**HTTP: Entendendo a web por baixo dos panos**

***H***yper***t***ext ***T***ransfer ***P***rotocol

No mundo de TI, temos muitas siglas e abreviações! O que menos importa é decorar esses nomes, mas é preciso entender o que há por trás. Nesse treinamento vamos focar nos principais conceitos do protocolo HTTP, aquilo que realmente importa para o desenvolvedor.

O cliente inicia a comunicação e o servidor responde. No entanto, qual é o papel do **HTTP** entre o cliente e o servidor?

Estabelecer regras de comunicação

Exatamente, o HTTP foi feito para estabelecer regras de comunicação entre o modelo Cliente-Servidor que funciona na Web.

O HTTP não é o único protocolo de comunicação que existe. Aliás, existem milhares de protocolos no mundo de TI, no entanto o HTTP é de longe o mais popular.

Na lista abaixo, há um item que **não** representa um **protocolo para internet**.

FTP

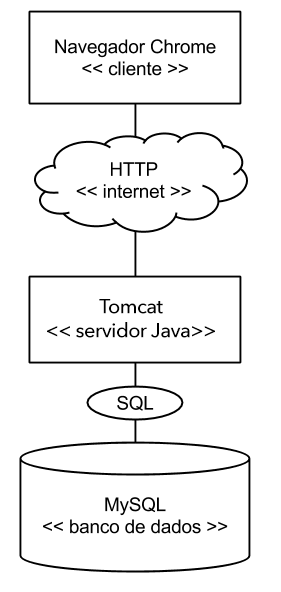
***F****ile****T****ransport****P****rotocol*, protocolo para transferir arquivos

BitTorrent

Além de ser um protocolo, também é um aplicativo para troca de arquivos na internet.

SMTP

***S****imple****M****ail****T****ransfer****P****rotocol*, protocolo para enviar e-mails.



**O que você aprendeu nesse capítulo?**

* A arquitetura Cliente-Servidor
* Um protocolo é um conjunto de regras
* HTTP é um protocolo que define as regras de comunicação entre cliente e servidor na internet.
* HTTP é o protocolo mais importante da Internet

Quando usamos HTTP, qualquer servidor no meio pode espionar os dados enviados, algo totalmente inseguro! Imagine se essas informações fossem relativas a contas bancárias. Não seria nada seguro!

Para estes outros cenários, existe o HTTP**S**, 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

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.

Dito isso, como tudo funciona? Os navegadores em posse da chave pública criptografam as informações e as enviam para o servidor que as descriptografa com a chave privada. É importante notar que apenas a chave privada descriptografa as informações criptografadas com a pública, e também que deve-se manter a chave privada segura.

**O que aprendemos nesse capítulo?**

* Só com **HTTPS a web é segura**.
* HTTPS significa usar um **certificado digital** no servidor.
* O certificado prova a **identidade** e tem **validade**
* O certificado possui uma **chave publica**.
* A chave é utilizada pelo navegador para criptografar os dados.

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.

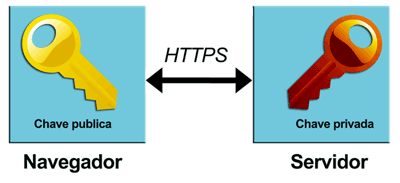
A chave privada fica apenas no lado do servidor.

Exato, a chave privada é utilizada para descriptografar os dados que foram criptografados com a chave pública, por isso ela é importante e deve ficar apenas em posse do servidor.

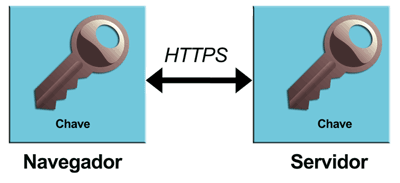
Essa garantia é feita através de uma **assinatura digital**. A autoridade certificadora (CA) **assina digitalmente** o certificado! Como na vida real, existem também no mundo digital: assinaturas!

Uma autoridade certificadora (CA - Certificate Authority) é um órgão que garante ao navegador e ao usuário que a identidade de um servidor (por exemplo o servidor da Alura) é realmente válida. Portanto, podemos trocar informações com este sem riscos!

Aprendemos no vídeo que 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 :)

**O que você aprendeu nesse capítulo?**

* Por padrão, os dados são trafegados como texto puro na web.
* Apenas com HTTPS a Web é segura
* O protocolo HTTPS nada mais é do que o protocolo HTTP mais uma camada adicional de segurança, a TLS/SSL
* O tipo de criptografia de chave pública/chave privada
* O que são os certificados digitais
* Certificados possuem identidade e validade
* As chaves públicas estão no certificado, a chave privada fica apenas no servidor
* O que é uma autoridade certificadora
* O navegador utiliza a chave pública para criptografar os dados

## DNS

A página principal do Google deve ser exibida. IP's são mais importantes para quem trabalha com rede. O desenvolvedor normalmente não precisa mexer com isso.

Observação: Esse IP pode mudar dependendo do servidor concreto onde o Google foi instalado.

Mas a gente não acessa a Google ou a Alura por um número como 52.206.164.173 e sim pela URL.

Ainda bem, não é verdade? Seria inviável decorar todos os serviços e sites que acessamos diariamente apenas por números.

Acontece que por baixo dos panos quando realizamos uma requisição essa URL é transformada em um número por um serviço transparente chamado de **DNS** (***D***omain ***N***ame ***S***ystem).

Esse serviço age como um grande banco de dados de domínios. Portanto quando fazemos uma requisição para alura.com.br o DNS age transformando para um IP e a requisição prossegue.

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/>

O domínio é o nome do site na Web. Ele facilita a navegação do usuário, que não precisa lembrar o IP de cada site.

Alternativa correta, o domínio é o nome do site na web e serve para facilitar a navegação do usuário, que acaba não precisando lembrar o IP de cada site.

O DNS tem como função realizar a tradução do nome de um domínio para o endereço de IP correspondente.

O **DNS** realiza a tradução do nome de um domínio para o endereço de IP. Existem vários servidores DNS no mundo e é fundamental para a nossa web o funcionamento deles.

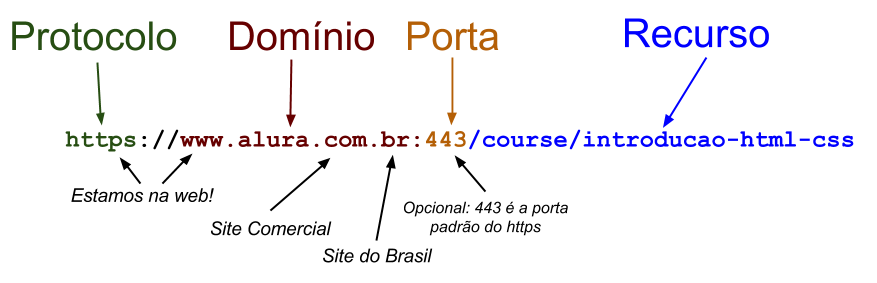
## Abrindo portas

A porta reservada para o protocolo HTTP é o 80

O protocolo HTTPS possui uma porta padrão, a porta 443

**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!



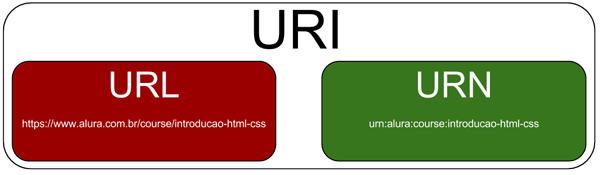
URI / URL

Muitas vezes, desenvolvedores usam a sigla **URI** (***U****niform* ***R****esource* ***I****dentifier*) quando falam de endereços na web. Alguns preferem **URL** (***U****niform* ***R****esource* ***L****ocator*), 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**.

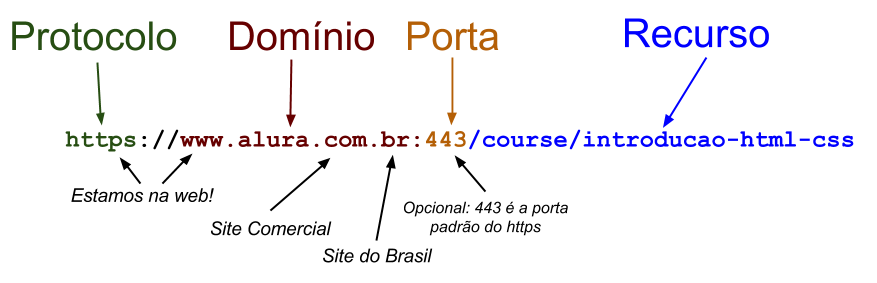
Humm ... ficou claro? Não? Vamos dar um exemplo! Existe um outro padrão que se chama **URN** (***U****niform* ***R****esource* ***N****ame*). Agora adivinha, os **URN's** também são **URI's**! Um **URN** segue também uma sintaxe bem definida, algo assim **urn:cursos:alura:course:introducao-html-css**. 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*.

## O que aprendemos nesse capítulo?

* **URL** são os endereços da Web
* Uma URL começa com o protocolo (por exemplo https://) seguido pelo **domínio** (www.alura.com.br)
* Depois do domínio pode vir a porta, se não for definida é utilizada a porta padrão desse protocolo
* Após o domínio:porta, é especificado o **caminho** para um **recurso** (/course/introducao-html-css)
* Um **recurso** é algo concreto na aplicação que queremos acessar



* 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.

O que é uma sessão HTTP?

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.

# O que é um cookie?

Vimos no vídeo o uso de um cookie para gravar um número, aquele Session ID. Mas o que é um cookie? Pesquise!

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**

# Login e Senha

Quando estamos autenticados em algum sistema, como a Alura, é necessário sempre enviar o e-mail e senha a cada requisição?

Quando enviamos uma requisição HTTP, todos os dados para que ela seja respondida devem ser enviados. Mas e o e-mail e a senha? Quando o login é feito, a Alura tem certeza de que um usuário existe e gera uma identificação quase aleatória pra esse usuário, lembra? E esse número fica salvo em um arquivo especial, chamado **cookie**, que é gerado e enviado por cada site :)

**O que você aprendeu nesse capítulo?**

* O protocolo HTTP segue o modelo **Requisição-Resposta**
* Sempre o **cliente** inicia a comunicação
* Uma **requisição** precisa ter **todas as informações** para o servidor gerar a resposta
* HTTP é **stateless**, não mantém informações entre requisições
* As plataformas de desenvolvimento usam **sessões** para guardar informações entre requisições

O método HTTP é **GET** e o código da resposta é **200**.

Lembrando que o método define a ação ou intenção da requisição HTTP (**GET** é igual a receber). O código da resposta dá uma dica ao cliente se a requisição foi um sucesso ou não, e qual foi o problema em caso de falha. O código **200** significa que tudo deu certo!

Quando passamos os parâmetros da requisição na URL, estamos fazendo uso do método **GET**. O que é super útil quando precisamos repetir a requisição com os mesmos parâmetros :)

Utilizando o método **GET**, tanto o login quanto a senha seriam passados como parâmetro na URL, coisa que não queremos que aconteça. O método **POST** deixa os parâmetros no corpo da requisição, assim evita que informações importantes, como a senha, fiquem explícitas na URL.

Usando o método GET, a URL ficaria:

GET /signin/?email=nico.steppat@caelum.com.br&senha=totalmentesecreta HTTP/1.1

HOST: https://www.alura.com.brCOPIAR CÓDIGO

Logo, o **POST** foi utilizado para que se enviasse os valores do formulário no corpo da requisição.

**Serviços Web -REST**

As aplicações que disponibilizam serviços para outras são chamadas de *webservices*. E uma *API* de utilização é documentada para uma integração eficiente entre sistemas.

Temos serviços web para trabalhar com pagamentos(*Paypal* é um exemplo famoso), upload de imagens, transformação de CEP em endereços textuais e diversos outros. Tudo isso é feito através do poderoso protocolo **HTTP**.

Mas como especificar à aplicação de serviço que gostaríamos de receber em um formato JSON? Via cabeçalho **HTTP**!

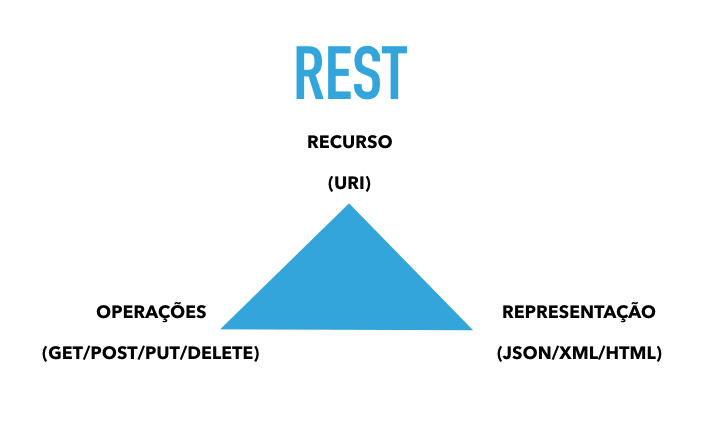
Para indicar que queremos resposta no formato JSON usa-se um **Accept: application/json** como cabeçalho HTTP. Por outro lado já na resposta uma indicação desse conteúdo é especificado pelo cabeçalho **Content-Type: application/json**.

**O que é REST?**

**O padrão REST**

**Logo podemos perceber que o padrão usado pela equipe do webservice define que uma requisição web tem três tipos de componentes importantes: recursos (URI), operações (GET, POST, PUT, DELETE/...) e representação de dados(XML, JSON, ...).**

**Esses três componentes em conjuntos seguindo algumas práticas são a base para o modelo arquitetural REST(Representational State Transfer) ou em português Transferência de Estado Representacional.**

****

As semânticas principais são:

* GET - recupera informações sobre o recurso identificado pela URI. Ex: listar restaurante, visualizar o restaurante 1. Uma requisição GET não deve modificar nenhum recurso do seu sistema, ou seja, não deve ter nenhum efeito colateral, você apenas recupera informações do sistema.
* POST - adiciona informações usando o recurso da URI passada. Ex: adicionar um restaurante. Pode adicionar informações a um recurso ou criar um novo recurso.
* PUT - adiciona (ou modifica) um recurso na URI passada. Ex: atualizar um restaurante.
* DELETE - remove o recurso representado pela URI passada. Ex: remover um restaurante.

Representação

Quando fazemos uma aplicação não trafegamos um recurso pela rede, apenas uma representação dele. E essa representação pode ser feita de diferentes formas como JSON, XML ou HTML.

Conclusão

Nossas URIs devem representar recursos, as operações no recurso devem ser indicadas pelos métodos HTTP e podemos falar qual é o formato em que conversamos com o servidor com o Content-Type e Accept que são cabeçalhos do HTTP.

* REST é um padrão arquitetural para comunicações entre aplicações
* Ele aproveita a estrutura do HTTP
* Recursos são definidos via URI
* Operações com métodos HTTP(GET/POST/PUT/DELETE)
* Cabeçalhos(Accept/Content-Type) são usados para especificar as representações(JSON,XML,...)

**HTTP2 - Dados binários, GZIP ativo e TLS**

**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.

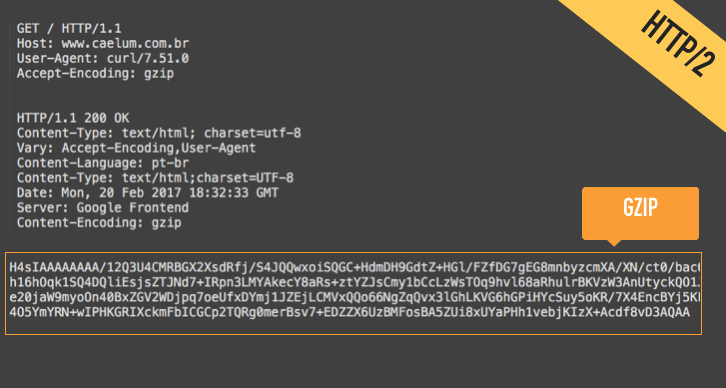
Com a chegada do mundo mobile novas preocupações apareceram e otimizações são cada vez mais necessárias para uma boa performance. Por isso uma mudança foi necessária e em 2015 depois de alguns anos de especificações e reuniões surgiu a versão 2 desse protocolo.

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.

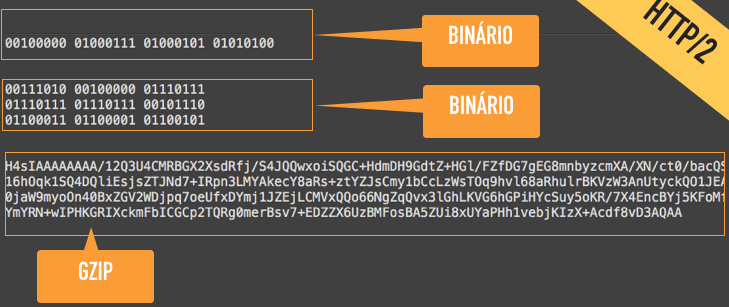
No HTTP 1.1, para melhorar a performance, habilitamos o **GZIP** no servidor para comprimir os dados das respostas. É uma excelente prática, mas que precisa ser habilitada explicitamente. No HTTP/2, o **GZIP é padrão e obrigatório**.

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

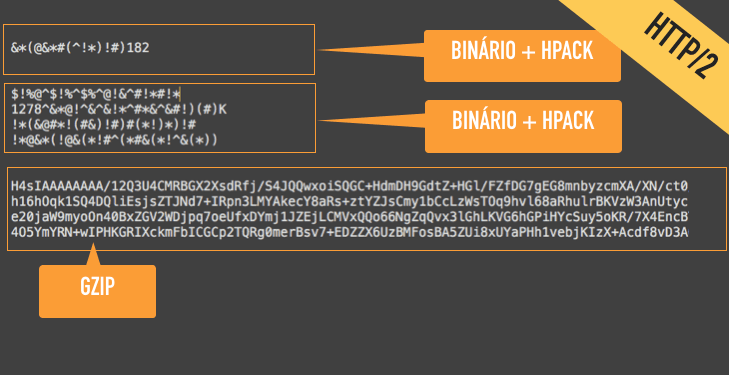


Mas, se você já olhou como funciona uma requisição HTTP, vai notar que só GZIPar as respostas resolve só metade do problema. Tanto o request quanto o response levam vários cabeçalhos (headers) que não são comprimidos no HTTP 1.1 e ainda viajam em texto puro.

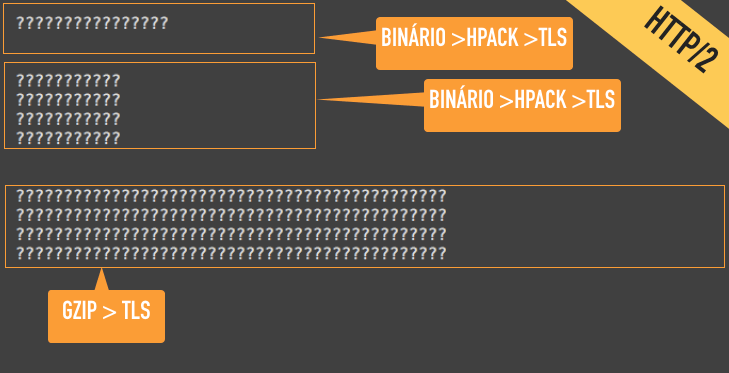
Já na nova versão, os headers passam a ser binários:



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.



## Cabeçalhos diferentes

Agora, se temos uma imagem, os cabeçalhos podem mudar, por exemplo, o Host, que pode estar especificado na página principal. Logo, na primeira requisição, o conteúdo HTML especificou que tem que buscar uma imagem do Host, que é **image.caelum.com.br**, um subdomínio dentro da nossa aplicação. Então, esse cabeçalho terá que ser alterado, logo enviaremos apenas os cabeçalhos que são diferentes.

Isso está especificado no HTTP2, para que uma requisição fique mais leve e não onere tanto o usuário. Isso é conhecido como **Headers Stateful**.

No início do curso, comentamos que o HTTP era stateless, ou seja, ele não guarda informações das requisições passadas. E isso continua valendo, mas no caso dos cabeçalhos, existe um ambiente que guarda estado.

Como a tecnologia de ***Headers Stateful*** pode nos ajudar a economizar dados?

Parte superior do formulário

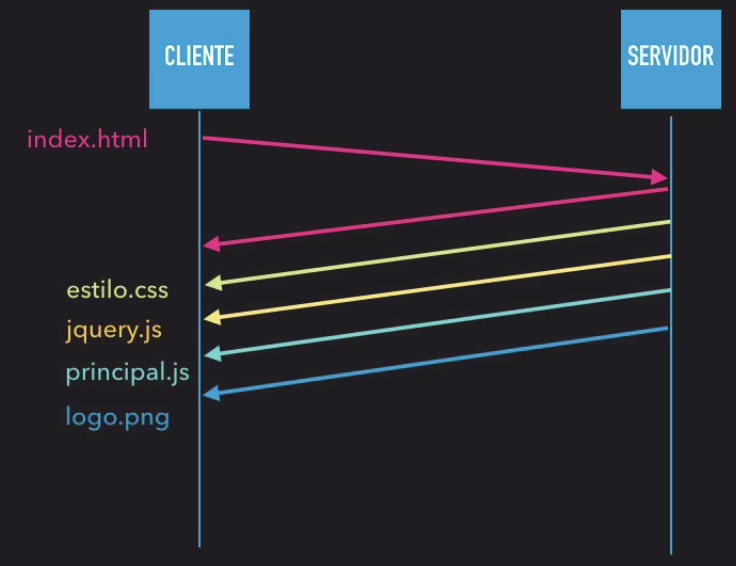
Como trafegamos apenas os headers que mudam de uma requisição para outra, acabamos por economizar uma boa quantidade de dados, pois não precisamos enviar headers que mudam poucas vezes a todo momento, como o Accept.

Correto, os Headers Stateful permitem que apenas os cabeçalhos que mudem sejam enviados a cada requisição, economizando muita banda que seriam cabeçalhos repetidos.

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.Parte inferior do formulário

**HTTP2 - Server Push**

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

O servidor pode empurrar para o clientes certos recursos antes mesmo de serem solicitados, pois ele consegue analisar o HTML e ver o que mais é preciso para carregar a página fazendo com que não seja necessário gastar tempo pedindo todos os outros recursos.

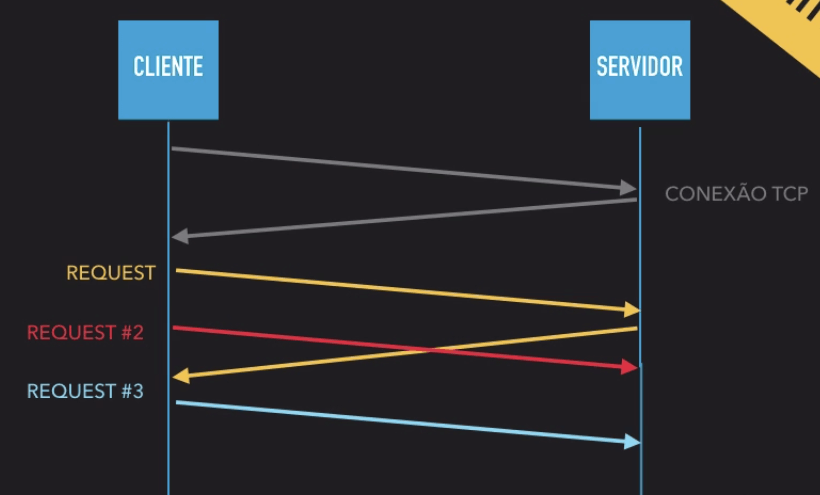
Correto, o servidor pode empurrar certas respostas para o cliente antes mesmo delas serem requisitadas.

**HTTP2 – Multiplexação**

**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.