HTTP messages

HTTP messages are the mechanism used to exchange data between a server and a client in the HTTP protocol. There are two types of messages: **requests** sent by the client to trigger an action on the server, and **responses**, the answer that the server sends in response to a request.

Developers rarely, if ever, build HTTP messages from scratch. Applications such as a browser, proxy, or web server use software designed to create HTTP messages in a reliable and efficient way. How messages are created or transformed is controlled via APIs in browsers, configuration files for proxies or servers, or other interfaces.

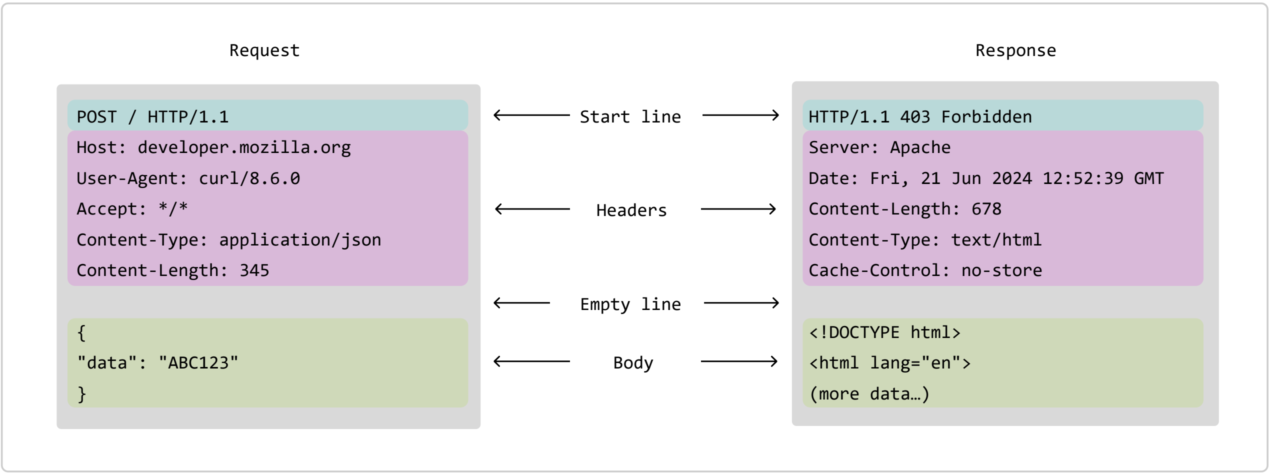
In HTTP protocol versions up to HTTP/2, messages are text-based, and are relatively straightforward to read and understand after you've familiarized yourself with the format. In HTTP/2, messages are wrapped in binary framing, which makes them slightly harder to read without certain tools. However the underlying semantics of the protocol are the same, so you can learn the structure and meaning of HTTP messages based on the text-based format of HTTP/1.x messages, and apply this understanding to HTTP/2 and beyond.

This guide uses HTTP/1.1 messages for readability, and explains the structure of HTTP messages using the HTTP/1.1 format. We highlight some differences that you might need for describing HTTP/2 in the final section.

Note: You can see HTTP messages in a browser's **Network** tab in the developer tools, or if you print HTTP messages to the console using CLI tools such as curl, for example.

Anatomy of an HTTP message

To understand how HTTP messages work, we'll look at HTTP/1.1 messages and examine the structure. The following illustration shows what messages in HTTP/1.1 look like:



```
POST / HTTP/1.1
Host: developer.mozilla.org
User-Agent: curl/8.6.0
Accept: */*
Content-Type: application/json
Content-Length: 345

{
"data": "ABC123"
}
```

```
HTTP/1.1 403 Forbidden
Server: Apache
Date: Fri, 21 Jun 2024 12:52:39 GMT
Content-Length: 678
Content-Type: text/html
Cache-Control: no-store

<!DOCTYPE html>
<html lang="en">
(more data…)
```

Both requests and responses share a similar structure:

1. A *start-line* is a single line that describes the HTTP version along with the request method or the outcome of the request.
2. An optional set of *HTTP headers* containing metadata that describes the message. For example, a request for a resource might include the allowed formats of that resource, while the response might include headers to indicate the actual format returned.
3. An empty line indicating the metadata of the message is complete.
4. An optional *body* containing data associated with the message. This might be POST data to send to the server in a request, or some resource returned to the client in a response. Whether a message contains a body or not is determined by the start-line and HTTP headers.

The start-line and headers of the HTTP message are collectively known as the *head* of the requests, and the part afterwards that contains its content is known as the *body*.

HTTP requests

Let's look at the following example HTTP `POST` request that's sent after a user submits a form on a web page:

HTTP

```http
POST /users HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 49

name=FirstName+LastName&email=bsmth%40example.com
```

The start-line in HTTP/1.x requests (`POST /users HTTP/1.1` in the example above) is called a "request-line" and is made of three parts:

```http
<method> <request-target> <protocol>
```

`<method>`

The HTTP method (also known as an *HTTP verb*) is one of a set of defined words that describes the meaning of the request and the desired outcome. For example, `GET` indicates that the client would like to receive a resource in return, and `POST` means that the client is sending data to a server.

`<request-target>`

The request target is usually an absolute or relative URL, and is characterized by the context of the request. The format of the request target depends on the HTTP method used and the request context. It is described in more detail in the Request targets section below.

`<protocol>`

The *HTTP version*, which defines the structure of the remaining message, acting as an indicator of the expected version to use for the response. This is almost always `HTTP/1.1`, as `HTTP/0.9` and `HTTP/1.0` are obsolete. In HTTP/2 and above, the protocol version isn't included in messages since it is understood from the connection setup.

Request targets

There are a few ways of describing a request target, but by far the most common is the "origin form". Here's a list of the types of targets and when they are used:

1. In *origin form*, the recipient combines an absolute path with the information in the `Host` header. A query string can be appended to the path for additional information (usually in `key=value` format). This is used with `GET`, `POST`, `HEAD`, and `OPTIONS` methods:

   ```http
   GET /en-US/docs/Web/HTTP/Guides/Messages HTTP/1.1
   ```

2. The *absolute form* is a complete URL, including the authority, and is used with `GET` when connecting to a proxy:

   ```http
   GET https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Messages HTTP/1.1
   ```

3. The *authority form* is the authority and port separated by a colon (`:`). It is only used with the `CONNECT` method when setting up an HTTP tunnel:

HTTP

```
CONNECT developer.mozilla.org:443 HTTP/1.1
```

4. The *asterisk form* is only used with `OPTIONS` when you want to represent the server as a whole (`*`) as opposed to a named resource:

HTTP

```
OPTIONS * HTTP/1.1
```

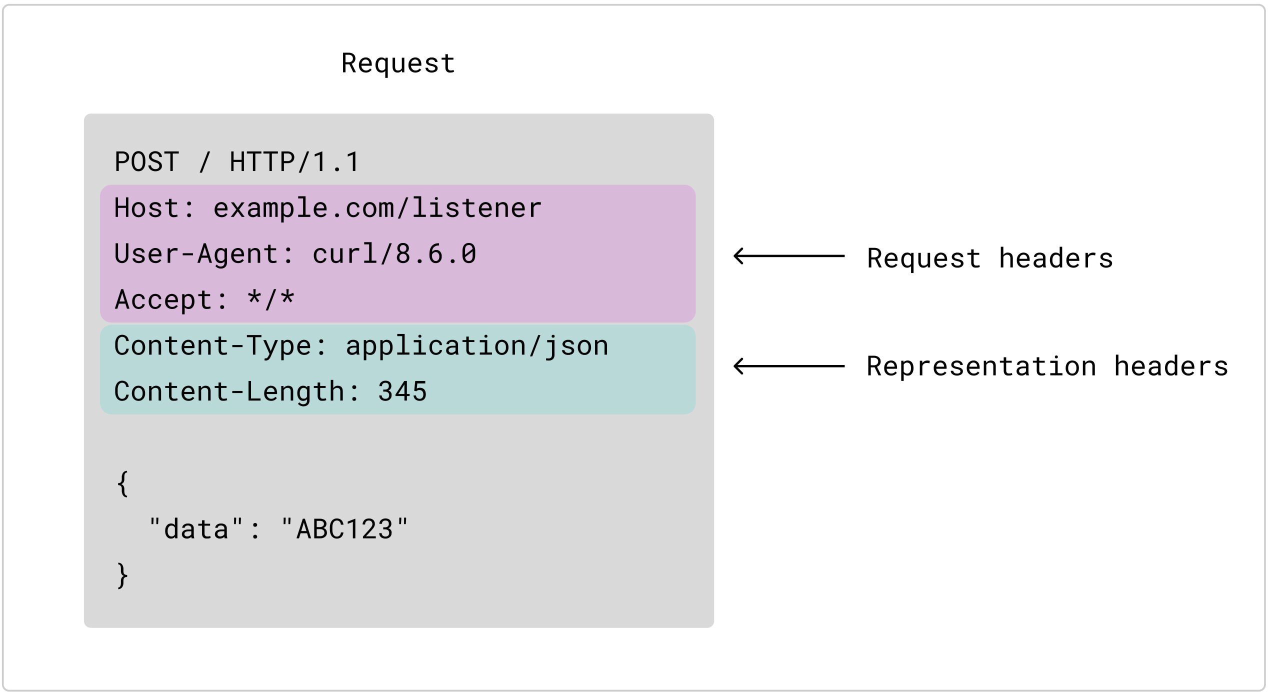
Request headers

Headers are metadata sent with a request after the start line and before the body. In the form submission example above, they are the following lines of the message:

HTTP

```
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 49
```

In HTTP/1.x, each header is a **case-insensitive** string followed by a colon (`:`) and a value whose format depends on the header. The whole header, including the value, consists of one single line. This line can be quite long in some cases, such as the `Cookie` header.



```
POST / HTTP/1.1
Host: example.com/listener
User-Agent: curl/8.6.0
Accept: */*
Content-Type: application/json
Content-Length: 345

{
  "data": "ABC123"
}
```

Some headers are exclusively used in requests, while others can be sent in both requests and responses, or might have a more specific categorization:

- Request headers provide additional context to a request or add extra logic to how it should be treated by a server (e.g., conditional requests).
- Representation headers are sent in a request if the message has a body, and they describe the original form of the message data and any encoding applied. This allows the recipient to understand how to reconstruct the resource as it was before it was transmitted over the network.

Request body

The request body is the part of a request that carries information to the server. Only `PATCH`, `POST`, and `PUT` requests have a body. In the form submission example, this part is the body:

HTTP

```http
name=FirstName+LastName&email=bsmth%40example.com
```

The body in the form submission request contains a relatively small amount of information as `key=value` pairs, but a request body could contain other types of data that the server expects:

JSON

```json
{
  "firstName": "Brian",
  "lastName": "Smith",
  "email": "bsmth@example.com",
  "more": "data"
}
```

or data in multiple parts:

HTTP

```http
--delimiter123
Content-Disposition: form-data; name="field1"

value1
--delimiter123
Content-Disposition: form-data; name="field2"; filename="example.txt"

Text file contents
--delimiter123--
```

HTTP responses

Responses are the HTTP messages a server sends back in reply to a request. The response lets the client know what the outcome of the request was. Here's an example HTTP/1.1 response to a `POST` request that created a new user:

HTTP

```http
HTTP/1.1 201 Created
Content-Type: application/json
Location: http://example.com/users/123

{
  "message": "New user created",
  "user": {
    "id": 123,
    "firstName": "Example",
    "lastName": "Person",
    "email": "bsmth@example.com"
  }
}
```

The start-line (`HTTP/1.1 201 Created` above) is called a "status line" in responses, and has three parts:

HTTP

```http
<protocol> <status-code> <status-text>
```

`<protocol>`

The *HTTP version* of the remaining message.

`<status-code>`

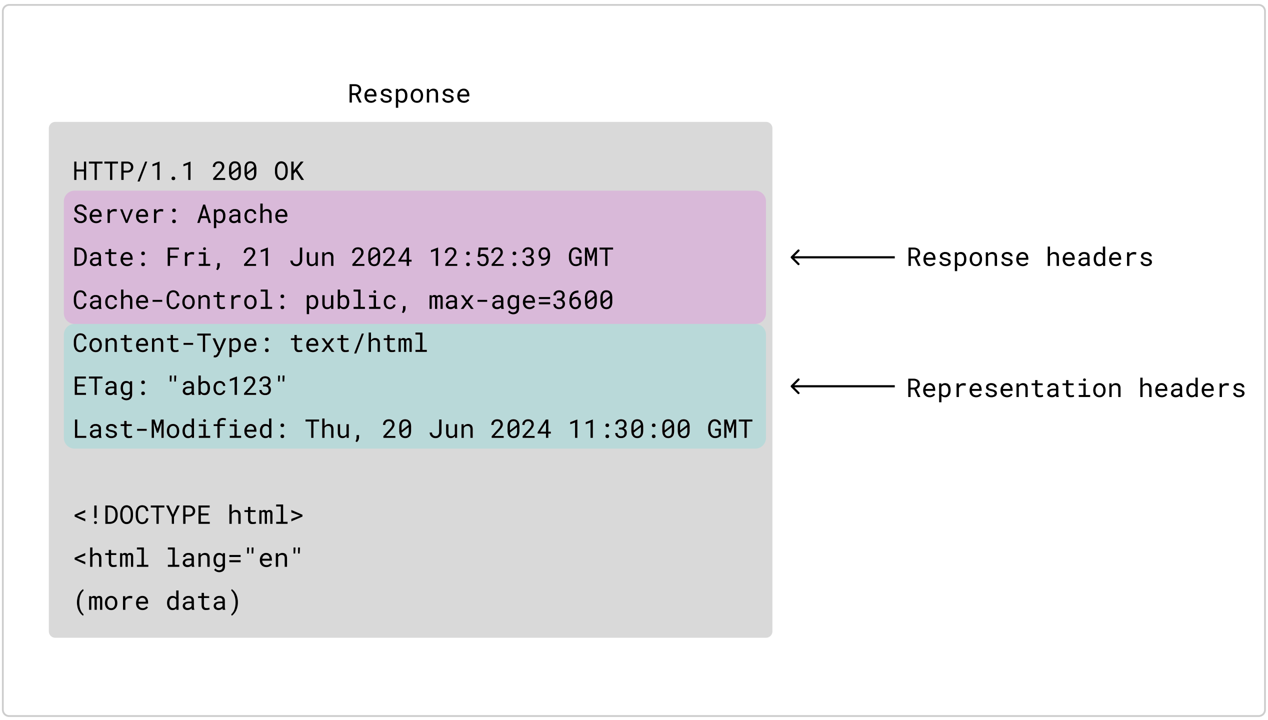
A numeric status code that indicates whether the request succeeded or failed. Common status codes are `200`, `404`, or `302`.

`<status-text>`

The status text is a brief, purely informational, textual description of the status code to help a human understand the HTTP message.

Response headers

Response headers are the metadata sent with a response. In HTTP/1.x, each header is a **case-insensitive** string followed by a colon (`:`) and a value whose format depends upon which header is used.



```
HTTP/1.1 200 OK
Server: Apache
Date: Fri, 21 Jun 2024 12:52:39 GMT
Cache-Control: public, max-age=3600
Content-Type: text/html
ETag: "abc123"
Last-Modified: Thu, 20 Jun 2024 11:30:00 GMT

<!DOCTYPE html>
<html lang="en"
(more data)
```

Like request headers, there are many different headers that can appear in responses, and they are categorized as:

- Response headers that give additional context about the message or add extra logic to how the client should make subsequent requests. For example, headers like `Server` include information about the server software, while `Date` includes when the response was generated. There is also information about the resource being returned, such as its content type (`Content-Type`), or how it should be cached (`Cache-Control`).
- Representation headers if the message has a body, they describe the form of the message data and any encoding applied. For example, the same resource might be formatted in a particular media type such as XML or JSON, localized to a particular written language or geographical region, and/or compressed or otherwise encoded for transmission. This allows a recipient to understand how to reconstruct the resource as it was before it was transmitted over the network.

Response body

A response body is included in most messages when responding to a client. In successful requests, the response body contains the data that the client asked for in a `GET` request. If there are problems with the client's request, it's common for the response body to describe why the request failed, and hint as to whether it's permanent or temporary.

Response bodies may be:

- Single-resource bodies defined by the two headers: `Content-Type` and `Content-Length`, or of unknown length and encoded in chunks with `Transfer-Encoding` set to `chunked`.
- Multiple-resource bodies, consisting of a body that contains multiple parts, each containing a different piece of information. Multipart bodies are typically associated with HTML Forms, but may also be sent in response to Range requests.

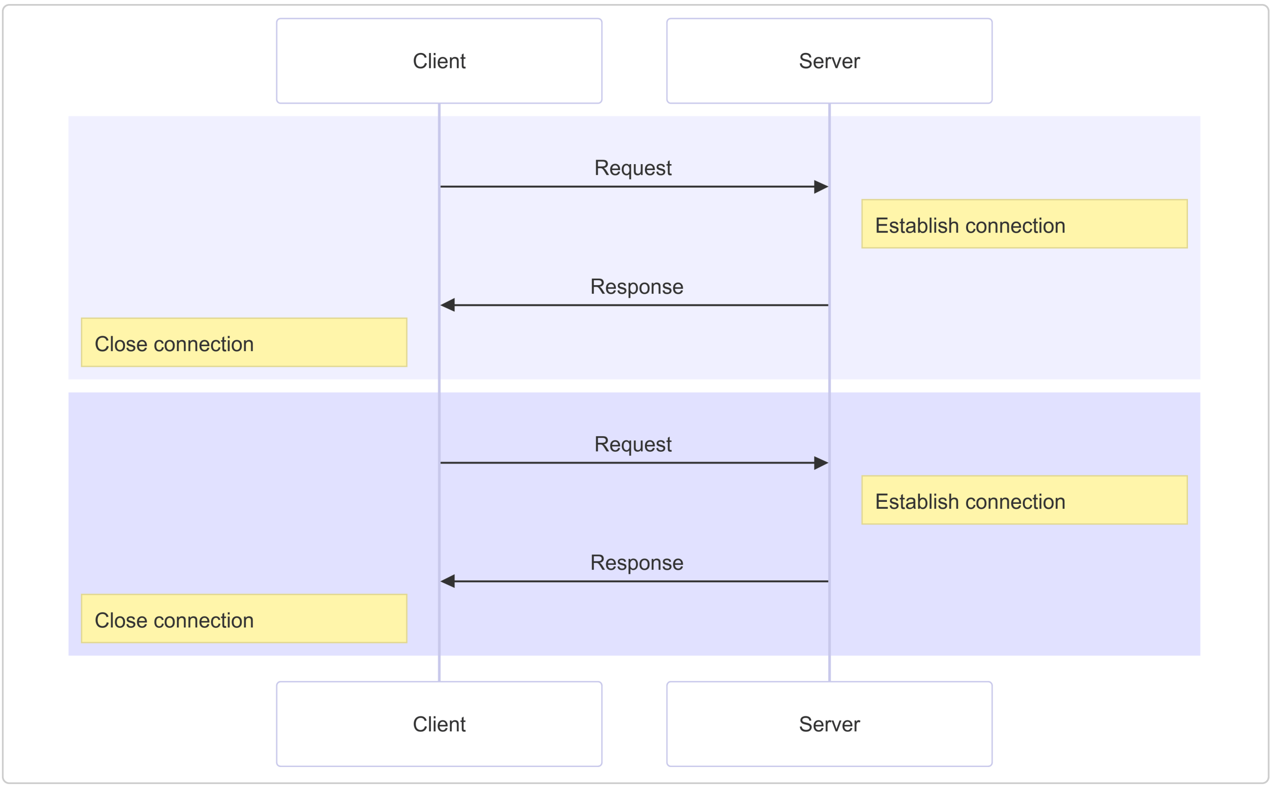
Responses with a status code that answers the request without the need to include message content, such as `201 Created` or `204 No Content`, do not have a body.

HTTP/2 messages

HTTP/1.x uses text-based messages that are straightforward to read and construct, but as a result have a few downsides. You can compress message bodies using `gzip` or other compression algorithms, but not headers. Headers are often similar or identical in a client-server interaction, but they are repeated in successive messages on a connection. There are many known methods to compress repetitive text that are very efficient, which leaves a large amount of bandwidth savings unutilized.

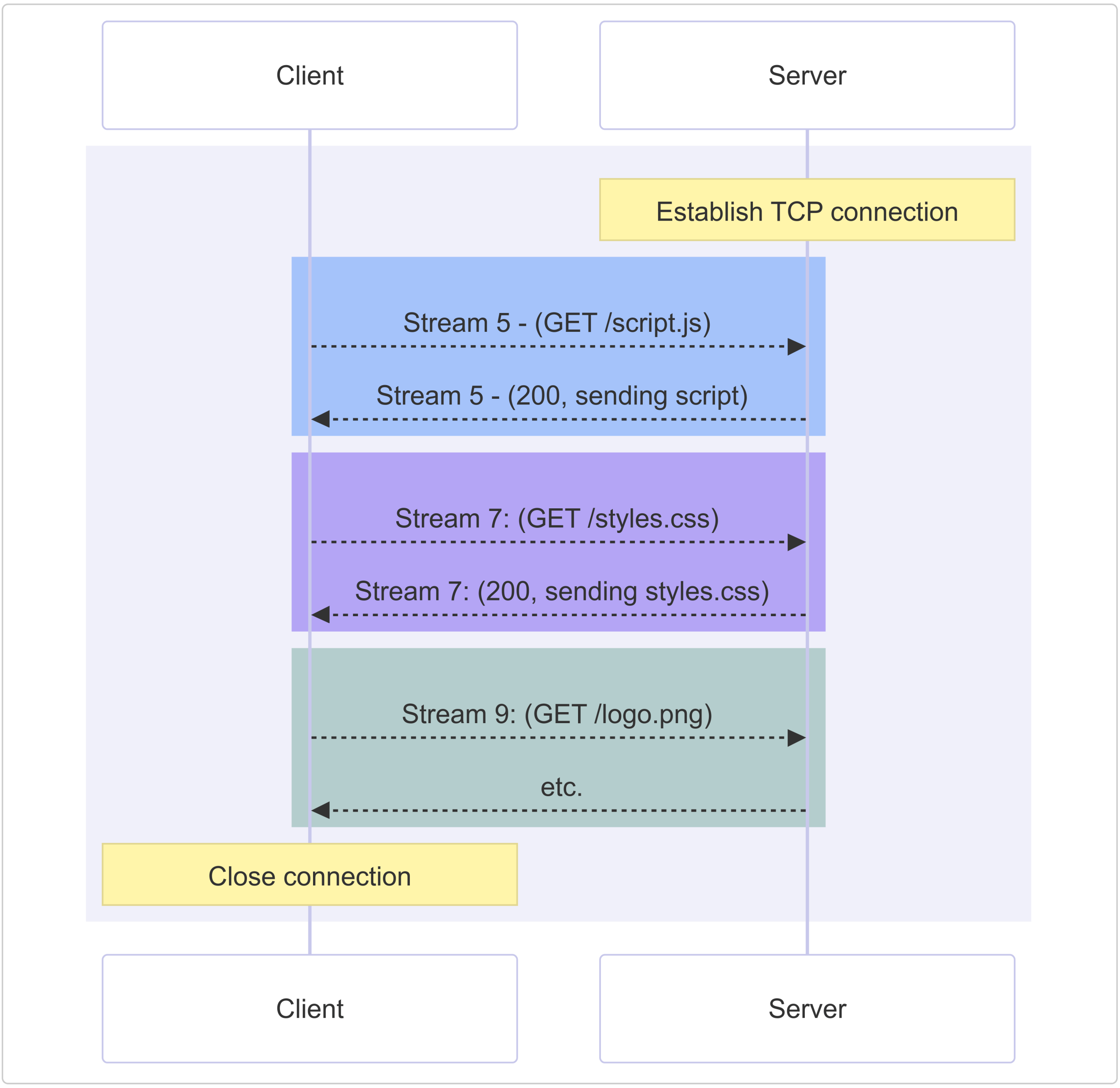
HTTP/1.x also has a problem called head-of-line (HOL) blocking, where a client has to wait for a response from the server before sending the next request. HTTP pipelining tried to work around this, but poor support and complexity means it's rarely used and difficult to get right. Several connections need to be opened to send requests concurrently; and warm (established and busy) connections are more efficient than cold ones due to TCP slow start.

In HTTP/1.1 if you want to make two requests in parallel, you have to open two connections:



This means that browsers are limited in the number of resources that they can download and render at the same time, which has typically been limited to 6 parallel connections.

HTTP/2 allows you to use a single TCP connection for multiple requests and responses at the same time. This is done by wrapping messages into a binary frame and sending the requests and responses in a numbered **stream** on a connection. Data and header frames are handled separately, which allows headers to be compressed via an algorithm called HPACK. Using the same TCP connection to handle multiple requests at the same time is called *multiplexing*.


Client
Server
Establish TCP connection
Stream 5 - (GET /script.js)
Stream 5 - (200, sending script)
Stream 7: (GET /styles.css)
Stream 7: (200, sending styles.css)
Stream 9: (GET /logo.png)
etc.
Close connection
Client
Server

Requests are not necessarily sequential: stream 9 doesn't have to wait for stream 7 to finish, for instance. The data from multiple streams are usually interleaved on the connection, so stream 9 and 7 can be received by the client at the same time. There's a mechanism for the protocol to set a priority for each stream or resource. Low-priority resources take up less bandwidth than higher-priority resources when they're being sent over different streams, or they could effectively be sent sequentially on the same connection if there are critical resources that should be handled first.

In general, despite all of the improvements and abstractions added over HTTP/1.x, virtually no changes are needed in the APIs used by developers to make use of HTTP/2 over HTTP/1.x. When HTTP/2 is available in both the browser and the server, it is switched on and used automatically.

Pseudo-headers

One notable change to messages in HTTP/2 are the use of pseudo-headers. Where HTTP/1.x used the message start-line, HTTP/2 uses special pseudo-header fields beginning with `:`. In requests, there are the following pseudo-headers:

- `:method` - the HTTP method.
- `:scheme` - the scheme portion of the target URI, which is often HTTP(S).
- `:authority` - the authority portion of the target URI.
- `:path` - the path and query parts of the target URI.

In responses, there is only one pseudo-header, and that's the `:status` which provides the code of the response.

We can make a HTTP/2 request using nghttp to fetch `example.com`, which will print out the request in a form that's more readable. You can make the request using this command where the `-n` option discards the downloaded data and `-v` is for 'verbose' output, showing reception and transmission of frames:

BASH

```bash
nghttp -nv https://www.example.com
```

If you look down through the output, you'll see the timing for each frame transmitted and received:

```
[  0.123] <send|recv> <frame-type> <frame-details>
```

We don't have to go into too much detail on this output, but look out for the `HEADERS` frame in the format `[ 0.123] send HEADERS frame ...`. In the lines after the header transmission, you will see the following lines:

HTTP

```http
[  0.447] send HEADERS frame ...
          ...
          :method: GET
          :path: /
          :scheme: https
          :authority: www.example.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.61.0
```

This should look familiar if you're already comfortable working with HTTP/1.x and the concepts covered in the earlier section of this guide still apply. This is the binary frame with the `GET` request for `example.com`, converted into a readable form by `nghttp`. If you look further down the output of the command, you will see the `:status` pseudo-header in one of the streams received from the server:

HTTP

```
[  0.433] recv (stream_id=13) :status: 200
[  0.433] recv (stream_id=13) content-encoding: gzip
[  0.433] recv (stream_id=13) age: 112721
[  0.433] recv (stream_id=13) cache-control: max-age=604800
[  0.433] recv (stream_id=13) content-type: text/html; charset=UTF-8
[  0.433] recv (stream_id=13) date: Fri, 13 Sep 2024 12:56:07 GMT
[  0.433] recv (stream_id=13) etag: "3147526947+gzip"
...
```

And if you remove the timing and stream ID from this message, it should be even more familiar:

HTTP

```
:status: 200
content-encoding: gzip
age: 112721
```

Digging further into message frames, stream IDs and how the connection is managed is beyond the scope of this guide, but for the purpose of understanding and debugging HTTP/2 messages, you should be well-equipped using the knowledge and tools in this article.

Conclusion

This guide provides a general overview of the anatomy of HTTP messages, using the HTTP/1.1 format for illustration. We also explored HTTP/2 message framing, which introduces a layer between the HTTP/1.x syntax and the underlying transport protocol without fundamentally modifying HTTP's semantics. HTTP/2 was introduced to solve the head-of-line blocking issues present in HTTP/1.x by enabling multiplexing of requests.

One issue that remained in HTTP/2 is that even though head-of-line blocking was fixed in the protocol level, there is still a performance bottleneck due to head-of-line blocking within TCP (at the transport level). HTTP/3 addresses this limitation by using QUIC, a protocol built on UDP, instead of TCP. This change improves performance, reduces connection setup time, and enhances stability on degraded or unreliable networks. HTTP/3 retains the same core

HTTP semantics, so features like request methods, status codes, and headers remain consistent across all three major HTTP versions.

If you understand HTTP/1.1's semantics, you already have a solid foundation for grasping HTTP/2 and HTTP/3. The main difference lies in **how** these semantics are implemented at the transport level. By following the examples and concepts in this guide, you should now feel equipped to work with HTTP and understand the meaning of messages, and how applications use HTTP to send and receive data.

See also

- Evolution of HTTP
- Protocol upgrade mechanism
- Glossary terms:
  - HTTP
  - HTTP/2
  - QUIC

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Jul 4, 2025 by MDN contributors.