Web Services Framework
Windchill ® 9.1 M050
7 April 2010

Windchill and Info*Engine offer a tool set that facilitates deployment of the same Info*Engine-based web service that leverages JAX-WS (Java API for XML Web Services) in order to handle the SOAP/XML and security-related web service details. This tool set is available in addition to the Info*Engine-based web services that are supplied by the SimpleTaskDispatcher servlet. The web service implementation, which is based on Info*Engine tasks and task delegation, remains mostly consistent. However, the XML representation of the corresponding SOAP interactions will be different based upon the manner in which JAX-WS uses JAXB (Java Architecture for XML Binding) to bind Java objects and Info*Engine data structures to XML. For backward compatibility, access to the same services is available using the SimpleTaskDispatcher.

Aside from leveraging JAX-WS for the SOAP-related details, a major advantage to using this new framework is the addition of configurable and efficient security. Legacy Info*Engine web services rely solely upon web server authentication for security. This means that SOAP clients must have complete access to user credentials, and must supply those credentials to the web server per its authentication requirements (typically HTTP Basic Authentication). However, this presents a fairly significant roadblock to developing applications that implement SSO (Single Sign On), which is when a separate application or application server has independently authenticated a user and a trust relationship exists between a client and Windchill. Relying solely upon web server authentication means that an SSO application or application server must have access to the user's password to reauthenticate that user within an Info*Engine web service.

A downside of leveraging more advanced security policies is the complexity of the mechanisms involved. The web service client must be capable of complying with the configured security policy, which often involves encryption and a complex XML representation of the security information embedded within SOAP requests. If the client in question is Java, then the tooling supplied with Info*Engine hides these complexities so that you can write your client without complex coding for security purposes. If you use a separate third party SOAP framework to connect to Windchill web services, then it is that client's responsibility to comply with the security policy chosen for a given web service.

With web service security support in X-20, you can establish a trust relationship between an application (such as a portal server) and an Info*Engine web service by exchanging public key information, requiring that SOAP message exchanges be properly signed and encrypted, and requiring that they contain a valid username token that the web service should trust. As long as the incoming SOAP message meets the security requirements and can be decrypted using a public key found in the server's truststore, the web service then executes a request that has been authenticated with the username. Similarly, depending on the requirements of the security policy in use, the client may be required to have access to the server's public key to decrypt responses.

Each web service endpoint is represented by an instance of a Java servlet. As a result, a JAX-WS based Info*Engine web service can be deployed multiple times if each time the web service is secured by a different security policy and uses a unique servlet name for each deployment.

Optionally, given additional manual configurations, it is possible to place a JAX-WS based Info*Engine web service behind web server authentication and forgo the cryptography-related requirements of selecting another security policy.

Tooling Overview

This is a basic overview of how the Info*Engine tooling for web services is organized. After reviewing this section, you should be able to locate tools and understand their purpose. The actual use of the tools is discussed in greater depth in later sections.

The supplied tooling is enabled using Apache Ant, which is installed in *<Windchill>*/ant (where <*Windchill*> is the Info*Engine installation directory). The tooling is intended to be run from a windchill shell, in which case the ant command is already in your path.

The directory that contains the Apache Ant build tools for deploying a web service is located in *<Windchill>*/bin/adminTools/WebServices and contains the following:

- build.xml: The primary Apache Ant build script used to deploy web services and associate those services with security policies.
- security.properties: A properties file containing the security-related configuration information used when deploying a web service. This file contains the security policy to use, the server side truststore and keystore configurations, and any additional policy-specific configuration (if necessary) that is to be used at deployment time.
- new-project.xml: An Apache Ant build script used to generate new Apache Ant projects, which are then used to build or deploy new web services or web service clients.
- xslt/: A subdirectory containing XSL documents used during the web service deployment by build.xml.
- client/: A subdirectory containing client-side security configuration properties.
- client/security.properties: The client-side security configuration (truststore and keystore configuration information).

  Note

  The paths specified to the truststore and keystore must be specific to the corresponding location where those files reside on the client machine. These paths are included in security deployment descriptors and are not configurable.

- `common/`: A subdirectory containing the common Apache Ant framework for web services projects.

Of the previously mentioned resources, you should edit the `security.properties` files to specify your security configuration for client and server. To identify the changes needed, you can use the example provided in the *<Windchill>*`/prog_examples/jws` directory, which contains the root of the web service example projects. Each project was generated and developed using the `new-project.xml` script. In addition, this directory contains the `jws-stores.xml` Apache Ant build script that can be used to generate client/server keystore and truststore pairs.

Prerequisites

Windchill releases Java libraries that are duplicated by Java 1.6, but are newer than those packaged with Java. This presents no runtime issues, but before you can use the web service tooling to generate and deploy new services or clients, you must update your Java installation to include the necessary updated classes.

Copy the following JAR file:

<Windchill>`/srclib/webservices-api.jar`

to

${JAVA_HOME}`/jre/lib/endorsed`

Where *${JAVA_HOME}* is the installation location of the Java used by your Windchill installation. If your installation of Java does not already have the `endorsed` directory then you must create it.

Understanding the Security Requirements

This section explains the security policies available to you, and how to configure the security properties, truststores, and keystores.

Security Policies

Prior to deploying a web service, you must decide how you want the service to be secured. The following are all supported security policies.

SAML Sender Vouchers with Certificates

This mechanism protects messages with mutual certificates for integrity and confidentiality, as well as with a Sender Vouchers SAML token for authorization.

When using this security policy, the SOAP client provides an SAML Subject (within an SAML Assertion) containing the name identifier, which signifies the username the server should run the request as. In this case the message payload is signed and encrypted, because the client and server have established a trust relationship using the contents of their respective keystores and truststores.

In this situation the server only verifies that the supplied username is valid prior to handling the request.

This policy requires that the client have both a keystore and truststore corresponding to the server's. In addition, Java clients require a callback handler that provides the SAML Assertion containing the username that the server should trust.

Note

It is extremely important to note that when using this mechanism the client is "vouching" for the username associated with the SOAP request. The SOAP client is, in essence, impersonating a given user or users without being required to supply corresponding passwords. This mechanism is really only appropriate for an SSO (Single Sign On) type scenario, in which you are certain that the client application has appropriately authenticated the users, and you are willing to extend trust to that client using certificate exchange in the keystore or truststore configuration.

Username Authentication with Symmetric Key

The Username Authentication with Symmetric Key mechanism protects SOAP interactions for both integrity and confidentiality. Symmetric key cryptography relies on a single, shared secret key that is used to both sign and encrypt a message. Symmetric keys are usually faster than public key cryptography.

For this mechanism the client does not possess any certificate or key of its own, but instead sends its username and password for authentication. The client and the server share a secret key. This shared, symmetric key is generated at runtime and encrypted using the service's certificate. The client must specify the alias in the truststore by identifying the server's certificate alias.

In this situation, the server authenticates the given username using the corresponding supplied password.

This policy requires that the client have a truststore corresponding to the server's configuration. In addition, Java clients require callback handlers to provide the username and password to send along with the request for the server to validate.

Security Properties

The Apache Ant build tooling requires a security configuration for the server that specifies the security policy to use, as well as the truststore and keystore configurations. You must edit the `security.properties` file to specify this information before deploying a web service.

The server `security.properties` file is found at:

`<Windchill>/bin/adminTools/WebServices/security.properties`

It contains comments describing its contents.

For writing or testing Java clients, there is a parallel client properties file found at:

`<Windchill>/bin/adminTools/WebServices/client/security.properties`

It contains similar comments describing its contents.

Note

Both files, as released, contain a configuration that points to corresponding client and server truststore and keystore pairs, which can be generated for use with the examples (see the "Truststores and Keystores" section). While it is possible to use these generated truststores and keystores to secure web services, it is recommended that you acquire your own certificates and use them to populate truststores and keystores for your applications.

Truststores and Keystores

Unless you are manually configuring your web service behind web server authentication, you are required to secure your web service using X509 v3 certificates. Note that the security mechanisms used require the `SubjectKeyIdentifier` extension. The keytool utility released with Java does not generate this extension.

The Apache Ant build script located at `<Windchill>/prog_examples/jws/j-ws-stores.xml` is able to generate client or server keystore and truststore pairs for use by Info*Engine web service examples. You can also use this build script for testing

and development purposes. This script uses OpenSSL to generate client and server certificates, and then uses the Java keytool utility to import these certificates into client or server keystore and truststore files.

You can use the `jws-stores.xml` build script as an illustration on how to use your own certificate to create truststores and keystores for your web services and clients. This script only generates a single server and client certificate, and then imports those certificates into their corresponding keystores before generating the truststores for the client and server.

To generate these files, run the following script from a windchill shell:

```
% cd <Windchill>/prog_examples/jws
% ant -f jws-stores.xml
```

While the script is running, you are prompted several times for user input. You can choose to either accept the defaults (presented surrounded by brackets like `[ws-server]`) by simply pressing **Enter** when prompted, or supply your own input. If you choose the default input, then the released configured `security.properties` files should contain the proper configuration. If you decide to supply other input when running the script, then you need to update the corresponding `security.properties` configuration accordingly. When the script has finished running, there are four files in the *<Windchill>*`/prog_examples/jws/stores` directory:

- `server-keystore.jks`
- `server-truststore.jks`
- `client-keystore.jks`
- `client-truststore.jks`

Writing an Info*Engine-based Web Service

The methods for writing an Info*Engine-based web service have not changed since web services were first supported with Info*Engine release 7.0. The only area in which JAX-WS-based web services are now different is surrounding the support for SOAP with attachments, as discussed in Using SOAP Attachments. In addition to writing an Info*Engine-based web service, you can choose to write a Java-based web service. This is illustrated with released examples found in *<Windchill>*`/prog_examples/jws`, which is discussed in Truststores and Keystores.

An Info*Engine web service consists of a set of Info*Engine tasks, each representing a web service method that is associated with a type identifier. In the case of a web service, the type identifier is used simply as a way to logically group a set of Info*Engine tasks together to form a web service, and can be thought of as analogous to a Java class that exposes only static methods (each method being the Info*Engine task supporting the corresponding web service operation).

At runtime, Info*Engine requires that the type identifier and corresponding Info*Engine tasks be registered with the Info*Engine configuration in the LDAP. You can do this manually, but since the process can be involved and is potentially error prone, Info*Engine is released with Apache Ant tooling that packages and installs the Info*Engine tasks backing your web service. These utilities are integrated with the Apache Ant framework.

This section walks you through writing a simple Info*Engine task-based web service and client from scratch. This includes the following steps:

1. Create a simple project (which will be a simple directory structure) to hold your web service tasks and client source.
2. Write a few very basic tasks.
3. Secure and deploy your JAX-WS web service, which consists of deploying your Info*Engine tasks and corresponding servlet.

Before You Begin

The following are prerequisites to following the steps in this section:

- All commands are assumed to be run from a windchill shell.
- It is assumed that you have run the Apache Ant script documented earlier in Truststores and Keystores.
- The example commands assume a Unix environment. If you are in a Windows environment then you must alter the operating system-specific commands to suit your environment (for example, `cd %Windchill%` rather than `cd <Winchill>`, and using \ rather than / in paths).

Create a Project

Released with the web service framework are tools to simplify authoring new web services and clients.

This project example creates a simple web service that performs basic math operations. This service is for illustration purposes only and is not the type of thing you would typically use Info*Engine to accomplish.

Create your project using the *<Windchill>*/bin/adminTools/WebServices-/new-project.xml Apache Ant build script. This script can be used to create a client project, a service project, or both:

```
% cd <Windchill>
% mkdir prog_examples/jws/MyProject
% ant -Dproject.dir=<Windchill>/prog_examples/jws/MyProject
-Dservlet.name=MathService -Dsecurity.policy=userNameAuthSymmetricKeys
-Dservice.type.id=org.myorg.MathService -Dmain.class=org.myorg.MathClient
-f bin/adminTools/WebServices/new-project.xml create
```

Note

The project.dir property is an absolute path so that the project is created in the appropriate location. To see what input options are supported by the new-project.xml build script, you should run it with no arguments. For example:

```
ant -f bin/adminTools/WebServices/new-project.xml
```

This creates the following structure within the prog_examples/jws/MyProject directory:

- src/ : The base directory for your web service (server side).
- src/build.xml: The build script used to build and deploy your web service.
- src/tasks/org/myorg/MathService: A subdirectory where you can create your Info*Engine tasks. This is only a suggestion; you can create another directory hierarchy that suits your needs if you would like. The tools package your service using the type identifier you specified on the service.type.id property when creating the project.
- src_client/: The base directory for your web service client.
- src_client/build.xml: The build script used to build your web service client.
- src_client/org/myorg/MathClient.java: The beginnings of your web service client.

Review the contents of the two generated build.xml scripts to gain some understanding on how they work.

To further familiarize yourself with the framework from within the `src` and `src_client` directories, run the following `ant` commands:

```
% ant -p
% ant usage
```

The first command shows you the public targets for the associated build script with their descriptions. The second command gives you a detailed description of how you can tailor your build script if you find it necessary to do so. For most basic projects this should not be necessary.

Write the Info*Engine Tasks for Your Web Service

This step creates four simple tasks (`Add.xml`, `Divide.xml`, `Multiply.xml`, `Subtract.xml`) in the following locations with the given source. Note that these tasks are very simple and primarily contain documentation.

Add.xml

Source: *<Windchill>*/prog_examples/jws/MyProject/src/tasks/org/myorg/MathService/Add.xml

```
<%@page language="java"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>
<%@taglib uri="/com/infoengine/tlds/iejstl.tld" prefix="c"%>

<!--com.infoengine.soap.rpc.def
Generates the sum of two integers.
<p>Supply two integers, a and b. You can subtract
integers using the {@link Subtract} method.

@param int a The first integer.
@param int b The second integer.
@return int ${output[0]result[0]} The sum of a and b.
@see Multiply
@see Divide
@see Subtract
-->
<ie:webject name="Create-Group" type="GRP">
  <ie:param name="GROUP_OUT" data="output"/>
</ie:webject>
<!-- just so non-web service clients could call this task
     coerce a to an int, in case it isn't already -->
<c:set var="A" value="${0+@FORM[0]a[0]}" />
<c:set var="${output[0]result}" value="${A+@FORM[0]b[0]}" />
```

Divide.xml

Source: *<Windchill>*/prog_examples/jws/MyProject/src/tasks/org/myorg/MathService/Divide.xml

```
<%@page language="java"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>
<%@taglib uri="/com/infoengine/tlds/iejstl.tld" prefix="c"%>

<!--com.infoengine.soap.rpc.def
Generates the result of dividing two integers.
<p>Supply two integers, a and b. You can multiply
integers using the {@link Multiply} method.

@param int a The first integer.
@param int b The second integer.
@return float ${output[0]result[0]} The sum of a and b.
@see Multiply
@see Divide
@see Subtract
-->
<ie:webject name="Create-Group" type="GRP">
  <ie:param name="GROUP_OUT" data="output"/>
</ie:webject>
<!-- just so non-web service clients could call this task
     coerce a to an float, in case it isn't already -->
<c:set var="A" value="${0.0+@FORM[0]a[0]}" />
<c:set var="${output[0]result}" value="${A/@FORM[0]b[0]}" />
```

Multiply.xml

Source: *<Windchill>*/prog_examples/jws/MyProject/src/tasks/org/myorg/MathService/Multiply.xml

```
<%@page language="java"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>
<%@taglib uri="/com/infoengine/tlds/iejstl.tld" prefix="c"%>

<!--com.infoengine.soap.rpc.def
Generates the product of two integers.
<p>Supply two integers, a and b. You can divide
integers using the {@link Divide} method.

@param int a The first integer.
@param int b The second integer.
@return int ${output[0]result[0]} The sum of a and b.
@see Multiply
@see Divide
```

```
@see Subtract
-->
<ie:webject name="Create-Group" type="GRP">
  <ie:param name="GROUP_OUT" data="output"/>
</ie:webject>
<!-- just so non-web service clients could call this task
     coerce a to an int, in case it isn't already -->
<c:set var="A" value="${0+@FORM[0]a[0]}" />
<c:set var="${output[0]result}" value="${A*@FORM[0]b[0]}" />
```

Subtract.xml

Source: *<Windchill>*/prog_examples/jws/MyProject/src/tasks/org/myorg/MathService/Subtract.xml

```
<%@page language="java"%>
<%@ taglib uri="http://www.ptc.com/infoengine/taglib/core" prefix="ie"%>
<%@taglib uri="/com/infoengine/tlds/iejstl.tld" prefix="c"%>

<!--com.infoengine.soap.rpc.def
Generates the difference of two integers.
<p>Supply two integers, a and b. You can add
integers using the {@link Add} method.

@param int a The first integer.
@param int b The second integer.
@return int ${output[0]result[0]} The sum of a and b.
@see Multiply
@see Divide
@see Subtract
-->
<ie:webject name="Create-Group" type="GRP">
  <ie:param name="GROUP_OUT" data="outputd"/>
</ie:webject>
<!-- just so non-web service clients could call this task
     coerce a to an int, in case it isn't already -->
<c:set var="A" value="${0+@FORM[0]a[0]}" />
<c:set var="${output[0]result}" value="${A-@FORM[0]b[0]}" />
```

Deploy Your Web Service

When we created the project we supplied a parameter named "`security.policy`" and explicitly passed a value of "`userNameAuthSymmetricKeys`" for that property. This example requires that you provide both a username and password for your web service client.

To generate and deploy your web service, run the following commands:

```
% cd <Windchill>/prog_examples/MyProject/src
% ant
```

Windchill does not need to be running to run this script. At this point start (or restart) Windchill to finish deployment.

Running this Apache Ant script with no arguments packages, installs, and deploys your web service. The following is a brief description of what the script performs:

- The service is packaged, compiled, and installed into your Windchill installation. In the case of an Info*Engine task-based service, your tasks are packaged in to a PTCTAR file and installed. In this example the resulting PTCTAR file is:

  `<Windchill>/prog_examples/jws/MyProject/dist_server/mathservice.ptctar`

  If your web service requires Java objects, then you can simply create the package hierarchy and source for those objects within the `src` directory. The Apache Ant script automatically compiles and package those classes for you. Classes associated with your web service are packaged in `<Windchill>/codebase/WEB-INF/lib` within a JAR file that reflects the servlet name of your service.

- The `<Windchill>/bin/adminTools/WebServices/build.xml` script is then run to generate and compile the server-side artifacts, as well as secure and deploy the web service. This build script can also be run on its own depending on your needs. This is discussed further in the section Using the Web Service's Deployment Build Script.

  Note

  If you use this Apache Ant framework you should not need to run this build script manually. The Apache Ant framework can also be used to deploy an existing legacy web service that is already installed on your system. In this case the `src` directory contains only the `build.xml` script and no task source.

Now that your service is deployed and Windchill is running, you can review your web service WSDL by visiting a URL similar to the following:

`http://<host>/Windchill/servlet/MathService?wsdl`

If you are writing a Java client, as described in the section Writing a Java-based Web Service Client, then you do not need to know how to access the WSDL. However, you must access it if you plan to integrate your service with another web service client.

Undeploy Your Web Service

If you want to undeploy your web service from your installation, simply run the `undeploy` target of your project's `src/build.xml` script. For example:

```
% cd <Windchill>/prog_examples/MyProject/src
% ant undeploy
```

This does following:

- Removes the servlet configuration from `web.xml`
- Removes the web service configuration from `sun-jaxws.xml`
- Removes the security policy configuration file
- Removes the JAR file supporting your web service
- Uninstalls the Info*Engine tasks (if your service is based on Info*Engine tasks)

Info*Engine Web Service Parameters and Task Documentation

You should familiarize yourself with the process of properly documenting Info*Engine tasks. Visit the online documentation for your new web service by opening the documentation with a web browser to:

`http://host/Windchill/infoengine/jsp/tools/doc/index.jsp`

In the web browser from the **Repository Types** frame select **com.ptc.windchill** and from the **Classes** frame select your service type identifier **org.myorg.MathService**. This documentation is based on the contents of the tasks you deployed. To learn more about how to write task documentation visit:

`http://host/Windchill/infoengine/jsp/tools/doc/howto.html`

For more information on defining parameters and return types for your web service tasks, see the *Info*Engine User's Guide*.

Writing a Java-based Web Service Client

The project directory (`src_client`) and the example Java source file `src_client/org/myorg/MathClient.java` were created in the step Create a Project. The source (`MathClient.java`) is dependent upon source artifacts, which are generated from your deployed web service. Before actually being able to complete

your client source you must generate and compile these artifacts. As generated, the `MathClient.java` Java source compiles, but does nothing until you use it to invoke a web service method.

In order to get these artifacts you must compile your empty client by entering the following commands:

```
% cd <Windchill>/prog_examples/jws/MyProject/src_client
% ant
```

The Apache Ant script also generated an executable JAR file of your application, which you can find here:

`<Windchill>/prog_examples/jws/MyProject/dist_client/MathServiceClient.jar`

However, since your main class does nothing, then executing the JAR also does nothing. Running the previous `ant` command processes the WSDL for your web service and generates Java source artifacts to the following:

`<Windchill>/prog_examples/jws/MyProject/gensrc_client`

This creates a lot of Java source code within the com.ptc.jws.service.org.myorg.mathservice package. To complete your client, you must now edit your client source code:

1. In a text editor, open the following file:

   `<Windchill>/prog_examples/jws/MyProject/src_client/org/myorg/MathClient.java`

   The source should look something like:

```
package org.myorg;

// import the classes generated when compiling your client
//import com.ptc.jws.service.?.*;
import com.ptc.jws.client.handler.*;

public class MathClient
{
    public static void main ( String [] args ) throws java.lang.Exception
    {
        // depending on your security requirements
            you may need to specify credentials up
        // front, or on the command-line
        //SamlCallbackHandler.setUsername ( trustedUser );
        // TODO implement your client
        /*
        MathServiceImplService service = new MathServiceImplService();
        MathServiceImpl port = service.getMathServiceImplPort ();
```

```
        //invoke an action
        //port.methodName ( <arguments> );
        */
    }
}
```

2. Update the source to import the server side web service artifacts and invoke a method as follows:

```
package org.myorg;

// import the classes generated when compiling your client
import com.ptc.jws.service.org.myorg.mathservice.*;

public class MathClient
{
    public static void main ( String [] args ) throws java.lang.Exception
    {
        int a = args.length > 0 ? Integer.parseInt ( args[0] ) : 0;
        int b = args.length > 1 ? Integer.parseInt ( args[1] ) : 0;

        MathServiceImplService service = new MathServiceImplService();
        MathServiceImpl port = service.getMathServiceImplPort ();
        System.out.printf ( "a+b=%d\n", port.add ( a, b ) );
        System.out.printf ( "a-b=%d\n", port.subtract ( a, b ) );
        System.out.printf ( "a*b=%d\n", port.multiply ( a, b ) );
        System.out.printf ( "a/b=%f\n", port.divide ( a, b ) );
    }
}
```

3. Now recompile your client and update your executable JAR to include the updated classes:

```
% ant compile
% ant dist
```

4. And now run your new web service client from the command line:

```
% java -jar ../dist_client/MathServiceClient.jar 10 20
Username:<password>
Password:<username>
a+b=30
a-b=-10
a*b=200
a/b=0.500000
```

Where <*username*> is the username and <*password*> is the password you have supplied. When running your client, you are prompted on the command line for credentials.

Client Callback Handlers

When creating the web service project, a property (`-Dsecurity.policy=usernameAuthSymmetricKeys`) was supplied. This instructed the framework to explicitly secure the web service with the security policy, which is described in the section Username Authentication with Symmetric Key. If this property had not been explicitly provided, then the security policy would have defaulted to the value of the security.policy property as configured in *<Windchill>*`/bin/adminTools/WebServices/security.properties`.

When generating the client portion of the project, this policy causes a default client side callback handler configuration to be used. This is specified in the handler.config property of the `src_client/build.xml` build script. To view the format of the handler.config property, run the `ant usage` command from with in the `src_client` directory. The com.ptc.jws.client.handler package also contains some basic callback handlers for your web service client to use for the supported security policies (these handler classes are automatically packaged for you when you build your client). You can choose to use these classes to provide security information for your web service clients, or you can choose to author your own by providing a class that implements javax.security.auth.callback.CallbackHandler and then providing that class as part of the handler.config property.

The default callback handler supplied for `usernameAuthSymmetricKeys` prompts for credentials on the command line. It can be configured to prompt the user for credentials using swing dialogs or to fetch the credentials from system properties. It only prompts for credentials once per thread, and reuses the credentials until you programatically clear them.

Portable Web Service Clients

There are certain issues that must be addressed to allow a web service client to run on a client machine properly or interact with multiple identical services hosts on separate Windchill instances. Security policy configuration is essentially hard-coded at compile time, requiring a new client to be recompiled for each client host, or a single client can be manually reconfigured. Similarly, the address of the web service is fixed at compile

time, and therefore writing a client that interacts with separate service instances requires compiling the client for each address. Alternatively, you can explicitly write the client to allow it communicate with a duplicate service.

Using Security Policies

When securing a web service within one of the supported security policies (other than web server authentication), a Java client needs access to its own keystore and truststore. Paths to these files (keystore and truststore) are configured for use by the build framework in:

`<Windchill>/bin/adminTools/WebServices/client/security.properties`

The paths to the keystore and truststore are absolute, and must be compiled into deployment descriptors packaged with your client JAR file in its META-INF directory. If you compile a web service client from the same host where Windchill is running with the standard configuration, then these files are available to your client, but only on that host. Before redistributing your web service client to another host, you must update the `client/security.properties` file (or override its contents in your client build script) to contain paths to the location where these files exist on the new client host, and then rebuild your client. If this is not done then your client will not run on the remote host.

You also have the option to manually reconfigure a client JAR file by editing its `/META-INF/<ServletName>_Wsdl-wsit.xml` file and thereby updating its configuration. You can update your project to make use of different security properties that are specific to your project by simply including those properties in your build script.

For example:

```
% cd <Windchill>/prog_examples/jws/MathService/src_client
% cp <Windchill>/bin/adminTools/WebServices/client/security.properties
```

Update the local copy of `security.properties` as necessary for the individual client and then include this properties file (before any XML entity references) within the `src_client/build.xml` file:

```
<property file="security.properties"/>
```

This ensures that these security properties take precedence over the common properties found in `<Windchill>/bin/adminTools/WebServices/client/security.properties`.

Dynamically Specifying a Web Service URL

When creating a Java-based web service client, the http connection information is determined from your Windchill configuration. When building the JAR for your web service client, the framework stores a local copy of the WSDL to your web service and references that WSDL from a catalog. This allows your client to avoid having to access the WSDL over the network at runtime. The URL used to connect to your web service is essentially hard-coded into your Java client through this WSDL. You should therefore not need to specify this information to run your client.

If, however, you have multiple Windchill installations and want to write a client that can communicate with a common web service deployed on each installation, or you generate your web service client against a development Windchill installation and want to run it against another installation, then you can manually specify the URL you want your client to connect to in Java as follows:

```
import javax.xml.ws.BindingProvider;
...
        String url = "http://host/Windchill/servlet/MathService";
        MathServiceImpl port = service.getMathServiceImplPort ();
        ((BindingProvider)port).getRequestContext().put
         (BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
...
```

The URL specified must be an identical web service to the one you generated and developed your client against.

Web Server Authenticated Services and Clients

Using the standard security policies, such as SAML Sender Vouches or Username Authentication with Symmetric Keys, means that your web service servlet is deployed in a manner that is unprotected by web server authentication. In these cases it is the responsibility of the web service, not the web server, to enforce authentication.

You can also deploy your servlet behind web server authentication. In order to do this, use the `webServerAuthenticated` security policy when deploying, as documented in:

<Windchill>`/bin/adminTools/WebServices/security.properties`

However, when using this security policy the tooling does not automatically update your web server configuration to require authentication for your servlet. You must therefore update your web server configuration independently.

For example, in the case of Apache you can enter the following commands:

```
% cd ${APACHE_HOME}
% ant -DappName=Windchill -Dresource=servlet/MyServlet -f
webAppConfig.xml addAuthResource
```

Then restart Apache.

Deploying a web service using this method means that your client must supply credentials without being able to use the released callback handlers. The following is an example of how you can provide credentials for your client:

```
import javax.xml.ws.BindingProvider;
...
MathServiceImpl port = service.getMathServiceImplPort ();
((BindingProvider)port).getRequestContext ().put
                ( BindingProvider.USERNAME_PROPERTY, "demo" );
((BindingProvider)port).getRequestContext ().put
                 ( BindingProvider.PASSWORD_PROPERTY, "demo" );
...
```

Writing a Java-based Web Service

You also have the option of manually implementing your service directly in Java. In this case, you can generate a web service project using the following Apache Ant script:

<Windchill>/bin/adminTools/WebServices/new-project.xml

with a service.type of `java`. Then, in place of specifying the service.type.id property, instead specify the class to implement your Java using the service.class property.

For example:

```
% cd <Windchill>
% mkdir prog_examples/jws/MyJavaProject
% ant -Dproject.dir=<Windchill>/prog_examples/jws/MyJavaProject
-Dservlet.name=MyJavaService -Dsecurity.policy=userNameAuthSymmetricKeys
-Dmain.class=org.myorg.MyJavaServiceClient
-Dservice.type=java -Dservice.class=org.myorg.MyJavaService -f
bin/adminTools/WebServices/new-project.xml create
```

Creating the new project generates a base class for your new web service, which can be found at:

<Windchill>/prog_examples/jws/MyJavaProject/src/org/myorg/MyJavaService.java

This contains source code that looks something like:

```
package org.myorg;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

import com.ptc.jws.servlet.JaxWsWebService;

@WebService()
public class MyJavaService extends JaxWsWebService
{
    @WebMethod(operationName="add")
    public int add ( int a, int b )
    {
      return a+b;
    }
}
```

You can also simply replace the example **add** method with your own web service method. You can also add other web service methods and any required supporting classes within the `src` directory of your project. Then compile, package, and deploy your new service as follows:

```
% cd <Windchill>/prog_examples/jws/MyJavaService/src/
% ant
```

This compiles your web service and packages it in a JAR file based on your servlet name within *<Windchill>*`/codebase/WEB-INF/lib`. This also deploys the service, which is secured using the security policy you specified when creating your project. Then you can implement your web service client in the same manner as described in Writing a Java-based Web Service Client.

If you choose to generate the client portion of your web service project, the client source code example might not compile properly if uncommented (it is commented out in the main method of your web service client). The client-generated source code assumes the web service in question is Info*Engine-based, and therefore generates source code based upon that assumption. You must then compile your client, examine the client side source code artifacts in the `gensrc_client` directory, and then adjust your client source accordingly. Finally, you must then rerun the `ant compile` and `ant dist` commands to recompile and package your client.

Consult the standard Java `javadoc` for the javax.jws, javax.xml.ws, and related packages to learn more about writing a Java web service. You can download these at http://java.sun.com.

Note that in the generated source code example that your new web service class extends com.ptc.jws.servlet.JaxWsService. This is not a requirement, but it does provide you with some basic conveniences such as:

- Access to an implicitly instantiated log4j logger in the **$log** instance variable, the logger name of which is based on your web service class name.

  You can log messages by simply invoking methods on the logger such as:

  ```
  $log.debug ("my message");
  ```

  Consult the log4j documentation for more information on logging with log4j, found at http://logging.apache.org/log4j/.
- Access to the javax.xml.ws.WebServiceContext associated with a web service request in the **$wsc** instance variable.
- Access to the javax.xml.ws.handler.MessageContext associated with a web service request using the **getMessageContext()** method.
- Access to the associated javax.servlet.http.HttpServletRequest object using the **getServletRequest()** method.

Note

*Standard Java web services provide access to the **HttpServletRequest** object through the **MessageContext** object stored under the `MessageContext.SERVLET_REQUEST` key. You should NOT use this servlet request object, but rather use the one returned using the **getServletRequest()** method of your web service. Using the **HttpServletRequest** object would create a conflict, as the security layer that preprocesses web service requests extracts and validates the security credentials from the incoming SOAP requests based on your security policy, and then imposes those credentials using a javax.servlet.http.HttpServletRequestWrapper object. The **MessageContext** key SERVLET_REQUEST is an immutable property, and so the security layer must store its version of the **HttpServletRequest** object under another key within the **MessageContext**. This object has the validated username imposed in the **getRemoteUser()** and **getUserPrincipal()** methods of this **HttpServletRequest** object.*

Using the Web Service's Deployment Build Script

Released with Windchill and Info*Engine is an Apache Ant built script that can be used to both generate and deploy a web service. If you are manually writing your web service and are making use of the project support provided, then this script is automatically called for you, This means that you do not need to understand its functionality directly (aside from the `security.properties` configuration). If, however, you have a legacy web service already installed that you do not want to repackage and deploy (for example the com.ptc.windchill.ws Generic Web Services released with Windchill), then you can choose to run this script manually (the same can also be achieved using the supplied Apache Ant framework). Running this script without any input properties only displays its requirements, and at a minimum you must supply the servlet.name and type.id properties as input. On the command line you can also choose to explicitly override properties included from `security.properties` by adding `-D<propertyName>=<propertyValue>` arguments. An example is specifying a security.policy property value that differs from the default values you have configured.

You can also choose to redeploy the com.ptc.windchill.ws web service after it has been secured using SAML Sender Vouches. To redeploy it under a servlet named "GenericWebService" run the following commands:

```
% cd <Windchill>/bin/adminTools/WebServices
% ant -Dservlet.name=GenericWebService -Dtype.id=com.ptc.windchill.ws
-Dsecurity.policy=samlsv generate
```

As discussed earlier, you can choose to explicitly specify the security.policy property on the command line. If you simply want to use the security policy configuration in the `security.properties` file, then this property is not necessary.

Running the command line above carries out the following:

1. Generate a JAX-WS based web service that exposes the `com.ptc.windchill.ws` Info*Engine based web service.
2. Use the wsgen utility to generate the required Java source artifacts in support of the web service deployment.
3. Compile the previously generated source.
4. Based on the configured security policy, it applies the appropriate security policy information for use by the service. For most policies this results in a file in

<Windchill>/codebase/WEB-INF named "*<wsit-class>*.xml," which contains the security policy instructions for the web service layer.

5. JAR the compiled web service classes for use by Windchill.
6. Add servlet and servlet-mapping deployment information for your web service servlet to:

   <Windchill>/codebase/WEB-INF/web.xml
7. Create or update *<Windchill>*/codebase/WEB-INF/sun-jaxws.xml to deploy the web service, and associate the server side callback handler responsible for extracting security information from the SOAP requests corresponding to the configured security policy.

It is not necessary for Windchill to be running to deploy a web service; however, if Windchill is running, you should restart Windchill after running the script to deploy or redeploy a web service.

Using SOAP Attachments

JAX-WS web services support the ability to include binary data in a scalable fashion in your SOAP requests.

Just as with legacy web services, to upload binary data (available on your Info*Engine task's input stream) simply add a parameter to your task with a type of javax.activation.DataSource.

Unlike legacy Info*Engine web services, which used standard MIME SOAP attachments, JAX-WS web services use attachment references to make binary data appear as inline web service parameters. However, this approach no longer implicitly allows a SOAP client to include associated information like the filename or content type with the attachment using the **Content-Disposition** MIME header. As a result, Info*Engine web services now allow the filename to be specified as a separate parameter associated with the attachment. If an additional parameter is not used to specify an attachment filename, then the attachment has a default filename of "unknown."

You can also explicitly specify the content type of the attachment. This, however, only alters the content type that is exposed to Info*Engine components for the incoming data. If not specified, then the default value for the attachment content type is `application/octet-stream`.

For example:

```
@param string fileName The filename
@param javax.activation.DataSource file The file to stage {fileName:fileName}
 {contentType:image/jpeg}
```

In the previous example, your task receives a BLOB (attachment) and the filename of the attachment is specified by the accompanying **fileName** parameter value. The **Content-Type** of that attachment is exposed to the Info*Engine task as `image/jpeg`. It is important to note that this does nothing more than provide a value for the **Content-Type** of the incoming attachment. It does not force a web service client to comply with that **Content-Type** when supplying the data.

As with legacy web services, return an attachment from a task by simply specifying its return type as `java.io.InputStream`. In both the upload and download scenario, a Java web client is presented with a parameter or return value with a type of `javax.activation.DataHandler`.

In both the upload and download scenarios, an Info*Engine web service tunnels data between your task and the web service client to efficiently transfer it to and from Windchill. An efficient upload, however, is somewhat dependent upon how the client supplies the data. Small amounts of data can be internalized to memory, but when sending large amounts of data clients should be sure to chunk the MIME data. When using a Java-based client you must explicitly instruct the client to chunk data destined for the server or your client will likely die with an `OutOfMemoryError`. You should add the HTTP_CLIENT_STREAMING_CHUNK_SIZE property in your request context prior to uploading an attachment to Windchill.

For example:

```
import java.io.File;
import javax.activation.FileDataSource;
import javax.activation.DataHandler;
import javax.xml.ws.BindingProvider;
import com.sun.xml.ws.developer.JAXWSProperties;
...
((BindingProvider)port).getRequestContext ().put ( JAXWSProperties.
```

```
   HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8192 );
File f = new File ( path );
DataHandler file = new DataHandler ( new FileDataSource ( f ) );
port.upload ( f.getName(), file );
```

In addition, when performing downloads it is important to test the resulting **DataHandler** to see if it is an instance of **StreamingDataHandler**. When downloading large amounts of data, the Sun classes may store that data in a temporary file. If you do not explicitly close an instance of **StreamingDataHandler**, then this temporary file remains on your client machine occupying disk space.

For example:

```
import javax.activation.DataHandler;
import com.sun.xml.ws.developer.StreamingDataHandler;
...
DataHandler file = null;
try
{
  file = port.download ( ... );
  ..
  // transfer the data to another stream/file, etc.
  file.writeTo ( ... );
 }
finally
{
  if ( file instanceof StreamingDataHandler )
    ((StreamingDataHandler)file).close(); // remove any temporary data
}
```

Examples

There are several web service examples released with Windchill to illustrate the basics on how to write a web service and clients. You can find the examples in your installation at *<Windchill>*/prog_examples/jws.

For each, review the service source contained in the src directory and the client source contained in the src_client directory.

Each example was developed using the `new-project.xml` script, and are therefore organized in a manner consistent with this documentation. To exercise each example, enter the following commands:

```
% cd <Windchill>/prog_examples/jws/<ExampleDir>/src
% ant
```

This compiles, packages, and deploys the web service. At this point you should start Windchill, or restart it if it is already running. Then run the following commands:

```
% cd <Windchill>/prog_examples/jws/<ExampleDir>/src_client
% ant
```

This generates, compiles, and builds the distributable JAR for the client. Some example clients take optional parameters. Read the client source code within the `src_client` directory to see what parameters, if any, they may use.

To run the client from the `src_client` directory, enter the following:

```
% java -jar ../dist_client/<ClientJarFile>
```

Running `ant clobber` from both the `src` and `src_client` directories removes all generated and compiled files, which restores the example directory back to its original state.

After you are done reviewing a particular example web service, you can undeploy the service using the following:

```
% cd <Windchill>/prog_examples/jws/<ExampleDir>/src
% ant undeploy
```