# **O que é API?**

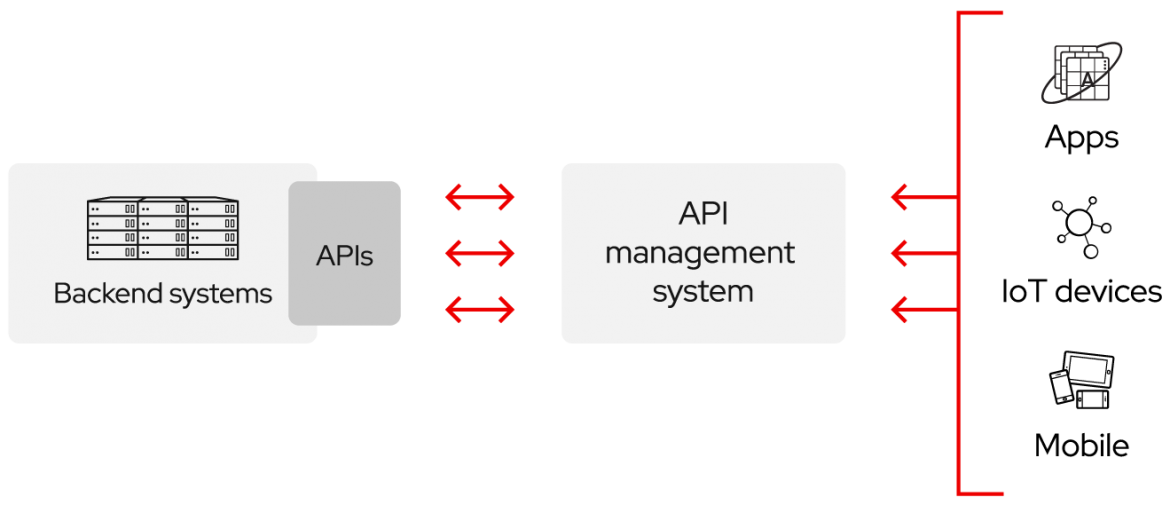
API é um conjunto de definições e protocolos usado no desenvolvimento e na [integração](https://www.redhat.com/pt-br/topics/integration) de software de aplicações. API é um acrônimo em inglês que significa interface de programação de aplicações.

Uma API permite que sua solução ou serviço se comunique com outros produtos e serviços sem precisar saber como eles foram implementados. Isso simplifica o desenvolvimento de aplicações, gerando economia de tempo e dinheiro. Ao desenvolver novas ferramentas e soluções (ou ao gerenciar aquelas já existentes), as APIs oferecem a flexibilidade necessária para simplificar o design, a administração e o uso, além de fornecer oportunidades de inovação.

As APIs costumam ser vistas como contratos, com documentações que representam um acordo entre as partes interessadas. Se uma dessas partes enviar uma solicitação remota estruturada de uma forma específica, isso determinará como o software da outra parte responderá.

As APIs simplificam a forma como os desenvolvedores integram novos componentes de aplicações a uma arquitetura preexistente. Por isso, elas ajudam na colaboração entre as empresas e as equipes de TI. Muitas vezes, as necessidades empresariais mudam rapidamente para responder aos mercados digitais em transformação. Nesse ambiente, novos concorrentes podem redefinir o setor inteiro com uma nova aplicação. Para manter a competitividade, é importante oferecer suporte à implantação e desenvolvimento rápidos de serviços inovadores. O desenvolvimento de [aplicações nativas em cloud](https://www.redhat.com/pt-br/topics/cloud-native-apps) é uma forma conhecida de aumentar a velocidade de criação. Ele depende de uma arquitetura de aplicações de [microsserviços](https://www.redhat.com/pt-br/topics/microservices) conectada por meio de uma API.

As APIs são uma maneira simplificada de conectar a própria infraestrutura por meio do desenvolvimento de aplicações nativas em nuvem. No entanto, elas também possibilitam o compartilhamento de dados com clientes e outros usuários externos. As APIs públicas agregam valor comercial porque simplificam e ampliam a forma como você se conecta aos parceiros, além de possivelmente monetizar seus dados. Um exemplo famoso é a API do [Google Maps](https://www.google.com/maps).



Por exemplo, imagine uma empresa distribuidora de livros. Essa distribuidora de livros *poderia* oferecer aos clientes uma aplicação em que os atendentes de uma livraria pudessem verificar a disponibilidade de um título diretamente com a distribuidora. Essa aplicação poderia ser cara de desenvolver, limitada pela plataforma e exigiria longos períodos de desenvolvimento e manutenção contínua.

Como alternativa, a distribuidora, poderia fornecer uma API para verificar a disponibilidade no estoque. Essa abordagem proporciona vários benefícios, incluindo:

* O acesso aos dados por meio de uma API ajuda os clientes a consolidarem informações sobre seu inventário em um único local.
* A distribuidora de livros pode fazer alterações nos sistemas internos sem causar impacto nos clientes, contanto que o comportamento da API não mude.
* Com uma API disponibilizada publicamente, os desenvolvedores que trabalham para a distribuidora de livros, os vendedores ou terceiros poderiam desenvolver uma aplicação para ajudar os clientes a encontrar os livros que procuram. Isso poderia resultar no aumento das vendas ou outras oportunidades de negócios.

Para resumir, com as APIs, você libera o acesso aos seus recursos sem abrir mão da [segurança e do controle](https://www.redhat.com/pt-br/topics/security). É você quem determina como isso será feito e quem terá acesso. [A segurança das APIs](https://www.redhat.com/pt-br/topics/security/api-security) dependem de um bom gerenciamento. É possível conectar APIs, bem como criar aplicações que fazem uso dos dados ou funcionalidades disponibilizadas por elas, usando uma plataforma de integração distribuída que ligue todos os elementos, incluindo sistemas legados e dispositivos de Internet das Coisas (IoT).

Existem três abordagens para políticas de lançamento de APIs.

#### **API Privada**

A API é usada apenas internamente. Isso oferece às empresas um maior controle.

#### **API de Parceiros**

A API é compartilhada com parceiros de negócios específicos. Isso pode fornecer fluxos de receita adicionais sem comprometer a qualidade.

#### **API Pública**

A API é disponibilizada para todos. Terceiros podem desenvolver aplicações que interajam com a sua API e isso pode se tornar uma fonte de inovação.

### **Inovação com API**

A exposição das suas APIs aos parceiros ou ao público pode:

* Criar novos canais de receita ou ampliar os existentes.
* Expandir o alcance da sua marca.
* Facilitar a inovação aberta ou aumentar a eficiência por meio da colaboração e de desenvolvimento externos.

Parece bom, não é? Mas como as APIs podem fazer tudo isso?

Vamos voltar para o exemplo da empresa distribuidora livros.

Suponhamos que um dos parceiros da empresa desenvolva uma aplicação para ajudar as pessoas a encontrar livros nas prateleiras de livrarias. Essa experiência aprimorada atrai mais consumidores para livraria (que é cliente da distribuidora) e amplia o canal de receita existente.

Talvez um terceiro use uma API pública para desenvolver uma aplicação que permita que as pessoas comprem livros diretamente da distribuidora em vez de em uma livraria. Isso abre um novo canal de receita para a distribuidora de livros.

O compartilhamento de APIs (com parceiros selecionados ou aberta a todos) pode ter efeitos positivos. Cada parceria amplia o reconhecimento da sua marca para além dos esforços de marketing da sua empresa. Abrir a tecnologia para todos, como fazem as APIs públicas, estimula os desenvolvedores a criar um ecossistema de aplicações com base na sua API. Quanto mais pessoas usam sua tecnologia, maior é a sua chance de gerar novos negócios.

Tornar a tecnologia pública pode gerar resultados novos e inesperados. Às vezes, esses resultados revolucionam setores inteiros. No caso da nossa empresa distribuidora de livros, novas organizações (um serviço de empréstimos de livros, por exemplo) podem mudar fundamentalmente a maneira de fazer negócios. Com as APIs públicas e em parceria, você pode se beneficiar dos esforços criativos de uma comunidade muito maior do que a sua equipe interna de desenvolvedores. Novas ideias podem surgir de qualquer lugar. Por disso, as empresas precisam estar cientes das mudanças no mercado e prontas para agir. As APIs podem ajudar.

### **Um breve histórico das APIs**

As APIs surgiram nos primeiros dias da computação, muito antes do computador pessoal. Naquela época, elas eram normalmente usadas como bibliotecas para sistemas operacionais. Embora a API enviasse mensagens entre mainframes em alguns momentos, ela era quase sempre local para os sistemas em que operava. Depois de quase 30 anos, as APIs se expandiram para além dos ambientes locais. No início dos anos 2000, elas estavam se tornando uma tecnologia importante para a integração remota de dados.

## **APIs remotas**

As APIs remotas foram projetadas para interagir por meio de uma rede de comunicações. Quando falamos “remota”, queremos dizer que os recursos manipulados pela API estão em algum lugar fora do computador que faz a solicitação. Como a rede de comunicações mais usada é a Internet, a maioria das APIs são projetadas com base em padrões da web. Nem todas as APIs remotas são web, mas é justo afirmar que, em geral, as APIs web são remotas.

As APIs web normalmente usam o protocolo HTTP para mensagens de solicitação e fornecem uma definição da estrutura das mensagens de resposta. Essas mensagens de resposta geralmente têm o formato de arquivo XML ou JSON. Tanto XML quanto JSON são formatos de preferência porque apresentam os dados de forma simplificada, o que facilita a manipulação por outras aplicações.

## **Quais foram as melhorias feitas nas APIs?**

Conforme as APIs se transformavam em APIs web, vários esforços foram feitos para facilitar um pouco a sua criação e torná-las mais úteis.

### **Um pouco de SOAP e um monte de REST**

Com a proliferação das APIs web, uma especificação de protocolo foi desenvolvida para ajudar a padronizar a troca de informações: o Simple Object Access Protocol, mais conhecido como SOAP. As APIs projetadas com SOAP usam o XML como formato de mensagem e recebem solicitações por HTTP ou SMTP. O SOAP facilita o compartilhamento de informações por aplicações executadas em ambientes diferentes ou escritos em linguagens diferentes.

Outra especificação é a Representational State Transfer (REST). APIs web que adotam as restrições de arquitetura da REST são chamadas de APIs RESTful. A REST é fundamentalmente diferente do SOAP: o SOAP é um protocolo e a REST é um estilo de arquitetura. Isso significa que não há um padrão oficial para APIs RESTful web. Conforme definido na dissertação de Roy Fielding [“Architectural Styles and the Design of Network-based Software Architectures”](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm), as APIs serão consideradas RESTful se estiverem em conformidade com seis restrições de arquitetura:

* **Arquitetura cliente-servidor:** a arquitetura REST é composta por clientes, servidores e recursos. Ela lida com as solicitações via HTTP.
* **Sem monitoração de estado:** nenhum conteúdo do cliente é armazenado no servidor entre as solicitações. Em vez disso, as informações sobre o estado da sessão são mantidas com o cliente.
* **Capacidade de cache:** o armazenamento em cache pode eliminar a necessidade de algumas interações entre o cliente e o servidor.
* **Sistema em camadas:** as interações entre cliente e servidor podem ser mediadas por camadas adicionais. Essas camadas podem oferecer recursos extras, como balanceamento de carga, caches compartilhados ou segurança.
* **Código sob demanda (opcional):** os servidores podem ampliar a funcionalidade de um cliente por meio da transferência de códigos executáveis.
* **Interface uniforme:** essa restrição é essencial para o design de APIs RESTful e inclui quatro vertentes:
  + **Identificação de recursos nas solicitações:** os recursos são identificados nas solicitações e separados das representações retornadas para o cliente.
  + **Manipulação de recursos por meio de representações:** os clientes recebem arquivos que representam recursos. Essas representações precisam ter informações suficientes para permitir a modificação ou exclusão.
  + **Mensagens autodescritivas:** cada mensagem retornada para um cliente contém informações suficientes para descrever como ele deve processá-las.
  + **Hipermídia como plataforma do estado das aplicações:** depois de acessar um recurso, o cliente REST pode descobrir todas as outras ações disponíveis no momento por meio de hiperlinks.

Essas restrições podem parecer excessivas, mas são muito mais simples do que um protocolo prescrito. Por isso, as APIs RESTful estão se tornando mais comuns do que as APIs SOAP.

Nos últimos anos, as especificações da [OpenAPI](https://www.openapis.org/) se tornaram o padrão na hora de definir APIs REST. A OpenAPI permite que desenvolvedores de todas as linguagens criem interfaces de API REST compreensíveis com o mínimo de suposições.

### **SOA ou arquitetura de microsserviços**

As duas abordagens de arquitetura que mais usam APIs remotas são a arquitetura orientada a serviços (SOA) e a arquitetura de [arquitetura de microsserviços](https://www.redhat.com/pt-br/topics/microservices). A SOA, a mais antiga das duas abordagens, começou como um aprimoramento sobre as aplicações monolíticas. Considerando que um único aplicativo monolítico faz tudo, algumas funções podem ser fornecidas por aplicações diferentes levemente acopladas por meio de um padrão de integração, como um barramento de serviços corporativos (ESB).

Embora a SOA seja, de forma geral, mais simples que uma arquitetura monolítica, ela traz o risco de mudanças em cascata por todo o ambiente caso as interações entre os componentes não sejam compreendidas claramente. Essa complexidade adicional traz de volta alguns dos problemas que a SOA foi criada para solucionar.

As arquiteturas de microsserviços são parecidas com os padrões SOA no que diz respeito à utilização de serviços levemente acoplados. No entanto, elas vão além no rompimento com as arquiteturas tradicionais. Na arquitetura de microsserviços, os serviços usam um framework de mensageria comum, como as APIs RESTful. As APIs RESTful são usadas na comunicação entre serviços sem transações de conversão de dados complicadas ou camadas de integração adicionais. O uso de APIs RESTful permite – e até estimula – a entrega mais rápida de novos recursos e atualizações. Cada serviço é independente. Um serviço pode ser substituído, aprimorado ou descartado sem afetar nenhum outro na arquitetura. Com essa arquitetura leve, é possível otimizar os serviços distribuídos ou de cloud e oferecer suporte à escalabilidade dinâmica para serviços individuais.

link: <https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces>

# 

# 

# 

# 

# **HTTP**

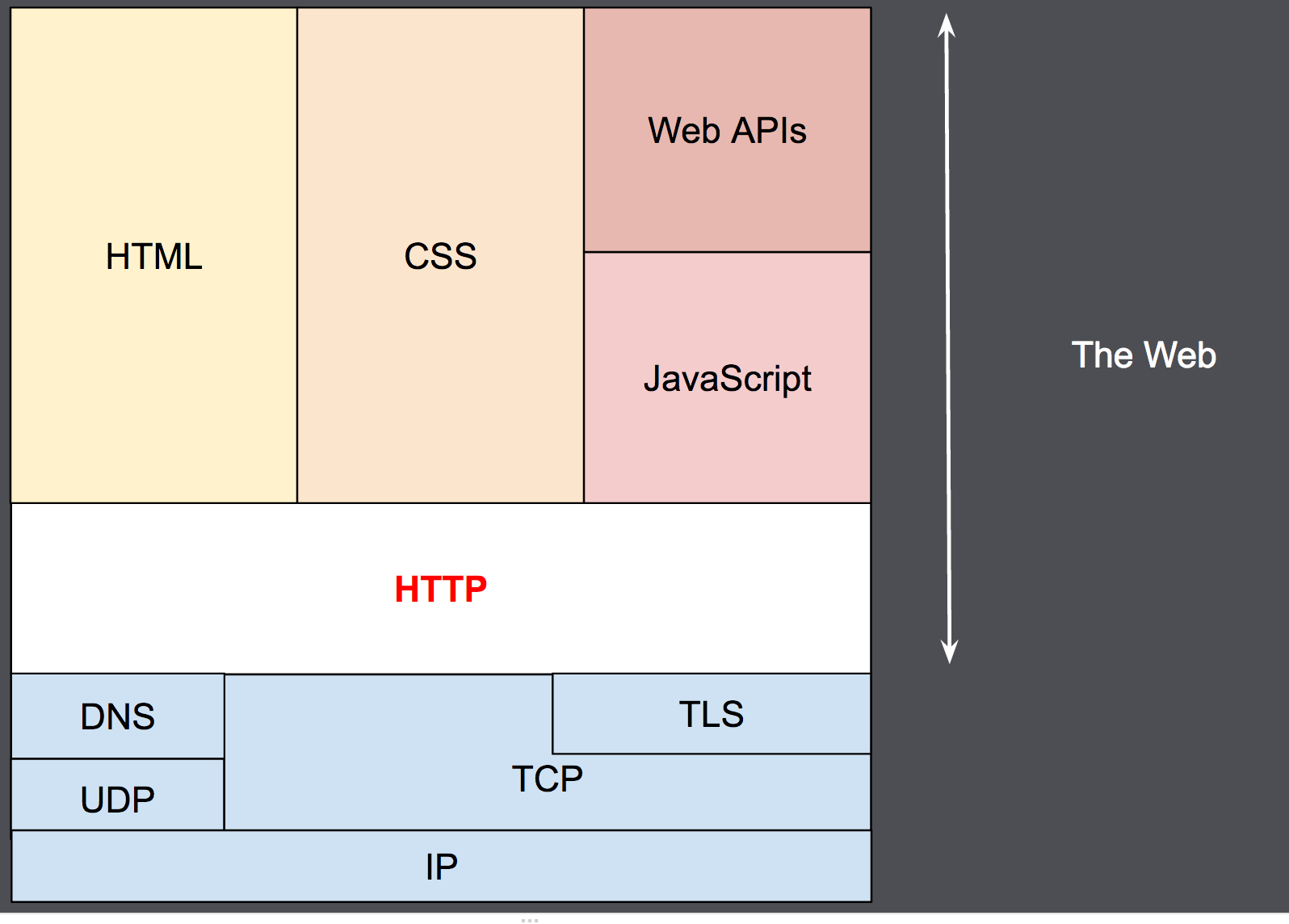
# 

# **Uma visão geral do HTTP**

**HTTP** é um [protocolo](https://developer.mozilla.org/en-US/docs/Glossary/Protocol) que permite a busca de recursos, como documentos HTML. É a base de qualquer troca de dados na Web e é um protocolo cliente-servidor, o que significa que as solicitações são iniciadas pelo destinatário, geralmente o navegador da Web. Um documento completo é reconstruído a partir dos diferentes subdestratos buscados, por exemplo, texto, descrição do layout, imagens, vídeos, scripts e muito mais.



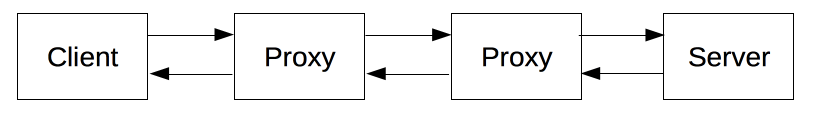
Clientes e servidores se comunicam trocando mensagens individuais (em oposição a um fluxo de dados). As mensagens enviadas pelo cliente, geralmente um navegador da Web, são chamadas *de solicitações* e as mensagens enviadas pelo servidor como resposta são *chamadas de respostas.*

**Projetado no início da década de 1990, HTTP é um protocolo extensível que evoluiu ao longo do tempo. É um protocolo de camada de aplicativo que é enviado através [do TCP](https://developer.mozilla.org/en-US/docs/Glossary/TCP), ou sobre uma conexão TCP criptografada [TLS,](https://developer.mozilla.org/en-US/docs/Glossary/TLS)embora qualquer protocolo de transporte confiável possa teoricamente ser usado. Devido à sua extensibilidade, ele é usado não apenas para buscar documentos de hipertexto, mas também imagens e vídeos ou para postar conteúdo para servidores, como com resultados de formulário HTML. HTTP também pode ser usado para buscar partes de documentos para atualizar páginas da Web sob demanda.

## [**Componentes de sistemas baseados em HTTP**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#components_of_http-based_systems)

HTTP é um protocolo cliente-servidor: as solicitações são enviadas por uma entidade, pelo usuário-agente (ou por um proxy em nome dele). Na maioria das vezes, o usuário-agente é um navegador da Web, mas pode ser qualquer coisa, por exemplo, um robô que rastreia a Web para preencher e manter um índice de mecanismo de pesquisa.

Cada solicitação individual é enviada para um servidor, que lida com ele e fornece uma resposta, chamada *de resposta*. Entre o cliente e o servidor existem inúmeras entidades, coletivamente chamadas [de proxies,](https://developer.mozilla.org/en-US/docs/Glossary/Proxy_server)que realizam diferentes operações e atuam como gateways ou [caches,](https://developer.mozilla.org/en-US/docs/Glossary/Cache)por exemplo.



In reality, there are more computers between a browser and the server handling the request: there are routers, modems, and more. Thanks to the layered design of the Web, these are hidden in the network and transport layers. HTTP is on top, at the application layer. Although important to diagnose network problems, the underlying layers are mostly irrelevant to the description of HTTP.

### [Client: the user-agent](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#client_the_user-agent)

The *user-agent* is any tool that acts on the behalf of the user. This role is primarily performed by the Web browser; other possibilities are programs used by engineers and Web developers to debug their applications.

The browser is **always** the entity initiating the request. It is never the server (though some mechanisms have been added over the years to simulate server-initiated messages).

To present a Web page, the browser sends an original request to fetch the HTML document that represents the page. It then parses this file, making additional requests corresponding to execution scripts, layout information (CSS) to display, and sub-resources contained within the page (usually images and videos). The Web browser then mixes these resources to present to the user a complete document, the Web page. Scripts executed by the browser can fetch more resources in later phases and the browser updates the Web page accordingly.

A Web page is a hypertext document. This means some parts of displayed text are links which can be activated (usually by a click of the mouse) to fetch a new Web page, allowing the user to direct their user-agent and navigate through the Web. The browser translates these directions in HTTP requests, and further interprets the HTTP responses to present the user with a clear response.

### [The Web server](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#the_web_server)

On the opposite side of the communication channel, is the server, which *serves* the document as requested by the client. A server appears as only a single machine virtually: this is because it may actually be a collection of servers, sharing the load (load balancing) or a complex piece of software interrogating other computers (like cache, a DB server, or e-commerce servers), totally or partially generating the document on demand.

A server is not necessarily a single machine, but several server software instances can be hosted on the same machine. With HTTP/1.1 and the [Host](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Host) header, they may even share the same IP address.

### [Proxies](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#proxies)

Between the Web browser and the server, numerous computers and machines relay the HTTP messages. Due to the layered structure of the Web stack, most of these operate at the transport, network or physical levels, becoming transparent at the HTTP layer and potentially making a significant impact on performance. Those operating at the application layers are generally called **proxies**. These can be transparent, forwarding on the requests they receive without altering them in any way, or non-transparent, in which case they will change the request in some way before passing it along to the server. Proxies may perform numerous functions:

* caching (the cache can be public or private, like the browser cache)
* filtering (like an antivirus scan or parental controls)
* load balancing (to allow multiple servers to serve the different requests)
* authentication (to control access to different resources)
* logging (allowing the storage of historical information)

## [**Basic aspects of HTTP**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#basic_aspects_of_http)

### [HTTP is simple](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_simple)

HTTP is generally designed to be simple and human readable, even with the added complexity introduced in HTTP/2 by encapsulating HTTP messages into frames. HTTP messages can be read and understood by humans, providing easier testing for developers, and reduced complexity for newcomers.

### [HTTP is extensible](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_extensible)

Introduced in HTTP/1.0, [HTTP headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers) make this protocol easy to extend and experiment with. New functionality can even be introduced by a simple agreement between a client and a server about a new header's semantics.

### [HTTP is stateless, but not sessionless](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_stateless_but_not_sessionless)

HTTP is stateless: there is no link between two requests being successively carried out on the same connection. This immediately has the prospect of being problematic for users attempting to interact with certain pages coherently, for example, using e-commerce shopping baskets. But while the core of HTTP itself is stateless, HTTP cookies allow the use of stateful sessions. Using header extensibility, HTTP Cookies are added to the workflow, allowing session creation on each HTTP request to share the same context, or the same state.

### [HTTP and connections](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_and_connections)

A connection is controlled at the transport layer, and therefore fundamentally out of scope for HTTP. Though HTTP doesn't require the underlying transport protocol to be connection-based; only requiring it to be *reliable*, or not lose messages (so at minimum presenting an error). Among the two most common transport protocols on the Internet, TCP is reliable and UDP isn't. HTTP therefore relies on the TCP standard, which is connection-based.

Before a client and server can exchange an HTTP request/response pair, they must establish a TCP connection, a process which requires several round-trips. The default behavior of HTTP/1.0 is to open a separate TCP connection for each HTTP request/response pair. This is less efficient than sharing a single TCP connection when multiple requests are sent in close succession.

In order to mitigate this flaw, HTTP/1.1 introduced *pipelining* (which proved difficult to implement) and *persistent connections*: the underlying TCP connection can be partially controlled using the [Connection](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Connection) header. HTTP/2 went a step further by multiplexing messages over a single connection, helping keep the connection warm and more efficient.

Experiments are in progress to design a better transport protocol more suited to HTTP. For example, Google is experimenting with [QUIC](https://en.wikipedia.org/wiki/QUIC) which builds on UDP to provide a more reliable and efficient transport protocol.

## [**What can be controlled by HTTP**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#what_can_be_controlled_by_http)

This extensible nature of HTTP has, over time, allowed for more control and functionality of the Web. Cache or authentication methods were functions handled early in HTTP history. The ability to relax the *origin constraint*, by contrast, has only been added in the 2010s.

Here is a list of common features controllable with HTTP.

* [*Caching*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching)How documents are cached can be controlled by HTTP. The server can instruct proxies and clients, about what to cache and for how long. The client can instruct intermediate cache proxies to ignore the stored document.
* *Relaxing the origin constraint*To prevent snooping and other privacy invasions, Web browsers enforce strict separation between Web sites. Only pages from the **same origin** can access all the information of a Web page. Though such constraint is a burden to the server, HTTP headers can relax this strict separation on the server side, allowing a document to become a patchwork of information sourced from different domains; there could even be security-related reasons to do so.
* *Authentication*Some pages may be protected so that only specific users can access them. Basic authentication may be provided by HTTP, either using the [WWW-Authenticate](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/WWW-Authenticate) and similar headers, or by setting a specific session using [HTTP cookies](https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies).
* [*Proxy and tunneling*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Proxy_servers_and_tunneling)Servers or clients are often located on intranets and hide their true IP address from other computers. HTTP requests then go through proxies to cross this network barrier. Not all proxies are HTTP proxies. The SOCKS protocol, for example, operates at a lower level. Other protocols, like ftp, can be handled by these proxies.
* *Sessions*Using HTTP cookies allows you to link requests with the state of the server. This creates sessions, despite basic HTTP being a state-less protocol. This is useful not only for e-commerce shopping baskets, but also for any site allowing user configuration of the output.

## [**HTTP flow**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_flow)

When a client wants to communicate with a server, either the final server or an intermediate proxy, it performs the following steps:

1. Open a TCP connection: The TCP connection is used to send a request, or several, and receive an answer. The client may open a new connection, reuse an existing connection, or open several TCP connections to the servers.

Send an HTTP message: HTTP messages (before HTTP/2) are human-readable. With HTTP/2, these simple messages are encapsulated in frames, making them impossible to read directly, but the principle remains the same. For example:  
GET / HTTP/1.1

Host: developer.mozilla.org

1. Accept-Language: fr

Read the response sent by the server, such as:  
HTTP/1.1 200 OK

Date: Sat, 09 Oct 2010 14:28:02 GMT

Server: Apache

Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT

ETag: "51142bc1-7449-479b075b2891b"

Accept-Ranges: bytes

Content-Length: 29769

Content-Type: text/html

1. <!DOCTYPE html... (here comes the 29769 bytes of the requested web page)
2. Close or reuse the connection for further requests.

If HTTP pipelining is activated, several requests can be sent without waiting for the first response to be fully received. HTTP pipelining has proven difficult to implement in existing networks, where old pieces of software coexist with modern versions. HTTP pipelining has been superseded in HTTP/2 with more robust multiplexing requests within a frame.

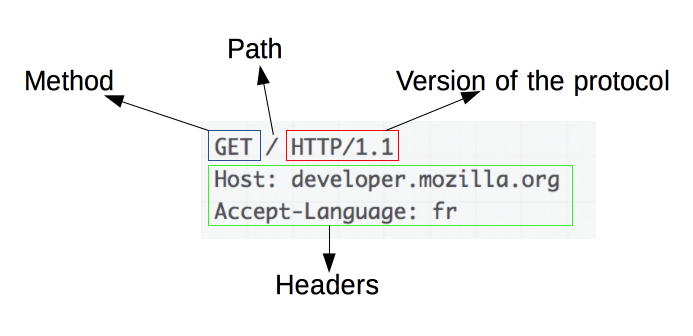
## [**HTTP Messages**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_messages)

HTTP messages, as defined in HTTP/1.1 and earlier, are human-readable. In HTTP/2, these messages are embedded into a binary structure, a *frame*, allowing optimizations like compression of headers and multiplexing. Even if only part of the original HTTP message is sent in this version of HTTP, the semantics of each message is unchanged and the client reconstitutes (virtually) the original HTTP/1.1 request. It is therefore useful to comprehend HTTP/2 messages in the HTTP/1.1 format.

There are two types of HTTP messages, requests and responses, each with its own format.

### [Requests](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#requests)

An example HTTP request:

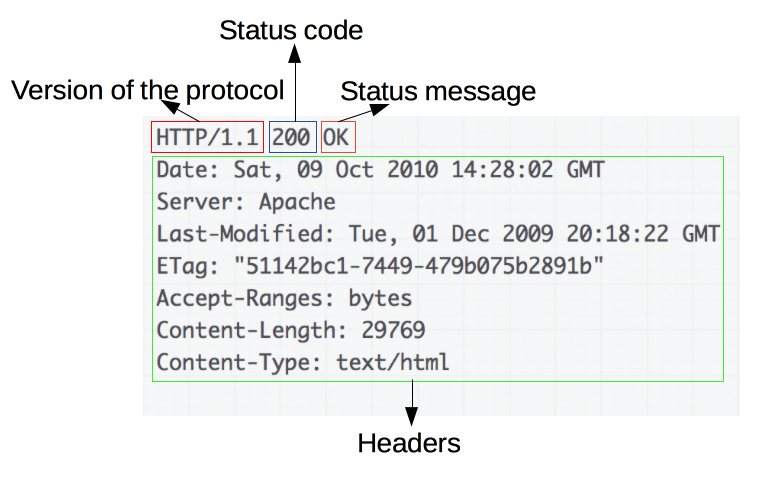


As solicitações consistem nos seguintes elementos:

* Um [método](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)HTTP , geralmente um verbo como [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET), [POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST) ou um substantivo como [OPTIONS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS) ou [HEAD](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD) que define a operação que o cliente deseja realizar. Normalmente, um cliente quer buscar um recurso (usando ) ou postar o valor de um [formulário HTML](https://developer.mozilla.org/en-US/docs/Learn/Forms) (usando), embora mais operações possam ser necessárias em outros casos.GETPOST
* O caminho do recurso para buscar; a URL do recurso retirado de elementos que são óbvios do contexto, por exemplo, sem o [protocolo](https://developer.mozilla.org/en-US/docs/Glossary/Protocol) (), o [domínio](https://developer.mozilla.org/en-US/docs/Glossary/Domain) (aqui), ou a [porta](https://developer.mozilla.org/en-US/docs/Glossary/Port) TCP (aqui, ).http://developer.mozilla.org80
* A versão do protocolo HTTP.
* [Cabeçalhos opcionais](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers) que transmitem informações adicionais para os servidores.
* Ou um corpo, para alguns métodos como, semelhantes aos de respostas, que contêm o recurso enviado.POST

### [Respostas](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#responses)

Uma resposta de exemplo:



As respostas consistem nos seguintes elementos:

* A versão do protocolo HTTP que eles seguem.
* Um [código de status,](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)indicando se a solicitação foi bem sucedida, ou não, e por quê.
* Uma mensagem de status, uma descrição curta não autorizada do código de status.
* [Cabeçalhos](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers)HTTP, como os pedidos.
* Opcionalmente, um corpo contendo o recurso buscado.

## [**APIs baseadas em HTTP**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#apis_based_on_http)

A API mais usada com base no HTTP é a API [XMLHttpRequest,](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) que pode ser usada para trocar dados entre um [agente de usuário](https://developer.mozilla.org/en-US/docs/Glossary/User_agent) e um servidor. A [API fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) moderna fornece os mesmos recursos com um conjunto de recursos mais potente e flexível.

Outra API, [eventos enviados por servidor,](https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events)é um serviço unidiretor que permite que um servidor envie eventos ao cliente, usando HTTP como mecanismo de transporte. Usando a interface [EventSource,](https://developer.mozilla.org/en-US/docs/Web/API/EventSource) o cliente abre uma conexão e estabelece manipuladores de eventos. O navegador cliente converte automaticamente as mensagens que chegam no fluxo HTTP em objetos de [eventos](https://developer.mozilla.org/en-US/docs/Web/API/Event) apropriados, entregando-as aos manipuladores de eventos que foram registrados para o [tipo](https://developer.mozilla.org/en-US/docs/Web/API/Event/type) de eventos, se conhecidos, ou para o manipulador de eventos [onmessage](https://developer.mozilla.org/en-US/docs/Web/API/EventSource/onmessage) se nenhum manipulador de eventos específico do tipo foi estabelecido.

## [**Conclusão**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#conclusion)

HTTP é um protocolo extensível que é fácil de usar. A estrutura cliente-servidor, combinada com a capacidade de adicionar cabeçalhos, permite que o HTTP avance junto com os recursos estendidos da Web.

Embora o HTTP/2 adicione alguma complexidade, incorporando mensagens HTTP em quadros para melhorar o desempenho, a estrutura básica das mensagens permaneceu a mesma desde HTTP/1.0. O fluxo de sessão permanece simples, permitindo que ele seja investigado e depurado com um simples [monitor de mensagens HTTP](https://developer.mozilla.org/en-US/docs/Tools/Network_Monitor).

link: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>