# Connector 4.0 Developer's Guide:

Google provides search appliance connectors for non-HTTP content. This guide explains how to write your own connector using Connectors Framework 4.0 as a model. These connectors were called adaptors before they were released to the public. This is why the term *adaptor* appears in classes, packages and some documentation.

## Table of Contents

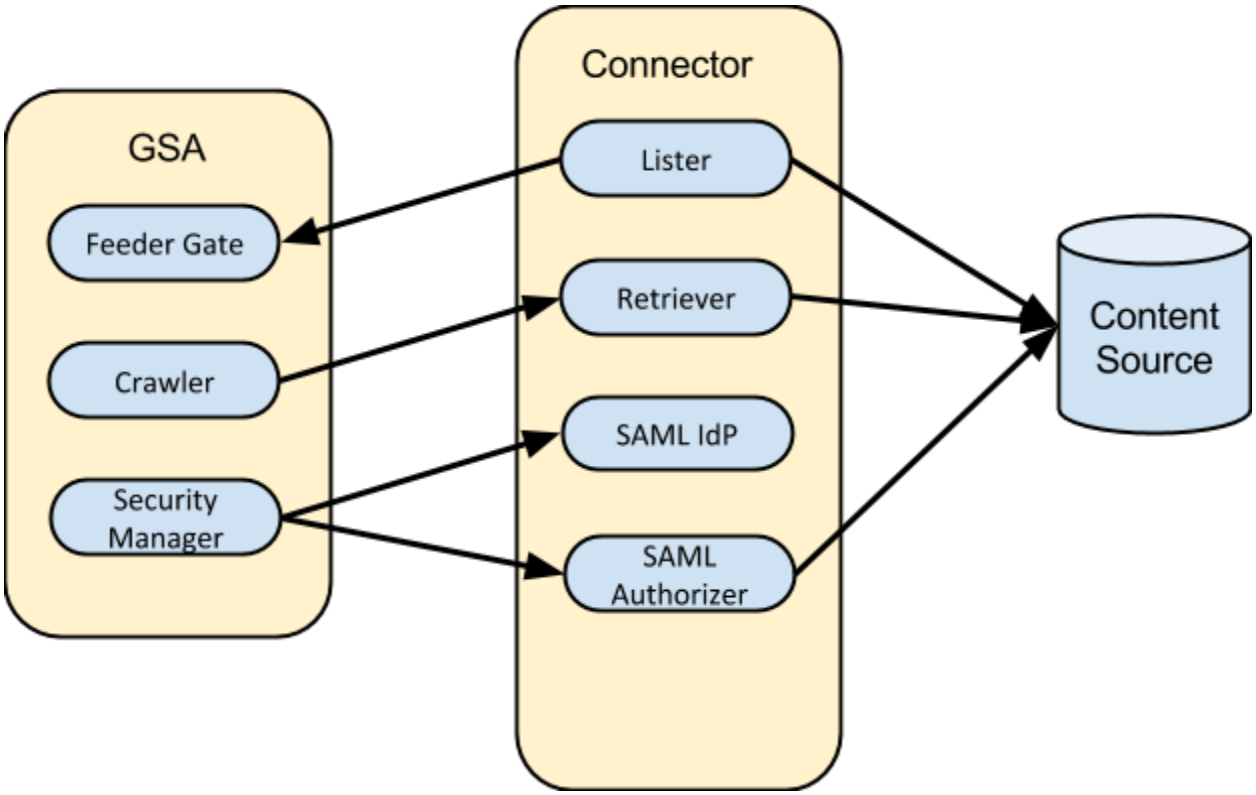
# Audience

This document is intended for Java developers. It assumes that you are familiar with Google Search Appliance, Eclipse, and concepts for Connectors 4.0. For more information about Google Connectors for non-HTTP content, visit the Connectors 4.0 repository. After reading this document, you will be able to write your own connector.

# How it works

A Connector tells the search appliance where to locate documents, and provides the contents of those documents when the search appliance requests them. Connectors have the following components:

- A Document Lister
- A Document Retriever
- A SAML IdP endpoint  (optional)
- A Document Authorizer (optional)




## Write your first connector

Let's build the simplest connector first.

### Set up HelloWorldConnector project

The following instructions apply to the Eclipse Juno release. In this tutorial we'll use a Hello World connector as an example. Follow these steps to set up your project:

1. Create a new Java project in Eclipse called **HelloWorldConnector**.
2. Create a **lib** directory under this project.
3. Download the binary and unzip it.
4. Copy the connector v4 libraries, to the **lib** directory under the **HelloWorldConnector** project.
5. In the left navigation pane, right click the project name **HelloWorldConnector**, and select **Refresh** from the menu. This causes the project to pick up the newly added JAR files.
6. Right click the **HelloWorldConnector** project name, and choose **Properties** from the menu.
7. In the **Properties** window, navigate to **Java Build Path** > **Libraries**, and choose **Add JARs**.

8. Navigate to the **lib** directory, select the newly added JAR files, and click **OK**.

Now that you have set up the project, you can add the main connector class, which inherits from and extends **`com.google.enterprise.adaptor.AbstractAdaptor`**:

```
public class HelloWorldConnector extends AbstractAdaptor {

	@Override
	public void getDocContent(Request request, Response response)
			throws IOException, InterruptedException {
		// TODO Auto-generated method stub

	}

	@Override
	public void getDocIds(DocIdPusher pusher) throws IOException,
			InterruptedException {
		// TODO Auto-generated method stub

	}
}
```

## The Lister

The first thing a connector must do is identify document locations to the search appliance using the Lister. In the Connector interface, the Lister is the **getDocIds()** method.

```
public void getDocIds(DocIdPusher pusher) throws IOException, InterruptedException
```

The Lister method is to perform traversing the entire content repository and triggering the search appliance to retrieve `DocIds`. The Lister method can be written to simply feed `DocIds` one by one to the search appliance, as shown in the following example.

```
public void getDocIds(DocIdPusher pusher) throws InterruptedException {
    ArrayList<DocId> mockDocIds = new ArrayList<DocId>();
    mockDocIds.add(new DocId("doc_1"));
    mockDocIds.add(new DocId("doc_2"));
    pusher.pushDocIds(mockDocIds);
}
```

The Lister method lists all available documents so that the search appliance can index them using the **Retriever** method.

### The Retriever

```
public void getDocContent(Request request, Response response)
                    throws IOException, InterruptedException
```

The search appliance crawls the DocIds URLs sent by the Lister method connector's listers similar to the way it crawls just like crawling any other web feeds. It also crawls already indexed documents from the connector. The crawling is multi-threaded. The connector is essentially a web server that works like a proxy: it channels the call to thefrom HTTP handler via to the method `getDocContent()` of the Adaptor interface. This method then gets the document from the content repository.

```
public void getDocContent(Request req, Response resp) throws IOException {
       DocId id = req.getDocId();
       if ("1001".equals(id.getUniqueId())) {
              Writer writer = new OutputStreamWriter(resp.getOutputStream());
              writer.write("Menu 1001 says latte");
              writer.close();
       } else if ("1002".equals(id.getUniqueId())) {
              Writer writer = new OutputStreamWriter(resp.getOutputStream());
              writer.write("Menu 1002 says cappuccino");
              writer.close();
       }
}
```

### Run the Connector

First, you need to create a text file "adaptor-config.properties" under the root directory of the project "HelloWorldConnector"; add the location of the search appliance:

gsa.hostname=<host name or IP address>

Now you can run the connector in Eclipse or from command line. You will first see metadata-and-url feeds show up, then the indexed documents will show up in **Index** -> **Index Diagnostics** with the following format:

http://<connector-hostname-or-ip>:5678/1001
http://<connector-hostname-or-ip>:5678/1002

As you can see, the URLs of the documents fed by the connector have the following format:

http://<connector-hostname-or-ip>:<port>/doc/<docid>

`<docid>` can be anything that uniquely identifies a document: a number, name, alphanumeric value, or even a URL.

If you want to take a look at what the search appliance gets from the Retriever, you can open a browser and point to the URL. You need to configure the connector to allow hosts other than the configured search appliance to access:

server.fullAccessHosts=<host name or IP address of the work station where you want to call the retriever from browser>

## The Lister/Retriever Model - Continued

Of course, connectors in real world are far more complicated than the simple example above. Before we dive into more details of the Lister/Retriever model, we need to be familiar with some helper classes from the connector library.

### Config

By using this class, you can design your connector to interact with settings in adaptor-config.properties. You can add custom entries that are only meaningful to your adaptor.

### AdaptorContext

This interface and its implementation in connector framework is designed for your adaptor to communicate with other classes in the adaptor library. For example, your connector can get hold of the **Config** object by calling AdaptorContext.getConfig(). Your connector has to implement an init() method where the object of AdaptorContext is passed in.

```
public void init(AdaptorContext context) throws Exception;
```

There are other classes that you will implement and register with AdaptorContext so that these classes can be invoked by the framework for the relevant functions.

### Controlling full listing

The Lister is used to perform a full listing - traverses the entire content repository. This full Lister is run on a schedule, which you can control by adding this entry in adaptor-config.properties:

**adaptor.fullListingSchedule**

By default, getDocIds() is invoked when the connector is started. You can disable it by adding this entry:

**adaptor.pushDocIdsOnStartup=false**

Full listing is not efficient since it has to go over the whole repository. If there is a way that you can discover updated or added documents, you can use incremental listing instead.

### Listing changed documents

A connector can perform incremental listing - discover recently changed documents and inform the appliance. In order to do that, you have to implement the following interface:

```
public interface PollingIncrementalLister {
  public void getModifiedDocIds(DocIdPusher pusher) throws IOException,
      InterruptedException;
}
```

Besides implementing the only method `getModifiedDocIds()`, you also need to register the incremental lister to the `AdaptorContext`. You can do so by overriding the `init()` method of the adaptor:

```
public class HelloWorldConnector extends AbstractAdaptor
       implements PollingIncrementalLister {
       ...
       @Override
       public void init(AdaptorContext context) throws Exception {
       context.setPollingIncrementalLister(this);
       ...
 }
```

You can use various approaches to discover modified documents: by remembering the last running time of the full crawl; by storing a checkpoint in a file, or by relying on the features of the

target content source. It's recommended that the connector be kept stateless, or very light weight at least.

Threads for `getModifiedDocIds()` and `getDocIds()` won't run at the same time.

### More control over feeds

DocIds are converted to records in feeds. However, `DocIdPusher.pushDocIds()` doesn't give you a lot of control. You can use the following method from `DocIdPusher`:

```
public Record pushRecords(Iterable<Record> records)
      throws InterruptedException;
```

`Record` is a class that holds attributes of a record in feed. It uses a Builder pattern to set the attibutes, just like many other classes in Connector Framework.

```
DocIdPusher.Record record
             = new DocIdPusher.Record.Builder(virtualServerDocId)
             .setCrawlImmediately(true).build();
pusher.pushRecords(Collections.singleton(record));
```

### Handling deleted documents

The search appliance checks all documents in the index and updates them when it detects changes. When a document has been removed from the content source, you inform the search appliance with the `respondNotFound()` method:

```
public void getDocContent(Request req, Response resp) throws IOException {
    ...
    resp.respondNotFound();
}
```

### What if documents have not changed?

The search appliance crawling process avoids retrieving a document if it's unnecessary to do so. One way it does this is to use the "`Last-Modified`" header in the HTTP response. If a document is indexed with this header set, the search appliance will sends out an HTTP request with the "`If-Modified-Since`" header. This feature improves connector efficiency by

identifying whether a document has changed from the content source. You use the following method of the class `Response` to send in `Last-Modified` header:

```
public void setLastModified(Date lastModified);
```

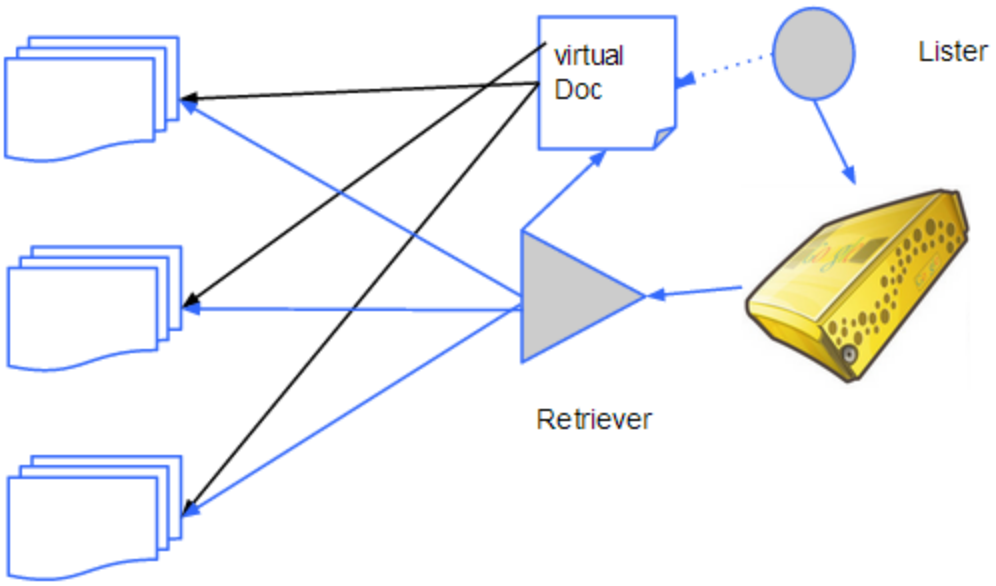
Use the following  code to tell the search appliance that nothing has changed. You must verify that  the search appliance is sending the "`If-Modified-Since`" header. If the search appliance does not use the `If-Modified-Since` header, and the search appliance uses `respondNotModified()` an error occurs in the index.

To help determine if a document has changed, you can use a function named `hasChangedSinceLastAccess()` that compares the date specified by the `If-Modified-Since` header with the last modified date of the document .

```
public void getDocContent(Request req, Response resp) throws IOException {
	DocId id = req.getDocId();
	Date lastModified;
	... //retrieve lastModified Date of the document
	if (!req.hasChangedSinceLastAccess(lastModified)) {
		//if document has not changed
		resp.respondNotModified();
	}
}
```

### Graph Traversal - Using Retriever as Lister

The Lister is a nice way to send all the "virtual" URLs to the appliance, but it's not the only way. There are times when this is not the best approach. For example, if the connector is used on a large repository, it could take a long time to retrieve the  unique document identifiers. The thread will block on `getDocIds()` for a long time. If anything breaks during the document ID retrieval process, , the process  would have to restart from beginning.




All the links contained in documents retrieved by the search appliance will be crawled if they are allowed by the “Follow and Crawl” patterns. You  can leverage this and use “virtual” documents to list all documents instead of using `getDocIds()`. In other words, you can use `getDocIds()` to specify the “virtual” IDs of documents.  documents. In the `getDocContent()`, you return these “virtual” documents with links to actual documents to be indexed.

```
public void getDocIds(DocIdPusher pusher) throws InterruptedException {
       ArrayList<DocId> mockDocIds = new ArrayList<DocId>();
       mockDocIds.add(new DocId("vdoc_1"));
       ...
       pusher.pushDocIds(mockDocIds);
}

public void getDocContent(Request req, Response resp) throws IOException {
       DocId id = req.getDocId();
       if ("vdoc_1".equals(id.getUniqueId()) {
              Writer writer = new OutputStreamWriter(resp.getOutputStream());
              writer.write("<!DOCTYPE html>\n<html><body>");
              writer.write("<a href=\"1001\">doc_1</a>");
              writer.write("<br>");
              writer.write("<a href=\"1002\">doc_2</a>");
              writer.write("<br>");
              writer.write("<a href=\"1003\">doc_3</a>");
              writer.write("<br>");
```

```
            writer.write("</body></html>");
            writer.close();
        }
}
```

This approach is more scalable especially when the content source has a hierarchical structure. The connector can create one virtual document per level, and print links to all files under it, including virtual documents for the sub-directories.

There are two potential problems that need to be avoided:

- The URLs of these “virtual” documents should not conflict with the URLs of actual documents. Both these “virtual” documents and URLs of actual documents start the connector’s host name. You need to design the rules of DocIds to avoid any conflicts.
- When users perform a  search, these “virtual” documents should not show up in search results. This can be resolved by specifying exclusion patterns in the search appliance’s collections.

### Sending additional information

The Retriever is not only used to get the content of a document, but also to send other information, such as metadata and ACL.

To add metadata to a document, you can use the the following method of `Response`:

```
public void addMetadata(String key, String value);
```

To set the display URL and other attributes:

```
public void setDisplayUrl(URI uri);
public void setCrawlOnce(boolean crawlOnce);
public void setLoc(boolean lock);
```

These additional attributes are sent in as custom headers, for example:

```
X-gsa-external-metadata: key1=value1,key2=value2
X-gsa-doc-controls: display_url=
X-gsa-doc-controls: crawl_once=false
X-gsa-doc-controls: lock=false
```

## Secure Search with ACL

### Sending ACLs for documents

Document ACLs are sent along with document content by the Retriever method. You can set all the attributes for an ACL first - groups, users, inheritance and case sensitivity. Then you can use the following method from `Response` object to set the ACL on a document:

```
public void setAcl(ACL acl);
```

The ACL objects are designed to be immutable. The Builder pattern is used to construct the ACLs. For example:

```
public void getDocContent(Request req, Response resp) throws IOException {
	...
	ArrayList<Principal> permits = new ArrayList<Principal>();
	permits.add(new UserPrincipal("user1", "Default"));
	permits.add(new GroupPrincipal("group1", "Default"));
	ArrayList<Principal> denies = new ArrayList<Principal>();
	denies.add(new UserPrincipal("user2", "Default"));
	denies.add(new GroupPrincipal("group2", "Default"));
	resp.setAcl(new Acl.Builder()
		.setEverythingCaseInsensitive()
		.setPermits(permits).setDenies(denies).build());
}
```

### Supporting ACL inheritance

To specify that a document inherits ACLs from another document, add the “inheritance type” and “inherit from” to the ACL. For example:

```
public void getDocContent(Request req, Response resp) throws IOException {
	...
	resp.setAcl(new Acl.Builder()
		.setEverythingCaseInsensitive()
		.setInheritFrom(siteDocId)
		.setInheritanceType(Acl.InheritanceType.AND_BOTH_PERMIT);
		.setPermits(permits)
```

```
            .setDenies(denies).build());
}
```

### Inheriting from Named Resource

Most of the time, ACLs are tied to documents or folders. Documents inherit ACLs from their parents. However, there are content sources that have ACLs in separate objects from documents. Google Search Appliance supports this scenario with Named Resources, which are ACLs without documents. Named resources can be pushed to the search appliance as feeds using the following method from the interface `DocIdPusher`,

```
public DocId pushNamedResources(Map<DocId, Acl> resources) throws InterruptedException,
```

With this method, named resources are fed to the search appliance:

```
public void getDocIds(DocIdPusher pusher) throws InterruptedException {
      ...
      HashMap<DocId, Acl> aclParent = new HashMap<DocId, Acl>();
      ArrayList<Principal> permits = new ArrayList<Principal>();
      permits.add(new UserPrincipal("user1", "Default"));
      aclParent.put(new DocId("AclNode1"), new Acl.Builder()
            .setEverythingCaseInsensitive()
            .setPermits(permits)
            .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
            .build());
      pusher.pushNamedResources(aclParent);
      ...
}
```

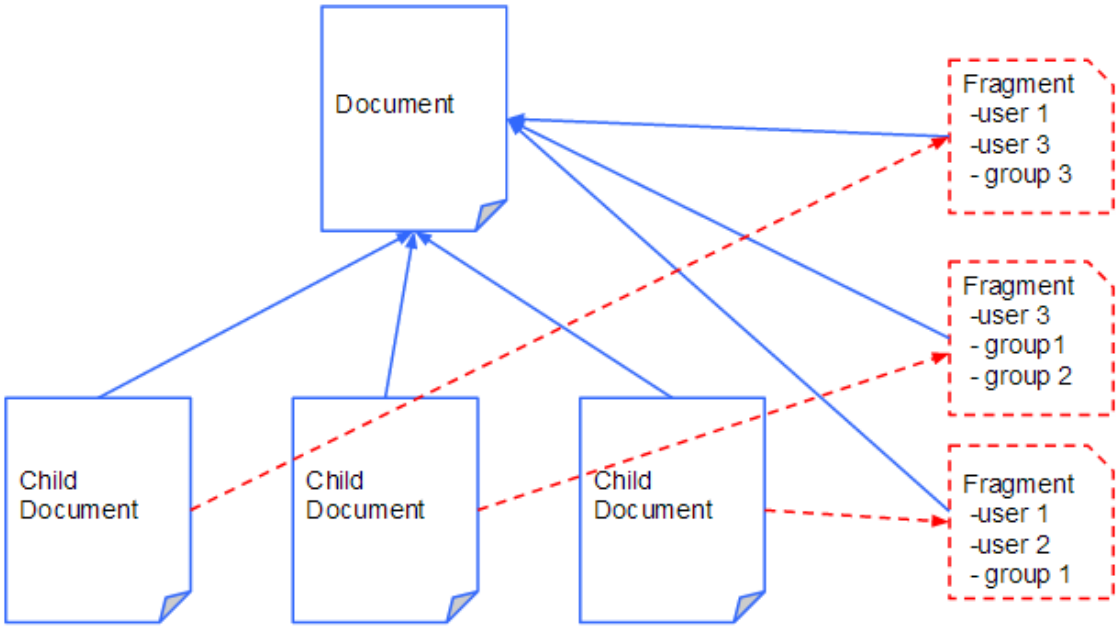
The feed looks as follows. . The ACL element is a child of Group element. Refer to the Feeds developer guide for details.

```
<group>
<acl url="https://connector-host:5678/doc/AclNode1"
inheritance-type="parent-overrides">
<principal scope="user" case-sensitivity-type="everything-case-insensitive"
access="permit">user1</principal>
```

```
</acl>
</group>
```

### Inheriting from Fragments

"Fragment" is a connector term for having a Named Resource that starts off with the same URL as that of  an existing document. Fragments are useful when a single ACL isn't enough to represent the access controls of a document. For example, in Windows Shares a folder's ACL is interpreted differently by child folders one level down, by child files one level down, by child folders two levels down, and by child files two levels down. In the connector for File Systems, four extra ACLs are sent as fragments of the folder. Child items inherit from these fragments. See the code in this example of a fragrament .




You cannot push fragments using `DocIdPusher`. You must  use Response interface of a connector to provide fragments, which results in named resources that are "rooted" in the returned document:

```
public void putNamedResource(String fragment, Acl acl);
```

First, you need to send the fragment to the search appliance when the content for the main document is being crawled:

```
public void getDocContent(Request req, Response resp) {
       DocId id = req.getDocId();
       ...
```

```
        if("1006".equals(id.getUniqueId())) {
                //prepare the fragment
                ArrayList<Principal> permits = new ArrayList<Principal>();
                permits.add(new GroupPrincipal("group5", "Default"));
                resp.putNamedResource("AclDoc2",
                        new Acl.Builder()
                                .setEverythingCaseInsensitive()
                                .setInheritFrom(new DocId("1005"))
                                .setPermits(permits)
                                .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                                .build());
                //prepare the ACL on the main document, this document has its own
                //ACL and inherit from another document
                ArrayList<Principal> permits2 = new ArrayList<Principal>();
                permits2.add(new GroupPrincipal("group4", "Default"));
                ArrayList<Principal> denies = new ArrayList<Principal>();
                denies.add(new GroupPrincipal("group4", "Default"));
                resp.setAcl(new Acl.Builder()
                        .setEverythingCaseInsensitive()
                        .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                        .setInheritFrom(new DocId("1005"))
                        .setPermits(permits2).setDenies(denies).build());
                //write the main document content
                Writer writer = new OutputStreamWriter(resp.getOutputStream());
                writer.write("Document 1006 says kiwi");
                writer.close();
        }
        ...
}
```

The fragment can inherit from another document or named resource and is  pushed together with the main document. The main document can still have its own ACL and inherit from another document.

You can use the fragment as parent of other documents. Class Acl's Builder has a method `setInheritFrom`:

```
public Builder setInheritFrom(DocId inheritFrom, String fragment)
```

You can use this method to specify the main document and the fragment name from which the other documents inherit from. Here is the sample code:

```
public void getDocContent(Request req, Response resp) {
       DocId id = req.getDocId();
       ...
       if("1007".equals(id.getUniqueId())) {
              ArrayList<Principal> denies = new ArrayList<Principal>();
              denies.add(new GroupPrincipal("group5", "Default"));
              resp.setAcl(new Acl.Builder()
                     .setEverythingCaseInsensitive()
                     .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                     .setInheritFrom(new DocId("1006"), "AclDoc2")
                     .setDenies(denies).build());
              Writer writer = new OutputStreamWriter(resp.getOutputStream());
              writer.write("Document 1007 says coffee");
              writer.close();
       }
}
```

Document 1007 will have this parent in Index Diagnostics:

```
This ACL also inherits the following ACL from https://connector-host:5678/doc/1007?AclId2 w
type PARENT_OVERRIDES
                    Permitted Groups
                    ( default ) . group5 (ECI)
```

### Group Resolution

Connector Framework provides support for Both the SharePoint connector and Active Directory connector use the 7.2 Groups Database  to push groups to the search appliance . If you use this feature, you only need to push group definitions to the search appliance. The search appliance handles the serve time group resolution.

The pushing of group definition should be done in the same methods as pushing of DocIds: `getDocIds` and `getModifiedDocIds`. `DocIdPusher` has this method:

```
public GroupPrincipal pushGroupDefinitions(
      Map<GroupPrincipal, ? extends Collection<Principal>> defs,
      boolean caseSensitive, ExceptionHandler handler)
      throws InterruptedException;
```

The Active Directory Connector pushes group definitions only. It shows how to use `Mutex` to prevent `getDocIds` and `getModifiedDocIds` from stepping on each other's toes. It also uses `response.respondNotFound()` when `getDocContent()` is called.

The SharePoint Connector pushes both group definitions and document IDs in `getDocIds` and `getModifiedDocIds`.

## Authentication by Connector

A connector can be built to provide authentication (including group resolution) and authorization (late binding, non-ACL based). Connector Framework has embedded support for SAML 2.0 protocol for integration with the search appliance. At the moment, only POST Binding is supported. All the SAML plumbing has been provided, and you only need to implement several predefined interfaces. In this chapter we will only discuss authentication. Authorization will be discussed in next chapter.

### Configuration

There are several places that you need to configure in order to make the connector work as a SAML IdP:

- Connector Configuration (depends on version of the search appliance)

In adaptor-config.properties of your connector, add the following entry to identify the appliance as the service provider:

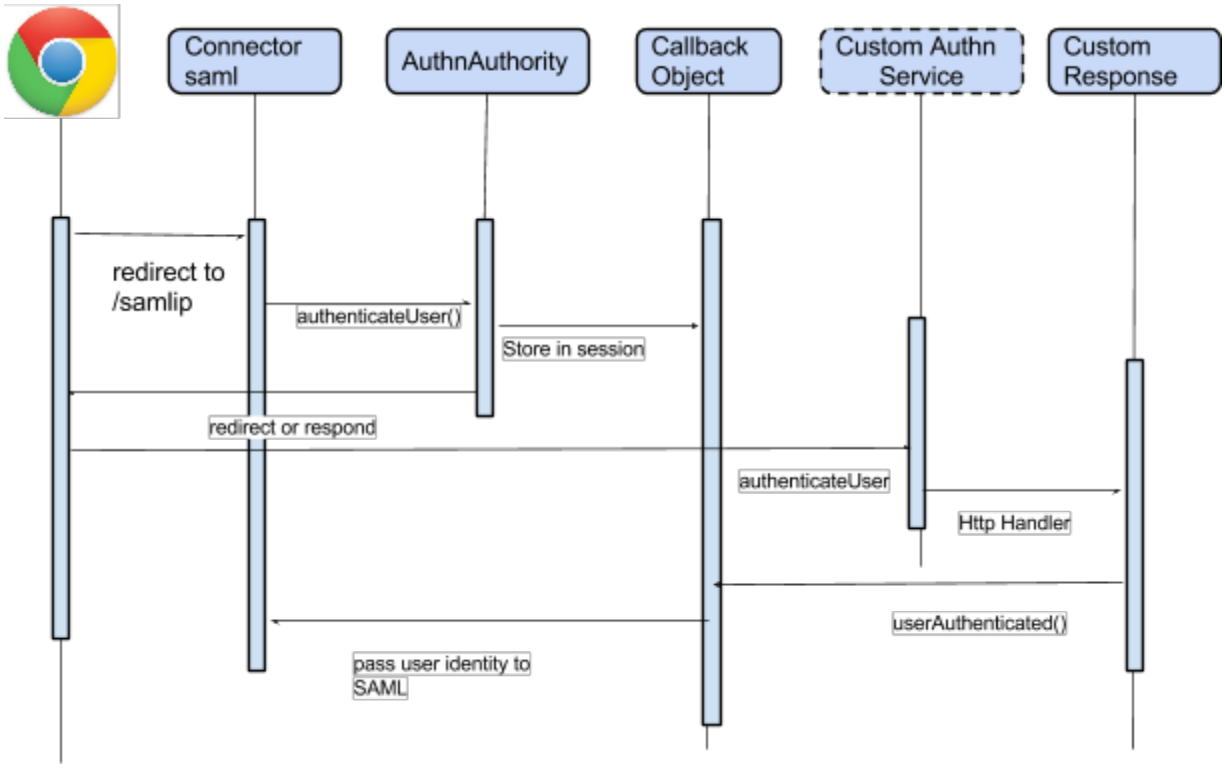
  - gsa.samlEntityId=http://google.com/enterprise/gsa/<APPLIANCE_ID>/security-manager
  - server.samlEntityId=<Any string that uniquely identifies the connector instance>
  - server.keyAlias=<The alias of the key used by the connector in the Keystore with alias>

  The exact format of the EntityId varies by search appliance version. You can check the Security Manager log to see exactly what it is. The above format is what version 7.2 uses.

- ULF Configuration for Authentication
  - IDP Entity ID: It must be the same as configured in server.samlEntityId of the connector configuration file.
  - Login URL: https://connector-host-name:port/samlip
  - Public Key: The public key of the key pair used by the connector and identified by the entry "server.keyAlias" in adaptor-config.properties.

## Workflow

The diagram below shows the sequence during connector authentication.




- Connector as a SAML IdP has an endpoint “/samlip”. When user performs a secure search, the browser is redirected to this endpoint.
- After SAML processing, the connector invokes your implementation of AuthnAuthority interface method authenticateUser().
- The connector can perform the authentication by itself, or have another service do it:
    - If the connector performs authentication, AuthnAuthority.authenticateUser() will respond to user with challenge. Depending on the authentication protocol being used, it might have to handle one or more user responses before identifying a user.
    - If the connector let other security service perform the authentication, AuthnAuthority.authenticateUser() should redirect the browser to that service with a return URL so that when the authentication is completed, the other service will know where to return the user so that the connector can continue with the SAML protocol.

- In either case, you need to create an HTTP handler in your connector to process the response from the user, and to pass the user's identity to the SAML code so that it can return the SAML assertion to the search appliance.

To make this happen, you need to implement at least three classes:

- One class to implement AuthnAuthority
- One or more class to implement HttpHandler. These HttpHandlers will process the user responses or redirects from external authentication service
- One class to implement AuthnIdentity, which is  a data holder to pass user credential to connector's SAML implementation to generate assertion.

### Implementation

Now let's take a look at some sample code. In the connector's init() method, you need to register the AuthnAuthority and HTTP handler:

```
public void init(AdaptorContext context) throws Exception {
    ...
    HelloWorldAuthenticator authenticator = new HelloWorldAuthenticator(context);
    context.setAuthnAuthority(authenticator);
    context.createHttpContext("/google-response", new ResponseHandler(context));
    ...
}
```

You need to implement the authenticateUser() method of AuthnAuthority. The following code prints a login form:

```
public class HelloWorldAuthenticator implements AuthnAuthority{

    ...
       public void authenticateUser(HttpExchange exchange, Callback callback)
                    throws IOException {

              context.getUserSession(exchange, true).setAttribute("callback", callback)

              Headers responseHeaders = exchange.getResponseHeaders();
              responseHeaders.set("Content-Type", "text/html");
              exchange.sendResponseHeaders(200, 0);
              OutputStream os = exchange.getResponseBody();
              String str = "<html><body><form action=\"/google-response\" method=Get>" +
       "<input type=text name=userid/><input type=password" +
" name=password/><input type=submit value=submit>" +
"</form></body></html>";
```

```
            os.write(str.getBytes());
            os.flush();
            os.close();
            exchange.close();
    ...
}
```

The above code does mainly two things:

1. Store the `callback` object has been created by upstream code and passed to this method. You need to store this object as a session object associated with the user. Connector framework doesn't provide a full fledged servlet container that handles all the session handling and request dispatching. However, it does provide some classses and interfaces to you help you. This code uses `AdaptorContext.getUserSession()` method that you can use to store the callback object in the session.
2. Prints an HTML login form. This is just to illustrate the point - it can redirect to an external authentication service in a real project.

When the search user types in user name, password and submits the form, the request needs to be intercepted by your code. You need to create an HttpHandler. The handler processes the final response from user, and retrieves the session object.

```
public  class ResponseHandler implements HttpHandler {
            Session session = context.getUserSession(ex, false);
            Callback callback = (Callback) session.getAttribute("callback");
            if (callback == null) {
                  log.warning("Something is wrong, callback object is missing");
                  return;
            }
            Map parameters = extractQueryParams(ex.getRequestURI().toString());
            if (parameters.size() == 0 || null == parameters.get("userid")) {
                  log.warning("missing userid");
                  callback.userAuthenticated(ex, null);
                  return;
            }
            String userid = (String) parameters.get("userid");
            MyAuthnIdentity identity = new MyAuthnIdentity();
//you can also set Groups on identity
            identity.setUser(new UserPrincipal(userid));
            callback.userAuthenticated(ex, identity);
}
```

The above code does three things:

3. Retrieve the previously stored `callback` object.
1. Retrieve the userid from the query parameters. Again, this is just to illustrate the point - it can be a userid from another service. It also skips the user authentication process and goes directly to step three.
2. Constructs a `AuthnIdentity` object, and pass on to `callback.userAuthenticated()`. You can set both userid and groups on the identity object.

Note that there is a method `extractQueryParams` to retrieve the userid:

```
public Map extractQueryParams(String request) {
	String query = request.substring(request.lastIndexOf("?") + 1);
	String params[] = query.split("&");
	Map paramMap = new HashMap();
	try {
		for (int i = 0; i < params.length; ++i) {
			String param[] = params[i].split("%2F=");
			paramMap.put(URLDecoder.decode(param[0], "UTF-8"),
					URLDecoder.decode(param[1], "UTF-8"));
		}
	} catch (Exception e) {
		log.warning(e.getMessage());
	}
	return paramMap;
}
```

You also need to implement your own version of AuthnIdentity - unless com.google.enterprise.adaptor.AuthnIdentityImpl class is made public:

```
import java.util.Set;

import com.google.enterprise.adaptor.AuthnIdentity;
import com.google.enterprise.adaptor.GroupPrincipal;
import com.google.enterprise.adaptor.UserPrincipal;


public class MyAuthnIdentity implements AuthnIdentity {

	UserPrincipal user;
	Set<GroupPrincipal> groups;

	public void setGroups(Set<GroupPrincipal> groups) {
		this.groups = groups;
```

```
	}

	public void setUser(UserPrincipal user) {
		this.user = user;
	}

	@Override
	public UserPrincipal getUser() {
		return user;
	}

	@Override
	public String getPassword() {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public Set<GroupPrincipal> getGroups() {
		return groups;
	}
}
```

The password is optional - depending on whether the password explicitly collected during authentication.

## Authorization by Connector

Connectors can send ACLs with documents to GSA - that's early binding for document level authorization. There are times when ACLs cannot be easily constructed. Late binding via connector has to be used in that case.

The URLs indexed via connector has the following pattern: https://connector-host-name:port/doc. If no ACLs are defined, the search appliance will use Head Request based on the default configuration of Flex Authz table. The search appliance will use the current search user's credentials (user name and password if available, cookies, Kerberos ticket etc). When requested that way, the connector will not be able to respond properly. The proper way to use connector for authorization is through the SAML interface, which also improves performance by bundling multiple URLs in a single request.

## Configuration

In Search -> Flexible Authorization, add another SAML rule:

- URL Pattern: https://connector-host-name:port/doc
- Authorization service ID: entity id of the connector from server.samlEntityId
- Authorization service URL: https://connector-host-name:port/saml-authz
- Check “Use batched SAML Authorization Requests”

After adding this rule, you need to move it above “Head Request” rule.

## Implementation

First of you, the documents should be indexed as secure but without ACL. You can use Response.setSecure() to achieve the result.

```
public void getDocContent(Request req, Response resp) throws IOException {
...
resp.setSecure(true);
Writer writer = new OutputStreamWriter(resp.getOutputStream());
writer.write("Document 1009 says espresso");
writer.close();
...
}
```

Then you need to define a class that implements the interface **AuthzAuthority** to handle connector authorization:

```
public interface AuthzAuthority {
 public Map<DocId, AuthzStatus> isUserAuthorized(AuthnIdentity userIdentity,
      Collection<DocId> ids) throws IOException;
}
```

You can implement both AuthnAuthority and AuthzAuthority interfaces in the same class:

```
public class HelloWorldAuthenticator implements AuthnAuthority, AuthzAuthority {
...
}
```

You need to register the AuthzAuthority with connector:

```
public void init(AdaptorContext context) throws Exception {
    ...
    HelloWorldAuthenticator authenticator = new HelloWorldAuthenticator(context);
    context.setAuthzAuthority(authenticator);
    ...
}
```

Here is an example of how to implement the only method of interface **AuthzAuthority**:

```
public class HelloWorldAuthenticator implements AuthnAuthority, AuthzAuthority {
...
       public Map<DocId, AuthzStatus> isUserAuthorized(AuthnIdentity userIdentity,
                     Collection<DocId> ids) throws IOException {
              HashMap authorizedDocs = new HashMap();
              for (Iterator iterator = ids.iterator(); iterator.hasNext();) {
                     DocId docId = (DocId) iterator.next();
                     // if authorized
                     authorizedDocs.put(docId, AuthzStatus.PERMIT);
              }
              return authorizedDocs;
       }
...
}
```

## Customization

Through interfaces and configurations, the connector framework allows you to customize the behavior of an existing connector without changing the source code of the connector. Currently, only metadata and ACL can be modified. the connector framework uses a “pipeline” approach where you can cascade multiple transformations, where the output from one transformation will be the input of another transformation. Document content transformation is not supported.

### Metadata transform

Transformers need to be configured in adaptor-config.properties.

- transform.pipeline=step1, step2...stepX

- transform.pipeline.step1.factoryMethod
- transform.pipeline.step1.<arg1>=
- transform.pipeline.step1.<arg2>=
- transform.pipeline.step1.factoryMethod
- transform.pipeline.stepX.<argX>=
- transform.pipeline.stepX.<argX>=
- transform.pipeline.stepX.factoryMethod

**transform.pipeline** is required, so is **transform.pipeline.<stepX>.factoryMethod.** There can be one to multiple “steps”, and you can use different values instead of “stepx”. The arguments and the number of them are optional. It depends on your transformer’s needs and implementation.

The **factoryMethod** is a method defined by your connector that is responsible for creating the transformer instance and set the configuration.

```
public static transformer_class create(Map<String, String> cfg) {
    return new transformer_class(...);
  }
```

A few requirements of the factory method:

- It has to be static
- It has to return an instance of the transformer
- The method argument is fixed - a Map of configuration name/value pair passed in by connector framework’s transform pipeline loader.

You decide what you need to get from the configuration map: the name(s) of the metadata, the values of the metadata. For example, the class in connector framework example “com.google.enterprise.adaptor.examples.MetadataTransformExample” expects the configuration has two entries source metadata name (src) and destination metadata name (dest). This transformer replaces the metadata values identified by “src” with metadata values identified by “dest”.

```
In adaptor_config.properties:
transform.pipeline=replace
transform.pipeline.replace.factoryMethod=com.google.enterprise.adaptor.examples.Metadat
TransformExample.create
transform.pipeline.replace.src=taste
transform.pipeline.replace.dest=flavor


public static MetadataTransformExample create(Map<String, String> cfg) {
```

```
    return new MetadataTransformExample(cfg.get("src"), cfg.get("dest"));
  }
```

In the HelloWorldConnector, there is another metadata transformer.

```
In adaptor_config.properties:
transform.pipeline=step1
transform.pipeline.step1.taste=mango,peach
transform.pipeline.step1.factoryMethod=MetadataAddition.load

public class MetadataAddition implements DocumentTransform {
...
       public static MetadataAddition load(Map<String, String> cfg) {
              return new MetadataAddition(cfg.get("taste"));
       }
...
}
```

In this example, the name of the metadata is hard coded in the code (“taste”), and the values are loaded from the config file.

Metadata transformer must implement the following interface.

```
public interface DocumentTransform {
   public void transform(Metadata metadata, Map<String, String> params);
}
```

The transformer is called for every document. The first parameter **Metadata** holds all the metadata associated with this document. Changes are to be made on this object. The second parameter **params** is a map of strings that holds two entries: “DocId” and “Content-Type”. The DocId can be used to identify documents and as part of the logic to process the metadata. If the “Content-Type” is not available from document, you can explicitly set it here.

### ACL transform

Unlike Metadata transform pipeline where you can create your own transformer, ACL transformation is implemented as rule based configuration. You cannot inject your own transformation without modifying connector framework code. The rules should follow this format in adaptor-config.properties:

```
transform.acl.<rule sequence>=<ACL entry in document to be matched>;<ACL to be converte
to>
```

Here are some examples:

- transform.acl.0=type=user, domain=gsatestlab; domain=gsaprodlab

for user principals with domain “gsatestlab”, change the domain name to “gsaprodlab”

- transform.acl.1=type=group, domain=gsatestlab; domain=gsatestlab.com

for group principal with domain “gsatestlab”, change the domain name to “gsatestlab.com”

- transform.acl.2=type=user, name=user1; domain=gsatestlab, name=user2

for user principal “user1”, set the domain to “gsatestlab”, and rename the user name to “user2”

## Appendix: Building Connector Framework

The following steps are based on Eclipse IDE to build the Connector Framework.

Open an new workspace,

1. From menu File -> Import.
    1. If You don't see "Git" listed, you need to get it.
    2. From menu Help -> Install New Software, if no "git" listed, click "Available software sites".
    3. Pick EGit P2 Repository - http://download.eclipse.org/egit/updates
    4. Select "Eclipse Git Team Provider"
    5. Eclipse will be updated and restarted
2. Select "Projects from Git", Next
3. Select "Clone URI", Next
4. Type https://code.google.com/p/plexi/ to get connector framework project, Next
5. To create a new Java project using wizard
6. Select "using an existing built.xml to create project", and select Plexi's project.
7. You will see build error, Javax.crypto.SecretKey is not found. You need to copy the jce.jar from Apache Tomcat. You also need to set the right jars in build path.