Social Web Protocols

W3C Editor's Draft 20 June 2017

This version:
https://w3c-social.github.io/social-web-protocols

Latest published version:
https://www.w3.org/TR/social-web-protocols/

Latest editor's draft:
https://w3c-social.github.io/social-web-protocols

Editor:
Amy Guy, University of Edinburgh, amy@rhiaro.co.uk

Repository:
Git repository
Issues
Commits

Abstract

The Social Web Protocols are a collection of standards which enable various aspects of decentralised social interaction on the Web. This document describes the purposes of each, and how they fit together.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the Social Web Working Group as an Editor's Draft. Comments regarding this document are welcome. Please send them to public-socialweb@w3.org (subscribe, archives).

Publication as an Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 March 2017 W3C Process Document.

Table of Contents

1. Overview

People and the content they create are the core components of the social web; they make up the social graph. This document describes a standard way in which people can:

- connect with other people and subscribe to their content;
- create, update and delete social content;
- interact with other peoples' content;
- be notified when other people interact with their content;

regardless of *what that content* is or *where it is stored*.

These components are core building blocks for interoperable social systems.

Each of these components can be implemented independently as needed, or all together in one system, as well as extended to meet domain-specific requirements. Users can store their social data across any number of compliant servers, and use compliant clients hosted elsewhere to interact with their own content and the content of others. Put simply, this document tells you, according the recommendations of the Social Web Working Group:

- how to expose/consume social content (reading).
- what to post, and where to, in order to create, update or delete content.
- how to ask for notifications about content (subscribing).
- how to deliver notifications about content or users (delivery).
- how to expose profiles and relationships.

1.1 Contents

This is an overview of the current state of specifications of the Social Web Working Group. This document also describes Social Web Working Group specifications in context of other W3C recommendations and relevant drafts from other communities outside of the W3C. As many of these specs are under ongoing development, this document is subject to change alongside them.

You might also want to take a look at the Working Group's Social API Requirements to understand the division of concerns.

Due to the diversity of viable technical approaches proposed within the Working Group, several specifications on the recommendation track provide overlapping functionality through differing mechanisms. This document exists to provide informative guidance on how and when to implement the various specifications, as well as guidance to implement bridges between recommendations with overlapping functionality.

1.2 Recommendations

These specifications are finished, though new implementation reports and feedback are always welcome (details for where to submit these are at the top of each document).

ActivityStreams 2.0 (AS2) (Core, Vocabularly)
The syntax and vocabulary for representing social activities, actors, objects, and collections in JSON and RDF.

Linked Data Notifications

An RDF-based protocol for delivery.

Micropub
A form-encoding and JSON-based API for publishing, updating and deleting.

Webmention
A form-encoding-based protocol for delivery.

1.3 Specifications in progress

These specifications are still being worked on. Your attention, implementation, and issue-raising is particularly appreciated.

ActivityPub (Candidate Recommendation)
JSON-based APIs for reading, publishing, updating, deleting, subscribing, delivery and profiles.

WebSub (Candidate Recommendation)
A protocol for subscription to any resource and delivery of updates about it.

JF2 (First Public Working Draft (Note))
An alternative JSON serialization for representing objects and feeds of social data.

Post Type Discovery (Working Draft)
A protocol to bridge implicitly typed objects and explicitly typed objects.

1.4 Other specifications

ActivityStreams 1.0 [External]
The predecessor to [ActivityStreams2].

Annotation Protocol [WAWG CR]
Transport mechanisms for creating and managing annotations on the Web.

Linked Data Platform [LDP REC]
A protocol for reading and writing Linked Data.

1.5 Requirements

This table shows the high level requirements according to the Social Web Working Group charter and the Social API Requirements, and how the specifications of the Working Group overlap with respect to each.

	Data model				Social API			Federation API	
	Vocabulary	Syntax	Profiles	Read	Create	Update	Delete	Subscription	Delivery
ActivityPub			X	X	X	X	X	X	X
ActivityStreams 2.0	X	X	X						
Linked Data Notifications									X
Micropub					X	X	X		
WebSub								X	
Webmention									X
JF2	X	X							
Post-type discovery				X					

1.6 How do these specifications relate to each other?

As a quick reference, some key relationships between specifications are as follows:

- **ActivityPub and ActivityStreams 2.0**: ActivityPub uses the AS2 syntax and vocabulary for the payload of all requests.
- **ActivityPub and Linked Data Notifications**: ActivityPub specialises LDN as the mechanism for delivery of notifications by requiring that payloads are AS2. *Inbox* endpoint discovery is the same. LDN receivers can understand requests from ActivityPub federated servers, but ActivityPub servers can't necessarily understand requests from generic LDN senders.
- **ActivityStreams 2.0 and Linked Data Notifications**: LDN MAY use the AS2 syntax and vocabulary for the payload of notification requests.
- **Webmention and Linked Data Notifications**: Overlapping functionality that needs to be bridged due to different content types of requests. An LDN request MAY contain the equivalent data as a Webmention request, but not necessarily vice versa.
- **ActivityPub and Micropub**: Overlapping functionality that needs to be bridged due to different vocabularies and possibly different content types of requests. Micropub specifies

client-to-server interactions for content creation; ActivityPub specifies this, *plus* side-effects and server-to-server interactions.

- **Micropub and Webmention**: Are complementary but independent. Content could be created with Micropub, then Webmention discovery can be commenced on any URLs in the content.
- **Micropub and Linked Data Notifications**: Are complementary but independent. Content could be created with Micropub, then LDN discovery can be commenced on any relevant resources identified by the server.
- **Micropub and WebSub**: Are complementary but independent. Content could be created with Micropub, then passed to a WebSub *hub* for delivery to subscribers.

2. Reading

If you are a content publisher, this section is about how you should publish your content. If you are a content consumer, this is what you should expect to consume.

2.1 Content representation

[ActivityStreams2] is the recommended syntax and vocabulary for social data. ActivityStreams 2.0 represents content and interactions as *Objects* and *Activities*. The ActivityStreams vocabulary defines a finite set of common Object and Activity types and properties, as well as an extension mechanism for applications which want to expand on or specialise these.

ActivityStreams 2.0 content MUST be served with the Content-Type `application/activity+json` or for JSON-LD extended implementations, `application/ld+json; profile="https://www.w3.org/ns/activitystreams"`. Consumers should recognise both of these Content-Types as ActivityStreams 2.0.

In order to claim ActivityStreams 2.0 conformance, a data publisher MUST use ActivityStreams 2.0 vocabulary terms where available, and *only* use other vocabularies *in addition*, or where no appropriate ActivityStreams 2.0 term exists.

> NOTE: ActivityStreams 1.0
>
> ActivityStreams 2.0 builds upon [AS1] and is not fully backwards compatible; the relationship between AS1 and AS2 is documented in the AS2 spec. If you have implemented [AS1], you should transition to ActivityStreams 2.0.

[Activitypub] uses ActivityStreams 2.0 for all data, and also *extends* ActivityStreams 2.0 with additional terms. Thus, ActivityPub requires requests have the Content-Type `application/ld+json; profile="https://www.w3.org/ns/activitystreams"`.

To make content available as ActivityStreams 2.0 JSON, one could do so directly when requested with an appropriate `Accept` header (eg. `application/activity+json` or `application/ld+json`), or indirectly via a `rel="alternate" type="application/activity+json"` link . This link could be to a different domain, for third-party services which dynamically generate ActivityStreams 2.0 JSON on behalf of a publisher.

2.1.1 Other ways of representing content

If you don't like ActivityStreams 2.0, there are still some specifications you can use for particular tasks.

[LDN] notification contents can use any vocabulary, so long as the data is available in JSON-LD. Thus notifications MAY use ActivityStreams 2.0, but don't have to.

[Micropub] clients which expect to read data (this would usually be clients for *updating*) are expecting it as JSON in the parsed microformats2 syntax.

[WebSub] is agnostic as to the Content-Type used by publishers; hubs are expected to deliver the new content to subscribers as-is produced by the publisher, at the publisher's `topic` URL.

> NOTE: JF2
>
> JF2 is an alternative syntax and vocabulary to ActivityStreams 2.0, based on a simplification of the microformats2 parsing algorithm, which may be published as a Working Group Note. It is not currently referenced by any other specifications in the WG.

2.2 Objects

All objects MUST have URLs in the `id` property, which return the properties of an object according to content representation, and depending on the requester's right to access the content.

2.3 Streams

Each stream MUST have a URL which MUST result in the contents of the stream (according to the requester's right to access, and could be paged), and MAY include additional metadata about the stream (such as title, description).

Each object in a stream MUST contain at least its URL, which can be dereferenced to retrieve all properties of the object, and MAY contain other properties of the object.

One user may publish one or more streams of content. Streams may be generated automatically or manually, and might be segregated by post type, topic, audience, or any arbitrary criteria decided by the curator of the stream. A user profile MAY include links to multiple streams, which a consumer could follow to read or subscribe to.

2.3.1 Special streams

[Activitypub] specifies two streams that MUST be accessible from a profile via the following properties:

- `inbox`: A reference to an [ActivityStreams2] collection comprising all the objects received by the actor.
- `outbox`: An [ActivityStreams2] collection comprising all the objects produced by the actor.

[Activitypub] also specifies four further properties for accessing additional streams, which SHOULD or MAY be included in a profile:

- `following`: An [ActivityStreams2] collection of the actors that this actor is following.
- `followers`: An [ActivityStreams2] collection of the actors that follow this actor
- `likes`: An [ActivityStreams2] collection of every `object` from all of the actor's `Like` activities (generated automatically by the server).
- `streams`: A list of supplementary Collections which may be of interest.

[Activitypub] permits arbitrary streams to be created through specifying special behavior for the server when it receives activities with types `Add` and `Remove`. When a server receives such an activity in the `outbox`, and the `target` is a Collection, it MUST add the `object` to the `target` (for `Add`) or remove the `object` from the `target` (for `Remove`).

2.3.2 Inboxes

An Inbox is a collection or stream of objects to which new objects may be delivered. The contents of this collection may also be read. While ActivityPub and LDN align for *writing* to an Inbox, due to irritating but ultimately unavoidable compatibility requirements with AS2 and LDP respectively, they use different vocabularies when Inbox contents are returned for *reading*. Fortunately for consuming clients to check for both vocabularies or for publishers to publish using both is not a huge implementation hurdle, so bridging is fairly trivial.

- ActivityPub Inboxes are an [ActivityStreams2] `OrderedCollection`, and link to their contents with the `items` property.
- LDN Inboxes are an [LDP] `Container`, and link to their contents with the `contains` property.

In addition, ActivityPub has more constraints on the content type of the `inbox` stream, so LDN receivers either need to conform to these constraints (basically just making sure all your JSON-LD is compacted) or ActivityPub clients need to incorporate proper RDF parsing when reading an `inbox` stream.

In summary:

ActivityPub servers wishing to be read by LDN consumers MUST:

- serve content with the content-type `application/ld+json; profile="http://www.w3.org/ns/activitystreams"`.
- include along with the Inbox `Collection`, every item in the Inbox related with the `ldp:contains` property.

ActivityPub clients wishing to read from LDN Inboxes MUST:

- request content with the content-type `application/ld+json`. Running it through JSON-LD compacting should help.
- look for items in the Inbox via the `ldp:contains` property (rather than AS2 `items`).

LDN receivers wishing to interoperate with ActivityPub consuming clients MUST:

- treat the content-type `application/activity+json` as equivalent to `application/ld+json; profile="http://www.w3.org/ns/activitystreams"`.
- use compacted JSON-LD.
- serve Inbox contents as an AS2 `Collection` (where each notification is related to the inbox with the AS2 `items` property).

LDN consumers wishing to read from inboxes on ActivityPub servers MUST:

- if content is returned with the content-type `application/activity+json` and missing an `@context`, apply the default AS2 context, and treat it as JSON-LD.
- look for items in the Inbox via the AS2 `items` property (rather than `ldp:contains`).

3. Publishing

Publishing in this context incorporates creating new content, and updating or deleting existing content. Content generated through a client (such as a web form, mobile app, sensor, smart device) is created when it is sent to a server for processing, where it is typically stored and usually published (either publicly or to a restricted audience, in human- and/or machine-readable forms). Clients and servers may independently support creating, updating and deleting; there are no dependencies between them.

Authentication and authorization between clients and servers for creating content are left out of scope.

The two specifications recommended by the Social Web Working Group for publishing are [Activitypub] and [Micropub]. They use similar high level mechanisms, but differ in requirements around both the vocabularies and content types of data. ActivityPub contains a client-to-server API for creating ActivityStreams 2.0 objects and activities, and specifies additional responsibilities for clients around addressing objects, and for servers around the side-effects of certain types of objects. Micropub provides a basic client-to-server API for creating blog-post type content which can be implemented alone and is intended as a quickstart for content creation, on top of which more complex (but optional) actions can be layered.

Both provide similar media endpoints for uploading files.

> NOTE: REST
>
> Neither ActivityPub nor Micropub define APIs for publishing based on HTTP verbs. If you're looking for something more RESTful, you may like the Linked Data Platform ([**LDP**]).

3.1 Creating

The publishing endpoint of **[Activitypub]** is the `outbox`. Clients are assumed to have the URL of a (ideally authenticated) user profile as a starting point, and discover the value of the `https://www.w3.org/ns/activitystreams#outbox` property found at the profile URL (which should be available as JSON). The client then makes an HTTP `POST` request with an [ActivityStreams2] activity or object as a JSON payload with a content type of `application/ld+json; profile="https://www.w3.org/ns/activitystreams"`. The URL of the created resource is generated at the discretion of the server, and returned in the `Location` HTTP header. This is an appropriate protocol to use when:

- You want to send/receive a JSON or JSON-LD payload.
- Your data is described with [ActivityStreams2] (optionally extensible via JSON-LD).
- You want serves to carry out a known set of actions upon content creation.

Side-effects of creating content with ActivityPub are for the most part adding things to various different collections collections (likes, follows, etc); but also include requirements about blocking users, and a hook to enable federated servers.

The publishing endpoint for **[Micropub]** is the `micropub` end point. Clients discover this from a user's URL via a `rel="micropub"` link (in an HTTP `Link` header, or an HTML element). Clients make a `x-www-form-urlencoded` POST request containing the key-value pairs for the attributes of the object being created. The URL of the created resource is generated at the discretion of the server, and returned in the `Location` HTTP header. Clients and servers must support attributes from the Microformats 2 [h-entry] vocabulary. Micropub also defines special reserved attributes (prefixed with `mp-`) which can be used as commands to the server. Any additional key names sent outside of these vocabularies may be ignored by the server.

[Micropub] requests may alternatively be sent as a JSON payload, the syntax of which is derived from the Microformats 2 parsing algorithm. This is an appropriate protocol to use when:

- You want to send/receive a form-encoded or JSON payload.
- Your data is described with the [h-entry] syntax and vocabulary.
- You can rely on out-of-band agreements between clients and servers for vocabulary extensibility.

3.2 Updating

Content is updated when a client sends changes to attributes (additions, removals, replacements) to an existing object. If a server has implemented a delivery or subscription mechanism, when an object is updated, the update MUST be propagated to the original recipients using the same mechanism.

[Activitypub] clients send an HTTP `POST` request to the `outbox` containing an [ActivityStreams2] `Update` activity. The `object` of the activity is an existing object, and the fields to update should be nested. If a partial representation of an object is sent, omitted fields are *not* deleted by the server. In order to delete specific fields, the client can assign them a `null` value. However, when a federated server passes an `Update` activity to another server's *`inbox`*, the recipient must assume this is the *complete* object to be replaced; partial updates are not performed server-to-server.

[Micropub] clients perform updates, as either form-encoded or JSON POST requests, using the `mp-action=update` parameter, as well as a `replace`, `add` or `delete` property containing the updates to make, to the Micropub endpoint. `replace` replaces all values of the specified property; if the property does not exist already, it is created. `add` adds new values to the specified property without changing the existing ones; if the property does not exist already, it is created. `delete` removes the specified property; you can also remove properties by value by specifying the value.

3.3 Deleting

Content is deleted when a client sends a request to delete an existing object. If a server has implemented a delivery or subscription mechanism, when an object is deleted, the deletion MUST be propagated to the original recipients using the same mechanism.

[Activitypub] clients delete an object by sending an HTTP POST request containing an [ActivityStreams2] `Delete` activity to the `outbox` of the authenticated user. Servers MUST either *replace* the `object` of this activity with a tombstone and return a `410 Gone` status code, or return a `404 Not Found`, from its URL.

[Micropub] delete requests are two key-value pairs, in form-encoded or JSON: `mp-action`: `delete` and `url`: `url-to-be-deleted`, sent to the Micropub endpoint .

4. Subscribing

An agent (client or server) may *ask* to be notified of changes to a content object (eg. edits, new replies) or stream of content (eg. objects added or removed from a particular stream). This is *subscribing*. Specifications which contain subscription mechanisms are ActivityPub and WebSub.

> NOTE: Subscription vs delivery
>
> Nothing should rely on implementation of a subscription mechanism. That is, implementations may set themselves up to receive notifications without always being required to explicitly ask for them from a sender or publisher: see delivery.

4.1 Subscribing with `as:Follow`

[Activitypub] servers maintain a *Followers* collection for all users. This collection may be directly addressed, or addressed automatically or by default, in the `to`, `cc` or `bcc` field of any Activity, and as a result, servers deliver the Activity to the `inbox` of each user in the collection.

Subscription requests are essentially requests to be added to this collection. They are made by the subscriber's server `POST`ing a `Follow` Activity to the target's `inbox`. This request should be authenticated, and therefore doesn't need additional verification. The target server then SHOULD add the subscriber to the target's Followers collection. Exceptions may be made if, for example, the target has blocked the subscriber.

This is a suitable subscription mechanism when:

- The subscriber wants to request updates from a specific actor (rather than objects, streams or threads).
- The subscriber and publisher both speak ActivityStreams 2.0.
- The publisher is aware of who has subscribed, and capable of delivering content to subscribers itself.

Since delivery is only a requirement for federated servers, prospective subscribers will not be able to `POST` their `Follow` activity to the `inbox` of a profile which is on a non-federated server (expect a `405 Method Not Allowed`), and thus are not able to subscribe to these profiles. In this case, prospective subscribers may wish to periodically *pull* from the publisher's `outbox` instead.

4.2 Delegating subscription handling

[WebSub] provides a mechanism to delegate subscription handling and delivery of content to subscribers to a third-party, called a *hub*. All publishers need to do is link to their chosen hub(s) using HTTP `Link` headers or HTML `<link>` elements with `rel="hub"`, and then notify the hub when new content is available. The mechanism for notifying the hub is left deliberately unspecified, as publishers may have their own built in hub, and therefore use an internal mechanism.

> NOTE: Notifying the hub
>
> Hubs and publishers which would like to agree on a standard mechanism to communicate might consider employing an existing delivery mechanism, namely Linked Data Notifications or Webmention.

The subscriber discovers the hub from the publisher, and sends a form-encoded `POST` request containing values for `hub.mode` ("subscribe"), `hub.topic` (the URL to subscribe to) and `hub.callback` (the URL where updates should be sent to, which should be 'unguessable' and unique per subscription). The hub and subscriber carry out a series of exchanges to verify this request.

When the hub is notified of new content by the publisher, the hub fetches the content of the `topic` URL, and delivers this to the subscriber's `callback` URL.

This is a suitable subscription mechanism when:

- The subscriber wants to request updates from any resource (not just user profiles), and of any content type.
- Subscription requests are not authenticated, so you need a way to verify them.
- The publisher wants to delegate distribution of updates to another service (the hub) instead of doing it itself.

> NOTE: WebSub and Inboxes
>
> LDN Receivers can receive deliveries from WebSub hubs by using the `inbox` URL as the `hub.callback` URL and *either* only subscribing to resources published as JSON-LD *or* accepting content-types other than JSON-LD.

5. Delivery

A user or application may wish to push a notification to another user that the receiver has *not explicitly asked* for. For example to send a message or some new information; because they have linked to (replied, liked, bookmarked, reposted, etc) their content; because they have linked to (tagged, addressed) the user directly; to make the recipient aware of a change in state of some document or resource on the Web. The Social Web Working Group specifications contain several mechanisms for carrying out delivery; they are listed here from general to specialsed.

> NOTE: Delivery vs subscription
>
> We need to leave it open for users to refuse content they have not explicitly subscribed to, ie. nothing else should rely on implementation of Delivery.

5.1 Targeting and discovery

The target of a notification is usually the addressee or the subject, as referenced by a URL. The target may also be someone who has previously requested notifications through a subscription request. Once you have determined your target, you need to discover where to send the notification for that particular target. Do this by fetching the target URL and looking for a link to an endpoint which will accept the type of notification you want to send (read on, for all of your exciting options).

Bear in mind that many potential targets will *not* be configured to receive notifications at all. To avoid overloading unsuspecting servers with discovery-related requests, your application should employ a "back-off" strategy when carrying out discovery multiple times to targets on the same domin. This could involve increasing the period of time between subsequent requests, or caching unsuccessful discovery attempts so those domains can be avoided in future. You may wish to send

a `User-Agent` header with a reference to the notification mechanism you are using so that recipient servers can find out more about the purpose of your requests.

Your application should also respect relevant cache control and retry headers returned by the target server.

5.2 Generic notifications

[LDN] provides a protocol for sending, receiving and consuming notifications which may contain any content, or be triggered by any person or process. Senders, receivers and consumers can all be on different domains, thus this meets the criteria for a federation protocol. This is a suitable notification mechanism when:

- Notifications need to be identifiable with their own URLs and exposed by the receiver for other applications to discover and re-use.
- Notifications are represented as a JSON-LD payload (ie. a 'fat ping').
- You need to advertise constraints on the type or contents of notifications accepted by a receiver.

[LDN] functionality is divided between *senders*, *receivers* and *consumers*. The endpoint to which notifications are sent is the `inbox`. Any resource (a user profile, blog post, document) can advertise its `inbox` so that it may be discovered through an HTTP `Link` header or the document body in any RDF syntax (including JSON-LD or HTML+RDFa). To this Inbox, senders make a `POST` request containing the JSON-LD (or other RDF syntax per [Accept-Post] negotation with the receiver) payload of the notification. The receiver returns a URL from which the notification data can be retrieved, and also adds this URL to a list which is returned upon a `GET` request to the Inbox. Consumers can retrieve this Inbox listing, and from there the individual notifications, as JSON-LD (optionally content negotiated to another RDF syntax). An obvious type of consumer is a script which displays notifications in a human-readable way.

> NOTE: LDP compatibility
>
> An existing [LDP] implementation can serve as an [LDN] receiver; publishers simply advertise any `ldp:Container` as the `inbox` for a resource.

The payload of notifications is deliberately left open so that LDN may be used in a wide variety of use cases. However, receivers with particular purposes are likely to want to constrain the types of

notifications they accept. They can do this transparently (such that senders are able to attempt to conform, rather than having their requests rejected opaquely) by advertising data shapes constraints such as [SHACL]. Advertisement of such constraints also allows consumers to understand the types of notifications in the Inbox before attempting to retrieve them. Receivers may reject notifications on the basis of internal, undisclosed constraints, and may also access control the Inbox for example by requiring an `Authorization` header from both senders and consumers.

[WebSub] *publishers* deliver content to their *hub*, and *hubs* to their *subscribers* using HTTP `POST` requests. The body of the request is left to the discretion of the sender in the first case, and in the latter case must match the Content-Type of and contain contents from the `topic` URL.

5.3 Activity notifications

[Activitypub] uses [LDN] to send notifications with some specific constraints. These are:

- The notification payload MUST be a single [ActivityStreams2] Activity.
- The notification payload MUST be compact JSON-LD.
- The receiver MUST verify the notification by fetching its source from the origin server.
- All notification `POST` requests are authenticated.

[Activitypub] specifies how to define the *target(s)* to which a notification is to be sent (a pre-requisite to [LDN] sending), via the [ActivityStreams2] audience targeting and object linking properties.

[Activitypub] also defines side-effects that must be carried out by the server as a result of notification receipt. These include:

- Creating, updating or deleting new objects upon receipt of `Create`, `Update` and `Delete` activities.
- Reversing the side-effects of prior activities upon receipt of the `Undo` activity.
- Updating specialised collections for `Follow`, `Like` and `Block` activities.
- Updating any other collections upon receipt of `Add` and `Remove` activities.
- Carrying out further delivery to propagate activities through the network in the case of federated servers.

> NOTE: The inbox property
>
> ActivityPub actor profiles are linked to their inboxes via the `https://www.w3.org/ns/activitystreams#inbox` property. This is an *alias* (in the AS2 JSON-LD context) for LDN's `http://www.w3.org/ns/ldp#inbox`. Applications using a full JSON-LD processor to parse these documents will see these terms as one and the same. Applications doing naive string matching on terms may wish to note that if you find an `ldp:inbox` it will accept `POST` requests in the same way as an `as:inbox`.

5.4 Mentioning

[Webmention] provides an API for sending and receiving notifications when a relationship is created, updated, or deleted between two documents by including the URL of one document in the content of another. It works when the two documents are on different domains, thus serving as a federation protocol. This is a suitable notification mechanism when:

- You have a document (source) which contains the URL of another document (target).
- The owner of the endpoint has access to view the source (so the request can be verified).
- The only data you need to send over the wire are the URLs of the source and target documents (ie. a 'thin ping').

If a Webmention is received and this is the first time this `source` and `target` combination has been seen by the receiver, this indicates a relationship between the `source` and `target` has been created. If the receiver has seen these values before, the receiver can fetch and parse the `source` to determine what has changed; either the content of the `source` has been updated (in which case the receiver can update local copies of the content, if it stored them in the first place), or the entire source has been deleted (expect a `410 Gone` response).

There are no constraints on the syntax of the source and target documents. Discovery of the [Webmention] endpoint (a script which can process incoming webmentions) is through a link relation (`rel="webmention"`), either in the HTTP `Link` header or HTML body of the target. This endpoint does not need to be on the same domain as the target, so webmention receiving can be delegated to a third party.

Webmentions are verified by the server dereferencing the source and parsing it to check for the existence of the target URL. If the target URL isn't found, the webmention MUST be rejected.

[Webmention] uses `x-www-form-urlencoded` for the source and target as parameters in an HTTP POST request. Beyond verification, it is not specified what the receiver should do upon receipt of a Webmention. What the webmention endpoint should return on a `GET` request is also left unspecified.

5.5 Delivery interop

This section describes how receiver implementations of either Webmention or LDN may create bridging code in order to accept notifications from senders of the other. This can also be read to understand how a sender of either Webmention or LDN should adapt their discovery and payload in order to send to a receiver of the other.

5.5.1 LDN to Webmention

Webmention receivers wishing to also accept LDN `POST`s at their Webmention endpoint MUST:

- Advertise the webmention endpoint via `rel="http://www.w3.org/ns/ldp#inbox"` in addition to `rel="webmention"` (in the `Link` header, HTML body or JSON body of a target).
- Accept `POST` requests with the Content-Type `application/ld+json`. Expect the body of the request to be:

  > EXAMPLE 1
  >
  > ```
  > {
  >   "@context": "http://www.w3.org/ns/webmention#",
  >   "@id": "",
  >   "source": { "@id": "https://waterpigs.example/post-by-barnaby" },
  >   "target": { "@id": "https://aaronpk.example/post-by-aaron" }
  > }
  > ```

  Use the `source->@id` and `target->@id` values as the `source` and `target` of the Webmention, and proceed with verification.
- If returning a `201 Created`, it MUST return a `Location` header with a URL from which the contents of the request posted can be retrieved. `202 Accepted` is still fine.
- Note than when verifying the source, there's a good chance you can request/parse it as RDF.

5.5.2 Webmention to LDN

LDN receivers wishing to also accept Webmentions to their Inbox MUST:

- Advertise the Inbox via `rel="webmention"` in addition to `rel="http://www.w3.org/ns/ldp#inbox"` (in the `Link` header, HTML body or JSON body of a target).
- Accept `POST` requests with a content type `application/x-www-form-urlencoded`. Convert these requests from:

> EXAMPLE 2
>
> ```
> source=https://waterpigs.example/post-by-barnaby&
> target=https://aaronpk.example/post-by-aaron
> ```

  to:

> EXAMPLE 3
>
> ```
> {
>   "@context": "http://www.w3.org/ns/webmention#",
>   "@id": "",
>   "source": { "@id": "https://waterpigs.example/post-by-barnaby" },
>   "target": { "@id": "https://aaronpk.example/post-by-aaron" }
> }
> ```

  and proceed per [LDN]; receivers MAY add other triples at their discretion.
- Receivers MUST return a `201 Created` with a `Location` header or `202 Accepted`.
- Receivers MUST verify the request by retrieving the source document and checking a link to the target document is present. If the Webmention is not verified, recievers MUST NOT keep it.

5.5.3 Webmention as AS2

A webmention may be represented as a persistent resource with [ActivityStreams2]. This could come in handy if a Webmention sender *mentions* a user known to be running an [Activitypub] federated server. In this case, the sender can use an AS2 payload and carry out delivery of the notification per ActivityPub/LDN.

> EXAMPLE 4
>
> ```
> {
>   "@context": "https://www.w3.org/ns/activitystreams#",
>   "type": "Relationship",
>   "subject": "https://waterpigs.example/post-by-barnaby",
>   "object": "https://aaronpk.example/post-by-aaron"
> }
> ```

A receiver or sender may want to augment this representation with the relationship between the two documents, and any other pertinent data. In the receiver's case, this could be gathered when they parse the source during the verification process. For example:

> EXAMPLE 5
>
> ```
> {
>   "@context": "https://www.w3.org/ns/activitystreams#",
>   "type": "Relationship",
>   "subject": {
>     "id": "https://waterpigs.example/post-by-barnaby",
>     "name": "Hi Aaron, great post."
>   },
>   "object": {
>     "id": "https://aaronpk.example/post-by-aaron",
>     "name": "Aaron's first post."
>   },
>   "relationship": "inReplyTo"
> }
> ```

6. Profiles

> ISSUE 1
>
> Stub, needs expansion

The subject of a profile document can be a person, persona, organisation, bot, location, … the type of the subject of the profile is not required. Each profile document MUST have a URL which SHOULD

return attributes of the subject of the profile; SHOULD return at least one link to a stream of content and MAY return content the subject has created. A JSON format MUST be available; other content types MAY be returned as well.

6.1 Relationships

Semantics and representation of personal relationships are implementation specific. This specification deals with relationships only when distribution of content is affected, for example if one user 'friending' another triggers a subscription request from the first user's server to the second. Lists of other relationships MAY be discoverable from a user profile, SHOULD be represented according to the ActivityStremas 2 syntax and MAY (and are likely to) use extension vocabularies as needed.

- **Activitypub:** When a server receives a `Follow` Activity in its `inbox`, the subject is added to a `Followers Collection`, which is discoverable from the subject's profile.

6.2 Authorization and access control

Servers may restrict/authorize access to content however they want?

- **ActivityPub:** see auth
- **Indieweb:** see private posts, private webmention
- **Solid:** see acl

A. Change Log

A.1 Changes from 4 May 2017 WD to this version

- Update document status of specifications.

A.2 Changes from 16 November 2016 WD to this version

- Update document status of progressing specs.
- s/PubSub/WebSub

- Add list of relationships between specific specs to Overview.
- Update AP streams info.
- Bring contents of Reading up to date.
- Remove references to external specs that aren't explicitly mentioned in a SWWG spec.
- Capture Create/Update/Delete under Publish header.
- Clean up text in Publish.
- Update Delivery for each spec and make section headings based on function rather than spec names.
- Add WebSub detail for Subscription.

A.3 Changes from 1 November 2016 WD to this version

- Update document status of recently published drafts.

A.4 Changes from 12 October 2016 WD to this version

- Updated WebSub since it was published as FPWD.

A.5 Changes from 23 August to 12 October 2016

- Added targeting and discovery section, including "backoff" strategy advice.

A.6 Changes from 3 June to 23 August 2016

- Add new WG drafts as they were published by the group: JF2, PTD, LDN, PuSH.
- Update the publication statuses of existing WG drafts as they advanced.
- Describe LDN in Delivery and Reading.
- Describe PuSH in Subscribing and Delivery.
- Delivery interop:
  - how Webmention senders/receivers and LDN senders/receivers may implement a bridge.
  - how a Webmention could be represented as an AS2 Relationship.

- Refactor Subscribing and Reading to prioritise WG specs and clarify relations between them.
- Editorial improvements to introductory and overview text.

B. References

B.1 Informative references

[Accept-Post]
The Accept-Post HTTP Header.. J. Arwe; S. Speicher; E. Wilde. IETF. Internet Draft. URL: http://tools.ietf.org/html/draft-wilde-accept-post

[Activitypub]
ActivityPub. Christopher Webber; Jessica Tallon. W3C. 5 December 2017. W3C Proposed Recommendation. URL: https://www.w3.org/TR/activitypub/

[ActivityStreams2]
ActivityStreams 2.0. James Snell; Evan Prodromou. W3C. W3C Candidate Recommendation. URL: http://www.w3.org/TR/activitystreams

[AS1]
JSON Activity Streams 1.0.. J. Snell; M. Atkins; W. Norris; C. Messina; M. Wilkinson; R. Dolin. http://activitystrea.ms. Unofficial. URL: http://activitystrea.ms/specs/json/1.0/

[h-entry]
h-entry. Tantek Çelik. microformats.org. Living Specification. URL: http://microformats.org/wiki/h-entry

[LDN]
Linked Data Notifications. Sarven Capadisli; Amy Guy. W3C. W3C Proposed Recommendation. URL: https://www.w3.org/tr/ldn/

[LDP]
Linked Data Platform 1.0. Steve Speicher; John Arwe; Ashok Malhotra. W3C. 26 February 2015. W3C Recommendation. URL: https://www.w3.org/TR/ldp/

[Micropub]
Micropub. Aaron Parecki. W3C. 23 May 2017. W3C Recommendation. URL: https://www.w3.org/TR/micropub/

[SHACL]
Shapes Constraint Language (SHACL). Holger Knublauch; Dimitris Kontokostas. W3C. 20 July 2017. W3C Recommendation. URL: https://www.w3.org/TR/shacl/

[Webmention]
Webmention. Aaron Parecki. W3C. W3C Recommendation. URL: http://www.w3.org/TR/webmention

[WebSub]
WebSub. Julien Genestoux. W3C. W3C Editor's Draft. URL: https://www.w3.org/TR/websub

↑