Google Search Appliance Connectors V4

Developer's Guide

Google Search Appliance Connectors software version 4
Google Search Appliance software version 7.2

4 December 2014

Table of Contents

Introduction

A Google Search Appliance Connector is a custom program that sits between the Google Search Appliance (GSA) and your non-HTTP content repository. The connector provides all of the information that the GSA requires to index your content, including security and metadata.

Purpose

This document guides developers through the creation of a simple connector, and demonstrates the many useful features that developers can use to customize the indexed content.

Audience

This document is intended for software developers who are familiar with the GSA and need to index content sources not easily crawled by the GSA (such as portal content or non-HTTP content). This document may also serve as a resource for architects and systems analysts who require an architectural overview of the communication and security model between the GSA and GSA connector.

Assumptions

This document makes the following assumptions:

- The reader is a developer who has a basic understanding of the GSA, and is familiar with compiling and running code.
- Although you may communicate with a connector using any language, the connector software is written in Java®
- The developer must be able to sign in to the GSA to configure and validate the connector.
- From a network perspective, the GSA must be able to communicate with the computer where the connector code is running.

Conventions

Type	**Style**	**Example**
File names, configuration variables, domain names, Class and Method names	Courier New font	`HelloWorldConnector`
Literal strings and commands in the GSA Admin Console	Bold	Click **Administration** > **SSL Settings**.
URLs	Normal text	http://googlegsa.github.io/adaptor/index.html

GSA example use cases

Altostrat College is using GSA to index and serve information from departmental webpages, meeting minutes, memos, and so on. They also have a legacy, in-house academic personnel information system that includes information about hires, terminations, promotions, sabbaticals, and so on. The college administration would like to have the records from the academic personnel information system indexed and served by the GSA, along with the other content. To accomplish this goal, they can develop a custom connector. Using the connector, the GSA can crawl, index, and serve the records in the index.

Another use case involves an organization that wants the GSA to index content from an SAP® database, for which there is no connector. By developing and implementing a custom connector for the SAP database, the organization enables the GSA to crawl, index, and serve SAP database records to search users.

Updates to the GSA Connector V4

The GSA Connector V4 improves the developer experience, making it simple to create a custom connector while still providing security, performance, and scalability.

Google has developed several connectors to connect the GSA to common non-HTTP sources, such as Microsoft® SharePoint, Microsoft® SharePoint User Profiles, Microsoft Windows® Shares, and Microsoft® Active Directory. You can download these connectors with their deployment guides.

One of the new features of the GSA Connector V4 is the ability to communicate with a connector using any programming language and a command line adaptor connector. You can view a full description in Non-Java connectors.

Get started

This section guides you through the process of running a sample connector. Running a sample connector demonstrates communication from the GSA to the connector and ensures that your development environment is properly setup.

1. Set up your development environment
2. Test your setup

Set up your development environment

Before you can run a sample connector, you must first download the Java library, and add it to your development environment.

1. Go to the Google Search Appliance Adaptors resource page.
2. In the "Instructions for developing your own Adaptor" section, click the **Download the library** link.
3. After you've downloaded the library, unzip the file.
4. Create a new Java project.
5. Add the downloaded connector libraries to your Java project.

Test your setup

With your development environment set up with the connector libraries, you can test a sample connector and validate that the GSA has indexed sample content.

These steps guide you through the processes of testing a sample connector in your development environment:

1. Set up the GSA
2. Add a sample connector
3. Run the connector
4. Validate indexed content in GSA
5. Review the Index Diagnostics
6. Validate the GSA search results

Set up the GSA

Set up the GSA to allow the hostname/IP address to be crawled.

1. In the GSA Admin Console, go to **Content Sources** > **Web Crawl** > **Start and Block URLs.**
2. In the **Follow Patterns** section, click **Add**.
3. Enter the URL where the connector will reside.
   For example, you might enter **http://connector.example.com:5678/doc/** where **connector.example.com** is the hostname of the machine that hosts the connector. By default, the connector runs on port 5678.
4. Click **Save**.

If the GSA is not set up to trust feeds from all IP addresses, then add your IP address as follows:

1. In the GSA Admin Console, go to **Content Sources** > **Feeds**.
2. In the **List of Trusted IP Addresses** section, if the **Only trust feeds from these IP addresses** option is selected, add the IP address for the connector to the list.
3. Click **Save**.

Add a sample connector

1. Download the sample connector.
2. Add this connector class to your project.

3. Create a properties file at the root of your project named `adaptor-config.properties`.
4. Add a single entry within this file that defines the hostname or IP address of your GSA: `gsa.hostname=`*`(add your GSA hostname or IP address)`*

Run the connector

Execute the `main()` method of the `AdaptorTemplate` class. For example:

```
java -cp adaptor-4.0.3-withlib.jar;examples/adaptor-4.0.3-examples.jar
com.google.enterprise.adaptor.examples.AdaptorTemplate
```

Validate indexed content in GSA

1. In the GSA Admin Console, go to **Content Sources** > **Feeds**.
2. In the **Current Feeds** section, you should see a feed that can be identified by **Source Name**.
3. Verify that the status is **In Progress** or **Completed**.
4. When feed status is **Completed**, review the Index Diagnostics.

Review the Index Diagnostics

1. In the GSA Admin Console, go to **Index** > **Diagnostics** > **Index Diagnostics**. There may be a delay of up to ten minutes before content is reflected in Index Diagnostics.
2. Verify that the Connector feed URL has a value greater than 0 for **Crawled URLs**. This indicates that the feed was successfully crawled.
3. Click the **Crawled URLs** number link, then the **doc** link to review the crawled documents. Verify that the two sample documents 1001 and 1002 are listed.

Validate the GSA search results

1. Go to the GSA search page.
2. Enter the following search terms: **apple orange**.
3. Validate that there's at least one search result.
4. Click a search result to view the file in your browser.

GSA connector architecture

There are 4 key aspects of the GSA connector architecture:

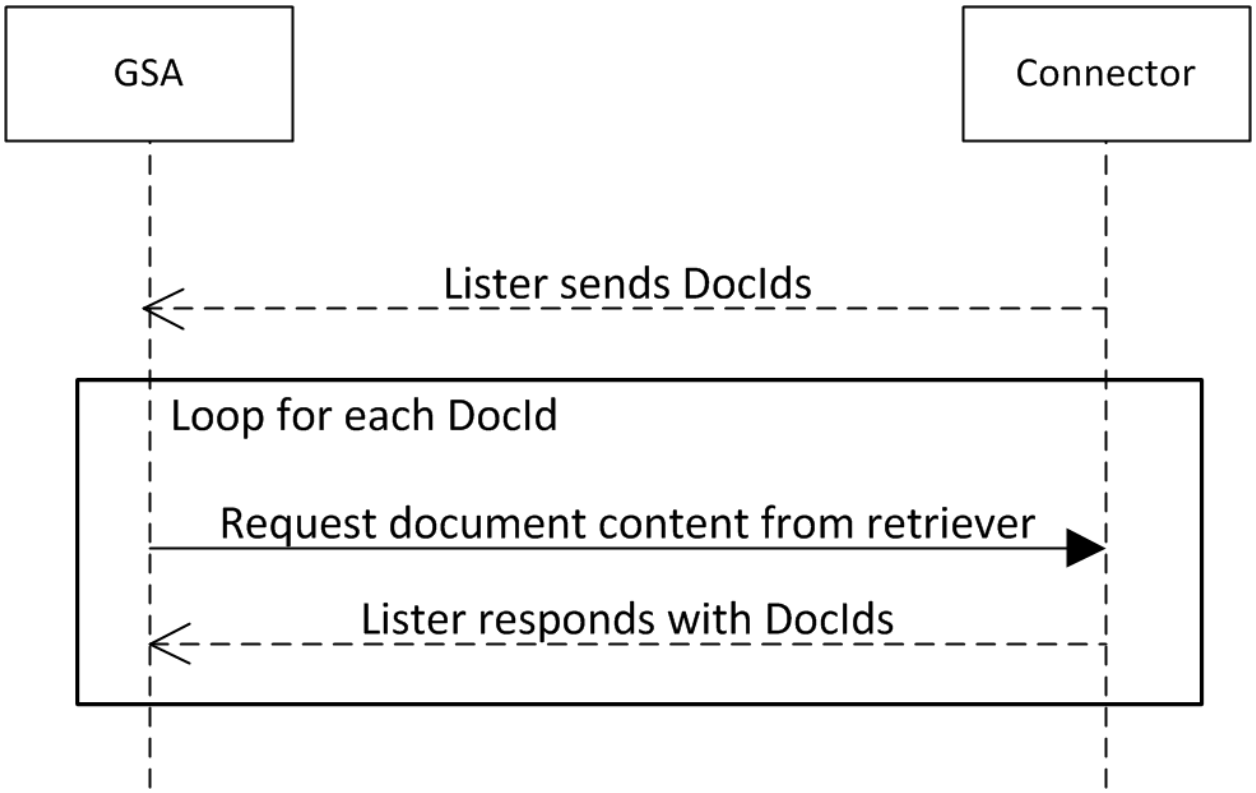
- Communication flow
- Security
- The Lister and Retriever model
- Secure search

Communication flow

GSA connectors enable the GSA to index content within a content repository. The connector provides all of the information required by the GSA to index the content and, optionally, provides metadata and security definitions.

The connector runs as a separate application and provides communication between the GSA and the content repository. After the connector has been registered with the GSA, a simple communication flow is as follows:

- The connector provides a list of Doc IDs for the GSA.
- The GSA requests the document contents for each Doc ID.
    - The connector provides the document contents and, optionally, provides metadata and security definitions.



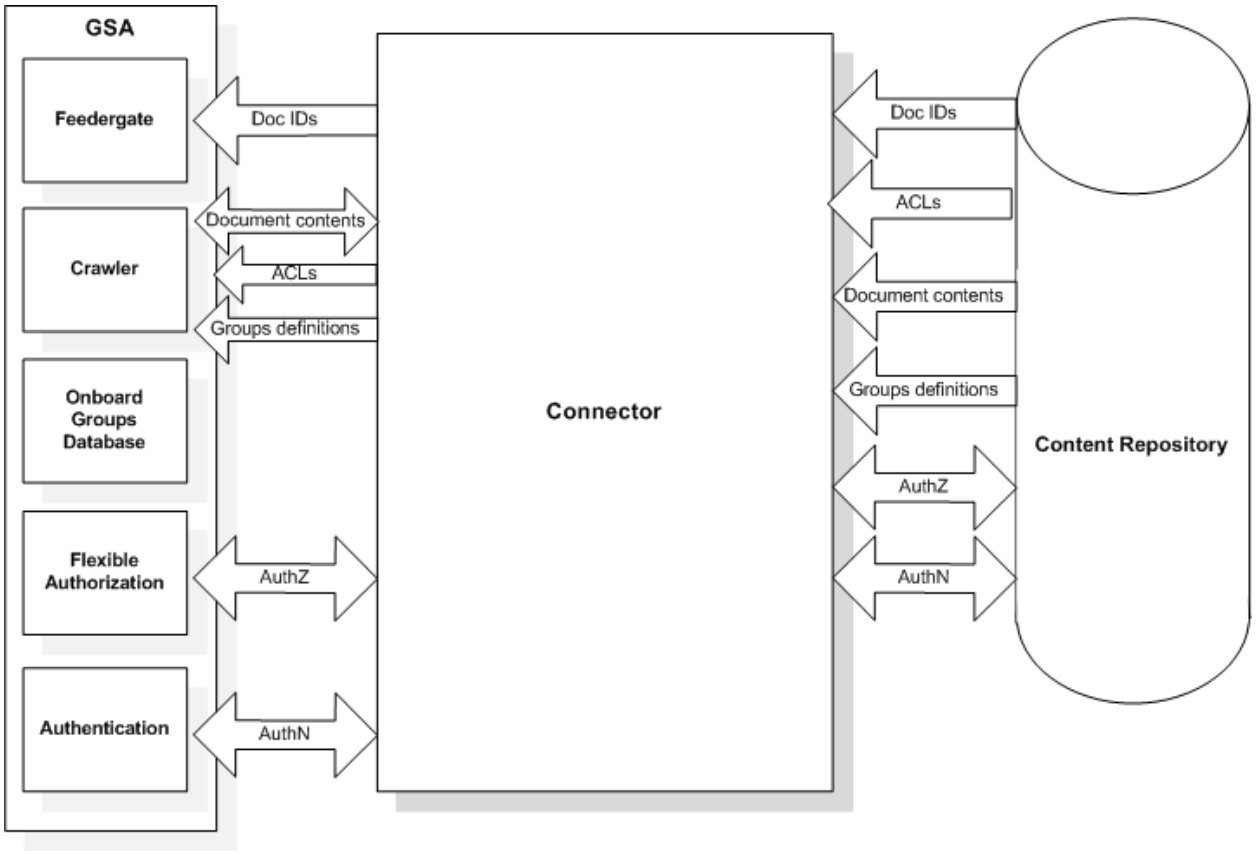
Another approach to provide the unique IDs to the GSA in smaller batches is called graph traversal. This technique provides a virtual webpage containing content links to the GSA to crawl and index.

Security

Connectors can send access control lists (ACLs) with documents to the GSA for early-binding, document-level authorization.

Late-binding authorization is also possible through a Security Assertion Markup Language (SAML) interface. In this case, the connector facilitates authorization and authentication using SAML as the communication protocol. Authentication can be performed within the connector, or authentication can be handed off to another service.

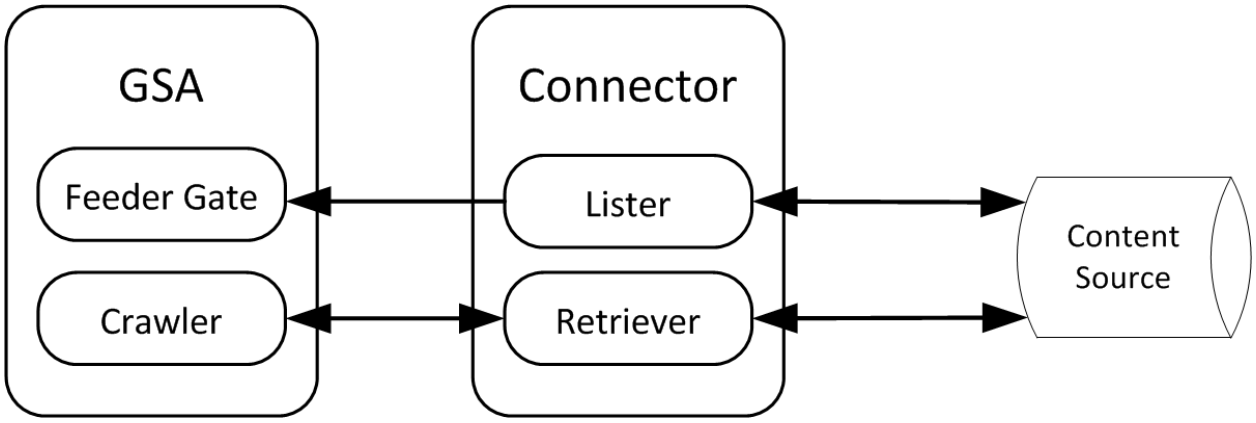
The following diagram shows the communication between the GSA, the connector, and the content repository:



The Lister and Retriever model

In order for the GSA to crawl the content within the repository, the connector must provide a Lister and a Retriever. The Lister provides a list of unique DocIds that the GSA uses to request the content to be indexed from the Retriever.

The following diagram demonstrates how the GSA communicates with the connector:



Secure search

There are 3 concepts related to securing your search:

- Early-binding authorization with ACLs
- Late-binding authorization
- Authentication

Early-binding authorization with ACLs

ACLs define document access for users and groups. You can program the connector to build ACLs and pass them to the GSA within the Retriever. If this configuration isn't appropriate for your environment, you can implement Late-binding authorization.

Late-binding authorization

The connector framework can handle both the authentication of users and authorization of GSA search results. Connectors have embedded support for the SAML 2.0 protocol that integrates with the GSA.

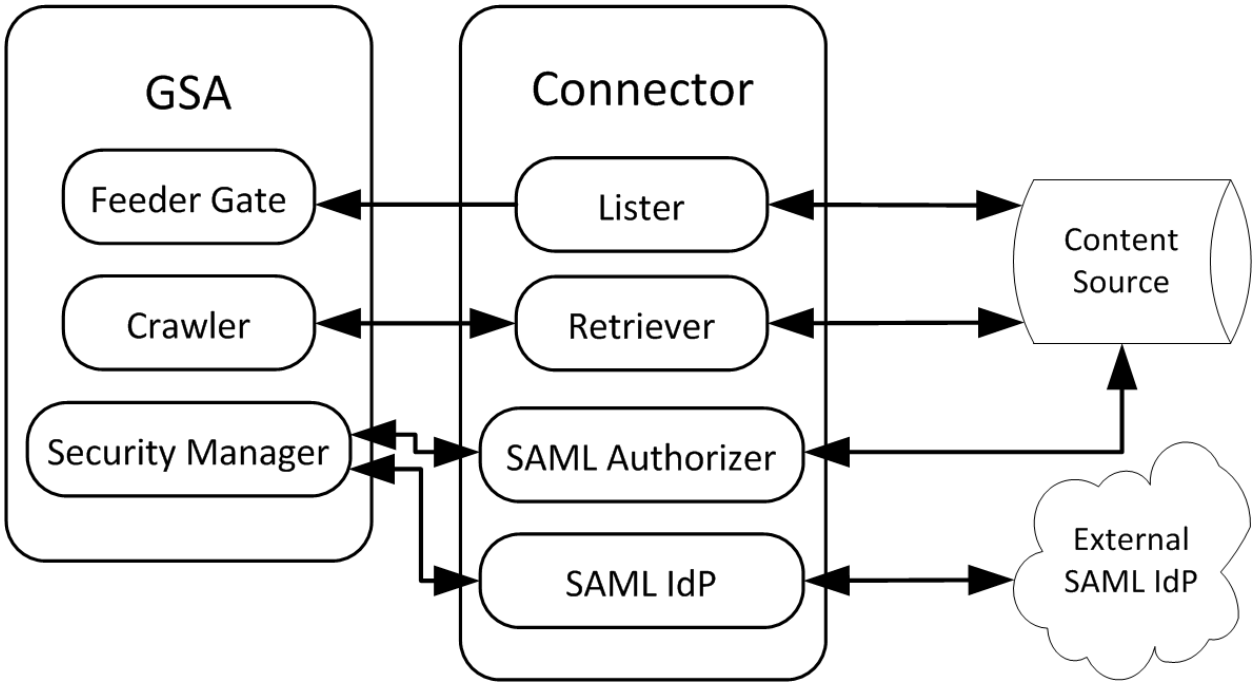
A connector can implement the following optional components:

- A SAML Identity Provider (IdP) endpoint (authentication)
- A document authorizer (authorization)

Authentication

The authentication process verifies the identity of the user so that authorization can be performed. A SAML IdP endpoint can be:

- Implemented within the connector
- Configured to use an external SAML IdP



Connector development

Developing a connector involves one or more of the following:

- Create a Lister and Retriever
- Configure index update behavior
- Customize the Connector
- Secure Your Content

Create a Lister and Retriever

To create a Lister and Retriever, your custom connector needs to implement the `Adaptor` interface. The `Adaptor` interface has the following methods:

Method	**Description**
`initConfig(Config config)`	Provides the opportunity for the connector to specify keys for its own configuration.
`init(AdaptorContext context)`	Provides available context -- including read in configuration -- to the connector.
`getDocIds(DocIdPusher pusher)`	Pushes all the DocIds to be indexed by the GSA.
`getDocContent(Request request, Response response)`	Provides contents, metadata and ACLs of a particular document.

`destroy()`	Shuts down and releases connector resources.

One common option is to extend the `AbstractAdaptor` class; it provides default implementations for common GSA Connector V4 methods.

```
public class HelloWorldConnector extends AbstractAdaptor
```

The `AbstractAdaptor` class provides a default implementation for most of the connector methods.
The connector must override the methods that serve as the Lister and Retriever. For a complete class description, the Javadocs can be found here.

At minimum, a connector is comprised of:

- The Lister
- The Retriever
- Connector configuration

The Lister

The `getDocIds()` method serves as the Lister. The implementation of the Lister would require you to obtain a list of unique DocIds that you want indexed by the GSA. The following code demonstrates returning some mock DocIds that will be fed to the GSA in preparation for indexing the content:

```
/** Gives list of document ids that you'd like on the GSA. */
@Override
public void getDocIds(DocIdPusher pusher) throws InterruptedException {
        ArrayList<DocId> mockDocIds = new ArrayList<DocId>();
        /* Replace this mock data with code that lists your repository. */
        mockDocIds.add(new DocId("1001"));
        mockDocIds.add(new DocId("1002"));
        pusher.pushDocIds(mockDocIds);
}
```

The Retriever

The `getDocContent()` method serves as the Retriever. The following code shows a skeleton Retriever method:

```
public void getDocContent(Request req, Response resp) throws IOException {
        // TODO Auto-generated method stub
}
```

The implementation of the Retriever contains the content lookup by DocId and returns this content back to the GSA for indexing. The following code demonstrates returning some mock content that is indexed by the GSA:

```
/** Gives the bytes of a document referenced with id. */
@Override
public void getDocContent(Request req, Response resp) throws IOException {
        DocId id = req.getDocId();
        String str;
        if ("1001".equals(id.getUniqueId())) {
                str = "Document 1001 says hello and apple orange";
        } else if ("1002".equals(id.getUniqueId())) {
                str = "Document 1002 says hello and banana strawberry";
        } else {
                resp.respondNotFound();
                return;
        }
        resp.setContentType("text/plain; charset=utf-8");
        OutputStream os = resp.getOutputStream();
        os.write(str.getBytes(encoding));
}
```

Connector configuration

A connector must have a configuration file that, at a minimum, defines the location of the GSA.

The file must be at the root of the connector project. By default, the expected configuration file name is `adaptor-config.properties`. If you would like to use a different configuration file name, it can be specified when executing your connector application by specifying an `adaptor.configfile` flag.

To define the location of the GSA, add the following entry within this file:
`gsa.hostname=`*`(add your GSA hostname or IP address)`*

Update the index

When the content in your repository changes, you'll want the GSA index to be updated. Depending on your freshness requirements, you may want to schedule full index or incremental updates, or provide more detailed control on a document-by-document basis.

The following sub-sections describe each of these processes:

- Full listing
- Incremental updates
- Deleted documents
- Runtime configuration

Full listing

The `getDocIds()` Lister provides a full list of DocIds from your content repository to the GSA for indexing. You can configure the connector to perform this full listing on a schedule by adding a configuration setting to the `adaptor-config.properties` file:
`adaptor.fullListingSchedule=`*`(value is in cron format (minute, hour, day of month, month, day of week). Defaults to 0 3 * * *)`

By default, a full listing is performed when the connector starts. To prevent this from occurring, add the following setting to the `adaptor-config.properties` file:
`adaptor.pushDocIdsOnStartup=false`

Incremental updates

As the GSA crawls the content within your repository via your connector, you can develop the Lister to only pass DocIds for the content that has been modified recently.
To do this, follow these steps:

1. Implement the `PollingIncrementalLister` interface.
2. Override the `getModifiedDocIds()` method.
3. Register the incremental lister with the `AdaptorContext` method.
4. Configure the polling period.

As shown below, the connector class implements the `PollingIncrementalLister` interface:

```
public class HelloWorldConnector extends AbstractAdaptor implements
                PollingIncrementalLister
```

The `getModifiedDocIds()` method would be developed to obtain a list of modified DocIds within your repository. The following example shows a single mockDocId identified as a modified document:

```
@Override
public void getModifiedDocIds(DocIdPusher pusher) throws IOException,
                InterruptedException {
        ArrayList<DocId> mockDocIds = new ArrayList<DocId>();
        mockDocIds.add(new DocId("1002"));
        pusher.pushDocIds(mockDocIds);
}
```

The incremental lister must be registered with the `AdaptorContext` method within the `init()` method:

```
@Override
public void init(AdaptorContext context) throws Exception {
        context.setPollingIncrementalLister(this);
}
```

If it's not easy to identify modified documents within your repository, then don't use this interface. Without the `PollingIncrementalLister` interface, the GSA can be left to naturally find changes.

The default polling period is 900 seconds, but you can configure the polling time by adding the following setting to the `adaptor-config.properties` file:
`adaptor.incrementalPollPeriodSecs=`*`(number of seconds between polling)`*

Deleted documents

When the Retriever attempts to load a document that's been deleted from the content repository, the GSA should be informed that the deleted document should be removed from the index. You can do this by using the `respondNotFound()` method of the `Response` object.

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

Runtime configuration

In a basic configuration, the Lister only provides DocIds to the GSA. However, the Lister can provide a more detailed configuration along with each DocId by passing a `DocIdPusher.Record` object to the `pushRecords()` method in the Lister.

The `Record` class uses the builder pattern to set the appropriate attributes. The following code demonstrates the following runtime configuration:

- The `setCrawlImmediately(true)` method indicates that the GSA should give the record priority to re-crawl.
- The `setLastModified()` method specifies the last modified date.

```
DocIdPusher.Record record = new DocIdPusher.Record.Builder(new DocId(
                "1009")).setCrawlImmediately(true).setLastModified(new Date())
                .build();
pusher.pushRecords(Collections.singleton(record));
```

Customize the connector

You can customize the behavior of the connector through the implementation of interfaces and through configuration. Some of the frequently used customizations are described in this guide, but you can explore all customization options by reviewing the following sources:

- Complete listing of available configurations
- Complete Javadoc library

The available connector customizations include :

- Display URL
- Add metadata
- Authorization by ACL
- Crawling behavior
- Lock document index
- Additional response attributes

Display URL

You can set the display URL in the Retriever by using the `setDisplayUrl()` method of the `Response` class.

```
resp.setDisplayUrl(new URI("http://fake.com/a"));
```

This method changes the URL returned in the search results, displaying `http://fake.com/a` instead of the connector-based URL.

Add metadata

You can index additional metadata, along with the content, by using the `addMetadata()` method of the `Response` class.

```
resp.addMetadata("flavor", "vanilla");
resp.addMetadata("flavor", "hazel nuts");
resp.addMetadata("taste", "strawberry");
```

As demonstrated above, the connector can send multiple values for the same key by making multiple calls to the `addMetadata()` method. The metadata is returned in the `X-GSA-External-Metadata` header. Learn how external metadata is sent in an HTTP header.

Crawling behavior

By default, documents are re-crawled periodically. To customize the behavior to only crawl a document once, use the `setCrawlOnce()` method of the `Response` class.

```
resp.setCrawlOnce(true);
```

Lock document index

When the GSA license limit is reached, unlocked documents are deleted before locked documents. To lock a document within the GSA index, use the `setLock()` method of the `Response` class.

```
resp.setLock(true);
```

Additional response attributes

For a complete listing of available `Response` attributes, review the `Response` class Javadoc.

Secure your content

Using the connector framework, you can secure content through an authentication mechanism, and you can identify the documents a user is authorized to view.

Securing your content involves the following topics:

- Authorization by ACL

- Authentication
- Authorization by connector

Authorization by ACL

ACLs define the users and groups that are either permitted or denied access to the document. You can build an ACL that can be sent along with the document content by the Retriever method.

The ACL uses the Builder pattern for creation, the ACL includes the following attributes:

- Permitted and denied groups
- Permitted and denied users
- Inheritance
- Case sensitivity

You can use the `setAcl()` method of the Retriever's `Response` object to set the ACL on a document.

The following code demonstrates sending an ACL with a document from the Retriever:

```
public void getDocContent(Request req, Response resp) throws IOException {
        ArrayList<Principal> permits = new ArrayList<Principal>();
        permits.add(new UserPrincipal("user1", "Default"));
        permits.add(new UserPrincipal("eric", "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()
                        .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                        .setPermits(permits).setDenies(denies).build());

        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1005 says americano");
        writer.close();
}
```

The inheritance type that's defined with the ACL determines which ACL takes precedence when inheriting the ACL from an other document. The following table demonstrates the effect of setting different inheritance types:

Document ACL	Inherited from ACL	Inheritance Type	Resulting ACL
Permit Group A	Deny Group A	`Acl.InheritanceType.PARENT_OVERRIDES`	Deny Group A
Permit Group A	Deny Group A	`Acl.InheritanceType.CHILD_OVERRIDES`	Permit Group A

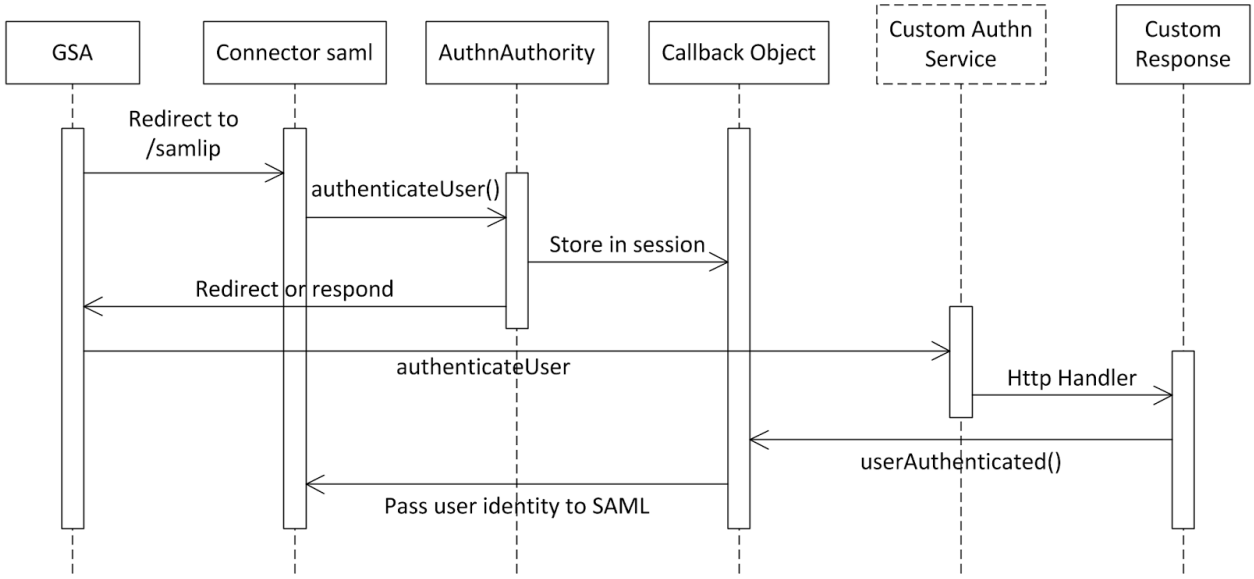
Authentication

The connector framework uses the SAML 2.0 protocol to integrate with the GSA. This allows the connector to authenticate a user and pass identity information to the GSA used by the authorization process described in Authorization.

The authentication flow is as follows:

1. User performs a secure search.
2. The browser is redirected to the SAML IdP (/samlip).
3. The connector's implementation of the `AuthnAuthority` interface `authenticateUser()` method is called.
    - The credentials can be authenticated either by the connector itself, or by a another service. One option for the connector to perform the authenticaton is ri to have the `authenticateUser()` method respond with a login form for the user to enter their credentials.
    - If another service is used, the `authenticateUser()` method redirects the browser to that service. The redirect would include a return URL so that when the authentication is completed, the other service knows where to return the user and the connector can continue with the SAML protocol.
4. An HTTP handler that's defined within the connector processes the response from the user. The handler passes the user's identity to the SAML code so that it can return the SAML assertion to the search appliance.

The following diagram illustrates the authentication flow:



The process of configuring and implementing authentication is as follows:

1. Set up system
2. Configure the connector as a SAML IdP.
3. Implement authentication for the connector.
4. Configure the GSA Universal Login form.

Set up system

To secure the link between the connector and GSA, follow the instructions provided in the "Enable Connector Security" section of the *Google Search Appliance Connectors Administration Guide.* You can perform SAML authentication over an insecure connection between GSA and connector, but that is highly discouraged for production systems.

Configure the connector as a SAML IdP

You can configure the connector as a SAML IdP by adding the following settings to the `adaptor-config.properties` file:

`gsa.samlEntityId`	http://google.com/enterprise/gsa/<APPLIANCE_ID>
`server.samlEntityId`	A string that uniquely identifies the connector instance.
`server.keyAlias`	A keystore alias where encryption keys are stored. Default is “adaptor”.

To obtain the correct Entity ID from the GSA, go to **Search > Secure Search > Access Control**, and copy the value defined under SAML Issuer Entity ID.

Implement authentication within the connector

Implementing authentication for the connector consists of the following guidelines:

- A class that implements the `AuthnAuthority` interface, which is responsible for handling the authentication calls.
- One or more classes that implement the `HttpHandler` interface, which handles the authentication form submission or external service callback.
- Registration of an HTTP handler within the Connector's `init()` method. This handler is used to process the response from the user, or redirect from an external authentication service.
- A class that implements the `AuthnIdentity` interface.This is used to pass user credential to the connector's SAML implementation to generate assertion.

The following example initializes the authentication components within the connector's `init()` method:

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

In this example, the `HelloWorldAuthenticator` class is registered as the `AuthnAuthority` interface and as the HTTP handler. Therefore, the class must implement both `AuthnAuthority` and `HttpHandler` interfaces.

```
class HelloWorldAuthenticator implements AuthnAuthority, HttpHandler {
```

By implementing the `AuthnAuthority` interface, the class must override the `authenticateUser()` method. In this example, the connector is to perform the user authentication.

```
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();
}
```

When the `authenticateUser()` method is called, a `CallBack` object is passed, which is used later to send the identity information back to the GSA. The connector framework provides the ability to store objects in `Session`.

This sample code writes a simple login form for demonstration purposes, but the code could be written to redirect to an external authentication service. After the user submits the completed form, the request is intercepted by the registered `HttpHandler` interface, and the `handle()` method is called.

```
public void handle(HttpExchange ex) throws IOException {
        log.entering("HelloWorldAuthenticator", "handle");

        callback = getCallback(ex);
        if (callback == null) {
                return;
        }

        Map<String, String> parameters = extractQueryParams(ex.getRequestURI()
                .toString());
        if (parameters.size() == 0 || null == parameters.get("userid")) {
                log.warning("missing userid");
                callback.userAuthenticated(ex, null);
                return;
        }
        String userid = parameters.get("userid");
        SimpleAuthnIdentity identity = new SimpleAuthnIdentity(userid);
```

```
        callback.userAuthenticated(ex, identity);
}
```

Utility methods retrieve the `CallBack` object from `Session` and the query parameters from the submitted form. This simple example skips the process of verifying the passed credentials. A `SimpleAuthnIdentity` object is created that consists of the `userid`. You can also add groups to the identity object. This identity is then is passed back through the `callback.userAuthenticated()` method.

You can view the utility methods used in the `HelloWorldAuthenticator` class on github.

The `SimpleAuthnIdentity` class implements the `AuthnIdentity` interface and is used to pass identity information back to SAML.

```
class SimpleAuthnIdentity implements AuthnIdentity {

  private UserPrincipal user;
  private Set<GroupPrincipal> groups;

  public SimpleAuthnIdentity(String uid) throws NullPointerException {
    if (uid == null) {
      throw new NullPointerException("Null user not allowed");
    }
    this.user = new UserPrincipal(uid);
  }

  //Constructor with user & single group
  public SimpleAuthnIdentity(String uid, String gid)
      throws NullPointerException {
    this(uid);
    this.groups = new TreeSet<GroupPrincipal>();
    if (gid != null && !"".equals(gid)) {
      this.groups.addAll(Collections.singleton(new GroupPrincipal(gid)));
    }
    this.groups =
        Collections.unmodifiableSet(this.groups);
  }

  // Constructor with user & groups
  public SimpleAuthnIdentity(String uid, Collection<String> gids)
      throws NullPointerException {
    this(uid);
```

```
    this.groups = new TreeSet<GroupPrincipal>();
    for (String n : gids) {
      if (n != null && !"".equals(n)) {
        this.groups.addAll(Collections.singleton(new GroupPrincipal(n)));
      }
    }
    this.groups =
        Collections.unmodifiableSet(this.groups);
  }

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

  @Override
  public String getPassword() {
    return null;
  }

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

Configure the GSA Universal Login form

Before a user can perform a secure search, they must submit their credentials to view the authorized search results. They submit their credential using the GSA Universal Login form. You must configure the GSA to present the Universal Login form from the connector.

Within the GSA Admin console, go to **Search** > **Secure Search** > **Universal Login Auth Mechanisms** > **SAML**. Complete the form as follows:

- Mechanism Name—Type a unique name for the authentication mechanism.

- IDP Entity ID—Add the SAML Entity IdP ID to match the `server.samlEntityId` entry value in the connector configuration file.

- Login URL—Add the SAML IdP URL for the connector: https://connector-host-name:port/samlip
    - The default connector port is 5678.

- Artifact Resolver URL—This value should be left blank. as the Public Key value is used for signing an assertion.

- Public Key of IDP—Enter the public key that was configured in Set up system. This public key must match that as identified by the value of `server.keyAlias` in the `adaptor-config.properties` connector configuration file.

- Timeout (seconds)—Enter the time allowed for making a network connection. If left blank, the default value is 3 seconds.

Learn more about the Google Search Appliance.

Authorization by connector

Instead of sending the ACLs with the document content by the Retriever, you can implement late-binding authorization of documents by way of the connector.

Implementing late-binding authorization requires you to:

- Add a SAML rule to the GSA.
- Implement authorization within the connector.

Add a SAML rule to the GSA

Within the GSA Admin Console, perform the following steps:

1. Go to to **Search** > **Secure Search > Flexible Authorization.**

2. From the drop-down list, select the **SAML** rule and click **Add another rule**.

3. Complete the form as follows:
    a. URL Pattern—`https://connector-host-name:port/doc`
    b. Authorization service ID—Entity ID of the connector from `server.samlEntityId`
    c. Authorization service URL—https://connector-host-name:port/saml-authz
    d. Check the **Use batched SAML Authorization Requests** box.

4. Once saved, move this **SAML** rule above “HEADREQUEST”.

Implement authorization within the connector

To implement authorization within the connector:

1. Identify the content as being secure content in the Retriever.

2. Create an authorizer class that implements the `AuthzAuthority` interface.
3. Register the authorizer class with the connector.

To identify the content as being secure, add a call to the `setSecure()` method of the `Response` class:

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

Create an authorizer class that implements the `AuthzAuthority` interface and contains an `isUserAuthorized()` method:

```
class HelloWorldAuthorizer implements AuthzAuthority {
        public Map<DocId, AuthzStatus> isUserAuthorized(AuthnIdentity userIdentity,
                        Collection<DocId> ids) throws IOException {

                HashMap<DocId, AuthzStatus> authorizedDocs = new HashMap<DocId,
AuthzStatus>();

                for (Iterator<DocId> iterator = ids.iterator(); iterator.hasNext();) {
                        DocId docId = iterator.next();
                        // if authorized
                        authorizedDocs.put(docId, AuthzStatus.PERMIT);
                }
                return authorizedDocs;
        }
}
```

For brevity, this simple sample excludes the logic that would be performed to determine if the user is authorized to access the documents. For each document that the user is permitted to access, the `docId` is added to a `Map` along with the `AuthzStatus.PERMIT` value.

This authenticator class is then registered with the connector:

```
public void init(AdaptorContext context) throws Exception {
        ...
```

```
        HelloWorldAuthorizer authorizer = new HelloWorldAutherizer(
                        context);
        context.setAuthzAuthority(autherizer);
        ...
}
```

Inherit security definitions

There are 3 mechanisms that allow security definitions to be inherited:

- Document ACL inheritance
- Named resource ACL inheritance
- Fragment ACL inheritance

Document ACL inheritance

When defining the ACL for a document in the Retriever, you may inherit an ACL from another document. When you build an ACL by using the `Acl.Builder()`, you specify a DocId with `setInheritFrom()` and the type of inheritance with `setInheritanceType()`.

The following example demonstrates inheriting an ACL from another document:

```
public void getDocContent(Request req, Response resp) throws IOException {
        ...
        // Inherit ACLs from 1005
        ArrayList<Principal> permits = new ArrayList<Principal>();
        permits.add(new GroupPrincipal("group3", "Default"));
        ArrayList<Principal> denies = new ArrayList<Principal>();
        denies.add(new GroupPrincipal("group3", "Default"));

        resp.setAcl(new Acl.Builder().setEverythingCaseInsensitive()
                        .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                        .setInheritFrom(new DocId("1005")).setPermits(permits)
                        .setDenies(denies).build());

        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1006 says misto");
        writer.close();
        ...
}
```

Named resource ACL inheritance

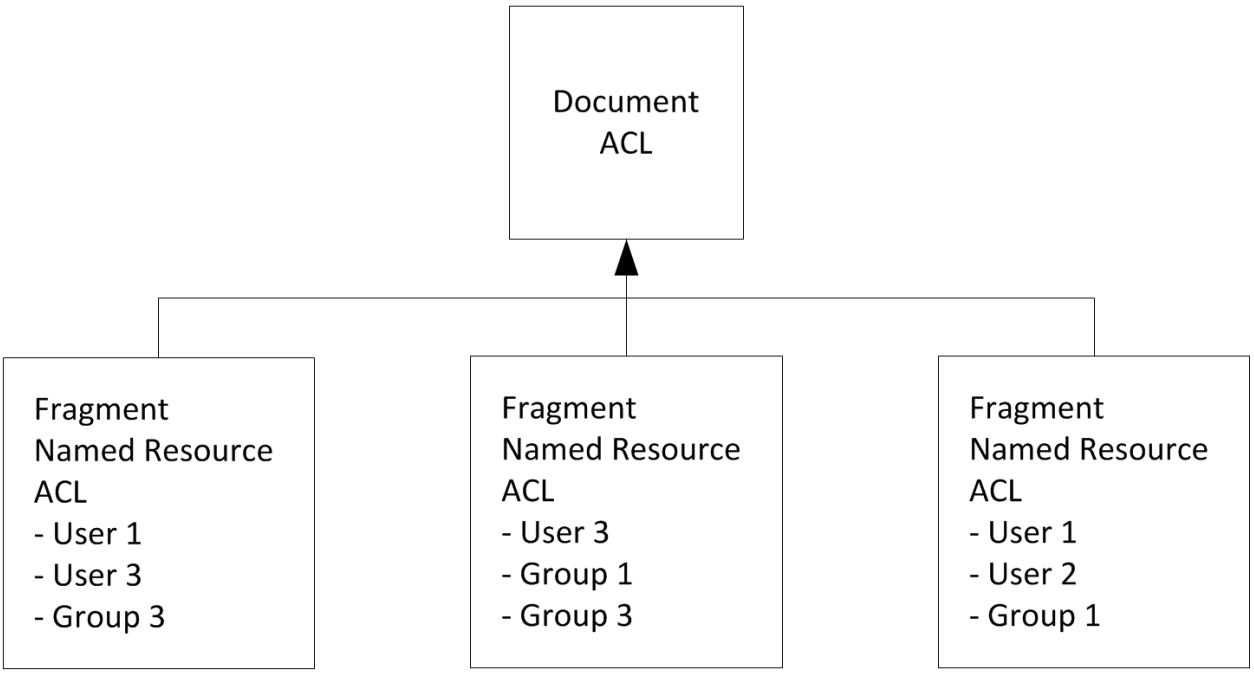
In some content repositories, the ACL information is defined separately from the content. The GSA supports this case through named resources. A named resource is an ACL that's not associated with a document. Named resources can be pushed to the GSA using the `pushNamedResources()` method of the `DocIdPusher`. Documents may inherit these named resource ACLs; see Document ACL inheritance.

The following example demonstrates the creation of a named resource ACL:

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

Advanced Access Control : Fragment ACL

For situations which require multiple ACLs for every folder or document there is a facility which allows ACLs to be rooted at particular folder or document.



The following example demonstrates a Retriever returning a document that includes a fragment ACL definition along with the document ACL:

```
public void getDocContent(Request req, Response resp) throws IOException {
        ...
        // Inherit ACLs from 1005 & 1006
        ArrayList<Principal> permits = new ArrayList<Principal>();
        permits.add(new GroupPrincipal("group5", "Default"));

        resp.putNamedResource(
                        "Whatever",
                        new Acl.Builder()
                                        .setEverythingCaseInsensitive()
                                        .setInheritFrom(new DocId("1006"))
                                        .setPermits(permits)
                                        .setInheritanceType(

Acl.InheritanceType.PARENT_OVERRIDES)
                                        .build());

        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());

        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1007 says frappuccino");
        writer.close();
        ...
}
```

The inheritance type that's defined with the ACL determines which ACL takes precedence when inheriting the ACL from an other document. The following table demonstrates the effect of setting different inheritance types:

Document ACL	Inherited from ACL	Inheritance Type	Resulting ACL
Permit Group A	Deny Group A	`Acl.InheritanceType.PARENT_OVERRIDES`	Deny Group A
Permit Group A	Deny Group A	`Acl.InheritanceType.CHILD_OVERRIDES`	Permit Group A

Non-Java connectors

You can provide data to the GSA through a connector using any programming language. You can do this by using a command line adaptor connector.

The 3 steps to develop and run a non-Java connector are:

1. Create the Executable Lister and Retriever programs
2. Configure the Connector
3. Run the Connector

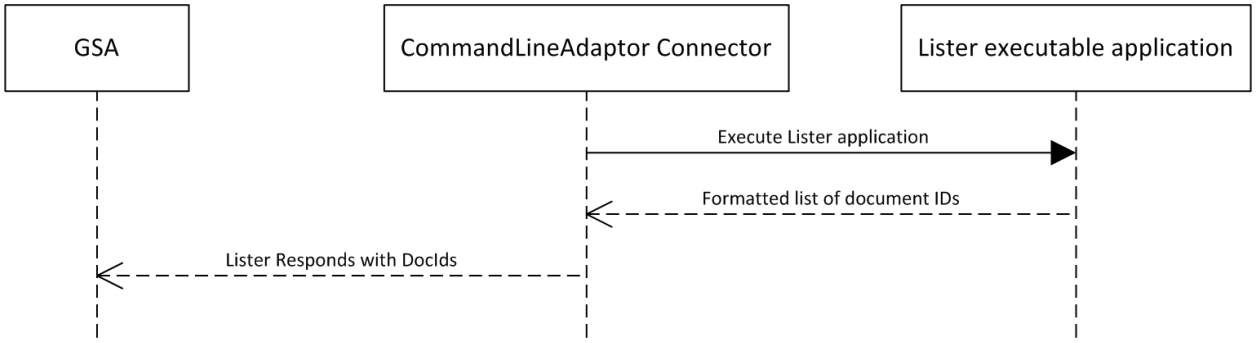
Create the Executable Lister and Retriever programs

To provide the content from your repository to the GSA, create two executable programs:

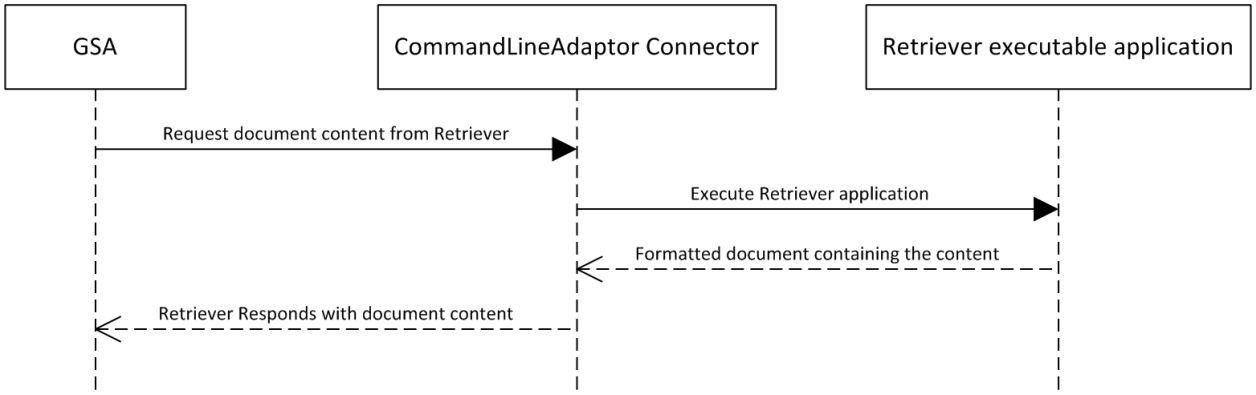
- One to feed the connector the DocIds (for the Lister)
- One to provide the document content (for the Retriever)

The command line adaptor connector is configured to define the location of these two executable application files. The Lister executable application provides a formatted list of DocIds for the connector's Lister to consume. The Retriever executable application provides the document content for the connector's Retriever to consume.

The basic flow of operation is as follows:



- The command line adaptor connector executes the Lister executable application.
- The Lister executable application provides a formatted list of DocIds to be indexed.
- The formatted data is converted to `Records`.
- The `Records`are pushed to the `DocIdPusher`, which is sent to the GSA.



GSA requests document content for indexing:

- The command line adaptor connector executes the Retriever executable application, passing the DocId as a parameter.
- The Retriever executable application provides a formatted document containing the document content.
- The document content is extracted from the formatted output provided by the Retriever executable application and is sent to the GSA for indexing.

Two executable applications are required to communicate with the GSA through the command line adaptor connector:

1. Lister executable application
2. Retriever executable application

Lister executable application

The Lister executable application provides the DocIds used by the Lister. A header is required as the first line and must adhere to the following format:

```
GSA Adaptor Data Version 1 [<delimiter>]
```

The spacing of this header text must match the provided sample. The version number may change in the future if the format is enhanced. The delimiter within the square brackets is to match the delimiter used to separate commands in the output from the Lister executable application. The delimiter usage has some restrictions that are described in the `CommandStreamParser` Javadoc.

At minimum, the Lister executable application must provide a list of DocIds. There are two recommended approaches. The first is to specify the `id` command and the DocId value on each line:

```
GSA Adaptor Data Version 1 [\n]
 id=/home/repository/docs/file1
 id=/home/repository/docs/file2
```

The second approach is to use the `id-list` command, followed by all of the DocIds.

```
GSA Adaptor Data Version 1 [\n]
 id-list
 /home/repository/docs/file1
 /home/repository/docs/file2
 /home/repository/docs/file3
 /home/repository/docs/file4
 /home/repository/docs/file5
```

You can send additional information to the Lister. The following example demonstrates setting the "`last-modified` date, and instructs the GSA to crawl these documents immediately:

```
GSA Adaptor Data Version 1 [\n]
 id=/home/repository/docs/file1
 id=/home/repository/docs/file2
 crawl-immediately
 last-modified=20110803 16:07:23
```

For a complete description of the available commands, see the `CommandStreamParser` Javadoc..

Retriever executable application

The Retriever executable application provides the content commands used by the Retriever. A header is required as the first line as described in Lister executable application .

The Retriever executable application must provide the DocId and at least one command. The following demonstrates the Retriever executable application providing the content to be indexed by the GSA:

```
GSA Adaptor Data Version 1 [\n]
 id=/home/repository/docs/file1
 content
 Document says hello
```

For a complete description of the available commands, see the `CommandStreamParser` Javadoc.

Connector configuration

The connector configuration must define the location of the Lister executable application and the Retriever executable program. The following example demonstrates the entries within `adaptor-config.properties`:

```
...
commandline.lister.cmd=./adaptor-template-get-doc-ids.sh
commandline.retriever.cmd=./adaptor-template-get-doc-contents.sh
...
```

Run the connector

The configuration properties may be set from the command line when starting the connector. The following demonstrates running a command line adaptor connector and passing in the required configuration:

```
java -cp adaptor-withlib.jar:examples/adaptor-examples.jar
    com.google.enterprise.adaptor.prebuilt.CommandLineAdaptor
          -Dgsa.hostname=mygsahostname
```

```
-Dcommandline.lister.cmd=./adaptor-template-get-doc-ids.sh
-Dcommandline.retriever.cmd=./adaptor-template-get-doc-contents.sh
```

Transforms

A transform is a process that makes modifications to a document before the GSA can index the document.

The two types of transforms currently supported are:

- Metadata transforms
- ACL transforms

Metadata transforms

When a connector is used to index documents from the content repository, it can be customized to modify metadata values using a metadata transform. This transform could be written to move metadata values from one key to another, modify metadata values, or perform other required metadata customizations. The mechanism that allows this metadata customization is called a *metadata transform*.

The aspects of developing a Metadata transform include:

- Class requirements
- Configuration
- Factory method
- Transform method

Class requirements

A metadata transform class must implement the `com.google.enterprise.adaptor.DocumentTransform` interface, override the `transform()` method, contain a factory method, and be properly configured within the `adaptor-config.properties` file.

Configuration

The configuration of the metadata transform is added to the `adaptor-config.properties` file and uses a pipeline structure. The output of one pipeline stage is fed into the next pipeline stage. This allows for reuse of pipeline stages in other transforms.

The first configuration line defines all of the stages in the `transform` pipeline:

```
transform.pipeline=stage1, stage2...stageX
```

The next line defines the factory method of the `stage1` pipeline:

```
transform.pipeline.stage1.factoryMethod=<fully qualified factory class name>
```

The next lines define all of the variables required by this transform:

```
transform.pipeline.stage1.<arg1>=<value1>
transform.pipeline.stage1.<arg2>=<value2>
transform.pipeline.stage1.<arg3>=<value3>
```

The above example refers to `stage1`, but you can assign the pipeline stage names (as appropriate) to best describe the pipeline stage purpose.

Factory method

The factory method is a static class within the transform class that's responsible for returning an instance of the transform class with the configuration data.

The following example demonstrates a factory method that returns an instance of the `MetadataTransformExample` class, passing in the configuration values for the `src` and `dest` keys:

```
public static MetadataTransformExample create(Map<String, String> cfg) {
        return new MetadataTransformExample(cfg.get("src"), cfg.get("dest"));
}
```

The method argument is fixed as a `Map` of configuration name/value pairs. These values are passed in by the connector framework's transform pipeline loader.

The `HelloWorldConnector` class contains a metadata transform example that adds configured `"taste"` values to the metadata of the indexed documents. You can download the sample `MetadataAddition` transform class from this URL.

For this metadata transform, the factory method only requires the setting of a single `"taste"` value from the configuration file.

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

Transform method

The `transform()` method performs the customized modifications to the metadata for each document before the GSA indexes it. Two parameters are passed into this method: a `Metadata` object and a `params` Map.

The `Metadata` object contains all of the metadata associated with the document before the transform. The transform makes modifications to this object. The `params` Map contains two entries: one for `DocId` and the other for `Content-Type`. You can use these values as needed for the transform logic.

Using the sample `MetadataAddition` transform class (in Factory method), the `transform()` method demonstrates reading the "`taste`" values from the configuration file, and adding them to the existing "`taste`" metadata values.

```
public void transform(Metadata metadata, Map<String, String> params) {
        Set<String> values = metadata.getAllValues(META_TASTE);
        if (values.isEmpty()) {
                log.log(Level.INFO, "no metadata {0}. Skipping", META_TASTE);
        } else {
                log.log(Level.INFO,
                                "adding values {1} for existing metadata {0}  ",
                                new Object[] { META_TASTE, valuesToAdd });
                metadata.set(META_TASTE, combine(values, valuesToAdd));
        }
}
```

ACL transforms

You can use ACL transforms to conditionally modify the ACL rules of a document before it's indexed by the GSA.

ACL transforms are accomplished through the Configuration of your connector.

Configuration

An ACL transform differs from a metadata transform: there's no ability to override a transform class. All of the ACL transform rules are added to the `adaptor-config.properties` file, following a specific format:

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

The ACL transform uses a rules-based configuration, and each configured rule is applied to each document. The `rule sequence` starts at 0, and increments for each additional rule entry. The rule value contains a semicolon, delimited list defining the ACL entry for each document to be matched, and the ACL to be converted to.

The “ACL match” portion of this value is what the transform uses to identify a matching ACL to be changed. The “ACL to be converted” portion of the value defines the change to be made to the ACL.

The following ACL transform configuration demonstrates three different ACL transforms:

```
transform.acl.0=type=user, domain=gsatestlab; domain=gsaprodlab
transform.acl.1=type=group, domain=gsatestlab; domain=gsatestlab.com
transform.acl.2=type=user, name=user1; domain=gsatestlab, name=user2
```

The above rule entries behave as follows:

1. Rule 0— For documents where the ACL matches user principles with a domain of “gsatestlab”, change the domain to be “gsaprodlab”.

2. Rule 1—For documents where the ACL matches group principles with a domain of “gsatestlab”, change the domain to be “gsatestlab.com”.

3. Rule 2—For documents where the ACL matches for user principles with a name of “user1”, change the name to be “user2” and the domain to be “gsatestlab”.

Graph traversal

As described in The Lister, the Lister provides all of the unique DocIds to the GSA for indexing. There’s another approach to provide the unique IDs to the GSA. This approach is called *graph traversal.*

Graph traversal is a technique of providing virtual webpages containing content links for the GSA to crawl and index.
To implement graph traversal, develop your connector to Use the Retriever as a Lister.

Use the Retriever as a Lister

The GSA can be configured to “Follow and Crawl” links within the content. It’s this feature of the GSA that’s utilized to send the document links in logical batches to be indexed.

For content stored in a hierarchy, the virtual HTML pages contain links to the content found within a folder, and virtual links to sub folders. The Retriever determines what action to take

based on the passed DocId. If the DocId is identified as a virtual page ID, then a virtual HTML page is constructed and sent to the GSA. Otherwise, the document content is provided to the GSA for indexing.

The following example demonstrates the creation of a virtual HTML page that contains links to documents and a virtual link to a sub folder:

```
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("<br></br>");
                writer.write("<a href=\"1001\">doc 1001</a>");
                writer.write("<br></b r>");
                writer.write("<a href=\"1002\">doc 1002</a>");
                writer.write("<br></br>");
                writer.write("<a href=\"1003\">doc 1003</a>");
                writer.write("<br></br>");
                writer.write("<a href=\"vdoc_2\">Sub Folder</a>");
                writer.write("<br></br>");
                writer.write("</body></html>");
                writer.close();
        }
        // Retriever implementation removed for brevity
}
```

When using this approach, ensure that the virtual IDs used to identify the logical grouping of content don't conflict with the DocIds in the content repository.

Tutorials

The tutorials walk through the implementation of many features offered by GSA Connectors 4.0. These contain everything that you'll need to develop your own connector, to allow the GSA to index your non-HTTP content.

All of the tutorial source code can be downloaded from the Plexi resource Wiki site.

The included tutorials are:

- Write your first connector
- HelloWorld
- Authentication and authorization
- Meta transforms

Write your first connector

If you're not familiar with the connector Lister or Retriever model, we recommend to first review the overview in the Lister or Retriever model.

This tutorial covers the most basic connector and can be downloaded here:

In this tutorial, you will:

- Create a class that implements the `Adaptor` interface.
- Create the Lister and Retriever methods.

The connector class must implement the `Adaptor` interface and override two methods; `getDocIds` (the Lister), and `getDocContent` (the Retriever). One common option is to extend the `AbstractAdaptor` class; it provides default implementations for common GSA Connector V4 methods.

The following code shows an example of adding new DocId objects to an array that's passed back to the `DocIdPusher` to allow the GSA to begin the next step, which is to call the Retriever. These unique DocIds represent a meaningful identifier to your content, which could be a file, a node in an XML document, a row in a database, or any other content source.

```
/** Gives list of document ids that you'd like on the GSA. */
@Override
public void getDocIds(DocIdPusher pusher) throws InterruptedException {
        ArrayList<DocId> mockDocIds = new ArrayList<DocId>();
        /* Replace this mock data with code that lists your repository. */
        mockDocIds.add(new DocId("1001"));
        mockDocIds.add(new DocId("1002"));
        pusher.pushDocIds(mockDocIds);
}
```

The following Retriever demonstrates how the GSA requests the content to be indexed by ID. In this example:

```
/** Gives the bytes of a document referenced with id. */
@Override
public void getDocContent(Request req, Response resp) throws IOException {
        DocId id = req.getDocId();
        String str;
        if ("1001".equals(id.getUniqueId())) {
                str = "Document 1001 says hello and apple orange";
        } else if ("1002".equals(id.getUniqueId())) {
                str = "Document 1002 says hello and banana strawberry";
```

```
        } else {
                resp.respondNotFound();
                return;
        }
        resp.setContentType("text/plain; charset=utf-8");
        OutputStream os = resp.getOutputStream();
        os.write(str.getBytes(encoding));
}
```

The default main method is used to execute the connector:

```
/** Call default main for adaptors. */
public static void main(String[] args) {
        AbstractAdaptor.main(new AdaptorTemplate(), args);
}
```

HelloWorld

The HelloWorld tutorial showcases most of the features available for use when developing a connector. You can download the files required for this tutorial.

The aspects of developing the HelloWorld connector includes:

- Create a connector class
- Initialize the connector
- Create the Lister method
- Create the Retriever method
- Add support for incremental updates

Create a connector class

The minimum prerequisite for a connector class is that it must implement the `Adaptor` interface and implement the `getDocIds(DocIdPusher)` Lister method and the `getDocContent(Request,Response)` Retriever method. In this example, the connector extends the `AbstractAdaptor` class (which implements the `Adaptor` interface) and also implements the `PollingIncrementalLister` interface, which is used to discover recently changed documents and inform the GSA.

```
public class HelloWorldConnector extends AbstractAdaptor implements
  PollingIncrementalLister
```

Initialize the connector

The following code shows the `init()` method initializing the connector with the current context:

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

The incremental lister is registered with the `AdaptorContext` object. For further details of the `PollingIncrementalLister` interface, see Incremental Updates.

An instance of the custom `HelloWorldAuthenticator` class is registered with the `AdaptorContext` object to be identified as providing authentication and authorization. An HTTP handler is registered for use by the authentication process. For further information, see Authentication and Authorization.

Create the Lister method

The following code demonstrates a Lister that provides an example of pushing simple DocIds to the GSA, a more detailed Record, and a named resource. For a description of a Lister, see Lister. Configuring and pushing Records is detailed in Runtime configuration. Building and pushing named resources is described in Named resource ACL inheritance.

```
@Override
public void getDocIds(DocIdPusher pusher) throws InterruptedException {
        log.entering("HelloWorldConnector", "getDocIds");
        ArrayList<DocId> mockDocIds = new ArrayList<DocId>();
        // push docids
        mockDocIds.add(new DocId(""));
        mockDocIds.add(new DocId("1001"));
        mockDocIds.add(new DocId("1002"));
        pusher.pushDocIds(mockDocIds);
        // push records
        DocIdPusher.Record record = new DocIdPusher.Record.Builder(new DocId(
                        "1009")).setCrawlImmediately(true).setLastModified(new Date())
                        .build();
        pusher.pushRecords(Collections.singleton(record));
        // push named resources
```

```
        HashMap<DocId, Acl> aclParent = new HashMap<DocId, Acl>();
        ArrayList<Principal> permits = new ArrayList<Principal>();
        permits.add(new UserPrincipal("user1", "Default"));
        aclParent.put(new DocId("fakeID"), new Acl.Builder()
                        .setEverythingCaseInsensitive().setPermits(permits)
                        .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                        .build());
        pusher.pushNamedResources(aclParent);
}
```

Create the Retriever method

The following code demonstrates a Retriever method, which is detailed in Retriever:

```
/** Gives the bytes of a document referenced with id. */
@Override
public void getDocContent(Request req, Response resp) throws IOException {
        log.entering("HelloWorldConnector", "getDocContent");
        DocId id = req.getDocId();
        log.info("DocId '" + id.getUniqueId() + "'");
```

The following portion of the Retriever method demonstrates the creation of a virtual HTML page. This page contains links that the GSA can crawl. This technique is detailed in Using the Retriever as a Lister.

```
// Hard-coded list of our doc ids
if ("".equals(id.getUniqueId())) {
        // this is a the root folder, write some URLs
        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("<!DOCTYPE html>\n<html><body>");
        writer.write("<br></br>");
        writer.write("<a href=\"1001\">doc_not_changed</a>");
        writer.write("<br></b r>");
        writer.write("<a href=\"1002\">doc_changed</a>");
        writer.write("<br></br>");
        writer.write("<a href=\"1003\">doc_deleted</a>");
        writer.write("<br></br>");
        writer.write("<a href=\"1004\">doc_with_meta</a>");
        writer.write("<br></br>");
        writer.write("<a href=\"1005\">doc_with_ACL</a>");
        writer.write("<br></br>");
        writer.write("<a href=\"1006\">doc_with_ACL_Inheritance</a>");
        writer.write("<br></br>");
        writer.write("<a href=\"1007\">doc_with_Fragment</a>");
```

```
        writer.write("<br></br>");
        writer.write("<a href=\"1008\">doc_with_Fragment</a>");
        writer.write("<br></br>");
        writer.write("</body></html>");
        writer.close();
```

The following portion of the Retriever method checks the last modified date and conditionally modifies the content to be indexed by the GSA. Normally, the last-modified date is retrieved from the document and represents the time of the last modification of that the document.

```
} else if ("1001".equals(id.getUniqueId())) {
        // Example with If-Modified-Since
        // Set lastModifiedDate to 10 minutes ago
        Date lastModifiedDate = new Date(
                        System.currentTimeMillis() - 600000);
        if (req.hasChangedSinceLastAccess(lastModifiedDate)) {
                if (req.getLastAccessTime() == null) {
                        log.info("Requested docid 1001 with No If-Modified-Since");
                } else {
                        log.info("Requested docid 1001 with If-Modified-Since < 10 minutes");
                }
                resp.setLastModified(new Date());
                Writer writer = new OutputStreamWriter(resp.getOutputStream());
                writer.write("Menu 1001 says latte");
                writer.close();
        } else {
                log.info("Docid 1001 Not Modified");
                resp.respondNotModified();
        }
```

The following portion of the Retriever is an example of content being sent to the GSA without any further information:

```
} else if ("1002".equals(id.getUniqueId())) {
        // Very basic doc
        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1002 says cappuccino");
        writer.close();
```

The following portion of the Retriever demonstrates a condition where the content is not found in the repository. This `respondNotFound()` method is sent back to the GSA to update the index.

```
} else if ("1003".equals(id.getUniqueId())) {
        // Alternate between doc and a 404 response
        if (provideBodyOfDoc1003) {
                Writer writer = new OutputStreamWriter(resp.getOutputStream());
                writer.write("Menu 1003 says machiato");
                writer.close();
        } else {
                resp.respondNotFound();
        }
        provideBodyOfDoc1003 = !provideBodyOfDoc1003;
```

This demonstration uses the `provideBodyOfDoc1003` variable to toggle whether the content body, or a `respondNotFound` response is returned.

The following portion of the Retriever demonstrates sending additional metadata and a custom display URL along with the document content:

```
} else if ("1004".equals(id.getUniqueId())) {
        // doc with metdata & different display URL
        resp.addMetadata("flavor", "vanilla");
        resp.addMetadata("flavor", "hazel nuts");
        resp.addMetadata("taste", "strawberry");

        try {
                resp.setDisplayUrl(new URI("http://fake.com/a"));
        } catch (URISyntaxException e) {
                log.info(e.getMessage());
        }
        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1004 says espresso");
        writer.close();
```

The following portion of the Retriever shows the creation of ACLs that are associated with a document:

```
} else if ("1005".equals(id.getUniqueId())) {
        // doc with ACLs
        ArrayList<Principal> permits = new ArrayList<Principal>();
        permits.add(new UserPrincipal("user1", "Default"));
        permits.add(new UserPrincipal("eric", "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()
                        .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                        .setPermits(permits).setDenies(denies).build());

        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1005 says americano");
        writer.close();
```

The following portion of the Retriever demonstrates ACL inheritance. For more information, see Document ACL inheritance

```
} else if ("1006".equals(id.getUniqueId())) {
        // Inherit ACLs from 1005
        ArrayList<Principal> permits = new ArrayList<Principal>();
        permits.add(new GroupPrincipal("group3", "Default"));
        ArrayList<Principal> denies = new ArrayList<Principal>();
        denies.add(new GroupPrincipal("group3", "Default"));

        resp.setAcl(new Acl.Builder().setEverythingCaseInsensitive()
                        .setInheritanceType(Acl.InheritanceType.PARENT_OVERRIDES)
                        .setInheritFrom(new DocId("1005")).setPermits(permits)
                        .setDenies(denies).build());

        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1006 says misto");
        writer.close();
```

The following portion of the Retriever demonstrates fragment ACL. This technique is used when a single document provides different access controls to different children documents. For more information, see Fragment ACL Inheritance.

```
} else if ("1007".equals(id.getUniqueId())) {
        // Inherit ACLs from 1005 & 1006
        ArrayList<Principal> permits = new ArrayList<Principal>();
        permits.add(new GroupPrincipal("group5", "Default"));

        resp.putNamedResource(
                        "Whatever",
                        new Acl.Builder()
```

```
                                        .setEverythingCaseInsensitive()
                                        .setInheritFrom(new DocId("1006"))
                                        .setPermits(permits)
                                        .setInheritanceType(
                                                Acl.InheritanceType.PARENT_OVERRIDES)
                                        .build());

        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());

        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1007 says frappuccino");
        writer.close();
```

The following portion of the Retriever demonstrates ACL inheritance; see Document ACL inheritance:

```
} else if ("1008".equals(id.getUniqueId())) {
        // Inherit ACLs from 1007
        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("1007"), "Whatever")
                        .setDenies(denies).build());
        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1008 says coffee");
        writer.close();
```

The following portion of the Retriever shows that a document may be identified as a `secure` document. For more information, see Authorization by Connector.

```
} else if ("1009".equals(id.getUniqueId())) {
        // Late Binding (security handled by connector)
```

```
        resp.setSecure(true);
        Writer writer = new OutputStreamWriter(resp.getOutputStream());
        writer.write("Menu 1009 says espresso");
        writer.close();
```

The following portion of the Retriever demonstrates the condition where document content cannot be found for the passed DocId. The GSA is notified that the document cannot be found so that it may delete it from the index.

```
} else {
        resp.respondNotFound();
}
```

Add support for incremental updates

The following method is used to identify modified documents. For more information, see Incremental Updates.

```
@Override
public void getModifiedDocIds(DocIdPusher pusher) throws IOException,
                InterruptedException {
        ArrayList<DocId> mockDocIds = new ArrayList<DocId>();
        mockDocIds.add(new DocId("1002"));
        pusher.pushDocIds(mockDocIds);
}
```

The `main` method is used to execute the connector.

```
/** Call default main for adaptors. */
public static void main(String[] args) {
        AbstractAdaptor.main(new HelloWorldConnector(), args);
}
```

Authentication and authorization

Implementing authentication and authorization for the HelloWorld example includes:

- Create the Authenticator class
- Authenticate the user
- Determine if the user is authorized
- Handle the submitted login

Create the Authenticator class

As described in Authentication and Authorization, a custom class is required. In this example, the same class provides both functions, so it must implement the following interfaces: `AuthnAuthority`, `AuthzAuthority`, and `HttpHandler`.

```
class HelloWorldAuthenticator implements AuthnAuthority, AuthzAuthority,
                HttpHandler
```

Authenticate the user

For authentication, the `authenticateUser()` method of the `AuthnAuthority` interface must be overridden. See Implement authentication within the connector for a description of this process.

```
@Override
public void authenticateUser(HttpExchange exchange, Callback callback)
                throws IOException {

        log.entering("HelloWorldAuthenticator", "authenticateUser");
        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();
}
```

Determine if the user is authorized

To handle the authorization, the `isUserAuthorized()` method of the `AuthzAuthority` interface must be overridden. See Implement authorization within the connector for a description of this process.

```
@Override
public Map<DocId, AuthzStatus> isUserAuthorized(AuthnIdentity userIdentity,
        Collection<DocId> ids) throws IOException {

    HashMap<DocId, AuthzStatus> authorizedDocs = new HashMap<DocId,
AuthzStatus>();

    for (Iterator<DocId> iterator = ids.iterator(); iterator.hasNext();) {
        DocId docId = iterator.next();
        // if authorized
        authorizedDocs.put(docId, AuthzStatus.PERMIT);
    }
    return authorizedDocs;
}
```

Handle the submitted login

As described in Implement authentication within the connector, the `handle()` method of the `AuthnAuthority` interface intercepts the submitted login form and returns the identity to the GSA.

```
/**
 * Handle the form submit from /samlip<br>
 * If all goes well, this should result in an Authenticated user for the
 * session
 */
@Override
public void handle(HttpExchange ex) throws IOException {
    log.entering("HelloWorldAuthenticator", "handle");

    callback = getCallback(ex);
    if (callback == null) {
        return;
    }

    Map<String, String> parameters = extractQueryParams(ex.getRequestURI()
            .toString());
```

```
        if (parameters.size() == 0 || null == parameters.get("userid")) {
                log.warning("missing userid");
                callback.userAuthenticated(ex, null);
                return;
        }
        String userid = parameters.get("userid");
        SimpleAuthnIdentity identity = new SimpleAuthnIdentity(userid);
        callback.userAuthenticated(ex, identity);
}
```

The following utility method is used by the sample `getCallback()` method. It sends a response that include a String message and a `200` code (request has succeeded).

```
// Return a 200 with simple response in body
private void sendResponseMessage(String message, HttpExchange ex)
                throws IOException {
        OutputStream os = ex.getResponseBody();
        ex.sendResponseHeaders(200, 0);
        os.write(message.getBytes());
        os.flush();
        os.close();
        ex.close();
}
```

The following code is used to retrieve the `Callback` object from `Session`. As described in Implement authentication within the connector, this `Callback` object is used to send the identity information back to the GSA.

```
// Return the Callback method,
// or print error if the handler wasn't called correctly
private Callback getCallback(HttpExchange ex) throws IOException {
        Session session = context.getUserSession(ex, false);
        if (session == null) {
                log.warning("No Session");
                sendResponseMessage("No Session", ex);
                return null;
        }
        Callback callback = (Callback) session.getAttribute("callback");
        if (callback == null) {
                log.warning("Something is wrong, callback object is missing");
                sendResponseMessage("No Callback Specified", ex);
        }
```

```
        return callback;
}
```

The following utility method is used to extract the username and password parameters from the submitted form:

```
// Parse user/password/group params
private Map<String, String> extractQueryParams(String request) {
        Map<String, String> paramMap = new HashMap<String, String>();
        int queryIndex = request.lastIndexOf("?");

        if (queryIndex == -1) {
                return paramMap;
        }
        String query = request.substring(queryIndex + 1);
        String params[] = query.split("&", 4);
        if (query.equals("")) {
                return paramMap;
        }
        try {
                for (int i = 0; i < params.length; ++i) {
                        String param[] = params[i].split("%2F=", 2);
                        paramMap.put(URLDecoder.decode(param[0], "UTF-8"),
                                        URLDecoder.decode(param[1], "UTF-8"));
                }
        } catch (UnsupportedEncodingException e) {
                log.warning("Request parameters may not have been properly encoded: "
                                + e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
                log.warning("Wrong number of parameters specified: "
                                + e.getMessage());
        }
        return paramMap;
}
```

Meta transforms

The `MetaDataTransformExample` class demonstrates the process of moving metadata values from one key to another as defined in the Connector configuration file.
This example moves the values from one key to another key. As an example, the values of "creator" are moved to "author".

The process of developing a meta transform is detailed in Metadata transforms.

Implementing metadata transform for the HelloWorld example includes:

- Create the transform class
- Create custom constructor and factory methods
- Create the transform method

Create the transform class

The custom transform class must implement the `DocumentTransform` interface and override the `transform()` method.

```
public class MetadataTransformExample implements DocumentTransform
```

Create custom constructor and factory methods

The private constructor receives the keys used for this transform and sets them with the object.

```
private MetadataTransformExample(String originalKey, String changedKey) {
        if (null == originalKey || null == changedKey) {
                throw new NullPointerException();
        }
        this.src = originalKey;
        this.dest = changedKey;
        if (src.equals(dest)) {
                log.log(Level.WARNING,
                                "original and destination key the same: {0}", src);
        }
}
```

The following static factory method example returns an instance of the `MetadataTransformExample` class:

```
/** Makes transform from config with "src" and "dest" keys. */
public static MetadataTransformExample create(Map<String, String> cfg) {
        return new MetadataTransformExample(cfg.get("src"), cfg.get("dest"));
}
```

Create the transform method

The following `transform()` method example demonstrates moving values from one metadata key to another:

```
@Override
public void transform(Metadata metadata, Map<String, String> params) {
        if (src.equals(dest)) {
                return;
        }
        Set<String> valuesToMove = metadata.getAllValues(src);
        if (valuesToMove.isEmpty()) {
                log.log(Level.FINE, "no values for {0}. Skipping", src);
        } else {
                log.log(Level.FINE, "moving values from {0} to {1}: {2}",
                                new Object[] { src, dest, valuesToMove });
                Set<String> valuesAlreadyThere = metadata.getAllValues(dest);
                metadata.set(dest, combine(valuesToMove, valuesAlreadyThere));
                log.log(Level.FINER, "deleting source {0}", src);
                metadata.set(src, Collections.<String> emptySet());
        }
}
```

References

- GSA Adaptors Page
- Connector 4 Javadocs
- Developer FAQ
- GSA Help Center
- GSA Administration Guide for Connectors
- Search Protocol Reference
- GSA Product Documentation 7.2
- Connector Framework Source Code
- Connector Examples
- Enabling Security