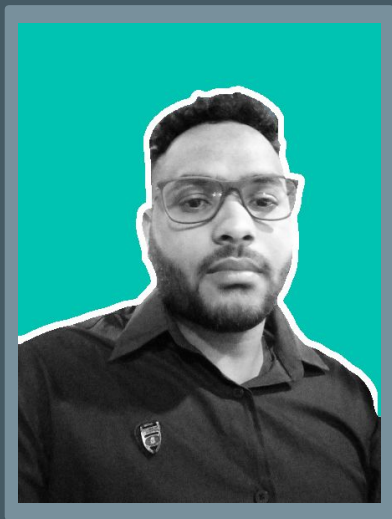




# REST

Fundamentos para sistemas  
com arquiteturas REST



# Olá!

## Eu sou Willian Brito

- ❖ Desenvolvedor FullStack na Msystem Software
- ❖ Formado em Analise e Desenvolvimento de Sistemas
- ❖ Pós Graduação em Segurança Cibernética
- ❖ Certificação SYCP (Solyd Certified Pentester) v2018

**“REST, é um estilo arquitetural que define um conjunto de princípios e restrições a serem usados para o desenvolvimento de web services e APIs.”**

# Introdução

O pleno entendimento de **REST** começa pela interpretação do que o acrônimo significa. **RE**presentational **S**tate **T**ransfer literalmente expressa a intencionalidade do estilo arquitetural: **transferência de estado através de representações**. Ele foi introduzido e definido, no ano 2000, por **Roy Fielding** (**Cientista da Computação norte-americano**), em sua tese de doutorado, com o objetivo de tentar resolver problemas do protocolo **SOAP**, como **alta complexidade de implementação** e **lentidão na comunicação**. Uma curiosidade é que Roy Fielding é um dos criadores do **Protocolo HTTP**.

As transferências de estado, em sentido amplo, não se restringem apenas pelo conjunto de dados associados a um recurso, mas também às operações que este recurso suporta. Os diversos componentes computacionais que formam uma solução REST “transferem estado” entre si, a fim de preservar a consistência.

# Diferença entre REST e RESTful

Você já deve ter ouvido os termos Rest e Restful e ter ficado em dúvida. Afinal, eles são a mesma coisa? Veja a diferença.

- **Rest**: É um conjunto de princípios e restrições de arquitetura.
- **Restful**: É uma condição única de aplicar os conceitos de Rest nas aplicações web.

Enquanto o primeiro é algo mais **abstrato** e está voltado à criação de serviços disponibilizados na web, o segundo é algo mais **concreto** que está ligado à **implementação correta** do padrão REST.

# Maturidade de Richardson

É um modelo criado por **Leonard Richardson** que quebra os elementos de uma API REST em **3 níveis**. Sendo assim, para sua API ser considerada RESTful você teria que alcançar o nível 3.

## ❑ Nível 0: HTTP

Você usa **HTTP** como forma de comunicação sem qualquer critério para a utilização de verbos, ou de rotas.

## ❑ Nível 1: HTTP + Recursos

Sua API está exposta (roteada) seguindo um **mapeamento de recursos**. Como **/users/** para listar todos os usuários e **/users/123/** para obter um usuário específico, lembrando que para mapear um recurso de forma correta, os nomes das rotas devem ser **substantivos** e **não verbos**.

## ❑ Nível 2: HTTP + Recursos + Verbos

Os verbos HTTP são usados de forma semântica na sua API. **GET** para leitura, **POST** para inserir, **PUT** para substituir um registro, **DELETE** para excluir, e os retornos com **Status Code** corretos.

## ❑ Nível 3: HTTP + Recursos + Verbos + HATEOAS

A sua API deve retornar uma **relação dos recursos** que é uma lista de recursos (rotas) com tudo o que é possível fazer a partir da chamada original, como **navegação** e **operações**.



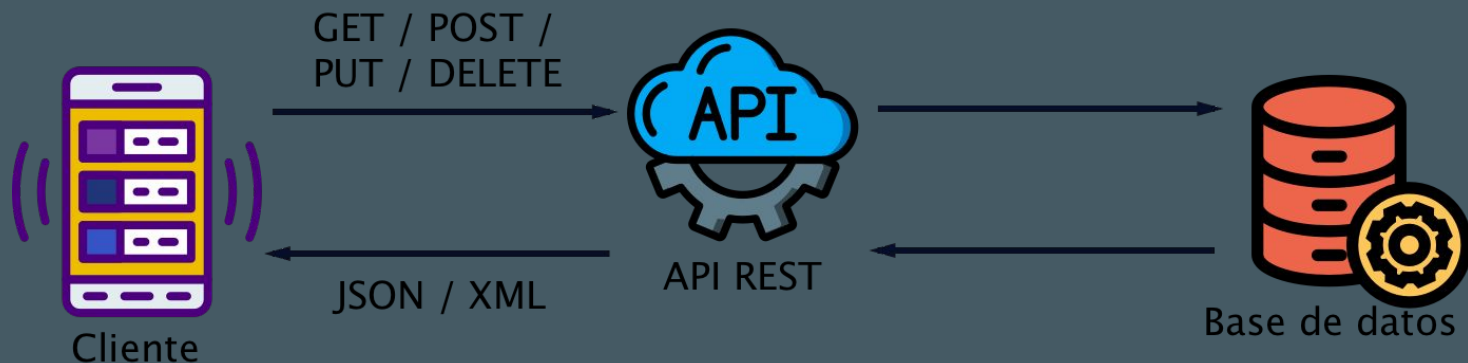
# APIs

Interface de Programação de Aplicação

# O que são APIs ?

A expressão **Application Programming Interface**, ou, em português, Interface de Programação de Aplicativos, originou o acrônimo **API**. Ou seja, APIs são “**pontes**” com a função de conectar sistemas, softwares e aplicativos. Dessa forma, é possível separar funcionalidades mantendo uma boa arquitetura de software.

APIs têm importância crescente no design de aplicações. Boas APIs facilitam o estabelecimento de parcerias, permitindo que sistemas se comuniquem de maneira sólida, facilitando a automação dos negócios.





# Para que serve uma API ?

Uma API é criada quando uma empresa de software tem a intenção de que outros criadores de software desenvolvam produtos associados ao seu serviço.

As APIs podem ser utilizadas de diversas maneiras, integrando diferentes sistemas para maior eficiência na hora do uso, e têm uma função estratégica na rotina das empresas. Afinal, existem diversos sistemas e aplicativos usados em um negócio e todos esses recursos interagem com outros softwares, via APIs.

O exemplo mais comum de serventia da API é o de um funcionário que precisa emitir notas e boletos para finalizar o pedido de um cliente. Nesse caso, a API pode conectar o sistema de gestão da empresa (ERP) ao sistema de geração de boletos do banco e ao de emissão de notas da prefeitura. Assim, o colaborador precisa apenas inserir os dados uma vez e finalizar o processo com poucos cliques.

# Arquitetura de APIs

O projeto de APIs, é uma atividade fundamental de arquitetura. É essencial que o arquiteto responsável garanta a seleção do estilo mais adaptado às necessidades. Desde o início dos anos 2000, vem ganhando destaque APIs projetadas conforme o **estilo arquitetural REST**, embora, infelizmente, as implementações não sejam, muitas vezes, “fiéis” aos princípios estabelecidos nesse estilo.

APIs que utilizam verbos padrões HTTP: **GET**, **PUT**, **POST** e **DELETE** e entregam conteúdo em **XML** ou **JSON** são designadas como **RESTful**. Entretanto, estas características não são suficientes para tal designação.

**“Não projete APIs para serem RESTful, projete-as para terem as propriedades que você necessita!”**

**Roy Fielding**



WWW

World Wide Web

# O que é World Wide Web ?

World Wide Web, o famoso WWW, é um sistema de informação onde documentos e outros recursos são interligados através de identificados URLs (Uniform Resource Locators), como <https://example.com/> e executados na Internet que permitem o acesso às informações apresentadas no formato de hipertexto.

Os documentos da world wide web podem estar na forma de vídeos, sons, hipertextos e figuras, e para visualizar a informação, utiliza-se um programa de computador chamado navegador para descarregar essas informações, e mostrá-las na tela do usuário. Os navegadores mais famosos são: Internet Explorer, Mozilla Firefox, Google Chrome e Safari.

# História do WWW

A ideia de World Wide Web surgiu em 1980, na Suíça. O precursor da ideia foi o britânico [Tim Berners-Lee](#). Um computador NeXTcube foi usado por Berners-Lee como primeiro servidor web e também para escrever o primeiro navegador enquanto trabalhava no CERN (European Organization for Nuclear Research), o [WorldWideWeb](#), em 1990.

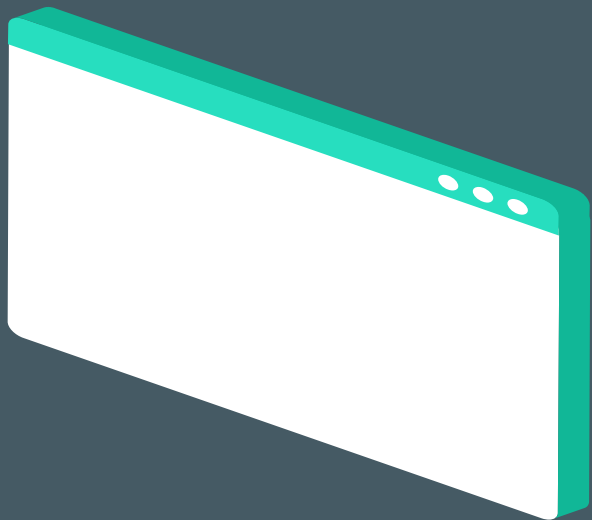
O navegador foi lançado fora do CERN para outras instituições de pesquisa a partir de janeiro de 1991, e depois para o público em geral em 6 de agosto de 1991 onde Tim Berners-Lee postou um resumo sobre todas as suas ideias e projetos no grupo de notícias de nome alt.hypertext. Esta data marca a estréia oficial da Web como um serviço publicado na Internet. A Web começou a entrar no uso diário em 1993, quando sites de uso geral começaram a se tornar disponíveis.

Desde então, a Internet cresceu em proporções gigantescas. A quantidade de informações que está disponível no universo online é muito mais do que você poderia assimilar durante uma vida inteira. A chance de se perder em meio à tanta informação é muito grande, por esta razão é muito importante a forma como tais informações estão dispostas. É aí que entra o hipertexto.

# Páginas de Hipertexto

Os hipertextos são textos exibidos em formato digital, os quais podem conter informações em formato de imagens, sons, vídeos, etc. O acesso a tais informações se dá por meio de links, que servem como uma ponte entre os mais diversos sites da Internet e seus conteúdos.

O hipertexto é codificado com a linguagem HTML, que possui um conjunto de marcas de codificação que são interpretadas pelos clientes WWW, em diferentes plataformas. O protocolo usado para a transferência de informações no WWW é o **HTTP**, que é um protocolo do nível de aplicação que possui objetividade para suportar sistemas de informação distribuídos, cooperativos e de hipermídia.



# HTTP

HyperText Transfer Protocol

# Protocolo HTTP

Como mencionado anteriormente, o modelo REST foi desenvolvido por Roy Fielding, um dos criadores do protocolo HTTP. Portanto, fica naturalmente evidente, para quem conhece ambos os modelos, as semelhanças entre ambos, sendo que, na realidade, REST é idealmente concebido para uso com o protocolo HTTP. Portanto, para ser um bom usuário de REST, é necessário ter um bom conhecimento do protocolo HTTP.

O protocolo HTTP (HyperText Transfer Protocol - Protocolo de Transferência de Hipertexto) data de 1996, época em que os trabalhos conjuntos de Tim Berners-Lee, Roy Fielding e Henrik Frystyk Nielsen levaram à publicação de uma RFC (Request for Comments) descrevendo este protocolo. Trata-se de um protocolo de camada de aplicação (segundo o modelo OSI) e, portanto, de relativa facilidade de manipulação em aplicações.



# Fundamentos HTTP

Este protocolo foi desenvolvido de maneira a ser o mais flexível possível para comportar diversas necessidades diferentes. Em linhas gerais, este protocolo segue o seguinte formato de requisições:

<método> <URL> HTTP/<versão>

<Cabeçalhos - Sempre vários, um em cada linha>

<corpo da requisição>

Exemplo de requisição:

```
GET /cervejaria/clientes HTTP/1.1
Host: localhost:8080
Accept: text/html
```

# Fundamentos HTTP

Com isso, o método GET foi utilizado para solicitar o conteúdo da URL `/cervejaria/clientes`. Além disso, o protocolo HTTP versão 1.1 foi utilizado.

Note que o host, ou seja, o servidor responsável por fornecer estes dados, é passado em um cabeçalho à parte. No caso, o host escolhido foi localhost, na porta 8080. Além disso, um outro cabeçalho, Accept, foi fornecido.

A resposta para as requisições seguem o seguinte formato geral:

```
HTTP/<versão> <código de status> <descrição do código>  
<cabeçalhos>  
<resposta>
```

# Fundamentos HTTP

Por exemplo, a resposta para esta requisição foi semelhante à seguinte:

Aqui, note que o código 200 indicou que a requisição foi bem-sucedida, e o cabeçalho Content-Length trouxe o tamanho da resposta no caso, o XML que contém a listagem de clientes. Além disso, o cabeçalho Content-Type indica que a resposta é, de fato, XML através de um tipo conhecido como Media Type.

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 245
```

```
<clientes>
  <cliente id="1">
    <nome>Alexandre</nome>
    <dataNascimento>2012-12-01</dataNascimento>
  </cliente>
  <cliente id="2">
    <nome>Paulo</nome>
    <dataNascimento>2012-11-01</dataNascimento>
  </cliente>
</clientes>
```

# Métodos HTTP

A versão corrente do HTTP, 1.1, define oficialmente oito métodos, embora o protocolo seja extensível em relação a estes métodos.

Hoje, estes oito são:

- GET
- POST
- PUT
- DELETE
- OPTIONS
- HEAD
- TRACE
- CONNECT

Cada método possui particularidades e aplicações de acordo com a necessidade. Estas particularidades são definidas em termos de idempotência, segurança e mecanismo de passagem de parâmetros. Além disso, por cada um possui particularidades de uso, neste conteúdo considere apenas os seis primeiros os métodos TRACE e CONNECT não serão abordados.

# Idempotência

A idempotência de um método é relativa às modificações que são realizadas em informações do lado do servidor. Trata-se do efeito que uma mesma requisição tem do lado do servidor, se a mesma requisição, realizada múltiplas vezes, provoca alterações no lado do servidor como se fosse uma única, então esta é considerada idempotente.

Por exemplo, considere as quatro operações de bancos de dados: SELECT, INSERT, UPDATE e DELETE. Realizando um paralelo destas com o conceito de idempotência.

observe o seguinte:

```
SELECT * from CLIENTES;
```

Note que esta requisição, para um banco de dados, terá o mesmo efeito todas as vezes em que for executada (obviamente, assumindo que ninguém está fazendo alterações nas informações que já estavam gravadas).

Agora, observe o seguinte:

```
INSERT INTO CLIENTES VALUES (1, 'Alexandre');
```

# Idempotência

Esta requisição, por sua vez, provocará diferentes efeitos sobre os dados do banco de dados todas as vezes que for executada, dado que está aumentando o tamanho da tabela. Portanto, ela não é considerada idempotente.

Note que este conceito não está relacionado à realização ou não de modificações.

Por exemplo, considere as operações UPDATE e DELETE:

```
UPDATE CLIENTES SET NOME = 'Willian Brito' WHERE ID = 1;  
DELETE FROM CLIENTES WHERE ID = 1;
```

Note que, em ambos os casos, as alterações realizadas são idênticas à todas as subsequentes.

Por exemplo, suponha os dados:

ID	NOME
1	Alexandre

# Idempotência

Se a requisição de atualização for enviada, estes dados ficarão assim:

```
ID NOME
```

```
-----
```

```
1 Willian Brito
```

E então, caso a mesma requisição seja enviada repetidas vezes, ainda assim provocará o mesmo efeito que da primeira vez, sendo considerada, portanto, idempotente.

# Segurança

Quanto à segurança, os métodos são assim considerados se não provocarem quaisquer alterações nos dados contidos. Ainda considerando o exemplo das operações de bancos de dados, por exemplo, o método SELECT pode ser considerado seguro INSERT, UPDATE e DELETE, não.

Em relação a estas duas características, a seguinte distribuição dos métodos HTTP é feita:

	Idempotente	Seguro
GET	X	X
POST		
PUT	X	
DELETE	X	
HEAD	X	X
OPTIONS	X	X



# Parâmetros

Os métodos HTTP suportam parâmetros sob duas formas: os chamados query parameters e body parameters.

Os query parameters são passados na própria URL da requisição. Por exemplo, considere a seguinte requisição para o serviço de busca do Google:

<http://www.google.com.br/?q=HTTP>

Esta requisição faz com que o servidor do Google automaticamente entenda a string HTTP como um parâmetro. Inserir esta URL no browser automaticamente indica para o servidor que uma busca por HTTP está sendo realizada.

Os query parameters são inseridos a partir do sinal de interrogação. Este sinal indica para o protocolo HTTP que, de ali em diante, serão utilizados query parameters. Esses são inseridos com formato <chave>=<valor> (assim como no exemplo, em que q é a chave e HTTP, o valor). Caso mais de um parâmetro seja necessário, os pares são separados com um &. Por exemplo, a requisição para o serviço do Google poderia, também, ser enviada da seguinte maneira:

<http://www.google.com.br/?q=HTTP&oq=HTTP>

# Parâmetros

Os query parameters são enviados na própria URL. Isto quer dizer que a requisição para o Google é semelhante à seguinte:

```
GET /?q=HTTP&oq=HTTP HTTP/1.1  
Host: www.google.com.br
```

Note que o caracter de espaço é o separador entre o método HTTP, a URL e a versão do HTTP a ser utilizada. Desta forma, se for necessário utilizar espaços nos query parameters, é necessário utilizar uma técnica chamada de codificação da URL. Esta técnica adapta as URL's para que elas sejam compatíveis com o mecanismo de envio de dados, e codificam não apenas espaços como outros caracteres especiais e caracteres acentuados.

Esta codificação segue a seguinte regra:

- Espaços podem ser codificados utilizando + ou a string %20.
- Letras maiúsculas e minúsculas, números e os caracteres ., -, ~ e \_ são deixados como estão.
- Caracteres restantes são codificados de acordo com sua representação ASCII / UTF-8 e codificados como hexadecimal.

# Parâmetros

Assim, a string às vezes é codificada como %C3%A0s+vezes (sendo que os bytes C3 e A0 representam os números 195 e 160 que, em codificação UTF-8, tornam-se a letra à).

Note que uma limitação dos query parameters é a impossibilidade de passar dados estruturados como parâmetro. Por exemplo, não há a capacidade de relacionar um query param com outro, levando o desenvolvedor a criar maneiras de contornar esta limitação. Por exemplo, suponha que seja necessário desenvolver uma pesquisa de clientes baseada em vários tipos de impostos devidos num certo período de tempo.

Esta pesquisa deveria fornecer:

- O nome do imposto.
- O período (data inicial e data final da busca).

Se esta pesquisa foi baseada em um único imposto, não há problema algum. Mas tome como base uma pesquisa baseada em uma lista de impostos.

Ela seria semelhante a:

```
/pessoas?imposto.1=IR&data.inicio.1=2011-01-01&data.inicio.2=2012-01-01
```

# Parâmetros

Onde a numeração, neste caso, seria o elemento agrupador dos dados, ou seja, um workaround para a limitação de dados estruturados. Quando há a necessidade de fornecer dados complexos, no entanto, é possível fazê-lo pelo corpo da requisição.

Algo como:

```
POST /clientes HTTP/1.1
Host: localhost:8080
Content-Type: text/xml
Content-Length: 93
```

```
<cliente>
  <nome>Alexandre</nome>
  <dataNascimento>2012-01-01</dataNascimento>
</cliente>
```

Quanto a este tipo de parâmetro, não há limitações. O servidor faz a interpretação dos dados a partir do fato de que esses parâmetros são os últimos do documento, e controla o tamanho da leitura utilizando o cabeçalho Content-Length.

# Cabeçalhos

Os cabeçalhos, em HTTP, são utilizados para trafegar todo o tipo de meta informação a respeito das requisições. Vários destes cabeçalhos são padronizados, no entanto, eles são facilmente extensíveis para comportar qualquer particularidade que uma aplicação possa requerer nesse sentido.

Por exemplo, ao realizar uma requisição, pelo Firefox, para o site <http://www.casadocodigo.com.br/>, as seguintes informações são enviadas:

```
GET / HTTP/1.1
Host: www.casadocodigo.com.br
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:19.0) Gecko/20100101 Fire
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

# Cabeçalhos

Portanto, os seguintes cabeçalhos são enviados: Host, User-Agent, Accept, Accept-Language, Accept-Encoding e Connection. Cada um desses tem um significado para o servidor, no entanto, o protocolo HTTP não exige nenhum deles. Ao estabelecer esta liberdade, tanto o cliente quanto o servidor estão livres para negociar o conteúdo da maneira como acharem melhor.

Obviamente, isto não quer dizer que estes cabeçalhos não são padronizados. Da lista acima, cada cabeçalho possui uma explicação:

- **Host:** mostra qual foi o DNS utilizado para chegar a este servidor.
- **User-Agent:** fornece informações sobre o meio utilizado para acessar este endereço.
- **Accept:** realiza negociação com o servidor a respeito do conteúdo aceito.
- **Accept-Language:** negocia com o servidor qual o idioma a ser utilizado na resposta.
- **Accept-Encoding:** negocia com o servidor qual a codificação a ser utilizada na resposta.
- **Connection:** ajusta o tipo de conexão com o servidor (persistente ou não).

# Media Types

Ao realizar uma requisição para o site `http://www.casadocodigo.com.br`, o Media Type `text/html` é utilizado, indicando para o navegador qual é o tipo da informação que está sendo trafegada (no caso, HTML). Isto é utilizado para que o cliente saiba como trabalhar com o resultado (e não com tentativa e erro, por exemplo), dessa maneira, os Media Types são formas padronizadas de descrever uma determinada informação.

Os Media Types são divididos em tipos e subtipos, e acrescidos de parâmetros (se houverem). São compostos com o seguinte formato: `tipo/subtipo`. Se houver parâmetros, o `;` (ponto e vírgula) será utilizado para delimitar a área dos parâmetros. Portanto, um exemplo de Media Type seria `text/xml; charset="utf-8"` (para descrever um XML cuja codificação seja UTF-8).

Os tipos mais comuns são:

- Application
- Audio
- Image
- Text
- Video
- vnd

# Media Types

Cada um desses tipos é utilizado com diferentes propósitos. `Application` é utilizado para tráfego de dados específicos de certas aplicações. `Audio` é utilizado para formatos de áudio. `Image` é utilizado para formatos de imagens. `Text` é utilizado para formatos de texto padronizados ou facilmente inteligíveis por humanos. `Video` é utilizado para formatos de vídeo. `Vnd` é para tráfego de informações de softwares específicos (por exemplo, o Microsoft Office).

Em serviços `REST`, vários tipos diferentes de Media Types são utilizados. As maneiras mais comuns de representar dados estruturados, em serviços REST, são via `XML` e `JSON`, que são representados pelos Media Types `application/xml` e `application/json`, respectivamente. O XML também pode ser representado por `text/xml`, desde que possa ser considerado legível por humanos.

Note que subtipos mais complexos do que isso podem ser representados com o sinal `+` (mais). Este sinal é utilizado em vários subtipos para delimitação de mais de um subtipo. Este é o caso com `XHTML`, por exemplo, que denota o tipo HTML acrescido das regras de XML, e cujo media type é `application/xhtml+xml`. Outro caso comum é o do protocolo `SOAP`, cujo media type é `application/soap+xml`.

Os Media Types são negociados a partir dos cabeçalhos `Accept` e `Content-Type`. O primeiro é utilizado em requisições, e o segundo, em respostas.



# Media Types

Ao utilizar o cabeçalho Accept, o cliente informa ao servidor qual tipo de dados espera receber. Caso seja o tipo de dados que não possa ser fornecido pelo servidor, o mesmo retorna o código de erro 415, indicando que o Media Type não é suportado. Se o tipo de dados existir, então os dados são fornecidos e o cabeçalho Content-Type apresenta qual é o tipo de informação fornecida.

Existem várias formas de realizar esta solicitação. Caso o cliente esteja disposto a receber mais de um tipo de dados, o cabeçalho Accept pode ser utilizado para esta negociação. Por exemplo, se o cliente puder receber qualquer tipo de imagem, esta solicitação pode utilizar um curinga, representado por \* (asterisco). O servidor irá converter esta solicitação em um tipo adequado. Esta negociação será similar à seguinte:

## Requisição:

```
GET /foto/1 HTTP/1.1
Host: brejaonline.com.br
Accept: image/*
```

## Resposta:

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 10248
```

# Media Types

Além disso, também é possível solicitar mais de um tipo de dados apenas separando-os por, (vírgula). Por exemplo, um cliente de uma página web poderia realizar uma solicitação como a seguinte:

```
GET / HTTP/1.1
Host: brejaonline.com.br
Accept:text/html,*/*
```

Note, assim, que o cliente solicita uma página HTML que se uma não estiver disponível, qualquer tipo de conteúdo é aceito, como pode ser observado pela presença de dois curingas, em tipo e subtipo de dados. Neste caso, a prioridade atendida é a da especificidade, ou seja, como text/html é mais específico, tem mais prioridade.

# Media Types

Em caso de vários tipos de dados que não atendam a esta regra, o parâmetro **q** pode ser adotado para que se possa realizar diferenciações de peso. Este parâmetro recebe valores variando entre 0.1 e 1, contendo a ordem de prioridade. Por exemplo, suponha a seguinte requisição:

```
GET / HTTP/1.1  
Host: brejaonline.com.br  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

Isto significa que tanto HTML quanto XHTML têm prioridade máxima. Na falta de uma representação do recurso com qualquer um desses tipos de dados, o XML é aceito com prioridade 90%. Na ausência do XML, qualquer outra representação é aceita, com prioridade 80%.

# Código de Status

Toda requisição que é enviada para o servidor retorna um código de status. Esses códigos são divididos em cinco famílias: 1xx, 2xx, 3xx, 4xx e 5xx, sendo:

- 1xx - Informativos.
- 2xx - Códigos de sucesso.
- 3xx - Códigos de redirecionamento.
- 4xx - Erros causados pelo cliente.
- 5xx - Erros originados no servidor.



# Código de Status

De acordo com Leonard Richardson e Sam Ruby, os códigos mais importantes e mais utilizados são:

## 2xx

### 200 - OK

Indica que a operação indicada teve sucesso.

### 201 - Created

Indica que o recurso desejado foi criado com sucesso. Deve retornar um cabeçalho Location, que deve conter a URL onde o recurso recém-criado está disponível.

### 202 - Accepted

Indica que a solicitação foi recebida e será processada em outro momento. É tipicamente utilizada em requisições assíncronas, que não serão processadas em tempo real. Por esse motivo, pode retornar um cabeçalho Location, que trará uma URL onde o cliente pode consultar se o recurso já está disponível ou não.

### 204 - No Content

Usualmente enviado em resposta a uma requisição PUT, POST ou DELETE, onde o servidor pode recusar-se a enviar conteúdo.

### 206 - Partial Content

Utilizado em requisições GET parciais, ou seja, que demandam apenas parte do conteúdo armazenado no servidor (caso muito utilizado em servidores de download).

# Código de Status

## 3xx

### 301 - Moved Permanently

Significa que o recurso solicitado foi realocado permanentemente. Uma resposta com o código 301 deve conter um cabeçalho Location com a URL completa (ou seja, com descrição de protocolo e servidor) de onde o recurso está atualmente.

### 303 - See Other

É utilizado quando a requisição foi processada, mas o servidor não deseja enviar o resultado do processamento. Ao invés disso, o servidor envia a resposta com este código de status e o cabeçalho Location, informando onde a resposta do processamento está.

### 304 - Not Modified

É utilizado, principalmente, em requisições GET condicionais - quando o cliente deseja ver a resposta apenas se ela tiver sido alterada em relação a uma requisição anterior.

### 307 - Temporary Redirect

Similar ao 301, mas indica que o redirecionamento é temporário, não permanente.

# Código de Status

## 4xx

### 400 - Bad Request

É uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente do serviço.

### 401 - Unauthorized

Utilizado quando o cliente está tentando realizar uma operação sem ter fornecido dados de autenticação (ou a autenticação fornecida for inválida).

### 403 - Forbidden

Utilizado quando o cliente está tentando realizar uma operação sem ter a devida autorização.

### 404 - Not Found

Utilizado quando o recurso solicitado não existe.

### 405 - Method Not Allowed

Utilizado quando o método HTTP utilizado não é suportado pela URL. Deve incluir um cabeçalho Allow na resposta, contendo a listagem dos métodos suportados (separados por ",").

# Código de Status

## 4xx

### 409 - Conflict

Utilizado quando há conflitos entre dois recursos. Comumente utilizado em resposta a criação de conteúdos que tenham restrições de dados únicos - por exemplo, criação de um usuário no sistema utilizando um login já existente. Se for causado pela existência de outro recurso (como no caso citado), a resposta deve conter um cabeçalho Location, explicitando a localização do recurso que é a fonte do conflito.

### 410 - Gone

Semelhante ao 404, mas indica que um recurso já existiu neste local.

### 412 - Precondition failed

Comumente utilizado em resposta a requisições GET condicionais.

### 415 - Unsupported Media Type

Utilizado em resposta a clientes que solicitam um tipo de dados que não é suportado, por exemplo, solicitar JSON quando o único formato de dados suportado é XML.



# Código de Status

## 5xx

### 500 - Internal Server Error

É uma resposta de erro genérica, utilizada quando nenhuma outra se aplica.

### 503 - Service Unavailable

Indica que o servidor está atendendo requisições, mas o serviço em questão não está funcionando corretamente. Pode incluir um cabeçalho Retry-After, dizendo ao cliente quando ele deveria tentar submeter a requisição novamente.

# Conclusão

Você pôde conferir, nesta seção, os princípios básicos do `protocolo HTTP`. Estes princípios são o uso de `cabeçalhos`, `métodos`, `códigos de status` e `formatos de dados` que formam a base do uso eficiente de `REST`.

Você irá conferir, nas próximas seções, como REST se beneficia dos princípios deste protocolo de forma a tirar o máximo proveito desta técnica.



**XML**

eXtensible Markup Language

# XML

XML é uma sigla que significa eXtensible Markup Language, ou linguagem de marcação extensível. Por ter esta natureza extensível, conseguimos expressar grande parte de nossas informações utilizando este formato. XML é uma linguagem bastante semelhante a HTML (HyperText Markup Language), porém, com suas próprias particularidades. Por exemplo, todo arquivo XML tem um e apenas um elemento-raiz (assim como HTML), com a diferença de que este elemento-raiz é flexível o bastante para ter qualquer nome.

Além disso, um XML tem seções específicas para fornecimento de instruções de processamento, ou seja, seções que serão interpretadas por processadores de XML que, no entanto, não fazem parte dos dados. Estas seções recebem os nomes de prólogo (quando estão localizadas antes dos dados) e epílogo, quando estão localizadas depois. Por exemplo, é comum encontrar um prólogo que determina a versão do XML e o charset utilizado.

Este prólogo tem o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

# XML

Ou seja, diferente de tags regulares de XML (que têm o formato `<tag></tag>`), uma instrução de processamento tem o formato `<?nome-da-instrução ?>`. No exemplo, eu estou especificando a versão de XML utilizada (1.0) e o charset (UTF-8).

Afora esta informação, as estruturas mais básicas em um XML são as tags e os atributos, de forma que um XML simples tem o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tag atributo="valor">conteúdo da tag</tag>
```

Como observado no exemplo, os dados transportados pela sua aplicação podem estar presentes tanto na forma de atributos como de conteúdo das tags. Em um exemplo mais próximo do caso de cervejaria, uma cerveja pode ser transmitida da seguinte forma:

```
<?xml version="1.0" encoding="UTF-8" ?>
<cerveja id="1">
  <nome>Stella Artois</nome>
</cerveja>
```

# Conhecendo XML Schema

Por ser um formato amplamente utilizado por diversos tipos de aplicação, XML contém diversos tipos de utilitários para vários fins, a saber:

- Validação de formato e conteúdo.
- Busca de dados.
- Transformação.

A ferramenta XML mais utilizada quando se trata de XML são os XML Schemas. Trata-se de arquivos capazes de descrever o formato que um determinado XML deve ter (lembre-se, XML é flexível a ponto de permitir qualquer informação). Um XML Schema, como um todo, é análogo à definição de classes em qualquer linguagem de programação: define-se os pacotes (que, em um XML Schema, são definidos como namespaces) e as classes, propriamente ditas (que, em XML Schemas, são os tipos).

# Conhecendo XML Schema

Utiliza-se XML Schemas para que tanto o cliente quanto o servidor tenha um “acordo” a respeito do que enviar/receber (em termos da estrutura da informação).

Por exemplo, suponha que cada uma das cervejarias possua um ano de fundação, assim:

```
<cervejaria>  
  <fundacao>1850</fundacao>  
</cervejaria>
```

Se o cliente não tiver uma referência a respeito do que enviar, ele pode enviar os dados assim:

```
<cervejaria>  
  <anoFundacao>1850</anoFundacao>  
</cervejaria>
```

# Conhecendo XML Schema

Note que, desta forma, a informação não seria inteligível do ponto de vista do cliente e/ou do serviço. Assim, para manter ambos cientes do formato a ser utilizado, pode-se utilizar um **XML Schema**.

Um XML Schema simples pode ser definido da seguinte forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://brejaonline.com.br/comum/v1"
  xmlns:tns="http://brejaonline.com.br/comum/v1">
</schema>
```

Um XML Schema é sempre definido dentro da tag schema. Esta tag deve conter, obrigatoriamente, a referência para o XML Schema <http://www.w3.org/2001/XMLSchema> e o atributo targetNamespace, que aponta qual deve ser o namespace utilizado pelo XML que estiver sendo validado por este XML Schema. Além disso, por convenção, o próprio namespace é referenciado no documento através da declaração xmlns:tns, indicando que o prefixo tns poderá ser utilizado no escopo deste XML Schema. Esta prática não é obrigatória, mas é sempre recomendada.



# Conhecendo XML Schema

A estrutura dos dados, em XML Schemas, são definidas em termos de elementos. Os elementos são utilizados para definir informações a respeito das tags: nome da tag, número de repetições permitido, quais sub-tags são permitidas, quais atributos uma tag deve ter, etc.

Para definir um elemento dentro de um XML Schema, basta utilizar a tag `element`. Por exemplo, para definir uma tag nome em um XML Schema, basta utilizar o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://brejaonline.com.br/comum/v1"
  xmlns:tns="http://brejaonline.com.br/comum/v1">
  <element name="nome" type="string" />
</schema>
```

# Conhecendo XML Schema

Conforme mencionado anteriormente, a definição dos dados propriamente ditos é feita através de tipos. Estes tipos são divididos entre simples e complexos, sendo os simples aqueles que são strings, datas, números ou derivados destes, já os complexos, são definidos a partir da junção de elementos de tipos simples e/ou outros tipos complexos.

Tipos novos podem ser criados ou estendidos à vontade, sejam estes tipos simples ou complexos. Por exemplo, para criar um tipo simples novo, basta utilizar o seguinte:

```
<simpleType name="CEP" />
```

Neste mesmo schema, um tipo complexo pode ser criado da seguinte forma:

```
<complexType name="Endereco">
  <sequence>
    <element name="cep" type="tns:CEP" />
    <element name="logradouro" type="string" />
  </sequence>
</complexType>
```

# Conhecendo XML Schema

Note que a definição do tipo complexo envolve a tag sequence. Esta tag é utilizada para determinar que os elementos nela envolvidos devem ser inseridos nesta ordem.

Por exemplo, ao implementar este tipo complexo, o seguinte é aceito:

```
<endereco>  
  <cep>12345-678</cep>  
  <logradouro>Rua das cervejas</logradouro>  
</endereco>
```

Porém, o inverso não é aceito:

```
<endereco>  
  <logradouro>Rua das cervejas</logradouro>  
  <cep>12345-678</cep>  
</endereco>
```

# Tipos Simples de XML

Os tipos simples, como dito antes, oferecem extensibilidade em relação a outros tipos simples. Esta extensão pode ser feita para os fins mais diversos, sendo que o uso mais comum desta facilidade é para prover restrições sobre estes tipos simples.

Por exemplo, no caso do CEP, sabemos que este segue uma estrutura bem clara de validação, que é o padrão **cinco dígitos (traço) três dígitos**. Ou seja, uma expressão regular pode ser utilizada para esta validação.

Esta restrição é representada utilizando-se a tag **restriction**. Esta tag, por sua vez, possui um atributo base, que é utilizado para indicar qual será o “tipo pai” a ser utilizado por este dado. Por exemplo, no caso de um CEP, o tipo a ser utilizado como tipo pai será string. Assim, o elemento CEP pode ser representado da seguinte forma:

```
<simpleType name="CEP">  
  <restriction base="string">  
  </restriction>  
</simpleType>
```

# Tipos Simples de XML

Finalmente, dentro da tag `restriction` é que podemos definir o tipo de restrição que será aplicada ao utilizar este tipo. Podemos definir uma série de restrições, como enumerações de dados (ou seja, só é possível utilizar os valores especificados), comprimento máximo da string, valores máximos e mínimos que números podem ter, etc. Como queremos utilizar uma expressão regular, utilizamos a tag `pattern`, que define um atributo `value`. Através deste atributo, especificamos a expressão regular.

```
<simpleType name="CEP">
  <restriction base="string">
    <pattern value="\d{5}-\d{3}" />
  </restriction>
</simpleType>
```

**Expressões Regulares:** Expressões regulares são utilizadas para determinar formatos que certas strings devem ter. Por exemplo, estas expressões podem ser utilizadas para validar endereços de e-mail, números de cartão de crédito, datas, etc. Por ser um assunto complexo, me limito aqui a apenas explicar o significado do padrão acima.

O símbolo `\d` significa um dígito (qualquer um). Ao ter o sinal `{5}` anexado, indica que cinco dígitos são aceitos. O traço representa a si próprio.

# Tipos Complexos de XML

Como apresentado anteriormente, um tipo complexo, em um XML Schema, é semelhante a uma classe em qualquer linguagem de programação. Por exemplo, a definição de um endereço pode ser feita da seguinte forma:

```
<complexType name="Endereco">
  <sequence>
    <element name="CEP" type="tns:CEP" />
    <element name="logradouro" type="string" />
  </sequence>
</complexType>
```

Note a referência ao tipo "CEP", definido anteriormente. O prefixo `tns`, conforme mencionado, faz referência ao namespace do próprio arquivo (semelhante a uma referência a um pacote Java que, no entanto, possui uma forma abreviada, que é o prefixo).

Os tipos complexos comportam diversas facilidades, como herança e definição de atributos. Por exemplo, considere o tipo complexo `Pessoa`:

```
<complexType name="Pessoa"> </complexType>
```

# Tipos Complexos de XML

Supondo que uma Pessoa seja uma super definição para uma pessoa física ou jurídica, queremos que o tipo Pessoa seja abstrato. Assim, podemos definir o atributo `abstract`:

```
<complexType name="Pessoa" abstract="true"> </complexType>
```

Note que, desta forma, não haverá uma implementação do que for definido neste tipo, apenas de seus subtipos. Para criar uma extensão deste tipo, utiliza-se as tags `complexContent` e `extension`, assim:

```
<complexType name="PessoaFisica">  
  <complexContent>  
    <extension base="tns:Pessoa">  
    </extension>  
  </complexContent>  
</complexType>
```

# Tipos Complexos de XML

Assim, é possível definir elementos tanto no super tipo quanto no subtipo. Por exemplo, considere o seguinte:

```
<complexType name="Pessoa" abstract="true">
  <sequence>
    <element name="nome" type="string" />
  </sequence>
</complexType>

<complexType name="PessoaFisica">
  <complexContent>
    <extension base="tns:Pessoa">
      <sequence>
        <element name="cpf" type="tns:CPF" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```



# Tipos Complexos de XML

Assim, a implementação deste tipo pode ficar assim:

```
<peessoaFisica>  
  <nome>Alexandre</nome>  
  <cpf>123.456.789-09</cpf>  
</peessoaFisica>
```

Além disso, é possível definir atributos nos tipos complexos. Por exemplo, considere o seguinte:

```
<complexType name="Pessoa" abstract="true">  
  <attribute name="id" type="long" />  
</complexType>
```

Assim, a implementação deste tipo ficaria assim:

```
<peessoaFisica id="1" />
```

# Tipos Complexos de XML

Um XML Schema também pode importar outros, notavelmente para realizar composições entre vários tipos diferentes. Por exemplo, considere as duas definições de XML Schemas:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/endereco/v1"
  elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://brejaonline.com.br/endereco/v1">

  <simpleType name="CEP">
    <restriction base="string">
      <pattern value="\d{5}-\d{3}" />
    </restriction>
  </simpleType>

  <complexType name="Endereco">
    <sequence>
      <element name="cep" type="tns:CEP" />
      <element name="logradouro" type="string" />
    </sequence>
  </complexType>
</schema>
```

# Tipos Complexos de XML

Note que este XML Schema possui o `targetNamespace` definido como <http://brejaonline.com.br/endereco/v1>.

Agora, considere um segundo XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/pessoa/v1"
  elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://brejaonline.com.br/pessoa/v1">

  <simpleType name="CPF">
    <restriction base="string">
      <pattern value="\d{3}\.\d{3}\.\d{3}-\d{2}" />
    </restriction>
  </simpleType>

  <complexType name="Pessoa" abstract="true">
    <sequence>
      <element name="nome" type="string" />
    </sequence>
    <attribute name="id" type="long" />
  </complexType>

  <complexType name="PessoaFisica">
    <complexContent>
      <extension base="tns:Pessoa">
        <sequence>
          <element name="cpf" type="tns:CPF" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</schema>
```

# Tipos Complexos de XML

Note que, neste segundo XML Schema, o namespace é `http://brejaonline.com.br/pessoa/v1` (ou seja, diferente do namespace de endereços).

Para utilizar o XML Schema de endereços no XML Schema de pessoas, é necessário efetuar dois passos: o primeiro, é definir um prefixo para o namespace de endereços. Por exemplo, para definir o prefixo `end`, utiliza-se o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/pessoa/v1"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://brejaonline.com.br/pessoa/v1"
  xmlns:end="http://brejaonline.com.br/endereco/v1">

  <!-- restante -->
</schema>
```

# Tipos Complexos de XML

O próximo passo é informar ao mecanismo a localização do outro XML Schema, através da tag `import`:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/pessoa/v1"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://brejaonline.com.br/pessoa/v1"
  xmlns:end="http://brejaonline.com.br/endereco/v1">

  <import namespace="http://brejaonline.com.br/endereco/v1"
    schemaLocation="Endereco.xsd" />

  <!-- restante -->
</schema>
```

Assim, para utilizar os tipos definidos no novo arquivo, basta utilizar o prefixo `end`:

```
<complexType name="Pessoa">
  <sequence>
    <element name="endereco" type="end:Endereco" />
  </sequence>
</complexType>
```

# Tipos Complexos de XML

Da mesma forma, é possível utilizar a tag `maxOccurs` para determinar o número máximo de ocorrências (ou seja, que um dado elemento representa uma lista).

O número máximo de ocorrências pode ser delimitado a partir de um número fixo ou, caso não haja um limite definido, utiliza-se o valor `unbounded`. Assim, para determinar que uma pessoa possui vários endereços, pode-se utilizar o seguinte:

```
<complexType name="Pessoa">
  <sequence>
    <element name="endereco" type="end:Endereco" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

# Tipos Complexos de XML

Isso provoca o seguinte resultado:

```
<peessoaFisica>
  <endereco>
    <cep>12345-678</cep>
    <logradouro>Rua Um</logradouro>
  </endereco>
  <endereco>
    <cep>87654-321</cep>
    <logradouro>Rua Dois</logradouro>
  </endereco>
</peessoaFisica>
```

A este ponto, você deve ter notado que o elemento raiz `peessoaFisica` não foi especificado. O elemento raiz é definido fora de tipos em um XML Schema, utilizando a tag `element`. Desta forma, é possível ter o seguinte:

```
<complexType name="PessoaFisica">
  <!-- definição de pessoa fisica -->
</complexType>

<element name="peessoaFisica" type="tns:PessoaFisica" />
```

# Validação de XML

Para realizar os testes de validação do XML, é possível utilizar o site disponível no endereço <https://dfe-portal.svrs.rs.gov.br/Nfe/ValidadorXML> e verificar o retorno do PARSER.

## Validador Xml

Cole aqui o conteúdo da mensagem

```
<peessoaFisica>
  <endereco>
    <cep>12345-678</cep>
    <logradouro>Rua Um</logradouro>
  </endereco>

  <endereco>
    <cep>87654-321</cep>
    <logradouro>Rua Dois</logradouro>
  </endereco>
</peessoaFisica>
```

PARSER XML: OK 





# JSON

JavaScript Object Notation

# JSON

JSON é uma sigla para JavaScript Object Notation. É uma linguagem de marcação criada por Douglas Crockford e descrito na RFC 4627, e serve como uma contrapartida ao XML.

Tem por principal motivação o tamanho reduzido em relação a XML, e acaba tendo uso mais propício em cenários onde largura de banda (ou seja, quantidade de dados que pode ser transmitida em um determinado intervalo de tempo) é um recurso crítico.

Atende ao seguinte modelo:

- Um objeto contém zero ou mais membros.
- Um membro contém zero ou mais pares e zero ou mais membros.
- Um par contém uma chave e um valor.
- Um membro também pode ser um array.

# JSON

O formato desta definição é o seguinte:

```
{
  "nome do objeto" : {
    "nome do par" : "valor do par"
  }
}
```

Por exemplo, a pessoa física pode ser definida da seguinte maneira:

```
{
  "pessoaFisica" : {
    "nome" : "Alexandre",
    "cpf" : "123.456.789-09"
  }
}
```

# JSON

Caso seja uma listagem, o formato é o seguinte:

```
{
  "nome do objeto" : [
    { "nome do elemento" : "valor" },
    { "nome do elemento" : "valor" }
  ]
}
```

Novamente, a pessoa física pode ser definida da seguinte maneira:

```
{
  "pessoaFisica" : {
    "nome" : "Alexandre",
    "endereco" : [
      {
        "cep" : "12345-678",
        "logradouro" : "Rua Um"
      }
    ],
    "cpf" : "123.456.789-09"
  }
}
```

# Validação de JSON

Assim como em XML, JSON também possui um sistema de validação, conhecido como JSON Schema. Trata-se de um tipo de arquivo que, como XML Schemas, também possui um formato próprio para especificação de tipos JSON. Essa especificação está disponível no seu site,

Um JSON Schema possui como declaração mais elementar o título (descrito no JSON Schema como uma propriedade title). O título é o que denomina este arquivo de validação. Por exemplo, supondo que desejamos começar a descrever um JSON Schema para pessoas, o descritivo terá o seguinte formato:

```
{  
  "title": "Pessoa"  
}
```

# Validação de JSON

Na sequência, é necessário definir, para este formato, o tipo (descrito no JSON Schema como uma propriedade type). O tipo pode ser um dos seguintes:

- `array` - uma lista de dados.
- `boolean` - um valor booleano (`true` ou `false`).
- `integer` - um número inteiro qualquer. Note que JSON não define um número de bits possível, apenas o fato de ser um número pertencente ao conjunto dos números inteiros.
- `number` - um número pertencente ao conjunto dos números reais. Segue a mesma regra de `integer`, sendo que `number` também contempla `integers`.
- `null` - um valor nulo.
- `object` - um elemento que contém um conjunto de propriedades.
- `string` - uma cadeia de caracteres.

# Validação de JSON

Como pessoa é um objeto, podemos definir o tipo da seguinte forma:

```
{  
  "title": "Pessoa",  
  "type": "object"  
}
```

Obviamente, um objeto precisa conter propriedades. Estas podem ser definidas a partir da propriedade `properties`. Por exemplo, se quisermos definir uma propriedade `nome` para uma pessoa, podemos utilizar o seguinte formato:

```
{  
  "title": "Pessoa",  
  "type": "object",  
  "properties": {  
    "nome": {  
      "type": "string"  
    }  
  }  
}
```

# Validação de JSON

Note que, definida a propriedade nome, o conteúdo deste passa a ter o mesmo formato da raiz pessoa, ou seja, existe uma recursividade. É possível, portanto, definir um atributo title dentro de nome, definir nome como um objeto com subpropriedades, etc.

Assim sendo, podemos, portanto, definir um elemento endereco dentro de pessoa, que também será um objeto. Dessa forma, temos:

```
{
  "title": "Pessoa",
  "type": "object",
  "properties": {
    "nome": {
      "type": "string"
    },
    "endereco": {
      "type": "object",
      "properties": {
        "cep": {
          "type": "string"
        },
        "logradouro": {
          "type": "string"
        }
      }
    }
  }
}
```



# Validação de JSON

Finalmente, vale a pena observar que, por padrão, o **JSON Schema** não te limita aos formatos pré-definidos (ao contrário do **XML Schema**, que apenas se atém ao definido). Por exemplo, de acordo com o JSON Schema acima, o seguinte JSON seria considerado válido:

```
{
  "nome": "Alexandre",
  "cpf": "123.456.789-09"
}
```

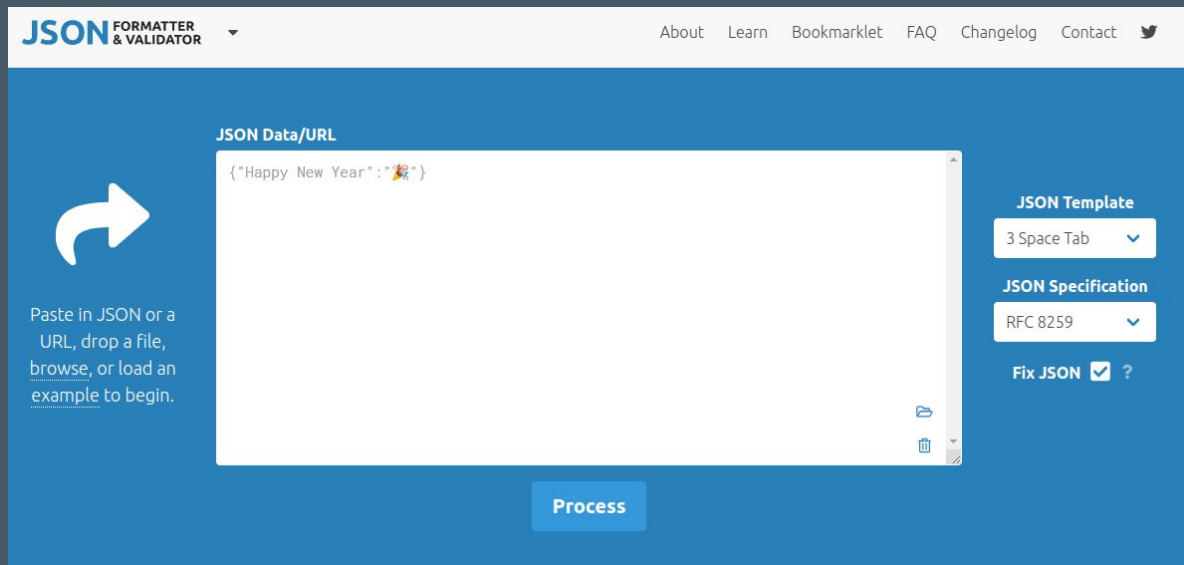
Note que, ainda que o campo `cpf` não esteja descrito, a presença do mesmo não invalida o JSON, tornando os campos meramente descritivos (ou seja, encarados como “esperados”). Caso este comportamento não seja desejável, é possível definir o atributo **`additionalProperties`** (que é do tipo booleano) como **`false`**, assim:

```
{
  "title": "Pessoa",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "nome": {
      "type": "string"
    },
    "endereco": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "cep": {
          "type": "string"
        },
        "logradouro": {
          "type": "string"
        }
      }
    }
  }
}
```

# Validação de JSON

Desta forma, o JSON descrito seria considerado inválido.

Para realizar os testes de validação JSON, é possível utilizar o site disponível no endereço <https://jsonformatter.curiousconcept.com/>



The screenshot shows the 'JSON FORMATTER & VALIDATOR' website. The header includes navigation links: About, Learn, Bookmarklet, FAQ, Changelog, and Contact, along with a Twitter icon. The main content area has a blue background. On the left, a large white arrow points right, with the text 'Paste in JSON or a URL, drop a file, browse, or load an example to begin.' below it. In the center, a text input field labeled 'JSON Data/URL' contains the JSON string: `{"Happy New Year": "🍷"}`. On the right, there are two dropdown menus: 'JSON Template' set to '3 Space Tab' and 'JSON Specification' set to 'RFC 8259'. Below these is a 'Fix JSON' checkbox which is checked, followed by a question mark icon. At the bottom center, there is a blue 'Process' button.

# Conclusão

Nesta seção, você conheceu mais a respeito dos formatos mais utilizados em REST, ou seja, XML e JSON. Você conheceu mais a respeito do ecossistema que os cerca, ou seja, no caso do XML, XML Schemas. No caso de JSON, os JSON Schemas.

Deste momento em diante, você já tem em mãos os meios que serão utilizados para construção de serviços REST, ou seja, você já conhece os princípios de REST e já sabe utilizar os tipos de dados disponíveis para utilização deste.



# Conceitos REST

Princípios e restrições  
para aplicações REST

# Conceito Fundamental

## ❏ Null Style

Em sua tese, Roy Fielding, defende a existência de duas abordagens para a elaboração de um design arquitetural.

A **primeira** abordagem é aquela onde um projeto de sistema começa “do zero”, como que em uma “tela em branco” e vai sendo elaborado utilizando elementos familiares até que, em dado momento, satisfaça as necessidades pretendidas.

A **segunda** abordagem parte de um sistema “pronto”, com todos os elementos demandados para atender uma determinada necessidade já presentes, porém sem restrições. Então, restrições são aplicadas para “aparar” os elementos do sistema até que ele satisfaça apenas a um conjunto de especificações bem determinado. Este “ponto de partida”, sem restrições, é designado por Roy Fielding como **Null Style**.

**REST, como estilo arquitetural, tem como Null Style a World Wide Web.**

# Princípios Básicos

Recapitulando **REST** é um estilo arquitetural para desenvolvimento de aplicações web, APIs e web services que teve origem na tese de doutorado de **Roy Fielding**. Este, por sua vez, é co-autor de um dos protocolos mais utilizados no mundo, o **HTTP** (HyperText Transfer Protocol). Assim, é notável que o protocolo REST é guiado (dentre outros preceitos) pelo que seriam as boas práticas de uso de HTTP:

- ❖ Uso adequado dos métodos HTTP;
- ❖ Uso adequado de URL's;
- ❖ Uso de códigos de status padronizados para representação de sucessos ou falhas;
- ❖ Uso adequado de cabeçalhos HTTP;
- ❖ Interligações entre vários recursos diferentes.

# Recurso

Um **recurso** é qualquer artefato na rede que tenha uma **identidade**. Exemplos comuns são documentos eletrônicos, imagens, serviços ou, eventualmente, coleção de outros recursos.

**URI**, acrônimo para **Uniform Resource Identifier**, é uma identificação simples para um recurso. Enquanto isso, **URL**, acrônimo para **Uniform Resource Locator**, especifica também como este recurso (indicando protocolo) deve ser acessado.

“Todas URLs são URIs, mas nem toda URI é uma URL.”

# Recurso

Tomando como exemplo o caso da listagem de clientes, é possível decompôr a URL utilizada para localização da listagem em várias partes:

<http://localhost:8080/cervejaria/clientes>

- **http://** - Indica o protocolo que está sendo utilizado (no caso, HTTP);
- **localhost:8080** - Indica o servidor de rede que está sendo utilizado e a porta (quando a porta não é especificada, assume-se que é a padrão, no caso do protocolo HTTP, 80);
- **cervejaria** - Indica o contexto da aplicação, ou seja, a raiz pela qual a aplicação está sendo fornecida para o cliente. Vou me referir a esta, daqui em diante, como contexto da aplicação ou apenas contexto;
- **clientes** - É o endereço, de fato, do recurso, no caso, a listagem de clientes.



# Responsabilidades no REST

Existe no REST um princípio chamado **STATELESS** (sem estado), onde o servidor não precisa saber em qual estado o cliente está e vice-versa. Mas o que é um servidor e um cliente?

**Cliente:** é o componente solicitante de um serviço e envia solicitações para vários tipos de serviços ao servidor.

**Servidor:** É o componente que é o provedor de serviços e fornece continuamente serviços ao cliente conforme as solicitações

Nesta arquitetura ou modelo, cliente-servidor ajuda na separação de responsabilidades entre a interface do usuário e o armazenamento de dados. Ou seja, quando uma solicitação REST é realizada, o servidor envia uma representação dos estados que foram requeridos.

Não há limite superior no número de clientes que podem ser atendidos por um único servidor. Também não é obrigatório que o cliente e o servidor residam em sistemas separados, porém é recomendável, para evitar o alto acoplamento entre os componentes.

# Elementos da Arquitetura REST

Ao projetar dados, um componente “servidor” pode-se adotar uma das seguintes três estratégias:

1. Renderizar os dados para uma representação específica;
2. Enviar os dados em sua representação nativa, acompanhados de código que realiza a renderização para uma representação específica;
3. Enviar os dados em sua representação nativa, acompanhados de metadados para que o componente “cliente” adote uma estratégia de processamento;

São elementos arquiteturais relacionados com dados:

- o próprio recurso;
- sua identificação, geralmente uma URI ou URN (Uniform Resource Name - identificação única ao recurso);
- alternativas para representação, como HTML, JSON e XML;
- metadados da representação, como cabeçalhos como media-type e last-modified;
- metadados do recurso, com cabeçalhos como alternates e vary;
- metadados de controle, com cabeçalhos como if-modified-since e cache-control;

# Elementos da Arquitetura REST

## ❑ Conectores

Os **conectores** são as tecnologias responsáveis por acessar e transportar os dados entre dois ou mais componentes. Basicamente, eles podem ser agrupados em tecnologias clientes, servidoras, para caching, transporte (tunnel) e resolução de nomes (ex: servidores DNS).

Os principais tipos de conectores são “cliente” e “servidor”. A diferença essencial entre os dois é que um cliente inicia a comunicação fazendo uma requisição, enquanto um servidor escuta as conexões e responde às requisições para fornecer acesso aos seus serviços.

## ❑ Componentes

Os **componentes** são elementos de processamento que se conectam a outros por meio de conectores. Os componentes mais comuns são o “cliente” e o “servidor”. Em um modelo em camadas, um mesmo componente pode ser, ao mesmo tempo, “cliente” e “servidor” realizando algum tipo de transformação ou segurança.

# Restrições

A **primeira restrição** definida por Fielding é que sistemas que seguem o estilo arquitetural REST sejam implementação cliente-servidor. Cabe ao componente “cliente” disponibilizar interfaces com o usuário. Enquanto isso, componentes “servidor” fornecem dados. Em sistemas RESTful, eventualmente, um mesmo componente de software poderá operar como cliente e servidor.

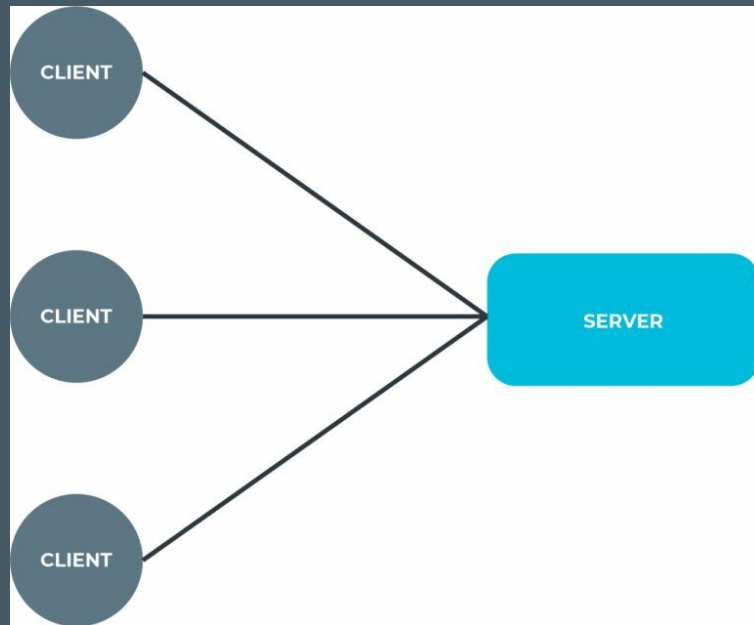


Um software navegador utilizado para navegar na internet atua como cliente. Afinal, envia requisições HTTP, para um servidor, de forma a acessar e manipular dados.

# Restrições

A separação de responsabilidades, segundo Fielding, reforça atributos de qualidade como portabilidade, permitindo que implementações “cliente” independentes sejam realizadas em diversas plataformas e a escalabilidade, simplificando os componentes “servidor”. Além disso, por habilitar que “cliente” e “servidor” evoluam de forma independente, colaboram para o evolvability (capacidade de evolução).

A **segunda restrição** definida por Fielding é que sistemas RESTful sejam **stateless**. Ou seja, cada requisição de componentes “cliente” deve conter todas as informações contextuais necessárias para serem atendidas, sem depender de qualquer dado de contexto armazenado no componente “servidor”. Assim, informações de sessão devem ser mantidas, inteiramente, apenas no componente “cliente”.



# Restrições

Por serem stateless, sistemas RESTful têm melhor visibilidade, confiabilidade e escalabilidade. Afinal:

- Cada solicitação pode ser verificada, completamente, de maneira isolada, já que contém todos os dados necessários para a análise;
- Por não manterem estado de contexto, é mais fácil recuperar o “lado servidor” de falhas parciais;
- Por não haver necessidade de manter estado de contexto, sistemas RESTful são mais fáceis de implementar e de escalar;

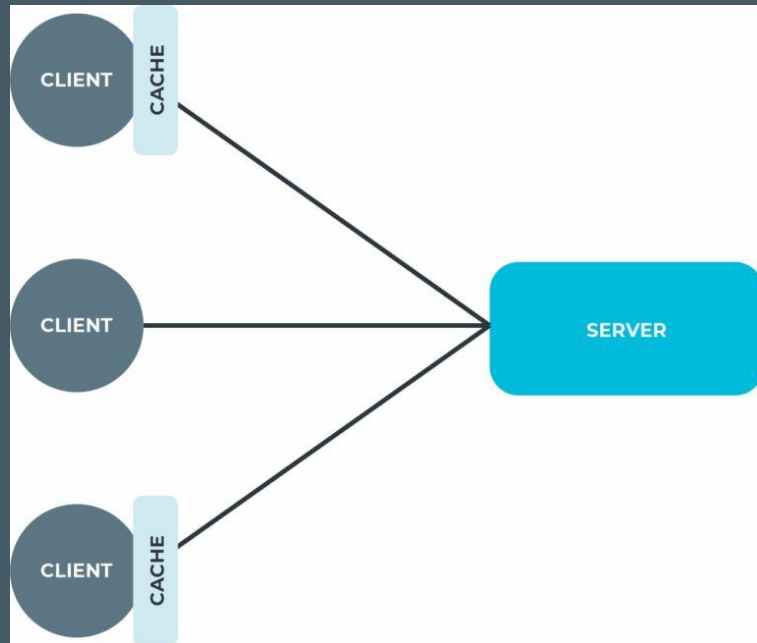
Como pontos potencialmente negativos, a restrição de que componentes “servidor” sejam stateless implica em tráfego de dados maior na rede, pelo transporte da informação de estado contexto. Além disso, há potencial excesso de autonomia dos componentes “cliente” (que operam sem supervisão do servidor).

# Restrições

A **terceira restrição** definida por Fielding é que sistemas RESTful devem indicar a possibilidade de adoção de **caching** nos componentes “cliente”. Para isso, todas as respostas do componente “servidor” devem ser identificadas, implícita ou explicitamente, como armazenáveis ou não armazenáveis em cache.

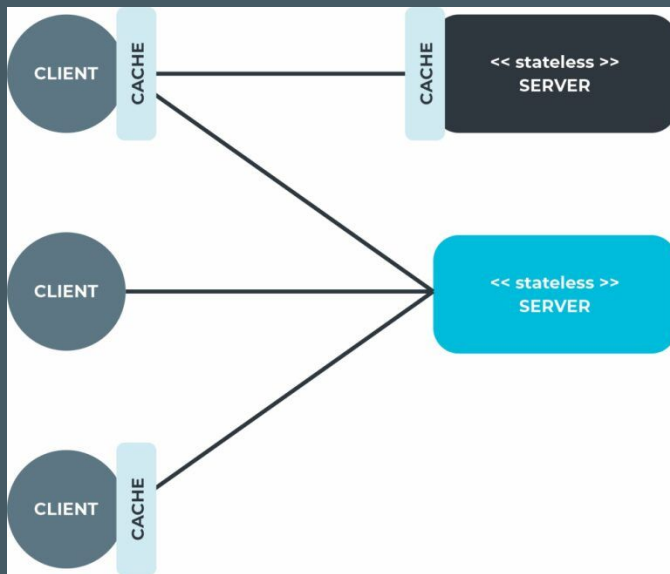
A adoção de caching melhora a eficiência, colaborando para a escalabilidade e desempenho.

Entretanto, dados armazenados em cache podem ficar desatualizados, antes de serem invalidados, comprometendo a confiabilidade.



# Restrições

A **quarta restrição** definida por Fielding é que as comunicações entre componentes “cliente” e “servidor”, em sistemas RESTful, ocorram através de interfaces uniformes (consistentes). Dessa forma, a implementação dos componentes “cliente” e “servidor” podem ocorrer de forma desacoplada (e independente).





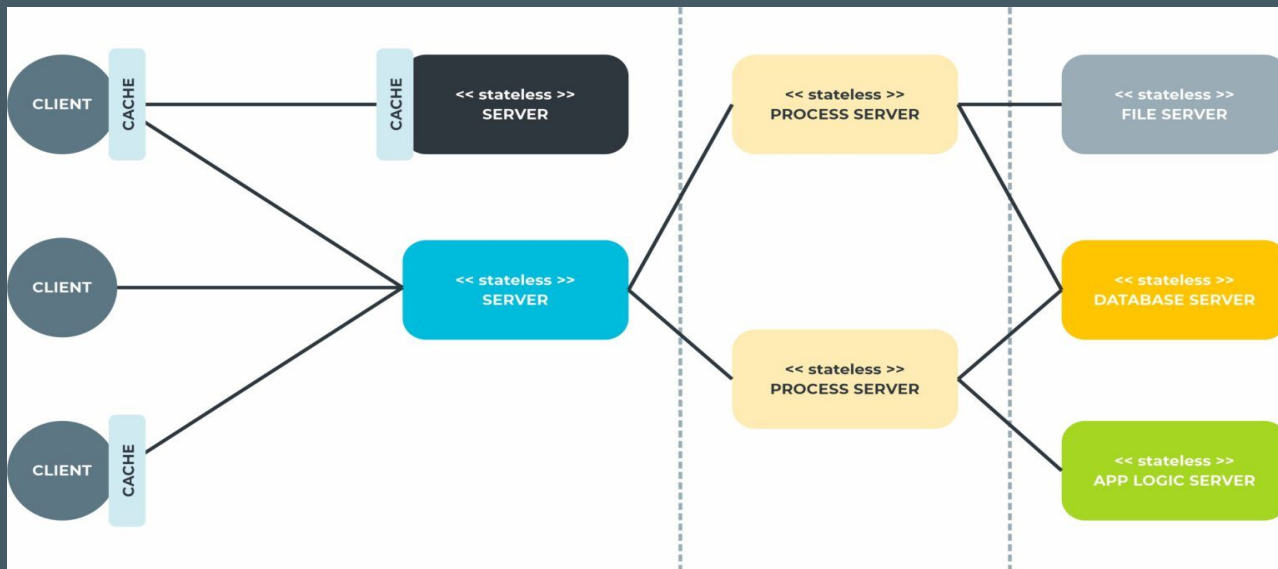
# Restrições

Há quatro restrições arquiteturais adicionais derivadas da decisão de adotar interfaces uniformes:

1. **Recursos devem ser identificados consistentemente.** Na prática, recursos em um sistema RESTful devem estar associados a identificações estáveis, ou seja, que não mudam durante as interações, tampouco quando o estado interno do recurso (conjunto de dados que o compõem) é alterado.
2. **Recursos devem ser manipulados a partir de representações.** A representação de um recurso é composta por uma sequência de bytes e metadados que a descrevem. Opcionalmente, uma representação também contém metadados que descrevem o recurso em si. Em implementações REST usando HTTP, por exemplo, entidades de domínio são recursos que podem ser expostos em representações usando XML, JSON e mais, enquanto isso, metadados descrevendo uma representação são fornecidos no cabeçalho;
3. **Mensagens devem ser auto-descritivas.** Ou seja, devem conter todas as informações para que o receptor cliente ou servidor tenha condições de realizar interpretação correta, sem necessitar de mensagens adicionais ou documentação em separado.
4. **Hipermídia deve ser utilizada como mecanismo de estado (HATEOAS).** Utilizando hiperlinks e, possivelmente, modelos consistentes de URI, como parte dos metadados que descrevem o recurso, seja para descrever documentos ligados como ativações de estado.

# Restrições

A quinta restrição definida por Fielding é que componentes “servidor” podem ser desenvolvidos a partir de composições que obedecem o estilo de arquitetura em camadas. Ou seja, componentes “cliente” tem acesso apenas a camada mais externa, e esta pode utilizar recursos de camadas inferiores, em diversas representações, para compor novos recursos.



# Restrições

Camadas podem ser utilizadas para encapsular sistemas legados, protegendo tanto novos “clientes” de “servidores” legados, como para proteger novos “servidores” de “clientes” legados.

Camadas diferentes podem operar, inclusive, em níveis de abstração distintos, prevenindo mudança de acoplamento. Por exemplo:

- Em nível mais baixo, expondo “entidades” como recursos nativos de sistemas pesados,
- Em nível intermediário, expondo recursos relacionados a processos de negócio que são composições das “entidades”
- Em nível mais alto, expondo recursos relacionados a “experiência” que são composições dos “processos”.

Finalmente, tecnicamente, camadas também podem ser utilizadas para melhorar a escalabilidade habilitando funcionalidades como balanceamento de carga, protegendo sistemas legados não preparados para escalabilidade.

# Restrições

A **sexta**, e última, **restrição** indicada por Fielding é que componentes “servidor” podem fornecer code-on-demand para componentes “cliente”. Ou seja, em sistemas RESTful componentes “servidor” podem estender funcionalidades de componentes “cliente” através do download de código executável na forma de applets ou scripts. Teoricamente, isso deve simplificar a implementação de features comuns com implementações pré-implementadas.

Hypermedia as the Engine of Application State, ou **HATEOAS**, é uma “maneira” de implementar APIs REST utilizando hipermídia para indicar que ações ou navegações estão disponíveis para um determinado recurso. Estas ações e a navegação são derivadas do estado do recurso e, eventualmente, da própria API. Elas são disponibilizadas para o cliente através de uma coleção de links.

```
{
  "customerId": 1,
  "firstName": "John",
  "lastName": "Doe",

  // ...

  "links": []
}
```

# Restrições

Não há uma definição universalmente aceita sobre onde esta coleção de links deve ser fornecida. Há quem defenda a inclusão na representação do recurso. Outros, defendem a utilização dos cabeçalhos da “response HTTP”. Quanto aos dados necessários em cada link, há a [RFC \(5598\)](#).

## ❑ O link para Self

O primeiro link da coleção costuma ser uma referência para o próprio objeto.

Dessa forma, caso a representação do recurso “se perca” na aplicação cliente, ou, caso ela, eventualmente, seja persistida ou compartilhada com outros contextos, em qualquer momento, seria possível determinar sua origem sem “inferências no código”.

```
{
  "customerid": 1,
  "firstName": "John",

  "links": [{
    "rel": "self",
    "method": "GET",
    "href": "/customers/1"
  }]
}
```

# Restrições

## ❏ Links para as ações suportadas pelo recurso

Para entender, efetivamente HATEOAS, devemos aplicar o mesmo raciocínio que [Don Norman](#) recomenda em [The Design of Everyday Things](#) para recursos. Ou seja, devemos perguntar, sempre, que operações um “recurso” espera.

No exemplo ao lado, por exemplo, temos um exemplo (obviamente simplificado) de um recurso representando uma conta bancária, cujo links relacionam as operações que este recurso suporta.

Abaixo, verificamos uma versão atualizada do recurso, com operações restritas em função do saldo negativo.

```
{
  "id": "123",
  "balance": {
    "currency": "usd",
    "amount": 150
  },

  "links": [{
    "rel": "self",
    "method": "GET",
    "href": "/accounts/123"
  }, {
    "rel": "withdrawals",
    "method": "GET",
    "href": "/accounts/123/withdrawals"
  }, {
    "rel": "withdrawals",
    "method": "POST",
    "href": "/accounts/123/withdrawals",
    "sample": {
      "currency": "usd",
      "amount": 0
    }
  }, {
    "rel": "deposits",
    "method": "GET",
    "href": "/accounts/123/deposits"
  }, {
    "rel": "deposits",
    "method": "POST",
    "href": "/accounts/123/deposits",
    "sample": {
      "currency": "usd",
      "amount": 0
    }
  }
}]
}
```

# Restrições

A presença da coleção de links direciona a composição da experiência do usuário direcionando que opções mostrar e quais não mostrar. Fornecer a lista de ações possíveis para um recurso na coleção de links permite que essa lógica seja implementada totalmente no serviço e simplifica muito a implementação da aplicação cliente.

```
{
  "id": "123",
  "balance": {
    "currency": "usd",
    "amount": 0
  },

  "links": [{
    "rel": "self",
    "method": "GET",
    "href": "/accounts/123"
  }, {
    "rel": "withdrawals",
    "method": "GET",
    "href": "/accounts/123/withdrawals"
  }, {
    "rel": "deposits",
    "method": "GET",
    "href": "/accounts/123/deposits"
  }, {
    "rel": "deposits",
    "method": "POST",
    "href": "/accounts/123/deposits",
    "sample": {
      "currency": "usd",
      "amount": 0
    }
  }
}]
}
```

# Restrições

## ❏ Links para navegação

Eventualmente, um recurso retornado pela API possuirá outros recursos relacionados. Por exemplo, em uma aplicação empresarial, um “cliente” possuirá uma coleção de “pedidos” que realizou.

```
{
  "customerid": 1,
  "links": [{
    "rel": "orders",
    "method": "GET",
    "href": "http://mydomain/customers/1/orders"
  }]
}
```



# Maturidade de Richardson

A arquitetura REST é relativamente nova, embora tenha ganhado bastante destaque nos últimos anos, sobretudo depois da popularização da arquitetura de microserviços. O problema é que nem sempre implementamos o padrão REST da forma que foi originalmente especificada por Roy Fielding, já que para ele uma API só pode ser considerada RESTful se de fato respeitar um conjunto de princípios e implementar uma série de restrições.

Para padronizar e facilitar o desenvolvimento de APIs REST, Leonard Richardson propôs um modelo de maturidade para esse tipo de API, definido em 4 níveis. E o objetivo dessa seção é justamente apresentar esse modelo de maturidade proposto por Leonard Richardson.

Ainda há uma discussão em relação a esse modelo, já que alguns defendem que para de fato ser considerada REST, uma API deve implementar os 4 níveis do modelo, outros consideram que implementando os 3 primeiros níveis, uma API já pode ser considerada RESTful. O objetivo desta seção não é entrar nessa discussão, queremos aqui apresentar os 4 níveis do modelo.

# Maturidade de Richardson

## ❑ Nível 0: HTTP

Esse é considerado o nível mais básico e uma API que implementa apenas esse nível não pode ser considerada REST. Nesse nível os nomes dos recursos não seguem qualquer padrão e estão sendo usados apenas para fazer invocação de métodos remotos. Nesse nível usamos o **protocolo HTTP para comunicação**, mas sem seguir qualquer tipo de regras para implementar os métodos.

Para facilitar o entendimento podemos observar no quadro abaixo a modelagem de uma API para um CRUD de clientes.

Verbo HTTP	URI	Operação
GET	/getClientes/1	Pesquisar
POST	/salvarCliente	Criar
POST	/alterarCliente/1	Alterar
GET/POST	/excluirCliente/1	Excluir

# Maturidade de Richardson

## ❏ Nível 1: HTTP + Recursos

Nesse nível fazemos uso de recursos para modelar a API, para representar cada **recurso** fazemos uso de **substantivos** no plural. No exemplo do CRUD de cliente, os recursos seriam identificados pelo substantivo “clientes”.

Um detalhe interessante é que no nível 1 já usamos os verbos HTTP de forma correta, já que se os verbos não fossem usados, as rotas de pesquisar, alterar e incluir ficariam ambíguas.

Verbo HTTP	URI	Operação
GET	/clientes/1	Pesquisar
POST	/clientes	Criar
PUT	/clientes/1	Alterar
DELETE	/clientes/1	Excluir

# Maturidade de Richardson

## ❏ Nível 2: HTTP + Recursos + Verbos

Como já foi adiantado no nível 1, o nível 2 se encarrega de **garantir que os verbos HTTP sejam usados de forma correta**. Os verbos mais utilizados são **GET**, **POST**, **PUT** e **DELETE**.

Os métodos GET, PUT e DELETE são considerados idempotente. Um método é considerado idempotente quando uma requisição idêntica pode ser executada várias vezes sem alterar o estado do servidor.

Verbo HTTP	Função
GET	Recuperar dados
POST	Gravar dados
PUT	Alterar dados
DELETE	Excluir dados

# Maturidade de Richardson

## ❏ Nível 3: HTTP + Recursos + Verbos + HATEOAS

O nível 3 é sem dúvidas o menos explorado, muitas APIs existentes no mercado não implementam esse nível.

HATEOAS significa **H**ypermedia **a**s **t**he **E**ngine **o**f **A**pplication **S**tate (A hipermídia como o motor do estado de aplicação). Uma API que implementa esse nível fornece aos seus clientes links que indicarão como poderá ser feita a navegação entre seus recursos. Ou seja, quem for consumir a API precisará saber apenas a rota principal e a resposta dessa requisição terá todas as demais rotas possíveis.

```
{
  "id": 1,
  "nome": "John",
  "sobrenome": "Doe",
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/clientes/1"
    },
    {
      "rel": "alterar",
      "href": "http://localhost:8080/clientes/1"
    },
    {
      "rel": "excluir",
      "href": "http://localhost:8080/clientes/1"
    }
  ]
}
```

# Maturidade de Richardson

## ❏ Nível 3: HTTP + Recursos + Verbos + HATEOAS

No exemplo acima, podemos ver o resultado de uma API que implementa HATEOAS, veja que na resposta dessa API há uma coleção “links”, cada link aponta para uma rota dessa API. No caso desse exemplo, temos um link para a própria rota, um link para alterar um cliente e outro para excluir.



# SOAP X REST

Diferentes abordagens para  
criação de APIs

# SOAP X REST

REST e SOAP são duas abordagens diferentes de transmissão de dados online. Em específico, eles definem como as APIs são criadas, o que possibilita a comunicação dos dados entre aplicações web. O REST é um conjunto de princípios de arquitetura. Já o SOAP é um protocolo oficial mantido pela World Wide Web Consortium (W3C). A principal diferença é que SOAP é um protocolo e REST, não. Normalmente, uma API será baseada em REST ou SOAP, dependendo do caso de uso e das preferências do desenvolvedor.



# SOAP

Com a proliferação das **APIs Web**, uma especificação de protocolo foi desenvolvida para ajudar a padronizar a troca de informações: o **Simple Object Access Protocol**, mais conhecido como **SOAP**. As APIs projetadas com SOAP usam o **XML** como formato de mensagem e recebem solicitações por **HTTP** ou **SMTP**. O SOAP facilita o compartilhamento de informações por aplicações executadas em ambientes diferentes ou escritos em linguagens diferentes.

Como se trata de um protocolo, ele impõe regras integradas que **aumentam sua complexidade** e sobrecarga, **desacelerando o tempo de carregamento das páginas**. No entanto, esses padrões também proporcionam conformidade integrada, fazendo com que SOAP seja uma opção recomendada para casos empresariais. Isso inclui segurança, atomicidade, consistência, isolamento e durabilidade (ACID), que é um conjunto de propriedades para assegurar transações confiáveis de bancos de dados.

# SOAP

As especificações de serviço web comuns incluem:

- **Segurança de serviços web (WS-Security):** padroniza como as mensagens são protegidas e transferidas por meio de identificadores exclusivos chamados de tokens.
- **WS-ReliableMessaging:** padroniza o processamento de erros entre as mensagens transferidas por uma infraestrutura de TI não confiável.
- **Endereçamento de serviços web (WS-Addressing):** empacota informações de roteamento como metadados em cabeçalhos SOAP, em vez de armazená-las em camadas mais a fundo na rede.
- **Linguagem de descrição de serviços web (WSDL):** descreve a atividade de um serviço web e onde ele é iniciado e finalizado.

# SOAP

Quando uma solicitação de dados é enviada a uma API SOAP, ela pode ser processada por meio de qualquer protocolo de camada da aplicação: HTTP (em navegadores da web), SMTP (em e-mails), TCP e muito mais. No entanto, depois que a solicitação é recebida, as mensagens SOAP precisam ser retornadas como documentos XML: uma linguagem de marcação que pode ser lida por máquinas e pessoas. Um navegador não pode armazenar em cache uma solicitação concluída a uma API SOAP. Por isso, não é possível acessá-la depois sem fazer o reenvio à API.

# REST

APIs web que adotam as restrições de arquitetura da REST são chamadas de APIs RESTful. A REST é fundamentalmente diferente do SOAP: o SOAP é um protocolo e a REST é um estilo de arquitetura. Isso significa que não há um padrão oficial para APIs RESTful web. Conforme definido na dissertação de Roy Fielding “Architectural Styles and the Design of Network-based Software Architectures”, as APIs serão consideradas RESTful se estiverem em conformidade com seis restrições de arquitetura:

# REST

- **Arquitetura cliente-servidor:** a arquitetura REST é composta por clientes, servidores e recursos. Ela lida com as solicitações via HTTP.
- **Sem monitoração de estado:** nenhum conteúdo do cliente é armazenado no servidor entre as solicitações. Em vez disso, as informações sobre o estado da sessão são mantidas com o cliente.
- **Capacidade de cache:** o armazenamento em cache pode eliminar a necessidade de algumas interações entre o cliente e o servidor.
- **Sistema em camadas:** as interações entre cliente e servidor podem ser mediadas por camadas adicionais. Essas camadas podem oferecer recursos extras, como balanceamento de carga, caches compartilhados ou segurança.
- **Código sob demanda (opcional):** os servidores podem ampliar a funcionalidade de um cliente por meio da transferência de códigos executáveis.

# REST

- **Interface uniforme:** essa restrição é essencial para o design de APIs RESTful e inclui quatro direções:
  - **Identificação de recursos nas solicitações:** os recursos são identificados nas solicitações e separados das representações retornadas para o cliente.
  - **Manipulação de recursos por meio de representações:** os clientes recebem arquivos que representam recursos. Essas representações precisam ter informações suficientes para permitir a modificação ou exclusão.
  - **Mensagens autodescritivas:** cada mensagem retornada para um cliente contém informações suficientes para descrever como ele deve processá-las.
  - **Hipermídia como plataforma do estado das aplicações:** depois de acessar um recurso, o cliente REST pode descobrir todas as outras ações disponíveis no momento por meio de hiperlinks.

# REST

Essas restrições podem parecer excessivas, mas são muito mais simples do que um protocolo prescrito. Por isso, as APIs RESTful estão se tornando mais comuns do que as APIs SOAP.

Nos últimos anos, as especificações da OpenAPI se tornaram o padrão na hora de definir APIs REST. A OpenAPI permite que desenvolvedores de todas as linguagens criem interfaces de API REST compreensíveis com o mínimo de suposições.

Outro padrão de API emergente é o **GraphQL**, uma linguagem de consulta e ambiente de execução voltado a servidores alternativa ao REST. A prioridade do GraphQL é fornecer exatamente os dados que os clientes solicitam e nada além. Como alternativa à arquitetura REST, o GraphQL permite aos desenvolvedores construir solicitações que extraem os dados de várias fontes em uma única chamada de API.

# SOAP X REST

Enquanto muitos sistemas legados ainda usam SOAP, REST surgiu depois e costuma ser vista como uma alternativa mais rápida nos casos baseados em web. REST é um conjunto de diretrizes que oferece uma implementação flexível. Já SOAP é um protocolo com requisitos específicos, como a mensageria XML.

As APIs REST são leves e ideais para contextos mais modernos, como a Internet das Coisas (IoT), desenvolvimento de aplicações mobile e serverless. Os serviços web SOAP oferecem segurança integrada e transações em conformidade que atendem a muitas necessidades empresariais, mas que também os deixam mais pesados. Além disso, muitas APIs públicas, como a do Google Maps, seguem as diretrizes REST.



# Conclusão

Neste conteúdo foi apresentado a origem, conceitos e os principais fundamentos do REST.

É possível notar que o desenvolvimento de sistemas **RESTful** exalta a clara separação das equipes em desenvolvedores de componentes “cliente” e componentes “servidor”, evitando o alto acoplamento entre os componentes.

Quando o modelo “em camadas” de componentes “servidor” é adotado, é comum que cada camada seja mantida por um time dedicado. Aliás, em casos extremos, cada componente “servidor”, em cada camada, poderá ser mantido por um time especialista.

# Obrigado!

## Alguma Pergunta?

`willian_brito00@hotmail.com`

`www.linkedin.com/in/willian-ferreira-brito`

`github.com/willian-brito`

