mapeamento”, Jackson resolve tudo ;-)
- No browser/javascript, não há a necessidade de biblioteca extra para converter os dados
  - string = JSON.stringify(objeto)
  - Objeto = JSON.parse(string)
- Muito comum em serviços web, aplicativos e sites de dados

```
{
  "nome": "Carla",
  "email": "carla@gmail.com",
  "telefone": "2245-6789",
}
```

### XML

- Tags não são pré-definidas (como em html), é possível criar tags customizadas.
- Criado principalmente para transportar dados
- Código de marcação é fácil de ser entendido e legível por seres humanos
- Formato bem estruturado, fácil de por seres humanos.
- Possibilidade de se ter validadores de formatos
- Muito comum em aplicações/ferramentas corporativas
- Utilizado em arquivos de configuração e documentos (Office)

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <nome>Carla</nome>
  <email>carla@gmail.com</email>
  <telefone>2245-6789</telefone>
</root>
```

# REST: Status



Toda requisição, em sua resposta, retorna um código de status informa o estado do recurso requisitado. Na tabela abaixo temos as categorias dos códigos de status

```
curl -X HEAD http://www.google.com -I
HTTP/1.1 200 OK
....
```

Categoria	Descrição
1xx: Informacional	Comunica informações no nível do protocolo de transferência. <ul style="list-style-type: none"><li>• 101 - indica mudança de protocolo (ex: de http para websocket)</li></ul>
2xx: Sucesso	Indica que a requisição foi completada: <ul style="list-style-type: none"><li>• 200 - ok</li><li>• 201 - criado</li></ul>
3xx: Redirecionar	Indica que o cliente deve realizar alguma operação adicional. <ul style="list-style-type: none"><li>• 301 - indica que o recurso mudou de localização (indicando a nova no cabeçalho)</li></ul>
4xx: Erro do Cliente	Alguma informação fornecida pelo cliente é inválida <ul style="list-style-type: none"><li>• 400 - Bad Request (dados inválidos ou incompletos)</li><li>• 401 - não autorizado</li><li>• 403 - proibido</li><li>• 404 - não encontrado</li><li>• 405 - método não permitido (POST e PUT proibidos)</li></ul>
5xx: Erro no Servidor	Erro de responsabilidade do servidor <ul style="list-style-type: none"><li>• 500 - Erro interno (ex.: exception não tratada no java)</li></ul>

# REST: Exemplo Acesso API



## Acesso a APIs rest

Uma página web pode acessar um serviço rest utilizando o componente XMLHttpRequest do javascript:

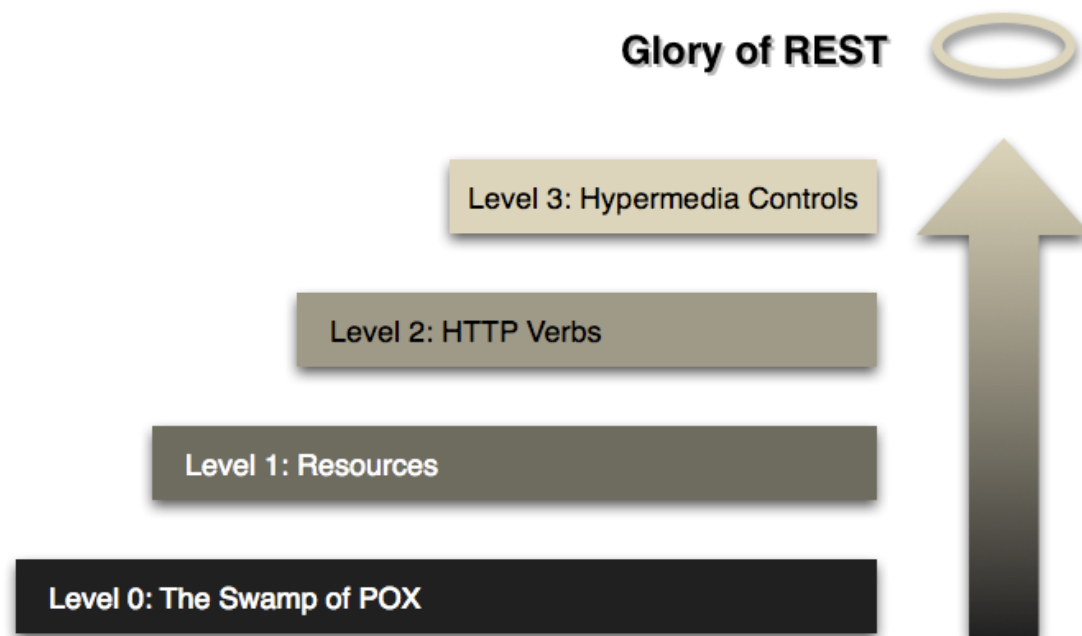
```
var url = 'https://viacep.com.br/ws/25720322/json/';
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        var json = JSON.parse(xmlhttp.responseText);
        console.log(json);
    }
}
xmlhttp.open("GET", url, true);
xmlhttp.send();
```

```
> var url = 'https://viacep.com.br/ws/25720322/json/';
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        var json = JSON.parse(xmlhttp.responseText);
        console.log(json);
    }
}
xmlhttp.open("GET", url, true);
xmlhttp.send();
< undefined

{cep: '25720-322', logradouro: 'Rua Vigário Correa', complemen
'Petrópolis', ...}
  bairro: "Corrêas"
  cep: "25720-322"
  complemento: ""
  ddd: "24"
  gia: ""
  ibge: "3303906"
  localidade: "Petrópolis"
  logradouro: "Rua Vigário Correa"
  siafi: "5877"
  uf: "RJ"
  [[Prototype]]: Object
```



# REST: Modelo de Maturidade



O modelo de maturidade rest foi desenvolvido por Leonard Richardson.

Fonte: <https://martinfowler.com/articles/richardsonMaturityModel.html>

Ele foi desenvolvido para padronizar e facilitar a implementação de APIs REST, sendo dividido em quatro níveis.

# REST: Modelo de Maturidade



## Nível 0 – POX (Plain Old XML)

Este é o nível básico, geralmente uma API que implementa este nível não é considerada REST. Os nomes dos recursos não seguem qualquer padrão. Nesse nível usamos o protocolo HTTP para comunicação, mas sem regras para implementação dos métodos.

Verbo	URI	Ação
GET	/buscarAluno/2	Pesquisar
POST	/salvarAluno	Cadastrar
POST	/alterarAluno/3	Modificar
GET/POST	/excluirAluno/2	Excluir

Neste nível temos outro problema que é a manipulação incorreta de códigos HTTP. No exemplo abaixo foi retornado o código **200 OK** mesmo o aluno não sendo encontrado. A resposta HTTP apresenta uma informação totalmente diferente com o código 404

```
GET /buscarAluno/2
HTTP/1.1 200 OK
<buscarAluno>
    <status>Aluno não encontrado</status> <codigo>404</codigo>
</buscarAluno>
```

# REST: Modelo de Maturidade



## Nível 1 – Recursos

Neste nível passamos a utilizar os recursos para organizar e modelar a API. Cada recurso utiliza substantivos no plural. Neste nível os verbos passam a ser utilizados de forma correta.

Verbo	URI	Ação
GET	/alunos/2	Pesquisar
POST	/alunos	Cadastrar
PUT	/alunos/3	Modificar
DELETE	/alunos/2	Excluir

# REST: Modelo de Maturidade



## Nível 2 – Verbos HTTP

O nível garante que os verbos mais utilizados (GET, POST, PUT e DELETE) sejam aplicados de forma correta.

### Requisição

#### POST /alunos

<Aluno>

<Nome>Leonardo de Oliveira</Nome>

<Telefone>2234-2345</Telefone>

</Aluno>

### Resposta

201 Created

Location: /alunos/2

#### Principais características:

- Neste nível temos também o tratamento correto dos códigos de resposta.
- Presença do header “Location”. Esse header informa em qual endereço o recurso criado se encontra disponível.

# REST: Modelo de Maturidade



## Nível 3 – HATEOAS (*Hypermedia as the Engine of Application State*)

Neste nível a API fornece links que indicarão as ações que estarão disponíveis para os clientes. Nem todas as APIs utilizam este nível de maturidade.

A resposta não contém somente os dados do aluno, mas inclui uma URL com o endereço de onde as informações desse aluno podem ser localizadas.

- **rel** - significa relacionamento. No nosso caso o link faz referência ao próprio aluno.
- **href** - é uma URL completa que define um único recurso.

```
{
  "matricula": 324,
  "nome": "João",
  "telefone": "2234-0989",
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/alunos/1"
    }
  ]
}
```

# REST: Modelo de Maturidade



## Nível 3 – HATEOAS (*Hypermedia as the Engine of Application State*)

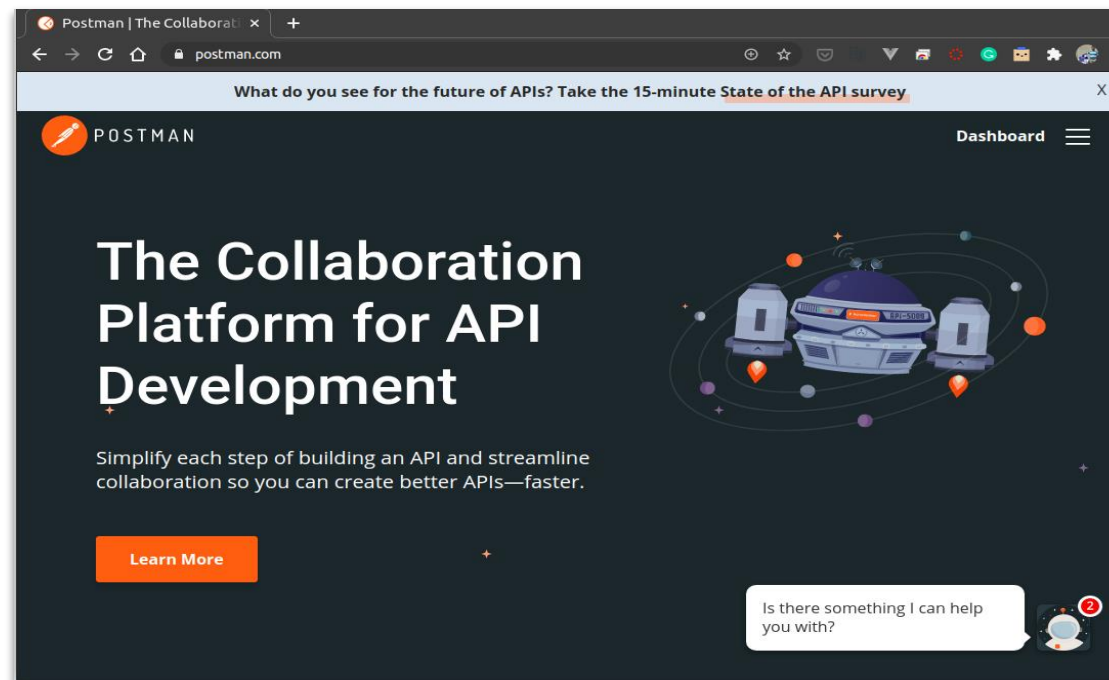
Nos exemplos abaixo, dependendo do saldo do cliente a API disponibiliza respostas diferentes, se o saldo for igual ou inferior a zero só o link de depósito é habilitado.

```
{
  "numero": 3134,
  "saldo": 500,
  "links": [
    {
      "rel": "deposito",
      "href": "http://localhost:8080/conta/3134/deposito"
    },
    {
      "rel": "saque",
      "href": "http://localhost:8080/conta/3134/saque"
    }
  ]
}
```

```
{
  "numero": 3134,
  "saldo": 0,
  "links": [
    {
      "rel": "deposito",
      "href": "http://localhost:8080/conta/3134/deposito"
    }
  ]
}
```

# REST: prática com Postman

Ferramenta de apoio ao desenvolvedor Rest  
<https://www.postman.com/>



# REST: Postman



Começando a trabalhar com o Postman.  
Vamos criar uma nova coleção clicando  
no botão conforme a imagem abaixo



Insira o nome e a descrição da coleção

CREATE A NEW COLLECTION

Name

Introdução ao Postman

Description Authorization Pre-request Scripts Tests Variables

This description will show in your collection's documentation, along with the descriptions of its folders and requests.

Teste com exemplos para utilização do Postman.

Descriptions support [Markdown](#)

Cancel Create



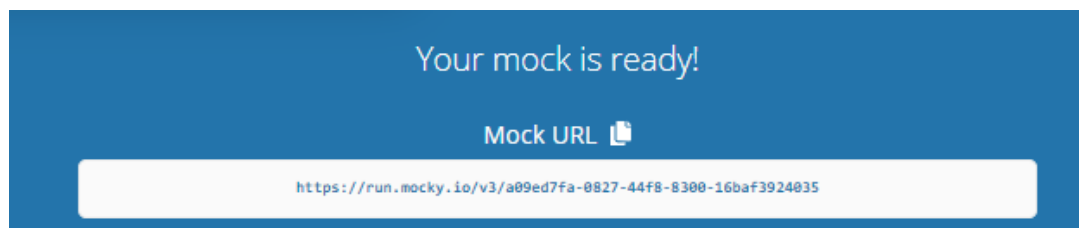
# REST: Postman



Vamos realizar um teste no Postman, para isso vamos o site <https://designer.mocky.io/> para criar uma api “mock” (imitação/simulação):

1. Acesse o site <https://designer.mocky.io/>
2. Clique em em new mock para criar uma nova requisição
3. preencha o corpo da requisição como na imagem
4. Clique no botão **Generate HTTP Response**

Será gerada uma url conforme a imagem abaixo:



Copie a url para que possamos usá-la no postman.

The image shows the 'Generate HTTP Response' form on the Mocky.io website. It includes the following sections:

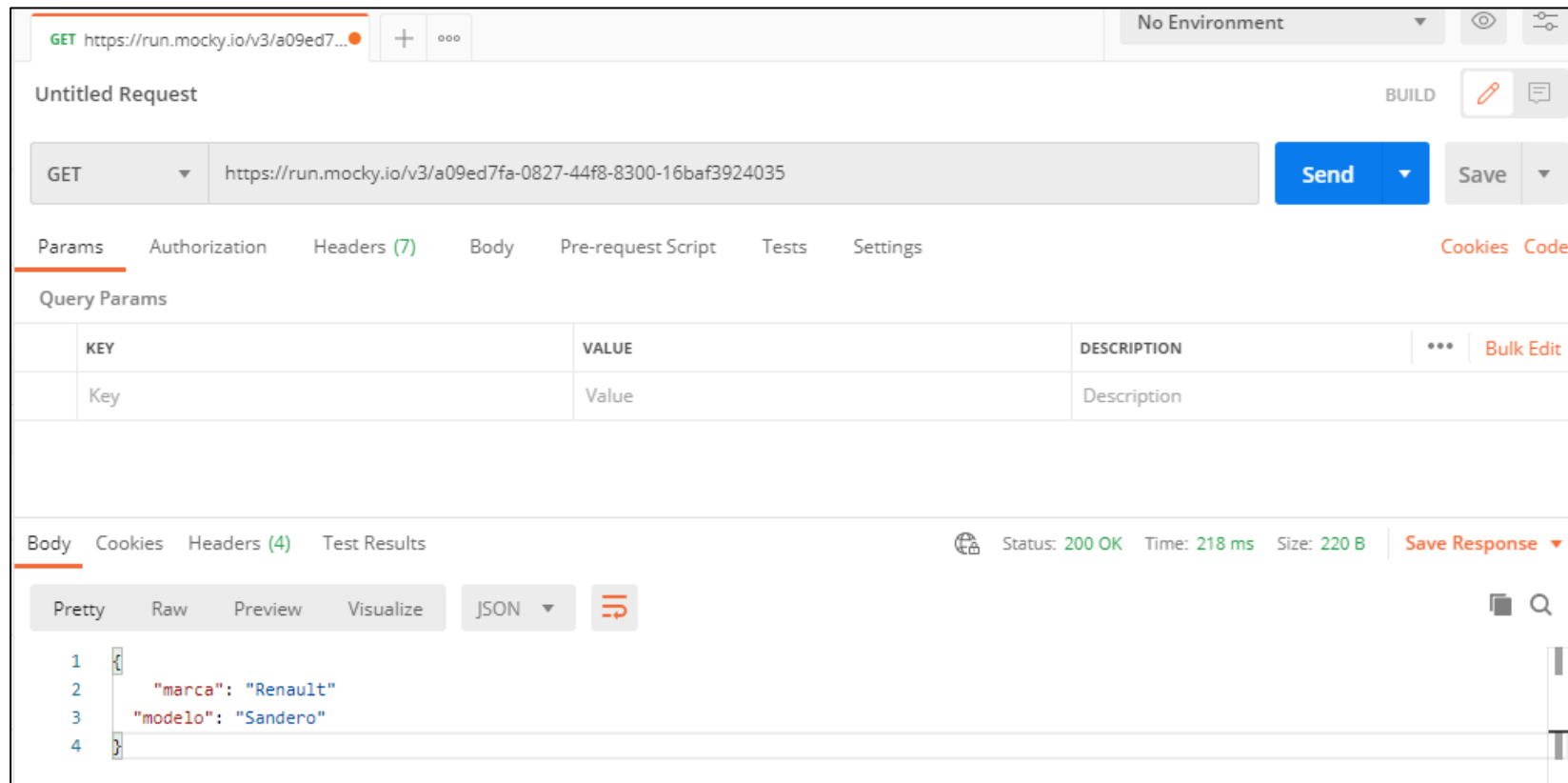
- HTTP Status** (REQUIRED): A dropdown menu set to '200 - OK'.
- Response Content Type** (REQUIRED): A dropdown menu set to 'application/json'.
- Charset** (REQUIRED): A dropdown menu set to 'UTF-8'.
- HTTP Headers** (OPTIONAL): A text area containing a JSON object: `{ "X-Foo-Bar": "Hello World" }`.
- HTTP Response Body** (OPTIONAL): A text area containing a JSON object: `{ "marca" : "Result", "modelo" : "Sander" }`.
- Options to manage your mock after its creation** (OPTIONAL): Two input fields for 'Secret token' and 'Mock identifier'.

At the bottom, there is a large blue button labeled 'GENERATE MY HTTP RESPONSE' and a 'Never expire' checkbox.

# REST: Postman



Cole a url como na imagem abaixo e clique no botão Send para enviar a requisição



Requisição

Resposta

# REST: Postman



The screenshot shows the Postman REST client interface. The left sidebar displays a 'History' tab with a list of recent requests. The main area shows an 'Untitled Request' with a 'GET' method selected. The 'Params' tab is active, showing a table for adding parameters. The 'Send' button is visible. The interface is annotated with numbers 1 through 7, corresponding to the list on the right.

1. Histórico de requisições realizadas

2. URL para onde irá disparar a requisição.

3. Botão para enviar.

4. Parâmetros e valores que serão adicionados a URL.

5. Parâmetros e valores que serão adicionados ao Header.

6. Selecione o tipo de requisição que irá enviar.

7. Parâmetros e valores que serão adicionados ao Body.

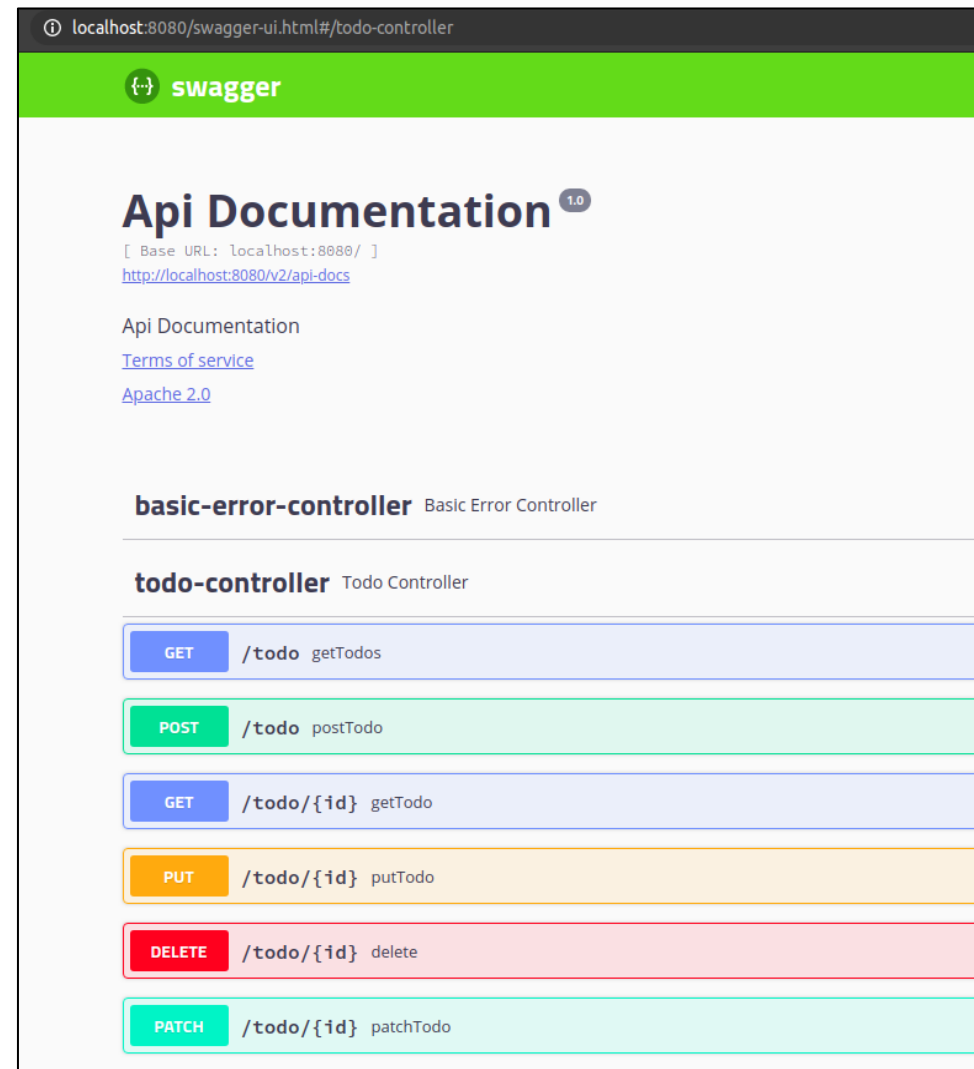
# REST: Swagger



O Swagger é um conjunto de ferramentas de software de código aberto para projetar, criar, documentar e usar serviços da Web RESTful, desenvolvidos pela SmartBear Software. Inclui documentação automatizada, geração de código e geração de casos de teste.

Wikipedia (inglês)

***Nem todos os serviços disponíveis na Web implementam o Swagger.***



# Servidores Web

Servidor web (web server) pode referir-se ao hardware, ao software, ou ambos.

- Hardware: computador que armazena arquivos que compõem os sites (documentos HTML, imagens, folhas de estilo, e arquivos JavaScript) e os entrega para o dispositivo do usuário final. Está conectado à Internet e pode ser acessado através do seu nome de domínio (DNS), como por exemplo mozilla.org.
- Software: sistema que inclui diversos componentes. Estes permitem que usuários acessem arquivos hospedados no servidor através de um protocolo de comunicação (HTTP). Ele processa URLs (endereço web) para identificar a informação a ser retornada (recurso)



## Conteúdo Estático

É chamado "estático" porque o servidor envia seus arquivos tal como foram armazenados (hospedados) no navegador. Também pode servir de "gateway" para outros servidores.

Exemplos:

- Apache Http Server
- Nginx

## Conteúdo Dinâmico

Também conhecido como servidor de aplicações (application server). É denominado de "dinâmico" porque o conteúdo é gerado por aplicações dinamicamente, utilizando dados do usuário conectado, bancos de dados, arquivos estáticos e até mesmo conteúdo de outros servidores.

# Servidores de Aplicação Java



Um Servidor Java WEB fornece um conjunto de classes (API - Application Program Interface) que pode ser utilizado para o desenvolvimento de aplicações web java. Essas aplicações são “instaladas” (deploy) dentro do servidor java web, também chamado de servidor de aplicação.

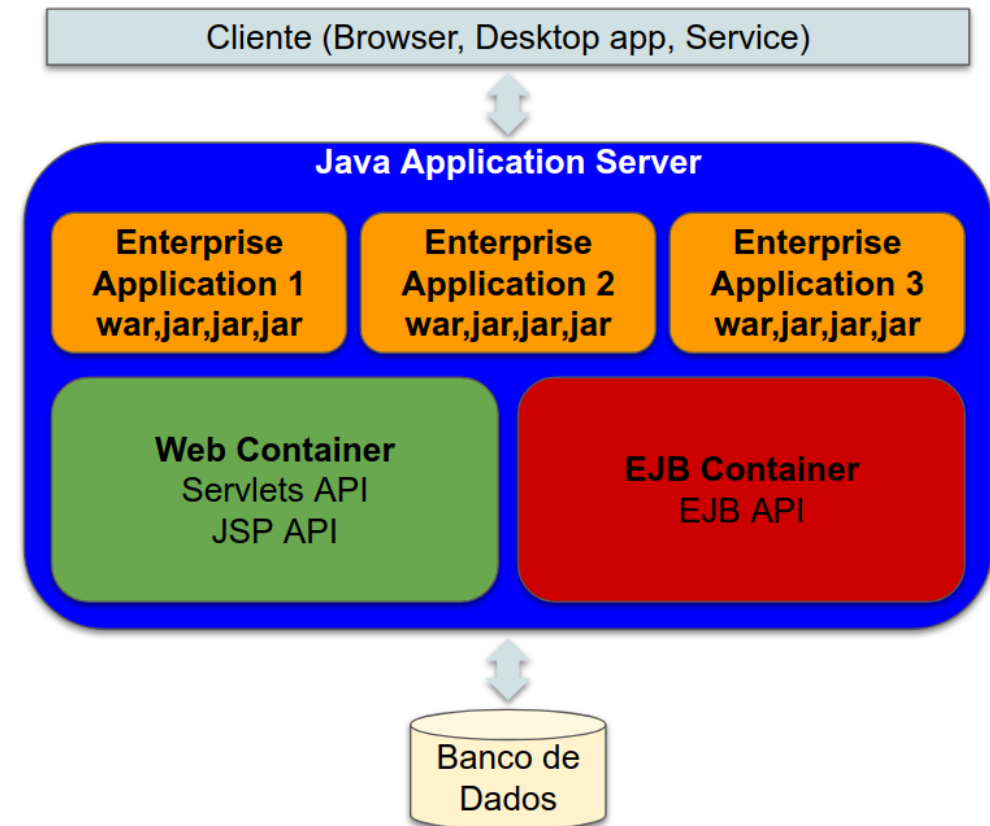
**Aplicação J2EE**

**Servidor de Aplicação**

**Java Virtual Machine**

**Sistema Operacional**

**Hardware**



# Servidores de Aplicação Java



Softwares que implementam a especificação J2EE (Java 2 Enterprise Edition) ou simplesmente JEE:

## JEE 8 - 2017

### Enterprise Application Technologies

- Batch Applications for the Java Platform 1.0
- Concurrency Utilities for Java EE 1.0
- Contexts and Dependency Injection for Java 2.0
- Dependency Injection for Java 1.0
- Bean Validation 2.0
- Enterprise JavaBeans 3.2
- Interceptors 1.2
- Java EE Connector Architecture 1.7
- Java Persistence 2.2
- Common Annotations for the Java Platform 1.3
- Java Message Service API 2.0
- Java Transaction API (JTA) 1.2
- JavaMail 1.6

### Web Services Technologies

- Java API for RESTful Web Services (JAX-RS) 2.1
- Implementing Enterprise Web Services 1.3
- Web Services Metadata for the Java Platform 2.1
- Java API for XML-Based RPC (JAX-RPC) 1.1 (Optional)
- Java API for XML Registries (JAXR) 1.0 (Optional)
- Management and Security Technologies
- Java EE Security API 1.0
- Java Authentication Service Provider Interface for Containers 1.1
- Java Authorization Contract for Containers 1.5
- Java EE Application Deployment 1.2 (Optional)
- J2EE Management 1.1
- Debugging Support for Other Languages 1.0
- Java EE-related Specs in Java SE
- Java Management Extensions (JMX) 2.0
- SOAP with Attachments API for Java (SAAJ) Specification 1.3
- Streaming API for XML (StAX) 1.0
- Java API for XML Processing (JAXP) 1.6
- Java Database Connectivity 4.0
- Java Architecture for XML Binding (JAXB) 2.2
- Java API for XML-Based Web Services (JAX-WS) 2.2
- JavaBeans Activation Framework (JAF) 1.1



# Servidores de Aplicação Java



**ORACLE®**

WEBLOGIC SERVER



**GlassFish**



**IBM**  
**WebSphere**

**Servlet Containers**



**Apache Tomcat**

**jetty://**

# Servidores de Aplicação Java



Uma aplicação web java é distribuída em um arquivo war (zip):

- As classes compiladas da aplicação
  - classes de regra de negócio
  - classes de acesso ao banco de dados
  - servlets (classes java que estendem a classe HttpServlet da API JEE. São responsáveis por receber requisições dos clientes, e gerar uma saída - html, texto, imagem, etc)
- Páginas jsp (página com codificação HTML e com codificação Java, inserida entre as tag's <% e %>)
- Arquivos estáticos
  - páginas htmls
  - imagens
  - arquivos javascript
  - arquivos css
- Arquivo web.xml de configuração
- Os arquivos jars das bibliotecas usadas na aplicação

Este arquivo é “instalado” no servidor de aplicação:

- Em uma pasta específica (pasta deploy no jboss ou webapp no tomcat)
- Usando um console administração (página do servidor que permita fazer o “upload” da aplicação)

# Páginas dinâmicas com Servlets

HelloWorld.java

```
package br.org.residencia.olamundo;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE html >\n"+
            "<html>\n"+
            "<head><title>Olá mundo</title></head>\n"+
            "<body>\n"+
            "<h1> Bem vindos ao curso de APIs</h1>"+
            "</body>"+
            "</html>");
    }
}
```

web.xml

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
        version="3.1">
    <servlet>
        <servlet-name>helloworld</servlet-name>
        <servlet-class>org.serratec.HelloWorld</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>helloworld</servlet-name>
        <url-pattern>/olamundo/*</url-pattern>
    </servlet-mapping>
</web-app>
```

# Páginas dinâmicas com JSPs

```
<body>
  <%
    Connection con = DriverManager.getConnection(.....);
    Statement stmt = con.createStatement();
    String qry = "SELECT autor, titulo, preco from livros";
    Resultset rs = stmt.executeQuery(qry);
  %>
  <table>
    <tr><th>Autor</th><th>Titulo</th><th>Preço</th></tr>
    <%
      while (rs.next()) {
        <%>
        <tr>
          <td><%=rs.getString("autor")</td>
          <td><%=rs.getString("titulo")</td>
          <td><%=rs.getDouble("preco")</td>
        </tr>
        <%
      } %>
    </table>
  </body>
```

# Páginas dinâmicas com JSPs (“compiladas”)

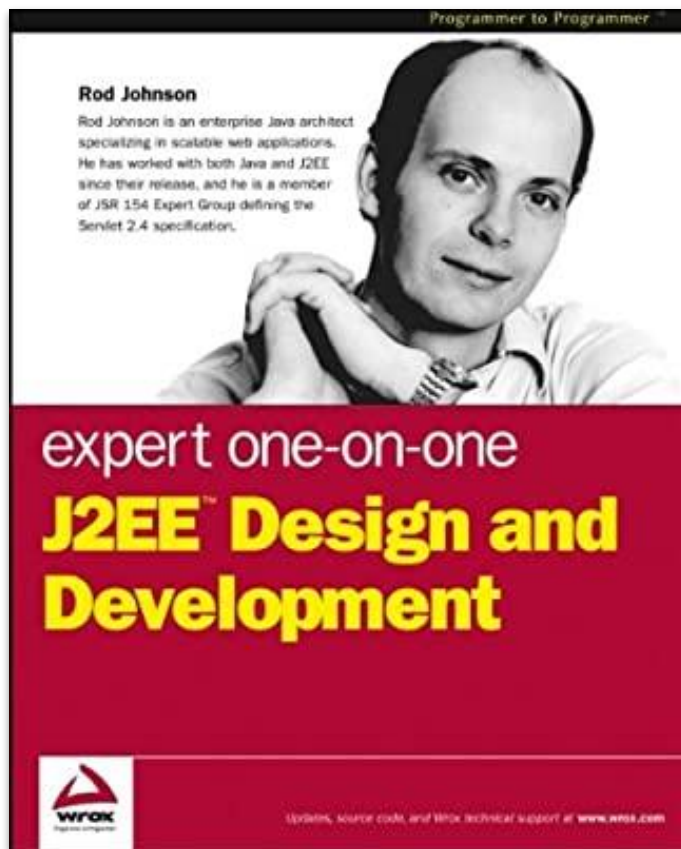
```
<body>
<%
    Connection con = DriverManager.getConnection(.....);
    Statement stmt = con.createStatement();
    String qry = "SELECT autor, titulo, preco from livros";
    Resultset rs = stmt.executeQuery(qry);
%>
<table>
    <tr><th>Autor</th><th>Titulo</th><th>Preço</th></tr>
    <%
        while (rs.next()) {
            <%>
            <tr>
                <td><%=rs.getString("autor")</td>
                <td><%=rs.getString("titulo")</td>
                <td><%=rs.getDouble("preco")</td>
            </tr>
            <%
        } %>
    </table>
</body>
```

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException{
    PrintWriter out = response.getWriter();
    out.println("...");
    out.println("<body>\n");
    Connection con = DriverManager.getConnection(.....);
    Statement stmt = con.createStatement();
    String qry = "SELECT autor, titulo, preco from livros";
    ResultSet rs = stmt.executeQuery(qry);
    out.println("<table>\n"+
               "<tr><th>Autor</th>\n"+
               "<th>Titulo</th>\n"+
               "<th>Preço</th></tr>");
    while (rs.next()) {
        out.println("<tr><td>");
        out.println(rs.getString("autor"));
        out.println("</td>");
        ....
    }
    out.println("</table>\n");
    ....
}
```

# Spring vs JEE

JEE introduzido em 1999

- Excesso de configurações em xml
- Configuração complexa
- Dependia de um comitê para evoluir (JCP - java community process)
- Demorava anos para sofrer atualizações



Em outubro de 2002, Rod Johnson lança o livro “Expert One-on-One J2EE Design and Development”, contendo 30.000 linhas de código.

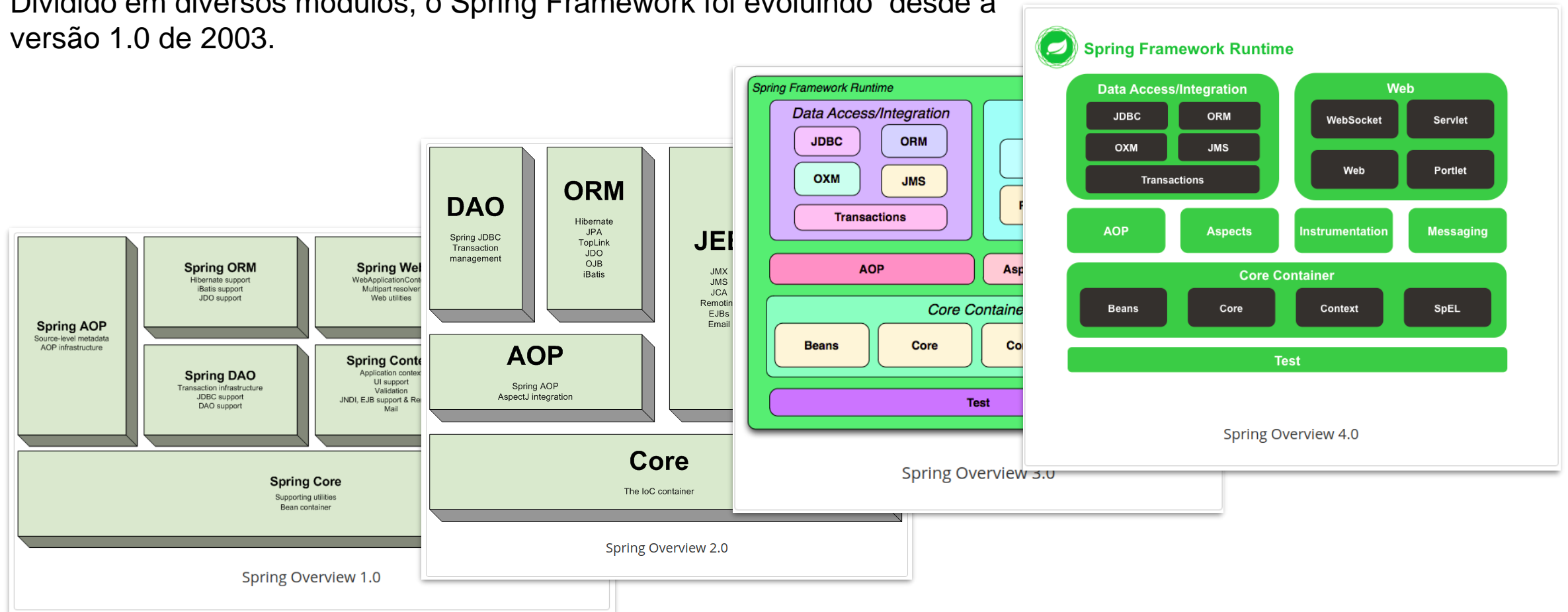
O livro apresentava diversas soluções e alternativas para a complexidade e deficiências do JEE.

Posteriormente, este código se tornou a base da primeira versão do Spring.

# Introdução - Spring Framework



Dividido em diversos módulos, o Spring Framework foi evoluindo desde a versão 1.0 de 2003.





# Introdução - Spring Framework



## Principais módulos do Spring:

- **Spring Core Module:** principal módulo do spring, dando suporte a todos os outros módulos
- **Spring AOP Module:** responsável pela parte de programação orientada a aspectos. Auxilia na parte de controle de transação, log e monitoramento.
- **Spring ORM Module:** utilizado para o acesso ao bancos de dados utilizando mapeamento objeto relacional.
- **Spring Web MVC Module:** dá suporte a aplicações web
- **Spring Web Flow Module:** especializado em implementação de workflows (processos em etapas)
- **Spring Web DAO Module:** simplifica acesso a fontes de dados
- **Spring Application Context Module:** baseado no Core Module, este módulo é a espinha dorsal do Spring, implementando diversas funcionalidades que dão suporte a todos os módulos do Spring

# Introdução - Spring Framework



O desenvolvedor pode priorizar o desenvolvimento das regras de negócio da aplicação, enquanto o Spring resolve o código comum que aplicações necessitam:

## Acesso a banco JDBC

```
public List<Cliente> getClientesJDBC() {
    List <Cliente> clientes = new ArrayList<>();
    try (Connection con=getConnection()) {
        ResultSet rs = con.createStatement()
                                .executeQuery("Select * from
Clientes");
        while (rs.next()) {
            Cliente c = new Cliente();
            c.setId(rs.getInt("id"));
            c.setNome(rs.getString("nome"));
            c.setCpf(rs.getString("cpf"));
            clientes.add(c);
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally{
        try {
            con.close();
        } catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
    return clientes;
}
```

## Acesso a banco Spring

```
public List<Cliente> getClientesSpring() {

    List<Cliente> lista = new JdbcTemplate(getDataSource())
        .query("Select * from Clientes", new BeanPropertyRowMapper(Cliente.class));

    return lista;
}
```

# Introdução - Spring Boot



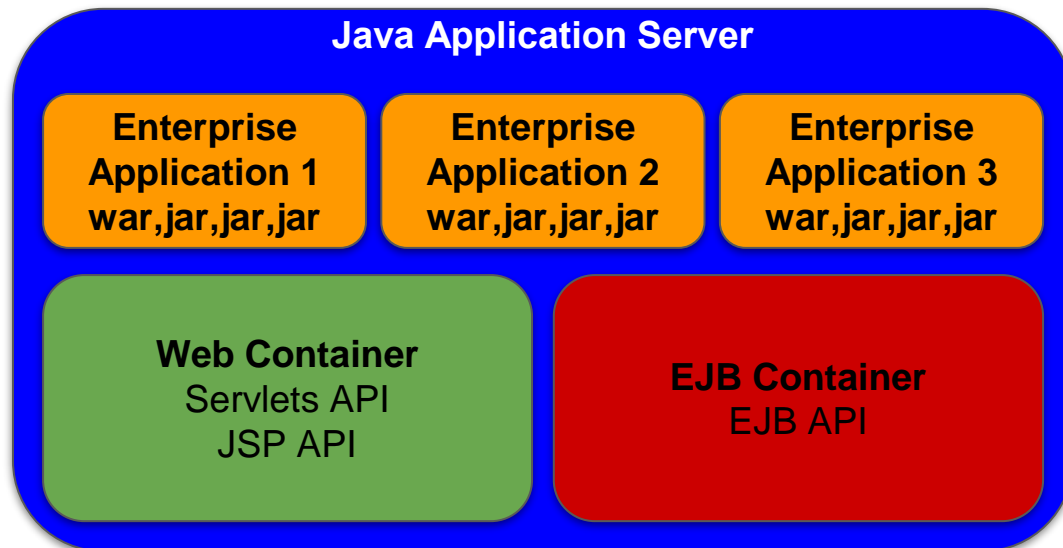
- Simplifica o desenvolvimento de aplicações Java Web com Spring
- Início de desenvolvimento mais rápido
- Minimizar, ou até mesmo eliminar a necessidade de configurações em arquivos XML
- Maioria dos componentes pré-configurados e opcionais
  - Segurança
  - Acesso a banco de dados
  - Métricas
  - **Servidor de Aplicações embutido**

# Introdução - Spring Boot

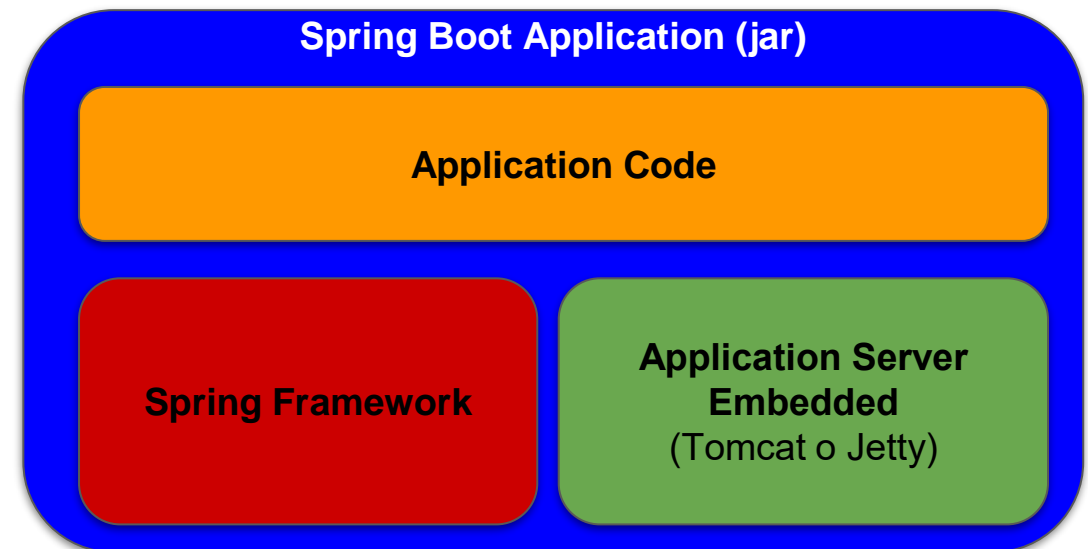


Aplicações Java web mais modernas utilizam servidores java web “embutidos”, como o tomcat ou o jetty. Estas aplicações são distribuídas em arquivos jars, como aplicações java normais, e geralmente não geram páginas html dinâmicas. São utilizadas principalmente para serviços, como APIs (application program interface), fornecendo e recebendo dados em formato JSON ou XML.

Ambiente JEE padrão



Aplicação desenvolvida em Spring Boot



# Spring Boot Framework

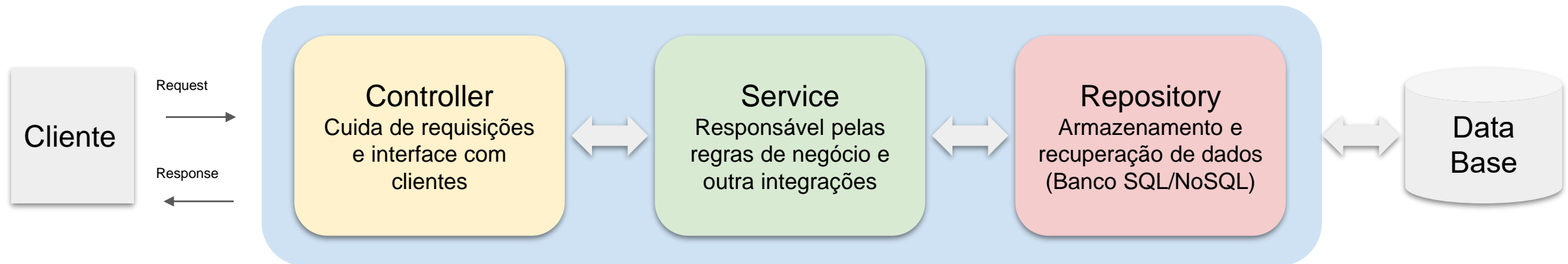


***É uma boa plataforma para desenvolvedores Java desenvolverem uma aplicação Spring independente e pronta para produção que você pode simplesmente "roda".***

- Microframework
- Utiliza o “eco-sistema” do Spring Framework
- *Convenção sobre Configuração*
- Baseado em POJOs - Plain Old Java Object

## **Vantagens**

- Começar com configurações mínimas
- Fácil de entender e desenvolver
- Aumento de produtividade
- Reduzir o tempo de desenvolvimento



# Spring Boot Framework



Programador com foco em negócio ao invés da tecnologia

Controller

```
@RestController("/todo")
public class TodoController {
    @Autowired
    TodoService todoService;
    @GetMapping("/count")
    public Integer getCount() {
        return todoService.getCount();
    }
}
```

Service

```
@Service
public class TodoService {
    @Autowired
    TodoRepository todoRepository;
    @Transactional
    public Integer getCount() {
        return todoRepository.contar();
    }
}
```

Repository

```
@Repository
public interface TodoRepository extends
    JpaRepository<Todo,Integer>{

    @Query("select count(*) from Todo")
    public Integer contar() ;

}
```

Arquivo de Configuração

```
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
spring.datasource.url=jdbc:postgresql://localhost:5432/testdb
spring.datasource.username=usuario
spring.datasource.password=senha
```

Entity

```
@Entity
@Table(name="todo")
public class Todo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String titulo;
    private String descricao;
    private Boolean completado;
}
```

# REST: Exercícios



Acessar diversas APIs Rests abertas:

<https://dadosabertos.camara.leg.br/swagger/api.html>

<https://covid19-brazil-api-docs.now.sh/>

<https://api.le-systeme-solaire.net/swagger>

Mais informações sobre rest:  
<https://pt.wikipedia.org/wiki/REST>

# REST: Exercícios



Acessar o site [viacep.com.br](https://viacep.com.br) visualizar a documentação e digitar o cep da sua rua para consulta no Postman.

The screenshot shows a Postman interface with a GET request to `viacep.com.br/ws/25620000/json`. The response is a JSON object with the following data:

KEY	VALUE	DESCRIPTION
Key	Value	Description

The response body is displayed in JSON format:

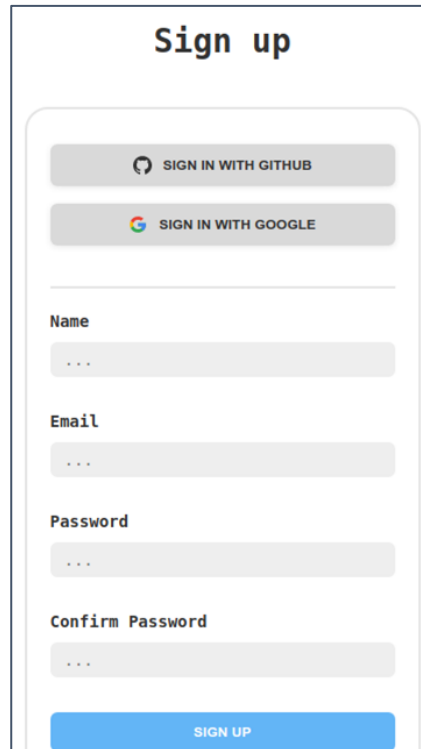
```
1 {  
2   "cep": "25620-000",  
3   "logradouro": "Rua do Imperador",  
4   "complemento": "até 552 - lado par",  
5   "bairro": "Centro",  
6   "localidade": "Petrópolis",  
7   "uf": "RJ",  
8   "ibge": "3303906",  
9   "gia": "",  
10  "ddd": "24",  
11  "siafi": "5877"  
12 }
```



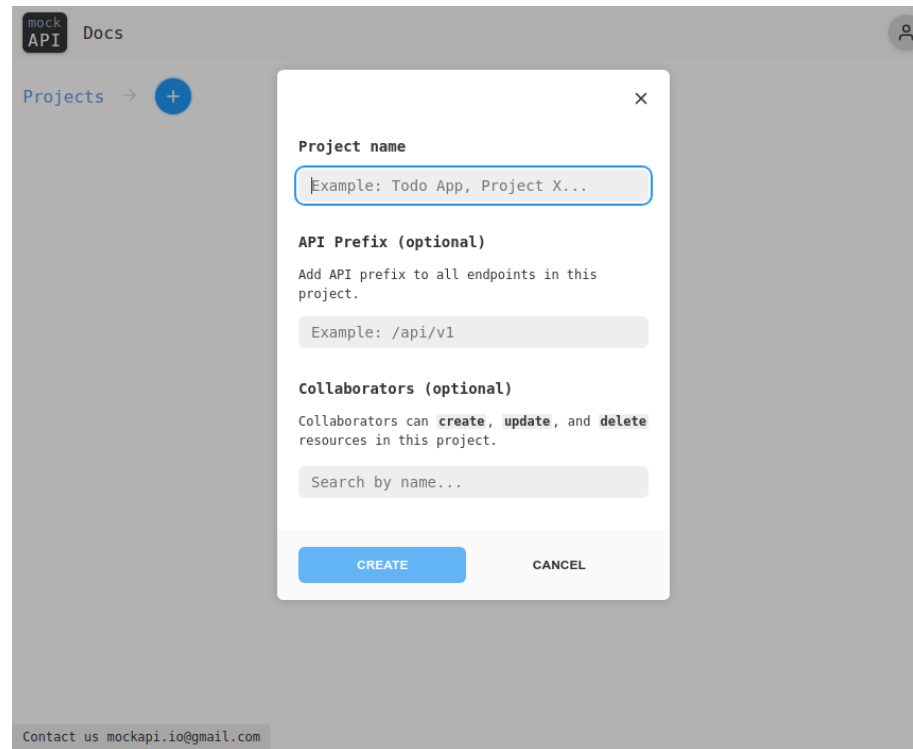
# REST: Exercícios



1) Acessar o site <https://www.mockapi.io/signup>, criar uma conta para efetuar testes com os verbos http no Postman



The image shows the 'Sign up' form on the MockAPI website. It features a clean, modern design with a white background and a blue border. At the top, the text 'Sign up' is displayed in a bold, sans-serif font. Below this, there are two buttons for social login: 'SIGN IN WITH GITHUB' and 'SIGN IN WITH GOOGLE'. Underneath these buttons, there are four input fields for user registration: 'Name', 'Email', 'Password', and 'Confirm Password'. Each input field has a placeholder text '...' and a small blue eye icon to toggle visibility. At the bottom of the form, there is a large blue button labeled 'SIGN UP'.



The image shows the 'Project creation' modal in the MockAPI application. The modal is a white box with a blue border, overlaid on a grey background. It contains the following fields and options:

- Project name:** A text input field with a placeholder 'Example: Todo App, Project X...'.
- API Prefix (optional):** A text input field with a placeholder 'Example: /api/v1'. Below it, a small text explains: 'Add API prefix to all endpoints in this project.'
- Collaborators (optional):** A section with a text input field labeled 'Search by name...' and a list of collaborators (not visible in the image).

At the bottom of the modal, there are two buttons: 'CREATE' (blue) and 'CANCEL' (grey). The background shows the 'mock API Docs' interface with a 'Projects' tab and a '+' button to add a new project.

2) Procurar outras API públicas e criar requisições no Postman para testes.