

# HTTP

O MÍNIMO QUE  
TODO  
DESENVOLVEDOR  
WEB PRECISA  
SABER



**MARCELO GALHEGO**

# HTTP

O mínimo que todo  
desenvolvedor web precisa  
saber

MARCELO GALHEGO

1ª edição

Marcelo Galhego Cardoso

HTTP

O mínimo que todo desenvolvedor web precisa saber

Prefácio

Cláudio Barizon

1ª Edição – Junho/2022

ISBN 9798840347102

1 – Tecnologia. 2 – Desenvolvimento Web.  
3 – Arquitetura Web. 4 – HTTP e HTTP/S.

*Para Marcelo, Maria Clara e Gabriel*

# Sumário

## PREFÁCIO

Cláudio Barizon

## INTRODUÇÃO

O Que Você Vai Encontrar Aqui

## PARTE I - HTTP

Capítulo 1: O Que é o HTTP?

Como funcionam o HTTP e o TCP?

Vejamos um exemplo de uma requisição GET:

Capítulo 2: O Que é o HTTPS?

Certificado Digital

Capítulo 3: Métodos (Verbos) HTTP

GET

PUT

POST

DELETE

HEAD

CONNECT

OPTIONS

TRACE

PATCH

Capítulo 4: Códigos de Retorno do HTTP/HTTPS

1xx: Informação (Informational Response)

2xx: Sucesso (Success)

3xx: Redirecionamento (Redirection)

4xx: Erro do Cliente (Client Error)

5xx: Erro do Servidor (Server Error)

Vejamos alguns exemplos:

Capítulo 5: Parâmetros de Requisições do HTTP/HTTPS

Capítulo 6: Mais Alguns Conceitos Importantes

DOMÍNIO

SUBDOMÍNIO

RECURSOS

[ENDEREÇO IP](#)

[DNS](#)

[PORTAS](#)

[URL \(Uniform Resource Locator\)](#)

[URN \(Uniform Resource Name\)](#)

[URI \(Uniform Resource Identifier\)](#)

[Capítulo 7: Modelo Requisição-Resposta \(\*Request-Response\*\)](#)

[Sessões e Cookies](#)

[Capítulo 8: Web Services](#)

[XML vs JSON](#)

[Exemplo de Arquivo JSON](#)

[Exemplo de Arquivo XML](#)

[Capítulo 9: O Padrão REST](#)

[Resumindo](#)

## **[PARTE II – HTTP/2](#)**

[Capítulo 10: HTTP/2](#)

[SERVER-PUSH](#)

[KEEP-ALIVE](#)

[Resumindo](#)

## **[PARTE III – TRILHA DE APRENDIZADO PARA O DESENVOLVEDOR WEB](#)**

[Capítulo 11: \*Roadmap\* do Desenvolvedor Web](#)

[Aprenda Inglês](#)

[Cultive Bons Hábitos](#)

[Trilha Full-Stack](#)

## **[CONCLUSÃO](#)**

## **[REFERÊNCIAS](#)**

## **[SOBRE O AUTOR](#)**

# PREFÁCIO

Cláudio Barizon

Há mais de 30 anos, de meados para o fim da década de 1980, escutava que o futuro estava na informática (usando o termo da época). E isso me animou a abraçá-la como profissão. Fiz escola técnica, depois universidade e gostava muito de modelar as entidades, o banco de dados e, principalmente, programar. Àquela altura, as transformações já eram grandes e muitas empresas estavam substituindo seus *mainframes* por plataformas baixas. A microinformática já estava tomando conta e os sistemas passaram a ser desenvolvidos usando a arquitetura cliente-servidor. Nada de internet ainda, mas o desenvolvimento de *software* era uma carreira promissora, com muitas oportunidades e muitos sistemas corporativos a serem criados.

Hoje em dia, vivemos um mundo digital, de transformações constantes e aceleradas. Um mundo de startups, que buscam resolver novos problemas, e corporações estabelecidas, que estão na luta para se reinventarem e se adequarem aos comportamentos dos consumidores cada vez mais voláteis e exigentes, empoderados por seus *smartphones* e uma internet cada vez mais rápida. A área de Tecnologia nunca esteve tão em alta, assim como a demanda por profissionais de desenvolvimento, em especial. A cada ano, o déficit de desenvolvedores só aumenta aqui no Brasil, principalmente, por conta da globalização e das facilidades do trabalho remoto, que permitem que os nossos profissionais trabalhem para empresas em qualquer parte do mundo ganhando em dólar ou euro. Portanto, se há 30 anos a informática era a profissão do futuro, trabalhar com Tecnologia é o presente e, certamente, continuará sendo assim por muito tempo.

As oportunidades estão aí e nós precisamos capacitar cada vez mais profissionais. Trata-se de uma jornada de conhecimento e o Marcelo Galhego te ajuda, com este livro, a dar o primeiro passo. É muito fácil cair na tentação de começar aprendendo uma linguagem de programação para *web* e fazer os seus primeiros *sites*. No entanto, o Marcelo te faz um convite: começar pelo começo e conhecer os fundamentos da *World Wide*

*Web*. E este entendimento é muito importante para você se transformar num ótimo desenvolvedor, como se fala hoje em dia, *full-stack*.

Há mais de 30 anos (e lá vou eu novamente), eu conheci o Marcelo. Ele foi o meu primeiro coordenador no primeiro estágio que fiz, como universitário já. Trabalhamos juntos por muito tempo e por 3 empresas distintas. Então, não é difícil imaginar que, depois de todo este tempo, de muito trabalho em equipe e alguns conflitos inevitáveis, tornamo-nos grandes amigos. Mas não estou aqui para escrever sobre o meu amigo, mas sim sobre o grande profissional que sempre foi e pelo qual me espelhava. Sempre quis ser como o Marcelo, “quando eu crescesse”: um cara que tinha uma grande facilidade em programar, era criativo e engenhoso, sempre buscando produzir bibliotecas para padronizar e otimizar o trabalho, encapsulando as complexidades das funções necessárias.

Assim, posso dizer, escute o que Marcelo fala e não deixe de prestar atenção nas preciosas dicas e ensinamentos que ele traz aqui de forma direta e objetiva. Até porque ele explica este assunto, pelo menos para mim, um pouco mais árido, de uma forma simples e didática. Então, não perca esta oportunidade, para não perder as oportunidades que aparecerão certamente para você, desenvolvedor *web*, conhecedor dos fundamentos!

Mãos à obra!

Cláudio Barizon

# INTRODUÇÃO

Toda jornada tem um início, e acredito que, se você está procurando ser um desenvolvedor web, mas não sabe como e por onde começar, este livro pode ser seu primeiro passo nesta fantástica viagem. Espero que este livro o ajude a alcançar seus objetivos e transformar seus sonhos em realidade.

Entendo que todo profissional da web deva saber o básico sobre seu funcionamento. Para se tornar um desenvolvedor *web*, primeiro você precisa conhecer seus fundamentos. A primeira coisa que fazemos ao construir uma casa são seus alicerces, e não seria diferente ao nos dedicarmos a qualquer aprendizado.

O entendimento da arquitetura básica da internet, pode, definitivamente, lhe dar uma melhor compreensão de seus objetivos como desenvolvedor.

As 3 principais linguagens, ao menos no “lado” cliente, que compõem a *World Wide Web* são HTML, CSS e JavaScript. Porém, entendo que a curva de aprendizado dessas ferramentas é sensivelmente atenuada se você domina um mínimo da arquitetura, e este é o cerne deste livro.

Não tenho a pretensão de torná-lo um especialista em redes, mas sim de lhe fornecer um conjunto de informações que lhe ajudarão na sua formação como profissional de desenvolvimento de aplicações para a internet.

O HTTP é um dos protocolos mais utilizados da internet. Ouso afirmar que o mais importante para o desenvolvedor *web*.

Meu objetivo é apresentar seus conceitos básicos, detalhando o que entendo ser o necessário em termos de conhecimento sobre a arquitetura do HTTP.

Eu ainda pretendo atingir um outro objetivo com este livro: que você entenda o que é necessário para se tornar um desenvolvedor *web*, seja de *front-end* (“lado” do cliente), *back-end* (“lado” do servidor) ou *full-stack* (cliente/servidor).



Vamos lá?

### **O Que Você Vai Encontrar Aqui**

A Parte I tem o objetivo de apresentar o protocolo HTTP e seus principais conceitos.

Na Parte II do livro, vou mostrar a evolução do HTTP, o HTTP/2.

Já a Parte III fornecerá uma trilha de aprendizado para você se tornar um desenvolvedor *web* completo.

# **PARTE I - HTTP**

# Capítulo 1: O Que é o HTTP?

Bem, antes de iniciarmos, vou reforçar o objetivo deste livro, que é fornecer uma base de como o HTTP funciona, o mínimo para quem busca se aventurar no desenvolvimento *web*. Por mais que um desenvolvedor *web* não trabalhe diretamente com o HTTP, faz-se necessário um mínimo de conhecimento em relação a este importante, senão o mais, protocolo da internet.



No final deste livro, tenha certeza de que você dominará o necessário para iniciar no fabuloso mundo da programação para internet.

HTTP significa *Hypertext Transfer Protocol*, e é usado para gerenciar as requisições (cliente) e as respostas (servidor) pela internet. O HTTP requer que os dados sejam transferidos de um ponto para outro pela rede.

A transferência de recursos ocorre por meio do TCP (*Transmission Control Protocol*). O TCP gerencia os canais entre o navegador (*browser*) e o servidor. O TCP é usado para gerenciar muitos tipos de conexões de internet em que um computador ou dispositivo deseja enviar algo para outro. HTTP é a linguagem de comando que os dispositivos em ambos os lados da conexão devem seguir para se comunicar.

Em qualquer comunicação é preciso existir regras para que as duas partes (interlocutoras) consigam se entender com sucesso. No caso da comunicação do seu navegador com qualquer *site*, esse conjunto de regras é um protocolo, neste caso o HTTP.

De forma resumida, o HTTP é o protocolo que define as regras de comunicação entre o cliente e o servidor na *web*.

O protocolo HTTP segue o modelo *Client-Server* (Cliente-Servidor). Nesta arquitetura, temos um servidor único e vários clientes que o acessam.

Existem outras arquiteturas de rede, como por exemplo o *Peer-to-Peer* ou P2P (Ponto-a-Ponto), que, embora não seja o foco deste livro, simplificando, funciona de forma em que cada nó da rede é servidor e cliente ao mesmo tempo.

#### **Como funcionam o HTTP e o TCP?**

Quando você digita um endereço como *www.google.com.br* em seu navegador *web* (*browser*), você está solicitando que ele abra um canal TCP para o servidor que responde a essa URL. Uma URL é como um endereço residencial, uma caixa postal ou um número de telefone, pois define como acessar um local ou uma pessoa, no caso um servidor (URL).

Neste cenário, o computador que está fazendo a requisição é chamado de cliente. A URL que você está solicitando é o endereço que pertence ao

servidor.

Depois que a conexão TCP é estabelecida, o cliente envia uma requisição HTTP GET ao servidor para recuperar a página da *web* que ele (cliente) deve exibir. Após o servidor ter enviado a resposta, ele fecha a conexão TCP. Se você abrir o site novamente em seu navegador, ou se seu navegador requisitar automaticamente algo do servidor, uma nova conexão será aberta seguindo este mesmo processo.

As requisições GET são um tipo de método HTTP que um cliente pode chamar. Existem outros métodos comuns (também chamados verbos), como POST, PUT e DELETE.

Não se preocupe, mais a frente, irei detalhar cada um desses métodos.

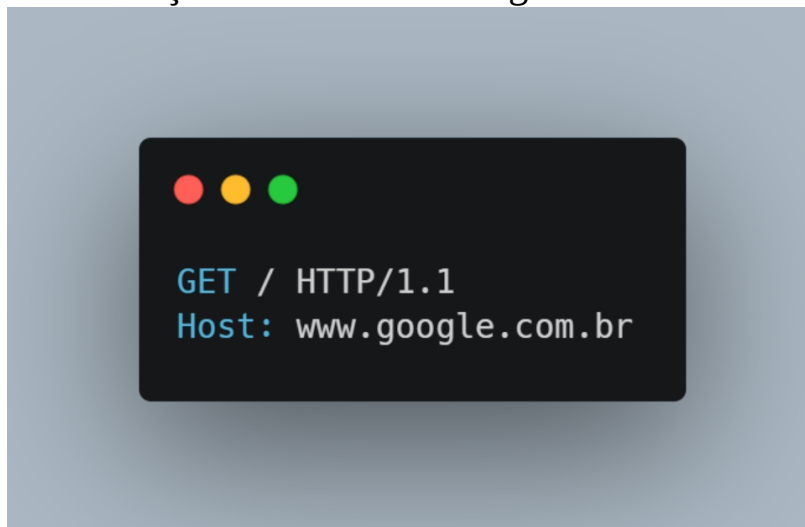
### **Vejamos um exemplo de uma requisição GET:**

As requisições GET são o tipo mais comum, e são usadas para que seu computador (o cliente) possa acessar recursos na web (servidor).

Ao digitar *http://google.com.br* em seu navegador, ele extrai a parte “*http*”, já que reconhece que se trata do protocolo de rede a ser usado, pega o nome de domínio da URL, no nosso exemplo “*google.com.br*”, e solicita a um Servidor DNS o endereço IP (*Internet Protocol*) do *site* a ser acessado.

Neste ponto, o cliente sabe o endereço IP do site que se quer acessar. Então ele abre uma conexão com o servidor nesse endereço, usando o protocolo HTTP. Em seguida, é realizada uma requisição GET para o servidor que contém o endereço IP do *host* (servidor) e, opcionalmente, um envio de dados adicionais.

A solicitação GET contém o seguinte formato:



Tais informações identificam o tipo de requisição (GET), o recurso em *www.google.com.br* (neste caso, “/”) e o protocolo “HTTP/1.1”.

HTTP/1.1 é uma revisão da primeira versão do HTTP, que agora é chamada de HTTP/1.0.

No HTTP/1.0, cada requisição de recurso requer uma conexão separada com o servidor. O HTTP/1.1 usa uma conexão mais de uma vez, para que o conteúdo adicional (como imagens ou folhas de estilo) seja recuperado

mesmo depois que a página for obtida. Como resultado, as requisições que usam HTTP/1.1 são mais rápidas daquelas que usam HTTP/1.0.

A segunda linha da requisição contém o endereço do servidor que é *www.google.com.br*. Também pode haver linhas adicionais dependendo de quais dados seu navegador escolher enviar, por exemplo, parâmetros de pesquisa ou identificação de dados a serem recuperados.

Se o servidor conseguir localizar a URL solicitada, o servidor responderá com o seguinte cabeçalho:



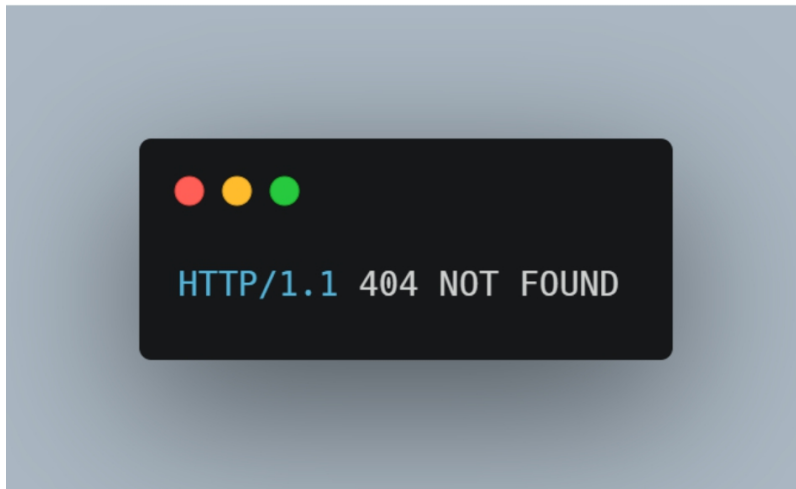
Este cabeçalho é seguido pelo conteúdo solicitado, que neste caso é a informação necessária para renderizar (exibir) a página *www.google.com.br*.

A primeira linha do cabeçalho, HTTP/1.1 200 OK, é a confirmação de que o servidor entende o protocolo com o qual o cliente deseja se comunicar (HTTP/1.1) e um código de status HTTP indicando que o recurso foi encontrado no servidor.

A segunda linha, “*Content-Type: text/html*”, mostra o tipo de conteúdo que será enviado ao cliente.

Se o servidor não conseguir localizar o caminho solicitado pelo cliente, ele responderá com o cabeçalho:





Nesse caso, o servidor identificou o protocolo HTTP, mas o código de status 404 NOT FOUND significa que o conteúdo solicitado não foi encontrado. Isso pode acontecer se o conteúdo foi movido ou se a URL foi digitada de forma incorreta, ou ainda se a página foi removida.

## Capítulo 2: O Que é o HTTPS?

Uma requisição HTTP pode ser lida por qualquer pessoa ao trafegar pela rede (são em texto puro, legível), não é seguro fornecer informações sensíveis, como por exemplo, um número de cartão de crédito.

Para sanar este problema muitos servidores suportam HTTPS, abreviação de *HTTP Secure*, que permite criptografar os dados que você envia e recebe.

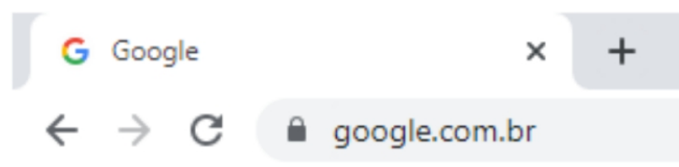
O HTTPS é basicamente o HTTP, porém com uma camada adicional de segurança/criptografia que, inicialmente era SSL, porém passou a ser também TLS. É muito comum que estas duas siglas sejam encontradas juntas como SSL/TLS (*Secure Sockets Layer / Transport Layer Security*).

É importante usar HTTPS ao passar informações confidenciais ou pessoais pela *web*.

Note que ao acessar *www.google.com.br* o navegador mostra *https://www.google.com.br/* (com um cadeado à esquerda), indicando que o *site* adota o protocolo HTTPS.



### HTTPS - Site Seguro



<https://mgcconsulting.com.br>



O HTTPS utiliza criptografia baseada em chaves públicas e privadas, que, para serem geradas, necessitam que o possuidor das chaves tenha sua identidade confirmada. Tal processo é realizado a partir de um certificado

digital, isto é, um certificado digital é utilizado para garantir a identidade de uma entidade, bem como para a geração das chaves de criptografia.

Os navegadores (clientes) ficam de posse da chave pública, e as utilizam para criptografar as informações a serem enviadas ao servidor, que por sua vez utiliza a chave privada para descriptografar as mesmas informações. Somente a chave privada descriptografa as informações criptografadas com a pública. Nota-se a importância de se manter as chaves privadas de forma segura.

O método de criptografia que utiliza duas chaves distintas (pública e privada) é chamado de criptografia assimétrica. Porém, a criptografia assimétrica tem um problema: a baixa performance.

Como alternativa, temos a criptografia simétrica, que usa a mesma chave para criptografar e descriptografar os dados.

A criptografia simétrica é muito mais rápida, mas em contrapartida, menos segura que a assimétrica. Como existe apenas uma chave, ela ficará espalhada pelos clientes (navegadores) e qualquer um, que tem a posse dessa chave, pode decifrar a comunicação.

Então como resolver a questão da segurança versus performance? Acontece que o HTTPS utiliza ambos os métodos de criptografia (assimétrica e simétrica).

O certificado possui a chave pública que o cliente utiliza, assim como o servidor possui a chave privada, que como falamos é a criptografia assimétrica (segura, mas mais lenta). O HTTPS faz uma comutação entre a criptografia assimétrica e a simétrica.

Na prática, o cliente gera uma chave simétrica, que é enviada ao servidor utilizando a criptografia assimétrica. A partir do momento que o servidor recebe a chave simétrica, ambos (cliente e servidor) passam a utilizar essa mesma chave, ou seja, passam a se comunicar de forma simétrica para as requisições seguintes.

Cabe às empresas, que mantêm os servidores seguros, configurá-los. Para oferecer suporte a HTTPS, a empresa deve solicitar um certificado de uma Autoridade Certificadora, que além de garantir a identidade do site, define também a validade do certificado.

### **Certificado Digital**

Dentro do contexto do HTTPS, um certificado digital é a prova da identidade para um site, onde temos informações sobre o seu domínio, a chave pública do titular (site/servidor), bem como a data de validade (expiração) desse certificado.

O Certificado Digital não é de uso exclusivo do protocolo HTTPS, na verdade, é uma ferramenta cada vez mais presente no nosso dia a dia, utilizado em uma infinidade de situações, como, por exemplo, acesso a informações do cidadão junto ao Governo Federal.

## Capítulo 3: Métodos (Verbos) HTTP

Como já visto, o HTTP utiliza uma série de métodos, também chamados verbos HTTP, que são aplicados para gerenciar as requisições dos clientes e as respostas do servidor. Cada verbo define o método de como a requisição será tratada.

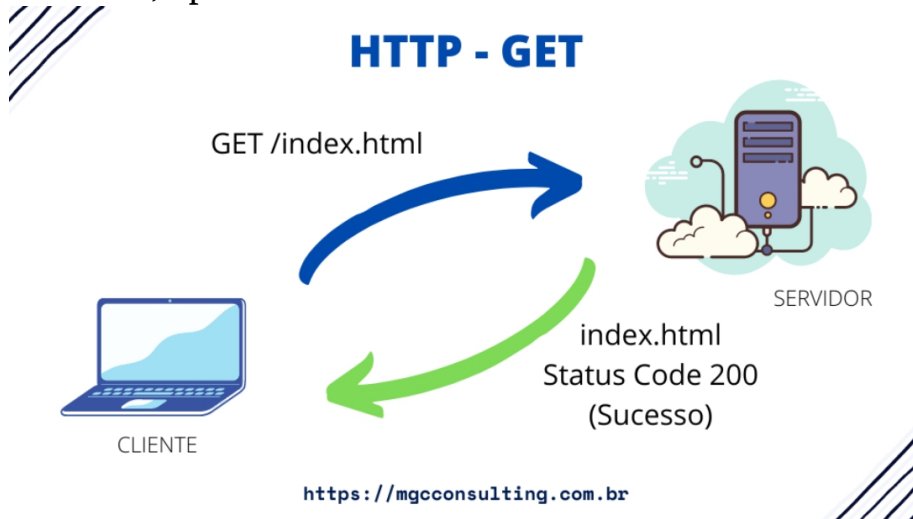
Então, os métodos de solicitações HTTP são as ações iniciadas do lado do cliente para executar determinadas funções do lado servidor.

Existem 9 métodos de solicitação HTTP: GET, PUT, POST, DELETE (sendo esses 4 primeiros, a meu ver, os principais e mais utilizados), HEAD, CONNECT, OPTIONS, TRACE e PATCH.

Vamos falar sobre eles um pouco mais detalhadamente.

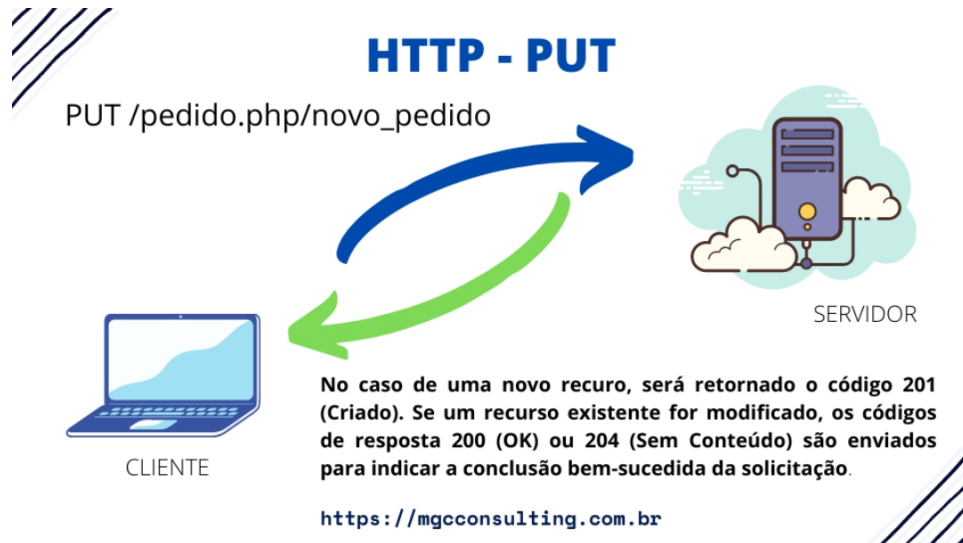
## GET

O método GET é o mais comum de todos os métodos de solicitação. O GET solicita ao servidor um recurso específico. Requisições GET retornam, normalmente, apenas dados. O método GET não afeta o estado do servidor.



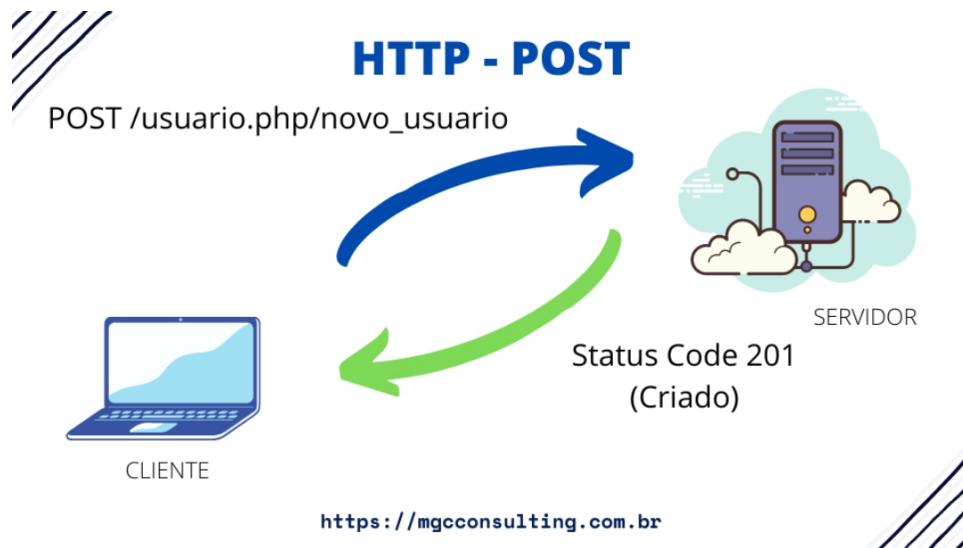
## PUT

O PUT normalmente é usado com o intuito de adicionar/incluir algo no servidor. Ou seja, quando enviamos dados, ou ainda para substituir uma representação do recurso de destino com os novos dados. O PUT altera o estado do servidor.



## POST

O POST é utilizado para realizar uma alteração nos dados do servidor. Fará uma substituição dos dados do recurso destino pelos dados da requisição. Pode também ser usado para criar um recurso.



### Nota importante:

A diferença entre o PUT e o POST é relativamente tênue. Ambos podem servir para criar/atualizar dados no servidor. PUT é idempotente, isto é, se executado múltiplas vezes o resultado será o mesmo (sem efeito colateral). Já o POST não é idempotente, se chamado várias vezes fornece resultados diferentes (com efeito colateral). Por exemplo, se acidentalmente chamarmos o POST várias vezes para criar um pedido de compra, vários registros idênticos serão criados, ao contrário do PUT.

## DELETE

O método DELETE exclui (remove) um recurso do servidor.



# HTTP - DELETE

DELETE /usuario.php/usuario\_id=123



CLIENTE



SERVIDOR

O método DELETE, quando bem sucedido, pode retornar os seguintes códigos de resposta: 202 (Aceito), 200 (OK) ou 204 (Sem Conteúdo).

<https://mgcconsulting.com.br>

## **HEAD**

O HEAD solicita uma resposta do servidor, sem a necessidade dela conter o corpo da resposta. A requisição realizada via HEAD solicita ao servidor uma resposta idêntica à de uma solicitação GET, mas é mais rápida, pois pequenos dados são transferidos.

## **CONNECT**

O método CONNECT estabelece uma conexão, ou túnel, com o servidor. Trata-se de uma comunicação bidirecional entre o cliente e o recurso solicitado.

## **OPTIONS**

O OPTIONS descreve as opções de comunicação com o recurso de destino.

## **TRACE**

O método TRACE executa um teste denominado *loopback* (envio de uma mensagem ao longo do caminho até o recurso alvo). Geralmente é usado para depuração (debug).

## **PATCH**

O PATCH aplica modificações parciais em um recurso (não altera os dados completamente).

# Capítulo 4: Códigos de Retorno do HTTP/HTTPS

O protocolo HTTP estabelece códigos padrões para determinar/informar o status das respostas, isto é, a cada requisição o HTTP envia a resposta acompanhada de um código (de 3 dígitos), que informa ao navegador *web* o status da resposta.

Os códigos são divididos em 5 diferentes classes:

## **1xx: Informação (Informational Response)**

Indica que a solicitação do cliente foi recebida e compreendida pelo servidor.

É uma resposta provisória enviada ao cliente enquanto sua solicitação ainda está sendo processada e solicita que o cliente aguarde a resposta final do servidor.

## **2xx: Sucesso (Success)**

A solicitação do cliente foi recebida, aceita e processada com sucesso pelo servidor.

## **3xx: Redirecionamento (Redirection)**

Informa que o recurso solicitado pelo cliente não existe onde o cliente enviou a solicitação original.

O servidor encontrou o recurso em outro local, então a solicitação do cliente será redirecionada para esse local.

## **4xx: Erro do Cliente (Client Error)**

Estes códigos são enviados quando uma solicitação realizada pelo cliente não é correta e o servidor não consegue entender.

Indica algo errado no pedido do cliente.

## **5xx: Erro do Servidor (Server Error)**

Enviados quando a solicitação realizada pelo cliente é válida, mas o servidor encontrou algum erro ao processar a solicitação.

Indica algo errado com o servidor.

**Vejamos alguns exemplos:**

Classe Código	Status	Exemplo
1xx	Informação	100: O servidor recebeu os cabeçalhos da solicitação e o cliente deve continuar enviando o corpo da solicitação.
2xx	Sucesso	200: OK, a requisição obteve sucesso, não houve erros.
3xx	Redirecionamento	301: Moved Permanently (Movido Permanentemente), a página solicitada foi movida para outra URL.
4xx	Erro do Cliente	404: Not Found (Não Encontrado), a página requisitada não foi encontrada.
5xx	Erro do Servidor	500: Internal Server Error (Erro no Servidor), um erro genérico ocorreu no servidor impedindo a resposta a requisição.

Apenas para consulta futura, segue a lista completa dos códigos de retorno (em inglês, pois entende-se que a tradução da lista seria prejudicial à compreensão):

1XX: INFORMATIONAL STATUS CODES	
Código	Ação
100	Continue
101	Switching Protocols
102	Processing Request
103	Early Hints

2XX: SUCCESSFUL STATUS CODES	
Código	Ação
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used

3XX: REDIRECTION	
Código	Ação
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	Switch Proxy
307	Temporary Redirect
308	Permanent Redirect

4XX: CLIENT ERROR	
Código	Ação
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
418	I'm a Teapot
421	Misdirected Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
425	Too Early
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
451	Unavailable For Legal Reasons

5XX: SERVER ERROR	
Código	Ação
501	Not Implemented
502	Bad Gateway
503	Service Unavailable

504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required

Como desenvolvedor *web*, você realmente não precisa memorizar tudo isso.

Na maioria dos cenários, você verá apenas os códigos mais comuns, como 200, 301 e 404.



## Capítulo 5: Parâmetros de Requisições do HTTP/HTTPS

Outra operação bastante comum no protocolo HTTP é a passagem de parâmetros via URL.

Por exemplo, suponhamos uma pesquisa no Google sobre HTTP. Ao digitar HTTP na barra de pesquisa do Google você notará que a URL foi alterada para algo mais ou menos assim: *https://www.google.com/search?q=http*.

Na verdade, o Google acrescenta uma série de outros parâmetros e informações, mas, para nosso exemplo, vamos assumir que a URL mudou conforme descrito acima.

Vamos analisar a URL. Já sabemos que “*https://www.google.com/*” é o endereço do Google na *web*, “*search/*” o nome do recurso que queremos acessar (na verdade uma pesquisa), em seguida nos deparamos com a letra “*q*”, seguida do sinal de igual (“*=*”) e terminando com o termo pesquisado, no caso “*HTTP*”.

Fica fácil entendermos que se trata de um parâmetro da nossa URL, onde “*q*” é o nome do parâmetro e “*HTTP*” o valor dele.

Com certeza, o mecanismo hospedado num dos servidores do Google entende que nossa requisição se trata da pesquisa (recurso “*search*”), onde é passado o parâmetro “*q*” com valor “*HTTP*”.

Note que podemos ter combinação de parâmetros, que são separados pelo caractere “*&*”.

Por exemplo, a título meramente didático:

*[https://www.google.com/search?q=http&lang=pt\\_BR](https://www.google.com/search?q=http&lang=pt_BR)*

Note que, nesse exemplo fictício, passamos 2 parâmetros, “*q*” e “*lang*”, com os valores “*HTTP*” e “*pt\_BR*” respectivamente.

Por fim, existe uma diferença básica na passagem de parâmetros nos métodos mais utilizados, o GET e o POST.

Enquanto no GET a passagem é feita via URL, no POST os parâmetros seguem no corpo da requisição. Há uma vantagem em termos de segurança, mas este assunto foge do escopo desse livro.

# Capítulo 6: Mais Alguns Conceitos Importantes

## DOMÍNIO

Domínio é o endereço eletrônico de um *site*, o nome pelo qual o *site* será encontrado e acessado.

Registro de Domínio é o serviço que cuida da administração dos nomes (domínios) na internet, que envolve o registro e a renovação dos endereços eletrônicos.

Normalmente um domínio é dividido em quatro partes, por exemplo, em *https://www.google.com.br*, onde:

- “*https*” é o protocolo de comunicação.
- “*www*” é um termo opcional que significa “*World Wide Web*”. Seu uso não é necessário para acessar um *site*.
- “*google*” é o nome do *site*, normalmente identifica a empresa ou entidade dona do domínio.
- “*.com*” identifica a natureza do *site*. Neste caso “*.com*”, inicialmente usado para nomear *sites* comerciais, mas que hoje está disponível para o público em geral. Podemos entender como um complemento ao nome principal do domínio. Há várias terminações, cada uma busca identificar um tipo de *site* (“*.gov*” – sites governamentais, “*.io*” é um domínio popular para *startups*, engenheiros, desenvolvedores e *geeks* em geral, entre outras terminações).
- “*.br*” identifica o país de origem do *site* (vamos aqui considerar como um complemento a extensão (natureza) do *site*, e não uma quinta parte do domínio).

## SUBDOMÍNIO

Subdomínios representam páginas/sessões específicas dentro de um *site*.

Por exemplo, no caso do Google temos, entre outros, o Gmail (*mail.google.com*) e o Google Drive (*drive.google.com*).

Note que os subdomínios apontam para páginas/recursos diferentes dentro do mesmo domínio.

## RECURSOS

Recurso (ou *Resource*) é basicamente o alvo de uma requisição HTTP/HTTPS.

Por exemplo na URL <https://developer.mozilla.org/pt-BR/docs/Learn/>, o alvo que buscamos é o recurso “*pt-BR/docs/Learn/*”, na verdade, de forma purista, o recurso é “*Learn/*”, já que “*pt-BR/docs/*” são caminhos intermediários que usamos para acessar o recurso.

## ENDEREÇO IP

IP significa *Internet Protocol*. O endereço IP identifica de forma única um *site*, uma rede ou ainda um dispositivo na internet (por exemplo uma impressora ou um roteador).

Um endereço IP é um número de 32 bits. Os endereços IP normalmente são expressos em formato decimal, com quatro grupos de números separados por pontos, como 192.158.1.38.

## DNS

Já falei anteriormente, de forma superficial, sobre a função de um servidor DNS, mas buscarei entrar agora em mais detalhes.

DNS (*Domain Name System*) é um serviço que, basicamente, traduz um nome do *site* (URL) em um endereço IP.

Então quando você digita em seu navegador [www.google.com.br](http://www.google.com.br), algum servidor DNS fará a transformação da URL digitada em um endereço IP, com isso sua requisição segue adiante.

O DNS é um elemento importante da infraestrutura da web, servindo como uma lista telefônica da internet (banco de dados de domínios), como falamos, a partir da URL ele devolve o endereço IP.

Existem diversos servidores DNS espalhados pela *web*, inclusive um do Google:

<https://developers.google.com/speed/public-dns/>

## PORTAS

Como já explicado, toda requisição de um cliente abre uma conexão com o servidor que roda em algum lugar na internet.

Para estabelecer uma conexão na rede é preciso saber qual é o endereço IP, e já vimos como descobri-lo.

Sabemos também que o HTTP é um protocolo de camada de aplicação que funciona com a ajuda do protocolo de camada de transporte, o TCP.

Agora, para acessar o servidor não basta apenas o endereço IP, precisamos saber por qual conexão realizar o acesso. Neste caso, por qual porta devemos conectar com o servidor.

A porta reservada para o protocolo HTTP é a 80. Já para o HTTPS é a 443. O navegador já sabe, dependendo do protocolo, qual porta utilizar. Por isso, não precisamos digitar a porta junto a URL. Porém, nada nos impede de fazê-lo, ou seja, *https:\\google.com.br* e *https:\\google.com.br:443* terá o mesmo efeito.

As principais portas do protocolo HTTP/HTTPS são:

Protocolo	Porta
FTP	21
SSH	22
TELNET	23
SMTP	25
DNS	53
HTTP	80
POP3	110
SFTP	115
IMAP	143
HTTPS	443
SMTP	587
SQUID	3128

## URL (Uniform Resource Locator)

São os endereços *web*. URLs podem ser digitados na barra de endereços do seu navegador, indicando qual página a ser carregada.



`https://mgcconsulting.com.br`

## URN (Uniform Resource Name)

Um URN define um recurso da *web*/internet. Uma URN identifica de forma única e persistente um recurso, independente da sua localização, permitindo a separação entre a identificação (nome único) e a localização (URL) de um determinado recurso.

## URI (Uniform Resource Identifier)

Segundo definição da Fundação Mozilla, temos:

*“A URI (Uniform Resource Identifier, ou Identificador Uniforme de Recursos) é uma string (sequência de caracteres) que se refere a um recurso. A mais comum é a URL, que identifica o recurso localizando-o na web. URNs (en-US), em contraste, refere-se a um recurso pelo nome, em dado namespace. Ex: o ISBN de um livro.”*

Note que uma URI é similar a uma URL: ambas definem endereços web. Porém, nem toda URI é uma URL, já que pode definir apenas o recurso, e não o endereço completo. Uma URL sempre define um endereço completo.



## Capítulo 7: Modelo Requisição-Resposta (*Request-Response*)

Agora que vimos os principais conceitos do protocolo HTTP/HTTPS, podemos nos aprofundar no modelo Requisição-Resposta.

Na prática, de forma geral, a comunicação entre um servidor e um cliente é sempre iniciada por um pedido (requisição) do cliente ao servidor, que por sua vez enviará uma resposta ao cliente, seja com a solicitação atendida, seja com um código de retorno de erro, como, por exemplo, o famoso 404 (*Page Not Found*).

Todo pedido de um cliente constitui uma nova requisição. Por exemplo, ao acessarmos <https://mgccconsulting.com.br/>, uma requisição é realizada e a página é enviada para o navegador (cliente). Ao clicarmos em algum *link* dentro da página, uma nova requisição é realizada e uma nova resposta é emitida pelo servidor.

Uma requisição é composta basicamente de 3 elementos: o Método, o Cabeçalho e o Corpo.

Como já visto, o Método é o comando geral enviado ao servidor. O Método é seguido pelo caminho para o recurso e a versão HTTP. Por exemplo: “*GET /add/user HTTP/1.1*”.

Além do “cabeçalho” método, existem cabeçalhos HTTP que contêm informações mais específicas sobre a solicitação. Alguns obrigatórios, outros opcionais.



Dentre os cabeçalhos mais utilizados temos:

1. *Host*: É o URL para o qual a solicitação é feita.
2. *User-Agent*: Informa ao servidor qual navegador e sistema operacional em uso, bem como o dispositivo (*desktop*, *smartphone*).
3. *Cookie*: Usado para armazenar informações, como as de *login*, permitindo que tais dados não precisem ser repetidos nas requisições seguintes.
4. *Referer*: Informa ao servidor o endereço da página *web* anterior.

O último elemento é o corpo da requisição. Nem todos os métodos requerem um corpo, por exemplo, GET, DELETE, HEAD. No entanto, por exemplo, para uma solicitação POST, em que novas informações são enviadas ao servidor, o corpo é o elemento no qual tais dados são transmitidos.

Por fim, as requisições são independentes entre si, o que chamamos de *stateless*, que significa que o HTTP não mantém o estado de requisições. Não há registro de requisições anteriores, logo cada requisição deve conter todas as informações necessárias para ser atendida.

## Sessões e Cookies

Reparem que, em *sites* que necessitam de *login*, esta ação é realizada apenas uma vez. Normalmente na primeira requisição. Tais *sites*, de alguma forma, “lembram” que você já realizou o *login*. Logo, nas próximas requisições, você já se encontra “logado”.

Neste ponto, se pensarmos que cada requisição é independente, imaginamos que o cliente envia ao servidor seus dados de *login* a cada requisição. Porém, não é assim que funciona.

Na verdade, ao realizar o *login* o servidor cria um número aleatório de identificação único, e de difícil adivinhação.

Tal número é devolvido ao cliente, e reenviado a cada nova requisição, em outras palavras, ele substitui o par *login/senha*.



Este número de sessão é gravado pelo navegador na forma de um *cookie*, que é diretamente relacionado ao *site* que estamos acessando.

Os *cookies* nada mais são do que dados que o servidor envia ao cliente.

Embora o protocolo HTTP em si seja *stateless* (não há conexão entre duas solicitações sucessivas) utiliza-se *cookies* para burlar tal restrição.

*Cookies* são utilizados para os mais diversos objetivos, como este de manter o “ID” do usuário.

Então, através dos *cookies* é que as plataformas gerenciam as sessões com os usuários.

Você pode acessar/visualizar os *cookies* mantidos pelo navegador, processo este que varia de navegador a navegador.

*No Chrome os cookies podem ser visualizados da seguinte forma:*

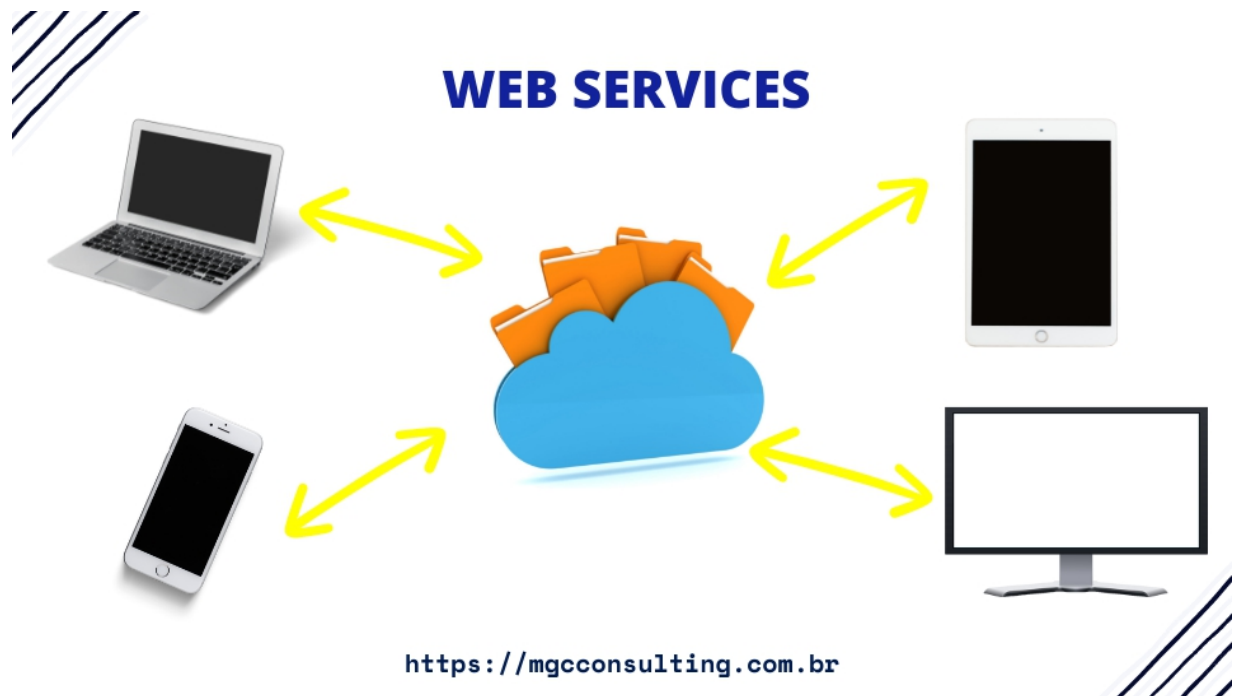
- 1) *Clique com o botão direito na página/site que você deseja visualizar os cookies;*
- 2) *Selecione “Inspecionar”;*
- 3) *Selecione a aba “Aplicativo”;*
- 4) *Na opção “Armazenamento” você acessa/visualiza os cookies utilizados pela página/site.*

*Nota: Este procedimento pode ser alterado no Chrome em novas atualizações. Verifique o modo de acesso de sua versão do navegador.*

Resumindo, uma sessão é o tempo em que o usuário se mantém conectado a um site. É criada para facilitar a comunicação cliente-servidor, agilizando e aumentando a segurança das requisições-resposta. Sessões armazenam dados entre requisições (nos arquivos chamados *cookies*).

## Capítulo 8: Web Services

*Web services* são aplicações que disponibilizam serviços (dados) para outras. É uma API (*Application Programming Interface*) que possui a documentação necessária para seu uso, permitindo uma comunicação eficiente entre sistemas.



*Web services* fazem uso principalmente do protocolo HTTP.

Podemos definir *web services* como um subconjunto de APIs. Porém, tecnicamente se comparados, os *web services* possuem certas limitações em relação as APIs.

Um exemplo clássico de *web service* é o que a partir do CEP retorna o endereço em forma textual. Ou seja, nome da rua, bairro, cidade e estado.

Então, a partir de uma requisição, um *web service* retorna, normalmente, um conjunto de dados (seja texto, imagens, áudio, vídeo, o que seja). Esta reposta é realizada seguindo alguns padrões de formato, para que o cliente entenda e possa fazer o uso correto dos dados retornados.

Os principais, ou pelo menos os mais usuais, padrões de retorno são:

1. JSON (*JavaScript Object Notation*)
2. XML (*eXtensible Markup Language*)
3. HTML (*HyperText Markup Language*)

Ao realizar uma requisição, informamos o padrão que desejamos como resposta a aplicação do servidor através do cabeçalho HTTP. Por exemplo, para retorno no formato JSON, utilizamos o seguinte cabeçalho “*Accept: application/json*”.

Já a resposta conterá o cabeçalho “*Content-Type: application/json*”.

Importante frisar que um *web service* não apenas retorna dados, pode também ser utilizado para alterar dados no servidor, então é intuitivo que *web services* utilizam métodos HTTP, na maioria dos casos: GET (recupera informações, não deve modificar, ou seja, sem efeito colateral do lado do servidor), POST (adiciona informações), PUT (atualiza/modifica, e eventualmente pode adicionar informações) e DELETE (exclui informações).

Neste contexto, informações são sinônimos de recursos. Então métodos, na verdade, são aplicados a recursos do lado servidor.

Lembremos que recursos são definidos via URI. Logo, temos 3 componentes básicos em um *web service*: recursos (URI), operações (métodos HTTP) e dados (XML, JSON, ...).

#### **XML vs JSON**

Vou explorar agora mais alguns detalhes dos formatos XML e JSON, pois entendo que na maioria dos casos esses dois serão os mais utilizados.

No JSON (*JavaScript Object Notation*), os dados são mapeados em pares de valores-chave, o que o torna bastante intuitivo e legível. A chave representa o nome e o formato dos dados e o valor representa os dados em si.

Prima por ser um padrão “leve”, pois contém apenas dados reais em vez da estrutura que o define.

Em contrapartida, tal simplicidade, em algumas situações, é uma desvantagem, pois não é bastante complicado adicionarmos metadados em arquivos JSON, tornando complexo obter maiores informações sobre os dados.

O XML (*eXtensible Markup Language*) é basicamente uma extensão do HTML. Amplamente utilizado para transmissão de dados, pois permite criar elementos personalizados.


Mais detalhado em comparação com JSON, o XML permite que definamos facilmente a estrutura de dados/metadados. Enquanto tal característica o torna menos legível por nós humanos, faz com que seja altamente “legível” para os computadores.

A maior complexidade do formato XML faz com que o seu gerenciamento seja mais “chato”, necessitando de mais esforço de desenvolvimento.

Você deverá escolher qual padrão utilizar de acordo com a demanda de sua aplicação.

Por último, vejamos um exemplo de formatos de retorno de um *web service*, no caso o da ViaCEP<sup>[1]</sup>, que retorna o endereço textual a partir de um CEP:

**Exemplo de Arquivo JSON**



```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "complemento": "lado ímpar",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
  "ibge": "3550308",
  "gia": "1004",
  "ddd": "11",
  "siafi": "7107"
}
```

**Exemplo de Arquivo XML**

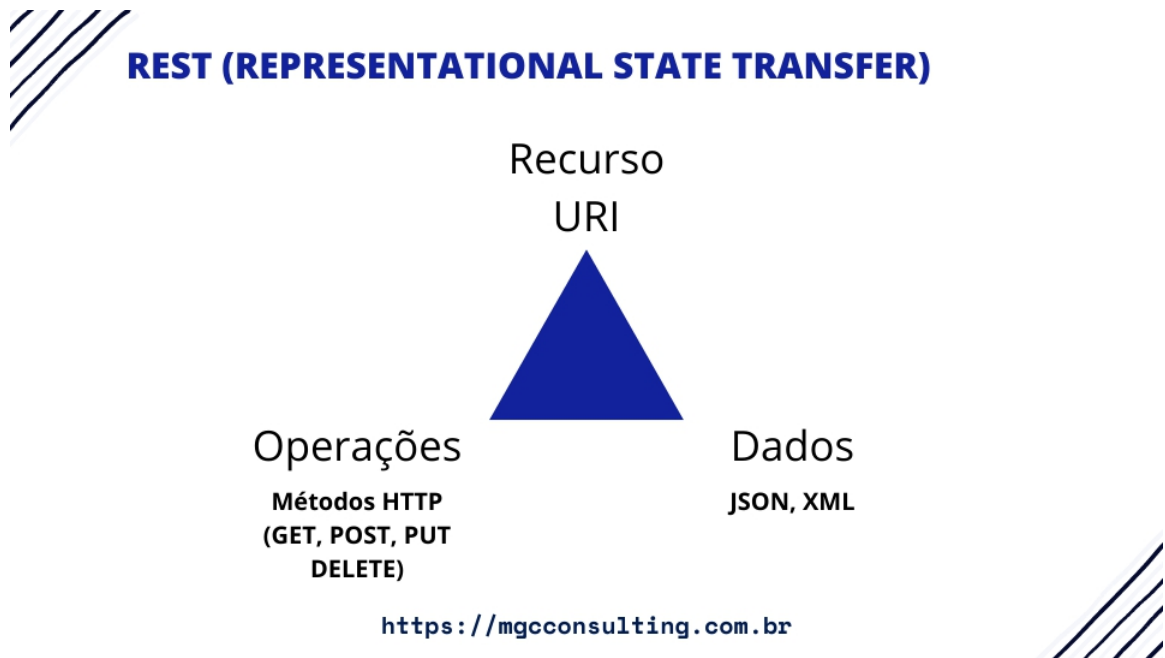




```
<?xml version="1.0" encoding="UTF-8"?>
<xmlcep>
<cep>01001-000</cep>
<logradouro>Praça da Sé</logradouro>
<complemento>lado ímpar</complemento>
<bairro>Sé</bairro>
<localidade>São Paulo</localidade>
<uf>SP</uf>
<ibge>3550308</ibge>
<gia>1004</gia>
<ddd>11</ddd>
<siafi>7107</siafi>
</xmlcep>
```

## Capítulo 9: O Padrão REST

No capítulo anterior aprendemos os 3 componentes que formam um *web service*. Esses componentes são a base de um padrão de arquitetura para interface (comunicação) entre aplicações que chamamos de REST (*Representational State Transfer*) ou API REST.



Falamos também que recursos são representados por URIs e, no padrão REST, tais URIs devem conter apenas substantivos, e não um par substantivo/verbo.

Por exemplo, a URI `/usuário`, utilizada com o método GET, recupera as informações de um usuário (ou usuários).

Em sistemas REST, as URIs, como citado, devem conter apenas substantivos, que são os recursos: `/restaurante/adiciona` não é uma boa URI, pois contém um verbo e não está identificando um recurso, mas sim uma operação.

Por último, conforme a figura acima, temos o componente dados, que nada mais é do que a definição do formato de dados que optamos por usar em determinado *web service*.

# Resumindo

- HTTP significa *Hypertext Transfer Protocol*.
- O HTTP é o principal protocolo de comunicação utilizado na internet. É usado para gerenciar as requisições (cliente) e as respostas (servidor) pela internet.
- O HTTPS é basicamente o HTTP, porém com uma camada adicional de segurança/criptografia.
- É importante usar HTTPS ao passar informações confidenciais ou pessoais pela *web*.
- O certificado digital é a identidade eletrônica de uma pessoa ou empresa.
- Verbos HTTP definem os métodos de como a requisição será tratada.
- Os principais verbos/métodos são: GET, PUT, POST e DELETE.
- O HTTP trabalha com códigos de retorno para identificar o resultado de uma requisição.
- Os códigos são divididos em grupos: 1xx: Informação, 2xx: Sucesso, 3xx: Redirecionamento, 4xx: Erro do Cliente e 5xx: Erro do Servidor.
- As requisições HTTP podem ser utilizadas com parâmetros, fornecendo informações adicionais ao servidor.
- São conceitos importantes do HTTP: Domínio, Subdomínios, Recursos, Endereço IP, DNS, Portas, URL, URI, URN, Sessões e *Cookies*.
- O HTTP se baseia na arquitetura Cliente-Servidor e usa o modelo Requisição-Resposta.
- *Web services* são aplicações que disponibilizam serviços (dados) para outras.
- REST é uma arquitetura para interface (comunicação) baseada em 3 componentes (Recursos, Operações e Dados).

## **PARTE II – HTTP/2**

## Capítulo 10: HTTP/2

Até agora, temos falado sobre a versão HTTP 1.1, especificada nos anos 1990. Desde então, a forma como utilizamos a internet mudou drasticamente.

A chamada WEB 1 se caracterizava pela estaticidade. Isto é, não havia interação entre o usuário e os *sites*. Podemos dizer que era utilizada apenas para leitura (usuário passivo).

Com o advento da WEB 2, passamos a ter interatividade, ou seja, os *sites* passaram de estático para dinâmicos (usuário ativo), porém ainda de forma centralizada, com os *sites* hospedados em servidores. Aqui, é importante destacar o surgimento das aplicações *mobile*, advento dos *smartphones* e aplicativos móveis.

Neste momento, estamos iniciando no que chamamos de WEB 3, na qual a principal característica é a descentralização, em outras palavras, o cliente passa a atuar também como servidor.

Para acompanhar a mudança da forma de se utilizar a internet o protocolo HTTP evoluiu, surgindo então a versão 2, nomeada HTTP/2 nos idos de 2015.

O HTTP/2 traz inúmeras evoluções, que visam otimizar a performance, aumentar a segurança e simplificar o desenvolvimento de aplicações.

Um dos gargalos das aplicações *web* é a velocidade de resposta: quando maior a quantidade de dados transmitidas nas requisições ou respostas, maior a chance de um possível problema de performance.

Uma das saídas para minimizar este problema no HTTP 1.1 foi a habilitação do GZIP no servidor, que compacta os dados das respostas. Tal ação, porém, deve ser habilitada de forma explícita, pois não é automática.

No HTTP/2 o GZIP é habilitado por padrão e é obrigatório.

Note que, no HTTP 1.1, a compactação só ocorre no corpo das respostas. Os cabeçalhos (*headers*), tanto nas respostas como nas requisições, continuam trafegando de forma textual, sem compressão.

No HTTP/2, por padrão, os cabeçalhos são binários, compactados através do algoritmo HPACK, reduzido o volume de dados trafegados nos cabeçalhos.

Na questão de segurança, no HTTP/2, o TLS passa a ser padrão e mandatório, ou seja, não há mais a forma não segura do HTTP 1.1. Assim, só há sites habilitados com o HTTPS.

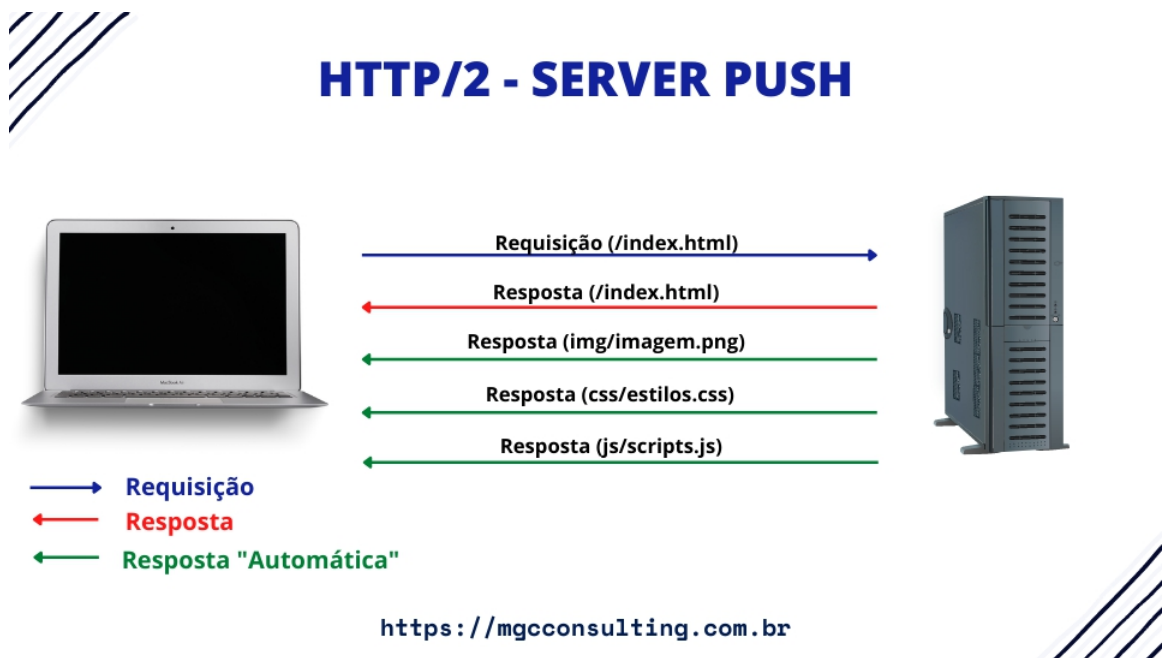
Outra mudança foi em relação ao estado dos cabeçalhos. No HTTP 1.1, os cabeçalhos não guardavam as informações das requisições anteriores (*stateless*): em toda requisição, o cabeçalho é reenviado.

No HTTP/2, os cabeçalhos são *statefull*. Em outras palavras, apenas os cabeçalhos que mudam são enviados, o que economiza tráfego, pois cabeçalhos que não sofreram alterações não são transmitidos.

## SERVER-PUSH

Vamos imaginar agora uma requisição de uma página feita via HTTP 1.1. Após a primeira requisição, o servidor retorna a página HTML. De posse desses dados, o navegador identifica que precisa para renderizar a página de uma imagem e de um arquivo de estilos (css). São então realizadas novas requisições para a obtenção desses recursos adicionais.

No HTTP/2, esse processo foi modificado. Ao requisitar uma página, o servidor, além de retornar o conteúdo inicial, “entende” que recursos adicionais serão necessários, e já os envia.



O cliente então “nota” que recebeu informações necessárias (que teria que solicitar via novas requisições), e já as utiliza sem realizar novos pedidos.

Este conceito é chamado de *server push*, no qual o servidor envia dados para o cliente sem que haja o pedido explícito (mas que serão necessários), visando otimizar o tráfego de dados.

Em resumo, o HTTP 1.1 lida com requisições seriais; já o HTTP/2, com paralelas.

## KEEP-ALIVE

Agora, uma última diferença. Como já citado, no HTTP 1.1, o par requisição/resposta é único. Quando uma requisição é feita é aberta uma conexão TCP, que é o protocolo de transporte. Este recurso, embora rápido em termos computacionais, demora a ser alocado. Porém, é necessário para a requisição HTTP seguir adiante.

Para minimizar este “problema”, o HTTP, ao invés de abrir e fechar um canal TCP a cada requisição, o que seria lógico, dado que cada requisição é única, mantém a conexão TCP aberta por um determinado tempo. Em outras palavras, conseguimos realizar uma série de requisições numa única conexão TCP.

Este recurso é chamado de *keep-alive*.

Os *browsers* atuais têm a capacidade de manter entre 4 e 8 de conexões simultâneas por domínio.

Então, o *keep-alive* também é implementado no HTTP/2, mas com uma diferença: o cliente pode realizar uma série de requisições antes de receber a resposta da primeira requisição. Ou seja, no HTTP 1.1 a comunicação se dá de forma síncrona, já no HTTP/2 de forma assíncrona.

O cliente vai realizando requisições e recebendo as respostas conforme o servidor consegue processar.

Esse conceito no HTTP2 também é chamado de *multiplexing*.

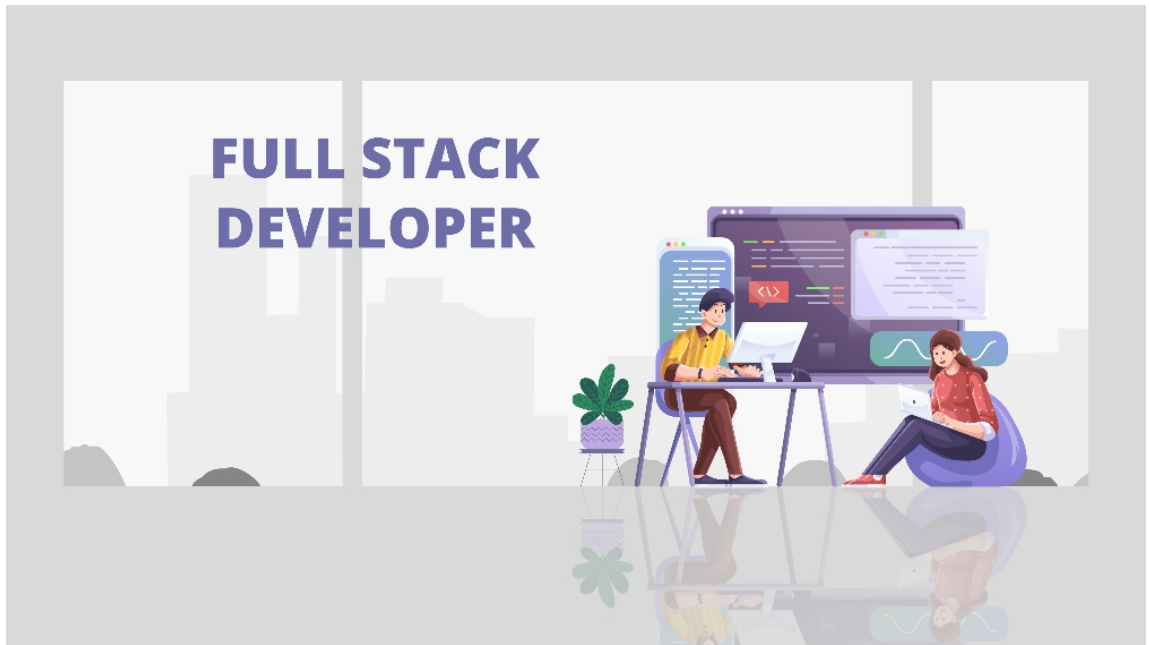


## Resumindo

- HTTP/2 é a evolução do HTTP 1.1.
- Esta evolução veio para atender as necessidades da forma como usamos a internet.
- Passamos pela WEB 1 (estática) e atualmente estamos na transição da WEB 2 (dinâmica/centralizada) para a WEB 3 (dinâmica/descentralizada).
- No HTTP/2, recursos de compactação (GZIP/HPACK) e segurança (TLS) são habilitados por padrão e obrigatórios.
- Os cabeçalhos das requisições, no HTTP/2, são *statefull*.
- O HTTP/2 implementa nativamente os recursos *server push* e *keep-alive*.

# **PARTE III – TRILHA DE APRENDIZADO PARA O DESENVOLVEDOR WEB**

# Capítulo 11: *Roadmap* do Desenvolvedor Web



## **Aprenda Inglês**

Antes de tudo, aprenda inglês. Seja fluente. Senão ao menos leia e escreva bem. Querendo ou não, é a língua padrão na comunidade de TI (Tecnologia da Informação).

Hoje, contamos com vários recursos que, teoricamente, ajudam nesta questão do idioma. Digo teoricamente pois, no meu entender, há inúmeros termos que não deveriam ser traduzidos, e quando o são, geram mais confusão do que clareiam o entendimento.

Mas não vamos nos alongar neste ponto. Fica a dica. Aprenda inglês.

## **Cultive Bons Hábitos**

Ainda, antes de entrar especificamente quais conceitos e ferramentas você deve dominar, vou falar um pouco sobre hábitos.

Este não é um livro de autoajuda, longe disto. Mas procure cultivar bons hábitos. Nada radical. Não precisa acordar as 5h da manhã e tomar banho frio. Procure apenas ser equilibrado.

Apenas como sugestão, que tal:

1. Menos Netflix, mais vídeos técnicos!
2. Menos consumo, mais estudo!
3. Menos gastos, mais investimentos!
4. Menos falar, mais fazer!
5. Menos reclamação, mais gratidão!
6. Menos zona de conforto, mais desafios!

### **Trilha Full-Stack**

Bem, a formação de um profissional de TI é um ramo bastante variado e complexo. Aqui, vamos focar no que penso ser o mínimo para que você se torne um desenvolvedor *web full-stack*.

Recordando, de forma simples, podemos dividir em 3 ramos o desenvolvimento para internet:

1. *Front-end*: Parte cliente, o que o usuário vê e interage. Interface Gráfica.
2. *Back-end*: Lado servidor, o que mantém a aplicação: Lógica da Aplicação, Processos e Banco de Dados.
3. *Full-stack*: *Front-end* + *Back-end*.

Então para uma formação *full-stack*, sugiro:

### **CONCEITOS BÁSICOS:**

- O que é Desenvolvimento *Web*?
- Como a *Web* funciona? (Aqui se encaixa este livro)

### **Como Aprender?**

Recomendo fortemente, principalmente se você não possui nenhum conhecimento em TI, que faça o curso CS50 da Universidade de Havard. Disponível em português gratuitamente no *site* Estudar na Prática, da Fundação Estudar (<https://materiais.napratica.org.br/cc50>).

Versão em inglês em:

*<https://www.edx.org/course/introduction-computer-science-harvardx-cs50x>.*

## FRONT-END BÁSICO:

- HTML
- CSS
- JavaScript

Como sugestão, também de forma gratuita:

*<https://www.freecodecamp.org/learn/2022/responsive-web-design/> (Em inglês, HTML e CSS).*

*<https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/> (Em inglês, Javascript).*

# FRONT-END AVANÇADO:

- Bootstrap (e/ou Tailwind)
- React (e/ou Angular e/ou Vue e/ou Next.js)
- Git e Github
- SCSS

Neste ponto há uma infinidade de recursos de aprendizado grátis. Recomendo que você busque no Google, e escolha o mais adequado para você.

# BACK-END PROGRAMAÇÃO<sup>[2]</sup>:

- PHP + Laravel e/ou
- Python + Django e/ou
- Java + Spring/Play/Hibernate

# BACK-END BANCO DE DADOS:

- Relacionais:
  - mySQL e/ou
  - PostgreSQL e/ou
  - SQL Lite
- Não Relacionais:
  - MongoDB

Para BACK-END prefiro deixar a sugestão que você pesquise no Google, pois vai depender da ferramenta escolhida,

Sugiro também que você busque na internet o significado de alguma ferramenta que você não tenha o conhecimento.

Deixo aqui a indicação do site <https://roadmap.sh/>, que fornece roteiros/trilhas específicos para diversos ramos de atuação na área de TI.

Encerro indicando o blog <https://mgcconsulting.com.br/>, onde escrevo sobre desenvolvimento web. Lá você encontrará várias dicas e sugestões para seu projeto web.

Opa, uma última dica: Desenvolva sempre que possível, a prática é fundamental para consolidar os conceitos teóricos. Pessoalmente acho que a prática constante seja de 70% a 80% numa formação de programação web.

# CONCLUSÃO

Procurei dar uma visão geral sobre o protocolo HTTP.

É virtualmente impossível cobrir todo o conteúdo relacionado ao HTTP em um livro curto como este.

Entendo que aqui foram passados os principais conceitos do HTTP que um Desenvolvedor Web precisa dominar, pois acredito que mais detalhes sobre o funcionamento do protocolo HTTP são necessários apenas para profissionais que irão lidar com infraestrutura de redes.

Por último, tentei deixar uma humilde sugestão sobre a trilha de conhecimento que acho necessária para você se tornar um desenvolvedor completo de aplicações *web*.

Espero que tenha apreciado a leitura.



# REFERÊNCIAS

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP>

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods>

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>

<https://developer.mozilla.org/pt-BR/docs/Glossary/URI>

[https://www.w3schools.com/tags/ref\\_httpmessages.asp](https://www.w3schools.com/tags/ref_httpmessages.asp)

<https://suporte.hostgator.com.br/hc/pt-br/articles/11500189157>

<https://suporte.hostgator.com.br/hc/pt-br/articles/115001891573>

<https://www.hostgator.com.br/blog/o-que-sao-os-cookies-da-web/>

<https://www.kaspersky.com.br/resource-center/definitions/what-is-an-ip-address>

<https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api>

# SOBRE O AUTOR



Marcelo Galhego é profissional de TI e tem mais de 30 anos de experiência. Com formação pela PUC-RJ e MBI em Gestão Estratégica de TI pelo NCE/UFRJ, é PMP e certificado pela ITIL Foundation. Especialista em Auditoria e Qualidade, possui as certificações AU - Management Systems Auditing (ISO 19011:2018), Connected Learning Live ISO 9001:2015 Auditor Líder (TPECS), QM - Sistemas de Gestão da Qualidade (Requisitos ISO 9001:2015) e TL - Leading Management Systems Audit Teams v3.1.

---

[1] <https://viacep.com.br/>

[2] Linguagem de Programação e Framework.