HTTP – Entendendo a Web por baixo dos panos

Assuntos abordados

* **HTTPS, que é a web segura**
* **HTTP trafega texto puro**
* **HTTP2 Nova versão e o que difere das anteriores**
* **Endereços, incluindo domínios, recursos e portas**
* **Sessão, cookie e o modelo de requisição e resposta do HTTP, mais ainda os parâmetros que são enviados na requisição, seja no seu corpo ou na URL.**
* **Serviços REST**

Fundamentos da web

A interação é realizada através de protocolos onde o principal é a especificação HTTP-**Hypertext Transfer Protocol**

onde o navegador e o Servidor provido via Internet. Repeita um modelo arquitetural chamado Client-Server

HTTP é o protocolo mais importante da Internet

Navegadores Http – Chrome,Firefox,IE/Edge,Opera

Em qualquer comunicação é preciso existir algumas regras para que as duas partes consigam se entender com sucesso. Pensando na comunicação do seu navegador entre algum outro site esse conjunto de regras é basicamente um protocolo, onde neste cenário é o HTTP.

A principal especificação do HTTP, esta no seguinte endereço: <https://tools.ietf.org/html/rfc2616>

Para que alguém consiga se comunicar com você, esse alguém deverá usar o idioma do qual a outra pessoa conhece. Isso significa que, sua regra (protocolo) de comunicação com o mundo é a língua portuguesa, que define a forma com que as informações devem chegar até você (através do vocabulário, regras de gramática e etc). Uma outra pessoa que conheça português irá usar do mesmo formato, já que vocês possuem um idioma em comum.

Na internet, como já vimos, o idioma mais comum é o HTTP. Ele é responsável por definir a forma de como os dados são trafegados na rede através de várias regras. Portanto, todo mundo que conhece o idioma HTTP poderá receber e enviar dados e participar dessa conversa!

P2P – Peer-To-Peer - Utilizado em torrents para transferencia de arquivos entre dois ou mais clientes

O modelo Cliente-Servidor tenta centralizar o trabalho no servidor, mas isso também pode gerar gargalos. Se cada Cliente pudesse ajudar no trabalho, ou seja, assumir um pouco da responsabilidade do servidor, seria muito mais rápido. Essa é a ideia do P2P! Não há mais uma clara divisão entre Cliente-Servidor, cada cliente também é servidor e vice-versa!

File Transport Protocol, protocolo para transferir arquivos

Simple Mail Transfer Protocol, protocolo para enviar e-mails.

smb realmente existe e é a abreviação de Server Message Block. Ele é utilizado para compartilhar arquivos dentro de uma rede local.

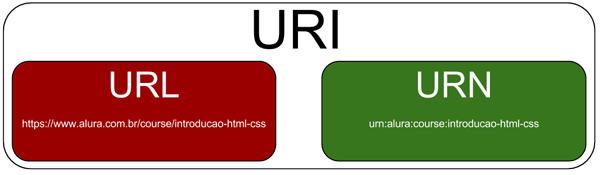
[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/http-fundamentos/task/25455/next)

Muitas vezes, desenvolvedores usam a sigla URI (Uniform Resource Identifier) quando falam de endereços na web. Alguns preferem URL (Uniform ResourceLocator), e alguns misturam as duas siglas à vontade. Há uma certa confusão no mercado a respeito e mesmo desenvolvedores experientes não sabem explicar a diferença. Então, qual é a diferença?

Resposta 1 (fácil): Uma URL é uma URI. No contexto do desenvolvimento web, ambas as siglas são válidas para falar de endereços na web. As siglas são praticamente sinônimos e são utilizadas dessa forma.

Resposta 2 (mais elaborada): Uma URL é uma URI, mas nem todas as URI's são URL's! Existem URI's que identificam um recurso sem definir o endereço, nem o protocolo. Em outras palavras, uma URL representa uma identificação de um recurso (URI) através do endereço, mas nem todas as identificações são URL's.

Humm ... ficou claro? Não? Vamos dar um exemplo! Existe um outro padrão que se chama URN (Uniform Resource Name). Agora adivinha, os URN's também são URI's! Um URN segue também uma sintaxe bem definida, algo assim urn:cursos:alura:course:introducao-html-css. Repare que criamos uma outra identificação do curso Introdução ao HTML e CSS da Alura, mas essa identificação não é um endereço.



Novamente, a resposta 2 vai muito além do que você realmente precisa no dia a dia. Normalmente URL e URI são usados como sinônimos.

HTTPS

Quando o navegador pede informações ao servidor, nessa comunicação há vários intermediários. Por exemplo, usando uma conexão Wi-Fi, os dados do navegador passam primeiro para o roteador Wi-Fi, e do roteador passam para o modem do provedor, do modem para algum servidor do provedor de internet.

o HTTPS, que basicamente é o HTTP comum, porém com uma camada adicional de segurança/criptografia que antes era SSL, mas posteriormente passou a ser também TLS. É muito comum que estas duas siglas sejam encontradas juntas como SSL/TLS por se tratarem da mesma questão de segurança. Sendo assim, temos dois termos:

1. HTTP: HyperText Transfer Protocol
2. SSL/TLS: Secure Sockets Layer / Transport Layer Security

**Cerfificado digital no mundo web é a identidade do Site**

Quando precisamos informar nossos dados a algum servidor, queremos ter certeza que este servidor realmente representa a entidade em questão. Queremos confiar em quem estamos fornecendo nossos dados!

Um certificado digital prova uma identidade para um site, onde temos informações sobre o seu domínio e a data de expiração desse certificado.

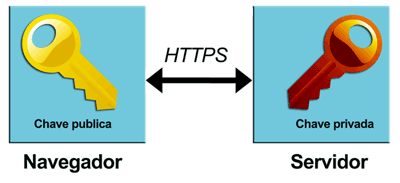
Além disso, o certificado ainda guarda a chave pública que é utilizada para criptografar (cifrar) os dados que são trafegados entre cliente e servidor.

O HTTPS para garantir segurança usa criptografia baseada em chaves públicas e privadas e para gerar essas chaves publicas e privadas é preciso garantir a identidade de quem possui essas chaves e isso é feito a partir de um certificado digital, ou seja, um certificado digital é utilizado para identificar determinada entidade e ainda é utilizada para geração das chaves de criptografia.

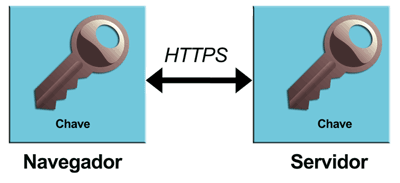
Apesar disso, ainda é necessário que uma autoridade certificadora, que nada mais é que um órgão ou entidade confiável, garanta não apenas a identidade do site mas também a validade do certificado. No caso da Alura a autoridade certificadora é a COMODO RSA Domain Validation, mas existem outras.

a principal função de uma entidade certificadora é garantir que os certificados que estão sendo utilizados podem ser confiados.

o HTTPS usa uma chave pública e uma chave privada. As chaves estão ligadas matematicamente, o que foi cifrado pela chave pública só pode ser decifrado pela chave privada. Isso garante que os dados cifrados pelo navegador (chave pública) só podem ser lidos pelo servidor (chave privada). Como temos duas chaves diferentes envolvidas, esse método de criptografia é chamado de criptografia assimétrica. No entanto, a criptografia assimétrica tem um problema, ela é lenta.



Por outro lado, temos a criptografia simétrica, que usa a mesma chave para cifrar e decifrar os dados, como na vida real, onde usamos a mesma chave para abrir e fechar a porta. A criptografia simétrica é muito mais rápida, mas infelizmente não tão segura. Como existe apenas uma chave, ela ficará espalhada pelos clientes (navegadores) e qualquer um, que tem a posse dessa chave, pode decifrar a comunicação.



Agora, o interessante é que o HTTPS usa ambos os métodos de criptografia, assimétrica e simétrica. Como assim? Muita calma, tudo o que aprendemos é verdade! Só faltou o grande final :)

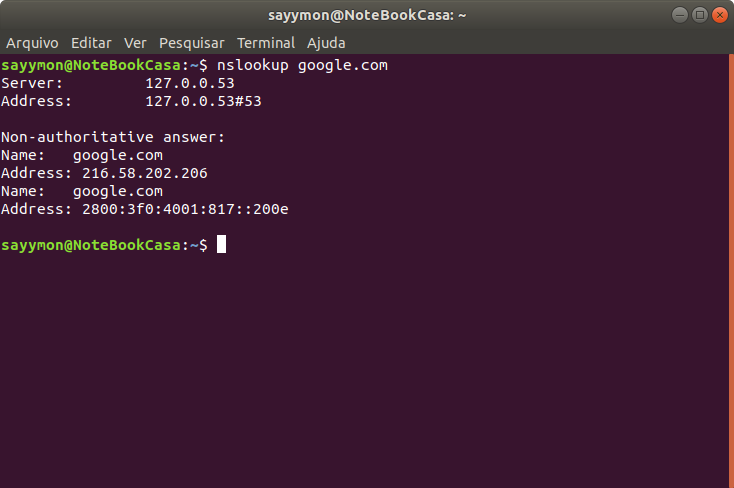
No certificado, vem a chave pública para o cliente utilizar, certo? E o servidor continua na posse da chave privada, ok? Isso é seguro, mas lento e por isso o cliente gera uma chave simétrica ao vivo. Uma chave só para ele e o servidor com o qual está se comunicando naquele momento! Essa chave exclusiva (e simétrica) é então enviada para o servidor utilizando a criptografia assimétrica (chave privada e pública) e então é utilizada para o restante da comunicação.

Então, HTTPS começa com criptografia assimétrica para depois mudar para criptografia simétrica. Essa chave simétrica será gerada no início da comunicação e será reaproveitada nas requisições seguintes. Bem-vindo ao mundo fantástico do HTTPS :)

//////

Endereções DNS - Domain Name System).

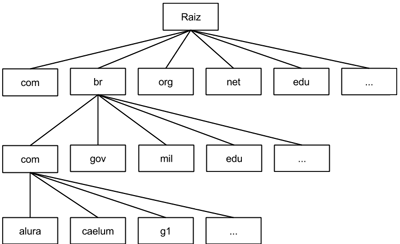
Converte o IP em endereço de nome com os dominios

nslookup google.com – Retorna as informações de Adress IP do endereços

Dominios

## **Analisando o domínio**

Vamos dar uma olhada com mais carinho no nome do domínio. Olhando da direita para a esquerda, o domínio começa com br, indicando um site do Brasil. O brrepresenta o top level domain, está na raiz do domínio. Depois vem o com, abreviação de comercial e alura. O com e o alura são sub-domínios.



O www representa também um sub-domínio, no entanto seu uso é opcional, tanto que alura.com.br e www.alura.com.br funcionam e mostram a mesma página. A maioria dos site usam o prefixo www e podemos dizer que isso é algo legado que continua ser popular apesar de não ser necessário.

## **Subdomínios**

Existe também a ideia de subdomínios, que representam sessões específicas dentro de um site. Por exemplo, no caso do Gmail temos o endereço: mail.google.com , ou ainda no caso do Google Drive: drive.google.com. Tanto Gmail como Drive são subdomínios do domínio Google.

Perceba que esses subdomínios apontam para páginas diferentes dentro do mesmo domínio (Google).

## **Endereços IP's**

O nome do domínio é organizado em uma hierarquia que foi criada para organizar os sites na internet e para a gente ter algo fácil para se lembrar. Para ser correto, a internet funciona na verdade sem esses domínios. Aqueles nomes são coisas dos humanos, as máquinas na internet têm uma outra forma de se endereçar. Elas usam o que se chama endereços de IP, que nada mais são do que números - muito difícil para gente decorar.

Podemos inclusive escolher um servidor DNS de preferência na nossa internet. Um bastante usado é o da própria Google: <https://developers.google.com/speed/public-dns/>

PORTAS :

## **Abrindo portas**

A porta reservada para o protocolo HTTP é o 80. Novamente um número, e como o navegador já sabe essa porta padrão, podemos omiti-la, mas nada nos impede adicioná-la no endereço, por exemplo:

http://www.alura.com.br:80

Vai funcionar normalmente, tanto que o navegador esconde automaticamente :80. Vamos tentar uma outra porta, outro numero, por exemplo 81:

http://www.alura.com.br:81

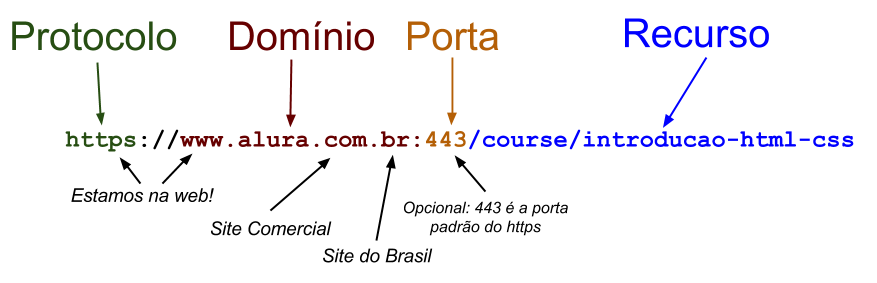
Não funciona, pois essa porta não está aberta no servidor, não podemos estabelecer uma conexão e o tempo de conexão vai esgotar. Igualmente, o protocolo HTTPS possui uma porta padrão, a porta 443, que também podemos omitir ao acessarmos um endereço HTTPS. Podemos testar também e ver que funciona normalmente.

https://www.alura.com.br:443

o FTP que usa 21ou SSH que usa 22.

## **Finalmente, a URL**

Repare que estamos usando umas regras bem definidas para descrever a localização de um recurso na web. Todos os endereços na web sempre seguem esse mesmo padrão: protocolo://dominio:porta/caminho/recurso. Esse padrão, na verdade, segue uma especificação que foi batizada de Uniform Resource Locator, abreviada como URL. Então, as URLs são os endereços na web!



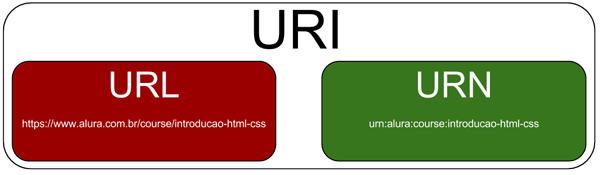
[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/http-fundamentos/task/25455/next)

Muitas vezes, desenvolvedores usam a sigla URI (Uniform Resource Identifier) quando falam de endereços na web. Alguns preferem URL (Uniform ResourceLocator), e alguns misturam as duas siglas à vontade. Há uma certa confusão no mercado a respeito e mesmo desenvolvedores experientes não sabem explicar a diferença. Então, qual é a diferença?

Resposta 1 (fácil): Uma URL é uma URI. No contexto do desenvolvimento web, ambas as siglas são válidas para falar de endereços na web. As siglas são praticamente sinônimos e são utilizadas dessa forma.

Resposta 2 (mais elaborada): Uma URL é uma URI, mas nem todas as URI's são URL's! Existem URI's que identificam um recurso sem definir o endereço, nem o protocolo. Em outras palavras, uma URL representa uma identificação de um recurso (URI) através do endereço, mas nem todas as identificações são URL's.

Humm ... ficou claro? Não? Vamos dar um exemplo! Existe um outro padrão que se chama URN (Uniform Resource Name). Agora adivinha, os URN's também são URI's! Um URN segue também uma sintaxe bem definida, algo assim urn:cursos:alura:course:introducao-html-css. Repare que criamos uma outra identificação do curso Introdução ao HTML e CSS da Alura, mas essa identificação não é um endereço.



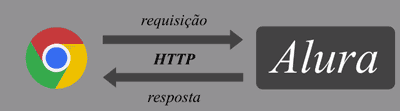
Novamente, a resposta 2 vai muito além do que você realmente precisa no dia a dia. Normalmente URL e URI são usados como sinônimos.

Já descobrimos que o HTTP é um protocolo que define as regras de comunicação entre cliente e servidor e de que as URLs são constituídas. Porém isso não é tudo, vejamos mais alguns detalhes sobre o funcionameto da Web e do HTTP.

Realizaremos um teste efetuando login na Alura. Quando preenchemos o formulário e clicamos no botão, o navegador envia o nosso login e a nossa senhapara o servidor através do protocolo HTTP! Vamos detalhar um pouco esta ação.

No mundo HTTP, a requisição enviada pelo navegador para o servidor é chamada de HTTP REQUEST. Recebemos a página /dashboard como resposta já que enviamos login e senha válidos. No mundo HTTP essa resposta é chamada de HTTP RESPONSE.

A comunicação segue sempre esse modelo: o cliente envia uma requisição e o servidor responde. Requisição e Resposta ou em inglês: Request-Response. Aqui é importante saber que a comunicação sempre começa com o cliente: é ele quem pede as informações. O servidor responde apenas o que foi requisitado e nunca inicia a comunicação!



## **Comunicação sem estado**

Vamos acessar rapidamente outro site: http://g1.globo.com. Para este acesso estamos enviando uma requisição para g1 e recebemos como resposta a página inicial.

Agora vamos navegar dentro do site e acessar algum artigo. Ao clicarmos enviamos uma nova requisição e percebemos que TODA página foi trocada. Fica mais claro ainda se acessarmos do menu acima algum link (globo esporte ou globo show). Podemos ver que todo o conteúdo do site foi trocado.

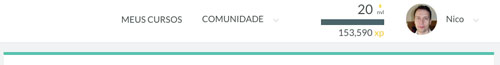
Isso também acontece no caso da Alura (talvez um pouco mais difícil de perceber). Ao acessarmos recursos diferentes todo o conteúdo no navegador foi trocado (apesar do menu parecer o mesmo, ele também foi trocado). A ideia do HTTP é justamente essa, cada recurso é independente do outro e não depende do anterior. Isso também se aplica para os dados enviados na requisição. Cada requisição é independente da outra e ela sempre deve conter todas informações para o servidor responder.

Pense que HTTP funciona como o envio de cartas pelo correio e uma carta representa uma requisição. Você fez uma viagem e gostaria de enviar 3 cartas para sua mãe. Adianta falar para os correios "eu vou colocar o endereço apenas na primeira carta, ai vocês já sabem para onde enviar a segunda e terceira carta"? Não adianta pois os correios tratam cada carta independentemente, e assim também funciona o HTTP. Cada requisição (carta) precisa ter todas as informações. A mesma coisa se aplica para a resposta, precisa ter todas as informações.

Essa característica de cada requisição ser independente é chamada de stateless. É esse nome bonito mesmo! O HTTP é um protocolo que não mantém o estado de requisições. Isso significa que só com HTTP não há como se lembrar das requisições anteriores enviadas para o servidor. Por isso precisamos incluir em cada requisição todas as informações, sempre. Para o desenvolvedor este conhecimento é importante pois é justamente essa característica stateless que o atrapalha no dia a dia. Ele precisa preparar a aplicação web para que funcione bem usando o protocolo HTTP, algo que veremos nos treinamentos da Alura.

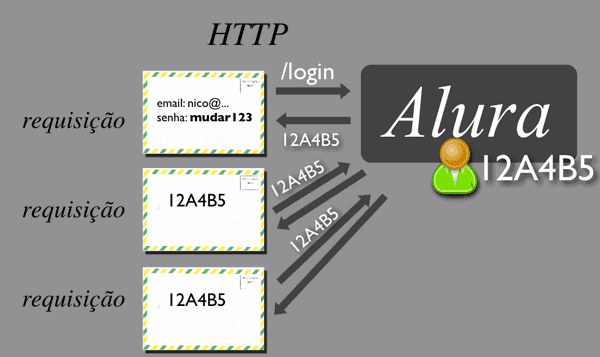
## **Lidando com Sessões**

Reparem que, mesmo após termos realizado o login e termos enviado várias requisições, aparece o ícone com a minha imagem no menu principal.



Ou seja, a Alura se lembra de alguma forma que eu fiz login em alguma requisição anterior. Como falamos antes, cada requisição deve enviar todas as informações para gerar a resposta. Isso significa que o navegador envia em cada requisição informações sobre o meu usuário! Se cada requisição for independente uma da outra, e não tiver como se lembrar das requisições anteriores, não tem outra explicação a não ser que o navegador envie os dados sobre o meu usuário em cada requisição! Lembre-se da carta postal, ela sempre precisa ter os dados do remetente e aqui não é diferente!

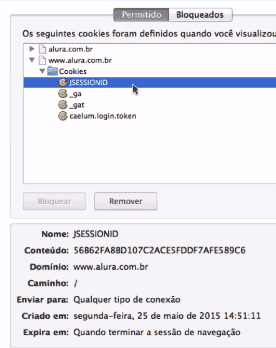
Então o navegador envia o login e senha em cada requisição? Não, não seria muito elegante nem seguro fazer isso. Mas ele faz algo parecido, acreditem ou não. Quando efetuamos o login, a Alura valida os nossos dados, certo? Nesse momento, o servidor tem certeza que o usuário existe e gera uma identificação quase aleatória para o usuário. Essa identificação é um número criado ao vivo e muito difícil de adivinhar. Esse numero é a identificação temporária do usuário e ele será devolvido na resposta.



## **Conhecendo cookies**

Então onde está esse número? O navegador grava esse número em um arquivo especial para cada site, são os famosos cookies. Como acessar esse cookie depende um pouco do navegador em uso. O mais importante é entender o porquê da existência desse número e onde ele foi gravado.

No Chrome podemos ver todos os cookies armazenados nas Configurações -> Privacidade -> Configurações de conteúdo... -> Todos os cookies e dados de site.... Se procurarmos por Alura, em cursos.alura.com.br, lá temos um cookie com o nome caelum.login.token, que contém o número da identificação. Se apagarmos esse cookie, perderemos nossa identificação, sendo assim, a Alura exigirá um novo login pois não saberá que já tínhamos logado.



Normalmente o nome do cookie é algo como session-id, dependendo da plataforma de desenvolvimento utilizada ele pode se chamar de PHPSESSID ou ASP.NET\_SessionId ou JSESSIONID ou outro nome que foi inventado! O Cookie será gerado de forma transparente pela tecnologia que você for utilizar para criar aplicativos web. Esse nome, PHPSESSIONID, JSESSIONID ou outro, é gerado pela ferramenta de gerenciamento de Sessão. Por isso ela muda o nome. Se você está usando PHP, então o PHP gerará o nome do Cookie e seu identificador (número aleatório) e chamará o cookie PHPSESSIONID. No Java já será usado o nome JSESSIONID.

Resumindo, todas as plataformas ajudam a gerar esse número e a criar o cookie de maneira transparente. É dessa forma que as plataformas gerenciam as SESSÕEScom o usuário. Como isso funciona de modo concreto você aprenderá nos cursos e carreiras específicas.

A ideia de manter dados entre requisições é algo muito comum no desenvolvimento de aplicações na web. Um usuário que se loga no sistema web causa a criação de uma sessão. Uma sessão então é útil para guardar informações sobre o usuário e ações dele. Um exemplo clássico é um carrinho de compras. Entre várias requisições estamos usando o mesmo carrinho de compras que guarda os nossos produtos escolhidos (fizemos uma sessão de compras online).

Resumindo teremos:

* O HTTP usa sessões para salvar informações do usuário
* Sessões só são possíveis por uso de Cookies
* Cookies são pequenos arquivos que guardam informações no navegador
* O HTTP é stateless, não mantem estado.

Uma sessão HTTP nada mais é que um tempo que o cliente permanece ativo no sistema! Isso é parecido com uma sessão no cinema. Uma sessão, nesse contexto, é o tempo que o cliente usa a sala no cinema para assistir a um filme. Quando você sai da sala, termina a sessão. Ou seja, quando você se desloga, a Alura termina a sua sessão.

COOKIES

Quando falamos de Cookies na verdade queremos dizer Cookies HTTP ou Cookie web. Um cookie é um pequeno arquivo de texto, normalmente criado pela aplicação web, para guardar algumas informações sobre usuário no navegador. Quais são essas informações depende um pouco da aplicação. Pode ser que fique gravado alguma preferência do usuário. Ou algumas informações sobre as compras na loja virtual ou, como vimos no vídeo, a identificação do usuário. Isso depende da utilidade para a aplicação web.

Um cookie pode ser manipulado e até apagado pelo navegador e, quando for salvo no navegador, fica associado com um domínio. Ou seja, podemos ter um cookie para www.alura.com.br, e outro para www.caelum.com.br. Aliás, um site ou web app pode ter vários cookies!

Podemos visualizar os cookies salvos utilizando o navegador. Como visualizar, depende um pouco do navegador em questão:

No Chrome: Configurações -> Privacidade -> Configurações de conteúdo... -> Todos os cookies e dados de site... -> Pesquisar alura

No Firefox: Preferências -> Privacidade -> remover cookies individualmente -> Pesquisar alura

<https://httpstatuses.com/>

Methodos HTTP GET,POST,PUT e DELETE

Quando enviamos parâmetros na URL, devemos iniciar pelo ?, o nome do parâmetro e um =, para separar o nome do parâmetro do seu valor:

?nome\_do\_parametro=seu\_valor

Quando usamos mais do que, um parâmetro devemos usar & para separá-los:

?nome\_do\_parametro=seu\_valor&nome\_do\_outro\_param=valor

Se o GET foi criado para receber dados, e o POST para adicionar algo no servidor, será que não existe algo para apagar e atualizar?

A resposta é sim, e os métodos se chamam DELETE e PUT.

Em alguns cabeçalhos do HTTP devemos especificar algum formato. Os formatos são chamados na documentação de MIME types. E na definição do cabeçalho usamos a seguinte estrutura: tipo/subtipo. São tipos conhecidos:

text, image, application, audio e video

E alguns subtipos:

text -> text/plain, text/html, text/css, text/javascript

image -> image/gif, image/png, image/jpeg

audio -> audio/midi, audio/mpeg, audio/webm, audio/ogg, audio/wav

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types>

Até agora sempre usamos o browser para realizar uma requisição. Mas podemos realizar fora dele usando a linha de comando por exemplo. Um programa famoso para isso é o CURL. No Linux e MacOS ele já vem instalado por padrão.

Caso esteja usando o Windows é necessário a instalação dele. O download deve ser feito por aqui: <https://curl.haxx.se/download.html>

Para realizar e depurar uma requisição via CURL podemos simplesmente executar no terminal o seguinte comando:

curl -v www.caelum.com.br

Uma saída típica dele seria:

Fabios-MacBook-Pro:~ fabiopimentel$ curl -v www.caelum.com.br

\* Rebuilt URL to: www.caelum.com.br/

\* Trying 172.217.29.51...

\* Connected to www.caelum.com.br (172.217.29.51) port 80 (#0)

> GET / HTTP/1.1

> Host: www.caelum.com.br

> User-Agent: curl/7.49.1

> Accept: \*/\*

>

< HTTP/1.1 200 OK

< Content-Type: text/html; charset=utf-8

< Vary: Accept-Encoding,User-Agent

< Content-Language: pt-br

< Content-Type: text/html;charset=UTF-8

< X-DNS-Prefetch-Control: on

< X-Cloud-Trace-Context: 3e5e270ee3ab1e79f81b10d2cdef53cd

< Date: Fri, 24 Mar 2017 19:20:12 GMT

< Server: Google Frontend

< Content-Length: 95776

<

<!DOCTYPE html>

<html class="no-js"lang="pt-br"> <head> <title>Caelum | Cursos de Java, .NET, Android, PHP, Scrum, HTML, CSS e JavaScript </title>

…

Pode-se notar pela saída que temos logo no começo as informações do request efetuado:

> GET / HTTP/1.1

> Host: www.caelum.com.br

> User-Agent: curl/7.49.1

> Accept: \*/\*

`

E após essas infos temos o cabeçalho da resposta obtida pelo servidor:

< HTTP/1.1 200 OK

< Content-Type: text/html; charset=utf-8

< Vary: Accept-Encoding,User-Agent

< Content-Language: pt-br

< Content-Type: text/html;charset=UTF-8

< X-DNS-Prefetch-Control: on

< X-Cloud-Trace-Context: 3e5e270ee3ab1e79f81b10d2cdef53cd

< Date: Fri, 24 Mar 2017 19:20:12 GMT

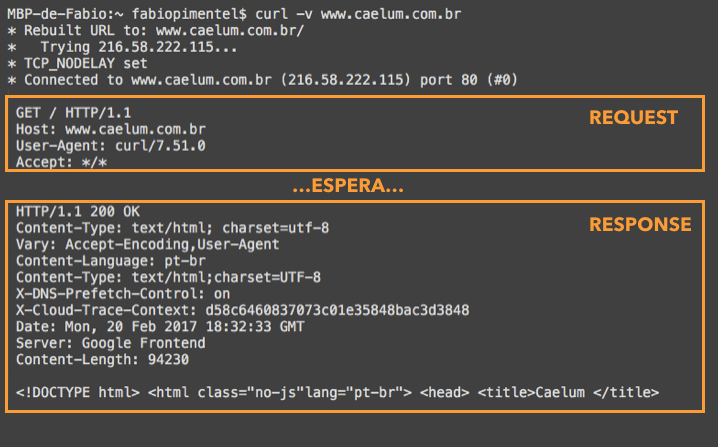
< Server: Google Frontend

< Content-Length: 95776

Logo depois vem o corpo da resposta (HTML da página requisitada):

<!DOCTYPE html> <html class="no-js"lang="pt-br"> <head> <title>Caelum | Cursos de Java, .NET, Android, PHP, Scrum, HTML, CSS e JavaScript </title> <meta name="viewport"content="width=device-width,initial-scale=1"> <meta name="format-detection"content="telephone=no"> <meta name="referrer"content="origin"> <meta name="description"content="A Caelum tem os cursos de Java, Scrum, Web, Front-end, PHP, .NET e Mobile mais reconhecidos no mercado, com didática diferenciada e instrutores qualificados."> <link rel="canonical"href="https://www.caelum.com.br/"> <style>.calendario .sem-turmas,.calendario-compacto .mais-turmas,.fm-message.fm-warning{font-style:italic}

Em resumo o output apresentando pelo CURL possui essa divisão:



# **HTTP/2**

O protocolo que estamos trabalhando até agora foi especificado na década de 90 e de lá até hoje muitas alterações foram feitas até na forma como usamos a internet.

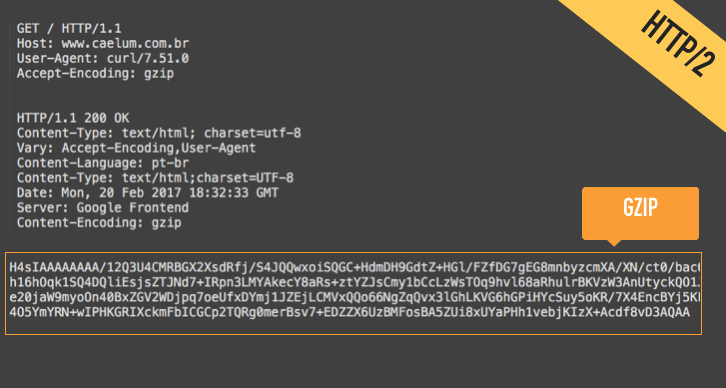
Com a chegada do mundo mobile novas preocupações apareceram e otimizações são cada vez mais necessárias para uma boa performance. Por isso uma mudança foi necessária e em 2015 depois de alguns anos de especificações e reuniões surgiu a versão 2 desse protocolo.

A nova versão é batizada de HTTP/2 e tem como página principal de documentação e referência essa: https://http2.github.io/.

A nova versão do protocolo HTTP traz mudanças fundamentais para a Web. Recursos fantásticos que vão melhorar muito a performance da Web além de simplificar a vida dos desenvolvedores.

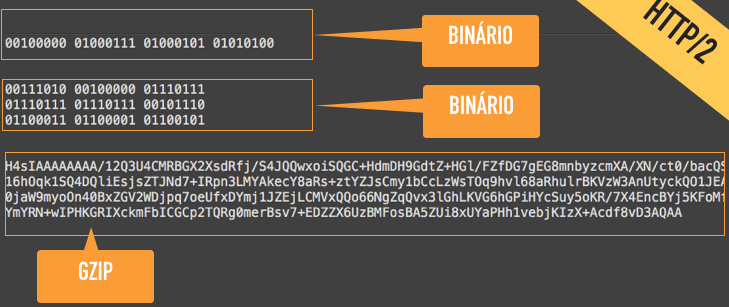
No HTTP 1.1, para melhorar a performance, habilitamos o GZIP no servidor para comprimir os dados das respostas. É uma excelente prática, mas que precisa ser habilitada explicitamente. No HTTP/2, o GZIP é padrão e obrigatório.

É como se a gente passasse a ter a resposta assim:

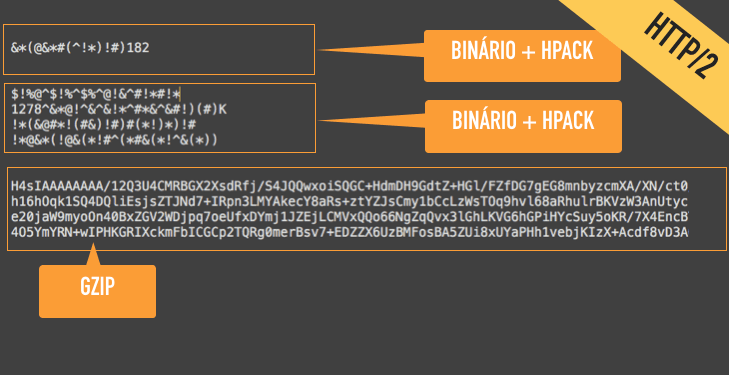


Mas, se você já olhou como funciona uma requisição HTTP, vai notar que só GZIPar as respostas resolve só metade do problema. Tanto o request quanto o responselevam vários cabeçalhos (headers) que não são comprimidos no HTTP 1.1 e ainda viajam em texto puro.

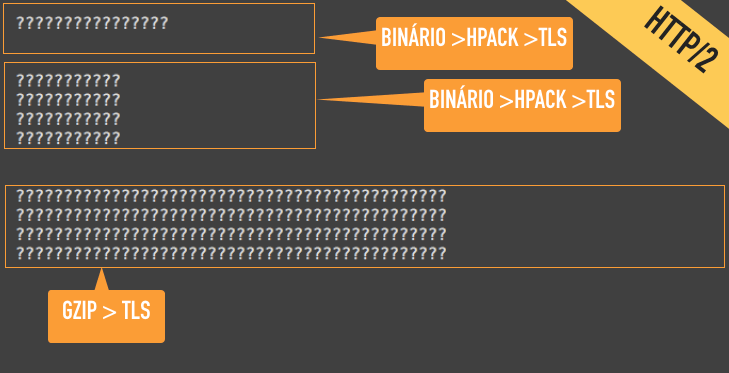
Já na nova versão, os headers passam a ser binários:



Além de binários eles são comprimidos usando um algoritmo chamado HPACK. Isso diminui bastante o volume de dados trafegados nos headers.



Além de todas essas otimizações para melhorar a performance ainda houve uma preocupação com segurança exigindo TLS por padrão também.



O HPACK é uma tecnologia especializada em comprimir os Headers das comunicações HTTP/2. Como toda requisição HTTP acompanha algum header por padrão, uma tecnologia de compressão embutida no protocolo é demasiadamente útil para economizar dados trafegados.

o HTTP/2 reforça bastante o uso do HTTPS, ao contrário do HTTP/1.1 em que isto era opcional. Apesar de não ser obrigatório em sua especificação, os browsers não suportam o HTTP/2 sem HTTPS, o que acaba fazendo com que o seu uso seja exclusivo em modo criptografado.

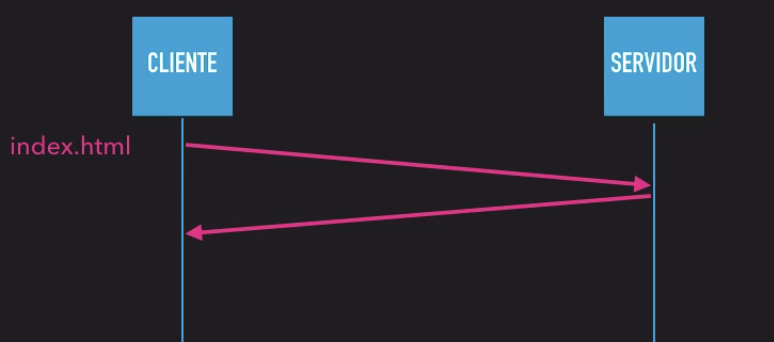
Headers Stateful.

Diminui o trafego de dados enviados no header da conexão

Quando estamos utilizando Headers Stateful, simplesmente colocamos nas requisições os cabeçalhos que se alteraram entre uma e outra, trazendo uma enorme economia de dados, visto que toda requisição HTTP possui um cabeçalho e que, muitas vezes, no HTTP/1.1, cabeçalhos repetidos eram trafegados em todas as requisições.

Server Push

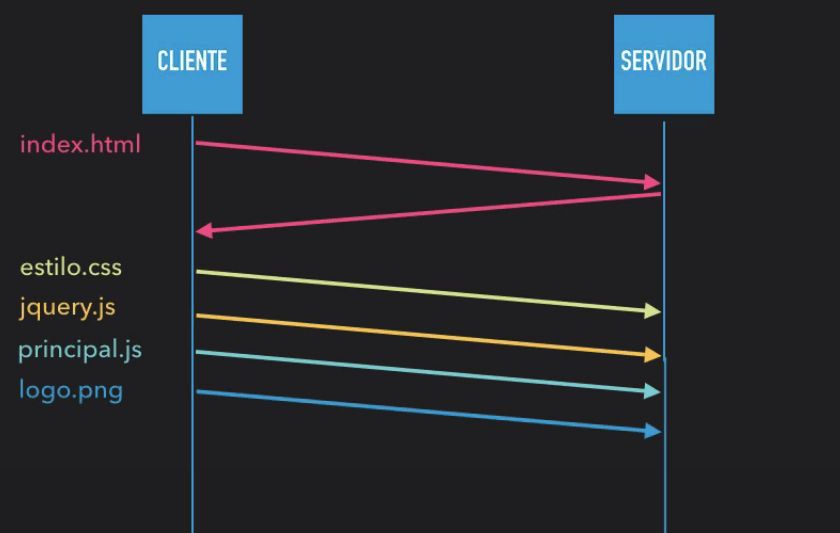
Temos o cliente e um servidor sendo representados. Podemos imaginar que estamos fazendo uma requisição para uma página principal, a index.html. Essa requisição bate no servidor e o servidor nos traz o conteúdo HTML.





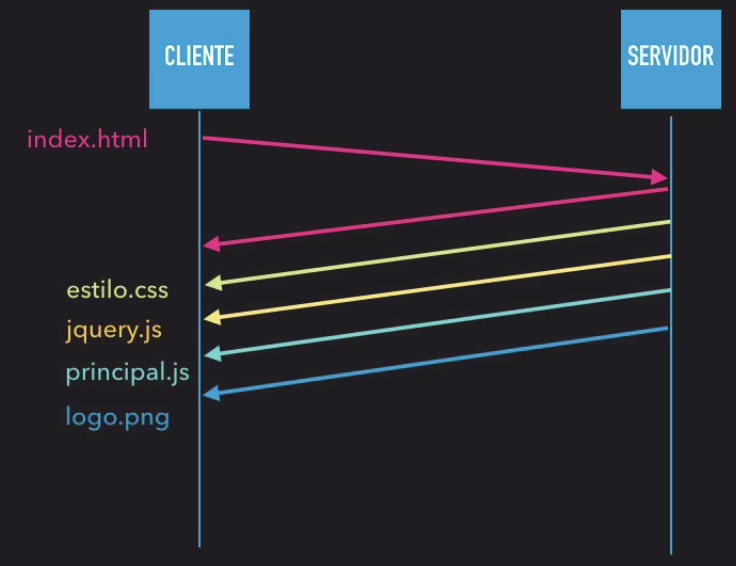
O HTML retornado pode ter o título Caelum, e então vai aparecer no nosso navegador essa informação. Além disso, temos um arquivo CSS, de estilização da página, que é o estilo.css, e dois arquivos JavaScript necessários para a página ser executada, o jquery.js e o principal.js. Além disso, no meio do corpo do HTML, tem um recurso que é de imagem, temos a imagem logo.png. Mas além desses, podemos ter vários outros recursos na nossa página.

Então, ao receber esse conteúdo, o browser tem que sair fazendo requisições de tudo o que é necessário para que ele renderize a página. O navegador interpreta esse conteúdo HTML de cima pra baixo, verifica que o primeiro recurso necessário é o estilo.css, aí ele vai lá buscar. O segundo recurso necessário, jquery.js, que é uma biblioteca JavaScript. E além disso, precisamos do principal.js e do logo.png:



Todos esses recursos especificados no HTML são novas requisições que o browser precisa fazer, e nosso cliente precisa executar. O servidor vai recebendo essas requisições, mas o cliente fica ali esperando até que essas respostas cheguem e o nosso browser consiga de fato renderizar o conteúdo para o usuário. Então há uma espera até essas respostas chegarem de fato, pois o servidor devolve as respostas das requisições na mesma sequência que foram geradas.

A partir do HTTP2, isso ficou um pouco diferente. Agora temos uma conversa mais paralela. Anteriormente estávamos trabalhando com conceitos de requisições seriais, fazíamos uma requisição e esperávamos receber, fazíamos outra requisição e esperávamos receber e por aí vai. No HTTP2, quando o cliente realiza uma requisição para \*index.html, o servidor devolve a página, mas ele já pode passar para o browser as informações necessárias para que essa página possa ser, de fato, exibida. Ou seja, ele consegue dar um passo além:



Isso é uma outra abordagem que surgiu no HTTP2, muito mais interessante. Mas quando o browser for interpretar essa página HTML, vai ter que passar pelo conteúdo que especifica o arquivo CSS? Sim, mas quando ele passar pelo estilo.css, vai verificar que já recebeu. Ou seja, ele percebe que já recebeu essas informações.

Este é o conceito de Server Push, ou seja, o server envia dados para o cliente sem que o cliente tenha solicitado, tornando o tráfego de dados muito mais otimizado.

Outra coisa importante de requisição é que temos o conceito de request e response. Cada requisição e cada resposta no HTTP1.1 são únicos.

“Por baixo dos panos”, antes dessa requisição de fato ser feita, há uma conexão, comunicação entre cliente e servidor, que chamamos de TCP. Para que consigamos realizar uma requisição via HTTP, antes existe um modelo de TCP, que é um protocolo de transporte. Isso é mais a nível de infraestrutura, pois quando trabalhamos com desenvolvimento, acabamos deixando isso pra lá, já que ficamos na camada acima dessa conexão.

Queremos mostrar é que quando fazemos uma requisição, ela é única. No HTTP, cada requisição deveria abrir uma conexão TCP, executar e fechar.

Mas isso seria muito ruim porque conexão TCP é recurso caro, é um recurso que demora a ser alocado. Claro que é muito rápido a nível computacional, mas é mais um passo antes da requisição HTTP prosseguir e recebermos uma resposta.

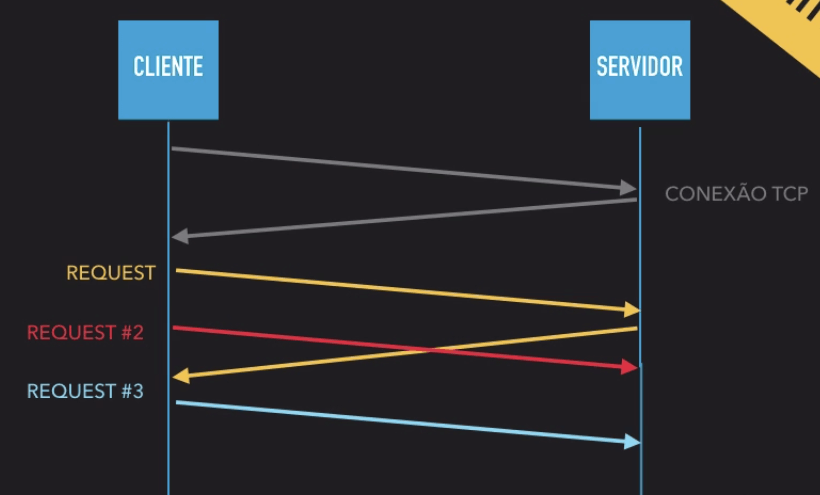
Então o que acontece, no HTTP1 existe um mecanismo chamado de Keep-Alive. O Keep-Alive determina quanto tempo, por exemplo, a nossa conexão pode ficar ativa. Ou seja, não encerra essa conexão TCP. Portanto, conseguimos realizar várias requisições com a mesma conexão TCP.

Hoje, na maioria dos browsers, temos um número entre 4 e 8 de conexões simultâneas por domínio. Significa que se fizermos uma requisição para a página da Caelum e a página da Caelum tiver mil recursos, o browser tem 4 a 8 conexões TCP ativas para conseguir realizar essas requisições em paralelo, e não serial. Mas isso na versão 1.1.

## **Keep-Alive no HTTP2**

O Keep-Alive continua existindo no HTTP2, só que ele trouxe uma novidade. Por exemplo, se temos uma conexão TCP aberta e realizamos uma requisição, poderíamos já dar prosseguimento às próximas requisições, isso em paralelo, sem de fato ficar esperando o resultado dela, de maneira assíncrona, e vamos recebendo essas respostas à medida em que o servidor for conseguindo processar.

Na imagem abaixo, fizemos a requisição 1 e requisição 2, quando íamos fazer requisição 3, já recebemos uma resposta:



Então, essas requisições e respostas vão chegando a todo tempo. É totalmente paralelo. A mesma coisa acontece com o servidor, não precisamos esperar uma resposta para enviar outra. Se já está pronta para ser enviada, ele já envia diretamente.

Esse conceito que surgiu no HTTP2 é chamado de Multiplexing e traz uma performance bastante relevante para o nosso HTTP.

Neste capítulo, o que aprendemos? Vimos que o HTTP2 atua sobre o que já conhecemos do HTTP. Ou seja, ele não muda nada em relação ao que já conhecemos de HTTP. E que todo o seu conteúdo é usado no HTTP2 de forma bastante simples.

Hoje, o que o HTTP2 especifica é mais a nível de servidor. E acaba que nós desenvolvedores não atuamos tanto nesse nível. Fica mais na outra ponta, que é quem vai produzir servidores e tudo mais, seguir esse novo protocolo.

Vimos que HTTP2 é nada mais que o HTTP com algumas melhorias, até porque o HTTP1 estava bastante desatualizado em relação ao que o mercado já vinha sofrendo.

Também vimos que os headers são binários e eles são comprimidos com algoritmos chamados de HPACK.

Vimos ainda, que o HTTP2 habilita o GZIP como padrão na resposta, logo, esses dados vêm zipado. Coisa que tínhamos que configurar manualmente na versão anterior, ou seja, HTTP1.1.

Além disso, no HTTP2, as requisições e respostas podem ser paralelas. Não precisamos ficar esperando que uma requisição termine pra fazer a próxima. Temos uma otimização maior.

Outro assunto foi que os cabeçalhos guardam status. Quando enviamos uma requisição, a próxima, para o mesmo domínio, não precisa enviar os mesmo dados que já foram trafegados na última. Conclui-se que no HTTP2 isso é evitado, ou seja, menos informação enviada, menos dados que enviamos, menos banda que usamos do usuário, mais feliz ele fica.

Além de Headers Stateful, vimos também que o HTTP2 especifica o famoso Server-push, que é o ato do servidor enviar dados sem que o browser tenha pedido, que foi o que aconteceu lá no index.html. O HTTP2 pode enviar dados diretamente para o browser sem ficar esperando uma requisição. Assim, ele dá um passo além.