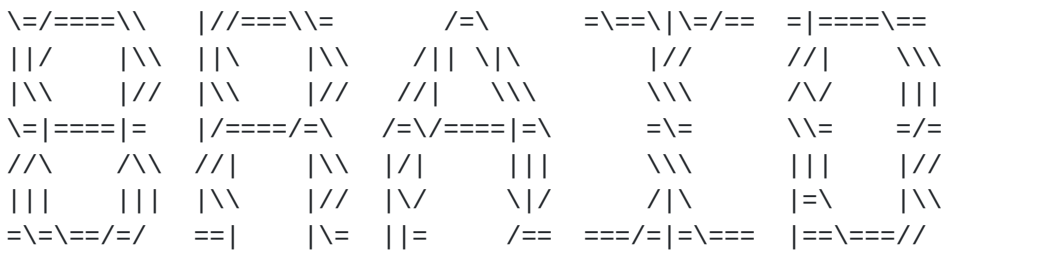
Internet-Draft
Expires: Jul 10, 2020
Intended status: Proposed Standard

M. Toomim
Invisible College
G. Little
Invisible College
R. Walker
Bard College
B. Bellomy
Invisible College

Mar 9, 2020



Braid-HTTP: Synchronization for HTTP

draft-toomim-httpbis-braid-http-02

Abstract

Braid is a set of extensions that generalize HTTP from a state *transfer* protocol into a state *synchronization* protocol. Braid puts the power of Operational Transform and CRDTs on the web, improving network performance and enabling natively peer-to-peer, collaboratively-editable, offline-first web applications.

Braid is composed of four extensions to HTTP:

1. VERSIONING on resources
2. SUBSCRIPTIONS on GET requests
3. PATCHES created from Range Requests
4. MERGE-TYPES that specify OT or CRDT behavior

These extensions are independent; each provides a distinct value for a stand-alone use-case. However, when used together, they enable a web resource to synchronize automatically across multiple clients, servers and proxies, and support arbitrary simultaneous edits by multiple writers, under arbitrary network delays and partitions, while guaranteeing consistency using a OT, CRDT, or other algorithm.

These synchronization features provide a step towards a standard for the dynamic internal state of websites. Web programmers currently synchronize state across clients and servers with layers of non-standard Javascript frameworks. A synchronization standard built upon REST can enable programmers to read and write the internal state of any website as easily as a local variable on their own site. This could enable a separation of UI from state, and allow any user to edit or choose their own UI for any website's state.

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), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://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."

The list of current Internet-Drafts can be accessed at https://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at https://www.ietf.org/shadow.html

Table of Contents

1. Introduction

HTTP transfers a static version of state within a single request and response. If the state changes, HTTP does not automatically update clients with the new versions. This design satisficed when webpages were mostly static and written by hand; however today's websites are dynamic, generated from layers of state in databases, and provide realtime updates across multiple clients and servers. Programmers today need to *synchronize*, not just *transfer* state, and to do this, they must work around HTTP.

The web has a long history of these workarounds. The original web required users to click reload when a page changed. XMLHTTPRequest [XHR] made it possible to update just part of a page, running a GET request behind the scenes. However, a GET request still could not push updates. To work around this, web programmers would poll the resource, which was inefficient. Long-polling was invented to overcome the inefficiencies, which was standardized as Server-Sent Events [SSE]. Yet, SSE provides semantics of an event-stream, not an update-stream, and although a programmer can encode a protocol within the event stream for updating a resource, there is still no standard way to express the update of a resource.

In practice, web programmers today often give up on using standards for "data that changes", and instead send custom messages over a WebSocket -- a hand-rolled synchronization protocol. Unfortunately, this forfeits the benefits of HTTP and ReST, such as caching, and a uniform interface [REST]. As the web becomes increasingly dynamic, web applications are forced to implement additional layers of non-standard Javascript frameworks to synchronize changes to state.

Braid generalizes HTTP into a synchronization protocol, and ReST into a synchronization architecture. It adds these features:

1. Versioning (Section 2)

   Each resource has a history of changes, ordered in time.

2. Subscriptions (Section 3)

   A Subscribe header can be added to GET requests, which promises to push all future versions to the client, until the client says forGET.

3. Range Patches [RANGE-PATCH]

   Express changes to versions in patches, with a uniform format based on Range Requests.

4. Merge Types [MERGE-TYPES]

   If multiple clients and servers simultaneously edit the same resource, they can guarantee a consistent resulting state by implementing the same Merge Type. Resources specify their Merge Type with a header.

Taken together, these features allow an arbitrary set of clients and servers to make arbitrary edits to resources, under arbitrary network delays and paritions, and merge all edits consistently, receiving updates as soon as they reconnect. This enables caches to support dynamic content, web applications to feature an offline mode, and textareas to support collaborative editing.

2. Versioning for Resources

Each Braid resource has a current version, and a version history. Versions are specified as a string in the [STRUCTURED-HEADERS] format. Each version string must be unique, to differentiate any two points in time. To specify the version of content in a request or response body, a Version header MAY be included in a request for a PUT, PATCH or POST, or in the response to a GET:

```
Version: "dkn7ov2vwg"
```

Every version has a set of Parents that denote the most recent parent version(s) that were known at the time the version was created. The full graph of parents forms a Directed Acyclic Graph (DAG), representing the known partial order of all versions. A version A is known to have occurred before a version B if and only if A is an ancestor of B in the partial order.

Parents are also specified with a header in a PUT request or GET response:

```
Parents: "ajtva12kid", "cmdpvkpll2"
```

The Parents header is a List of Strings, in the syntax of HTTP's [STRUCTURED-HEADERS]. Each string is a version. For any two parent versions A and B that are specified in a Parents header, A cannot be a descendent of B or vice versa. The ordering of versions in the list carries no meaning, and SHOULD be softed lexicographically.

If a client or server does not specify a Version for a resource it transfers, the recipient MAY generate a new version ID of its own

choosing. If a client or server does not specify a Parents header when transferring a new version, the recipient MAY presume that the most recent versions it has seen are the parents of the new version.

2.1. Comparison with ETag

The Version header is similar to an ETag, but has two differences:

1. ETags are sensitive to Content-Encoding. If you send the same version with a GZip Content-Encoding, it will have a different ETag, but the same Version.

2. A Version marks a unique point in time -- not unique content. If a resource is changed from version A to B, and then to C, such that the contents at A are the same as the contents at C, then it is possible versions A and C to have the same ETag, even though they have different Versions.

2.2. PUT a new version

When a PUT request changes the state of a resource, it can specify the new version of the resource, the parent versions that existed when it was created, and the way multiple simultaneous changes should be merged (the "Merge-Type"):

Request:

```
PUT /chat
Version: "ej4lhb9z78"                                  | Version
Parents: "oakwn5b8qh", "uc9zwhw7mf"                    |
Content-Type: application/json                         |
Merge-Type: sync9                                      |
Content-Length: 73                                     |
                                                       |
[{text: "Hi, everyone!",                               | | Body
  author: {type: "link", value: "/user/tommy"}}]       | |
```

Response:

```
HTTP/1.1 200 OK
Patches: OK
```

Merge-Types are specified in [MERGE-TYPES]. The Version and Parents headers are optional. If Version is omitted, the recipient may invent a version ID. If Parents is omitted, the recipient may assume that the current set of leaf versions on its machine is the version's context.

This example includes the entire new value of the state, but one can also send updates as patches.

2.3. PUT a new version as a patch

Not only are patches smaller, and thus more efficient; they also provide useful information for merging two simultaneous edits, for instance in collaborative editing.

One can send an update in a patch by setting the "Patches" header to a number, and then set the Message body to a sequence of that many patches, separated by blank lines:

Request:

```
PUT /chat
Version: "g09ur8z74r"                                  | Version
Parents: "ej4lhb9z78"                                  |
Content-Type: application/json                         |
Merge-Type: sync9                                      |
Patches: 2                                             |
                                                       |
Content-Length: 62                                     | | Patch
Content-Range: json .messages[1:1]                     | |
                                                       | |
[{text: "Yo!",                                         | |
  author: {type: "link", value: "/user/yobot"}]        | |
                                                       |
Content-Length: 40                                     | | Patch
Content-Range: json .latest_change                     | |
                                                       | |
{"type": "date", "value": 1573952202370}               | |
```

Response:

```
HTTP/1.1 200 OK
Patches: OK
```

In order to distinguish each patch within a Version, we need to know the length of the patch. To know the length of the patch, each patch must include one of the following headers:

```
Content-Length: N
Transfer-Encoding: chunked
```

Either of these provide a way to determine when the next message starts.

The previous example uses the Range Patch format, which is defined in [RANGE-PATCH]. However, one can use any patch format, by sending a patch with a Content-Type: set to a patch format with a defined behavior, such as application/json-patch+json (as specified in [RFC6902]):

Request:

```
PUT /chat
Version: "up12vyc5ib"                                  | Version
Parents: "2bcbi84nsp"                                  |
Content-Type: application/json                         |
Merge-Type: sync9                                      |
Patches: 1                                             |
                                                       |
Content-Length: 326                                    | | Patch
Content-Type: application/json-patch+json              | |
                                                       | |
[                                                      | |
  { "op": "test", "path": "/a/b/c", "value": "foo" },  | |
  { "op": "remove", "path": "/a/b/c" },                | |
  { "op": "add", "path": "/a/b/c", "value": [] },      | |
  { "op": "replace", "path": "/a/b/c", "value": 42 },  | |
  { "op": "move", "from": "/a/b", "path": "/a/d },     | |
  { "op": "copy", "from": "/a/d/d", "path": "/a/d/e" }| |
]                                                      | |
```

Response:

```
HTTP/1.1 200 OK
Patches: OK
```

2.4. GET a specific version

A server can optionally allow clients to request historical versions of a resource in GET requests. To request a historical version, a client includes a Version and/or Parents header in the request.

Request:

```
GET /chat
Version: "ej4lhb9z78"
```

Response:

```
HTTP/1.1 209 Subscription
```

```
      Subscribe: keep-alive

      Version: "ej4lhb9z78"                                  | Version
      Parents: "oakwn5b8qh", "uc9zwhw7mf"                    |
      Content-Type: application/json                         |
      Merge-Type: sync9                                      |
      Content-Length: 73                                     |
                                                             |
      [{text: "Hi, everyone!",                               | | Body
        author: {type: "link", value: "/user/tommy"}}]       | |
```

If a GET request contains a Version header:

- The Subscribe header (Section 3) MUST be absent.

- The server SHOULD return a single response, containing that version of the resource in its body, with the Version header set to the version requested by the client.

- If the server does not support historical versions, it MAY ignore the Version header and respond as usual, but MUST NOT include the Version header in its response.

If a GET request contains a Parents header:

- If the request does not also contain a Version, then the request MUST also contain a Subscribe header, and the server SHOULD send a set of versions connecting the Parents to the current Version, and then subscribe the client to future updates.

- If the request also contains a Version, then the server SHOULD respond with a set of versions that connect the specified Parents with the specified Version, and then close the connection.

- If the server does not support historical versions, then it MAY ignore the Parents header, but MUST NOT include the Parents header in its response.

A server MAY refactor or rebase the version history that it provides to a client, so long as it does not affect the resulting state, or the result of the patch-type's merges.

3. Subscriptions for GET

If a GET request includes the Subscribe header, it will return a stream of versions; a new version pushed with each change. Each version can contain either the new contents in its body, or a set of Patches.

Request:

```
   GET /chat
   Subscribe: keep-alive

Response:

   HTTP/1.1 209 Subscription
   Subscribe: keep-alive

   Version: "ej4lhb9z78"                                  | Version
   Parents: "oakwn5b8qh", "uc9zwhw7mf"                    |
   Content-Type: application/json                         |
   Merge-Type: sync9                                      |
   Content-Length: 73                                     |
                                                          |
   [{text: "Hi, everyone!",                               | | Body
     author: {type: "link", value: "/user/tommy"}}]       | |

   Version: "g09ur8z74r"                                  | Version
   Parents: "ej4lhb9z78"                                  |
   Content-Type: application/json                         |
   Merge-Type: sync9                                      |
   Patches: 1                                             |
                                                          |
   Content-Length: 62                                     | | Patch
   Content-Range: json .messages[1:1]                     | |
                                                          | |
   [{text: "Yo!",                                         | |
     author: {type: "link", value: "/user/yobot"}]        | |

   Version: "2bcbi84nsp"                                  | Version
   Parents: "g09ur8z74r"                                  |
   Content-Type: application/json                         |
   Merge-Type: sync9                                      |
   Patches: 1                                             |
                                                          |
   Content-Length: 68                                     | | Patch
   Content-Range: json .messages[2:2]                     | |
                                                          | |
   [{text: "Hi, Tommy!",                                  | |
     author: {type: "link", value: "/user/sal"}}]         | |

   Version: "up12vyc5ib"                                  | Version
   Parents: "2bcbi84nsp"                                  |
   Content-Type: application/json                         |
   Merge-Type: sync9                                      |
   Patches: 1                                             |
                                                          |
   Content-Length: 326                                    | | Patch
   Content-Type: application/json-patch+json              | |
                                                          | |
```

```
[                                                          | |
  { "op": "test", "path": "/a/b/c", "value": "foo" },      | |
  { "op": "remove", "path": "/a/b/c" },                    | |
  { "op": "add", "path": "/a/b/c", "value": [] },          | |
  { "op": "replace", "path": "/a/b/c", "value": 42 },      | |
  { "op": "move", "from": "/a/b", "path": "/a/d },         | |
  { "op": "copy", "from": "/a/d/d", "path": "/a/d/e" }      | |
]                                                          | |
```

3.1. Creating a Subscription

The "Subscribe" header on a GET request modifies the method semantics to request a subscription to future updates to the data, rather than only the current version of the representation data.

A client requests a subscription by issuing a GET request with a Subscribe header:

```
      Subscribe
or    Subscribe: keep-alive
or    Subscribe: keep-alive=<seconds>
```

If a server implements Subscribe, it MUST include a Subscribe header in its response. The server then SHOULD keep the connection open, and send updates over it.

In general, a server that implements subscriptions promises to keep its subscribed clients up-to-date by sending changes until the client closes the subscription. A subscription is different from a GET connection (e.g. a TCP connection, or HTTP/2 stream). If a client requests "Subscribe: keep-alive", then the subscription will be remembered even after the GET connection closes. A subscription can be resumed by the client issuing another GET with a Subscribe header.

3.2. Sending multiple updates per GET

To send multiple updates, a server concatenates multiple sub-responses into a single response body. Each sub-response must contain its own headers and body. Each sub-response must have a known length, which means it must contain one of the following headers:

- Content-Length: N
- Transfer-Encoding: chunked
- Patches: N

Each sub-response must have both headers and a body. The body may be zero-length.

3.3. Continuing a Subscription

Even if a connection closes, a subscription might still be active. If a server's response headers for a connection contained:

```
        Subscribe: keep-alive
   or   Subscribe: keep-alive=<seconds>
```

Then the server SHOULD keep the subscription open even after the connection closes. This means that the server promises to keep enough history to merge with the client when the client comes back online.

When the client reconnects, it may specify the most recent versions it saw from the server using the Parents header. This tells the server which versions of state to catch it up from.

The server can suggest how long it will wait for the client by responding with Subscribe: keep-alive=<seconds>. A server should wait at least <seconds> after a connection closes before dropping the subscription, and clearing its history.

3.4. Ending a Subscription

Servers and clients MAY drop a subscription at any time, no matter the value of keep-alive. A client may reconnect by issuing a new GET request with a new Subscribe header.

If a subscription is set to keep-alive, then closing the TCP/QUIC connection won't end the subscription. Thus a client needs a way to explicitly end the subscription. In HTTP/1, this is by sending the text "forGET\n" over the TCP connection. In HTTP/2, this is by issuing a CLOSE event to the GET request's stream. Alternatively, since today's web browsers do not support sending extra text after a request body, the client can issue a fresh request specified as a FORGET method.

3.5. Errors

If a server has dropped the history that a client requests, the server can return a 410 GONE response, to tell the client "sorry, I don't have the history necessary to synchronize with you."

4. Design Goals

This spec is designed to be:

1. Backwards-compatible with existing HTTP

2. Easy to implement simple synchronizers with. For instance, it should be easy to write a read-only synchronizer for an append-only log.

3. Possible to implement arbitrary synchronization algorithms. For instance, these extensions support any Operational Transform or CRDT algorithm.

5. Use Cases

5.1. Dynamic Resources: Animating a GIF

Braid allows resources to become inherently dynamic -- able to change over time. You can use this to make a resource animate.

In this example, a server streams changes to a GIF file in a sequence of patches. When the client renders the new state of the GIF after each patch, a new frame of animation is displayed.

```
  Request:
    GET /animated-braid.gif
    Subscribe

  Response:
    HTTP/1.1 209 Subscribe
    Content-Type: image/gif          | Version
    Patches: 2                       |
                                     |
    Content-Length: 1239             | | Patch
    Content-Range: bytes 100-200     | |
                                     | |
    <binary data>                    | |
                                     |
    Content-Length: 62638            | | Patch
    Content-Range: bytes 348-887     | |
                                     | |
    <binary data>                    | |
```

5.2. Dynamic Proxies and Caches

Since updates aren't pushed, today's web often uses timeouts to trigger a cache becoming stale. Unfortunately, sometimes the timeout is wrong, and caches become out-of-date, and we have to wait for an unknown cache to timeout before we can see the new version of something. As a result, programmers have learned to force-reload pages habitually, and caches become less efficient than necessary.

A cache supporting the Braid extensions, however, will automatically update whenever a change occurs. If a client starts a GET

Subscription with a proxy, the proxy will then start and maintain a GET Subscription with the origin server. The origin server will promise to send the proxy updates over its GET Subscription, and the proxy will then relay these changes to all connected clients. If a set of clients and servers all support Braid, they will never need to force-reload caches for any data amongst them.

5.3. A Serverless Chat Example

A Braid web application can operate offline. A user can use the app from an airplane, and their edits can synchronize when they regain internet connections. Additionally, the Braid protocol can be expressed over peer-to-peer transports (e.g. Braid-WebRTC) to support a peer-to-peer synchronization without a server. Braid-HTTP clients will be able to interoperate with Braid-WebRTC peers. For example, a chat application might be served and synchronized on Braid-HTTP, while also establishing redundant peer-to-peer connections on Braid-WebRTC. The server could then be shut down, and users of the chat app could continue to send messages to one another.

Imagine the server serves the current set of trusted clients' IP addresses at the /peers state. Each client then subscribes to the /peers state with:

```
GET /peers
Subscribe: keep-alive
-------
[ {ip: '13.55.32.158', pubkey: 'x371...8382'},
  {ip: '244.38.55.83', pubkey: 'o2u8...2s73'},
  ...
]
```

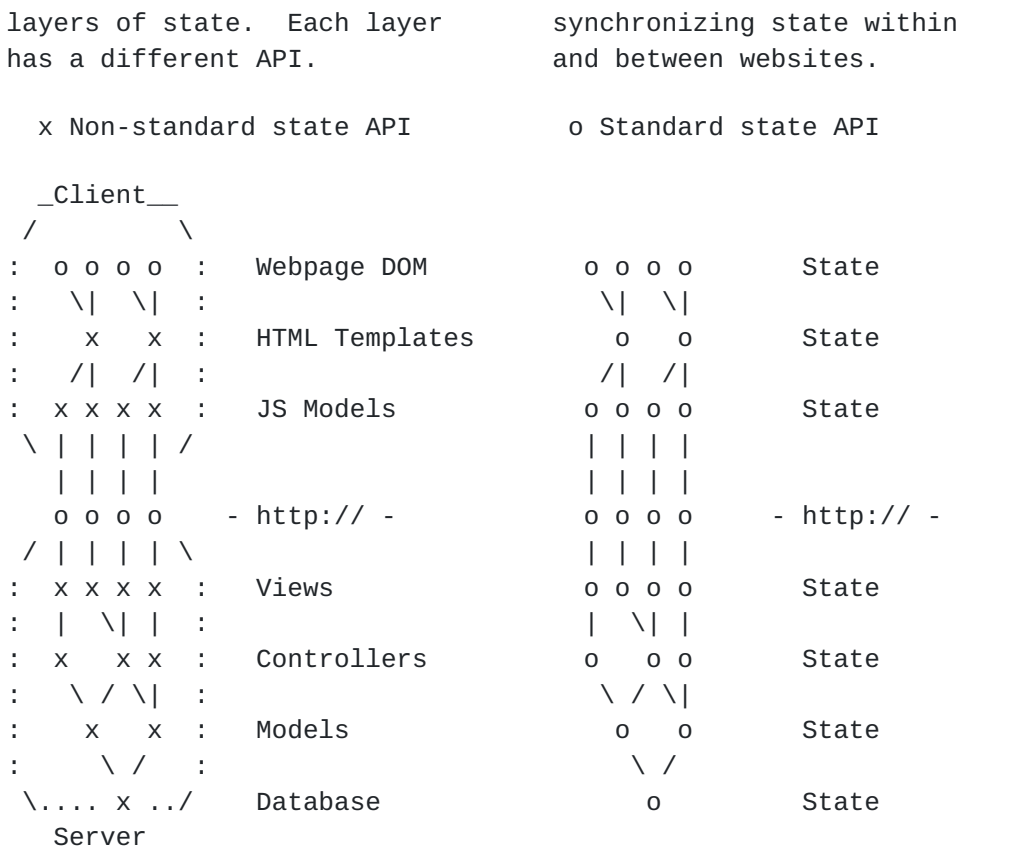
Each peer can then choose a set of those peers with whom to establish a WebRTC connection. It will then exchange Braid messages with those peers over that connection.

6. Related Work

6.1. Web Frameworks

Web applications typically synchronize the state of a client and server with layers of models, views, and controllers in web frameworks. By automating synchronization within HTTP, programmers have to write fewer layers of code on top of it.

====== Legacy Websites ======	====== Braid Websites ======
Today's webpages are generated from multiple	Braid generalizes HTTP into a standard for



6.2. Existing IETF Standards

A number of IETF specifications already standardize aspects of synchronization for specific domains. IMAP [RFC3501] provides synchronization of email. WebDAV provides the synchronization of "collections" [RFC6578], and has been extended specifically for calendar data in CalDAV [RFC4791], and vCards in [RFC6350]. More recently, JMAP [RFC8620] provides an updated method of synchronization, supporting mail, calendars, and contacts.

7. IANA Considerations

7.1. Header Field Registration

HTTP header fields are registered within the "Message Headers" registry maintained at <http://www.iana.org/assignments/message-headers/>.

This document defines the following HTTP header fields, so their associated registry entries have been updated according to the permanent registrations below (see [BCP90]):

Header Field Name	Protocol	Status	Reference
Version	http	experimental	Section 2
Parents	http	experimental	Section 2
Merge-Type	http	experimental	Section 2.2
Patches	http	experimental	Section 2.3
Subscribe	http	experimental	Section 4

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

8. Security Considerations

XXX Todo

9. Conventions

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

10. Copyright Notice

11. References

11.1. Normative References

[RFC7230] "HTTP/1.1 Message Syntax and Routing", RFC 7230.

[RFC7231] "HTTP/1.1 Semantics and Content", RFC 7231.

[RFC7233] "HTTP/1.2 Range Requests", RFC 7233.

[RFC7234] "HTTP/1.2 Caching", RFC 7234.

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

[MERGE-TYPES] draft-toomim-httpbis-merge-types-00

[RANGE-PATCH] draft-toomim-httpbis-range-patch-00

[STRUCTURED-HEADERS] draft-ietf-httpbis-header-structure-14

11.2. Informative References

[XHR] Van Kestern, A., Aubourg, J., Song, J., and R. M. Steen, H. "XMLHttpRequest", September 2019. <https://xhr.spec.whatwg.org/>

[SSE] Hickson, I. "Server-Sent Events", W3C Recommendation, February 2015. <https://www.w3.org/TR/2015/REC-eventsource-20150203/>

[REST] Fielding, R. "Architectural Styles and the Design of Network-based Software Architectures" Doctoral dissertation, University of California, Irvine, 2000.

[RFC3501] Crispin, M., "Internet Message Access Protocol - Version 4rev1", RFC 3501, DOI 10.17487/RFC3501, March 2003, <https://www.rfc-editor.org/info/rfc3501>.

[RFC6578] Daboo, C., Quillaud, A., "Collection Synchronization for Web Distributed Authoring and Versioning (WebDAV)", RFC 6578, DOI 10.17487/RFC6578, March 2012, <https://www.rfc-editor.org/info/rfc6578>.

[RFC4791] Daboo, C., Desruisseaux, B., Dusseault, L., "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <https://www.rfc-editor.org/info/rfc4791>.

[RFC6350] Perreault, S., "vCard Format Specification", RFC 6350, DOI 10.17487/RFC6350, August 2011, <https://www.rfc-editor.org/info/rfc6350>.

[RFC8620] Jenkins, N., Newman, C., "The JSON Meta Application Protocol (JMAP)", RFC 8620, DOI 10.17487/RFC8620, July 2019, <https://www.rfc-editor.org/info/rfc8620>.

[RFC6902] Bryan, P., Nottingham, M., "Javascript Object Notation (JSON) Patch", RFC 6902.

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

12. Acknowledgements

In addition to the authors, this spec contains intellectual contributions from the following people:

- Seph Gentle
- Mitar Milutinovic
- Sarah Allen
- Duane Johnson
- Travis Kriplean
- Marius Nita
- Paul Pham
- Morgan Dixon
- Karthik Palaniappan

We thank the following people for key feedback on previous drafts:

- Austin Wright
- Martin Thomson
- Eric Kinnear
- Olli Vanhoja
- Julian Reschke

We also thank Mark Nottingham, Fred Baker, Adam Roach, and Barry Leiba for facilitating a productive environment.

13. Authors' Addresses

For more information, the authors of this document are best contacted via Internet mail:

Michael Toomim
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

EMail: toomim@gmail.com

Web: https://invisible.college/@toomim

Greg Little
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

EMail: glittle@gmail.com
Web: https://glittle.org/

Rafie Walker
Bard College

EMail: slickytail.mc@gmail.com

Bryn Bellomy
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

EMail: bryn@signals.io
Web: https://invisible.college/@bryn