**\*\*\* PROTOCOLO HTTP**

HTTP é a sigla de HyperText Transfer Protocol.

Os protocolos são definidos, especificados e disponibilizados para implementação em ambas as partes, para consultar a especificação do HTTP, você pode utilizar o seguinte endereço: https://tools.ietf.org/html/rfc2616

O protocolo HTTP segue o modelo Client-Server. O que o navegador (como Chrome ou Firefox) representa nesse modelo? O cliente ou o servidor?

Responta: Exato, nós que estamos utilizando o navegador somos o cliente da Alura, que nos fornece o conteúdo, logo ela é o servidor.

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

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

Você já usou torrent para baixar algum arquivo na internet? Caso sim, aproveitou um outro modelo de comunicação, o P2P ou Peer-To-Peer!

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!

Usando algum aplicativo de Torrent, o protocolo utilizado não é o HTTP, e sim o protocolo P2P, como BitTorrent ou Gnutella.

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.

- FTP: File Transport Protocol, protocolo para transferir arquivos

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

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

- O protocolo **smb** realmente existe e é a abreviação de ***S****erver****M****essage****B****lock*. Ele é utilizado para compartilhar arquivos dentro de uma rede local.

\*\*\* PROTOCOLO HTTPS

É muito provável que existam outros servidores intermediários no provedor antes que os dados realmente cheguem no servidor da Alura. Com a resposta é a mesma coisa, ela volta passando por esses servidores no meio antes de chegar até nosso navegador. O problema é, 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 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

O que acontece com nossos dados quando usamos HTTP , ou seja sem a letra S ao final?

Resposta: Exato, nossos dados são enviados em texto puro, ficando visível para qualquer um que consiga interceptar nossa conexão!

\*\*\*\* CERTIFICADO DIGITAL

Reparem que no navegador, ao lado do https, aparece um cadeado e que ao clicarmos no cadeado podemos ver mais informações sobre HTTPS. Uma dessas informações indica que a Alura tem uma identidade confirmada. O que isso quer dizer?

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.

Sobre as características do HTTPS, selecione todas as opções abaixo que estejam corretas:

Respostas:

\* 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.

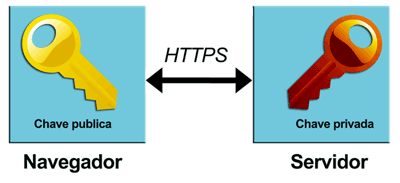
\* O certificado prova a identidade e tem validade => Correto, todo certificado tem uma data validade e serve para provar a identidade entre o cliente e o servidor.

\* O certificado guarda a chave pública => Perfeito, é no certificado digital que encontramos a chave pública utilizada para criptografar os nossos dados.

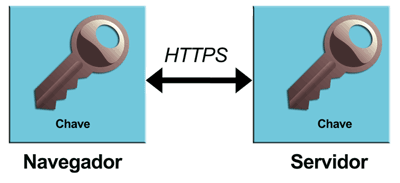
Qual é a finalidade das autoridades certificadoras?

Resposta: Garantir que podemos confiar naquele certificado (identidade) => Exato, a principal função de uma entidade certificadora é garantir que os certificados que estão sendo utilizados podem ser confiados.

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 :)

\*\*\* Endereços IP's

podemos acessar o Google usando um IP. Para quem é muito curioso, mas muito curioso mesmo e deseja saber qual é o IP do Google, pode digitar na linha de comando a seguinte instrução:

no cmd: nslookup google.com

Resultado:

Name: google.com

Address: 216.58.202.238

Esse comando procura o número IP do Google na internet. Podemos até usar esse endereço no navegador: [http://216.58.202.238](http://216.58.202.238/)

\*\*\* DNS

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

Acontece que por baixo dos panos quando realizamos uma requisição essa URL é transformada em um número por um serviço transparente chamado de DNS (Domain Name System).

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/>

Falamos bastante sobre o domínio nessa aula, mas o que é um domínio (ou domain name) e qual a sua importância?

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

Qual é o objetivo ou a função do **DNS** (***Domain Name System*** ou servidor de domínios)?

Resposta: 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. 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

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

Vários protocolos definem a sua porta padrão como por exemplo o FTP que usa 21 ou SSH que usa 22.

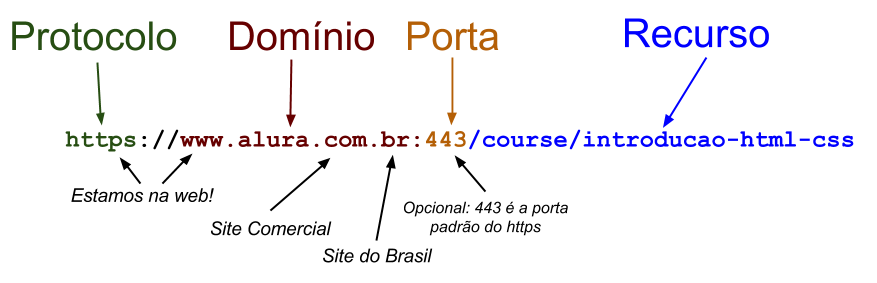
\*\*\* **Recursos**

Navegando dentro da Alura, mais informações aparecem depois do nome e do domínio. Por exemplo, para acessar a página principal dos cursos, usamos **https://cursos.alura.com.br/dashboard**. O */dashboard* é um recurso (***resource***) do site que gostaríamos de acessar. Existem vários outros recursos na Alura como as carreiras (/careers), ou o fórum de discussões (/forum). O importante é que cada *recurso possua o seu nome único*.

Navegando um pouco mais na Alura, podemos perceber que entre o domínio e o recurso podem vir outras informações . Por exemplo, para acessar o curso **HTML5 e CSS3 I: Suas primeiras páginas da Web**, usamos <https://cursos.alura.com.br/course/introducao-html-css>. Ou seja, para acessarmos o recurso /introducao-html-css, usamos um caminho intermediário, o /course. Há vários outros exemplos na Alura que usam caminhos para chegar ao recurso concreto, como por exemplo /courses/mine, e navegando na Alura você encontrará mais.

**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!



\*\*\* **Para saber mais: URI ou 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?

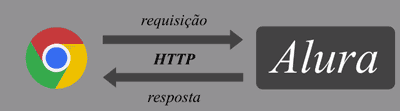
**Resposta 1 (fácil):** Uma **URL** é uma **URI**. No contexto do desenvolvimento web, ambas as siglas são válidas para falar de endereços na web. As siglas são praticamente sinônimos e são utilizadas dessa forma.

\*\*\* **Modelo Requisição e Resposta**

Realizaremos um teste **efetuando login** na Alura. Quando preenchemos o formulário e clicamos no botão, o navegador envia o nosso *login* e a nossa *senha* para 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!



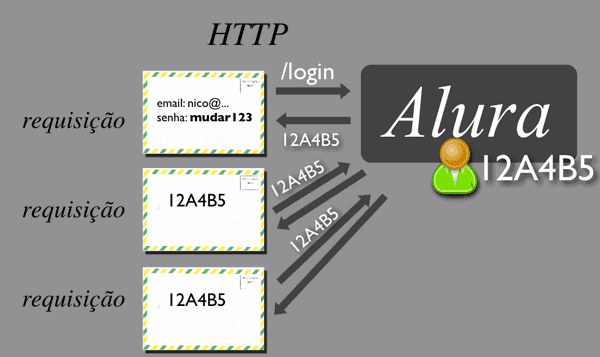
\*\*\* **Comunicação sem estado**

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.

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**

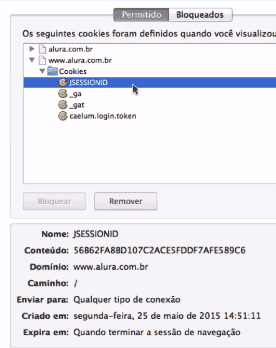
Quando efetuamos o login, a Alura valida os nossos dados, certo? Nesse momento, o servidor tem certeza que o usuário existe e gera uma identificação quase aleatória para o usuário. Essa identificação é um número criado ao vivo e muito difícil de adivinhar. Esse numero é a identificação temporária do 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 **SESSÕES** com 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).

**O HTTP e o estado das requisições**

Uma requisição sempre deve ser enviada com todas as informações necessárias, o que faz uma requisição ser sempre independente das demais.

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?**

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***

**Comunicação em HTTP**

Uma comunicação com HTTP sempre é iniciada pelo cliente que manda uma requisição ao servidor esperando por uma resposta.

\*\*\* **Depurando o método HTTP**

**Método GET do HTTP**

Vamos abrir o console de desenvolvedor e acessar o [http://www.alura.com.br](http://www.alura.com.br/). Aqui usaremos o navegador Chrome, mas nos outros navegadores o comportamento é bem parecido.

No console podemos ver todas as requisições HTTP executadas pelo Chrome. Mas não só isso, também aparecem alguns códigos e métodos, além do tempo de execução para cada requisição. Repare que chamamos apenas o http://www.alura.com.br, mas foram feitas várias outras requisições em seguida.



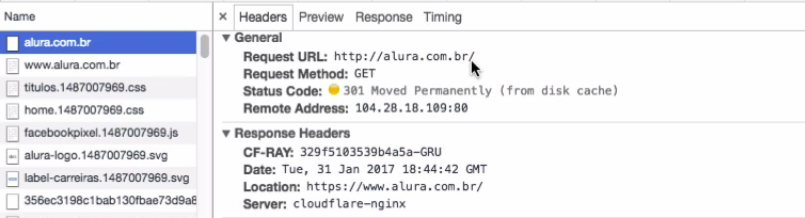
Na primeira coluna aparece a URL (o endereço) e na segunda coluna o método HTTP. O método HTTP indica qual é a **intenção ou ação dessa requisição**. Enviamos uma requisição com o método **GET**. Queremos receber informações, sem modificar algo no servidor, que é justamente a ideia do método *GET*.

**Primeiro código da resposta**

Como resposta recebemos o código de status 301. O protocolo HTTP define alguns códigos padrões para esclarecer a resposta. Indo com o mouse em cima do 301 o Chrome mostra o significado desse código: **Moved Permanently**. Ou seja, o site Alura foi movido para outro lugar! Eis a questão: Onde então está o site Alura?

A localização ou a URL concreta está na resposta HTTP. Vamos clicar em cima do código de status 301 para receber mais informações. Aqui o Chrome mostra todos os cabeçalhos da requisição e da resposta. São muitos (nem tantos) mas o que nos interessa é a nova localização do site. Dentro do item *Response Headers* podemos ver todos os cabeçalhos que o servidor devolveu e logo logo apareceu um com o nome **Location**. Esse cabeçalho indica a nova URL, só que agora usando **https**.

Quando o navegador recebe o status 301 ele já sabe que é preciso enviar uma nova requisição e procura a nova URL no cabeçalho de resposta Location.



**Redirecionando entre sites**

Se alguém acessa a Alura usando http (lembrando, *inseguro*) automaticamente é chamado o site seguro (https). Isto é um comportamento muito comum para garantir que usamos https sempre. Se esquecermos de usar https, o servidor devolve o status 301 com a nova localização, mas agora usando https.

O navegador, ao receber 301, chama automaticamente a nova URL. No mundo de desenvolvimento web este comportamento é chamado de *Redirecionamento pelo navegador*, ou **Redirecionamento no lado do cliente**. Fomos redirecionados para o recurso correto. A tarefa do desenvolvedor é definir o código de resposta e, no caso em que algum recurso tenha mudado a URL, o código 301 será usado com o cabeçalho Location.

**O código 200**

Continuando no console, a segunda requisição foi para https://www.alura.com.br, novamente usando o método HTTP GET com a intenção de receber dados. Agora o código de resposta foi 200. Este é um dos códigos mais comuns e significa que tudo deu certo! Dessa vez não foi preciso fazer um redirecionamento (não tem o cabeçalho Location na resposta) e não deu nenhum outro problema. A requisição foi aceita e processada corretamente - código 200. Perfeito.

**Tipos de dados diferentes**

No console podemos ver que aparecem mais requisições (cada linha representa um novo request). Quando o servidor Alura devolve a resposta para o navegador vem o conteúdo da página inicial em um formato especial, chamado de HTML. O HTML define a estrutura da nossa página, define os menus, botões, links, rodapé etc. Mas dentro do HTML não vêm as imagens e outros arquivos necessários para deixar o site perfeito. Dentro dele vem apenas a URL (endereço) desses outros recursos.

Então, ao receber o HTML, o navegador dispara várias outras requisições para carregar as imagens, fontes e outros dados. Como também são requisições HTTP, o console mostra suas informações. Podemos ver que na resposta vem o tipo do conteúdo, por exemplo text/html, text/css, image/svg+xml, entre outros.

O importante é saber que o protocolo HTTP não está preso em algum formato específico. Podemos trafegar qualquer informação com ele, seja texto ou binário!

\*\*\* **Depurando os códigos de resposta HTTP**

Vamos fazer um teste e acessar um recurso que não existe, por exemplo: https://www.alura.com.br/nao-existe. Bom, a Alura mostra uma imagem que indica que não achou o que procuramos, mas vamos dar uma olhada no console. Repare que o código agora é 404. No mundo HTTP 404 significa que o servidor não encontrou o recurso (*Not Found*).



Durante o desenvolvimento de uma aplicação web podem acontecer problemas no lado do servidor. Isto é normal pois alguma lógica pode falhar, erros acontecem no desenvolvimento! A notícia boa é que o protocolo HTTP vem preparado para isso. Quando algum problema no servidor acontecer, podemos avisar o cliente através do protocolo HTTP. O código mais comum para este tipo de problema é o 500 que significa: "deu pau no servidor :)". Talvez um ou outro aluno já tenha visto um erro 500 na Alura. Isso não deveria acontecer, mas onde há humanos também há problemas, não é mesmo?

**Categorias de códigos**

Existem muitos códigos de resposta definidos no protocolo HTTP. Há tabelas disponíveis na web que mostram esses códigos, descrevendo o significado de cada um deles. No entanto, no dia a dia, o desenvolvedor não precisa decorar todos esses códigos. Basta consultar quando for necessário, por exemplo [aqui](http://www.w3schools.com/tags/ref_httpmessages.asp).

O importante é saber que algo que começa com **2xx** significa coisa boa, a requisição foi executada com sucesso. Quando recebemos algo como **3xx** normalmente significa que o navegador precisa fazer algo a mais (o cliente precisa agir) pois algo mudou ou um recurso não existe mais. **4xx** significa que o navegador enviou dados errados, como por exemplo uma URL errada. Caso o servidor gere algum problema, serão utilizados os códigos **5xx**.

No dia a dia os códigos 200, 404 e 500 são de longe os mais utilizados!

\*\*\* **Parâmetros na requisição com métodos GET e POST**

Vamos ver um outro exemplo de uma URL e acessar [http://www.youtube.com](http://www.youtube.com/). Como tem muito conteúdo no YouTube, vamos pesquisar, por exemplo por *Ayrton Senna*. Repare que, ao pesquisar no YouTube, a URL mudou um pouco. O recurso acessado pela busca se chama **/results** (os resultados da pesquisa) mas agora temos um **parâmetro da requisição**, indicado pela **?**: **https://www.youtube.com/results?search\_query=Ayrton+Senna**

O parâmetro se chama **search\_query** com o valor **Ayrton+Senna**. Esses parâmetros da URL normalmente são chamados de ***Query Params***. O HTTP permite enviar mais de um parâmetro, basta concatenar o próximo parâmetro através do caractere **&**.

Por exemplo, a busca avançada do Google usa vários parâmetros para refinar a pesquisa como o idioma, o país ou data. Veja como o Google concatena os *Query Params*: https://www.google.com.br/?gws\_rd=ssl#lr=lang\_pt&tbs=lr:lang\_1pt&q=neymar

**Parâmetros com GET**

Reparem que no console sempre aparece o tipo (ou método) da requisição, sendo GET. Uma característica da requisição GET é enviar os parâmetros pela URL! Isso é útil quando queremos deixar os parâmetros visíveis. Assim podemos facilmente guardar a URL com os parâmetros para repetir a requisição algum momento depois. Mas será que isso também é uma boa opção na hora de enviar credenciais como login e senha? Queremos que apareça a senha de um usuário na URL?

Imagina que você efetue o login no seu banco e na URL apareça: https://www.bb.com.br/login?login=nico&password=supersecreto

Não parece ser uma boa solução, não é mesmo?

**O método HTTP POST**

Vamos efetuar um login na Alura para ver como esse sistema envia dados do usuário para o servidor.



Repare que a URL para enviar o *login* e *senha* se chama https://www.alura.com.br/signin. Repare também que o método HTTP utilizado mudou. Estamos usando o **HTTP POST**! Usando o POST, o navegador envia os dados do formulário no corpo da requisição e não na URL! Se fosse um GET, todos os dados seriam enviados através da URL. Como a Alura não deseja que os dados do *login* e *senha* apareçam na URL do navegador, foi utilizado um HTTP POST. Faz sentido?

**GET para receber, POST para criar?**

Um outro exemplo de um método POST na Alura é quando criamos uma pergunta no forum. Nesse momento o usuário submete um formulário com dados para criar um novo recurso na Alura (a pergunta do aluno se torna um recurso). O método POST foi inicialmente pensado para criar algo novo no servidor como acabamos de fazer. Ou seja, ao enviar uma requisição POST para o servidor, a nossa intenção é criar algo novo. No entanto, nem sempre isso é realmente utilizado dessa maneira. Por exemplo, acabamos de usar um POST para verificar o login, ou seja, não alteramos ou adicionamos nada na Alura. Nossa motivação para o POST era esconder os parâmetros apenas.

Como o servidor realmente reage quando recebe uma requisição POST depende da implementação, depende da lógica atrás. Os métodos como GET e POST definem uma intenção mas o que realmente será executado depende do servidor.

No dia a dia, vocês vão ver códigos usando GET para fazer pesquisas mas também para alterar ou remover algo no servidor. A mesma coisa podemos dizer sobre POST. Vocês vão usar o POST para inserir e alterar dados, mas também para pesquisar. As aplicações podem adaptar o significado dos métodos HTTP quando for necessário.

\*\*\* **Serviços Web -REST**

- ##Serviços WEB : 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**.

- ##Cenário de trabalho: Você muito provavelmente já teve uma péssima experiência quando estava sem conexão com a internet usando um aplicativo móvel. Alguns apps não funcionam sem um acesso a rede porque as principais funcionalidades são feitas via requisições **HTTP**.

Essas requisições são implementadas programaticamente pelo desenvolvedor. Podemos implementá-las em várias linguagens de programação: Java, PHP, Javascript etc.

Como retorno dessa requisição poderíamos receber o seguinte conteúdo HTML;

Logo percebemos que os responsáveis precisarão realizar uma análise do conteúdo HTML e extrair dele somente as informações necessárias. Esse ato de analisar o documento é chamado de realizar um parsing do arquivo.

Perceba que o HTML tem muito mais do que o necessário para essa equipe, é um formato de verbosidade considerável: temos cabeçalhos e tags diversas. Para piorar estamos trafegando muito mais informações do que o necessário e onerando até mesmo a banda do nosso usuário. Péssimo cenário não é mesmo?

Pensando nessa deficiência do HTML temos outros formatos que fazem mais sentido quando uma representação de um recurso (um restaurante) se faz necessário. Temos como exemplo mais legível o **XML(eXtensible Markup Language)**;

Outro famoso formato e onerando menos ainda a rede, por ser mais leve, é o **JSON(JavaScript Object Notation);**

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**.

No desenvolvimento Web, para requisição e envio de dados através de formulários, quais são os métodos HTTP que são mais utilizados pelos desenvolvedores no dia a dia?

Resposta: Os métodos GET e POST são de longe os métodos mais utilizados no desenvolvimento web, mas porque isso?

A resposta está no próprio HTML. Por padrão, as páginas HTML fazem apenas requisições do tipo GET ou POST. Para o tipo POST especificamente, precisamos fazer uso da tag form, configurando o atributo method como POST, ao invés do seu valor padrão que é GET. O outro caso é quando usamos a tag a que cria um link de uma página para outra, fazendo com o que o navegador execute uma requisição do tipo GET.

Para fazer requisições usando métodos como PUT ou DELETE, precisamos fazer com código JavaScript.

Marque todas as afirmações verdadeiras:

- Requisições HTTP podem ser enviadas através de qualquer aplicativo/software que entenda o protocolo (Afirmação correta , o HTTP não depende do navegador. Aliás, o tempo todo o nosso celular usa o HTTP para enviar requisições através de aplicativos!).

- HTTP é totalmente independente do formato dos dados(Afirmação correta pois HTTP não depende de um formato especifico!).

- HTTP pode trafegar tanto dados binários quanto dados textuais(Afirmação correta pois HTTP pode trafegar dados binários como imagens e dados textuais como HTML ou CSS!).

O protocolo **HTTP** envia dados em texto puro e já estudamos sobre as implicações disso em questões de segurança. Vimos inclusive como o **HTTPS** lida com isso.

Em alguns momentos se faz necessário avisar na própria requisição um formato específico esperado para a resposta.

De que forma podemos especificar o formato que esperamos que seja devolvido?

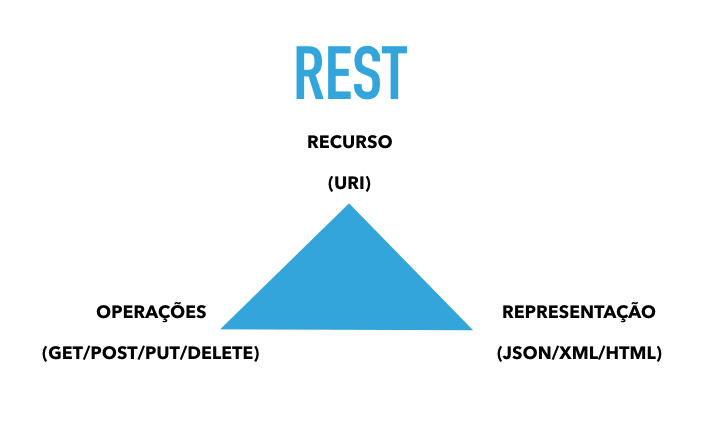
Resposta: Pelo cabeçalho **Accept**.

- Através dele podemos usar algum formato específico como: Accept: text/html, Accept: text/css, Accept: application/xml, Accept: application/json, Accept:image/jpeg e Accept: image/\*

- Ou até mesmo não definir nenhum formato específico colocando: Accept: \*/\*

\*\*\* **O que é REST?**

* **O padrão REST :** 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.



* **Recurso**

Ao criar as URIs do nosso sistema devemos levar em conta que elas representam recursos, não ações.

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

Para representar a adição de um restaurante podemos usar a URI /restaurante com o método HTTP POST, que representa que estamos adicionando alguma informação no sistema.

* **Operações**

O protocolo HTTP possui operações através de métodos como: GET, POST, PUT e DELETE.

Cada método tem uma semântica diferente e juntando o método à URI deveríamos conseguir representar todas as ações do nosso sistema.

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.

**\*\*\* PROTOCOLO HTTP**

**\*\*\* PROTOCOLO HTTP2**

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

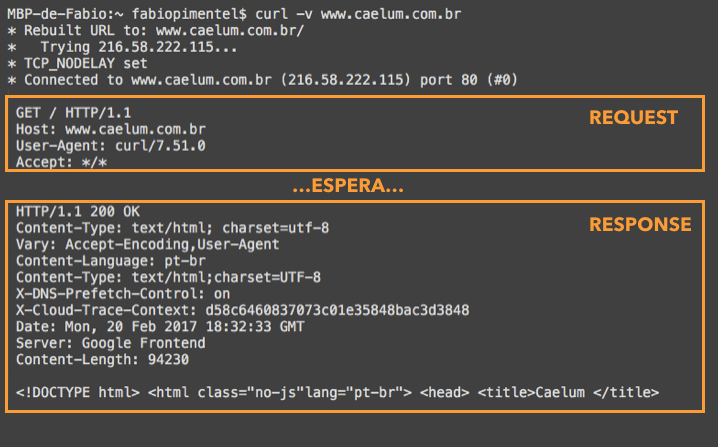
< Server: Google Frontend

< 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.

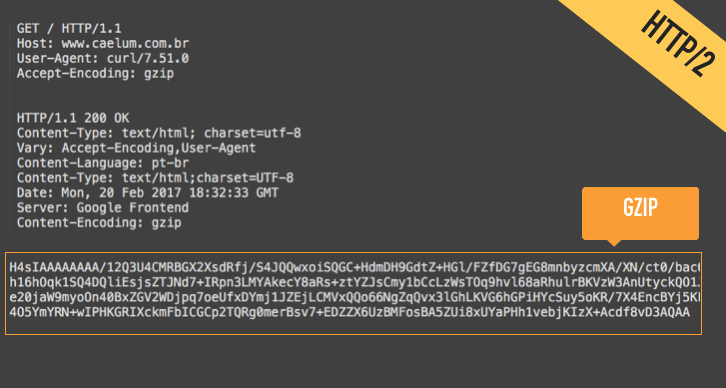
Com a chegada do mundo mobile novas preocupações apareceram e otimizações são cada vez mais necessárias para uma boa performance. Por isso uma mudança foi necessária e em 2015 depois de alguns anos de especificações e reuniões surgiu a versão 2 desse protocolo.

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.

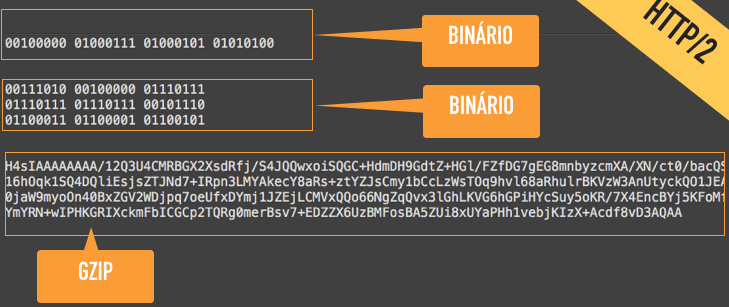
No HTTP 1.1, para melhorar a performance, habilitamos o **GZIP** no servidor para comprimir os dados das respostas. É uma excelente prática, mas que precisa ser habilitada explicitamente. No HTTP/2, o **GZIP é padrão e obrigatório**.

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

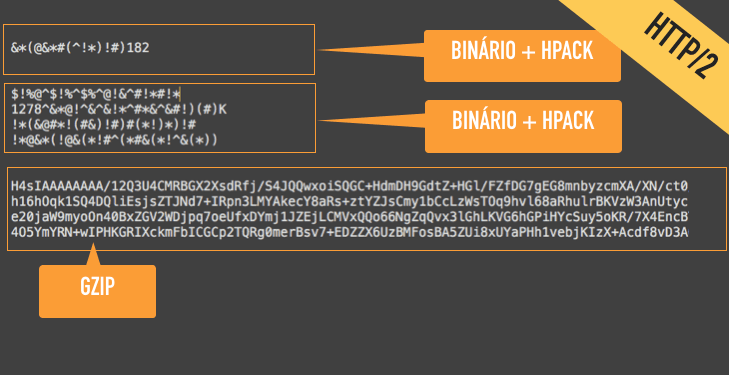


Mas, se você já olhou como funciona uma requisição HTTP, vai notar que só GZIPar as respostas resolve só metade do problema. Tanto o request quanto o response levam vários cabeçalhos (headers) que não são comprimidos no HTTP 1.1 e ainda viajam em texto puro.

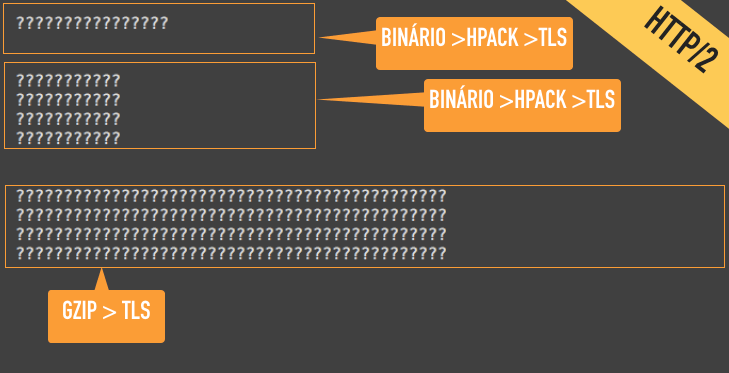
Já na nova versão, os headers passam a ser binários:



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.



Dado o que vimos neste capítulo, aponte quais dos motivos abaixo foram importantes quando decidiram criar uma nova versão do protocolo HTTP, que já estava tão concretizado e estabelecido na Web.

Resposta:

- Com o crescimento do número de dispositivos móveis conectados a Web, é cada vez mais importante que a quantidade de dados trafegada seja a menor possível, afinal este tipo de dispositivo não costuma ter uma conexão com muita banda larga. O protocolo HTTP/2 traz diversas tecnologias para diminuir o tamanho das requisições.

- Por padrão, no protocolo HTTP versão 1.1 não é necessário o uso da camada de segurança TTL/SSL. Como hoje em dia trafegamos muitos dados críticos na Web, como senhas, logins e dados bancários, um protocolo atualizado que faz uso dessa segurança parece quase uma necessidade.

Apesar do protocolo HTTP/1.1 ter sido de extrema importância para a Web ao longo de vários anos, como toda boa tecnologia, é necessário um *update*. A nova versão do HTTP veio para adequar este protocolo tão famoso a um mundo onde temos muito mais dados sendo trafegados na rede, e a velocidade de acesso e segurança do usuário se tornam bastante importantes.

Para que serve a tecnologia HPACK implementada no protocolo HTTP/2 ?

Resposta: Exato, a tecnologia HPACK é especialista em comprimir os Headers da requisições/respostas HTTP, deixando as mais leves.

Selecione as afirmativas verdadeiras sobre as versões 1.1 e 2.0 do protocolo HTTP:

Resposta:

- No HTTP/1.1 o Gzip não é nativo do protocolo, no HTTP/2 ele já vem por padrão.

- Correto, 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.

- No HTTP/2 os dados são trafegados em binário, no HTTP/1.1 eles são trafegados como texto.

\*\*\* **HTTP2 - Cabeçalhos Stateful**

Agora, queremos representar uma requisição. No código abaixo, estamos fazendo uma requisição através do método GET, que já conhecemos. Essa requisição está sendo feita para a raiz, bem parecido com o que fizemos no CURL no vídeo anterior.

GET /

Host: www.caelum.com.br

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:34.0)

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3

Accept-Encoding: gzip, deflate

Quando realizamos uma requisição para essa URL via Firefox, por exemplo, o navegador envia alguns cabeçalhos que são padrões. Por exemplo, no cabeçalho Host, temos o domínio para onde estamos realizando essa requisição, que é www.caelum.com.br. Como estamos realizando um GET para /, o path para onde estamos realizando a requisição é www.caelum.com.br/.

Além disso, vemos uma informação de User-Agent, que é basicamente a fonte dessa requisição, neste caso é o navegador, Mozilla. Quando realizarmos uma requisição pelo CURL, aparecerá CURL, e se for num Safari, Chrome, qualquer outro navegador, irá aparecer a informação de onde estamos realizando de fato a requisição. Ou seja, nesse cabeçalho a gente especifica quem é o usuário.

Nele também é dito que é aceito, por padrão, o HTML, na linguagem tanto português quanto inglês, e que estamos aceitando uma codificação GZIP, já que no HTTP1 podemos especificar que tipo de compressão nossa requisição está aceitando.

**Precisamos repetir os cabeçalhos enviados em uma requisição anterior?**

Agora vamos realizar uma outra requisição, mas dessa vez para o arquivo **principal.js**. Então, quando a requisição para página principal foi feita, provavelmente recebemos um HTML, e desse HTML foi necessário realizar uma outra requisição, porque era um recurso importante para a página ser exibida, como por exemplo um arquivo JavaScript.

GET /principal.js

Host: www.caelum.com.br

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:34.0)

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3

Accept-Encoding: gzip, deflate

Então, na próxima requisição teremos que enviar novos parâmetros, novos dados. Então, mais uma vez, é uma nova requisição, agora para /principal.js, um recurso do nosso sistema. Qual o Host? **www.caelum.com.br**. Qual o Agent? Mozilla, versão 5.0. O que é aceito? qual linguagem é aceita? que tipo de compressão estamos aceitando?

Podemos perceber que o que colocamos nessa segunda requisição é exatamente o mesmo que fizemos na primeira? Os mesmos dados estão sendo trafegados nas duas requisições. Seria ótimo se só trafegássemos isso uma única vez, pois se estamos enviando mais dados, oneramos ainda mais nosso usuário, usando mais banda, deixando essa requisição mais lenta.

Seria muito bom se só pudéssemos ou só tivéssemos que enviar uma única vez, e é exatamente isso que o HTTP2 faz. A partir do HTTP2, não precisamos mais repetir os **Headers**, os cabeçalhos que já enviamos em uma requisição anterior. Logo, quando fazemos uma requisição para o **principal.js**, onde teríamos os cabeçalhos exatamente iguais aos da requisição passada, nós não precisamos enviar novamente esses dados.

**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.

\*\*\* **HTTP2 - 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.

\*\*\* **HTTP2 – Multiplexação**

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.