Table of Contents

[NextGen SSO Environment Overview 2](#_Toc256240934)

[Reverse Proxying and Enforcement 3](#_Toc256240935)

[Canonical Versus Application Space Enforcement 3](#_Toc256240936)

[User Attribute Injection 4](#_Toc256240937)

[Fine Grained Permissions 4](#_Toc256240938)

[Interlinking Applications 4](#_Toc256240939)

[Application Resource Protection 4](#_Toc256240940)

[Application Roles 4](#_Toc256240941)

[Cookies and Site Domains 5](#_Toc256240942)

[SSO Environment Simulator 5](#_Toc256240943)

[Configuring the SSO Environment Simulator 6](#_Toc256240944)

[Getting the Simulator 7](#_Toc256240945)

[Starting the Simulator and Accessing the Console 7](#_Toc256240946)

[<config> 9](#_Toc256240947)

[<console-recording> 10](#_Toc256240948)

[Accessing the Rest API 10](#_Toc256240949)

[<sso-cookie> 11](#_Toc256240950)

[Aliases and Macros 12](#_Toc256240951)

[Preparing for SSO Traffic – etc/hosts 13](#_Toc256240952)

[<sso-traffic> 14](#_Toc256240953)

[Rewriting Redirects 16](#_Toc256240954)

[Rewritting Cookie Paths 17](#_Toc256240955)

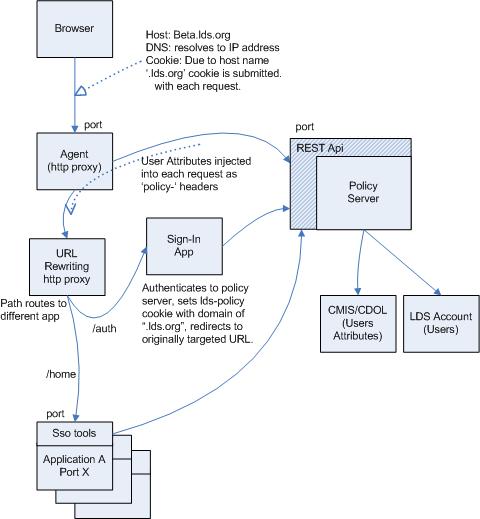
[<users> 18](#_Toc256240956)

[APPENDIX A - DTD 18](#_Toc256240957)

# NextGen SSO Environment Overview

The SSO Environment Simulator sometimes referred to as the “shim” is a standalone tool enabling simulation of the runtime environment seen by applications when running in the NextGen single sign-on environment. That environment leverages the OpenSSO solution with http web agents and reverse proxies to route traffic to back end applications as shown in Figure 1. Inherent in this environment is that all applications appear to be on the same domain server; beta.lds.org for example. Applications then reside at different sub-paths of this domain. These sub-paths are also used to uniquely identify packets associated with that application allowing for routing to the back-end cluster containing that application after it has passed through the agent.

Figure : Church Single Sign-On Environment



For example, https://beta.lds.org/mls/mbr/… identifies the MLS Web member application and all packets headed to URLs starting with /mls/mbr would be routed to the server cluster housing that application provided the user is allowed to access that URL resource. This structure is known as reverse proxying; all packets for a site come to a proxy which then routes them to any number of servers. Forward proxying refers to user agents like browsers having to use a proxy to access all sites to which it wishes to go.

## Reverse Proxying and Enforcement

As shown in Figure 1, traffic to the site is resolved by DNS to hit a special reverse proxy known in opensso terminology as an agent. As hinted to already, an agent looks at each request and checks to see if that URL is accessible to a user. It first consults a list of unenforced URLs. If the URL is found in that list, then the packet is proxied onward to the appropriate cluster. If not found then it must consult access policies to ensure access by the user is allowed. If the user does not have a currently active session, then the agent must redirect the user to a sign-in page. This sign-in page typically also happens to be an application protected by the agent but its URLs are included in the unenforced list allowing its traffic to pass through the agent. Upon receiving the user’s credentials the sign-in service then uses a REST api on the policy server to authenticate the user. If successful, the application sets the resultant token in a cookie and redirects the user agent back to the original URL.

Upon receiving the original request with the active cookie, the agent now contacts the policy server asking if the URL is allowed to be accessed by the user. The policy server consults all configured policies protecting that URL to see if the user meets any of the conditions for access and accordingly forbids access or allows the traffic onward to be routed to the targeted application.

## Canonical Versus Application Space Enforcement

A result of using a reverse proxy approach is that the agent evaluates URLs in what is referred to as the canonical space, the URLs that show in the user’s browser. The rewriting proxy can and does rewrite URLs to accommodate some requirements of application implementation technologies. The java context is a notable example. The application residing at /mls/mbr… is a java application and more applications will be coming that will be mapped beneath the mls tier. But java contexts must be a single path level and this application’s java context root happens to be /mls-membership.

When the incoming packet hits the java application server it had better have a path starting with /mls-membership and not /mls/mbr otherwise the web server won’t associate it with and pass it into a servlet container for handling. Such URLs arriving at the technology implementation point such as the servlet container and subject to requirements of that technology are in what is referred to as application space. Where a URL has a different form in application space than it does in canonical space the rewriting reverse proxy infrastructure is used to “rewrite” the URL before it hits the application server.

This is a subtle but important point. If enforcement of access were implemented within the applications themselves via a filter in java or a rewrite module in xquery for example, then the URLs used to enforce that access would be application space URLs not canonical space URLs. When problems arise with access to a specific resource and a user calls the help desk, the problematic URL being targeted can not be compared directly against the URLs configured in policies when using application space enforcement. With canonical space enforcement the URLs seen by the users match those in the policies.

## User Attribute Injection

In addition to enforcing access to URLs the agent is configured to inject a number of headers into each request that it passes onward to the application servers. Some headers are always injected for all requests whether a user is authenticated or not and whether the URL is in the unenforced list or not. Others are user specific like the lds-account-id and lds-mrn of a user. These headers will be discussed in more detail in section ???????.

## Fine Grained Permissions

One such header injected for all requests is “**policy-service-url**”. It is an indication of the location of the REST api used by the agent to protect resources as shown in Figure 1. A library of utilities provided by the SSO team allows applications to call to that REST api and evaluate URI access decisions themselves. Lets look at how such a feature could be used. In all such uses of an application soliciting access decisions directly from the policy server are collectively referred to as **fine grained permissions**.

### Interlinking Applications

Remember that access to an application is restricted by a URL or URL pattern for that application. Suppose that our application contains a link to that application. By leveraging this REST api our application can conditionally render the link to the other application only if the user has access to that other application. Even better, if the access policies change for that other application our application doesn’t have to be redeployed. It still asks the policy server for an access answer for the URL and the server now answers accordingly for the latest policy.

### Application Resource Protection

Just like links to other applications embedded within their application, this feature further enables developers to identify resources within their own application that should conditionally be rendered. The developer can assign that resource a URI within the URL space of the application and conditionally render that resource only if the user has been granted access to that URI. For such URIs that do not necessarily represent URLs that will be hit by the user agent, but rather are simply identifiers unique to the application and are solely for restricting access to a resource, the scheme of the URI is typically selected to be “app://” to set such URIs apart in the policies for that application.

For example, application B wants Bishops and members to see most of its landing page but one summary section should only be seen by Bishops. That section can conditionally be rendered by protecting it by a URI of, “app://beta.lds.org/app-B/restricted-block”, and then granted by a policy requiring the user to have a position of Bishop.

### Application Roles

For another example, application C uses the concept of roles to implement workflows. URIs representing each role can be crafted like, “app://beta.lds.org/app-C/role/ip-moderator”, and policies are created with a condition requiring the user have one of the lds account identifiers in a given list. By delegating to the policy server asking if the user can access that URI, the application can identify if the user has that role and act accordingly.

In all three examples, the greatest benefit of placing the access characteristics in policies external to the application is that the policies can be adjusted without having to redeploy the applications or requiring the application to implement its own security database and management screens. If we wish to broaden the access to the Bishop’s summary to his counselors, that is easily achieved in the policy. If we need to add an additional user to those having the IP Moderator role specific to application C, then we add them to the corresponding policy.

Such roles could be assigned by adding attributes into the ldap store for a user but this is an application role and applications come and go over time. Policies are designed to protect resources as in application **B** or aggregate those who should have a given role as in application **C**. As applications change so too will their resources and naturally require adjustment to the corresponding policies. Changes to the ldap store should be for longer lived, less changing data.

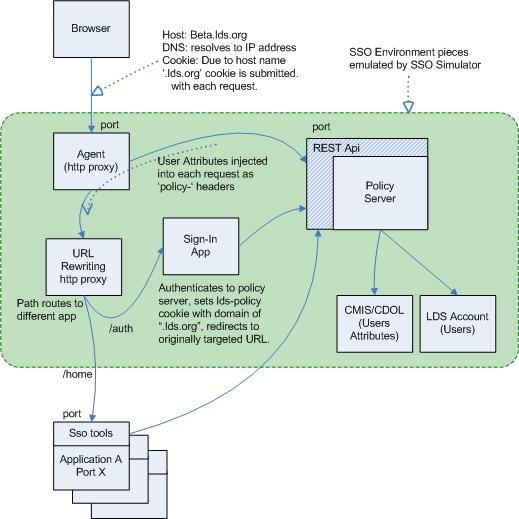
## Cookies and Site Domains

One more important characteristic must be highlighted in Figure 1. The cookie being used in the SSO environment matches the domain of the site. For example, the site is beta.lds.org and the domain of the cookies is configured to be “.lds.org”. Therefore, when requests are made to the site the cookie is passed along with the request by the user agent and identifies the user to the agent and to the application. If the domain of the cookie does not match that of the site then the cookie will not be submitted and result in a ping pong effect with the request hitting the agent, the agent redirecting to the sign-in page since it sees no valid seesion cookie, and the sign-in page then allowing the user to authenticate again to no avail.

# SSO Environment Simulator

Lets now discuss the SSO Environment Simulator or shim. The simulator appears to applications as shown in Figure 2. Specifically, it provides a proxying agent, policies, implementation of the opensso REST api for callbacks from applications, a user store (with optional support for coda users), rewriting and routing, injection of headers both general and user specific, and a sign-in page. Lets take a closer look.

Figure : Portion of NextGen SSO Environment Simulated



# Configuring the SSO Environment Simulator

As can be seen in Figure 2 by the arrows entering the green section from the browser and from the applications accessing the REST api, the simulator listens on two ports. One is known as the proxy port and is meant to simulate a next generation church site like labs.lds.org or beta.lds.org. Via configuration, traffic intended for applications under development and running on different ports on the local box is proxied to those other ports essentially mapping those applications into and making them appear as one seamless site.

The second port has two purposes: it implements the REST api of the OpenSSO policy server for calls from applications and it provides a number of console pages. An example of the REST api usage is the sign-in application known as ml-auth used for labs and beta.lds.org. That application accepts user credentials, calls the REST api to start a session in the policy server and acquire the session token and SSO cookie name, then sets the SSO session cookie in the browser as it redirects to the originally targeted URL. Using the simulator’s REST implementation allows ml-auth to be developed and tested locally without a full OpenSSO environment. Before we show how such a set up is configured lets start with a more simple example that simply demonstrates the available console pages within the simulator.

## Getting the Simulator

The simulator runtime environment and all dependents are available from the SSO team or from the their ldsteams site. Once unzipped on a local machine it creates a “simulator” directory. Within that directory is a sim.bat are a sim.sh file. These must be modified slightly for your machine to include the path to java’s tools.jar so that the simulator is able to compile its jsp pages. (TBD: Scott is working on an executable jar version. Update with that approach.)

Once modified, the simulator can be started by calling sim.bat or sim.sh and passing a single parameter which is the configuration file that should be used. This has two formats. It can be a path on the machine like myApp.xml which would be a file assumed to be in the current directory. Alternatively, it can be fully qualified. In addition to file based resources classpath based files can be used by prefixing their path within the classpath with the text “classpath:”. This is how we will be demonstrating the simulator in the following pages.

## Starting the Simulator and Accessing the Console

Included in the simulator’s jar are a number of example configuration files to help demonstrate its functionality. To start the simulator in its most simple configuration to show the console pages that are available to assist with troubleshooting issues execute the command shown in Listing 1.

Looking at the sim.bat script you can see that the entry point for the simulator is the org.lds.sso.appwrap.Service class and that our command line parameter is passed in to that class upon startup. This class takes a single command line argument which is the path to an XML configuration file. As already noted, if this path is prefixed with “classpath:” then the file is looked for on the classpath. Otherwise, it is looked for on the file system. Therefore, Listing 1 is specifying a path that will be looked for on the classpath and indeed is one of several sample configuration files included with the simulator to illustrate its various features.

Listing : Running the Simulator

./sim.bat classpath:config-samples/console-only.xml

Note that if you specify a path that is not found on the classpath you’ll get the error shown in Listing 2 clearly indicating the problem which was a leading ‘/’ character in this case.

Listing :Unable to find resource Error

IllegalArgumentException: Unable to find resource '/config-samples/console-only.xml' on classpath.

Additionally, when running the simulator and accessing its console for the first time you will likely incur the error in Listing 3. This is due to the classpath in sim.bat not containing the correct path to the tools.jar file in your local java development kit. Correct sim.bat for your local environment and try again.

Listing :Unable to compile JSP Error

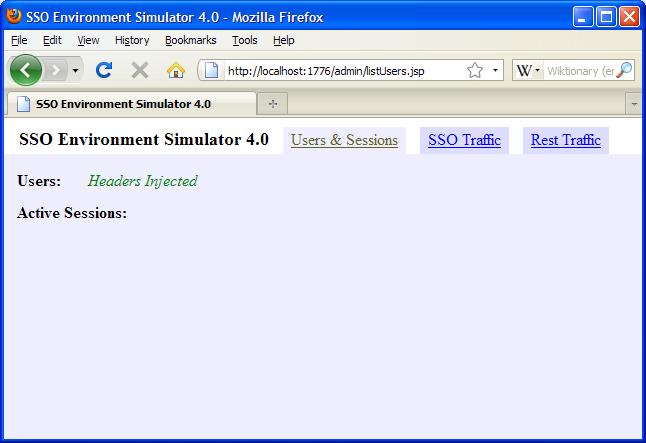
org.apache.jasper.JasperException: PWC6033: Unable to compile class for JSP

Assuming that you started it correctly you’ll see console output similar to that show in Listing 4. Point your browser to <http://localhost:1776/> and hit enter. You are presented with the simulator’s console with the User’s & Sessions tab selected as shown in Figure 3. The contents of the configuration file specified in Listing 1 are shown in Listing 5. Note that no users are defined in our configuration file nor did we declare any SSO traffic. Therefore, both the User & Sessions tab and SSO Traffic tab will appear empty for now. Lets discuss the elements found within this configuration file.

Listing : Successful startup console output

Using configuration file classpath:config-samples/console-only.xml  
2010-01-11 21:44:14.781::INFO: Logging to STDERR via org.mortbay.log.StdErrLog  
admin-rest port: 1776  
http proxy port: 80  
2010-01-11 21:44:14.843::INFO: jetty-6.1.7  
2010-01-11 21:44:14.890::INFO: Extract jar:file:/D:/saw/lib/appwrap-4.0.jar!/webapp to  
 C:\DOCUME~1\BOYDMR\LOCALS~1\Temp\Jetty\_0\_0\_0\_0\_1776\_webapp\_\_admin\_\_-usxbqe\webapp  
2010-01-11 21:44:15.359::INFO: Started [SocketConnector@0.0.0.0:1776](mailto:SocketConnector@0.0.0.0:1776)  
Started r-proxy on port 80

F igure : Console - Users Tab



## <config>

The root element of the simulator’s XML configuration is the *config* element. It supports the three attributes defined in Table 1. It also supports a number of nested elements as shown in Table 2. Note that the type for the port attributes must be an integer or an alias. Aliases are discussed in the section on **Aliases and Macros**.

Listing : Simple Config to Show Console Pages

<?xml version="1.0" encoding="UTF-8"?>

<config proxy-port="80" console-port="1776" allow-non-sso-traffic="false">

<console-recording sso="true" rest="true"/>

</config>

Table : Config - Attributes

|  |  |  |
| --- | --- | --- |
| **Atribute Name** | **Type** | **Description** |
| proxy-port | Integer or Alias | The port on which the simulated site will appear with all back-end applications mapped into its subdirectory space. Must be an integer. |
| console-port | Integer or Alias | The port on which the simulator’s console is located. Must be an integer. |
| allow-non-sso-traffic | [ ‘true’ | ‘false’ ] | Indicates if the simulator should allow for proxying non-SSO traffic as defined in section ?????. If allowed, then the simulator can be used a full http proxy and used by a browser by configuring the browser to use the simulator’s proxy-port as its http proxy. It defaults to false and caused the simulator to serve a 404 for any traffic that does not match the configured SSO traffic. |

Table :Config - Child Elements

|  |  |
| --- | --- |
| **Child Elements** | **See** |
| console-recording | Table 3 |
| sso-cookie | Table 4 |
| sso-header | ??? |
| sso-sign-in-url | ??? |
| sso-traffic | ??? |
| users | ??? |

## <console-recording>

This is an optional child element of config. It is an empty element and supports the attributes shown in Table 3. The console’s SSO Traffic and Rest Traffic tabs respectfully expose traffic that has passed through the simulator if recording is turned on either via the links within those pages or by this declaration. Recording for each is turned off by default.

Table : console-recording – Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Type** | **Description** |
| Sso | [ ‘true’ | ‘false’ } | Indicates if the console should record in-memory the traffic that is hitting the http-proxy port and display it on the SSO Traffic tab. Defaults to false. Can be ‘true’ or any other value which is interpreted as ‘false’. |
| Rest | [ ‘true’ | ‘false’ } | Indicates if the console should record in-memory the traffic that is hitting the rest API and display it on the Rest Traffic tab. Defaults to false. Can be ‘true’ or any other value which is interpreted as ‘false’. |

## Accessing the Rest API

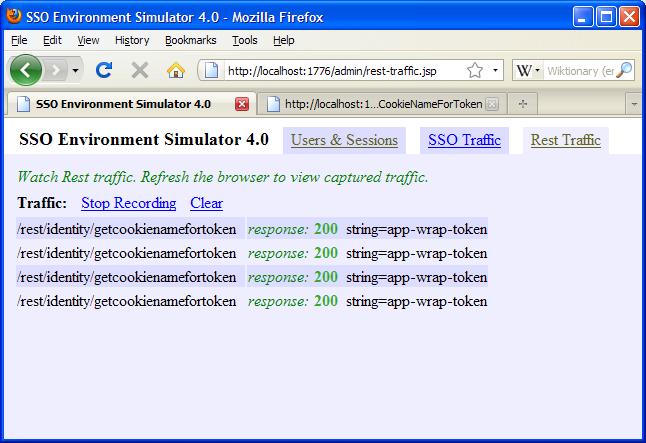
Even with the simple configuration shown in Listing 5 we can execute what is known as the rest API shown in Figure 1 : Church Single Sign-On Environment. Recall that applications can ask the policy server if a user can access a URL and thereby protect blocks of code or conditionally render screen elements. Another call in this API is the ability to ask the name of the SSO cookie. Either open a new browser tab or a new browser window and point it to:

<http://localhost:1776/rest/identity/getCookieNameForToken>

This will result in a single line of text, “string=app-wrap-token”. Congratulations, you just hit the REST API implemented by the simulator. This particular call returns the name of the cookie that should be set to hold the SSO session token value. Now return to your original browser window containing the console and select the Rest Traffic tab as shown in Figure 4. For each time that the API was accessed an entry was recorded and is displayed in the tab. The entries can be cleared with the Clear link or recording can be turned off with the Stop Recording link. If our console-recording element in our configuration file was missing or specified rest=’false’ then recording would have been disabled upon staring and the *Stop Recording* link would have been a *Start Recording* link.

Note that there is currently no automatic refresh mechanism for the console screens. So each time you view traffic in the console be sure to refresh the page to ensure that you are seeing everything that has passed through the simulator. Also note that for each API entry the lines alternate in background color to help identify all of the information for a single response. As will be seen, some responses consist of multiple lines in the console. Each response indicates the REST API that was accessed, the http response code, and the content of the response. We’ll revisit this output again later. Now lets add to our configuration.

Figure : Watching Rest API Traffic



## <sso-cookie>

Notice that the default cookie name returned from the simulator is “app-wrap-token”. The cookie used in the nextgen SSO environment is lds-policy. We can configure the name of the cookie with the *sso-cookie* element which is an optional child of the config element. It is an empty element and supports the attributes shown in Table 4. The values used to match the cookie used in the nextgen SSO environment is show in Listing 6.

Table : sso-cookie – Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | | **Type** | **Description** |
| Name | Literal text or Alias | | Indicates the name of the SSO token that will be returned from the getCookieNameForToken REST API call and looked for they the simulator’s proxy when performing URL enforcement and user header injection. Defaults to “app-wrap-token”. |
| Domain | Literal text or Alias | | Indicates the domain of the cookie that will be set by the simulator in the select user page. Defaults to “.lds.org”. |

Listing : Declaring our SSO Cookie Name and Domain

<sso-cookie name="lds-policy" domain=".lds.org"/>

Running the simulator with the command line shown in Listing 7 uses a configuration file nearly identical to that of Listing 5 but with the following sso-cookie directive as a child of the config element. Hitting the REST API again returns the new value showing that the cookie name is now set to “lds-policy”. This will be the value used in all examples hereafter.

Listing : Test of sso-cookie Directive

./sim.bat classpath:config-samples/console-only-lds-policy-cookie.xml

# Aliases and Macros

Before continuing on with further directives lets first discuss aliases and their use via macros in other directives. The simulator’s parser supports an XML processing instruction with a name of: “alias”. This instruction defines a key and value pair enabling use of references to values to be used in multiple places in the document. This instruction supports two patterns for defining alias values; literal text and classpath file references. In either case macro references for other previously defined aliases can be embedded within the literal text source to inject those values. Remembering that XML processing instructions start with “<?” and end with “?>” the formal definition of the alias declaration an macro references is define in Listing 8. Any configuration attributes defined in this document indicating that they can accept an alias can have their entire value consist of a macro or can have one or more macros embedded within literal text and such macros will be replaced with the alias values.

Listing : Alias/Macro Syntax Defined

[1] alias := “<?alias ” <name> “=” [<classpath-ref> | +[<clean-text> | <macro>]] “?>”  
[2] name := <clean-text>  
[3] clean-text := << any characters except patterns “<?”, “?>”, “{{“, or “}}” >>  
[4] classpath-ref := “classpath:” + << path to file on classpath >>   
[5] macro := “{{“ <name> “}}”

Examples of valid versions of this processing instruction are:

<?alias rest-port=1776?>

<?alias console-port={{rest-port}}?>

<?alias rest-api={{rest-port}}/identity?>

<?alias marks-lds-account-id=000111222333?>

<?alias is-ip-moderator=classpath:is-ip-mod.xml?>

The first four instructions use literal text values although the second and third’s values also contain a macro reference to the first alias. Hence, the console-port alias will have the same value as that of the rest-port alias and the rest-api value will be “1776/identity”.

The fourth instruction uses a classpath file reference. Suppose that the contents of that file contained the following text. What this text means and why it is used will be explained in section ???????.

<HasLdsAccountId>

<LdsAccount id='3431968674741880'/>

<LdsAccount id='{{marks-lds-account-id}}'/>

</HasLdsAccountId>

Upon processing the is-ip-moderator instruction the file will be searched and loaded as the value for that alias. Additionally, if its contents contain any alias macros they will be resolved. Hence the value of the is-ip-moderator alias will be:

<HasLdsAccountId>

<LdsAccount id='3431968674741880'/>

<LdsAccount id='000111222333'/>

</HasLdsAccountId>

## Preparing for SSO Traffic – etc/hosts

Now lets look at enabling some SSO traffic through the simulator so that we can see traffic recorded on the console’s *SSO Traffic* tab. Doing so will expose a unified site for our chosen domain but does so by proxying requests for that site to one or more back-end applications. To begin, recall from the section on *Cookies and Site Domains* that *a cookie will only be submitted for requests to domains that match the domain for which it was set*. All of the examples from here onward will use the sso-cookie directive from Listing 6. Therefore, we must add a declaration to our etc/hosts file so that we can use a sub-domain of “.lds.org” in the browser causing requests from the browser to be resolved to our localhost *and* the browser will submit our cookie with such requests. On window’s machines this file is located in:

C:\windoes\system32\drivers\etc\hosts

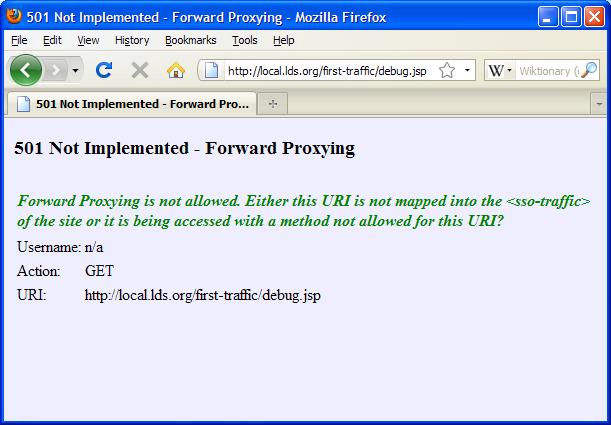
Our domain will be local.lds.org. To enable our machine to resolve this domain to our local box, add the line shown in Listing 9.

Listing : etc/hosts Enabling of local.lds.org

127.0.0.1 local.lds.org

Once saved, point your browser to <http://local.lds.org/first-traffic/debug.jsp>. You will receive a page as shown in Figure 5. As noted, this indicates that either forward proxying is turned off or our resource, “*local.lds.org/first-traffic/debug.jsp*”, hasn’t been configured in the simulator to be routed to a back-end application. This brings us to the sso-traffic element.

Figure : 501 Not Implemented



## <sso-traffic>

Since the simulator is both a forward and reverse proxy it must be told which requests passing through its proxy port should be considered SSO Traffic, have headers injected, be subject to access restrictions, and optionally be proxied to a back-end application. The <sso-traffic> tag supports no attributes but its nested elements listed in Table 5 are the mechanisms where-by we identify such traffic to the simulator and therby subject that traffic to SSO features. The by-site element is for mapping back end applications into a site’s sub-paths. The by-resource element is used for declaring fine-grained permission policies.

Table : sso-traffic - Child Elements

|  |  |
| --- | --- |
| **Child Elements** | **See** |
| by-site | ??? |
| by-resource | ??? |

Before we get into the details, we need some back-end application to which to route traffic. It turns out that the simulator includes a useful debugging page that is part of the console at the link in Listing 10. We will use this page to simulate an application to be exposed in our “site”.

Listing : Debug page in Simulator

<http://localhost:1776/admin/debug.jsp>

## <by-site>

Recall that in Listing 9 we defined **local.lds.org** as the domain of our site. So we want traffic destined to that domain to be considered SSO traffic. In the http requests passing through the proxy, the ***host*** header carries the domain and optionally a colon and port of the targeted site. For the default http port, port 80, the port is left off of the host header and is understood to be 80. The by-site element is used to declare the host, port, and scheme that packets must match to be subject to the policies of its nested elements. It supports the attributes listed in Table 6 and the child elements listed in Table 7.

Table : by-site – Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Type** | **Description** |
| scheme | Optional Literal text | Taken as-is without any validation. If not specified then it defaults to http which is all that makes sense until the simulator supports https. |
| host | Literal text or Alias | The value that will be compared to the host portion of the host header to determine if the packet matches this site and should be subject to its policies. |
| Port | Integer or Alias | The value that will be compared to the port portion of the host header to determine if the packet matches this site and should be subject to its policies. |

Table : by-site - Child Elements

|  |  |
| --- | --- |
| **Child Elements** | **See** |
| cctx-mapping | ??? |
| unenforced | ??? |
| Allow | ??? |

To illustrate all supported nested elements of the by-site element lets suppose that we want our site to contain the debug.jsp page at the two sub-paths shown in Listing 1. The “public” path will be unenforced allowing packets through whether authenticated or not. The “secure” path will require a user to authenticate before packets will make it through the simulator and arrive at our console page.

Listing : URLs for our First Site

<http://local.lds.org/public/debug.jsp>  
<http://local.lds.org/secure/debug.jsp>

The configuration that exposes our debug.jsp page at these two paths is shown in listing Listing 12. Don’t worry. We’ll be going over all of the elements listed here that have not been covered. First, look at our

Listing : Dual Debug Page Site Example

<?alias rest-port=1776?>

<?alias console-port={{rest-port}}?>

<?alias http-port=80?>

<config proxy-port=*"{{http-port}}"* console-port=*"{{console-port}}"*>

<console-recording sso=*"true"* rest=*"true"*/>

<sso-cookie name=*"lds-policy"* domain=*".lds.org"*/>

<sso-sign-in-url

value=*"http://local.lds.org:{{console-port}}/admin/selectUser.jsp"*/>

<sso-header name=*"policy-service-url"*

value=*"http://labs-local.lds.org:{{rest-port}}/rest/"*/>

<sso-traffic>

<by-site host=*"local.lds.org"* port=*"80"*>

<cctx-mapping

cctx=*"/public/\*"*

thost=*"127.0.0.1"*

tport=*"{{console-port}}"*

tpath=*"/admin/\*"*/>

<unenforced cpath=*"/public/debug.jsp"*/>

<cctx-mapping

cctx=*"/secure/\*"*

thost=*"127.0.0.1"*

tport=*"{{console-port}}"*

tpath=*"/admin/\*"*/>

<allow action=*"GET,POST"* cpath=*"/secure/debug\*"*/>

<allow action=*"GET,POST"* cpath=*"/secure/debug\*?\*"*/>

</by-site>

</sso-traffic>

<users>

<user name=*"ngia"* pwd=*"pwda"*/>

<user name=*"ngib"* pwd=*"pwdb"*/>

</users>

</config>

have defined in our etc/hosts file the console’s debug page in this example, we use the cctx-mapping element that supports the attributes defined in Table 5. This element does not support any nested children.

Table : cctx-mapping – Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Type** | **Description** |
| cctx | Literal text or Alias | The top level sub-path ending with an asterisk that URLs for the site must start with to match this rule and be routed to the targeted host and port. Ex: “/temples/\*”. |
| thost | Literal text or Alias | The target host to which the traffic will be proxied. This can be by IP address or dns name. |
| tport | Literal text or Alias | The port on the targeted host. This must be an integer. |
| tpath | Literal text of Alias | The top level sub-path ending with an asterisk that URLs should have when hitting the targeted host and port. In the request passing through the simulator, if this pattern is different from the value of cctx, then it replaces the cctx value in the requests URL thus “rewriting” thereby implementing translation between the canonical and application URL space. |

Table : cctx-mapping Attributes

|  |  |  |
| --- | --- | --- |
| Attribute Name | Type | Description |
| cctx | Literal text or alias |  |
|  |  |  |
|  |  |  |

The DTD for the configuration file is found in APPENDIX A - DTD. :

be the same with an embedded alias Lets look at an example. Suppose that we desire to use the simulator’s built-in sign-in page in our configuration for testing an application. The configuration could appear as follows. Notice how the console-port alias uses the rest-port alias and is then used in the config element to tell the simulator to listen on that port for its console and it is also used in specifying via the sso-sign-in-url element where the simulator’s agent should redirect traffic if a session is required before allowing access to a protected resource.

<?alias rest-port=1776?>

<?alias console-port={{rest-port}}?>

<?alias site-port=80?>

<config proxy-port="{{site-port}}" console-port="{{console-port}}">

<sso-sign-in-url value="http://labs-local.lds.org:{{console-port}}/admin/selectUser.jsp"/>

...

**Listening Ports**

Although the simulator is an http proxy supporting both forward and reverse proxying its main goal is to provide reverse proxying to applications running on the same box allowing developers to simulate the full SSO environment. without needing to be connected the latter to simulate our deployment environments at the church where numerous different technologies and server clusters can appear to be on the same site but located at different points of the and it provides a console both for monitoring traffic, users, and sessions and providing an implementation of the opensso policy server rest API. Hence it uses two ports one for http proxying and one for its console and rest service. The ports on which the simulator listens are defined on the XML document’s root element, config, which has two required attributes:

Proxy-port := [ integer | alias-macro ]

.for selecting a user or even an already established session. Alternatively, the The simulator requires two ports to run. The config elementValues for macros can be it to be maintained are ignored. either a file can be speand can be specified as aby starting the simulator on the command line XML file specified when

# Rewriting Redirects

Applications should be designed as much as possible to be cognizant of running in an environment where there is a difference between canonical and application URL space and and ensure that any redirects issued by it are canonical. But sometimes that is not possible when using third party libraries. In such cases a rewriting proxy like apache or its IBM variant HIS can be used to fix application space redirects that have no meaning in the canonical space where the browser is running..

Section 14.30 of RFC 2616 defines the “Location” response header used to convey a redirect instruction to the browser. It states that, “The field value consists of a single absolute URI.” Accordingly, the servlet specification dictates that servlet containers must convert relative URIs passed to the sendRedirect method of HttpServletResponse to absolute values. As noted in the 1.4 version of this method’s javadoc, “If the location is relative without a leading '/' the container interprets it as relative to the current request URI.” This means that it will prefix its path as needed including using the java context root. If the canonical space is different from the application space this can be a problem.

Lets look at an example. Suppose we have some bishop application written in java and having a java context of nextgen-bishop. We have taken care to ensure that we always issue absolute redirects that are cognizant of canonical space. But suppose further that it uses a third party ajax library that has embedded relative redirects over which we have no control. These will result in Location headers like so:

Location: http://labs-local.lds.org/nextgen-bishop/some/resource.html

But suppose that our application has been deployed at a canonical space of /bishop since we do not wish to have the “nextgen” identifier showing up in the browser. This will require that our Location header be modified to appear as follows:

Location: http://labs-local.lds.org/bishop/some/resource.html

The mechanism for achieving this in apache and HIS is the proxyPassReverse directive. The simulator supports this feature with its rewrite-redirect directive. There is no limit tn the number of such directives. They must be declared as a child of the sso-traffic element and have the following structure:

<rewrite-redirect from='from-value' to='to-value' />

Although it performs the same functionality as apache’s proxyPassRevers it varies in its declaration syntax by requiring fully qualified prefixes including the scheme, host, and port of the URI. Specifically, the Location header value will only be rewritten if it starts with the from-value. Further, only the the from-value portion of the Location header value will be replaced with the to-value portion. For example, to fix the improper redirect above to the correct value also shown above the following directive would be declared as a child element of the sso-traffic elements:

<rewrite-redirect from='http://labs-local.lds.org/nextgen-bishop/'   
 to='http://labs-local.lds.org/bishop/' />

# Rewritting Cookie Paths

Related to redirects are path based cookies. The java servlet specification dictates that the name of the session tracking cookie be “JSESSIONID”. Let suppose that we have two distinct java applications implementing different portions of our site at labs.lds.org. One will reside at /bishop in the canonical space with a java context root in the application space of /nextgen-bishop and traffic routed with suitable URL rewriting of requests as shown in ???? and Location header rewriting on redirect responses as shown in **Rewriting Redirects**. The other application will reside at /mls/mbr in the canonical space with a java context of /mls-membership in the application space with similar rewrites for requests and responses.

Now consider that both make use of java’s HttpSession functionality. When /bishop is accessed a JSESSIONID cookie is set in the browser to track the user’s session. If the user then accesses /mls/mbr the JSESSIONID cookie is submitted for the domain but does not match a session in that application. Accordingly, the application sets a new JSESSIONID cookie in the browser and any information in the /bishop application is now orphaned. Upon returning to /bishop the application sees a JSESSIONID that does not match any of its sessions and hence starts a new session and sets the JSESSIONID cookie accordingly in the browser orphaning the session from /mls/mbr.

To resolve such a problem typically the container specific functionality must be relied upon to set a path for the cookie which usually is the servlet context of the application. That means that for the /bishop application the JSESSIONID path will be /nextgen-bishop and that for the /mls/mbr application will be /mls-membership. Since the applications will be accessed from the browser using their canonical paths the cookie will never be submitted for either application.

Apache and IHS support directives for rewriting such cookie paths. The simulator provides a similar mechanism through its rewrite-cookie directive. It is similar in form to rewrite-redirect and must also be declared as a child of the sso-traffic element. It has the structure as shown in the example below which would handle the cookie path rewrites needed for our two applications:

<rewrite-cookie from-path='/nextgen-bishop' to-path='/bishop' />

<rewrite-cookie from-path='mls-membership' to-path='/mls/mbr' />

It is important to note that these again are prefix values. If a container supported setting a multi-level path like /mls-membership/app-1 then the rewriting only rewrites the matching portion and leaves the unmatched portion unchanged. For such a case the resulting cookie path of “/mls/mbr/app-1” would result.

Scratch pad------

The mapping of incoming traffic for a single site to various backend applications is known as reverse proxying. The simulator supports this feature through configuring what are known as context mappings as is explained in section ????????.

Forward proxying is when a user agent can not go to servers directly but is told it must route all traffic through an http proxy and the proxy will hit the resource for a given URL and spool the response back to the browser. Although disabled by default, the simulator can be configured to allow forward proxying for testing certain SSO simulations which are discussed in section ???????.

# <users>

Declares where user attributes are obtained. If there is no source attribute then nested user elements are the only source of user information… TBC

Dynamic users versus static users,

Dynamic attributes versus static values

Which takes precedence?

Table : users – Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute Name** | | **Type** | **Description** |
| source | (optional) URL of source for user header attributes | | If included must be the URL of a source that can return XML containing the user attribute values to be injected for SSO policy header. Currently, this only supports the coda XML dialect returned from <http://tech.lds.org/coda/services/1/member/osso/pholder> where pholder is the ID of the signed-in user. The URL can contain a macro of %%username%% that will be replaced with the user id of the currently signed-in user. Attributes are cached for the duration of the user’s session and reloaded each time they log in. |

# APPENDIX A - DTD

<!ELEMENT config ( sso-cookie, sso-sign-in-url, sso-header\*, sso-traffic, users ) >

<!ATTLIST config allow-non-sso-traffic CDATA #IMPLIED >

<!ATTLIST config console-port CDATA #REQUIRED >

<!ATTLIST config proxy-port CDATA #REQUIRED >

<!ELEMENT sso-cookie EMPTY >

<!ATTLIST sso-cookie domain CDATA #REQUIRED >

<!ATTLIST sso-cookie name CDATA #REQUIRED >

<!ELEMENT sso-sign-in-url EMPTY >

<!ATTLIST sso-sign-in-url value CDATA #REQUIRED >

<!ELEMENT sso-header EMPTY >

<!ATTLIST sso-header name CDATA #REQUIRED >

<!ATTLIST sso-header value CDATA #REQUIRED >

<!ELEMENT sso-traffic (( by-site|by-resource)\*) >

<!ELEMENT by-site ( allow | cctx-mapping | unenforced )\* >

<!ATTLIST by-site host CDATA #REQUIRED >

<!ATTLIST by-site port CDATA #REQUIRED >

<!ATTLIST by-site scheme CDATA #IMPLIED >

<!ELEMENT by-resource EMPTY >

<!-- one of allow or unenforced is required. it is an error to exclude both. -->

<!ATTLIST by-resource allow CDATA #IMPLIED >

<!-- conditions are optional. If not specified then the only requirement for

accessing the cpath is that the user be signed-in -->

<!ATTLIST by-resource condition CDATA #IMPLIED >

<!ATTLIST by-resource unenforced CDATA #IMPLIED >

<!ATTLIST by-resource uri CDATA #REQUIRED >

<!ELEMENT allow EMPTY >

<!ATTLIST allow action CDATA #REQUIRED >

<!-- conditions are optional. If not specified then the only requirement for

accessing the cpath is that the user be signed-in -->

<!ATTLIST allow condition CDATA #IMPLIED >

<!ATTLIST allow cpath CDATA #REQUIRED >

<!ELEMENT cctx-mapping EMPTY >

<!ATTLIST cctx-mapping cctx CDATA #REQUIRED >

<!ATTLIST cctx-mapping thost CDATA #REQUIRED >

<!ATTLIST cctx-mapping tpath CDATA #REQUIRED >

<!ATTLIST cctx-mapping tport CDATA #REQUIRED >

<!ELEMENT unenforced EMPTY >

<!ATTLIST unenforced cpath CDATA #REQUIRED >

<!ELEMENT users ( user+ ) >

<!-- session-timeout-seconds defaults to 300 seconds (five minutes) -->

<!ATTLIST users session-timeout-seconds CDATA #IMPLIED >

<!ELEMENT user ( sso-header+ ) >

<!ATTLIST user name CDATA #REQUIRED >

<!ATTLIST user pwd CDATA #REQUIRED >