GSI Local-Interface-Preserving Web Services

How to Create a Web Service using GSI Authentication and Proxy Delegation

The following is a description of how to implement and deploy an Apache-Axis based web service using the GSI Authentication/Proxy Delegation and local-interface preserving model adopted by our group.

The security mechanism is described in more detail at ncsa-gsihttps.

Here we will focus on how to generate the service infrastructure, how to write the necessary adapter code for a specific service, and how to deploy the service.

Definitions:

IMPLEMENTATION	the Java class or classes constituting the functionality of the service
LOCAL INTERFACE	the API and data objects used by the implementation; these are exposed to other classes, but not as remote objects
REMOTE INTERFACE	the service interface generated from the local interface by Axis which exports the latter as remote objects
CLIENT ADAPTER	a service client which uses the local rather than remote interface
ENDPOINT ADAPTER	the service skeleton which also translates from the remote interface back into the local interface of the implementation

Dependencies:

We assume that the build process will be done through Eclipse, and that the user is familiar with that environment. The various phases of the build are handled by calls to the `ncsa.services.build.ServiceBuilder`, run, however, as a Java Application (not as a plugin).

The following Java libraries are necessary for generating the service and client. For convenience, we have wrapped them as Eclipse plugins. Not included in this list are the Eclipse, Ant, and JUnit plugins, which should be part of the standard Eclipse distribution.

External Libraries / Plugins

LIBRARY	INCLUDED IN PLUGIN
dom4j-1.5.2.jar	org.dom4j
commons-beanutils-1.7.0.jar	org.apache.commons
commons-cli-1.0.jar	org.apache.commons
commons-discovery-0.2.jar	org.apache.commons
commons-lang-2.0.jar	org.apache.commons
commons-logging-1.0.4.jar	org.apache.commons
commons-collections-3.1.jar	org.apache.commons
commons-codec-1.3.jar	org.apache.commons.codec
axis-1.2.1.jar	org.apache.axis
jaxrpc-1.1.jar	org.apache.axis
saaj-1.2.jar	org.apache.axis
log4j-1.2.7.jar	org.apache.log4j
spring-1.1.5.jar	org.spring
jmx.jar	javax.management
servlet-api.jar	javax.servlet
qname.jar	com.ibm.wsdl
wsdl4j.jar	com.ibm.wsdl
cryptix-asn-1.0.jar	org.globus.jglobus
cryptix-random-1.0.jar	org.globus.jglobus
cryptix-32-1.0.jar	org.globus.jglobus
jce-jdk13-120.jar	org.globus.jglobus

jgss-1.0.jar	org.globus.jglobus
puretls-1.0.jar	org.globus.jglobus
cog-jglobus-1.2.1	org.globus.jglobus
cog-axis.jar	cog.axis
cog-url.jar	cog.axis

NCSA plugins

NCSA PLUGIN
ncsa.tools.common
ncsa.tools.common.eclipse.descriptors
ncsa.ca.certs
ncsa.services.build
ncsa.services.client
ncsa.services.endpoint

Optional (database)

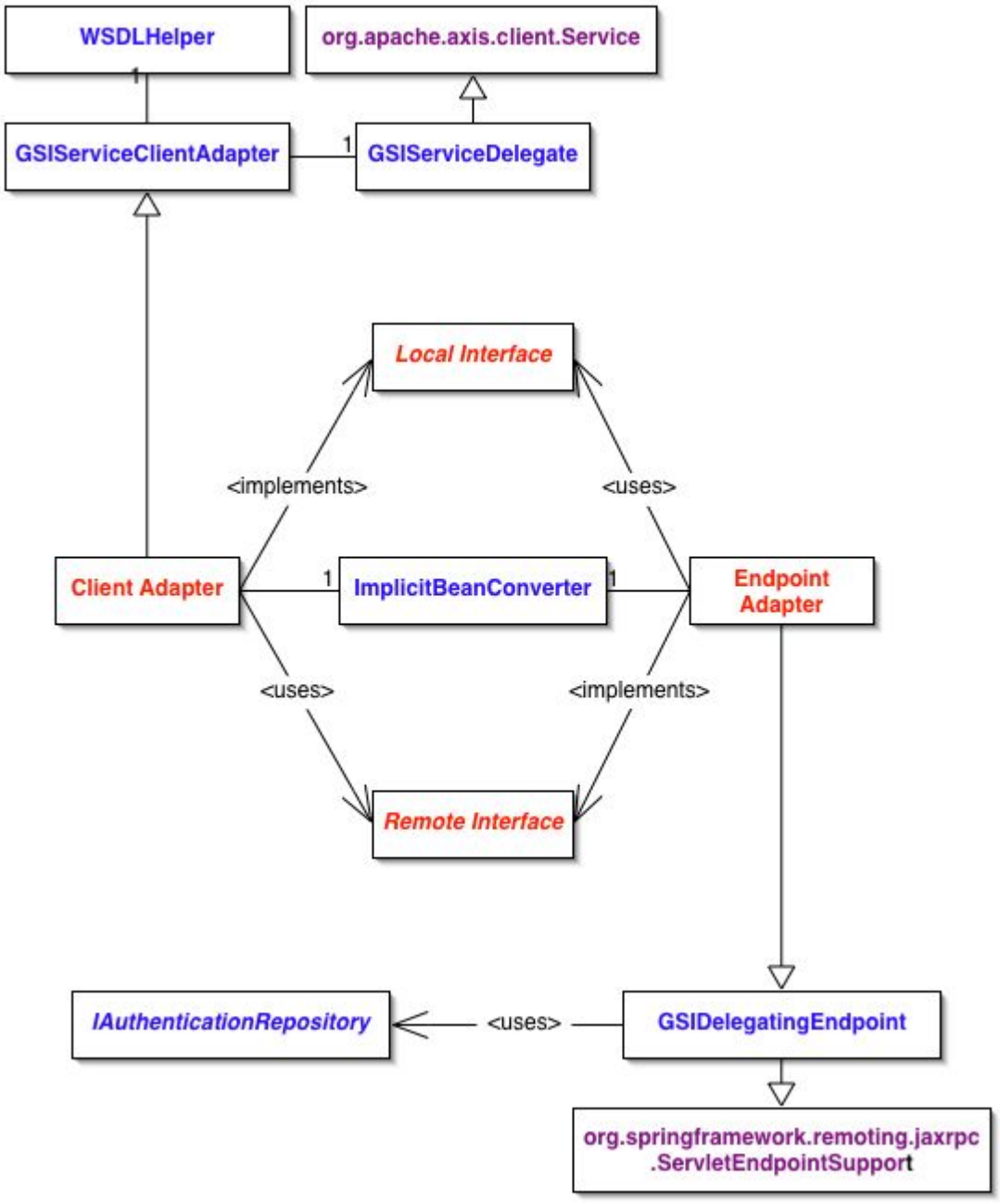
If you wish to use the service builder to do database initialization, then you will also need:

LIBRARY	INCLUDED IN PLUGIN
commons-dbcp-1.2.1.jar	net.sf.hibernate
aopalliance-1.0.jar	net.sf.hibernate
cglib-nodep-2.1-dev.jar	net.sf.hibernate
hibernate-2.1.7.jar	net.sf.hibernate
jta-1.2.jar	net.sf.hibernate
odmg-3.0.jar	net.sf.hibernate

along with some JDBC connector, such as mysql-jdbc-3.1.7.jar.

UML

The following diagram lays out the invariant structure of a typical service. Violet class names represent external packages; blue represents `ncsa.services.client` or `ncsa.services.endpoint` abstract classes or interfaces; red represents the classes or interfaces which need to be written or generated.



Step-by-step instructions.

Throughout the following, we use `Example` to stand for the local interface class.

---

(1) Set up the plugin structure.

Create the plugins.

For the **Example** service, we recommend creating the following plugins:

PLUGIN	CONTAINS	DEPENDENCIES	SUB-DIRECTORIES
`example.interface`	`Example`, plus any other special objects used by `Example`	-	src
`example.impl`	`ExampleImpl` and supporting classes (the implementation)	`example.interface`	src, +
`example.wstypes`	all the remote classes generated by `wsdl4j`	`example.interface`	src

example.client	ExampleClientAdapter plus the example.wsdl file	example.interface, example.wstypes	src, resources
example.endpoint	ExampleEndpointAdapter	example.interface, example.wstypes	src, deploy

The src directories will contain pre-existing code only in the interface and impl plugins; after generating the wstypes source code, you will write the client and endpoint classes.

The resources directory is another source folder which will contain the .wsdl.

The deploy directory is used to create the .war. It should have the following in it:

1. The usual .jsp and .html files used by Axis (optional): fingerprint.jsp, happyaxis.jsp, index.html;
2. A WEB-INF directory, set up in the usual manner (with classes and any other resource subdirectories necessary, along with the requisite configuration files, etc.); plus
    a. server-config.wsdd, containing the basic Axis deployment description
    b. web.xml, describing the web-apps (the Axis servlet) to Tomcat

Versions of the latter two can usually be copied and adapted from existing services.

Create a feature.

In addition, you will need to create a feature for building the service .war:

FEATURE	CONTAINS	DEPENDENCIES
example.feature	the .xml files for generating a .war	example.interface, example.wstypes, example.endpoint, example.impl, + all implementation dependencies

When you have finished expressing all the necessary dependencies for creating the .war file as the dependencies of the feature, select the feature.xml file in the Eclipse editor, right click (control-click) and select "PDE Tools >> Create Ant Build File" to generate all the necessary build files for the plugins in your workspace on which this feature depends.

---

(2) Create a properties file to be used by the service builder.

The following template can be used. Replace the following references appropriately:

${name}	e.g., Example
${pname}	e.g., example
${package}	e.g., a.b.example
${urnpackage}	e.g., example.b.a
${eclipse}	the path to the installation
${war.dir}	output directory for the .war file

```
service.name=${name}
wsdl.name=${pname}.wsdl
package.dir=${package}/service
endpoint.package=${package}.service.endpoint
client.plugin.path=${eclipse}/${package}.client
endpoint.plugin.path=${eclipse}/${package}.endpoint
wstypes.plugin.path=${eclipse}/${package}.wstypes
feature.build.path=${eclipse}/${package}.feature
war.path=${war.dir}/${pname}.war
build.dir=${eclipse}/ncsa.services.build/${package}
deploy.dir=${eclipse}/${package}.endpoint/deploy
server.config.path=${eclipse}/${package}.endpoint/deploy/WEB-INF/server-config.wsdd
impl.interface=Example
default.url=httpg://localhost:8443/${pname}/services/${name}
style=WRAPPED
namespace=urn:service.${urnpackage}
scope=application
verbose=true
includeStub=false
types.namespace=types.service.${urnpackage}
types.packages=
```

> *hibernate.mappings.root=${eclipse}/${package}.impl/resources/mappings*
> *hibernate.delimiter=:*
> *hibernate.properties.file=${eclipse}/${package}.endpoint/deploy/WEB-INF/classes/datasource.properties*
> *hibernate.output.file=${eclipse}/ncsa.services.build/schemas/${pname}.sql*
> *hibernate.quiet=false*
> *hibernate.text.only=false*
> *hibernate.drop=false*
> *hibernate.create=true*

Notes

1. The settings above are for generating wrapped (document-literal) wsdl from the local interface.
2. The packages of all data types referenced by `Example` should be included under `types.packages` using a comma-separated list (no spaces); these will all be mapped by the wsdl generator to the indicated `types.namespace`.

For an explanation of the Hibernate settings, see below.

Place this properties file in some convenient local directory.

---

(3) Create the service infrastructure.

Inside of Eclipse, select "Run", choosing the `ncsa.services.build.ServiceBuilder` from the `ncsa.services.build` plugin you have installed. Set the command-line arguments to:

- service <path to properties file>

and run it.

The following should happen:

1. A .wsdl file is generated;
2. From the .wsdl, the source code for the necessary remote types and interfaces is generated;
3. The service description is added to the indicated server configuration (at the server.config.path, above);
4. The source code is copied to the wstypes plugin;
5. The .wsdl file is copied to the client plugin resources directory.

When this automatic generation completes, you will need to map the generated classes in the wstypes plugin as extensions to the `ncsa.services.client.wstypes` extension point, to enable the client to run as a plugin.

The extension point is declared in the `ncsa.services.client` plugin; the extensions should be declared in the wstypes plugin where they reside.

---

(4) Implement the specific service-layer endpoints.

This is the coding phase of the build; in essence, the connecting code between LOCAL --> REMOTE (client) and REMOTE --> LOCAL (endpoint) needs to be developed.

However, our `ncsa.services.client` and `ncsa.services.endpoint` plugins, along with the utilities in `ncsa.tools.common` should go some way to facilitating, and lending uniformity to, the process.

Authentication on both the client and service side is handled by the abstract or delegate classes provided by these plugins, so there is no security work to do *per se*. In the case that your service needs to use the proxy delegated to it (e.g., in order to call another service or to do file transfers), then the endpoint needs to store the proxy so that the implementation has access to it. The abstract base class `ncsa.service.endpoint.GSIDelegatingEndpoint` automatically does this via a static in-memory implementation (see further below).

We will treat the endpoint and client adapters separately, in that order. However, in both cases, the process is similar, and involves the following:

1. Convert local to remote or remote to local objects (using a utility class);
2. Invoke the analogous method on the delegate class.

Object conversion.

Axis has trouble with Java `Collection` and `Map` types, so objects destined to go over the wire should substitute arrays for these.

Given the exclusion of such untyped containers along with the fact that wstypes are merely cloned classes with a different package name, conversion from local to remote object and back can be handled entirely by reflection. This is done via the `ncsa.tools.common.utils.bean.ImplicitBeanConverter`.

Endpoint Adapter.

This class should extend the `ncsa.services.endpoint.GSIDelegatingEndpoint` class and implement the remote service interface (i.e., the wstypes interface generated from the local interface). For each method of the remote interface, it should convert any special data types (i.e., those in the wstypes plugin) to their local variants, and then use those to call the corresponding method on the class implementing the local interface. Outgoing return objects or specialized exceptions need to be converted back to their wire equivalents.

By way of example, we show here the endpoint adapter for a service which accepts submissions of workflow "ensembles" to be managed.

"Ensemble Service Endpoint Adapter"

```
package ncsa.services.ensemble.service.endpoint;

import ncsa.services.endpoint.GSIDelegatingEndpoint;
import ncsa.services.ensemble.service.EnsembleBroker;
import ncsa.services.ensemble.service.types.EnsembleDescriptor;
import ncsa.services.ensemble.service.types.EnsembleHandle;
import ncsa.tools.common.util.bean.ImplicitBeanConverter;

public class EnsembleBrokerEndpoint extends GSIDelegatingEndpoint implements
                EnsembleBroker
{
        // INPUT
        private ncsa.services.ensemble.EnsembleBroker impl;

        public void onInit()
        {
                super.onInit();
                impl = ( ncsa.services.ensemble.EnsembleBroker )getWebApplicationContext()
                 .getBean( "ensembleBrokerImpl" );
        }

        public EnsembleHandle submit( EnsembleDescriptor descriptor, String instanceId )
        {
                 try {
                        logger.debug( "extracting user info " );
                        String user = extractAuthorizationInfo();
                        logger.debug( "user: " + user );
                        descriptor.setUser( user );
                        ImplicitBeanConverter ibc = new ImplicitBeanConverter();
                        ncsa.services.ensemble.EnsembleDescriptor d
                                = ( ncsa.services.ensemble.EnsembleDescriptor )ibc.convert
                                ( descriptor, ncsa.services.ensemble.EnsembleDescriptor.class );
                        logger.debug( "submitting: " + d + " , " + instanceId );
                        ncsa.services.ensemble.EnsembleHandle h = impl.submit( d, instanceId );
                        logger.debug( "got: " + h );
                        return ( ncsa.services.ensemble.service.types.EnsembleHandle )ibc.convert
                                ( h, ncsa.services.ensemble.service.types.EnsembleHandle.class );
                } catch ( Throwable t ) {
                        throw new RuntimeException( t );
                }
        }

        public void updateNodeStatus( int in0, String in1 )
        {
                impl.updateNodeStatus( new Integer( in0 ), in1 );
        }

}
```

In the case of `submit`, the converter is used to translate the wire object `ncsa.services.ensemble.service.types.EnsembleDescriptor` into its local equivalent `ncsa.services.ensemble.EnsembleDescriptor`, and then to translate the local return value `ncsa.services.ensemble.EnsembleHandle` into its wire equivalent `ncsa.services.ensemble.service.types.EnsembleHandle`.

Note how no conversion is necessary in the case of the second method, since all types belong to `java.lang`.

Security Context Information

As mentioned above, the base class stores the proxy in the in-memory repository. This happens via the call to `extractAuthorizationInfo`, which does the following:

1. retrieves the credential from the message context;
2. retrieves the user's DN from the message context;
3. uses the user's DN to find the user name on the host via the standard grid-map file (which should be in /etc/grid-security);
4. stores the credential by mapping it to the user name;

5. returns the user name.

NOTE: `ncsa.services.endpoint.GSIDelegatingEndpoint` depends on the Spring framework mechanism; it accesses a Spring configuration file to get the static instance of the in-memory repository; we have assumed that the implementation classes will also access this static instance via dependency injection. Hence, it will be necessary to provide a Spring configuration file. (Refer to Spring documentation.) The file should contain this entry:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org./dtd/spring-beans.dtd">

<beans>
...
   <bean name="authenticationRepository" class="ncsa.services.endpoint.security.repositories.
MemoryAuthenticationRepository"/>
...
</beans>
```

Should you wish to replace this storage mechanism with something else, then the new class should be substituted for the `MemoryAuthenticationRepository` in the bean definition; this class must implement `ncsa.services.endpoint.security.IAuthenticationRepository`. (Notice also how the endpoint example above uses the Spring context to retrieve its implementation.)

Client Adapter.

This class should extend the `ncsa.services.client.GSIServiceClientAdapter` class and implement the local interface. For each method of the local interface, it should convert any special types corresponding to those in the wstypes plugin to those remote variants, and then use those to call the superclass `invoke` method. Any return objects or specialized exceptions need to be converted back to their local equivalents.

As illustration, we give the "ensemble service" client counterpart to the endpoint above:

"Ensemble Service Client Adapter"

```
package ncsa.services.ensemble.service.client;

import java.rmi.RemoteException;

import org.osgi.framework.Bundle;

import ncsa.services.client.GSIServiceClientAdapter;
import ncsa.services.ensemble.osgi.ClientPlugin;
import ncsa.tools.common.util.bean.ImplicitBeanConverter;

public class EnsembleBrokerClientAdapter extends GSIServiceClientAdapter
        implements ncsa.services.ensemble.EnsembleBroker
{
        public EnsembleBrokerClientAdapter( String endpointURL ) throws Throwable
        {
                super( endpointURL );
        }

        public ncsa.services.ensemble.EnsembleHandle
                submit( ncsa.services.ensemble.EnsembleDescriptor d, String instanceId )
        {
                   try {
                        ncsa.services.ensemble.service.types.EnsembleDescriptor descriptor
                                = ( ncsa.services.ensemble.service.types.EnsembleDescriptor )
                                new ImplicitBeanConverter().convert( d, ncsa.services.ensemble.service.types.
EnsembleDescriptor.class );
                        Object handle = invoke( "submit", new Object[]{ descriptor, instanceId } );
                        return ( ncsa.services.ensemble.EnsembleHandle )
                                new ImplicitBeanConverter().convert(        handle, ncsa.services.ensemble.
EnsembleHandle.class );
                   } catch ( Throwable e ) {
                           logger.error( "submit", e );
                           throw new RuntimeException( e );
                   }
        }

        public void updateNodeStatus( Integer sessionId, String status )
        {
                   try {
                           if ( sessionId == null ) throw new RuntimeException( "no sessionId" );
                           invoke( "updateNodeStatus", new Object[]{ sessionId, status } );
                   } catch ( RemoteException e ) {
                           logger.error( "updateNodeStatus", e );
                           throw new RuntimeException( e );
                   }
        }

        // GSI SERVICE CLIENT ADAPTER

        protected String getWSDLName()
        {
                return "ensemble.wsdl";
        }

        protected Bundle getBundle()
        {
                ClientPlugin cp = ClientPlugin.getDefault();
                if ( cp != null ) return cp.getBundle();
                return null;
        }
}
```

The procedure here is symmetrical to that in the endpoint code. There are also two protected methods which need to be implemented (along with a string constructor taking the endpoint url and calling the superclass constructor with it):

1. `getWSDLName` is the name of the wsdl resource placed in the client .jar. This is used in order to initialize the base class (via a special WSDL helper class).
2. `getBundle` should return the OSGI (Eclipse) bundle for the client (this is necessary for running the client as a plugin).

---

(5) Build the service .war.

Inside of Eclipse, select "Run", choosing the `ncsa.services.build.ServiceBuilder` from the `ncsa.services.build` plugin you have installed. Set the command-line arguments to:

- war <path to properties file>

and run it.

This will build the .war file and place it in the location you indicated in the properties file.

NOTE

The JGlobus dependencies listed below should be placed in ${CATALINA_HOME}/common/lib, not in the individual service wars:

LIBRARY	INCLUDED IN PLUGIN
cryptix-asn-1.0.jar	org.globus.jglobus
cryptix-random-1.0.jar	org.globus.jglobus
cryptix-32-1.0.jar	org.globus.jglobus
jce-jdk13-120.jar	org.globus.jglobus
jgss-1.0.jar	org.globus.jglobus
puretls-1.0.jar	org.globus.jglobus
cog-jglobus-1.2.1	org.globus.jglobus

---

(6) Initialize the database (optional).

Most of the services we work with maintain state via a database. We have generally adopted the Spring/Hibernate mechanism for persisting Java objects to them.

As a matter of convenience, the service builder also contains a routine for initializing these Hibernate-based data stores.

The final properties in the file template given above are the necessary configuration settings for running this part of the builder. The first one points to the Hibernate mapping files defining the store (`.hbm.xml`). A properties file used in the configuration is also indicated, along with various task settings.

Once you have these in place, the procedure is similiar to the other two invocations of the service builder:

Inside of Eclipse, select "Run", choosing the `ncsa.services.build.ServiceBuilder` from the `ncsa.services.build` plugin you have installed. Set the command-line arguments to:

- db-init <path to properties file>

and run it.

This will initialize the database from the mappings. (Note that the initial creation of the database and the establishment of permissions for it is not included in this process.)

For more information, see Hibernate Documentation.

The following command:

- all <path to properties file>

will run all three phases (source, war, db-init) in that order.

Note that all of these commands accept multiple properties file arguments.