Hubs

A White Paper from Rebooting the Web of Trust III Design Workshop

by Daniel Buchner, Wayne Vaughan, and Ryan Shea

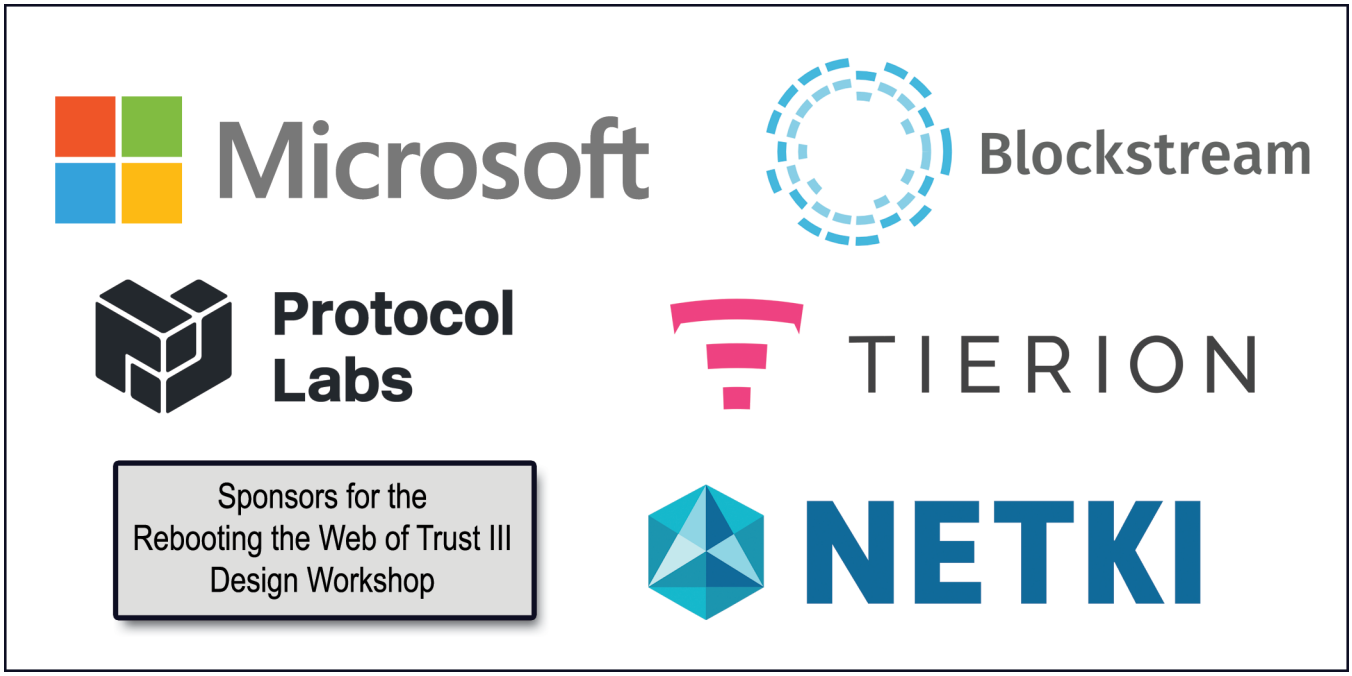
Hubs let you securely store and share data. A Hub is a datastore containing semantic data objects at well-known locations. Each object in a Hub is signed by an identity and accessible via a globally recognized API format that explicitly maps to semantic data objects. Hubs are addressable via unique identifiers maintained in a global namespace.

SINGLE ADDRESS FOR MULTIPLE HUB INSTANCES

A single entity may have one or more instances of a Hub, all of which are addressable via a URI-routing mechanism linked to the entity's identifier. Hub instances sync state changes, ensuring the owner can access data and attestations from anywhere, even when offline.

SYNCING DATA TO MULTIPLE HUBS

Hub instances must sync data without requiring master-slave relationships or forcing a single implementation for storage or application logic. This requires a shared replication protocol for broadcasting and resolving changes. CouchDB (http://docs.couchdb.org/en/2.0.0/replication/protocol.html), an open source Apache project, will be the data-syncing protocol Hubs must implement. It features an eventually consistent, master-master replication protocol that can be decoupled from the default storage layer provided by CouchDB.



WELL-KNOWN URIS

Existing web servers need to interact with Hubs. We are using the IETF convention for globally defined resources that predictably reside at well-known locations as detailed in RFC 5785 well-known URIs (https://tools.ietf.org/html/rfc5785) and the well-known URI directory (https://www.ietf.org/assignments/well-known-uris/well-known-uris.xml). Hubs are accessible via the path: /.well-known/identity/:id, wherein the last segment of the path is the target ID for the identity you wish to interact with.

API ROUTES

Each Hub has a set of top-level API routes:

`/.well-known/identity/:id/`*`profile`*

> ➜ The owning entity's primary descriptor object (schema agnostic).

`/.well-known/identity/:id/`*`permissions`*

> ➜ The access control JSON document

`/.well-known/identity/:id/`*`messages`*

> ➜ A known endpoint for the relay of messages/actions to the identity owner

`/.well-known/identity/:id/`*`stores`*

> ➜ Scoped storage space for user-permitted external entities

`/.well-known/identity/:id/`
`collections/:context`

> ➜ The owning entity's identity collections (access limited)

Hub Profile Objects

Each Hub has a `profile` object that describes the owning entity. The profile object should use the format of the schema object that best represents the entity. Here is an example of using the Schema.org `Person` schema to express that a hub belongs to a person:

```
{
    "@context": "http://schema.org",
    "@type": "Person",
    "name": "Daniel Buchner",
    "description": "Working on decentralized identity at Microsoft",
    "website": [
      {
        "@type": "WebSite",
        "url": "http://www.backalleycoder.com/"
      }
    ],
    "address": {
      "@type": "PostalAddress",
      "addressLocality": "Los Gatos, CA"
    }
}
```

Permissions

Agents are external parties that can access and modify Hub data. Hub owners can set permissions in a ACL JSON document, which you can learn more about via the ACL documentation and examples (https://github.com/decentralized-identity/acl/blob/master/examples/basic.json). This access control document designates:

- What factors can be used for authentication and modification of Hub data
- What data Agents have access to
- Which Agents are provided with a `store`

Messages

The `messages` open endpoint receives objects signed by other identities. Messages are not

constrained to the simple exchange of human-to-human communications. Rather, they are intended to be a singular, known endpoint where identities can transact all manner of messaging, notifications, and prompts for action.

Here is a list of examples to better understand the range of use-cases this endpoint is intended to support:

- Human user contacts another with a textual messages
- Service prompts a human to sign a document
- IoT device sends a notification to a human user about its state

The endpoint location for message objects shall be:

`/.well-known/identity/:id/messages/`

The encapsulating format for message payloads shall be:

http://schema.org/Message

If the intent of your message is to prompt the receiving Hub to perform a certain semantic activity, you can pass an Action object (http://schema.org/Action) via the Message's `potentialAction` property.

Stores

Stores are collections of identity-scoped data storage. Stores are addressable via the `/stores` top-level path, and keyed on the entity's decentralized identifier. Here's an example of the path format:

`/.well-known/identity/:id/stores/`*`ENTITY_ID`*

The data shall be a JSON object and should be limited in size, with the option to expand the storage limit based on user or provider discretion. Stores are not unlike a user-sovereign entity-scoped version of the W3C DOM's origin-scoped `window.localStorage` API.

Collections

Collections provide a known path for accessing standardized, semantic objects across all hubs, in way that asserts as little opinion as possible. The full scope of an identity's data is accessible via the following path

`/.well-known/identity/:id/collections/:context`, wherein the path structure is a 1:1 mirror of the schema context declared in the previous path segment. The names of object types may be cased in various schema ontologies, but hub implementations should always treat these paths as case insensitive. Here are a few examples of actual paths and the type of Schema.org objects they will respond with:

`/.well-known/identity/:id/`
`collections/schema.org:Event`

➜ http://schema.org/Event

`/.well-known/identity/:id/`
`collections/hl7.org:Device`

➜ https://www.hl7.org/fhir/device.html

`/.well-known/identity/:id/`
`collections/schema.org:Photograph`

➜ http://schema.org/Photograph

Data Portability

All Hub data associated with the identity must be portable. Transfer of a hub's contents and settings between environments should be seamless, without loss of data or operational state, including the permissions that govern access to identity data.

REQUEST/RESPONSE FORMAT

The REST API uses [JSON API's specification][2773b365 for request, response, and query formats, and leverages standard schemas for encoding stored data and response objects. Given the nature of the responses, only the Top-Level properties are in scope for this utilization. Requests should be formatted in accordance with the JSON API documentation: http://jsonapi.org/format/#fetching. The `Content-Type` and `Accept` header parameters must be set to `application/vnd.api+json`. This approach maximizes the use of existing standards and open source projects.

Authentication

The process of authenticating requests from the primary user or an agent shall follow the FIDO and Web Authentication specifications. These specifications may require modifications in order to support challenging globally known IDs with provably linked keys.

GET Requests

The REST routes for fetching and manipulating identity data should follow a common path format that maps 1:1 to the schema of data objects being transacted. Here is an example of how to send a `GET` request for an identity's Schema.org formatted music playlists:

`/.well-known/identity/jane.id/collections/schema.org:`*`MusicPlaylist`*

Requests will always return an array of all objects - *the user has given you access to* - of the related Schema.org type, via the response object's `collections` property, as shown here:


```
{
  "links": {
    "self": "/.well-known/identity/jane.id/collections/schema.org:MusicPlaylist"
  },
  "data": {
    "controls": {
      "4n93v7a4xd67": {
        "key": "...",
        "cache-intent": "full"
      },
      "23fge3fwg34f": {
        "key": "...",
        "cache-intent": "full"
      },
      "7e2fg36y3c31": {
        "key": "...",
        "cache-intent": "full"
      }
    },
    "payload": [{
      "@context": "http://schema.org",
      "@type": "MusicPlaylist",
      "@id": "4n93v7a4xd67",
      "name": "Classic Rock Playlist",
      "numTracks": 2,
      "track": [{
          "@type": "MusicRecording",
          "@id": "23fge3fwg34f",
          "byArtist": "Lynard Skynyrd",
          "duration": "PT4M45S",
          "inAlbum": "Second Helping",
          "name": "Sweet Home Alabama",
          "permit": "/.well-
known/identity/jane.id/collections/schema.org:Permit/ced043360b99"
        },
        {
          "@type": "MusicRecording",
          "@id": "7e2fg36y3c31",
          "byArtist": "Bob Seger",
          "duration": "PT3M12S",
          "inAlbum": "Stranger In Town",
          "name": "Old Time Rock and Roll",
          "permit": "/.well-
```

```
known/identity/jane.id/collections/schema.org:Permit/aa9f3ac9eb7a"
        }]
    }]
  }
}
```

POST Requests

POSTs are verified to ensure two things about the requesting party: 1) They are the decentralized identity they claim to be; and 2) They are authorized (as specified in the ACL JSON document) to write data to a specified route.

Addition of new data objects into a collection must follow a process for handling and insertion into storage:

1. The new objects must be assigned with an `@id` property
2. The Hub instance must keep an associated record that maps object IDs to the controls set for each. These control properties include:
- `key`: the symmetrical public key used to encrypt the object, which Hubs and entities use to reencrypt
- `cache-intent`: `full` | `light` | `min`
3. The object must be encrypted with the symmetrical key of the entities that have read privileges, as specified in the ACL JSON document.
4. The object shall be inserted into the Hub instance that is handling the request.
5. Upon completing the above steps, the change must be synced to the other Hub instances.

Query Filter Syntax

The Hub spec does not mandate specific storage and search implementations. For the purposes of interoperability and developer ergonomics hubs must accept a common search and filtering syntax regardless of the underlying implementation choice.

To avoid the introduction of a new syntax, we feel Apache Lucene's query filtering syntax balances the desire to select an option with broad, existing support, and the flexibility and expressiveness developers demand.

Filters can be applied via the `filter` parameter of your queries. Additionally, filters are used to enable more granular permissioning - see the ACL spec document for more info.

Additional Credits

Authors: Daniel Buchner, Wayne Vaughan, and Ryan Shea

About Rebooting the Web of Trust

This paper was produced as part of the **Rebooting the Web of Trust III** design workshop. On October 19th through October 21st, 2016, over 40 tech visionaries came together in San Francisco, California to talk about the future of decentralized trust on the internet with the goal of writing 3-5 white papers and specs. This is one of them.

Workshop Sponsors: Blockstack, Microsoft, Netki, Protocol Labs, Tierion

Workshop Producer: Christopher Allen

Workshop Facilitators: Christopher Allen and Brian Weller, additional paper editorial & layout by Shannon Appelcline, and additional support by Kiara Robles and Marta Piekarska.

What's Next?

The design workshop and this paper are just starting points for Rebooting the Web of Trust. If you have any comments, thoughts, or expansions on this paper, please post them to our GitHub issues page:

https://github.com/WebOfTrustInfo/rebooting-the-web-of-trust-fall2016/issues

The next Rebooting the Web of Trust design workshop is scheduled for Spring 2017 in Paris, France. If you'd like to be involved or would like to help sponsor these events, email:

ChristopherA@LifeWithAlacrity.com