Workgroup: HTTP APIs
Internet-Draft:
draft-wright-http-patch-byterange-01
Published: 23 January 2023
Intended Status: Experimental
Expires: 27 July 2023
Authors: A. Wright

Byte Range PATCH

Abstract

This document specifies a media type for PATCH payloads that overwrites a specific byte range, to allow random access writes, or allow a resource to be uploaded in several segments.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 July 2023.

Table of Contents

1. Introduction

Filesystem interfaces typically provide some way to write at a specific position in a file. While HTTP supports reading byte range offsets using the Range header (Section 14 of [RFC9110]), this technique cannot generally be used in PUT, because the server may ignore the Content-Range header while executing the write, causing data corruption. However, by using a method and media type that the server must understand, writes to byte ranges with Content-Range semantics becomes possible.

This media type is intended for use in a wide variety of applications where overwriting specific parts of the file is desired. This includes idempotently writing data to a stream, appending data to a file, overwriting specific byte ranges, or writing to multiple regions in a single operation (for example, appending audio to a recording in progress while updating metadata at the beginning of the file).

It is particularly designed to recover from interrupted uploads. Since HTTP is stateless, clients can recover from an interrupted connection by making a request that completes the partial state change. For downloads, the Range header allows a client to download only the unknown data. However, if an upload is interrupted, no

mechanism exists to upload only the remaining data; the entire request must be retried.

Byte range patches may be used to "fill in these gaps."

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses ABNF as defined in [RFC5234] and imports grammar rules from [RFC9112].

For brevity, example HTTP requests or responses may add newlines or whitespace, or omit some headers necessary for message transfer.

The term "byte" is used in the [RFC9110] sense to mean "octet." Ranges are zero-indexed and inclusive. For example, "bytes 0-0" means the first byte of the document, and "bytes 1-2" is a range with two bytes, starting one byte into the document. Ranges of zero bytes are described by an address offset rather than a range. For example, "at byte 15".

2. Modifying a content range with PATCH

Although the Content-Range field cannot be used in the request headers without risking data corruption, it may be used in conjunction with the PATCH method [RFC5789] as part of a media type whose semantics writes a subset of a document, at a particular byte offset. This document re-uses the "multipart/byteranges" media type, and defines the "message/byterange" media type, for this purpose.

A byte range patch lists one or more *parts*. Each part specifies two essential components:

1. Part fields: a list of HTTP fields that specify metadata, including the range being written to, the length of the body, and information about the target document that cannot be listed in the PATCH headers, e.g. Content-Type (where it would describe the patch itself, rather than the document being updated).

2. A part body: the actual data to write to the indicated location.

Each part MUST indicate a single contiguous range to be written to. Servers MUST reject byte range patches that don't contain a known

range with a 422 or 400 error. (This would mean the client may be using a yet-undefined mechanism to specify the target range.)

The Content-Range field is used to specify the range to write to for each part:

The unsatisfied-range form (e.g. bytes */1000) is not meaningful, it MUST be treated as a syntax error.

The client MAY indicate the anticipated final size of the document by providing the complete-length form, for example bytes 0-11/12. This value does not affect the success of the write, however the server MAY use it for other purposes, especially for preallocating an optimal amount of space, and deciding when an upload in multiple parts has finished.

If the client does not know or care about the final length of the document, it MAY use * in place of complete-length. For example, bytes 0-11/*. Most random access writes will follow this form.

Other "Content-" fields in the patch document have the same meaning as if used in PUT request with the complete document (patch applied).

Servers SHOULD NOT accept requests that write beyond, and not adjacent to, the end of the resource. This would create a sparse file, where some bytes are undefined. For example, writing at byte 601 of a resource where bytes 0-599 are defined; this would leave byte 600 undefined. Servers that accept sparse writes MUST NOT disclose contents of existing storage.

The expected length of the write can be computed from the part fields. If the actual length of the part body mismatches the expected length, this MUST be treated the same as a network interruption at the shorter length, but expecting the longer length. This may involve rolling back the entire request, or saving as many bytes as possible. The client can then recover the same way it would recover from a network error.

2.1. The multipart/byteranges media type

The following is a request with a "multipart/byteranges" body to write two ranges in a document:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: multipart/byteranges; boundary=THIS_STRING_SEPARATES
Content-Length: 206
If-Match: "xyzzy"
If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

--THIS_STRING_SEPARATES
Content-Range: bytes 2-6/25
Content-Type: text/plain

23456
--THIS_STRING_SEPARATES
Content-Range: bytes 17-21/25
Content-Type: text/plain

78901
--THIS_STRING_SEPARATES--
```

The syntax for multipart messages is defined in [RFC2046], Section 5.1.1. While the body cannot contain the boundary, servers MAY use the Content-Length field to skip to the boundary (potentially ignoring a boundary in the body, which would be an error by the client).

The body of each range MUST be exactly as long as indicated by the Content-Range, unless indicated otherwise by a Content-Length header.

The multipart/byteranges type may be used for operations where multiple regions must be updated at the same time; clients may have an expectation that if there's an interruption, all of the parts will be rolled back.

2.2. The message/byterange media type

When making a request with a single byte range, there is no need for a multipart boundary marker. This document defines a new media type "message/byterange" with the same semantics as a single byte range in a multipart/byteranges message, but with a simplified syntax.

The "message/byterange" form may be used in a request as so:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 272
If-Match: "xyzzy"
If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

Content-Range: bytes 100-299/600
Content-Type: text/plain

[200 bytes...]
```

This represents a request to modify a 600-byte document, overwriting 200 bytes of it, starting at a 100-byte offset.

The syntax is defined in Section 4.1.

2.3. The message/byterange+bhttp media type

The "message/byterange+bhttp" has the same semantics as "message/byterange" but follows a binary format similar "message/bhttp" [RFC9292], and may be more suitable for some clients and servers, as all variable length strings are tagged with their length.

2.4. Appending

An append is when the range being replaced starts inside or at the end of the target document, but extends beyond the end, increasing the document length. For example, writing to bytes 10-19 of a 15 byte document, resulting in a 20 byte document.

Aside from the fact the document size increases, it may be handled the same as an overwrite (below).

2.5. Splicing

A splice is when the range being replaced is a different length than the replacement, shifting the position of the bytes that come after. For example, deleting bytes 0-9 in a 20 byte document, resulting in a 10 byte document. Or prepending 10 bytes to a 20 byte document, resulting in a 30 byte document.

This can be an expensive operation that servers are not expected to support. Servers that do not support splicing will emit an error to clients as they attempt to process the request as an overwrite operation (see below).

As a special case, the "Content-Range" field may omit the "-" and last-pos to indicate insertion that does not overwrite any bytes:

```
Content-Range =/ range-unit SP first-pos "/" ( complete-length / "*" )
```

Splicing operations MUST include a Content-Length field, to indicate the expected length of the part body.

Clients intending to perform a splice MUST include a Content-Length field, so a server can compute if a patch will be a splice or not before.

This whole section may need to be removed, and re-introduced using a new field, maybe called "Content-Range-Target" since "Content-Range" was never intended to do anything resembling splicing.

2.6. Overwriting

An overwrite only changes existing bytes, and so does not change the length of the document.

Overwriting operations MAY include a Content-Length field. If provided in overwriting operations, it MUST exactly match the length of the range specified in the Content-Range field. Servers that do not support splicing MUST error when the Content-Length mismatches the length of the range.

2.7. Range units

Currently, the only defined range unit is "bytes", however this may be other, yet-to-be-defined values.

In the case of "bytes", the bytes that are read are exactly the same as the bytes that are changed. However, other units may define write semantics different from a read, if symmetric behavior would not make sense. For example, if a Content-Range field adds an item in a JSON array, this write may add a leading or trailing comma, not technically part of the item itself, in order to keep the resulting document well-formed. Units that change the overall length of the document might always be classified as "splice" operations.

3. Segmented document creation with PATCH

As an alternative to using PUT to create a new resource, the contents of a resource may be uploaded in segments, written across several PATCH requests.

A user-agent may also use PATCH to recover from an interrupted PUT request, if it was expected to create a new resource. The server will store the data sent to it by the user agent, but will not finalize the upload until the final length of the document is known and received.

1. The client makes a PUT or PATCH request to a URL, a portion of which is generated by the client, to be unpredictable to other

clients. This first request creates the resource, and should include If-None-Match: * to verify the target does not exist. If a PUT request, the server reads the Content-Length header and stores the intended final length of the document. If a PATCH request, the "Content-Range" field in the "message/byterange" patch is read for the final length. The final length may also be undefined, and defined in a later request.

2. If any request is interrupted, the client may make a HEAD request to determine how much, if any, of the previous response was stored, and resumes uploading from that point. The server will return 200 (OK), but this may only indicate the write has been saved; the server is not obligated to begin acting on the upload until it is complete.

3. If the client sees from the HEAD response that additional data remains to be uploaded, it may make a PATCH request to resume uploading. Even if no data was uploaded or the resource was not created, the client should attempt creating the resource with PATCH to mitigate the possibility of another interrupted connection with a server that does not save incomplete transfers. However if in response to PATCH, the server reports 405 (Method Not Allowed), 415 (Unsupported Media Type), or 501 (Not Implemented), then the client must resort to a PUT request.

4. The server detects the completion of the final request when the current received data matches the indicated final length. For example, a Content-Range: 500-599/600 field is a write at the end of the resource. The server processes the upload and returns a response for it.

For building POST endpoints that support large uploads, clients can first upload the data to a scratch file as described above, and then process by submitting a POST request that links to the scratch file.

For updating an existing large file, the client can upload to a scratch file, then execute a MOVE (Section 9.9 of [RFC4918]) over the intended target.

3.1. Example

A single PUT request that creates a new resource may be split apart into multiple PATCH requests. Here is an example that uploads a 600-byte document across three 200-byte segments.

The first PATCH request creates the resource:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 281
If-None-Match: *

Content-Range: bytes 0-199/600
Content-Type: text/plain
Content-Length: 200

[200 bytes...]
```

This request allocates a 600 byte document, and uploading the first 200 bytes of it. The server responds with 200, indicating that the complete upload was stored.

Additional requests upload the remainder of the document:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 283
If-None-Match: *

Content-Range: bytes 200-399/600
Content-Type: text/plain
Content-Length: 200

[200 bytes...]
```

This second request also returns 200 (OK).

A third request uploads the final portion of the document:

```
PATCH /uploads/foo HTTP/1.1
Content-Type: message/byterange
Content-Length: 283
If-None-Match: *

Content-Range: bytes 200-399/600
Content-Type: text/plain
Content-Length: 200

[200 bytes...]
```

The server responds with 200 (OK). Since this completely writes out the 600-byte document, the server may also perform final processing, for example, checking that the document is well formed. The server MAY return an error code if there is a syntax or other error, or in an earlier response as soon as it it able to detect an error, however the exact behavior is left undefined.

4. Registrations

4.1. message/byterange media type

The "message/byterange" media type patches the defined byte range to some specified contents. It is similar to the "multipart/byteranges" media type, except it omits the multipart separator, and so only allows a single range to be specified.

It follows the syntax of HTTP message headers and body. It MUST include the Content-Range header field. If the message length is known by the sender, it SHOULD contain the Content-Length header field. Unknown or nonapplicable header fields MUST be ignored.

The field-line and message-body productions are specified in [RFC9112].

```
byterange-document = *( field-line CRLF )
                     CRLF
                     [ message-body ]
```

This document has the same semantics as a single part in a "multipart/byteranges" document (Section 5.1.1 of [RFC2046]) or any response with a 206 (Partial Content) status code (Section 15.3.7 of [RFC9110]). A "message/byterange" document may be trivially transformed into a "multipart/byteranges" document by prepending a dash-boundary and CRLF, and appending a close-delimiter (a CRLF, dash-boundary, terminating "--", and optional CRLF).

4.2. message/byterange+bhttp media type

The "message/byterange+bhttp" media type patches the defined byte range to some specified contents. It has the same semantics as "message/byterange", but follows a syntax closely resembling "message/bhttp" [RFC9292]

```
Request {
  Framing Indicator (i) = 8,
  Known-Length Field Section (..),
  Known-Length Content (..),
  Padding (..),
}

Known-Length Field Section {
  Length (i),
  Field Line (..) ...,
}

Known-Length Content {
  Content Length (i),
  Content (..),
}

Field Line {
  Name Length (i) = 1..,
  Name (..),
  Value Length (i),
  Value (..),
}
```

5. Caveats

This section may be removed before final publication.

5.1. Indeterminate Length Uploads

There is no standard way for a Content-Range header to indicate an unknown or indeterminate-length body starting at a certain offset; the design of partial content messages requires that the sender know the total length before transmission. However it seems it should be possible to generate an indeterminate-length partial content response (e.g. return a continuously growing audio file starting at a 4MB offset). Fixing this would require a new header, update to HTTP, or a revision of HTTP.

Ideally, this would look something like:

```
Content-Range =/ range-unit SP first-pos "-*/" ( complete-length / "*" )
```

For example: "Content-Range: bytes 200-*/*" would indicate overwriting or appending content, starting at a 200 byte offset.

And "Content-Range: bytes 200-*/4000" would indicate overwriting an unknown amount of content, but not past 4000 bytes, starting at a 200 byte offset.

Note these are different than "Content-Range: bytes 200/*" which would indicate splicing in content at a 200 byte offset.

5.2. Sparse Documents

This pattern can enable multiple, parallel uploads to a document at the same time. For example, uploading a large log file from multiple devices. However, this document does not define any ways for clients to track the unwritten regions in sparse documents, and the existing conditional request headers are designed to cause conflicts. Parallel uploads may requires a byte-level locking scheme or conflict-free operators. This may be addressed in a later document.

5.3. Recovering from interrupted PUT

Servers do not necessarily save the results of an incomplete upload; since most clients prefer atomic writes, many servers will discard an incomplete upload. A mechanism to indicate a preference for atomic vs. non-atomic writes may be defined at a later time.

Byte range PATCH cannot by itself be used to recover from an interrupted PUT that updates an existing document. If the server operation is atomic, the entire operation will be lost. If the server saves the upload, it may not possible to know how much of the request was received by the server, and what was old content that already existed.

One technique would be to use a 1xx interim response to indicate a location where the partial upload is being stored. If PUT request is interrupted, the client can make PATCH requests to this temporary, non-atomic location to complete the upload. When the last part is uploaded, the original interrupted PUT request will appear.

6. Security Considerations

6.1. Unallocated ranges

The byterange media type technically permits writes to offsets beyond the bound of the file. This may have behavior not be predictable by the user.

Servers will normally only allow patch ranges to start inside or at the immediate end of the representation. Servers supporting sparse files MUST NOT return uninitialized memory or storage contents. Uninitialized regions may be initialized prior to executing the sparse write, or this may be left to the filesystem if it can guarantee this behavior.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[RFC9112] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <https://www.rfc-editor.org/rfc/rfc9112>.

7.2. Informative References

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/rfc/rfc2046>.

[RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, DOI 10.17487/RFC4918, June 2007, <https://www.rfc-editor.org/rfc/rfc4918>.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <https://www.rfc-editor.org/rfc/rfc5789>.

[RFC9292] Thomson, M. and C. A. Wood, "Binary Representation of HTTP Messages", RFC 9292, DOI 10.17487/RFC9292, August 2022, <https://www.rfc-editor.org/rfc/rfc9292>.

Author's Address

Austin Wright

Email: aaa@bzfx.net