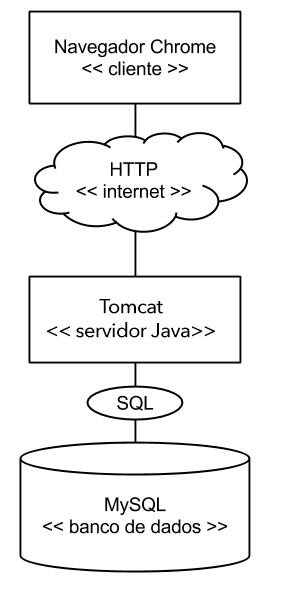
**CURSO DE HTTP**

## **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**
* **HTTP: HyperText Transfer Protocol**
* **SSL/TLS: Secure Sockets Layer / Transport Layer Security (Uma camada a mais de segurança) “ HTTPS ”**

**Certificado digital**

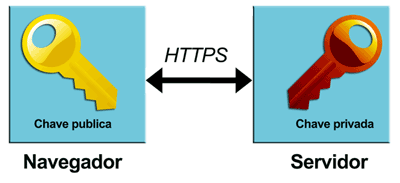
**Quando precisamos informar nossos dados a algum servidor, queremos ter certeza que este servidor realmente representa a entidade em questão. Queremos confiar em quem estamos fornecendo nossos dados!**

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

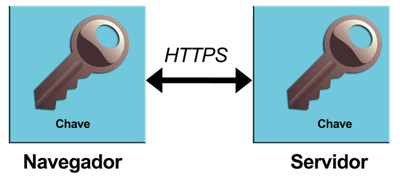
**Além disso, o certificado ainda guarda a chave pública que é utilizada para criptografar (cifrar) os dados que são trafegados entre cliente e servidor.**

**Para Saber Mais: As chaves do HTTPS**

**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.**

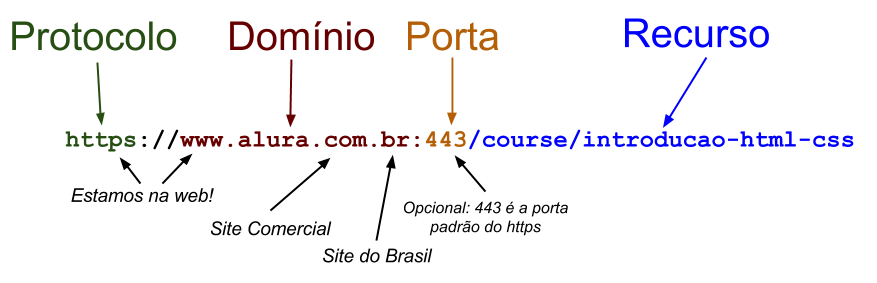
****

**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?**

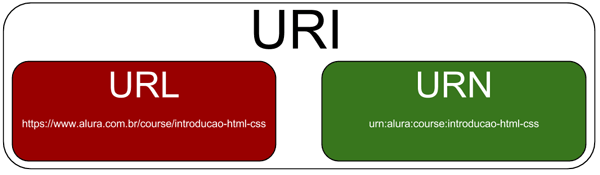
* **Por padrão, os dados são trafegados como texto puro na web.**
* **Apenas com HTTPS a Web é segura**
* **O protocolo HTTPS nada mais é do que o protocolo HTTP mais uma camada adicional de segurança, a TLS/SSL**
* **O tipo de criptografia de chave pública/chave privada**
* **O que são os certificados 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**

**Muitas vezes, desenvolvedores usam a sigla URI (*Uniform* *Resource* *Identifier*) quando falam de endereços na web. Alguns preferem URL (*Uniform* *Resource* *Locator*), 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'sque identificam um recurso sem definir o endereço, nem o protocolo. Em outras palavras, uma URLrepresenta uma *identificação* de um recurso (URI) através do endereço, mas nem todas as identificações são URL's.**

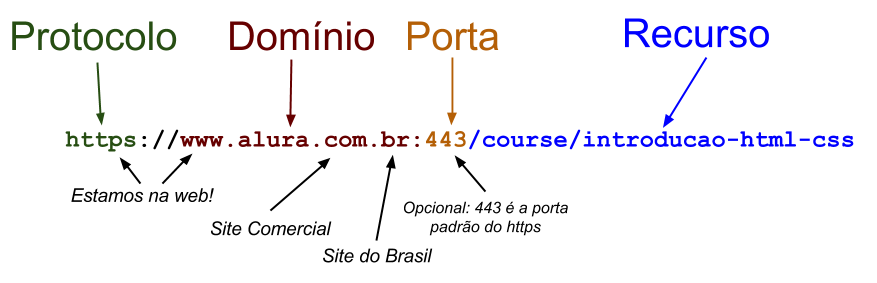
**Humm ... ficou claro? Não? Vamos dar um exemplo! Existe um outro padrão que se chama URN(*Uniform* *Resource* *Name*). Agora adivinha, os URN's também são URI's! Um URN segue também uma sintaxe bem definida, algo assim urn:cursos:alura:course:introducao-html-css. Repare que criamos uma outra identificação 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*.**

## **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**

****

**Pronto para continuar?**

**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***

**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**

# **Método GET do HTTP**

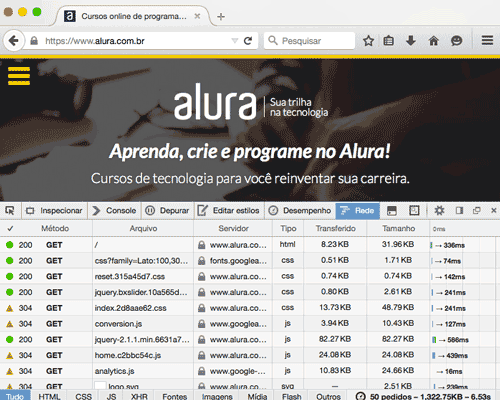
**Vamos abrir o console de desenvolvedor e acessar o** [**http://www.alura.com.br**](http://www.alura.com.br/)**. Aqui usaremos o navegador Chrome, mas nos outros navegadores o comportamento é bem parecido.**

**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.**

****

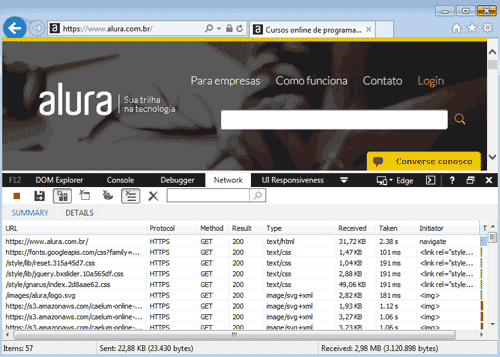
**Na primeira coluna aparece a URL (o endereço) e na segunda coluna o método HTTP. O método HTTP indica qual é a intenção ou ação dessa requisição. Enviamos uma requisição com o método GET. Queremos receber informações, sem modificar algo no servidor, que é justamente a ideia do método *GET*.**

**Para abrir o console no Mozilla Firefox basta apertar a tecla F12, ou CTRL + SHIFT + I. Ele também pode ser acessado pelo menu: Ferramentas -> Desenvolvedor web -> Exibir/Ocultar ferramentas\*\*.**

****

**No Microsoft Internet Explorer basta apertar a tecla F12 ou ir pelo menu de configuração: F12 Ferramentas do desenvolvedor.**

****

****

**No Safari basta usa o atalho COMMAND + SHIFT + C .**

**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 200significa que tudo deu certo!**

**Essa classe de códigos de status indica que a ação solicitada pelo cliente foi recebida, compreendida, aceita e processada com êxito.**

**A tabela completa de mensagens HTTP pode ser vista em:** [**https://www.w3schools.com/tags/ref\_httpmessages.asp**](https://www.w3schools.com/tags/ref_httpmessages.asp)**.**

**Usamos POST para incluir os parâmetros no corpo da requisição.**

# **Para saber mais: Parâmetros na URL**

**[PRÓXIMA ATIVIDADE](https://cursos.alura.com.br/course/http-fundamentos/task/25587/next)**

**Como, por exemplo, podemos enviar uma requisição usando o método GETpara carregarmos a página que exibe informações sobre o *usuário de número 2*? Devemos passar o parâmetro id com o valor 2. Como por exemplo:**

**http://meusite.com.br/usuario?id=2**

**Uma outra forma de fazer seria passar os valores na própria URL! Veja o exemplo:**

**http://meusite.com.br/usuario/2**

**Mas tem um probleminha, não estamos dizendo explicitamente que o valor 2 realmente representa o id. Quando um parâmetro irá receber um certo valor, devemos combinar com o servidor (com o desenvolvedor da aplicação). Neste caso, foi combinado que o parâmetro recebido seria equivalente ao id passado antes.**

**Vamos ver um exemplo prático, em um serviço que retorna informações sobre um endereço de um determinado CEP? Acesse a URL: <http://viacep.com.br/ws/20040030/json>**

**A resposta será todas as informações do CEP da Caelum Rio, como complemento, número e bairro, formatadas em JSON. Isso significa que foi combinado com o servidor, que o primeiro valor passado depois de ws deve ser o CEP e logo após, o formato em que os dados deverão chegar. No nosso caso, JSON. Tudo bem? :)**

**Experimente agora trocar para o CEP de sua casa e para outro formato de dados, como por exemplo, XML.**

**Já falamos bastante sobre os métodos (ou verbos) HTTP, GET e POST. Esses dois são utilizados na grande maioria das aplicações web, e fazem parte do dia a dia do desenvolvedor, no entanto existem diversos outros métodos.**

**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.**

**Novamente esses métodos normalmente não são utilizados quando se trata de uma aplicação web, e são mais importantes quando vem o assunto importante de Web Services.**

**Vimos o uso e as diferenças básicas entre os métodos GET e POST, que resumindo são: GET é utilizado para a busca de informações e tem seus parâmetros listados na URL, indicados pela presença da interrogação (?) seguido de pares de chave e valor, lembrando que vários parâmetros podem ser enviados simplesmente concatenando-os com o caractere &. Exemplo: https://www.youtube.com/results?search\_query=ayrton+senna&sp=cam%250**

**O POST por outro lado é mais utilizado para criação de recursos, informações no servidor e envia seus dados no corpo da requisição.**

**Para finalizar o capítulo, mencionamos que existem outros métodos HTTP como o DELETE e PUT (e acredite que tem mais ainda). O DELETE existe para enviar uma requisição com a intenção de remover um recurso, PUT para atualizar. No entanto, esses métodos são poucos utilizados no desenvolvimento de aplicações web, eles são mais importantes quando se tratam de serviços web.**

**Em geral, há mais recursos que o protocolo HTTP oferece, como vários outros cabeçalhos que especificam mais a requisição e resposta. Aqui nesse treinamento vimos os mais importantes métodos, códigos e cabeçalhos do protocolo HTTP.**

**[DISCUTIR NO FORUM](https://cursos.alura.com.br/forum/curso-http-fundamentos/exercicio-o-que-aprendemos/25712/novo)**

## **Serviços WEB**

**Hoje existem milhões de softwares rodando ou sendo desenvolvidos em várias linguagens de programação e frameworks. Tais softwares não vivem necessariamente isolados e podem querer se comunicar de alguma forma.**

**Um exemplo clássico é o login via rede social que estamos cada vez mais habituados. Essa conversa acaba sendo transparente para nós, usuários, já que exige uma autorização de acesso às nossas informações.**

**Em outros momentos as aplicações conversam sem que nada visual seja implementado ou mesmo uma autorização do cliente seja pedida.**

**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.**

**Para exemplificar vamos imaginar que temos aqui na *Alura* uma divisão responsável por uma aplicação de pedido de refeições que é a *AluraFood*.**

**A *AluraFood* tem duas equipes em ação: a do serviço web(ou simplesmente *API*web) e a dos apps mobile(*Android* e *iOS*).**

**Os desenvolvedores responsáveis pela tela de listagem de restaurantes vão precisar receber do serviço os detalhes de cada restaurante. Felizmente o pessoal responsável pelo *webservice* já documentou exatamente o que seria necessário:**

**Listagem de todos os restaurantes --> GET - http://alurafood.com/api/restaurante**

**Como retorno dessa requisição poderíamos receber o seguinte conteúdo HTML:**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<title>AluraFood</title>**

**<link rel="stylesheet" type="text/css" href="principal.css">**

**<script type="text/javascript" src="build.js"></script>**

**</head>**

**<body>**

**<table>**

**<tr>**

**<td>Bob's</td>**

**<td>8</td>**

**<td>R. do México 100</td>**

**<td><img src="http://alurafoods.com/uploads/logos/bobs.png"/></td>**

**</tr>**

**<tr>**

**<td>Subway</td>**

**<td>8.5</td>**

**<td>Av. Rio Branco 202</td>**

**<td><img src="http://alurafoods.com/uploads/logos/subway.png"/></td>**

**</tr>**

**<tr>**

**<td>Experimenta Lanches</td>**

**<td>9</td>**

**<td>R. do Brasil 545</td>**

**<td><img src="http://alurafoods.com/uploads/logos/e-lanches.png"/></td>**

**</tr>**

**</table>**

**</body>**

**</html>**

**Perceba que temos uma listagem de restaurante sendo apresentada dentro de uma tabela(elemento table do HTML) e cada linha(elemento tr) possui 4 colunas(td). Dentro de cada coluna temos as informações dos restaurantes: nome, nota de avaliação, endereço e logo.**

**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) que poderia ser devolvido como resposta e ter o seguinte conteúdo:**

**<?xml version="1.0"?>**

**<restaurantes>**

**<restaurante>**

**<nome>Bob'</nome>**

**<avaliacao>8</avaliacao>**

**<endereco>R. do México 100</endereco>**

**<logo>http://alurafoods.com/uploads/logos/bobs.png</logo>**

**</restaurante>**

**<restaurante>**

**<nome>Subway</nome>**

**<avaliacao>8.5</avaliacao>**

**<endereco>Av. Rio Branco 202</endereco>**

**<logo>http://alurafoods.com/uploads/logos/subway.png</logo>**

**</restaurante>**

**<restaurante>**

**<nome>Experimenta Lanches</nome>**

**<avaliacao>9</avaliacao>**

**<endereco>R. do Brasil 545</endereco>**

**<logo>http://alurafoods.com/uploads/logos/e-lanches.png</logo>**

**</restaurante>**

**</restaurantes>**

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

**[**

**{**

**"nome": "Bob'",**

**"avaliacao": "8",**

**"endereco": "R. do México 100",**

**"logo": "http://alurafoods.com/uploads/logos/bobs.png"**

**},**

**{**

**"nome": "Subway",**

**"avaliacao": "8.5",**

**"endereco": "Av. Rio Branco 202",**

**"logo": "http://alurafoods.com/uploads/logos/subway.png"**

**},**

**{**

**"nome": "Experimenta Lanches",**

**"avaliacao": "9",**

**"endereco": "R. do Brasil 545",**

**"logo": "http://alurafoods.com/uploads/logos/e-lanches.png"**

**}**

**]**

**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.**

## **Transcrição**

**Tudo certo para a listagem de restaurantes. Mas será que o app AluraFood se resume a listar restaurantes? Provavelmente não, dado que o usuário efetua pedidos, um restaurante tem cardápio que poderia sofrer alterações e por aí vai.**

**Algumas funcionalidades específicas aos responsáveis de um restaurante podem ser necessárias. E para isso o webservice deveria estar preparado também para lidar com essa necessidade:**

**Listagem de todos os restaurantes --> GET - /restaurante**

**Adicionar um restaurante --> POST - /restaurante**

**Perceba que no exemplo fictício as duas primeiras URIs são idênticas e a funcionalidade muda completamente a partir do método HTTP usado:**

**GET -> Listagem**

**POST -> Criação**

**A atualização poderia ter um outro endpoint como por exemplo:**

**PUT/PATCH - /restaurante/1**

**O número 1 ao final da URI indica um identificador a um restaurante específico.**

**A remoção de um restaurante poderia seguir o mesmo modelo:**

**DELETE - /restaurante/1**

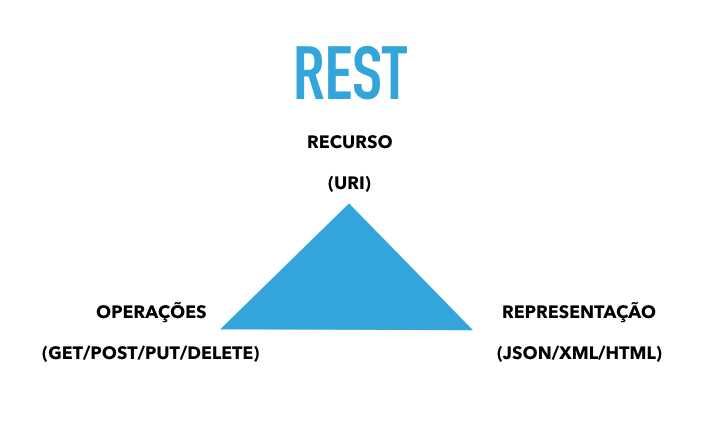
**Para a busca do cardápio de um restaurante específico o endpoint gerado poderia seguir o modelo:**

**GET - /restaurante/1/cardapio**

## **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.**

****

## **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.**

* **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 representaçEm 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**
* **video -> video/mp4**
* **application -> application/xml, application/pdf**
* **Para conhecer outros formatos aceitos você pode acessar aqui: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types>**
* **ões(JSON,XML,...)**

**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**

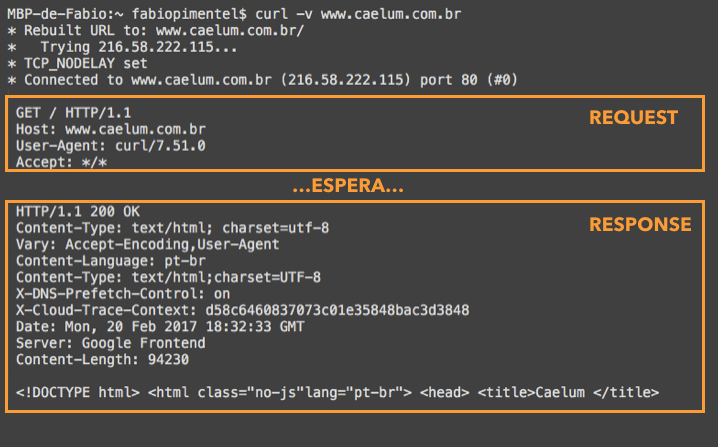
**< Server: Google Frontend**

**< 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.**

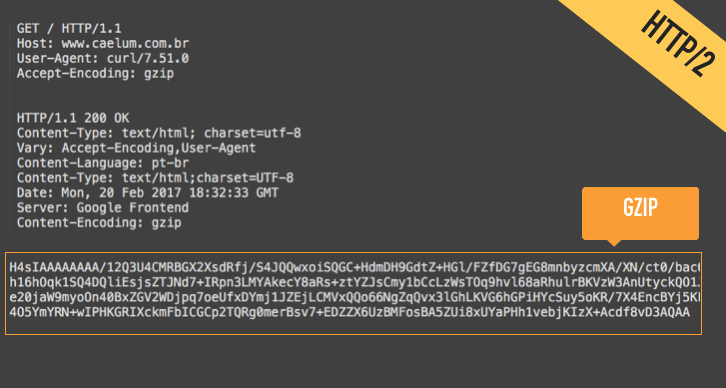
**Com a chegada do mundo mobile novas preocupações apareceram e otimizações são cada vez mais necessárias para uma boa performance. Por isso uma mudança foi necessária e em 2015 depois de alguns anos de especificações e reuniões surgiu a versão 2 desse protocolo.**

**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.**

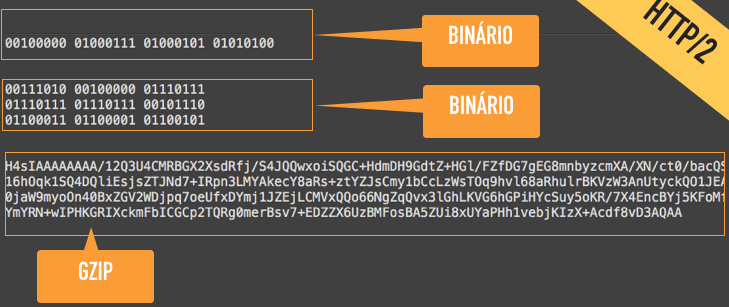
**No HTTP 1.1, para melhorar a performance, habilitamos o GZIP no servidor para comprimir os dados das respostas. É uma excelente prática, mas que precisa ser habilitada explicitamente. No HTTP/2, o GZIP é padrão e obrigatório.**

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

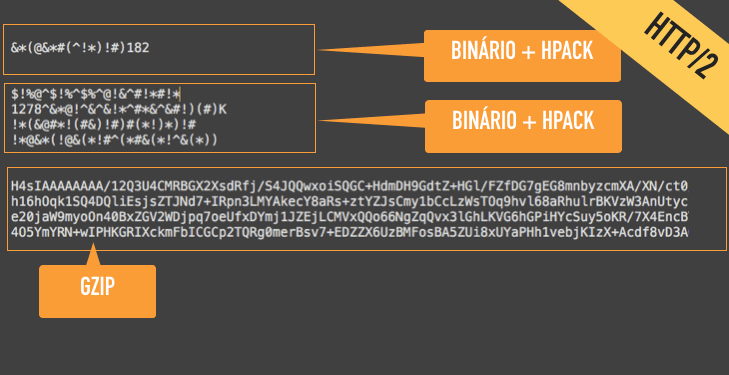
****

**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.**

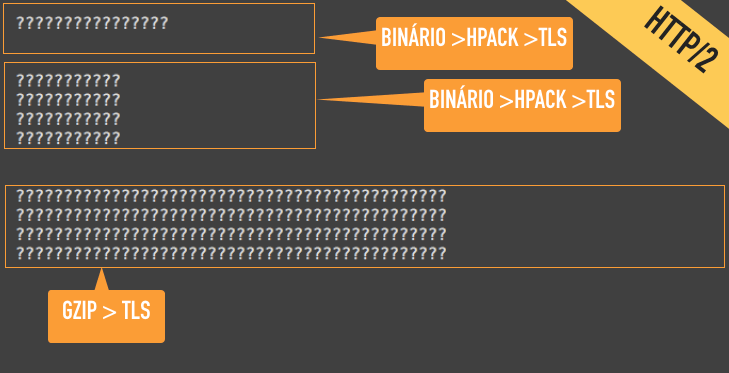
**Já na nova versão, os headers passam a ser binários:**

****

**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.**

****

**HPACK implementada no protocolo HTTP/2 ?**

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

## **Transcrição**

**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.**

**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**

**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 dáNeste 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**