Social Web Protocols

W3C Editor's Draft 27 May 2016

This version:
https://w3c-social.github.io/social-web-protocols/respec.html
Latest published version:
http://www.w3.org/TR/social-web-protocols/
Latest editor's draft:
https://w3c-social.github.io/social-web-protocols/respec.html

Editor:
Amy Guy, University of Edinburgh

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 document is governed by the 1 September 2015 W3C Process Document.

Table of Contents

1. Overview

People and the content they create are the core ~~componants~~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 people's content;
- be notified when other people interact with their content;

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

~~This provides the~~ These components are core building blocks for interoperable social systems.

Each of these specifications 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,
- how 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: ~~Actiivty~~ActivityStreams, Activitypub, Micropub and Webmention. This document also describes Social Web Working Group specifications in context of W3C recommendations and ~~related~~relevant drafts from other communities outside of the Social Web Working Group~~ where relevant~~. 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.

This document is on track to become a Working Group Note to provide guidance on how and when to implement the various specifications produced.

1.2 Social Web Working Group Drafts

ActivityPub [SWWG WD]
: JSON-based APIs for reading, creating, updating, deleting, subscribing, delivery and profiles.

ActivityStreams 2.0 (AS2) [SWWG WD Core, SWWG WD Vocab]
: The syntax and vocabulary for representing social data in JSON and RDF.

Micropub [SWWG WD]
: Minimal form-encoded API for creating, updating and deleting.

Webmention [SWWG CR]
: Minimal form-encoded API for delivery.

1.3 Other specifications

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

Annotations Protocol [AnnoWG WD]
: Transport mechanisms for creating and managing annotations on the Web.

JF2~~jf2~~ [ED] (External with SWWG representation)
: A ~~complimentary~~complementary social vocabulary to AS2 for object-centric (as opposed to activity-centric) data.

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

Post Type Discovery [ED] (External with SWWG representation)

A protocol to bridge object-centric and activity-centric vocabularies.

Solid [ED] (External with SWWG representation)

Extensions to LDP for use in social applications

1.4 Requirements

The high level requirements according to the Social Web Working Group charter and the Social API Requirements.

	Vocabulary	**Syntax**	**Read**	**Create**	**Update**	**Delete**	**Subscription**	**Delivery**	**Profiles**
ActivityPub			X	X	X	X	X	X	X
ActivityStreams 2.0	X	X							
Micropub				X	X	X			
Webmention								X	

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

Content *MUST* be available as [*ActivityStreams2*] JSON and *MAY* additionally be served as an alternative syntax. If a URL does not return [*ActivityStreams2*] JSON, consumers *SHOULD* look for the JSON format via a `rel="alternate" type="application/activity+json"` link (which could be to a different domain, for third-party services which dynamically generate [*ActivityStreams2*] JSON on behalf of the publisher). [*ActivityStreams2*] content *MUST* be served with the Content-Type `application/activity+json` or, if necessary for JSON-LD extended implementations, `application/ld+json; profile="http://www.w3.org/ns/activitystreams"`.

Content *MUST* be described using the [*ActivityStreams2*] vocabulary, and *MAY* use other vocabularies in addition or instead, per the [*ActivityStreams2*] extension mechanism.

[*ActivityStreams2*] 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 [*ActivityStreams2*].

[*Activitypub*] exposes all content as [*ActivityStreams2*] JSON.

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.

[***Activitypub***] specifies four feeds that *MUST* be accessible from a profile via the following properties:

- `inbox`: A reference to an [*ActivityStreams2*] collection comprising of all the messages received by the actor.
- `outbox`: An [*ActivityStreams2*] collection comprising of all the messages produced by the actor.
- `following`: An [*ActivityStreams2*] collection of the actors that this actor is following.
- `followers`: An [*ActivityStreams2*] collection of the actors that follow this actor.

[*Activitypub*] also specifies a further property for accessing additional feeds, which *MAY* be included in a profile:

- `streams`: A list of supplementary Collections which may be of interest.

Issue 1

Activitypub discovery mechanism for this to be specified, likely link relations per issue 50.

[*Activitypub*] specifies special ~~behaviour~~behavior 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`).

3. Creating content

Content generated through a client (such as a web form, mobile app, 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). Clients and servers may independently support creating, updating and deleting; there are no dependencies between them. Authentication and authorization for creating content is left out of scope.

[*Activitypub*] clients discover the `outbox` of the authenticated user from their JSON profile, then make an HTTP POST request with an appropriate [*ActivityStreams2*] ~~Activity~~activity as a JSON payload. [*Activitypub*] requires use of the [*ActivityStreams2*] vocabulary and syntax, which may be extended with JSON-LD. The URL of the created resource is generated at the discretion of the server. 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).

[*Micropub*] clients discover the Micropub endpoint from the current user's homepage (likely entered by the user as part of authentication) from a `rel="micropub"` link, then 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. Micropub requires use of the Microformats 2 [*h-entry*] vocabulary, as well as a set of reserved attributes as commands to the server.~~;~~ Any additional key names sent outside of this vocabulary 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.

[*LDP*] servers are written to by sending an HTTP POST request to a `Container` (any resource on the server which returns `Link: <http://www.w3.org/ns/ldp#Container>; rel="type"`), containing an RDF payload with attributes of the object being created. An identifier for the created resource is generated at the discretion of the server~~, and t~~The URL is created by appending the identifier to the URL of the `Container` that was posted to. [*Solid*] and [*AnnotationsProtocol*] both use slightly altered [*LDP*] to create content; [*Solid*] restricts use to only Basic Containers, while the [*AnnotationsProtocol*] specifies the [*AnnotationsVocab*] and primary syntax as JSON-LD. This is an appropriate protocol to use when:

- Your data is represented as RDF.
- Clients and servers are vocabulary agnostic.
- You prefer a more RESTful API.

3.1 Updating

~~Updating c~~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 replace the complete object by sending an HTTP POST request containing an

[*ActivityStreams2*] `Update` activity to the `outbox` of the authenticated user, where the `object` is a nested object, including an `id`, which serves as the replacement.

Issue 2

ActivityPub currently only supports updates of a whole object at once. Issue 85 for partial updates.

[*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 (discovery described in creating). `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.

[*LDP*] clients perform replacement-style updates by sending an HTTP PUT request to the URL of the object to be updated. Partial-style updates are performed with an HTTP PATCH request.

3.2 Deleting

~~Deleting c~~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 replace* the `object` of this activity with a tombstone, and return a `410 Gone` status code 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 (discovery described in creating).

[*LDP*] deletes are performed by sending an HTTP DELETE request to the URL of the object to be deleted.

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*. A server may also receive notifications of changes to content it has *not subscribed* to: see delivery.

Issue 3

Working group has yet to discuss this in depth, subject to significant change.

[*Activitypub*] servers distribute the activities of an actor by sending POST requests containing the activities as JSON payloads to the `inbox` of all actors in an actor's `Followers` collection. For subscription requests, special ~~behaviour~~behavior is defined for servers when an [*ActivityStreams2*] `Follow` activity is appears in an actor's `outbox` or `inbox`.

- For `outbox` (an outgoing subscription request): The activity is delivered as usual to the `object`'s `inbox`, and additionally the `object` is added to the actor's `Following` ~~C~~collection.
- For `inbox` (an incoming subscription request): The recipient's (the `object` of the `Follow` activity) server adds the `actor` of the `Follow` activity to the recipient's `Followers` collection, to which ~~subsequently~~ all new activities of the recipient are subsequently delivered ~~to~~.

~~It~~ This works when the publisher and subscriber are on different domains, thus serving as a federation protocol. This is a suitable subscription mechanism when:

- The objects and streams being subscribed to are described with [*ActivityStreams2*].
- The subscriber wants to request updates from a specific actor (rather than objects, streams or threads, see issue 80.
- You need to send or receive the whole requested resource over the wire (ie. a 'fat ping').
- You need to send or receive JSON payloads only.
- Subscriptions requests and fulfillment are handled ~~serverside~~server-side.
- The publisher is aware of who has subscribed; this is a *push* mechanism.

[*Solid*] uses websockets to allow ~~clientside~~client-side applications to subscribe to changes in objects ([*LDP*] Resources) and streams ([*LDP*] Containers). The websocket endpoint is advertised in the `Updates-Via` HTTP Link Header. The subscriber sends the keyword `sub` followed by an empty space and then the URI of the resource, to the target's websockets URI. The target's server sends a websockets message containing the keyword `pub`, followed by an empty space and the URI of the resource that has changed, whenever there is a change. It works when the publisher and subscriber are on different domains, thus serving as a federation protocol. This is a suitable subscription mechanism when:

- You send live updates over websockets.
- You can send and receive on the URL of the requested resource over the wire (ie. a 'thin ping').
- Subscriptions are requested ~~clientside~~client-side and fulfilled ~~serverside~~server-side.
- The publisher is aware of who has subscribed; this is a *push* mechanism.

[*Webmention*] can be deployed as a subscription mechanism through the [*Salmentions*] extension. Documents contain links to the sources of received webmentions along with the content. Every time the document is updated, including when new incoming webmentions are added, the server (re-)sends webmentions to every link. Thus, all documents which originally sent a webmention receive a webmention back, enabling them to parse the document for changes, and detect new replies in a thread, as well as changes to the original document. It works when the publisher and subscriber are on different domains, thus serving as a federation protocol. This is a suitable subscription mechanism when:

- The subscriber wants to subscribe to a specific thread or object.
- The subscriber has a document which links to that thread or object, and has sent a [*Webmention*]

about it.
- The publisher displays links to the sources of received webmentions with their content.
- The subscriber is linked to a document and wants to be notified of further changes ~~to a document linked to~~.
- The subscriber has implemented [*Webmention*] receiving.
- The publisher is aware of who has subscribed; this is a *push* mechanism.

Other options yet to be debated...

- **Web Push Protocol**: The subscriber follows the `urn:ietf:params:push` link relation to the target's Push Service, and then Subscribes for Push Messages
- **PubSubHubbub**: The subscriber discovers the target's hub, and sends a form-encoded `POST` request containing values for `hub.mode` ("subscribe"), `hub.topic` and `hub.callback`. When the target posts new content, the target's server sends a form-encoded `POST` to the hub with values for `hub.mode` ("publish") and `hub.url` and the hub checks the URL for new content and `POST`s updates to the subscriber's callback URL. (*See PuSH 0.4 and How To Publish And Consume PuSH*)
- Pull, ie. just reading an [*ActivityStreams2*] feed.

Note

Nothing should rely on implementation of a subscription mechanism.

5. Delivery

A user or application may wish to push a notification to another user, for example because they have linked to (replied, liked, bookmarked, reposted, …) their content or linked to (tagged, addressed) the user directly, or to make the recipient aware of a change in state of some document or resource on the Web.

[*Webmention*] provides an API for sending and receiving notifications when a relationship is created between two documents. 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 links to another document (target).
- The owner of the target has access to view the source (so that their webmention endpoint can parse the source and check the claimed link exists).
- The only data you need to send over the wire are the URLs of the source and target documents (ie. a 'thin ping').

Discovery of the [*Webmention*] endpoint (a script which can process incoming webmentions) is through a link relation (`rel="webmention"`), either in the HTTP Header or 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.

[*Webmention*] uses `x-www-form-urlencoded` for the source and target as parameters in an HTTP POST request. This means a webmention can be triggered from a web form or from the commandline with `curl`. Integration with existing applications is straightforward through translating this form-encoded data into JSON, RDF (namespace), or another preferred syntax at the recievers end if necessary. There are no constraints on the syntax of the source and target documents.

[*Webmention*] can be extended by adding additional parameters to the POST request; applications must have prior agreement in order to understand them, and there is potential for collision of extension terms.

[*Activitypub*] provides a JSON-based API for sending and receiving [*ActivityStreams2*] activities as notifications. It works when the two documents are on different domains, thus serving as a federation protocol. This is a suitable notification mechanism when:

- You describe the notification with [*ActivityStreams2*].
- You need to send or receive the whole notification over the wire (ie. a 'fat ping').
- You need to send or receive JSON payloads only.
- You need to add additional properties to your notifications unambiguously with JSON-LD.
- You need to send the same notification to multiple users from one activity.

The endpoint which processes incoming notifications is a user's [*Activitypub*] `inbox`, discoverable from the target user's profile. The recipient user may be addressed directly (through `object`, `target`, `to`, `cc`, and/or `bcc`), and/or inferred as the owner of another resource which is addressed (through `object`, `target` and/or `inReplyTo`).

An [*Activitypub*] server delivers the entire activity to all discovered endpoints as a JSON payload of an HTTP POST request.

An [*Activitypub*] server receiving such an activity must verify the notification is accurate (by fetching the source, or some prior trust arrangement with the sender, eg. through an authentication token or digital signature).

[*Activitypub*] delivery can be extended through the [*ActivityStreams2*] extension mechanism, with JSON-LD.

Issue 4

Add Solid Notifications for pure RDF usage.

Note

Note: 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.

6. Profiles

Issue 5

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?

- **ActivityPump:** see auth
- **Indieweb:** see private posts, private webmention
- **SoLiD:** see acl

A. References

A.1 Informative references

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

[ActivityStreams2]
James Snell; Evan Prodromou. W3C. *ActivityStreams 2.0*. W3C Working Draft. URL: http://w3.org/TR/activitystreams

[Activitypub]
Christopher Webber; Jessica Tallon; Owen Shepherd. W3C. *ActivityPub*. 28 January 2016. W3C Working Draft. URL: http://www.w3.org/TR/activitypub/

[AnnotationsProtocol]
Reference not found.

[AnnotationsVocab]
Reference not found.

[LDP]
Steve Speicher; John Arwe; Ashok Malhotra. W3C. *Linked Data Platform 1.0*. 26 February 2015.

W3C Recommendation. URL: http://www.w3.org/TR/ldp/

[Micropub]
Aaron Parecki. W3C. *Micropub*. 4 May 2016. W3C Working Draft. URL: http://www.w3.org/TR/micropub/

[Salmentions]
Ben Roberts, Tantek Çelik. IndieWebCamp. *Salmentions*. Living specification. URL: http://indiewebcamp.com/Salmention

[Solid]
Andrei Sambra. http://solid.mit.edu. *Solid*. W3C Editor's Draft. URL: https://github.com/solid/solid-spec

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

[h-entry]
Reference not found.