Introduction to RESTful Web services

REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model. In fact, REST has had such a large impact on the Web that it has mostly displaced SOAP- and WSDL-based interface design because it's a considerably simpler style to use.

REST didn't attract this much attention when it was first introduced in 2000 by Roy Fielding at the University of California, Irvine, in his academic dissertation, "Architectural Styles and the Design of Network-based Software Architectures," which analyzes a set of software architecture principles that use the Web as a platform for distributed computing. Now, years after its introduction, major frameworks for REST have started to appear and are still being developed because it's slated, for example, to become an integral part of Java™ 6 through JSR-311.

This article suggests that in its purest form today, when it's attracting this much attention, a concrete implementation of a REST Web service follows four basic design principles:

* Use HTTP methods explicitly.
* Be stateless.
* Expose directory structure-like URIs.
* Transfer XML, JavaScript Object Notation (JSON), or both.

The following sections expand on these four principles and propose a technical rationale for why they might be important for REST Web service designers.

Use HTTP methods explicitly

One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616. HTTP GET, for instance, is defined as a data-producing method that's intended to be used by a client application to retrieve a resource, to fetch data from a Web server, or to execute a query with the expectation that the Web server will look for and respond with a set of matching resources.

REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:

* To create a resource on the server, use POST.
* To retrieve a resource, use GET.
* To change the state of a resource or to update it, use PUT.
* To remove or delete a resource, use DELETE.

An unfortunate design flaw inherent in many Web APIs is in the use of HTTP methods for unintended purposes. The request URI in an HTTP GET request, for example, usually identifies one specific resource. Or the query string in a request URI includes a set of parameters that defines the search criteria used by the server to find a set of matching resources. At least this is how the HTTP/1.1 RFC describes GET. But there are many cases of unattractive Web APIs that use HTTP GET to trigger something transactional on the server—for instance, to add records to a database. In these cases the GET request URI is not used properly or at least not used RESTfully. If the Web API uses GET to invoke remote procedures, it looks like this:  
  
GET /adduser?name=Robert HTTP/1.1

It's not a very attractive design because the Web method above supports a state-changing operation over HTTP GET. Put another way, the HTTP GET request above has side effects. If successfully processed, the result of the request is to add a new user—in this example, Robert—to the underlying data store. The problem here is mainly semantic. Web servers are designed to respond to HTTP GET requests by retrieving resources that match the path (or the query criteria) in the request URI and return these or a representation in a response, not to add a record to a database. From the standpoint of the intended use of the protocol method then, and from the standpoint of HTTP/1.1-compliant Web servers, using GET in this way is inconsistent.

Beyond the semantics, the other problem with GET is that to trigger the deletion, modification, or addition of a record in a database, or to change server-side state in some way, it invites Web caching tools (crawlers) and search engines to make server-side changes unintentionally simply by crawling a link. A simple way to overcome this common problem is to move the parameter names and values on the request URI into XML tags. The resulting tags, an XML representation of the entity to create, may be sent in the body of an HTTP POST whose request URI is the intended parent of the entity (see Listings 1 and 2).

Listing 1. Before

GET /adduser?name=Robert HTTP/1.1

Show more

Listing 2. After

POST /users HTTP/1.1

Host: myserver

Content-Type: application/xml

<?xml version="1.0"?>

<user>

<name>Robert</name>

</user>

Show more

The method above is exemplary of a RESTful request: proper use of HTTP POST and inclusion of the payload in the body of the request. On the receiving end, the request may be processed by adding the resource contained in the body as a subordinate of the resource identified in the request URI; in this case the new resource should be added as a child of /users. This containment relationship between the new entity and its parent, as specified in the POST request, is analogous to the way a file is subordinate to its parent directory. The client sets up the relationship between the entity and its parent and defines the new entity's URI in the POST request.

A client application may then get a representation of the resource using the new URI, noting that at least logically the resource is located under /users, as shown in Listing 3.

Listing 3. HTTP GET request

GET /users/Robert HTTP/1.1

Host: myserver

Accept: application/xml

Show more

Using GET in this way is explicit because GET is for data retrieval only. GET is an operation that should be free of side effects, a property also known as *idempotence*.

A similar refactoring of a Web method also needs to be applied in cases where an update operation is supported over HTTP GET, as shown in Listing 4.

Listing 4. Update over HTTP GET

GET /updateuser?name=Robert&newname=Bob HTTP/1.1

This changes the name attribute (or property) of the resource. While the query string can be used for such

an operation, and Listing 4 is a simple one, this query-string-as-method-signature pattern tends to break down when used for more complex operations. Because your goal is to make explicit use of HTTP methods, a more RESTful approach is to send an HTTP PUT request to update the resource, instead of HTTP GET, for the same reasons stated above (see Listing 5).

Listing 5. HTTP PUT request

PUT /users/Robert HTTP/1.1

Host: myserver

Content-Type: application/xml

<?xml version="1.0"?>

<user>

<name>Bob</name>

</user>

Using PUT to replace the original resource provides a much cleaner interface that's consistent with REST's principles and with the definition of HTTP methods. The PUT request in Listing 5 is explicit in the sense that it points at the resource to be updated by identifying it in the request URI and in the sense that it transfers a new representation of the resource from client to server in the body of a PUT request instead of transferring the resource attributes as a loose set of parameter names and values on the request URI. Listing 5 also has the effect of renaming the resource from Robert to Bob, and in doing so changes its URI to /users/Bob. In a REST Web service, subsequent requests for the resource using the old URI would generate a standard 404 Not Found error.

As a general design principle, it helps to follow REST guidelines for using HTTP methods explicitly by using nouns in URIs instead of verbs. In a RESTful Web service, the verbs—POST, GET, PUT, and DELETE—are already defined by the protocol. And ideally, to keep the interface generalized and to allow clients to be explicit about the operations they invoke, the Web service should not define more verbs or remote procedures, such as /adduser or /updateuser. This general design principle also applies to the body of an HTTP request, which is intended to be used to transfer resource state, not to carry the name of a remote method or remote procedure to be invoked.

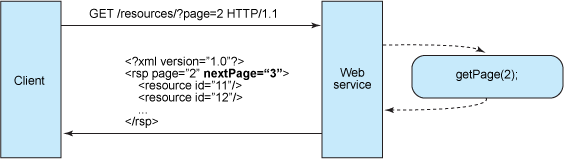
Be stateless

REST Web services need to scale to meet increasingly high performance demands. Clusters of servers with load-balancing and failover capabilities, proxies, and gateways are typically arranged in a way that forms a service topology, which allows requests to be forwarded from one server to the other as needed to decrease the overall response time of a Web service call. Using intermediary servers to improve scale requires REST Web service clients to send complete, independent requests; that is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests.

A complete, independent request doesn't require the server, while processing the request, to retrieve any kind of application context or state. A REST Web service application (or client) includes within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves Web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.

Figure 1 illustrates a stateful service from which an application may request the next page in a multipage result set, assuming that the service keeps track of where the application leaves off while navigating the set. In this stateful design, the service increments and stores a previousPage variable somewhere to be able to respond to requests for next.

Figure 1. Stateful design



Stateful services like this get complicated. In a Java Platform, Enterprise Edition (Java EE) environment stateful services require a lot of up-front consideration to efficiently store and enable the synchronization of session data across a cluster of Java EE containers. In this type of environment, there's a problem familiar to servlet/JavaServer Pages (JSP) and Enterprise JavaBeans (EJB) developers who often struggle to find the root causes of java.io.NotSerializableException during session replication. Whether it's thrown by the servlet container during HttpSession replication or thrown by the EJB container during stateful EJB replication, it's a problem that can cost developers days in trying to pinpoint the one object that doesn't implement Serializable in a sometimes complex graph of objects that constitute the server's state. In addition, session synchronization adds overhead, which impacts server performance.

Stateless server-side components, on the other hand, are less complicated to design, write, and distribute across load-balanced servers. A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application. In a RESTful Web service, the server is responsible for generating responses and for providing an interface that enables the client to maintain application state on its own. For example, in the request for a multipage result set, the client should include the actual page number to retrieve instead of simply asking for *next* (see Figure 2).

Figure 2. Stateless design

A stateless Web service generates a response that links to the next page number in the set and lets the client do what it needs to in order to keep this value around. This aspect of RESTful Web service design can be broken down into two sets of responsibilities as a high-level separation that clarifies just how a stateless service can be maintained:

Server

* Generates responses that include links to other resources to allow applications to navigate between related resources. This type of response embeds links. Similarly, if the request is for a parent or container resource, then a typical RESTful response might also include links to the parent's children or subordinate resources so that these remain connected.
* Generates responses that indicate whether they are cacheable or not to improve performance by reducing the number of requests for duplicate resources and by eliminating some requests entirely. The server does this by including a Cache-Control and Last-Modified (a date value) HTTP response header.

Client application

* Uses the Cache-Control response header to determine whether to cache the resource (make a local copy of it) or not. The client also reads the Last-Modified response header and sends back the date value in an If-Modified-Since header to ask the server if the resource has changed. This is called Conditional GET, and the two headers go hand in hand in that the server's response is a standard 304 code (Not Modified) and omits the actual resource requested if it has not changed since that time. A 304 HTTP response code means the client can safely use a cached, local copy of the resource representation as the most up-to-date, in effect bypassing subsequent GET requests until the resource changes.
* Sends complete requests that can be serviced independently of other requests. This requires the client to make full use of HTTP headers as specified by the Web service interface and to send complete representations of resources in the request body. The client sends requests that make very few assumptions about prior requests, the existence of a session on the server, the server's ability to add context to a request, or about application state that is kept in between requests.

This collaboration between client application and service is essential to being stateless in a RESTful Web service. It improves performance by saving bandwidth and minimizing server-side application state.

Expose directory structure-like URIs

From the standpoint of client applications addressing resources, the URIs determine how intuitive the REST Web service is going to be and whether the service is going to be used in ways that the designers can anticipate. A third RESTful Web service characteristic is all about the URIs.

REST Web service URIs should be intuitive to the point where they are easy to guess. Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood.

One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are subpaths that expose the service's main areas. According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and superordinate branches connected at nodes. For example, in a discussion threading service that gathers topics ranging from Java to paper, you might define a structured set of URIs like this:  
  
http://www.myservice.org/discussion/topics/{topic}

The root, /discussion, has a /topics node beneath it. Underneath that there are a series of topic names, such as gossip, technology, and so on, each of which points to a discussion thread. Within this structure, it's easy to pull up discussion threads just by typing something after /topics/.

In some cases, the path to a resource lends itself especially well to a directory-like structure. Take resources organized by date, for instance, which are a very good match for using a hierarchical syntax.

This example is intuitive because it is based on rules:  
  
http://www.myservice.org/discussion/2008/12/10/{topic}

The first path fragment is a four-digit year, the second path fragment is a two-digit day, and the third fragment is a two-digit month. It may seem a little silly to explain it that way, but this is the level of simplicity we're after. Humans and machines can easily generate structured URIs like this because they are based on rules. Filling in the path parts in the slots of a syntax makes them good because there is a definite pattern from which to compose them:  
  
http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}

Some additional guidelines to make note of while thinking about URI structure for a RESTful Web service are:

* Hide the server-side scripting technology file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URIs.
* Keep everything lowercase.
* Substitute spaces with hyphens or underscores (one or the other).
* Avoid query strings as much as you can.
* Instead of using the 404 Not Found code if the request URI is for a partial path, always provide a default page or resource as a response.

URIs should also be static so that when the resource changes or the implementation of the service changes, the link stays the same. This allows bookmarking. It's also important that the relationship between resources that's encoded in the URIs remains independent of the way the relationships are represented where they are stored.

Transfer XML, JSON, or both

A resource representation typically reflects the current state of a resource, and its attributes, at the time a client application requests it. Resource representations in this sense are mere snapshots in time. This could be a thing as simple as a representation of a record in a database that consists of a mapping between column names and XML tags, where the element values in the XML contain the row values. Or, if the system has a data model, then according to this definition a resource representation is a snapshot of the attributes of one of the things in your system's data model. These are the things you want your REST Web service to serve up.

The last set of constraints that goes into a RESTful Web service design has to do with the format of the data that the application and service exchange in the request/response payload or in the HTTP body. This is where it really pays to keep things simple, human-readable, and connected.

The objects in your data model are usually related in some way, and the relationships between data model objects (resources) should be reflected in the way they are represented for transfer to a client application. In the discussion threading service, an example of connected resource representations might include a root discussion topic and its attributes, and embed links to the responses given to that topic.

Listing 6. XML representation of a discussion thread

<?xml version="1.0"?>

<discussion date="{date}" topic="{topic}">

<comment>{comment}</comment>

<replies>

<reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>

<reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>

</replies>

</discussion>

Show more

And last, to give client applications the ability to request a specific content type that's best suited for them, construct your service so that it makes use of the built-in HTTP Accept header, where the value of the header is a MIME type. Some common MIME types used by RESTful services are shown in Table 1.

Table 1. Common MIME types used by RESTful services

| **MIME-Type** | **Content-Type** |
| --- | --- |
| **JSON** | application/json |
| **XML** | application/xml |
| **XHTML** | application/xhtml xml |

This allows the service to be used by a variety of clients written in different languages running on different platforms and devices. Using MIME types and the HTTP Accept header is a mechanism known as *content negotiation*, which lets clients choose which data format is right for them and minimizes data coupling between the service and the applications that use it.

# HTTP response status codes

HTTP response status codes indicate whether a specific [HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP) request has been successfully completed. Responses are grouped in five classes:

1. [Informational responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#information_responses) (100–199)
2. [Successful responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#successful_responses) (200–299)
3. [Redirection messages](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#redirection_messages) (300–399)
4. [Client error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses) (400–499)
5. [Server error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#server_error_responses) (500–599)

The below status codes are defined by [section 10 of RFC 2616](https://datatracker.ietf.org/doc/html/rfc2616#section-10). You can find an updated specification in [RFC 7231](https://datatracker.ietf.org/doc/html/rfc7231#section-6).

**Note:** If you receive a response that is not in [this list](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#information_responses), it is a non-standard response, possibly custom to the server's software.

## [Information responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#information_responses)

[100 Continue](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/100)

This interim response indicates that the client should continue the request or ignore the response if the request is already finished.

[101 Switching Protocols](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/101)

This code is sent in response to an [Upgrade](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Upgrade) request header from the client and indicates the protocol the server is switching to.

[102 Processing](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/102) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

This code indicates that the server has received and is processing the request, but no response is available yet.

[103 Early Hints](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/103)

This status code is primarily intended to be used with the [Link](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Link) header, letting the user agent start [preloading](https://developer.mozilla.org/en-US/docs/Web/HTML/Link_types/preload) resources while the server prepares a response.

## [Successful responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#successful_responses)

[200 OK](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200)

The request succeeded. The result meaning of "success" depends on the HTTP method:

* GET: The resource has been fetched and transmitted in the message body.
* HEAD: The representation headers are included in the response without any message body.
* PUT or POST: The resource describing the result of the action is transmitted in the message body.
* TRACE: The message body contains the request message as received by the server.

[201 Created](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/201)

The request succeeded, and a new resource was created as a result. This is typically the response sent after POST requests, or some PUT requests.

[202 Accepted](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/202)

The request has been received but not yet acted upon. It is noncommittal, since there is no way in HTTP to later send an asynchronous response indicating the outcome of the request. It is intended for cases where another process or server handles the request, or for batch processing.

[203 Non-Authoritative Information](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/203)

This response code means the returned metadata is not exactly the same as is available from the origin server, but is collected from a local or a third-party copy. This is mostly used for mirrors or backups of another resource. Except for that specific case, the 200 OK response is preferred to this status.

[204 No Content](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204)

There is no content to send for this request, but the headers may be useful. The user agent may update its cached headers for this resource with the new ones.

[205 Reset Content](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/205)

Tells the user agent to reset the document which sent this request.

[206 Partial Content](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/206)

This response code is used when the [Range](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Range) header is sent from the client to request only part of a resource.

[207 Multi-Status](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/207) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

Conveys information about multiple resources, for situations where multiple status codes might be appropriate.

[208 Already Reported](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/208) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

Used inside a <dav:propstat> response element to avoid repeatedly enumerating the internal members of multiple bindings to the same collection.

[226 IM Used](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/226) ([HTTP Delta encoding](https://datatracker.ietf.org/doc/html/rfc3229))

The server has fulfilled a GET request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance.

## [Redirection messages](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#redirection_messages)

[300 Multiple Choices](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/300)

The request has more than one possible response. The user agent or user should choose one of them. (There is no standardized way of choosing one of the responses, but HTML links to the possibilities are recommended so the user can pick.)

[301 Moved Permanently](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/301)

The URL of the requested resource has been changed permanently. The new URL is given in the response.

[302 Found](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302)

This response code means that the URI of requested resource has been changed temporarily. Further changes in the URI might be made in the future. Therefore, this same URI should be used by the client in future requests.

[303 See Other](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/303)

The server sent this response to direct the client to get the requested resource at another URI with a GET request.

[304 Not Modified](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/304)

This is used for caching purposes. It tells the client that the response has not been modified, so the client can continue to use the same cached version of the response.

305 Use Proxy Deprecated

Defined in a previous version of the HTTP specification to indicate that a requested response must be accessed by a proxy. It has been deprecated due to security concerns regarding in-band configuration of a proxy.

306 unused

This response code is no longer used; it is just reserved. It was used in a previous version of the HTTP/1.1 specification.

[307 Temporary Redirect](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/307)

The server sends this response to direct the client to get the requested resource at another URI with same method that was used in the prior request. This has the same semantics as the 302 Found HTTP response code, with the exception that the user agent must not change the HTTP method used: if a POST was used in the first request, a POST must be used in the second request.

[308 Permanent Redirect](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/308)

This means that the resource is now permanently located at another URI, specified by the Location: HTTP Response header. This has the same semantics as the 301 Moved Permanently HTTP response code, with the exception that the user agent must not change the HTTP method used: if a POST was used in the first request, a POST must be used in the second request.

## [Client error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses)

[400 Bad Request](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400)

The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

[401 Unauthorized](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/401)

Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.

[402 Payment Required](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/402) Experimental

This response code is reserved for future use. The initial aim for creating this code was using it for digital payment systems, however this status code is used very rarely and no standard convention exists.

[403 Forbidden](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403)

The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource. Unlike 401 Unauthorized, the client's identity is known to the server.

[404 Not Found](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404)

The server can not find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource itself does not exist. Servers may also send this response instead of 403 Forbidden to hide the existence of a resource from an unauthorized client. This response code is probably the most well known due to its frequent occurrence on the web.

[405 Method Not Allowed](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/405)

The request method is known by the server but is not supported by the target resource. For example, an API may not allow calling DELETE to remove a resource.

[406 Not Acceptable](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/406)

This response is sent when the web server, after performing [server-driven content negotiation](https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation#server-driven_negotiation), doesn't find any content that conforms to the criteria given by the user agent.

[407 Proxy Authentication Required](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/407)

This is similar to 401 Unauthorized but authentication is needed to be done by a proxy.

[408 Request Timeout](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/408)

This response is sent on an idle connection by some servers, even without any previous request by the client. It means that the server would like to shut down this unused connection. This response is used much more since some browsers, like Chrome, Firefox 27+, or IE9, use HTTP pre-connection mechanisms to speed up surfing. Also note that some servers merely shut down the connection without sending this message.

[409 Conflict](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409)

This response is sent when a request conflicts with the current state of the server.

[410 Gone](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/410)

This response is sent when the requested content has been permanently deleted from server, with no forwarding address. Clients are expected to remove their caches and links to the resource. The HTTP specification intends this status code to be used for "limited-time, promotional services". APIs should not feel compelled to indicate resources that have been deleted with this status code.

[411 Length Required](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/411)

Server rejected the request because the Content-Length header field is not defined and the server requires it.

[412 Precondition Failed](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/412)

The client has indicated preconditions in its headers which the server does not meet.

[413 Payload Too Large](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/413)

Request entity is larger than limits defined by server. The server might close the connection or return an Retry-After header field.

[414 URI Too Long](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/414)

The URI requested by the client is longer than the server is willing to interpret.

[415 Unsupported Media Type](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/415)

The media format of the requested data is not supported by the server, so the server is rejecting the request.

[416 Range Not Satisfiable](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/416)

The range specified by the Range header field in the request cannot be fulfilled. It's possible that the range is outside the size of the target URI's data.

[417 Expectation Failed](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/417)

This response code means the expectation indicated by the Expect request header field cannot be met by the server.

[418 I'm a teapot](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418)

The server refuses the attempt to brew coffee with a teapot.

[421 Misdirected Request](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/421)

The request was directed at a server that is not able to produce a response. This can be sent by a server that is not configured to produce responses for the combination of scheme and authority that are included in the request URI.

[422 Unprocessable Entity](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/422) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

The request was well-formed but was unable to be followed due to semantic errors.

[423 Locked](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/423) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

The resource that is being accessed is locked.

[424 Failed Dependency](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/424) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

The request failed due to failure of a previous request.

[425 Too Early](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/425) Experimental

Indicates that the server is unwilling to risk processing a request that might be replayed.

[426 Upgrade Required](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/426)

The server refuses to perform the request using the current protocol but might be willing to do so after the client upgrades to a different protocol. The server sends an [Upgrade](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Upgrade) header in a 426 response to indicate the required protocol(s).

[428 Precondition Required](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/428)

The origin server requires the request to be conditional. This response is intended to prevent the 'lost update' problem, where a client GETs a resource's state, modifies it and PUTs it back to the server, when meanwhile a third party has modified the state on the server, leading to a conflict.

[429 Too Many Requests](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429)

The user has sent too many requests in a given amount of time ("rate limiting").

[431 Request Header Fields Too Large](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/431)

The server is unwilling to process the request because its header fields are too large. The request may be resubmitted after reducing the size of the request header fields.

[451 Unavailable For Legal Reasons](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/451)

The user agent requested a resource that cannot legally be provided, such as a web page censored by a government.

## [Server error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#server_error_responses)

[500 Internal Server Error](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500)

The server has encountered a situation it does not know how to handle.

[501 Not Implemented](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/501)

The request method is not supported by the server and cannot be handled. The only methods that servers are required to support (and therefore that must not return this code) are GET and HEAD.

[502 Bad Gateway](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/502)

This error response means that the server, while working as a gateway to get a response needed to handle the request, got an invalid response.

[503 Service Unavailable](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503)

The server is not ready to handle the request. Common causes are a server that is down for maintenance or that is overloaded. Note that together with this response, a user-friendly page explaining the problem should be sent. This response should be used for temporary conditions and the Retry-After HTTP header should, if possible, contain the estimated time before the recovery of the service. The webmaster must also take care about the caching-related headers that are sent along with this response, as these temporary condition responses should usually not be cached.

[504 Gateway Timeout](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/504)

This error response is given when the server is acting as a gateway and cannot get a response in time.

[505 HTTP Version Not Supported](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/505)

The HTTP version used in the request is not supported by the server.

[506 Variant Also Negotiates](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/506)

The server has an internal configuration error: the chosen variant resource is configured to engage in transparent content negotiation itself, and is therefore not a proper end point in the negotiation process.

[507 Insufficient Storage](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/507) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

The method could not be performed on the resource because the server is unable to store the representation needed to successfully complete the request.

[508 Loop Detected](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/508) ([WebDAV](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

The server detected an infinite loop while processing the request.

[510 Not Extended](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/510)

Further extensions to the request are required for the server to fulfill it.

[511 Network Authentication Required](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/511)

Indicates that the client needs to authenticate to gain network access.