Internet Engineering Task Force (IETF)
Request for Comments: 5789
Category: Standards Track
ISSN: 2070-1721

L. Dusseault
Linden Lab
J. Snell

March 2010

PATCH Method for HTTP

Abstract

Several applications extending the Hypertext Transfer Protocol (HTTP) require a feature to do partial resource modification. The existing HTTP PUT method only allows a complete replacement of a document. This proposal adds a new HTTP method, PATCH, to modify an existing HTTP resource.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc5789.

Table of Contents

1. Introduction

This specification defines the new HTTP/1.1 [RFC2616] method, PATCH, which is used to apply partial modifications to a resource.

A new method is necessary to improve interoperability and prevent errors. The PUT method is already defined to overwrite a resource with a complete new body, and cannot be reused to do partial changes. Otherwise, proxies and caches, and even clients and servers, may get confused as to the result of the operation. POST is already used but without broad interoperability (for one, there is no standard way to discover patch format support). PATCH was mentioned in earlier HTTP specifications, but not completely defined.

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in [RFC2119].

Furthermore, this document uses the ABNF syntax defined in Section 2.1 of [RFC2616].

2. The PATCH Method

The PATCH method requests that a set of changes described in the request entity be applied to the resource identified by the Request-URI. The set of changes is represented in a format called a "patch document" identified by a media type. If the Request-URI does not point to an existing resource, the server MAY create a new resource, depending on the patch document type (whether it can logically modify a null resource) and permissions, etc.

The difference between the PUT and PATCH requests is reflected in the way the server processes the enclosed entity to modify the resource identified by the Request-URI. In a PUT request, the enclosed entity is considered to be a modified version of the resource stored on the origin server, and the client is requesting that the stored version be replaced. With PATCH, however, the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version. The PATCH method affects the resource identified by the Request-URI, and it also MAY have side effects on other resources; i.e., new resources may be created, or existing ones modified, by the application of a PATCH.

PATCH is neither safe nor idempotent as defined by [RFC2616], Section 9.1.

A PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application SHOULD use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag [RFC2616] in an If-Match header on the PATCH request.

There are also cases where patch formats do not need to operate from a known base-point (e.g., appending text lines to log files, or non-colliding rows to database tables), in which case the same care in client requests is not needed.

The server MUST apply the entire set of changes atomically and never provide (e.g., in response to a GET during this operation) a partially modified representation. If the entire patch document cannot be successfully applied, then the server MUST NOT apply any of the changes. The determination of what constitutes a successful PATCH can vary depending on the patch document and the type of resource(s) being modified. For example, the common 'diff' utility can generate a patch document that applies to multiple files in a directory hierarchy. The atomicity requirement holds for all directly affected files. See "Error Handling", Section 2.2, for details on status codes and possible error conditions.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. A response to this method is only cacheable if it

contains explicit freshness information (such as an Expires header or "Cache-Control: max-age" directive) as well as the Content-Location header matching the Request-URI, indicating that the PATCH response body is a resource representation. A cached PATCH response can only be used to respond to subsequent GET and HEAD requests; it MUST NOT be used to respond to other methods (in particular, PATCH).

Note that entity-headers contained in the request apply only to the contained patch document and MUST NOT be applied to the resource being modified. Thus, a Content-Language header could be present on the request, but it would only mean (for whatever that's worth) that the patch document had a language. Servers SHOULD NOT store such headers except as trace information, and SHOULD NOT use such header values the same way they might be used on PUT requests. Therefore, this document does not specify a way to modify a document's Content-Type or Content-Language value through headers, though a mechanism could well be designed to achieve this goal through a patch document.

There is no guarantee that a resource can be modified with PATCH. Further, it is expected that different patch document formats will be appropriate for different types of resources and that no single format will be appropriate for all types of resources. Therefore, there is no single default patch document format that implementations are required to support. Servers MUST ensure that a received patch document is appropriate for the type of resource identified by the Request-URI.

Clients need to choose when to use PATCH rather than PUT. For example, if the patch document size is larger than the size of the new resource data that would be used in a PUT, then it might make sense to use PUT instead of PATCH. A comparison to POST is even more difficult, because POST is used in widely varying ways and can encompass PUT and PATCH-like operations if the server chooses. If the operation does not modify the resource identified by the Request-URI in a predictable way, POST should be considered instead of PATCH or PUT.

2.1. A Simple PATCH Example

```
PATCH /file.txt HTTP/1.1
Host: www.example.com
Content-Type: application/example
If-Match: "e0023aa4e"
Content-Length: 100

[description of changes]
```

This example illustrates use of a hypothetical patch document on an existing resource.

Successful PATCH response to existing text file:

```
HTTP/1.1 204 No Content
Content-Location: /file.txt
ETag: "e0023aa4f"
```

The 204 response code is used because the response does not carry a message body (which a response with the 200 code would have). Note that other success codes could be used as well.

Furthermore, the ETag response header field contains the ETag for the entity created by applying the PATCH, available at http://www.example.com/file.txt, as indicated by the Content-Location response header field.

2.2. Error Handling

There are several known conditions under which a PATCH request can fail.

Malformed patch document: When the server determines that the patch document provided by the client is not properly formatted, it SHOULD return a 400 (Bad Request) response. The definition of badly formatted depends on the patch document chosen.

Unsupported patch document: Can be specified using a 415 (Unsupported Media Type) response when the client sends a patch document format that the server does not support for the resource identified by the Request-URI. Such a response SHOULD include an Accept-Patch response header as described in Section 3.1 to notify the client what patch document media types are supported.

Unprocessable request: Can be specified with a 422 (Unprocessable Entity) response ([RFC4918], Section 11.2) when the server understands the patch document and the syntax of the patch document appears to be valid, but the server is incapable of processing the request. This might include attempts to modify a resource in a way that would cause the resource to become invalid; for instance, a modification to a well-formed XML document that would cause it to no longer be well-formed. There may also be more specific errors like "Conflicting State" that could be signaled with this status code, but the more specific error would generally be more helpful.

Resource not found: Can be specified with a 404 (Not Found) status code when the client attempted to apply a patch document to a non-existent resource, but the patch document chosen cannot be applied to a non-existent resource.

Conflicting state: Can be specified with a 409 (Conflict) status code when the request cannot be applied given the state of the resource. For example, if the client attempted to apply a structural modification and the structures assumed to exist did not exist (with XML, a patch might specify changing element 'foo' to element 'bar' but element 'foo' might not exist).

Conflicting modification: When a client uses either the If-Match or If-Unmodified-Since header to define a precondition, and that precondition failed, then the 412 (Precondition Failed) error is most helpful to the client. However, that response makes no sense if there was no precondition on the request. In cases when the server detects a possible conflicting modification and no precondition was defined in the request, the server can return a 409 (Conflict) response.

Concurrent modification: Some applications of PATCH might require the server to process requests in the order in which they are received. If a server is operating under those restrictions, and it receives concurrent requests to modify the same resource, but is unable to queue those requests, the server can usefully indicate this error by using a 409 (Conflict) response.

Note that the 409 Conflict response gives reasonably consistent information to clients. Depending on the application and the nature of the patch format, the client might be able to reissue the request as is (e.g., an instruction to append a line to a log file), have to retrieve the resource content to recalculate a patch, or have to fail the operation.

Other HTTP status codes can also be used under the appropriate circumstances.

The entity body of error responses SHOULD contain enough information to communicate the nature of the error to the client. The content-type of the response entity can vary across implementations.

3. Advertising Support in OPTIONS

A server can advertise its support for the PATCH method by adding it to the listing of allowed methods in the "Allow" OPTIONS response header defined in HTTP/1.1. The PATCH method MAY appear in the "Allow" header even if the Accept-Patch header is absent, in which case the list of allowed patch documents is not advertised.

3.1. The Accept-Patch Header

This specification introduces a new response header Accept-Patch used to specify the patch document formats accepted by the server. Accept-Patch SHOULD appear in the OPTIONS response for any resource that supports the use of the PATCH method. The presence of the Accept-Patch header in response to any method is an implicit indication that PATCH is allowed on the resource identified by the Request-URI. The presence of a specific patch document format in this header indicates that that specific format is allowed on the resource identified by the Request-URI.

```
Accept-Patch = "Accept-Patch" ":" 1#media-type
```

The Accept-Patch header specifies a comma-separated listing of media-types (with optional parameters) as defined by [RFC2616], Section 3.7.

Example:

```
Accept-Patch: text/example;charset=utf-8
```

3.2. Example OPTIONS Request and Response

[request]

```
OPTIONS /example/buddies.xml HTTP/1.1
Host: www.example.com
```

[response]

```
HTTP/1.1 200 OK
Allow: GET, PUT, POST, OPTIONS, HEAD, DELETE, PATCH
Accept-Patch: application/example, text/example
```

The examples show a server that supports PATCH generally using two hypothetical patch document formats.

4. IANA Considerations

4.1. The Accept-Patch Response Header

The Accept-Patch response header has been added to the permanent registry (see [RFC3864]).

Header field name: Accept-Patch

Applicable Protocol: HTTP

Author/Change controller: IETF

Specification document: this specification

5. Security Considerations

The security considerations for PATCH are nearly identical to the security considerations for PUT ([RFC2616], Section 9.6). These include authorizing requests (possibly through access control and/or authentication) and ensuring that data is not corrupted through transport errors or through accidental overwrites. Whatever mechanisms are used for PUT can be used for PATCH as well. The following considerations apply especially to PATCH.

A document that is patched might be more likely to be corrupted than a document that is overridden in entirety, but that concern can be addressed through the use of mechanisms such as conditional requests using ETags and the If-Match request header as described in Section 2. If a PATCH request fails, the client can issue a GET request to the resource to see what state it is in. In some cases, the client might be able to check the contents of the resource to see if the PATCH request can be resent, but in other cases, the attempt will just fail and/or a user will have to verify intent. In the case of a failure of the underlying transport channel, where a PATCH response is not received before the channel fails or some other timeout happens, the client might have to issue a GET request to see whether the request was applied. The client might want to ensure that the GET request bypasses caches using mechanisms described in HTTP specifications (see, for example, Section 13.1.6 of [RFC2616]).

Sometimes an HTTP intermediary might try to detect viruses being sent via HTTP by checking the body of the PUT/POST request or GET response. The PATCH method complicates such watch-keeping because neither the source document nor the patch document might be a virus, yet the result could be. This security consideration is not

materially different from those already introduced by byte-range downloads, downloading patch documents, uploading zipped (compressed) files, and so on.

Individual patch documents will have their own specific security considerations that will likely vary depending on the types of resources being patched. The considerations for patched binary resources, for instance, will be different than those for patched XML documents. Servers MUST take adequate precautions to ensure that malicious clients cannot consume excessive server resources (e.g., CPU, disk I/O) through the client's use of PATCH.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

6.2. Informative References

[RFC4918] Dusseault, L., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, June 2007.

Appendix A. Acknowledgements

PATCH is not a new concept, it first appeared in HTTP in drafts of version 1.1 written by Roy Fielding and Henrik Frystyk and also appears in Section 19.6.1.1 of RFC 2068.

Thanks to Adam Roach, Chris Sharp, Julian Reschke, Geoff Clemm, Scott Lawrence, Jeffrey Mogul, Roy Fielding, Greg Stein, Jim Luther, Alex Rousskov, Jamie Lokier, Joe Hildebrand, Mark Nottingham, Michael Balloni, Cyrus Daboo, Brian Carpenter, John Klensin, Eliot Lear, SM, and Bernie Hoeneisen for review and advice on this document. In particular, Julian Reschke did repeated reviews, made many useful suggestions, and was critical to the publication of this document.

Authors' Addresses

Lisa Dusseault
Linden Lab
945 Battery Street
San Francisco, CA 94111
USA

EMail: lisa.dusseault@gmail.com

James M. Snell

EMail: jasnell@gmail.com
URI: http://www.snellspace.com