thiss.io clients Documentation

Release 1.0.5

Leif Johansson

Mar 25, 2021

Contents:

This is the client API package for services built on the thiss.io software - eg use.thiss.io or service.seamlessaccess.org. The package contains a set of javascript APIs for talking to a shared persistence service for Identity Provider selection and an implementation of the SAML Identity Provider Protocol in javascript.

CHAPTER 1

Introduction

The Identity Selector Software (thiss.io) is an implementation of an identity selector supported by the Coalition for Seamless Access. It implements a discovery service using the RA21.org recommended practices for discovery UX.

The Identity Selector Software suite is a front-channel identity selector for distributed identity ecosystems aka Federated Identity Management. The objective is to simplify the process of choosing an "identity provider" by having the browser remember the users choice in browser local store. Currently the system has been used for large-scale SAML-based identity federations but there are no intrinsic dependencies to SAML as such and the system could be easily adapted to other protocols that follow the common pattern of federation by relying on redirecting the user to an authentication provider of some sort.

The system was designed with privacy as the number one focus. No information is shared with the relying party during the identity provider choice process. This is ensured by relying on the browser security model and judicious use of inter-domain communicatiton using post-message.

This package (thiss-ds-js) contains the parts needed to write a client that talks to an instance of a thiss-js service (eg use.thiss.io or service.seamlessaccess.org).

1.1 Architecture

The Identity Selector Software (thiss.io) is a set of front-channel (aka browser-based) cross-domain APIs using post-message (built using the post-robot package):

- A persistence API that allows store & retrieval of information about the last N (3) identity providers used to authenticate a user. Unlike simlilar project (eg google account chooser) the information stored does not include any PII (eg email-addresses) but only identifies the identity provider used in a way consistent with the authentication protocol used.
- A discovery API that implements SAML identity provider discovery layered on top of the persistence API

Both of these APIs have a *server* and a *client* component. The client components can be found in this library and can be imported (using npm) in existing projects. The server components can be found in the thiss-js repository. The server component is implemented as javascript running in an iframe fetched from a service URI. This ORIGIN (in the sense of the w3c security model) protects access to the browser local store and ensures that the calling page only has

access to the intended API. The calling page (aka the client) is responsible for initializing the iframe but after this no longer has any control over the code executing inside it. The *server* iframe, while executing in the client browser, is therefore sandboxed from the calling page.

The persistence API is completely protocol agnostic eg has no dependency on SAML, all of which are in the discovery API. Future versions are expected to provide similar APIs for OpenID Connect supporting OpenID connect federation and possibly other protocols.

A relying party (aka SP) will typically not integrate directly with these APIs but will rely on higher-level services built using these APIs, eg those provided by and instance of thiss-js such as use.thiss.io or service.eamlessaccess.org

1.2 Audience

This documentation is targeted at developers who want to build their own identity provider selector service on top of the low-level APIs instead of relying on the highlevel services provided by an instance of thiss-js. Readers are assumed to have a working knowledge of front channel development and associated tooling (eg webpack, babel, npm etc).

CHAPTER 2

Installing thiss-ds-js

Install via npm is straight-forward:

```
# npm install [--save] @theidentityselector/thiss-ds
```

The thiss-ds package supports both CommonJS-style and ES6 import aswell as old-school CDN delivery:

CommonJS:

```
var thiss = require("thiss-ds.js");
```

ES6-style

```
import {DiscoveryService} from "thiss-ds";
import {PersistenceService} from "thiss-ds";
```

CDN (thanks to unpkg.com)

```
<script src="//unpkg.com/browse/@theidentityselector/thiss-ds" />
```

CHAPTER 3

Using thiss-ds

3.1 Overview

There are two main APIs - a highlevel DiscoveryService API (which in hindsight really should have been called a discovery client API but ...) and a lowlevel PersitenceService. The job of the PersistenceService is to keep track of previous IdP choices. The data is stored in namespace browser local storage (using the js-storage package). The namespace is called the "context" below (more about how contexts work later).

The DiscoveryService class is essentially a reference to an instance of the PersistenceService and a metadata query service (MDQ for short) which relies on fetch to retrieve JSON-objects that represent known identity providers. This class also contains some utility methods providing a way to implement SAML Identity Provider Discovery v1.0.

Create an instance of the DiscoveryService object thus (where my_context is a string or unkown which makes the instance default to the default (or global) context:

```
var ds = new DiscoveryService(function(entity_id) {
          // return json representation of entity_id
      },
      'https://use.thiss.io/ps',
      my_context):

var ds = new DiscoveryService('https://md.thiss.io/entities/',
     'https://use.thiss.io/ps',
     my_context):
```

Calling the metadata lookup service with the entityID of an IdP returns a Promise that resolves (if the lookup was successful) to a JSON object (or undefined) that represents the IdP. The "schema" of the JSON is based in large parts on the classical discojson format and is explained below.

```
ds.mdq('https://idp.unitedid.org/idp/shibboleth').then(entity => {
  // do something with entity
});
```

In order to implement a simple SAML discovery response:

```
ds.saml_discovery_response(entity_id, persist)
```

This call first calls the persistence service to record the users choice (possibly refreshing metadata using the mdq first) and then returns a SAML identity provider discovery protocol response. This could for instance called from an "onclick" method in a UI. In a typical implementation the mdq method is used to lookup metadata which is then used to drive the UI. When the user selects a particular IdP the above call persists the users choice and returns the discovery response via the SAML identity provider discovery protocol (which is essentially just a redirect) by setting the window.location.href to the assembled return URL.

The second parameter (persist) is a boolean (which by default is set to true) which if true causes the metadata returned from the lookup to be persisted in browser local store via the persistence service.

For situations where it is necessary to have more control over how the discovery response is created the following call is available:

```
ds.do_saml_discovery_response(entity_id, persist).then(entity => {
 // ... do something with entity JSON
});
```

In this case the caller has to process the Promise returned by the call and assemble and process a discovery response. This could be useful for implementing extensions to SAML discovery or even as a basis for federation-enabled OpenIDC discovery.

Another use-case for do_saml_discovery_response is to "pin" an IdP choice based on some other process (other than a UI). For instance it may be known that users with access to an intranet site by definition should have a certain IdP pre-selected. In this case a call to do_saml_discovery_response with a static entity_id acts as a way to "pin" that IdP. In combination with UX that displays previous user "choices" this means that intranet users would never have to visit a (possibly complex) IdP search UX.

Because this is an important use-case an alias for ds.do_saml_discovery_response called ds.pin is available:

```
ds.pin(enterprise_idp_entity_id);
```

Note that the mdq implementation provided to the instance of DiscoveryService must be able to resolve this entity_id.

Finally the remove method removes the chose entity_id from the persistence-service if present.

```
ds.remove(entity_id)
```

3.2 Metadata JSON schema

The following fields are currently used:

```
{
  "entity_icon": "a data: URI for direct inclusion in html",
  "descr": "a short description suitable for display inline",
  "title": "the name of the identity provider - primary display for users",
  "name_tag": "an upper-case SLUG - typically based on the non-TLD/ccTLD part of the
↪domain",
  "type": "idp or sp",
  "auth": "saml|opendic|other",
  "entity_id": "the entityID of the IdP",
  "hidden": "if hide-from-discovery is set",
  "scope": "a comma-separated list of domains/scopes associated with the IdP",
  "id": "sha1 ID as specified by the MDQ spec"
}
```

3.3 Context

The PersistenceService is initialized with a context. The context is a namespace string passed with each call to the API. The context is used to differentiate the persistence local storage to avoid overlap. This may seem counter intutitive as the point of the thiss.io persistence service is to share IdP choices among several services. However the goal is really to share IdP choice among services that share a common view of metadata. In order to make it possible for service to have overlapping or even conflicting metadata "views" the context can be used to differentiate between "metadata domains". A contexts may be protected in a given persistence service ORIGIN so some operations (such as removing a choice) may fail. Failures are always handled as rejected Promises and should be handled by the caller in the appropriate way.

CHAPTER 4

Example: Metadata lookup

To illustrate how to use the DiscoveryService API we will create and walk through a tiny discovery service. This represents the smallest (and arguably least practially useful) possible SAML-based DS. While not useful as such it can serve as a starting point for further work. This example can be found as demo.js and index.html in the thiss-ds-js distribution.

Start by creating an empty directory on a webserver and create in it an index.html with the following simple markup:

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/@theidentityselector/thiss-ds"></script>
  <script src="demo.js"></script>
  <title>Tiny Discovery Service</title>
</head>
<body>
  <main role="main">
    <h1>Search for your identity provider!</h1>
    <p>Enter an entityID and hit the lookup button</p>
    <input role="button" id="lookup" value="https://idp.unitedid.org/idp/shibboleth"␣
↪type="text"/>
    <input type="button" id="submit" value="lookup"/>
    <div id="info"></div>
  </main>
</body>
</html>
```

The only thing that goes on here is a couple of fields for letting the js code interact with the user. The real meat goes on in demo.js which we'll create next. Note that we load the thiss-js package from the unpkg CDN first.

Now in demo.js (in the same directory) put this:

```
function entity_to_dl(entity,dl) {
   let doc = window.document;
   Array.from(dl.children).forEach(c => o.removeChild(c));
   Object.getOwnPropertyNames(entity).forEach(k => {
```

(continues on next page)

(continued from previous page)

```
      let v = entity[k];
      let dt = doc.createElement("dt");
      dt.textContent = k;
      dl.appendChild(dt);
      let dd = doc.createElement("dd");
      if (k === "entity_icon") {
         let img = doc.createElement("img")
         img.setAttribute("src", v);
         dd.appendChild(img);
      } else if (typeof v === 'object') {
         let inner = doc.createElement("dl");
         entity_to_dl(v, inner);
         dd.appendChild(inner);
      } else {
         dd.textContent = v;
      }
         dl.appendChild(dd);
   })

}

window.onload = function() {
   let ds = new thiss.DiscoveryService("https://md.thiss.io/entities/",
                                       "https://use.thiss.io/ps/","test")
   let doc = window.document;
   let o = doc.getElementById("info");
   let i = doc.getElementById("lookup");
   let s = doc.getElementById("submit");
   let dl = doc.createElement("dl");
   o.appendChild(dl);

   s.onclick = function() {
      ds.mdq(i.value).then(entity => {
         if (!entity) { entity = {"error": "not found"} }
         entity_to_dl(entity, dl);
      });
   }
}
```

The function `entity_to_dl` is a simple callback that displays information about an entity. The onload callback sets up a click handler on the submit button that calls the mdq service and after completion of the Promise in the `then` function call, uses `entity_to_dl` to show the entity. In a real discovery service the user would of course be redirected to the idp for authentication so the `entity_to_dl` would be replaced by a redirect of some sort. In a real DS the user would also not be expected to type in an entityID of their IdP but would be presented with some UX that allows the user to identify the right IdP in some other way.

CHAPTER 5

Release Notes

5.1 Version 1.0.13

- Support for turning off persistence in DiscoveryService.do_saml_discovery_response

CHAPTER 6

Persistence API Reference

6.1 Classes

class PersistenceService(*:*)

A client for the thiss.io persistence service. The Persistence service methods all follow the same pattern - a call that returns a Promise that resolves to one or more item Objects. An item is an Object with two properties:

The constructor initializes an iframe in window.document setting src to the URL provided to the constructor.

Arguments

- **:** (*last_refresh*) – An entity object (discojson schema)
- **:** – A timestamp when this entity was last updated (used)

PersistenceService.**PersistenceService**

The constructor initializes an iframe in window.document setting src to the URL provided to the constructor.

PersistenceService.**entities**(*str*)

Returns 0-3 of the most recently used entities as a list of item Objects. Be sure to examine the last_time property to make sure the provided entities are "recent" enough to be used.

Arguments

- **str** (*context*) – The context to write to

Returns **Promise** – A Promise that resolves to a list of items on success.

PersistenceService.**entity**(*string*)

Fetch an entity from the context.

Arguments

- **string** (*entity_id*) – The entityID of the item to be removed.

Returns **Promise** – A Promise that resolves to an item containing the entity on success.

PersistenceService.**remove**(*string*)
Remove an entity from the context.

Arguments

- **string** (*entity_id*) – The entityID of the item to be removed.

Returns **Promise** – A Promise that resolves to nothing on success.

PersistenceService.**update**(*str*, *Object*)
Update an an entity object in browser local store tied to the ORIGIN of the service URL.

Arguments

- **str** (*context*) – The context to write to
- **Object** (*entity*) – A js object representing an entity. Uses the discojson schema.

Returns **Promise** – A Promise that resolves to an item containing the provided entity on success.

6.2 Examples

- Example using PersistenceService.entities
- Example using PersistenceService.update

CHAPTER 7

DiscoveryService API Reference

7.1 Functions

json_mdq_get (*string*)
An MDQ client using fetch (https://fetch.spec.whatwg.org/). The function returns a Promise which must be resolved before the object can be accessed.

Arguments

- **string** (*mdq_url*) – an entityID (must be urlencoded) or sha1 id
- **string** – a URL of an MDQ service incl trailing slash - eg https://md.thiss.io/entities/

Returns **Promise** – a Promise resolving an Object observing the discojson schema

parse_qs (*paramsArray*)
Parse an array of querystring components into an Object

Returns an object with each k,v-pair as properties.

ds_response_url (*Object*)
Create a SAML discovery service protocol response URL from the entity_id property of the entity object and the return and returnIDParam (if present) of the params object. Combine with a base URL to form a full discovery service response.

Arguments

- **Object** (*params*) – a discojson entity
- **Object** – an object from which 'returnIDParams' and 'return' will be used

Returns **string** – a query string

7.2 Classes

class DiscoveryService (*function (entity_id) {}|string*, *string|PersistenceService*, *string*)

A DiscoveryService class representing the business logic of a SAML disocvery service.

The constructor takes 3 parameters:

Arguments

- **function (entity_id) {}|string** (*mdq*) – a callable or a URL to be used for MDQ-style lookups of entity objects.
- **string|PersistenceService** (*persistence*) – the URL of a persistence service or an instance of the PersistanceService
- **string** (*context*) – the default context identifier

DiscoveryService.**DiscoveryService**

The constructor takes 3 parameters:

DiscoveryService.**do_saml_discovery_response** (*string*, *(persist)*)

The main entrypoint of the class. Performs the following actions in a Promise-chain: 1. fetches the entity from the persistence service 2. performs an MDQ lookup if the entity was not found 3. returns an item (entity+last_used timestamp)

Arguments

- **string** (*entity_id*) – the entityID of the SAML identity provider
- **(persist)** – [boolean] set to true (default) to persist the discovery metadata

DiscoveryService.**pin** (*string*)

Shorthand for do_saml_discovery_response. Convenience method for the case when you want to pre-populate (aka pin) an identity provider choice. The idea is to call this function, resolve the Promise but not redirect the user.

Arguments

- **string** (*entity_id*) – the entityID of the SAML identity provider

DiscoveryService.**remove** (*string*)

Removes an entity by calling the remove function of the underlying PersistenceService instance.

Arguments

- **string** (*entity_id*) – the entityID of the SAML identity provider to be removed

DiscoveryService.**saml_discovery_response** (*string*)

Call do_saml_discovery_response and then set window.top.location.href to the discovery response URL This assumes that the code is running on the discovery service URL so the relative redirect works.

Arguments

- **string** (*entity_id*) – an entityID of the chosen SAML identity provider.

DiscoveryService.**with_items** (*function (entity) {}*)

Preform callback on all entities in the persistence-service.

Arguments

- **function (entity) {}** (*callback*) – a callable taking a single entity parameter

CHAPTER 8

Indices and tables

- genindex
- search

Index

D

J

P