**HTTP request methods**

HTTP defines a set of **request methods** to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as *HTTP verbs*. Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be [safe](https://developer.mozilla.org/en-US/docs/Glossary/Safe/HTTP), [idempotent](https://developer.mozilla.org/en-US/docs/Glossary/Idempotent), or [cacheable](https://developer.mozilla.org/en-US/docs/Glossary/Cacheable).

[GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET)

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

[HEAD](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD)

The HEAD method asks for a response identical to a GET request, but without the response body.

[POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST)

The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.

[PUT](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PUT)

The PUT method replaces all current representations of the target resource with the request payload.

[DELETE](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/DELETE)

The DELETE method deletes the specified resource.

[CONNECT](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/CONNECT)

The CONNECT method establishes a tunnel to the server identified by the target resource.

[OPTIONS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS)

The OPTIONS method describes the communication options for the target resource.

[TRACE](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/TRACE)

The TRACE method performs a message loop-back test along the path to the target resource.

[PATCH](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH)

The PATCH method applies partial modifications to a resource.

### [Method Definitions](https://httpwg.org/specs/rfc9110.html#method.definitions)

#### [9.3.1.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.1) [GET](https://httpwg.org/specs/rfc9110.html#GET)

The GET method requests transfer of a current [selected representation](https://httpwg.org/specs/rfc9110.html#selected.representation) for the [target resource](https://httpwg.org/specs/rfc9110.html#target.resource). A successful response reflects the quality of "sameness" identified by the target URI ([Section 1.2.2](https://www.rfc-editor.org/rfc/rfc3986.html#section-1.2.2) of [[URI]](https://httpwg.org/specs/rfc9110.html#URI)). Hence, retrieving identifiable information via HTTP is usually performed by making a GET request on an identifier associated with the potential for providing that information in a [200 (OK)](https://httpwg.org/specs/rfc9110.html#status.200) response.

GET is the primary mechanism of information retrieval and the focus of almost all performance optimizations. Applications that produce a URI for each important resource can benefit from those optimizations while enabling their reuse by other applications, creating a network effect that promotes further expansion of the Web.

It is tempting to think of resource identifiers as remote file system pathnames and of representations as being a copy of the contents of such files. In fact, that is how many resources are implemented (see [Section 17.3](https://httpwg.org/specs/rfc9110.html#attack.pathname) for related security considerations). However, there are no such limitations in practice.

The HTTP interface for a resource is just as likely to be implemented as a tree of content objects, a programmatic view on various database records, or a gateway to other information systems. Even when the URI mapping mechanism is tied to a file system, an origin server might be configured to execute the files with the request as input and send the output as the representation rather than transfer the files directly. Regardless, only the origin server needs to know how each resource identifier corresponds to an implementation and how that implementation manages to select and send a current representation of the target resource.

A client can alter the semantics of GET to be a "range request", requesting transfer of only some part(s) of the selected representation, by sending a [Range](https://httpwg.org/specs/rfc9110.html#field.range) header field in the request ([Section 14.2](https://httpwg.org/specs/rfc9110.html#field.range)).

Although request message framing is independent of the method used, content received in a GET request has no generally defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling attack ([Section 11.2](https://httpwg.org/specs/rfc9112.html#request.smuggling) of [[HTTP/1.1]](https://httpwg.org/specs/rfc9110.html#HTTP11)). A client SHOULD NOT generate content in a GET request unless it is made directly to an origin server that has previously indicated, in or out of band, that such a request has a purpose and will be adequately supported. An origin server SHOULD NOT rely on private agreements to receive content, since participants in HTTP communication are often unaware of intermediaries along the request chain.

The response to a GET request is cacheable; a cache MAY use it to satisfy subsequent GET and HEAD requests unless otherwise indicated by the Cache-Control header field ([Section 5.2](https://httpwg.org/specs/rfc9111.html#field.cache-control) of [[CACHING]](https://httpwg.org/specs/rfc9110.html#CACHING)).

When information retrieval is performed with a mechanism that constructs a target URI from user-provided information, such as the query fields of a form using GET, potentially sensitive data might be provided that would not be appropriate for disclosure within a URI (see [Section 17.9](https://httpwg.org/specs/rfc9110.html#sensitive.information.in.uris)). In some cases, the data can be filtered or transformed such that it would not reveal such information. In others, particularly when there is no benefit from caching a response, using the POST method ([Section 9.3.3](https://httpwg.org/specs/rfc9110.html#POST)) instead of GET can transmit such information in the request content rather than within the target URI.

#### [9.3.2.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.2) [HEAD](https://httpwg.org/specs/rfc9110.html#HEAD)

The HEAD method is identical to GET except that the server MUST NOT send content in the response. HEAD is used to obtain metadata about the [selected representation](https://httpwg.org/specs/rfc9110.html#selected.representation) without transferring its representation data, often for the sake of testing hypertext links or finding recent modifications.

The server SHOULD send the same header fields in response to a HEAD request as it would have sent if the request method had been GET. However, a server MAY omit header fields for which a value is determined only while generating the content. For example, some servers buffer a dynamic response to GET until a minimum amount of data is generated so that they can more efficiently delimit small responses or make late decisions with regard to content selection. Such a response to GET might contain [Content-Length](https://httpwg.org/specs/rfc9110.html#field.content-length) and [Vary](https://httpwg.org/specs/rfc9110.html#field.vary) fields, for example, that are not generated within a HEAD response. These minor inconsistencies are considered preferable to generating and discarding the content for a HEAD request, since HEAD is usually requested for the sake of efficiency.

Although request message framing is independent of the method used, content received in a HEAD request has no generally defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling attack ([Section 11.2](https://httpwg.org/specs/rfc9112.html#request.smuggling) of [[HTTP/1.1]](https://httpwg.org/specs/rfc9110.html#HTTP11)). A client SHOULD NOT generate content in a HEAD request unless it is made directly to an origin server that has previously indicated, in or out of band, that such a request has a purpose and will be adequately supported. An origin server SHOULD NOT rely on private agreements to receive content, since participants in HTTP communication are often unaware of intermediaries along the request chain.

The response to a HEAD request is cacheable; a cache MAY use it to satisfy subsequent HEAD requests unless otherwise indicated by the Cache-Control header field ([Section 5.2](https://httpwg.org/specs/rfc9111.html#field.cache-control) of [[CACHING]](https://httpwg.org/specs/rfc9110.html#CACHING)). A HEAD response might also affect previously cached responses to GET; see [Section 4.3.5](https://httpwg.org/specs/rfc9111.html#head.effects) of [[CACHING]](https://httpwg.org/specs/rfc9110.html#CACHING).

#### [9.3.3.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.3) [POST](https://httpwg.org/specs/rfc9110.html#POST)

The POST method requests that the [target resource](https://httpwg.org/specs/rfc9110.html#target.resource) process the representation enclosed in the request according to the resource's own specific semantics. For example, POST is used for the following functions (among others):

* Providing a block of data, such as the fields entered into an HTML form, to a data-handling process;
* Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles;
* Creating a new resource that has yet to be identified by the origin server; and
* Appending data to a resource's existing representation(s).

An origin server indicates response semantics by choosing an appropriate status code depending on the result of processing the POST request; almost all of the status codes defined by this specification could be received in a response to POST (the exceptions being [206 (Partial Content)](https://httpwg.org/specs/rfc9110.html#status.206), [304 (Not Modified)](https://httpwg.org/specs/rfc9110.html#status.304), and [416 (Range Not Satisfiable)](https://httpwg.org/specs/rfc9110.html#status.416)).

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server SHOULD send a [201 (Created)](https://httpwg.org/specs/rfc9110.html#status.201) response containing a [Location](https://httpwg.org/specs/rfc9110.html#field.location) header field that provides an identifier for the primary resource created ([Section 10.2.2](https://httpwg.org/specs/rfc9110.html#field.location)) and a representation that describes the status of the request while referring to the new resource(s).

Responses to POST requests are only cacheable when they include explicit freshness information (see [Section 4.2.1](https://httpwg.org/specs/rfc9111.html#calculating.freshness.lifetime) of [[CACHING]](https://httpwg.org/specs/rfc9110.html#CACHING)) and a [Content-Location](https://httpwg.org/specs/rfc9110.html#field.content-location) header field that has the same value as the POST's target URI ([Section 8.7](https://httpwg.org/specs/rfc9110.html#field.content-location)). A cached POST response can be reused to satisfy a later GET or HEAD request. In contrast, a POST request cannot be satisfied by a cached POST response because POST is potentially unsafe; see [Section 4](https://httpwg.org/specs/rfc9111.html#constructing.responses.from.caches) of [[CACHING]](https://httpwg.org/specs/rfc9110.html#CACHING).

If the result of processing a POST would be equivalent to a representation of an existing resource, an origin server MAY redirect the user agent to that resource by sending a [303 (See Other)](https://httpwg.org/specs/rfc9110.html#status.303) response with the existing resource's identifier in the [Location](https://httpwg.org/specs/rfc9110.html#field.location) field. This has the benefits of providing the user agent a resource identifier and transferring the representation via a method more amenable to shared caching, though at the cost of an extra request if the user agent does not already have the representation cached.

#### [9.3.4.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.4) [PUT](https://httpwg.org/specs/rfc9110.html#PUT)

The PUT method requests that the state of the [target resource](https://httpwg.org/specs/rfc9110.html#target.resource) be created or replaced with the state defined by the representation enclosed in the request message content. A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent in a [200 (OK)](https://httpwg.org/specs/rfc9110.html#status.200) response. However, there is no guarantee that such a state change will be observable, since the target resource might be acted upon by other user agents in parallel, or might be subject to dynamic processing by the origin server, before any subsequent GET is received. A successful response only implies that the user agent's intent was achieved at the time of its processing by the origin server.

If the target resource does not have a current representation and the PUT successfully creates one, then the origin server MUST inform the user agent by sending a [201 (Created)](https://httpwg.org/specs/rfc9110.html#status.201) response. If the target resource does have a current representation and that representation is successfully modified in accordance with the state of the enclosed representation, then the origin server MUST send either a [200 (OK)](https://httpwg.org/specs/rfc9110.html#status.200) or a [204 (No Content)](https://httpwg.org/specs/rfc9110.html#status.204) response to indicate successful completion of the request.

An origin server SHOULD verify that the PUT representation is consistent with its configured constraints for the target resource. For example, if an origin server determines a resource's representation metadata based on the URI, then the origin server needs to ensure that the content received in a successful PUT request is consistent with that metadata. When a PUT representation is inconsistent with the target resource, the origin server SHOULD either make them consistent, by transforming the representation or changing the resource configuration, or respond with an appropriate error message containing sufficient information to explain why the representation is unsuitable. The [409 (Conflict)](https://httpwg.org/specs/rfc9110.html#status.409) or [415 (Unsupported Media Type)](https://httpwg.org/specs/rfc9110.html#status.415) status codes are suggested, with the latter being specific to constraints on [Content-Type](https://httpwg.org/specs/rfc9110.html#field.content-type) values.

For example, if the target resource is configured to always have a [Content-Type](https://httpwg.org/specs/rfc9110.html#field.content-type) of "text/html" and the representation being PUT has a Content-Type of "image/jpeg", the origin server ought to do one of:

1. reconfigure the target resource to reflect the new media type;
2. transform the PUT representation to a format consistent with that of the resource before saving it as the new resource state; or,
3. reject the request with a [415 (Unsupported Media Type)](https://httpwg.org/specs/rfc9110.html#status.415) response indicating that the target resource is limited to "text/html", perhaps including a link to a different resource that would be a suitable target for the new representation.

HTTP does not define exactly how a PUT method affects the state of an origin server beyond what can be expressed by the intent of the user agent request and the semantics of the origin server response. It does not define what a resource might be, in any sense of that word, beyond the interface provided via HTTP. It does not define how resource state is "stored", nor how such storage might change as a result of a change in resource state, nor how the origin server translates resource state into representations. Generally speaking, all implementation details behind the resource interface are intentionally hidden by the server.

This extends to how header and trailer fields are stored; while common header fields like [Content-Type](https://httpwg.org/specs/rfc9110.html#field.content-type) will typically be stored and returned upon subsequent GET requests, header and trailer field handling is specific to the resource that received the request. As a result, an origin server SHOULD ignore unrecognized header and trailer fields received in a PUT request (i.e., not save them as part of the resource state).

An origin server MUST NOT send a validator field ([Section 8.8](https://httpwg.org/specs/rfc9110.html#response.validator)), such as an [ETag](https://httpwg.org/specs/rfc9110.html" \l "field.etag) or [Last-Modified](https://httpwg.org/specs/rfc9110.html#field.last-modified) field, in a successful response to PUT unless the request's representation data was saved without any transformation applied to the content (i.e., the resource's new representation data is identical to the content received in the PUT request) and the validator field value reflects the new representation. This requirement allows a user agent to know when the representation it sent (and retains in memory) is the result of the PUT, and thus it doesn't need to be retrieved again from the origin server. The new validator(s) received in the response can be used for future conditional requests in order to prevent accidental overwrites ([Section 13.1](https://httpwg.org/specs/rfc9110.html#preconditions)).

The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation. The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource. Hence, the intent of PUT is idempotent and visible to intermediaries, even though the exact effect is only known by the origin server.

Proper interpretation of a PUT request presumes that the user agent knows which target resource is desired. A service that selects a proper URI on behalf of the client, after receiving a state-changing request, SHOULD be implemented using the POST method rather than PUT. If the origin server will not make the requested PUT state change to the target resource and instead wishes to have it applied to a different resource, such as when the resource has been moved to a different URI, then the origin server MUST send an appropriate [3xx (Redirection)](https://httpwg.org/specs/rfc9110.html#status.3xx) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

A PUT request applied to the target resource can have side effects on other resources. For example, an article might have a URI for identifying "the current version" (a resource) that is separate from the URIs identifying each particular version (different resources that at one point shared the same state as the current version resource). A successful PUT request on "the current version" URI might therefore create a new version resource in addition to changing the state of the target resource, and might also cause links to be added between the related resources.

Some origin servers support use of the [Content-Range](https://httpwg.org/specs/rfc9110.html#field.content-range) header field ([Section 14.4](https://httpwg.org/specs/rfc9110.html#field.content-range)) as a request modifier to perform a partial PUT, as described in [Section 14.5](https://httpwg.org/specs/rfc9110.html#partial.PUT).

Responses to the PUT method are not cacheable. If a successful PUT request passes through a cache that has one or more stored responses for the target URI, those stored responses will be invalidated (see [Section 4.4](https://httpwg.org/specs/rfc9111.html#invalidation) of [[CACHING]](https://httpwg.org/specs/rfc9110.html#CACHING)).

#### [9.3.5.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.5) [DELETE](https://httpwg.org/specs/rfc9110.html#DELETE)

The DELETE method requests that the origin server remove the association between the [target resource](https://httpwg.org/specs/rfc9110.html#target.resource) and its current functionality. In effect, this method is similar to the "rm" command in UNIX: it expresses a deletion operation on the URI mapping of the origin server rather than an expectation that the previously associated information be deleted.

If the target resource has one or more current representations, they might or might not be destroyed by the origin server, and the associated storage might or might not be reclaimed, depending entirely on the nature of the resource and its implementation by the origin server (which are beyond the scope of this specification). Likewise, other implementation aspects of a resource might need to be deactivated or archived as a result of a DELETE, such as database or gateway connections. In general, it is assumed that the origin server will only allow DELETE on resources for which it has a prescribed mechanism for accomplishing the deletion.

Relatively few resources allow the DELETE method — its primary use is for remote authoring environments, where the user has some direction regarding its effect. For example, a resource that was previously created using a PUT request, or identified via the Location header field after a [201 (Created)](https://httpwg.org/specs/rfc9110.html#status.201) response to a POST request, might allow a corresponding DELETE request to undo those actions. Similarly, custom user agent implementations that implement an authoring function, such as revision control clients using HTTP for remote operations, might use DELETE based on an assumption that the server's URI space has been crafted to correspond to a version repository.

If a DELETE method is successfully applied, the origin server SHOULD send

* a [202 (Accepted)](https://httpwg.org/specs/rfc9110.html#status.202) status code if the action will likely succeed but has not yet been enacted,
* a [204 (No Content)](https://httpwg.org/specs/rfc9110.html#status.204) status code if the action has been enacted and no further information is to be supplied, or
* a [200 (OK)](https://httpwg.org/specs/rfc9110.html#status.200) status code if the action has been enacted and the response message includes a representation describing the status.

Although request message framing is independent of the method used, content received in a DELETE request has no generally defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling attack ([Section 11.2](https://httpwg.org/specs/rfc9112.html#request.smuggling) of [[HTTP/1.1]](https://httpwg.org/specs/rfc9110.html#HTTP11)). A client SHOULD NOT generate content in a DELETE request unless it is made directly to an origin server that has previously indicated, in or out of band, that such a request has a purpose and will be adequately supported. An origin server SHOULD NOT rely on private agreements to receive content, since participants in HTTP communication are often unaware of intermediaries along the request chain.

Responses to the DELETE method are not cacheable. If a successful DELETE request passes through a cache that has one or more stored responses for the target URI, those stored responses will be invalidated (see [Section 4.4](https://httpwg.org/specs/rfc9111.html#invalidation) of [[CACHING]](https://httpwg.org/specs/rfc9110.html#CACHING)).

#### [9.3.6.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.6) [CONNECT](https://httpwg.org/specs/rfc9110.html#CONNECT)

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request target and, if successful, thereafter restrict its behavior to blind forwarding of data, in both directions, until the tunnel is closed. Tunnels are commonly used to create an end-to-end virtual connection, through one or more proxies, which can then be secured using TLS (Transport Layer Security, [[TLS13]](https://httpwg.org/specs/rfc9110.html#TLS13)).

CONNECT uses a special form of request target, unique to this method, consisting of only the host and port number of the tunnel destination, separated by a colon. There is no default port; a client MUST send the port number even if the CONNECT request is based on a URI reference that contains an authority component with an elided port ([Section 4.1](https://httpwg.org/specs/rfc9110.html#uri.references)). For example,

CONNECT server.example.com:80 HTTP/1.1

Host: server.example.com

A server MUST reject a CONNECT request that targets an empty or invalid port number, typically by responding with a 400 (Bad Request) status code.

Because CONNECT changes the request/response nature of an HTTP connection, specific HTTP versions might have different ways of mapping its semantics into the protocol's wire format.

CONNECT is intended for use in requests to a proxy. The recipient can establish a tunnel either by directly connecting to the server identified by the request target or, if configured to use another proxy, by forwarding the CONNECT request to the next inbound proxy. An origin server MAY accept a CONNECT request, but most origin servers do not implement CONNECT.

Any [2xx (Successful)](https://httpwg.org/specs/rfc9110.html#status.2xx) response indicates that the sender (and all inbound proxies) will switch to tunnel mode immediately after the response header section; data received after that header section is from the server identified by the request target. Any response other than a successful response indicates that the tunnel has not yet been formed.

A tunnel is closed when a tunnel intermediary detects that either side has closed its connection: the intermediary MUST attempt to send any outstanding data that came from the closed side to the other side, close both connections, and then discard any remaining data left undelivered.

Proxy authentication might be used to establish the authority to create a tunnel. For example,

CONNECT server.example.com:443 HTTP/1.1

Host: server.example.com:443

Proxy-Authorization: basic aGVsbG86d29ybGQ=

There are significant risks in establishing a tunnel to arbitrary servers, particularly when the destination is a well-known or reserved TCP port that is not intended for Web traffic. For example, a CONNECT to "example.com:25" would suggest that the proxy connect to the reserved port for SMTP traffic; if allowed, that could trick the proxy into relaying spam email. Proxies that support CONNECT SHOULD restrict its use to a limited set of known ports or a configurable list of safe request targets.

A server MUST NOT send any [Transfer-Encoding](https://httpwg.org/specs/rfc9112.html#field.transfer-encoding) or [Content-Length](https://httpwg.org/specs/rfc9110.html#field.content-length) header fields in a [2xx (Successful)](https://httpwg.org/specs/rfc9110.html#status.2xx) response to CONNECT. A client MUST ignore any Content-Length or Transfer-Encoding header fields received in a successful response to CONNECT.

A CONNECT request message does not have content. The interpretation of data sent after the header section of the CONNECT request message is specific to the version of HTTP in use.

Responses to the CONNECT method are not cacheable.

#### [9.3.7.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.7) [OPTIONS](https://httpwg.org/specs/rfc9110.html#OPTIONS)

The OPTIONS method requests information about the communication options available for the target resource, at either the origin server or an intervening intermediary. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action.

An OPTIONS request with an asterisk ("\*") as the request target ([Section 7.1](https://httpwg.org/specs/rfc9110.html#target.resource)) applies to the server in general rather than to a specific resource. Since a server's communication options typically depend on the resource, the "\*" request is only useful as a "ping" or "no-op" type of method; it does nothing beyond allowing the client to test the capabilities of the server. For example, this can be used to test a proxy for HTTP/1.1 conformance (or lack thereof).

If the request target is not an asterisk, the OPTIONS request applies to the options that are available when communicating with the target resource.

A server generating a successful response to OPTIONS SHOULD send any header that might indicate optional features implemented by the server and applicable to the target resource (e.g., [Allow](https://httpwg.org/specs/rfc9110.html#field.allow)), including potential extensions not defined by this specification. The response content, if any, might also describe the communication options in a machine or human-readable representation. A standard format for such a representation is not defined by this specification, but might be defined by future extensions to HTTP.

A client MAY send a [Max-Forwards](https://httpwg.org/specs/rfc9110.html#field.max-forwards) header field in an OPTIONS request to target a specific recipient in the request chain (see [Section 7.6.2](https://httpwg.org/specs/rfc9110.html#field.max-forwards)). A proxy MUST NOT generate a Max-Forwards header field while forwarding a request unless that request was received with a Max-Forwards field.

A client that generates an OPTIONS request containing content MUST send a valid [Content-Type](https://httpwg.org/specs/rfc9110.html#field.content-type) header field describing the representation media type. Note that this specification does not define any use for such content.

Responses to the OPTIONS method are not cacheable.

#### [9.3.8.](https://httpwg.org/specs/rfc9110.html#rfc.section.9.3.8) [TRACE](https://httpwg.org/specs/rfc9110.html#TRACE)

The TRACE method requests a remote, application-level loop-back of the request message. The final recipient of the request SHOULD reflect the message received, excluding some fields described below, back to the client as the content of a [200 (OK)](https://httpwg.org/specs/rfc9110.html#status.200) response. The "message/http" format ([Section 10.1](https://httpwg.org/specs/rfc9112.html#media.type.message.http) of [[HTTP/1.1]](https://httpwg.org/specs/rfc9110.html#HTTP11)) is one way to do so. The final recipient is either the origin server or the first server to receive a [Max-Forwards](https://httpwg.org/specs/rfc9110.html#field.max-forwards) value of zero (0) in the request ([Section 7.6.2](https://httpwg.org/specs/rfc9110.html#field.max-forwards)).

A client MUST NOT generate fields in a TRACE request containing sensitive data that might be disclosed by the response. For example, it would be foolish for a user agent to send stored user credentials ([Section 11](https://httpwg.org/specs/rfc9110.html#authentication)) or cookies [[COOKIE]](https://httpwg.org/specs/rfc9110.html#COOKIE) in a TRACE request. The final recipient of the request SHOULD exclude any request fields that are likely to contain sensitive data when that recipient generates the response content.

TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. The value of the [Via](https://httpwg.org/specs/rfc9110.html#field.via) header field ([Section 7.6.3](https://httpwg.org/specs/rfc9110.html#field.via)) is of particular interest, since it acts as a trace of the request chain. Use of the [Max-Forwards](https://httpwg.org/specs/rfc9110.html#field.max-forwards) header field allows the client to limit the length of the request chain, which is useful for testing a chain of proxies forwarding messages in an infinite loop.

A client MUST NOT send content in a TRACE request.

Responses to the TRACE method are not cacheable.

## [10.](https://httpwg.org/specs/rfc9110.html#rfc.section.10) [Message Context](https://httpwg.org/specs/rfc9110.html#context)

### [10.1.](https://httpwg.org/specs/rfc9110.html#rfc.section.10.1) [Request Context Fields](https://httpwg.org/specs/rfc9110.html#request.context)

The request header fields below provide additional information about the request context, including information about the user, user agent, and resource behind the request.

#### [10.1.1.](https://httpwg.org/specs/rfc9110.html#rfc.section.10.1.1) [Expect](https://httpwg.org/specs/rfc9110.html#field.expect)

The "Expect" header field in a request indicates a certain set of behaviors (expectations) that need to be supported by the server in order to properly handle this request.

[Expect](https://httpwg.org/specs/rfc9110.html#field.expect) = #[expectation](https://httpwg.org/specs/rfc9110.html#field.expect)

[expectation](https://httpwg.org/specs/rfc9110.html#field.expect) = [token](https://httpwg.org/specs/rfc9110.html#rule.token.separators) [ "=" ( [token](https://httpwg.org/specs/rfc9110.html#rule.token.separators) / [quoted-string](https://httpwg.org/specs/rfc9110.html#rule.quoted-string) ) [parameters](https://httpwg.org/specs/rfc9110.html#rule.parameter) ]

The Expect field value is case-insensitive.

The only expectation defined by this specification is "100-continue" (with no defined parameters).

A server that receives an Expect field value containing a member other than [100-continue](https://httpwg.org/specs/rfc9110.html#field.expect) MAY respond with a [417 (Expectation Failed)](https://httpwg.org/specs/rfc9110.html#status.417) status code to indicate that the unexpected expectation cannot be met.

A 100-continue expectation informs recipients that the client is about to send (presumably large) content in this request and wishes to receive a [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) interim response if the method, target URI, and header fields are not sufficient to cause an immediate success, redirect, or error response. This allows the client to wait for an indication that it is worthwhile to send the content before actually doing so, which can improve efficiency when the data is huge or when the client anticipates that an error is likely (e.g., when sending a state-changing method, for the first time, without previously verified authentication credentials).

For example, a request that begins with

PUT /somewhere/fun HTTP/1.1

Host: origin.example.com

Content-Type: video/h264

Content-Length: 1234567890987

Expect: 100-continue

allows the origin server to immediately respond with an error message, such as [401 (Unauthorized)](https://httpwg.org/specs/rfc9110.html#status.401) or [405 (Method Not Allowed)](https://httpwg.org/specs/rfc9110.html#status.405), before the client starts filling the pipes with an unnecessary data transfer.

Requirements for clients:

* A client MUST NOT generate a 100-continue expectation in a request that does not include content.
* A client that will wait for a [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) response before sending the request content MUST send an [Expect](https://httpwg.org/specs/rfc9110.html#field.expect) header field containing a 100-continue expectation.
* A client that sends a 100-continue expectation is not required to wait for any specific length of time; such a client MAY proceed to send the content even if it has not yet received a response. Furthermore, since [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) responses cannot be sent through an HTTP/1.0 intermediary, such a client SHOULD NOT wait for an indefinite period before sending the content.
* A client that receives a [417 (Expectation Failed)](https://httpwg.org/specs/rfc9110.html#status.417) status code in response to a request containing a 100-continue expectation SHOULD repeat that request without a 100-continue expectation, since the 417 response merely indicates that the response chain does not support expectations (e.g., it passes through an HTTP/1.0 server).

Requirements for servers:

* A server that receives a 100-continue expectation in an HTTP/1.0 request MUST ignore that expectation.
* A server MAY omit sending a [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) response if it has already received some or all of the content for the corresponding request, or if the framing indicates that there is no content.
* A server that sends a [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) response MUST ultimately send a final status code, once it receives and processes the request content, unless the connection is closed prematurely.
* A server that responds with a final status code before reading the entire request content SHOULD indicate whether it intends to close the connection (e.g., see [Section 9.6](https://httpwg.org/specs/rfc9112.html#persistent.tear-down) of [[HTTP/1.1]](https://httpwg.org/specs/rfc9110.html#HTTP11)) or continue reading the request content.

Upon receiving an HTTP/1.1 (or later) request that has a method, target URI, and complete header section that contains a 100-continue expectation and an indication that request content will follow, an origin server MUST send either:

* an immediate response with a final status code, if that status can be determined by examining just the method, target URI, and header fields, or
* an immediate [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) response to encourage the client to send the request content.

The origin server MUST NOT wait for the content before sending the [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) response.

Upon receiving an HTTP/1.1 (or later) request that has a method, target URI, and complete header section that contains a 100-continue expectation and indicates a request content will follow, a proxy MUST either:

* send an immediate response with a final status code, if that status can be determined by examining just the method, target URI, and header fields, or
* forward the request toward the origin server by sending a corresponding request-line and header section to the next inbound server.

If the proxy believes (from configuration or past interaction) that the next inbound server only supports HTTP/1.0, the proxy MAY generate an immediate [100 (Continue)](https://httpwg.org/specs/rfc9110.html#status.100) response to encourage the client to begin sending the content.

#### [10.1.2.](https://httpwg.org/specs/rfc9110.html#rfc.section.10.1.2) [From](https://httpwg.org/specs/rfc9110.html#field.from)

The "From" header field contains an Internet email address for a human user who controls the requesting user agent. The address ought to be machine-usable, as defined by "mailbox" in [Section 3.4](https://www.rfc-editor.org/rfc/rfc5322.html#section-3.4) of [[RFC5322]](https://httpwg.org/specs/rfc9110.html#RFC5322):

[From](https://httpwg.org/specs/rfc9110.html#field.from) = [mailbox](https://httpwg.org/specs/rfc9110.html#field.from)

[mailbox](https://httpwg.org/specs/rfc9110.html#field.from) = <mailbox, see [*[RFC5322]*](https://httpwg.org/specs/rfc9110.html#RFC5322), [Section 3.4](https://www.rfc-editor.org/rfc/rfc5322.html#section-3.4)>

An example is:

From: spider-admin@example.org

The From header field is rarely sent by non-robotic user agents. A user agent SHOULD NOT send a From header field without explicit configuration by the user, since that might conflict with the user's privacy interests or their site's security policy.

A robotic user agent SHOULD send a valid From header field so that the person responsible for running the robot can be contacted if problems occur on servers, such as if the robot is sending excessive, unwanted, or invalid requests.

A server SHOULD NOT use the From header field for access control or authentication, since its value is expected to be visible to anyone receiving or observing the request and is often recorded within logfiles and error reports without any expectation of privacy.

#### [10.1.3.](https://httpwg.org/specs/rfc9110.html#rfc.section.10.1.3) [Referer](https://httpwg.org/specs/rfc9110.html" \l "field.referer)

The "Referer" [sic] header field allows the user agent to specify a URI reference for the resource from which the [target URI](https://httpwg.org/specs/rfc9110.html#target.resource) was obtained (i.e., the "referrer", though the field name is misspelled). A user agent MUST NOT include the fragment and userinfo components of the URI reference [[URI]](https://httpwg.org/specs/rfc9110.html#URI), if any, when generating the Referer field value.

[Referer](https://httpwg.org/specs/rfc9110.html#field.referer) = [absolute-URI](https://httpwg.org/specs/rfc9110.html#uri.references) / [partial-URI](https://httpwg.org/specs/rfc9110.html#uri.references)

The field value is either an [absolute-URI](https://httpwg.org/specs/rfc9110.html#uri.references) or a [partial-URI](https://httpwg.org/specs/rfc9110.html#uri.references). In the latter case ([Section 4](https://httpwg.org/specs/rfc9110.html#uri)), the referenced URI is relative to the target URI ([[URI]](https://httpwg.org/specs/rfc9110.html#URI), [Section 5](https://www.rfc-editor.org/rfc/rfc3986.html#section-5)).

The Referer header field allows servers to generate back-links to other resources for simple analytics, logging, optimized caching, etc. It also allows obsolete or mistyped links to be found for maintenance. Some servers use the Referer header field as a means of denying links from other sites (so-called "deep linking") or restricting cross-site request forgery (CSRF), but not all requests contain it.

Example:

Referer: http://www.example.org/hypertext/Overview.html

If the target URI was obtained from a source that does not have its own URI (e.g., input from the user keyboard, or an entry within the user's bookmarks/favorites), the user agent MUST either exclude the Referer header field or send it with a value of "about:blank".

The Referer header field value need not convey the full URI of the referring resource; a user agent MAY truncate parts other than the referring origin.

The Referer header field has the potential to reveal information about the request context or browsing history of the user, which is a privacy concern if the referring resource's identifier reveals personal information (such as an account name) or a resource that is supposed to be confidential (such as behind a firewall or internal to a secured service). Most general-purpose user agents do not send the Referer header field when the referring resource is a local "file" or "data" URI. A user agent SHOULD NOT send a [Referer](https://httpwg.org/specs/rfc9110.html" \l "field.referer) header field if the referring resource was accessed with a secure protocol and the request target has an origin differing from that of the referring resource, unless the referring resource explicitly allows Referer to be sent. A user agent MUST NOT send a [Referer](https://httpwg.org/specs/rfc9110.html" \l "field.referer) header field in an unsecured HTTP request if the referring resource was accessed with a secure protocol. See [Section 17.9](https://httpwg.org/specs/rfc9110.html#sensitive.information.in.uris) for additional security considerations.

Some intermediaries have been known to indiscriminately remove Referer header fields from outgoing requests. This has the unfortunate side effect of interfering with protection against CSRF attacks, which can be far more harmful to their users. Intermediaries and user agent extensions that wish to limit information disclosure in Referer ought to restrict their changes to specific edits, such as replacing internal domain names with pseudonyms or truncating the query and/or path components. An intermediary SHOULD NOT modify or delete the Referer header field when the field value shares the same scheme and host as the target URI.