Interstage Support

Fujitsu America Inc.
1250 East Arques Avenue
Sunnyvale, CA 94085

shaping tomorrow with you

Lightweight Open Authentication

Updated: 14 September 2015
Editor: Keith Swenson

This is a design document for a protocol for the SSOFI Provider to allow for three-party authentiation.

Table of Contents

1 Executive Summary 2
2 Standard Three Party Interaction 3
2.1 The OpenID Provider 3
2.2 The JavaScript Client 3
2.3 The Server Component 3
3 Implementation 4
3.1 Step 1 – Client Learns User Identity 5
3.2 Step 2 – Client Initiates Authentication with Server 6
3.3 Step 3 – Deliver Challenge to Provider 7
3.4 Step 4 – Log in to Server 8
3.5 Step N – Log out 8
4 URLs Enhancements 9

1 Executive Summary

To leverage the SSOFI provider server in AngularJS style user interfaces, as well as authentication to third party servers, a simple, lightweight JSON/REST protocol has been provided. This document described how to use it.

2 Standard Three Party Interaction

There are three pieces of code that need to cooperate if (and only if) the user is properly authenticated.

1. The trusted SSOFI provider – the server the users register and login to.
2. The JavaScript client code
3. The server code written in Java running in a app server

2.1 The OpenID Provider

The SSOFI provider (server) supports the protocol described. The user accesses this server through a user interface, and logs in using the regular mechanism (providing a password). The SSOFI provider gives capabilities for changing passwords, recovering passwords, and specifying a name.

A public server is running at https://interstagebpm.com/eid/ which can be used for most testing and production uses. This server allows users to log in (give a password to prove who they are) and a browser cookie remembers their login session so that whenever they return to the provider they are recognized as that person. This server already allows one to change a password, and to reset the password if they forget. Their ID is an email address and it will send email to that address to prove that they have the email address. We would like to use this component with minimal changes.

2.2 The JavaScript Client

More applications are written in JavaScript to run in the browser. If the user has already logged in using that browser, it is trivial for this kind of client application to find out who the authenticated user is. A simple REST web service request to SSOFI provider (the WhoAmI request) will return a JSON structure telling the user's email address (id) and Full name. This works whenever the JS is running in the same browser that the user used when logging in, because JavaScript HTTP requests use the same cookies as the regular browser HTTP requests.

More important than this is the ability for the JS client to "prove" to another server that a particular user is logged in.

Most of this logging in can happen in the background, with the effect that the user will be logged in automatically (if they have logged into the SSOFI provider) without having to do anything.

2.3 The Server Component

The JS client needs to PROVE to the server that the user is really logged in – without giving the user's password to the server. The server cannot trust a client that is running remotely, on a user's machine. But the server CAN trust the (specific) SSOFI provider. This is the point of the LightweightOAuthProtocol. The server provides a challenge to the JS client, the client then gets a token from the SSOFI provider, and provides it to the server. The server can then verify that the user is who they say they are.

3 Implementation

These are the guiding rules:

- All requests will be HTTP requests, GET and POST
- All data structures will be JSON

The JSON structure will be a subset of the following 6 members:

```
{
    userName:  "Joe Schmo",
    userId:    "joe@example.com",
    challenge: "182B93847W56373",
    token:     "9922-eer-8374-rqq-7232",
    verified:  true,
    msg:       "Free-form, informative statement about what happened"
}
```

The following table clarifies which fields are requires/set at each step:

	apiWho	apiGenerate	apiVerify	apiLogout
userName	set	set	set	
userId	set	set	required	
challenge		required	required	
token		set	required	
msg	show logged in or not logged in	show success or error	show success or error	show that user is not logged in
verified			set true or false	

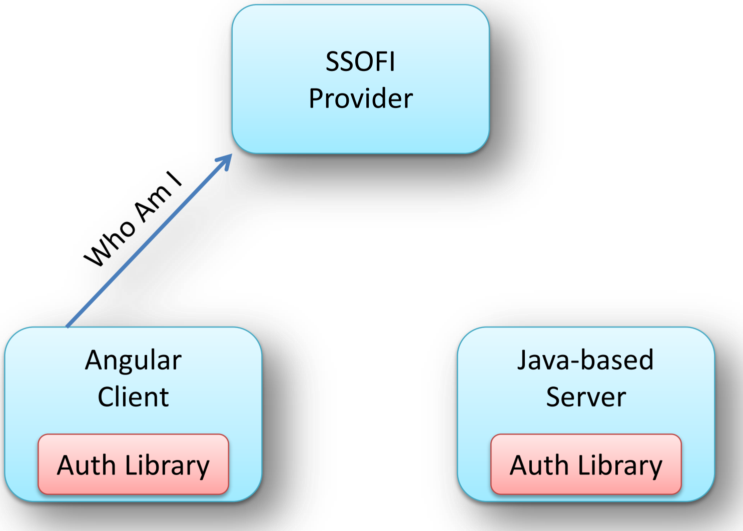
Because apiWho does not need any input, it can be accessed with either a GET or a POST request. apiGenerate requires that you be logged in, and that a challenge is included. The provider will generate a token specific for that challenge. Finally, in apiVerify the request can be made from a server that is a not authenticated, but as long as the challenge and the token match appropriately, it will verify and validate the specific user that had given it the challenge in the first place. The operations apiGenerate and apiVerify can only be called once for a given challenge – after that one call, the memory is cleared of that challenge. The apiVerify returns verified=true if successful (along with user details), and verified=false if not.

apiLogout needs no data sent to it, and results in the user session being voided, if there is one.

3.1 Step 1 – Client Learns User Identity

This is done at any time after the JS client has started, it makes a single GET request to the well-known provider. The request is "Who-Am-I" request.

- If the user is not logged in, it will return an empty JSON, or other indication that the user is not logged in.
- If user logged in, it will return the user key (arbitrary value) email address (user ID), and full display name for the user.



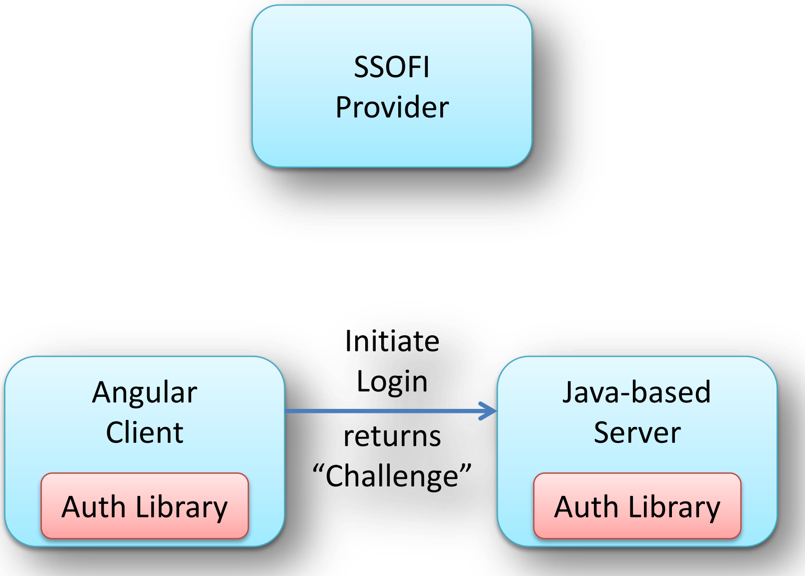
If the user is not logged in, the client should display a log-in link to the SSOFI provider so that the user interface there can take the job of logging the user in, and returning to the Angular client program. No other authentication activity can proceed until the user is logged in to the SSOFI provider, and the Who-Am-I request has returned a proper identification of the user.

3.2 Step 2 – Client Initiates Authentication with Server

Once the client knows who the user is (from step 1) it then starts the process of proving to the server who the user is. This can all proceed in the background without the user being overtly aware of it.

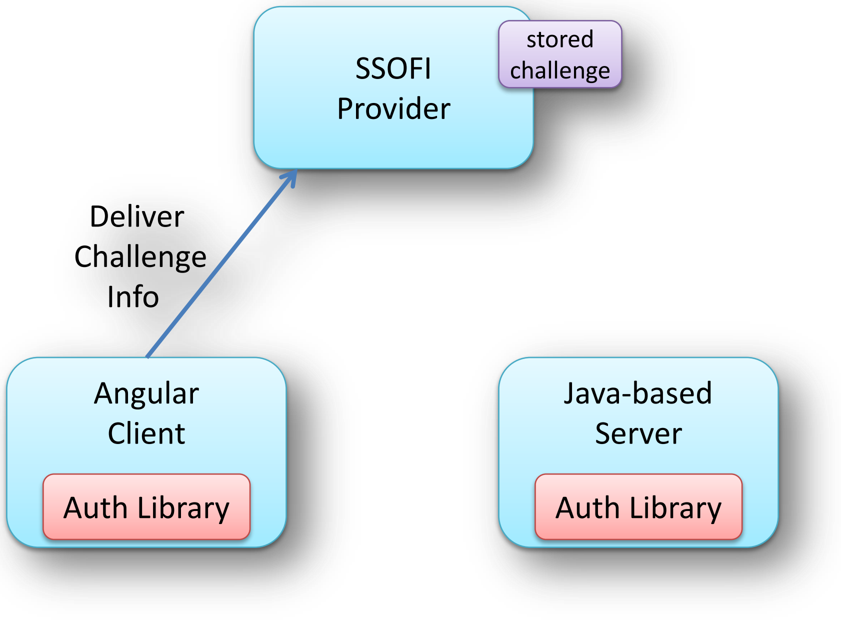
The browser will already have a "session" with the server (using cookies), so that the server can keep track of multiple requests from the browser and associate them appropriately.

The client makes an Initiate-Login request to the server. In the request it will specify the user ID of the user, as well as the other information about the user. This is NOT trusted by the server. The server will remember the username and return a unique nonce (a token that can be used only once) "challenge". The server must generate a new, unique challenge value for each request.



3.3 Step 3 – Deliver Challenge to Provider

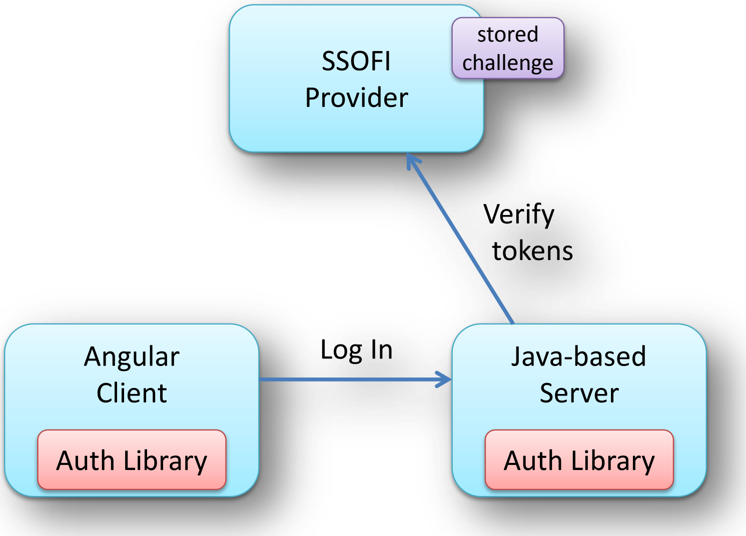
The JS client will deliver the challenge to the provider. This primes the provider to be ready for a verification call from the server. The provider remembers the information passed, and it returns a token of its own to be used in verification.



3.4 Step 4 – Log in to Server

Now the client passes the challenge and verification token to the server, in order to finally actually log in to the server. While the client is waiting for a response, the server takes the info, the user info, the challenge nonce, and the verification token, passes them to the provider. If everything is good, the provider will respond confirming the identity of the user. The server confirms back to the client that login has been successful.

If the verification fails for any reason, then the failure is reported back to the server, and then to the client, and the user remains unlogged in.



The verification will only be done once. The provider will forget the challenge & token information so that it cannot be used again. The challenge information will be good only for a few minutes in any case, and after 10 minutes will be forgotten.

The provider will need to allow for multiple simultaneous authentication attempts from different users or different servers. Since the information need be kept only 10 minutes this should not cause any difficulty.

3.5 Step N – Log out

The angular client may call the provider with the apiLogout parameter. That will assure that the user's session is voided, if there is one. If there is no session, there is no error, it simply returns in all cases that the user is not logged in. When the user is not logged it, the response message only contains the "msg" parameter. The other parameters (e.g. userName and userId) make no sense when nobody is logged in, so that are not included.

4 URLs Enhancements

The current SSOFI provider implements the standard OpenID protocol. In order to prevent making addresses that would disturb this, the three important services are identified by the URL parameter "opened.mode" as such:

- http://<server>/<ssofi>/?openid.mode=apiWho
- http://<server>/<ssofi>/?openid.mode=apiGenerate
- http://<server>/<ssofi>/?openid.mode=apiVerify
- http://<server>/<ssofi>/?openid.mode=apiLogout

You SSOFI provider is installed into a TomCat server, and the address of that TomCat server is "http://<server>" so replace this with the address of your server, and the port number if applicable.

The name of the application that you installed the ssofi provider to appears after the server. This is often "id" or "eid" but it could be anything specified by the person who does the install. Replace the <ssofi> with the name of the application it is running on.

In general, SSOFI OpenID look like this:

- http://<server>/<ssofi>/username@domain.com

For the purpose of the LightweightOAuthProtocol, you do not need a user id (email address) on the end, and it does not matter if it is there or not. Since in all cases you will be communicating with the server for the logged in user, it is better for consistency if you omit the user id, and use the raw addresses above without user id in them.

The parameter is named "opened.mode". This parameter is used for many things in the standard OpenID protocol. For this lightweight protocol, you must use the exact four values listed above. In all four cases, the method will accept a JSON document in the post body, and will return a JSON document back to you. If things work OK, you will get a status code 200, and the JSON document conforming to the structure in section 2. If an exception is thrown within the server, or if the server otherwise fails to handle the request, you will get a status code 500, and a JSON document describing the exception. Finally, if verification fails, you will get a 400 status code back.

If verification succeeds, and the server will get a 200 response. The server must check : The user ID from the provider must match the user ID that the client originally provided. If it is different in any way, then the client sent you a false user id which is very suspicious. The server probably should probably in this case return an error, and not consider the user authenticated.