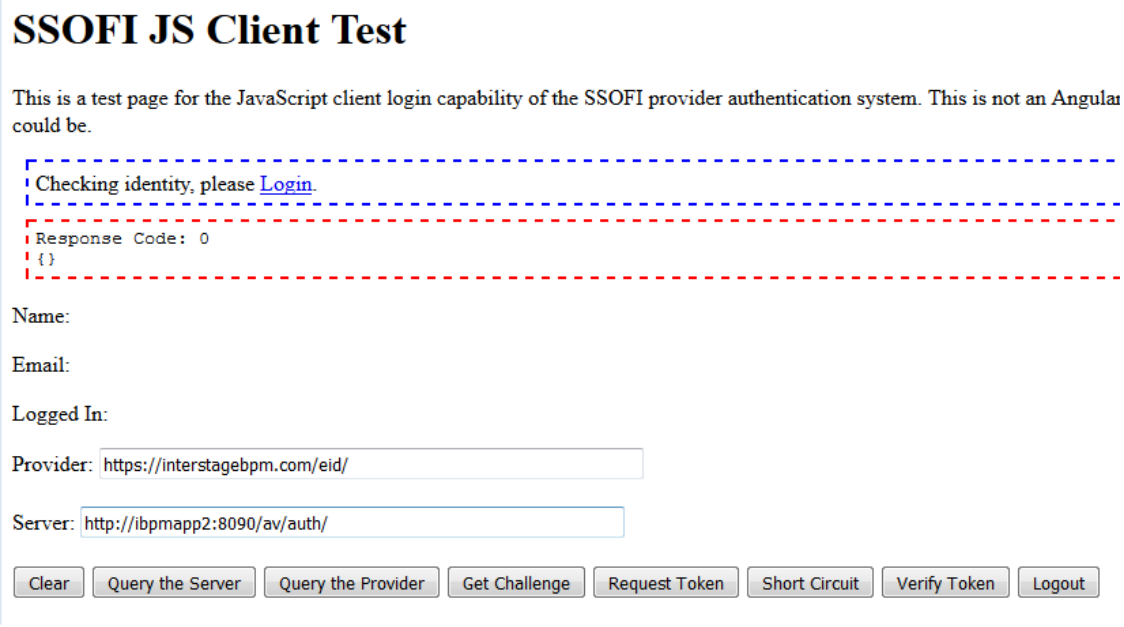


SSOFI Lightweight Authentication Protocol (SLAP)

Updated: 18 October 2020
Editor: Keith Swenson

This is a protocol specification for the SSOFI Provider to allow for three-party authentication.

Table of Contents

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 describes how to use it.

The 2015 version of this protocol used cookies to maintain a session with the SSOFI provider, but new changes in HTTP standard restrict the use of cookies in such a way that dev and test would be difficult, so the 2020 version eliminates the need to rely on cookies.

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. **Provider:** The trusted Identity provider – the server the users register and login to.
2. **Client:** The JavaScript client code.
3. **Server:** The server code written in Java running in an app server.

2.1 The Identity Provider

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

A public server is running at https://s06.circleweaver.com/ssofi/ 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.

The ability for the JS client to "prove" to another server that a particular user is logged in is more important.

Most of this exchange in can happen in the background, with the effect that the user will be logged in automatically to the server without having to do anything.

2.3 The Server Component

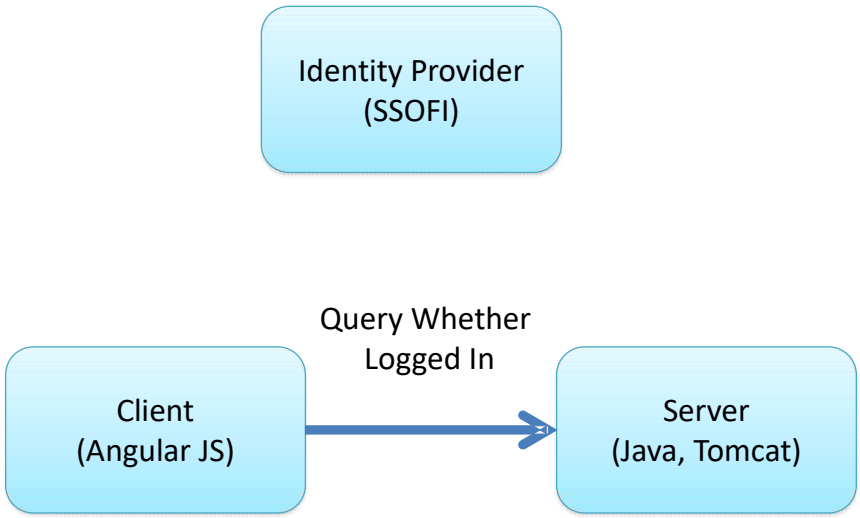
The JS client needs to PROVE to the server that the user is really logged in – without giving the user's password to the server. The server cannot trust a client that is running remotely, on a user's machine. But the server CAN trust the (specific) SSOFI provider. This is the point of SLAP. The server 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 Pattern of Interactions

Before we cover the details of the API and data structures, let's explore the pattern of calls. There is a specific sequence of calls that must be made in the right order to login.

3.1 Step 1 - Client Learns whether Already Logged into Server

The JavaScript client will also want to know whether the user has already logged into the server in a previous interaction. You don't want to log in again if you don't have to. So there is a way to query the server to find out if it thinks the client is logged in, and to find out the user that the server things is logged in as well, to make sure that the client represents that correct.

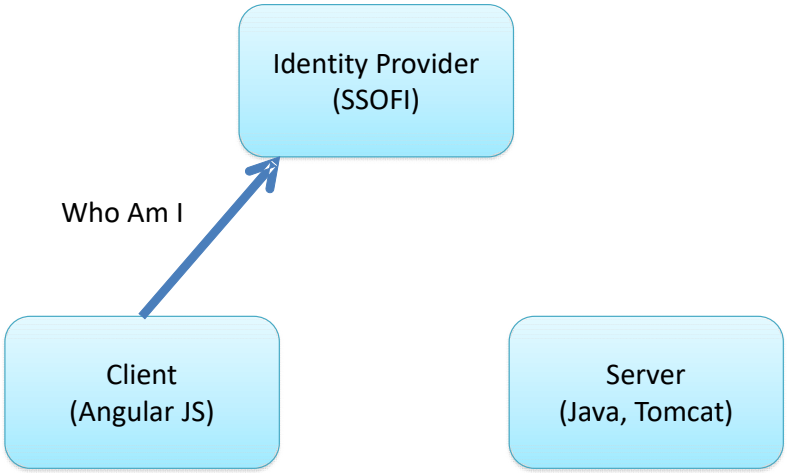


If the user is already logged in, then the server will return the user ID and the full name of the user. If the user is already logged, then you need not perform any more of the protocol.

If the user is not logged in, it will return an empty structure, or otherwise indicate that it does not know who the user is. In this case the client must move to the next step in the exchange.

3.2 Step 2 – Client Learns whether already logged into Identity Provider

Once the client knows that it is not authenticated to the server, the next step is to see if the user is already logged into the Identity Provider. This is the "Who-Am-I" request.



If the user is not logged in, it will return an JSON that is empty except the 'ss' value indicating the session id that the request has generated. At this point the client should display to the user a login prompt. To allow the user to log in, the client must redirect the browser to the SSOFI Provider which will then present some user interface to get the password from the user. Detail on how to redirect are in a lower section. After the user logs in, the browser is redirected back to the client, where it starts again from the top, and hopefully in this stage finds the user logged in. 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.

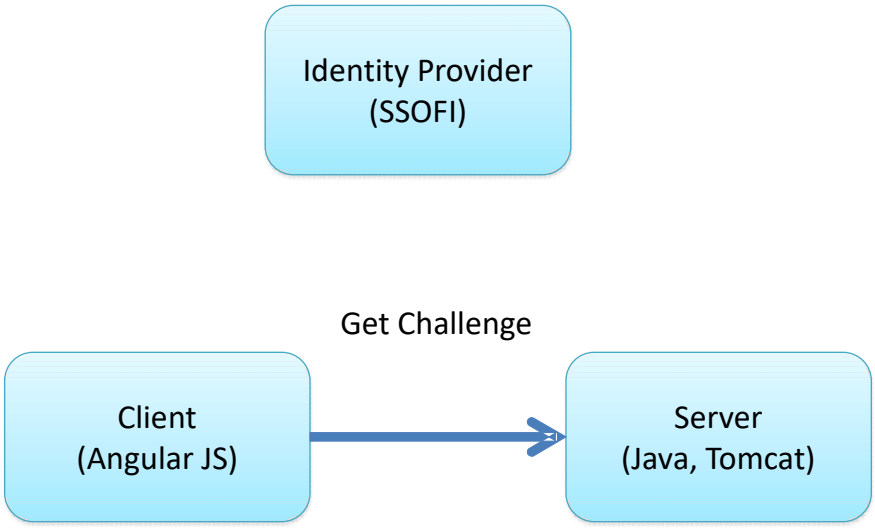
If user is logged in, the identity provider will return the user email address (user ID), and full display name for the user. This is enough to convince the client that the user is who they are, but the rest of the exchange is to convince the server of whom the user is.

3.3 Step 3 – 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.



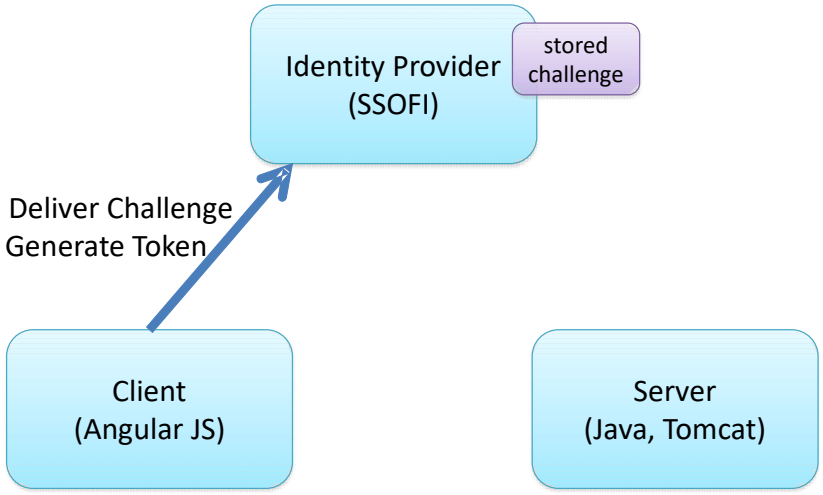
The client requests a "challenge" from the server. A challenge is an arbitrary value that the server makes up and it serves only to identify this exchange. It does not matter how the server generates the challenge. It matters only that it is unique – no two requests for a challenge should ever return the same value – and that it is long enough that it is hard to guess. Generally it is a good idea to use a timestamp to assure uniqueness, and then add in more random values to make the final string value hard to guess in advance.

The challenge is returned to the client in a JSON structure. The server feely gives this value out to any client that asks for one and will never restrict access in any way since this is the first step in authentication. It should never fail to give the challenge value out.

The server must have some mechanism to establish a session with the client, usually by way of a session cookie. The challenge that is given out should be tied with the session in such a way that only that client can return the challenge to it in a later call. Or, in other words, a different would never be allowed to return that challenge for any useful purpose. Only that one client given the challenge should be able to use the challenge value.

3.4 Step 4 – Deliver Challenge to Provider and get a Token

The JS client will make the getToken call on the identity provider passing the challenge as a parameter in a JSON structure, and receiving a token value back in a JSON result.

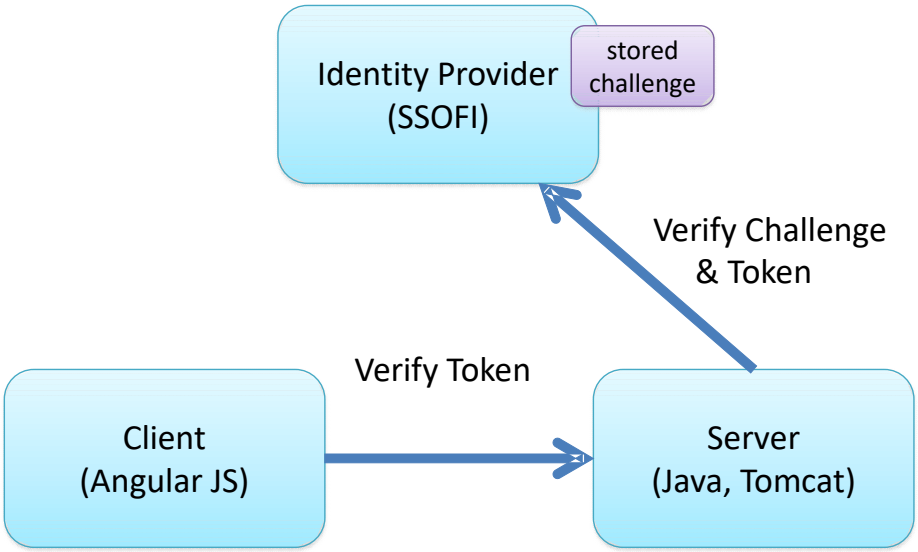


The token is another arbitrary value which the provider makes up. Like the challenge, the exact form of the token is unimportant. It matters only that the token is unique and that it is hard to guess.

The provider remembers the challenge and the token. They are associated with each other, and the pair are associated with the user that has logged in from that client. It needs to remember this information only for a few minutes since this exchange usually completes very quickly, and if it does not complete in a few minutes, it probably will never complete.

3.5 Step 5 – Verify the Token to establish Authentication

Now the client calls Verify Token, passing both the challenge and the token to the server. While the client is waiting for a response, the server must turn around and call the Verify Token on the identity provider.



If the provider is given a challenge and a token that are properly recorded in memory as being associated, then the login succeed, and the provider returns the user id and the full name of the user who is logged in at the client. This is authoritative: the server can trust this, and remember these user credentials as the logged in user. It returns to the client a confirmation that it is logged in.

If the provider can find no record of that challenge, or if the token does not match the one associated with the challenge, then verification fails, and the provider reports the failure back to the server. The server will in turn return a failure back to the user. The server will consider the client unauthenticated, even if the client had been logged in earlier, it clears any user session information when the verification fails.

The client should get only one chance to use a particular challenge. For a given challenge value, whether verification succeeds or fails, the challenge and the associated token should be forgotten, preventing a second client using the same pair of values to authenticate.

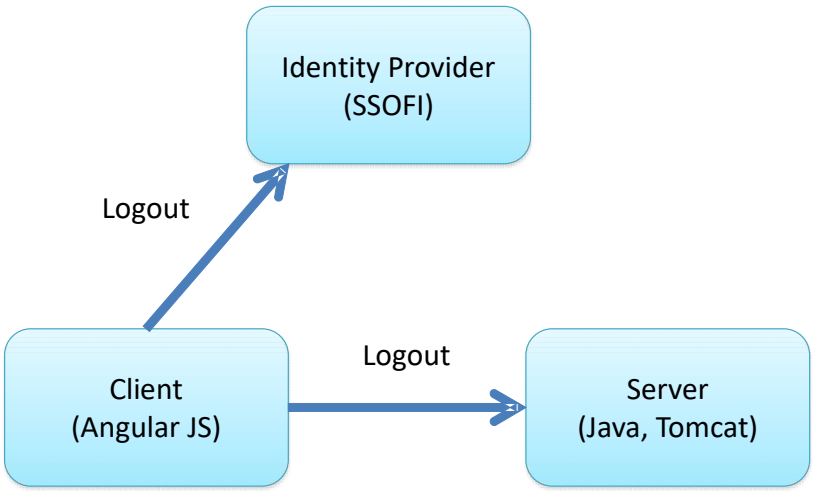
The provider will need to allow for multiple simultaneous authentication attempts from different users or different servers. Since the information need be kept only 10 minutes this should not cause any difficulty.

This is the end. If successful, the user is logged in. If unsuccessful, the process must start again at the beginning.

3.6 Step N – Log out

Once logged in, the user should be given the option to log out. Both the server and the provider must offer API calls that will result in forgetting about the user, and leaving the user logged out of the system.

Generally a user will do this either when using a shared computer and wanting to make sure that nobody else has access to their account, or when they hope to log in as another user. In either case, the client will want to log out of both the server and the provider. Logging out of one, and not the other, would leave them in a state that might either allow another access to their account or prevent them from logging in as a different user.



The logout command should never fail. It should respond with success whether the user is currently logged in or not. The result is that the user session is voided, regardless of whether there was an active session in place when the call was made. It does not matter which order there calls are made – they can be made simultaneously – but the code should be designed that if the network causes an error reaching one of them, this should not prevent the logout command being called on the other.

4 Data Structures

All operations in SLAP follow these guiding rules:

- All requests will be HTTP requests, GET and POST
- All data structures will be JSON
- All requests will use the same member names for specific values.

The JSON structure will be a subset of the following 8 members:

```
{
    ss:         "SZKJ-LU-ILQ-FK-JRH",
    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",
    error:      {<W3c-exception-object>}
}
```

Not all members will be used in the parameters to every operation, nor will every member be included in a response from the operation. See the detailed table on which members are used with which operations.

If the operation succeeds, it will return a HTTP response value of 200 with the JSON structure.

If the operation fails, for example if verify token does not successfully verify the token and challenge value as matching, then the operation should return an HTTP response value of 400. If some significant unexpected error occurs on the server, causing an exception to be thrown, the description of that problem should be return in the error member.

Because this protocol explicitly uses cross-site calls, that is, the JS code from one site is calling a provider on another site, and might be calling a server on yet another site, it is important the both the server and provider implement the right header settings so that the browser that is running the JS code allows it to run. In general, the following settings will need to be made:

```
String origin = req.getHeader("Origin");
if (origin==null || origin.length()==0) {
    origin="*";
}
resp.setHeader("Access-Control-Allow-Origin",          origin);
resp.setHeader("Access-Control-Allow-Credentials", "true");
resp.setHeader("Access-Control-Allow-Methods",     "GET, POST, OPTIONS");
resp.setHeader("Access-Control-Allow-Headers",     "Authorization");
resp.setHeader("Access-Control-Max-Age",            "1");
resp.setHeader("Vary",        "*");
```

5 SSOFI Provider Details

5.1 URL API Addresses

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

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

SSOFI provider runs on TomCat server. The address of that TomCat server is something like "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. The slash after the application name is required.

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.

5.2 User Login URL

If you find that the user is not logged into the provider, and the use clicks on the login button, then the client will need to redirect the browser to the following address:

- http://<server>/<ssofi>/?openid.mode=quick&go=<your client address>

This address produces an HTML user interface. You must include two URL parameters.

- "opened.mode" must be "quick" which means to just login and return.
- "go" parameter specifies where to return to. Pass the current address of the client in the "go" parameter, or whatever address you want the user to come back to.

The user might login, or might cancel and not log in. When the browsers returns, the client code must do all the same things again to check if logged into the server and provider, and continue the protocol from there.

It is possible to launch the login to the provider in a new, separate window in which the user logs in. The only trick is making sure that the client window somehow re-checks to see if the user is logged in, and proceed from there.

5.3 Data for each provider operation

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

operation	Who Am I	Generate Token	Verify Token	Logout
opened.mode=	apiWho	apiGenerate	apiVerify	apiLogout
userName	output		output	
userId	output		output	
challenge		input	input	
token		output	input	
msg	optional message describes result	optional message describes result	optional message describes result	optional message describes result
verified			output	
error	might be present if failure	might be present if failure	might be present if failure	might be present if failure

apiWho does not need any input, it can be accessed with either a GET or a POST request.

apiGenerate requires that the user be logged in, and that a challenge is included. The provider will generate a token specific for that challenge. It will generate an error/failure if no user logged in.

apiVerify the request can be made anonymously! No matter who calls, if the challenge and the token match appropriately, it will return information about the specific user that had given it the challenge in the first place. It returns verified=true if successful (along with user details), and verified=false if not.

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 and token.

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

6 Standard Server Details

6.1 URL Addresses

A standard implementation of a server should implement the protocol in this way.

- http://<server>/<server-auth>/query
- http://<server>/<server-auth>/getChallenge
- http://<server>/<server-auth>/verifyToken
- http://<server>/<server-auth>/logout

Again, the path that leads up to the final token in the URL is entirely dependent upon the implementation of the server. The implementation of the protocol should have no dependence on anything but the last token. The path may be any length, with any number of slashes in it. The last token distinguishes the operation in the protocol.

6.2 Data Members for Operations

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

operation	Query User	Get Challenge	Verify Token	Logout
last token	query	getChallenge	verifyToken	logout
ss	output	output	output	output
userName	output		output	
userId	output		output	
challenge		output	input	
token			input	
msg	optional message describes result	optional message describes result	optional message describes result	optional message describes result
verified			output	
error	might be present if failure	might be present if failure	might be present if failure	might be present if failure

6.3 Implementing Operations

If you are implementing the server part of this exchange, you will need to consider the following things about each of the operations.

Query User –

This lightweight call just returns the userid (email address) and full name of the user that is logged in, if there is any. It can return an empty JS object if there is no user logged in. Optionally, the server might include a "msg" member saying that no user is logged in -- useful for debugging. Since this operation needs no input parameters, it should work with both GET and POST method calls.

GetChallenge –

You will need to generate a challenge value. In order to make it unique, start with a current timestamp. Remember the timestamp and make sure that the next call is at least one more than the last timestamp used. Convert the timestamp to a sequence of letters or numbers. Then, either tack onto the end, or insert into the middle, 6 to 8 more random letters or numbers created with a random number generator. Randomize the random number generator so that subsequent starts of the server do not yield the same sequence of random values.

Remember the challenge that was handed out associated with the client session object. If there are any earlier challenges given out to that client, they should be discarded. You only need to remember the latest challenge given to any given client.

VerifyToken –

This is the most difficult operation to implement, because you must receive the request, and then make a call to the provider. Since all the input values are the same, you can take the JSON object and simply forward it to the provider's version of the verify token operation.

Only accept a verify request if the challenge matches the challenge that you recently gave to the client. If the challenge is not there (e.g. it was already cleared out) or a different challenge is there, then immediately return a failure and invalidate any existing session for the client.

You must check that the verified member is true on the return. The server must take the user id and name that the identity provider returned, and should not use any user id value that came from the client. The userid and full name must be returned to the client as verification that the new authenticated session is the one the client is expecting.

If the verification fails, the server should invalidate any existing session for that client.

Logout –

Invalidate any existing session for that client and return an empty JS object. If there is no session for the client, just ignore the call, and return the empty JS object. Do not fail if you can prevent it.

7 Client Implementation

The actual implementation might depend upon the details of what libraries the JS client is using, the following JavaScript code is a functional implementation of the client side of the protocol that should work in all current browsers.

```
            <div id="welcomeMessage">
                Not logged in
            </div>

<script>

loggedInNow = false;
loginInfo = {};
providerUrl = 'specify provider url';
serverUrl   = 'specify server url here';
currentPageURL = window.location;
responseCode = 0;

function getJSON(url, passedFunction) {
    console.log("calling GET");
    var xhr = new XMLHttpRequest();
    globalForXhr = xhr;
    xhr.open("GET", url, true);
    xhr.withCredentials = true;
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4 && xhr.status == 200) {
            try {
                responseCode = xhr.status;
                passedFunction(JSON.parse(xhr.responseText));
            }
            catch (e) {
                alert("Got exception ("+e+") trying to handle: "+url);
            }
        }
    }
    xhr.send();
}

function postJSON(url, data, passedFunction) {
    console.log("calling POST");
    var xhr = new XMLHttpRequest();
    globalForXhr = xhr;
    xhr.open("POST", url, true);
    xhr.withCredentials = true;
    xhr.setRequestHeader("Content-Type","text/plain");
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4 && xhr.status == 200) {
```

```
            try {
                responseCode = xhr.status;
                passedFunction(JSON.parse(xhr.responseText));
            }
            catch (e) {
                alert("Got exception ("+e+") trying to handle: "+url);
            }
        }
    }
    xhr.send(JSON.stringify(data));
}

function queryTheProvider() {
    var pUrl = providerUrl + "?openid.mode=apiWho";
    getJSON(pUrl, function(data) {
        loginInfo = data;
        displayWelcomeMessage();
        if (data.userId) {
            requestChallenge();
        }
    });
}

function requestChallenge() {
    var pUrl = serverUrl + "auth/getChallenge";
    postJSON(pUrl, loginInfo, function(data) {
        loginInfo = data;
        displayWelcomeMessage();
        getToken();
    });
}

function getToken() {
    var pUrl  = providerUrl + "?openid.mode=apiGenerate";
    postJSON(pUrl, loginInfo, function(data) {
        loginInfo = data;
        displayWelcomeMessage();
        verifyToken();
    });
}

function verifyToken() {
    var pUrl = serverUrl + "auth/verifyToken";
    postJSON(pUrl, loginInfo, function(data) {
        loginInfo = data;
        if (loginInfo.verified) {
            loggedInNow=true;
            window.location.reload();
        }
        else {
```

```
            alert("Internal Error: was not able to verify token:
"+JSON.stringify(data));
        }
        displayWelcomeMessage();
    });
}

function logOutProvider() {
    var pUrl = providerUrl + "?openid.mode=apiLogout";
    postJSON(pUrl, loginInfo, function(data) {
        loginInfo = data;
        logOutServer();
    });
}
function logOutServer() {
    var pUrl = serverUrl + "auth/logout";
    postJSON(pUrl, loginInfo, function(data) {
        loginInfo = data;
        loggedInNow = false;
        displayWelcomeMessage();
        window.location.reload();
    });
}


function displayWelcomeMessage() {
    var y = document.getElementById("welcomeMessage");
    if (responseCode==0) {
        y.innerHTML = 'Checking identity, please <a href="'
            +providerUrl
            +'&go='+currentPageURL+'">Login</a>.';
    }
    else if (!loginInfo.userName) {
        y.innerHTML = 'Not logged in, please <a href="'
            +providerUrl
            +'?openid.mode=quick&go='+currentPageURL+'">Login</a>.';
    }
    else if (loggedInNow == false) {
        y.innerHTML = 'Hello <b>'+loginInfo.userName
            +'</b>.  Attempting Automatic Login.';
    }
    else if (loggedInBeforePageFetch != loggedInNow) {
        //just refresh page
        y.innerHTML = '<a href="'+currentPageURL+'>Refresh Page</a>';
    }
    else {
        y.innerHTML = 'Welcome <b>'+loginInfo.userName
            +'</b>.  <a target="_blank" href="'
            +providerUrl
            +'?openid.mode=logout&go='+currentPageURL+'">Logout</a>.';
    }
```

```
}

//kick off the authentication with this:
queryTheProvider();

</script>
```

8 Public SSOFI Provider and Test Facility

There is a publicly availalble SSOFI server available at:

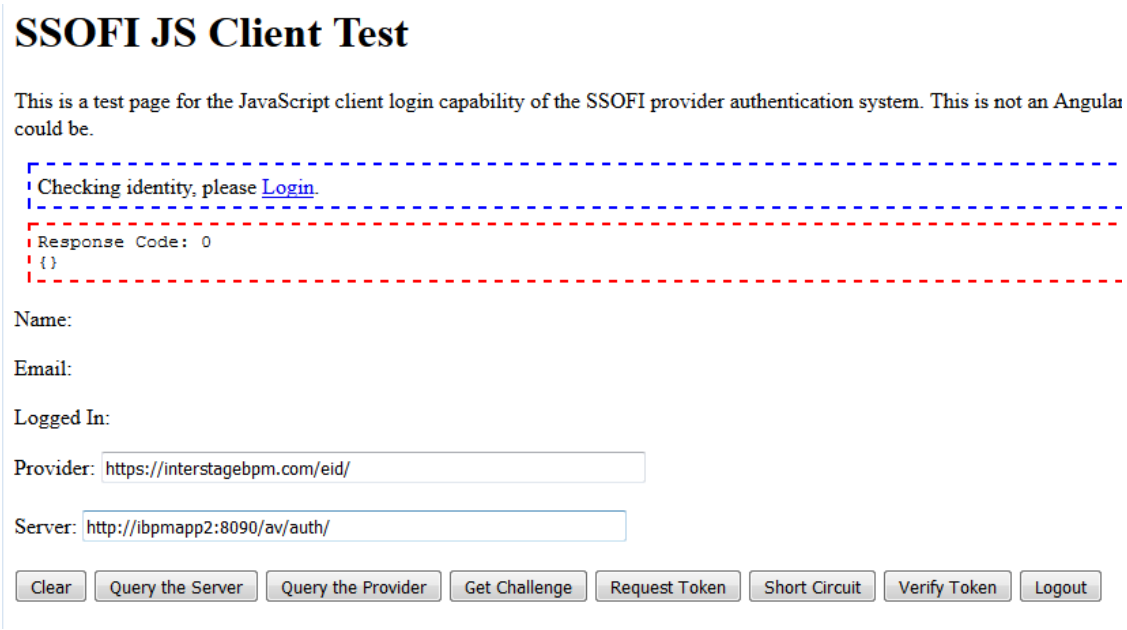
```
https://interstagebpm.com/eid
```

This should be readily available to most places on the internet and can be used to authenticate users. Users register an email address, set up their own passwords, and set their own full name.

There is a SLAP protocol test page available at:

```
https://interstagebpm.com/av/jsLogin.jsp
```

It looks like this at the current moment:



There are boxes to specify the server URL and the provider URL (prefilled with the public provider mentioned above). There are buttons to represent each step in the protocol. Specify your own server URL, make sure you server is set to verify with the public provider, and you can test to see that the protocol is implemented in a compatible way.