



Cúram 8.1.3

Web Services Guide

Note

Before using this information and the product it supports, read the information in [Notices on page 59](#)

Edition

This edition applies to Cúram 8.1, 8.1.1, 8.1.2, and 8.1.3.

© Merative US L.P. 2012, 2024

Merative and the Merative Logo are trademarks of Merative US L.P. in the United States and other countries.

Contents

Note	iii
Edition	v
1 Integrating with External Applications through Web Services	9
1.1 Using Web Services.....	9
Overview of Web Services.....	9
Benefits of the Apache Axis2 Platform.....	10
Types of Web Services.....	10
Web Services Security.....	13
1.2 Building Outbound Web Service Connectors.....	14
Including the WSDL Files in Your Components File System.....	14
Adding the WSDL File Location to the Outbound Web Services File.....	14
Generating the Web Service Stubs.....	15
Creating a Client and Starting the Web Service.....	15
Client Stub Pool Configuration.....	17
1.3 Developing Inbound Web Services.....	18
Getting Started.....	18
Modeling and Implementing an Inbound Web Service.....	19
Building and Packaging Web Services.....	22
Providing Security Data for Web Services.....	24
Providing Web Service Customizations.....	24
1.4 Securing Web Services.....	30
Axis2 Security and Rampart.....	31
Custom SOAP Headers.....	31
Encrypting Custom SOAP Headers.....	34
Using Rampart With Web Services.....	36
Securing Web Service Network Traffic with HTTPS/SSL.....	45
Creating Keystore Files.....	46
1.5 Inbound Web Service Properties: ws_inbound.xml.....	46
1.6 Deployment Descriptor File: services.xml.....	48
1.7 Troubleshooting.....	51
Initial Server Validation and Troubleshooting.....	51
Using an External Client to Validate and Troubleshoot.....	52
Troubleshooting Axis2 errors.....	53
Avoiding Use of anyType.....	55
Axis2 Exceptions.....	55
1.8 Including the Axis2 Admin Application in Your Web Services WAR File.....	56
1.9 Including the Axis2 SOAP Monitor in Your Web Services WAR File.....	57

Notices	59
Privacy policy.....	60
Trademarks.....	60

1 Integrating with External Applications through Web Services

Use this information to develop and secure Cúram web services. You can make business logic available as web services.

This information covers all aspects of Cúram web service development including modeling, building, securing, deploying, and troubleshooting. Developers must be familiar with web service concepts and their underlying technologies, including modeling and developing in an Cúram environment.

Note: Cúram web services are based on Apache Axis2. As the basis for the latest generation of web service standards, Axis2 brings improved architecture, performance, and standards support to your web services.

Related concepts

Related information

axis.apache.org/axis2/java/core/index.html

1.1 Using Web Services

An overview of web services and how to use them to integrate web-based applications. The basics of Apache Axis2 web services are introduced and how Cúram web services correspond to this web service functionality.

Overview of Web Services

The term web services describes a standardized way of integrating web-based applications. Web services allow different applications from different sources to communicate with each other. Because all communication is in XML, web services are not tied to one operating system or programming language.

This application-to-application communication is performed by using XML to tag the data, using:

- SOAP (Simple Object Access Protocol: A lightweight XML-based messaging protocol) to transfer the data.
- WSDL (Web Services Description Language) to describe the services available.
- UDDI (Universal Description, Discovery and Integration) to list what services are available.

Web services can be considered in terms of the direction of flow, outbound/accessing and inbound/implementing, which are supported by the Cúram infrastructure for development and deployment as described below:

- **Outbound Web Service Connector**

An outbound web service connector allows the Cúram application to access external applications that have exposed a web service interface. The WSDL file that is used to describe

this interface is used by the web service connector functionality in Cúram to generate the appropriate client code (stubs) to connect to the web service. This means developers can focus on the business logic to handle the data for the web service. For information about how to develop outbound web service connectors, see [1.2 Building Outbound Web Service Connectors on page 14](#).

- **Inbound Web Service**

Some functionality within the Cúram application can be exposed to other internal or external applications within the network. This can be achieved using an inbound web service. The Cúram infrastructure generates the necessary deployment artifacts and packages them for deployment. After the application EAR file is deployed, any application that wants to communicate with the Cúram application must implement the appropriate functionality based on the WSDL for the web service. The infrastructure relies on the web service class to be modeled and it utilizes Axis2 tooling in the generation step for inbound web services. For information about how to develop Cúram inbound web services, see [1.3 Developing Inbound Web Services on page 18](#).

Benefits of the Apache Axis2 Platform

Apache Axis2 is the supported platform, or stack, that is supported for web services. There are several benefits to using Axis2.

There are other web service platforms that you can adapt for use with Cúram instead of Axis2. However, the benefits of Axis2 web services include the following.

- Axis2 provides significant improvements in flexibility due to the new architecture and improved performance. Performance improvements come from a change in XML parser changes by using the StAX API. The StAX API is faster than the SAX event-based parsing that was used in the previous web services implementation.
- New message types are available - This third generation of web service support makes new message exchange patterns (MEPs) available. Rather than just in-out processing, in-only (also known as fire-and-forget) and other MEPs are now available.
- Support for new and updated standards such as SOAP (1.2 and 1.1) and WSDL (2.0 and 1.1).

Types of Web Services

Web services are categorized in a number of ways. One of the main groupings is the web service style and use that determines the way that web service operation parameters are handled.

The *style* option that is defined by the WSDL specification determines the structure of the SOAP message payload. The payload is the contents of the `<soap:body>` element.

- Document (also referred to as document-oriented web services, or DOWS). The contents of the web service payload are defined by the schema in the `<wsdl:type>` and is sent as a self-contained document. This style is flexible and can process parameters and return data, or by using IBM® Rational® Software Architect Designer modeling, can be a W3C Document that is passed as an argument and return value. Document is assumed to be the default style if not specified.

- **RPC:** The contents of the payload must conform to the rules specified in the SOAP specification, that is, `<soap:body>` and can contain one element only. The element is named after the operation. Also, all parameters must be represented as subelements of this wrapper element. Typically, subelements would be parameters and return values.

Regardless of the choice of style, the contents of the SOAP message payload might look the same for a SOAP message regardless of whether document or RPC style is specified in the WSDL. This is because of the freedom available in the case of the document style.

The *use* option determines the serialization rules that are used by the web service client and server to interpret the payload of the SOAP message.

- **Literal.** The type definitions are self-defining, following an XML schema definition in `<wsdl:types>` by using either the *element* or *type* attribute.
- **Encoded:** The rules to encode and interpret the payload application data are in a list of URIs specified by the *encodingStyle* attribute, from the most to least restrictive. The most common encoding is SOAP encoding, which specifies how objects, arrays, and so on, must be serialized into XML.

The style and use options for a web service are specified in the WSDL `<wsdl:binding>` section (see <http://www.w3.org/TR/wsdl> and <http://www.w3.org/TR/wsdl20>) as attributes and control the content and function of the resulting SOAP (see <http://www.w3.org/TR/soap11> and <http://www.w3.org/TR/soap12>) message.

The following WSDL fragment illustrates the context for these settings, where the different values for the options are separated by the pipe (|) character:

```
<wsdl:binding name="myService" ... >
  <soap:binding transport="..." style="document|rpc" />
  <wsdl:operation name="myOperation">
    <soap:operation soapAction="urn:op2" style="document" />
    <wsdl:input>
      <soap:body use="literal|encoded"
        encodingStyle="uri-list" ... />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal|encoded"
        encodingStyle="uri-list" ... />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

The *encoded* use option is discouraged by the Web Services Interoperability Organization (WS-I) and the Document/Literal is the preferred choice for web service style and use.

Within the context of the Document/Literal style, use pairing is the concept of "wrapped" and "unwrapped". This pairing is not a specific style or use, but a pattern that is characterized by a single part definition, each part definition in the WSDL references an element, not a type as in RPC (it's these referenced elements that serve as the "wrappers"), the input wrapper element must be defined as a complex type that is a sequence of elements, the input wrapper name must have the same name as the operation, the output wrapper name must have the same name as the operation with "Response" appended to it, and, the style must be "document" in the WSDL binding section. Based on the capabilities of Apache Axis2 only the "wrapped" pattern is

supported¹. However, it is not supported by WSDL 2.0. The following WSDL fragment illustrates this pattern by using a simple web service that multiplies two numbers and returns the results.

```

...
<wsdl:types>
  ...
  <xs:element name="simpleMultiply">
    <xs:complexType>
      <xs:sequence>
        <xs:element
          minOccurs="0"
          name="args0"
          type="xs:float"/>
        <xs:element
          minOccurs="0"
          name="args1"
          type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="simpleMultiplyResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element
          minOccurs="0"
          name="return" type="xs:float"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</wsdl:types>
...
<wsdl:message name="simpleMultiplyRequest">
  <wsdl:part name="parameters"
    element="ns:simpleMultiply"/>
</wsdl:message>
<wsdl:message name="simpleMultiplyResponse">
  <wsdl:part name="parameters"
    element="ns:simpleMultiplyResponse"/>
</wsdl:message>
...
<wsdl:operation name="simpleMultiply">
  <wsdl:input message="ns:simpleMultiplyRequest"
    wsaw:Action="urn:simpleMultiply"/>
  <wsdl:output message="ns:simpleMultiplyResponse"
    wsaw:Action="urn:simpleMultiplyResponse"/>
</wsdl:operation>
...
<wsdl:operation name="simpleMultiply">
  <soap:operation soapAction="urn:simpleMultiply"
    style="document"/>
  <wsdl:input>
    <soap:body use="literal"/>
  </wsdl:input>

```

¹ Because only the Document/Literal-wrapped pattern for Axis2 is supported, turning this off via `doclitBare` set to `true` in the `services.xml` descriptor file is not supported.

```

    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:operation>
...

```

The following table shows the various style and use combinations that are supported in Cúram.

Table 1: Summary of Web Service Style and Use Support

Style/Use	Cúram with Axis2
RPC/Literal	-
Document/Encoded	Not supported (not WS-I compliant)
Document/Literal (wrapped)	Supported

Of the supported style and use combinations, there are a number of relative strengths and weaknesses to consider when defining your web services.

Table 2: Summary of Web Service Style and Use Strengths and Weaknesses

Style/Use	Strengths	Weaknesses
Document/Literal (wrapped)	<ul style="list-style-type: none"> • WS-I compliant • No type encoding information • Can validate in a standard way • Operation name in SOAP message 	<ul style="list-style-type: none"> • Very complex WSDL
RPC/Literal (Axis2 only)	<ul style="list-style-type: none"> • WS-I compliant • WSDL is straightforward • Operation name is included in the WSDL • No type encoding information 	<ul style="list-style-type: none"> • Hard to validate the message
RPC/Encoded (legacy only)	<ul style="list-style-type: none"> • WSDL is straightforward • Operation name is included in the WSDL 	<ul style="list-style-type: none"> • Not WS-I compliant

Web Services Security

To ensure that your valuable and sensitive enterprise data remains safe, it is important to consider web service security in your planning, implementation, and runtime support of web services.

The security is implemented entirely by the facilities that are integrated with Axis2, which includes WS-Security, wss4j, and so on. However, with the support of web services with Axis2, there is the option (recommended and on by default) that requires clients of inbound web services to provide credentials by using Cúram custom SOAP headers.

1.2 Building Outbound Web Service Connectors

You can create Cúram outbound web services. An Cúram outbound web service connector allows the application to access external applications that expose a web service interface.

The WSDL file that describes this interface is used by the web service connector functionality in Cúram to generate the appropriate client code (stubs) to connect to the web service.

Including the WSDL Files in Your Components File System

You must have at least one WSDL file to generate client stubs. Place the WSDL file or files in the file system, which is usually under source control.

The directories and files that are used are structured as follows.

```

+ EJBServer
  + build
    + svr
      + wsc2
        + <service_name>
          - <service_name>.wsdl           - where modeled service
                                         WSDL files are built to
      + jav
        + src
          + wsconnector                 - default location for
                                         generated stub source;
                                         override with property
                                         axis2.java.outdir
        + wsconnector                   - default location for
                                         compiled stub code;
                                         override, with axis2.
                                         extra.wsdl2java.args
                                         property
    + components
      + custom
        + axis
          - ws_outbound.xml             - where you identify
                                         your WSDL files
        + <service_name>
          + <service_name>.wsdl         - where you might copy a
                                         WSDL file as pointed to
                                         by ws_outbound.xml

```

Place the WSDL files in the custom folder under the location that is represented by your `SERVER_DIR` environment variable, typically, `EJBServer/components/custom`, and specify the location in the `ws_outbound.xml`. Placing your WSDL in this structure ensures that your web services are isolated from Cúram web services. The base name of the root WSDL file must use the service name.

Adding the WSDL File Location to the Outbound Web Services File

For each component that you want to build an outbound web service connector for, you must specify the location of the WSDL file or files in a `ws_outbound.xml` file.

The location for this file is typically `EJBServer/components/custom/axis/ws_outbound.xml`.

Specify the location of the WSDL file or files as shown in this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<services>
  <service name="SomeService"
    location=
      "components/custom/axis/SomeService/SomeService.wsdl"/>
</services>
```

The *ws_outbound.xml* file contains one service entity for each web service, which specifies the service name (matching the WSDL file base name) and location (relative to the `SERVER_DIR` environment variable).

Generating the Web Service Stubs

Generate the web service stubs by invoking the build script. The generation of the web service stubs is based on the contents of the *ws_outbound.xml* files, as specified by your component structure, the setting of the `COMPONENT_ORDER` environment variable, and the files in your *components/custom/axis* directories.

Invoke the Cúram build script:

```
build wsconnector2
```

Each WSDL file that is identified by the *ws_outbound.xml* files is used to generate the stub source code, which is compiled to produce executable code. The generated source is located in the *EJBServer/build/svr/wsc2/jav/src/wsconnector* directory and any compiled Java™ code is located in the *EJBServer/build/svr/wsc2/jav/wsconnector* directory.

By default, the client stubs are generated with Axis2 data bindings (ADB). However, you can generate some or all of your stubs by using XMLBeans bindings. To generate all stubs by using XMLBeans bindings, run the `wsconnector2` Ant target with the argument: `-Daxis2.extra.wsdl2java.args="-d xmlbeans"`.

Sometimes not all clients are suitable for the same binding. You can override the ADB default selectively by adding the `extraWsdl2javaArgs="-d xmlbeans"` attribute to the service definitions in the *ws_outbound.xml* file, for example:

```
<service name="SomeService"
  location="components/custom/axis/SomeService/SomeService.wsdl"
  extraWsdl2javaArgs="-d xmlbeans"
/>
```

Creating a Client and Starting the Web Service

To start the web service, you must create and build a client, for example, a Java™ main program, that uses the generated stubs to prepare arguments, start the web service, and process the return results.

Starting the web service and using the generated code depends on your development environment. For example, assuming that the web service is deployed and tested, it might include the following steps.

1. Copy or reference the generated source and class files, for example, reference in Eclipse.
2. Code your client, for example, a Java™ main program. Typically, your steps include the following:
 - Instantiate the generated stub class.
 - Optionally, increase the client timeout threshold, especially for a client that might run first after the application server starts.
 - Set up the credentials in the custom SOAP header. For more information, see [Custom SOAP Headers on page 31](#).
 - Call the stub methods to instantiate objects and set their values to pass to the service.
 - Call the service operation.
 - Check the response.
3. Build and test.

Typically the generated stub code provides a number of options to start the web service. These sample code fragments can help illustrate that process.

The following sample web service client fragment calls a service that is named `simpleAdd` in class `WebServiceTest`. For which the external tooling generates `WebServiceTestStub` and related classes:

```
final WebServiceTestStub stub =
    new WebServiceTestStub();

// Set client timeout for slow machines.
ServiceClient client = stub._getServiceClient();
client.getOptions().setProperty(
    HTTPConstants.SO_TIMEOUT, new Integer(180000));
client.getOptions().setProperty(
    HTTPConstants.CONNECTION_TIMEOUT, new Integer(180000));

// test string and primitive data types
final WebServiceTestStub.SimpleAdd service =
    new WebServiceTestStub.SimpleAdd();
final int i = 20;
final int j = 30;
service.setArgs0(i);
service.setArgs1(j);

final WebServiceTestStub.SimpleAddResponse
    simpleAddResponse = stub.simpleAdd(service);
final long sum = simpleAddResponse.get_return();

client.cleanupTransport(); // Call when done with the service
                           // to avoid exhausting connection pool.
client.cleanup();         // Call when done with the client.
```

Sometimes, while the generated code is convenient, you need a little more control over your client environment. The following example illustrates how you might call an in-only service by

using a "hand-built" SOAP message, which in this case takes a simple String argument as input. A sample web service client that uses generated stub and custom code is shown.

```
final TestWSStub stub =
    new TestWSStub();

// Get client from stub
ServiceClient client;
client = stub._getServiceClient();

/*
 * Define SOAP using string
 */
final String xml = " <rem:testString "
    + "xmlns:rem=\"http://remote.testmodel.util.curam\"> "
    + " <rem:testString>"
    + " My test string!"
    + "</rem:testString>"
    + " </rem:testString>";

final ByteArrayInputStream xmlStream =
    new ByteArrayInputStream(xml.getBytes());
final StAXBuilder builder = new StAXOMBuilder(xmlStream);
final OMElement oe = builder.getDocumentElement();

// Send the message
client.fireAndForget(oe); // API for In-Only processing
Thread.sleep(10000); // Required for fireAndForget()
client.cleanupTransport(); // Call when done with the service
// to avoid exhausting connection pool.
client.cleanup(); // Call when done with the client.
```

Note: Later versions of Axis2 Javadoc indicate that unless your client sets the `callTransportCleanup` property to `true` (not recommended for performance reasons) on the `org.apache.axis2.client.Options` object that you must call the `org.apache.axis2.client.ServiceClient.cleanupTransport()` API after processing the response.

Client Stub Pool Configuration

The following configuration settings are available for configuring the client stub pool.

- **curam.ws.client_stub_pool_size_per_endpoint**
The number of cached client stubs per web service end point (IP-address:port:web-service). The value of this property must be \leq `curam.ws.client_max_host_connections`. The default value is 50.
- **curam.ws.client_stub_pool_idle_time**
The amount of time in milliseconds that a pool entry can sit idle before it is eligible for eviction. The default value is 300,000 (5 minutes).
- **curam.ws.client_stub_pool_eviction_run_interval**
The amount of time in milliseconds between runs of evictions checks. The default value is 60,000 (1 minute).
- **curam.ws.client_max_total_connections**
The number of allowed total outbound connections to web service end points. The value of this property must be \geq `curam.ws.client_max_host_connections`. The default value is 100.

- **curam.ws.client_max_host_connections**
The allowed number of connections to one host end point (IP-address:port). The value of this property must be \geq curam.ws.client_stub_pool_size_per_endpoint and \leq curam.ws.client_max_total_connections. The default value is 50.
- **curam.ws.client_connection_timeout**
The amount of time, in milliseconds, that a client stub waits for a connection to a web service end point. The default value is 60,000 (1 minute).
- **curam.ws.client_socket_timeout**
The amount of time, in milliseconds, that a socket operation waits before a timeout error is generated. The default value is 60,000 (1 minute).
- **curam.jmx.ws_outbound_statistics_enabled**
The application property that specifies whether the JMX statistics for outbound web services calls are enabled.

1.3 Developing Inbound Web Services

An inbound web service is Cúram application functionality that is exposed to other internal or external applications in the network. This information describes the infrastructure that supports these services and the steps that you must complete to use it.

Getting Started

An overview of the process for developing inbound web services.

- **Model your web service and provide implementation code**
You need to define the classes (WS Inbound) and operations in IBM® Rational® Software Architect Designer that you are implementing to provide the functionality that you want to expose as web services.

As with any Cúram process class, you must provide the implementation for the classes and operations you model that is described in the .
- **Build your web services and the web services EAR file**
The Cúram build system will build and package your web services. Use the server and EAR file build targets that are described in the [guide](#) and the deployment guide that is appropriate to your platform.
- **Provide security data for your web services**
By default your web services are not accessible until you: a) Provide security data (see [Providing Security Data for Web Services on page 24](#)) that defines the service class and operation and which security group(s) can access them; and b) Your clients must then provide credentials appropriate to those security definitions (see [Custom SOAP Headers on page 31](#) (unless you choose to disable this security functionality; see [Custom Credential Processing on page 26](#)).

Each of the above steps is explained in more detail in the sections that follow. To better understand the process just outlined the following illustrates the structure of directories and files used.

```
+ EJBServer
```

+ build	
+ svr	
+ gen	- where the generator places ws_inbound.xml property files
+ wsc2	- where modeled service WSDL files are generated
- <service_name>.wsdl	
+ components	
+ custom	
+ axis	
+ <service_name>	- where you might place a custom ws_inbound.xml property file
- ws_inbound.xml	- where you might place a custom services.xml descriptor file
- services.xml	- where optional schema validation code would go
+ source	- where you might place optional schema
+ schemas	- where you must place custom receiver code
+ webservice	

Figure 1: File System Usage For Inbound Web Services

Modeling and Implementing an Inbound Web Service

Based on your design decisions, you will need to model the necessary classes and operations, and set the appropriate properties in the Cúram model.

For more information about how to use the IBM® Rational® Software Architect Designer tool with the Cúram model, see *Working with the Cúram Model in Rational Software Architect Designer*.

You must also code your web service implementation classes in accordance with the standard Cúram development process that is described in the [guide](#).

When you model your web services, consider the following.

- The web service binding style - Document (recommended, default) or RPC.
- The web service binding use - Literal or Encoded.

Note: Not all combinations of binding style and use are supported. For more information, see [Types of Web Services on page 10](#).

- Whether the service is processing struct and domain types or a W3C Document.

Creating Inbound Web Service Classes

To add an Axis2 inbound web service class to a package in IBM® Rational® Software Architect Designer, select Add Class, WS Inbound from the right-click context menu and name the class.

Note: In Cúram, web service names are based on the class name that are specified in the Rational® Software Architect Designer model and must be unique within the environment.

If you require passing and returning a W3C Document instead of Cúram domain types or structs you must:

1. In the **Curam** properties tab for the WS Inbound class, select the WS_Is_XML_Document property (if passing W3C Documents providing schema validation is an optional activity and is detailed in [Providing schema validation on page 29](#));
2. Select `True` as the value from the drop down.

By default the web service style for the class is document, which is defined in the WS_Binding_Style property as " 0 - Unspecified ". If you require the RPC binding style:

1. In the **Curam** properties tab, select the WS_Binding_Style property;
2. Select " 2 - RPC " as the value from the drop down.

You can also set the value explicitly to " 1 - Document ", but the generator defaults the " 0 - Unspecified " value to be document.

The class properties above will apply uniformly to all operations of the web service class; so, you need to plan your design to account for this. That is, a class can contain W3C Document operations or operations that use native data types or Cúram structs, but not both. Similarly the binding style (WS_Binding_Style) will be applied to all operations of a class when passed as an argument to the Java2WSDL tool; so, any requirement for operations with a different binding style in generated WSDL would need to be handled in a separate modeled class.

Adding Operations to Inbound Web Service Classes

In IBM® Rational® Software Architect Designer, you add operations to Axis2 inbound web service classes by using the right-click context menu.

Add an operation to an inbound web service class.

1. Select **Operation** from the right-click context menu and choose **Default**.
2. In the Create 'default' Operation Wizard, name the operation, and select its return type.

The following are issues with Axis2 that are relevant to you when you model inbound web services:

- Certain method names on inbound web services do not operate as expected because, when handling an inbound web service call, Java reflection is used to find and start methods in your application. The Axis2 reflection code identifies methods by name only (that is, not by signature). This identification means that unexpected behavior can occur if your web service interface contains a method with the same name as an inherited method. Each inbound web service in your application causes a facade bean, that is, a stateless session bean to be generated.

So, in addition to your application methods, this class also contains methods that are inherited from `javax.ejb.EjbObject`, and possibly others generated by your application server tooling. For example: `remove`, `getEJBHome`, `getHandle`.

This limitation is logged with Apache in JIRA *AXIS2-4802*. Currently, the only workaround is to ensure that your inbound web service does not contain any methods whose names conflict with those that are in `javax.ejb.EjbObject`.

Adding Arguments and Return Types to Inbound Web Service Operations

You add arguments and return types to inbound web service operations in the same way that they are added to process and facade classes. However, they are only relevant for classes that don't specify support for W3C Documents (`WS_Is_XML_Document` property).

For more information about how to add arguments and return types to process classes, see the related link to *Modeling Cúram elements using Rational® Software Architect Designer*.

Note: When modeling a web service struct aggregation in IBM® Rational® Software Architect Designer graphical mode, Rational® Software Architect Designer automatically adds an aggregation label. This causes the WSDL to be generated incorrectly. Remove this label in the model before building and the WSDL will generate correctly.

Related information

Processing of Lists

An operation uses Cúram lists if its return value or any of its parameters utilize a struct which aggregates another struct using 'multiple' cardinality.

In the UML metamodel, you can model a `<<WS_Inbound>>` operation that uses parameters that contain lists, that is, a struct that aggregates one or more other structs as a list. All operations that are visible as a web service are normally also visible to the web client.

However, the web client does not support the following:

- List parameters.
- Non-struct parameters, that is, parameters which are domain definitions.
- Non-struct operation return types.

In these cases, the web client ignores the operations that it does not support, but these operations can be used for Axis2 inbound web services.

Data Types

The Cúram data types, except `Blob` (`SVR_BLOB`), can be used in Axis2 inbound web service operations.

The mappings between Cúram and WSDL data types are shown in the following table:

Table 3: Cúram to WSDL data types for Axis2

Cúram data type	WSDL data type
SVR_BOOLEAN	xsd:boolean

Cúram data type	WSDL data type
SVR_CHAR	xsd:string
SVR_INT8	xsd:byte
SVR_INT16	xsd:short
SVR_INT32	xsd:int
SVR_INT64	xsd:long
SVR_STRING	xsd:string
SVR_DATE	xsd:string (Format: <code>yyyymmdd</code>)
SVR_DATETIME	xsd:string (Format: <code>yyyymmddThhmmss</code>)
SVR_FLOAT	xsd:float
SVR_DOUBLE	xsd:double
SVR_MONEY	xsd:float

With the supported data types shown in [Data Types on page 21](#), only the related XML schema types that map to primitive Java types and `java.lang.String` are supported for inbound web services. For example, "xsd:boolean" and "xsd:long" that map to the boolean and long Java types, respectively, and "xsd:string" that maps to `java.lang.String` are supported. All other XML schema types that do not map to a Java primitive type or to `java.lang.String` are not supported.

An example of an unsupported XML schema type is "xsd:anyURI", which maps to `java.net.URI`. This limitation applies to inbound web services only and is due to the fact that inbound web services are generated based on what can be represented in a Cúram model. Outbound web services are not affected by this issue. For more details on related modeling topics, see *Working with the Cúram Model in Rational Software Architect Designer* and the .

Note: Passing or returning the raw Cúram data types, that is, Date, DateTime, Money, as an attribute to an Axis2 web service is restricted. Cúram data types must be wrapped inside a struct before they are passed as attributes to a web service.

Building and Packaging Web Services

Use the targets `websphereWebServices`, `weblogicWebServices`, and  `libertyWebServices` to build the web services EAR file.

The steps in this build process are as follows.

1. Package global WAR file directories: lib, conf, modules.
2. Iterate over the web service directories in `build/svr/gen/wsc2` (one directory per web service class) that are created by the generator.
 - Process the properties in the following order: custom, generator, defaults. For more information, see [Inbound Web Service Properties File on page 25](#).

- Unless a custom *services.xml* has been provided, generate the *services.xml* descriptor file, For more information, see [Deployment Descriptor File on page 25](#).
- Package the web service directory.

The following properties and customizations are available.

- You can turn off the generation of the *webservices2.war* by setting the property `disable.axis2.build`.
- You can specify an alternate location for the build to read in additional or custom Axis2 module files by setting the `axis2.modules.dir` property that will contain all the *.mar* files and the *modules.list* file to be copied into the `WEB-INF\modules` directory;
- You can include additional, external content into the *webservices.war* by setting either of the following properties.
 - `axis2.include.location` - that points to a directory containing a structure mapping to the Axis2 WAR file directory structure;
 - `axis2.include.zip` - that points to a zip file containing a structure mapping to the Axis2 WAR file directory structure.

With either of the two properties above, setting the `axis2.include.override` property causes these contents to override the Cúram packaged content in the WAR file. This capability is for including additional content into your WAR file. An example of how you might use this is to include the sample Version service to enable Axis2 to successfully validate the environment (see [Initial Server Validation and Troubleshooting on page 51](#)).

For example, to include the sample Version web service for IBM® WebSphere® Application Server you need to create a directory structure that maps to the *webservices2.war* file and includes the structure of *Version.aar* file as is shipped in the Axis2 binary distribution: `axis2-1.5.1-bin/repository/services/version.aar`. That structure would look like this:

```
+ WEB-INF
  + services
    + Version
      + META-INF
        - ./services.xml
      + sample
        + axisversion
          - ./Version.class
```

Then, if the location of the *Version* directory were in `C:\Axis2-includes`, you would specify the following property value at build time: `-Daxis2.include.location=C:\Axis2-includes`. Alternatively, you could package the above file structure into a zip file and specify the `-Daxis2.include.zip` property instead. In both cases the file structure specified would be overlaid onto the file structure (depending on the value of `axis2.include.override`) and packaged into the *webservice2.war* WAR file. (For Oracle® WebLogic Server the above would be changed to replace the contents of the *Version* directory with a *Version.aar* file, which is a compressed file.)

- You can set global, default web services credentials at build time through the following properties that are set in your *Bootstrap.properties* file.

- `curam.security.credentials.ws.username` - the username that is used when executing inbound web service calls;
- `curam.security.credentials.ws.password` - the password that is used when executing inbound web service calls. This password must be encrypted.

The above credentials must exist on the **Users** table, must be enabled, and should be assigned the appropriate security role.

Default credentials can streamline your development and testing processes, but should not be used in a production environment when working with sensitive data and/or processes.

Providing Security Data for Web Services

You must provide security data in order to make your web service usable. In Cúram, web services are not automatically associated with a security group. This is to ensure that web services are not vulnerable to a security breach.

As part of your development process, ensure that the appropriate security database entries are created. For example:

```
INSERT INTO SecurityGroupSid (groupname, sidname)
values ('WEBSERVICESGROUP', 'ServiceName.anOperation');
```

For more information about the contents of the Cúram security tables, see .

Providing Web Service Customizations

Providing customizations at build-time impacts the security and behavior of your web service at run time. With the default configuration, the web services EAR file build performs the following tasks:

- Assigns the appropriate Cúram message receiver for struct and domain types, for argument and operation return values, or for W3C Documents. This assignment is based on how you set the `WS_Is_XML_Document` property in Rational Software Architect for the "WS Inbound" (stereotype: `<<wsinbound>>`) class.
- Expects the web service client to pass a custom SOAP header with authentication credentials to start the web service.

To change the default behaviors, you require a custom receiver. For more information, see [Customizing Receiver Runtime Functionality on page 25](#). You might also need to customize the following.

- Implementing web services security (Apache Rampart). For more information, see [1.4 Securing Web Services on page 30](#).
- Providing external, non- Cúram functionality such as the Apache Axis2 Monitor. For more information, see [1.9 Including the Axis2 SOAP Monitor in Your Web Services WAR File on page 57](#).
- Providing other custom parameters for the deployment descriptor (`services.xml`), for example: `doclitBare, mustUnderstand`. For more information, see the Apache Axis2 documentation for more information ([Apache Axis2 Configuration Guide](#)).

To effectively customize your web services you need to know how Cúram processes web services at build time, which is explained in the following sections.

Inbound Web Service Properties File

Based on the web service classes modeled with IBM® Rational® Software Architect Designer, the generator creates a folder in the *build/svr/gen/wsc2* directory for each web service class modeled.

For more information, see [Getting Started on page 18](#). (This maps closely to how Axis2 expects services to be packaged for deployment.) In that folder a properties file, *ws_inbound.xml*, is generated.

To provide a custom *ws_inbound.xml* file, you can start with the generated copy that you will find in the *build/svr/gen/wsc2/<service_name>* directory after an initial build. Place your custom *ws_inbound.xml* file in your *components/custom/axis/<service_name>* directory (usually under source control). During the build the *ws_inbound.xml* files are processed to allow for a custom file first, overriding generated and default values. For more information about the property settings in this file, see [1.5 Inbound Web Service Properties: ws_inbound.xml on page 46](#).

Deployment Descriptor File

Each web service class requires its own deployment descriptor file (*services.xml*).

The build automatically generates a suitable deployment descriptor for the defaults in accordance with [1.5 Inbound Web Service Properties: ws_inbound.xml on page 46](#). The format and contents of the *services.xml* are defined by Axis2. See the *Apache Axis2 Configuration Guide* (<http://axis.apache.org/axis2/java/core/docs/axis2config.html>) for more information.

To provide a custom *services.xml* file, start with the generated copy that is located in the *build/svr/wsc2/<service_name>* directory after an initial build of the web services WAR/EAR file. This is illustrated in [Getting Started on page 18](#).

Place your custom *services.xml* file in your *components/custom/axis/<service_name>* directory (usually under source control). For details about this contents file, see [1.6 Deployment Descriptor File: services.xml on page 48](#). During the build, the *services.xml* files are packaged into the web services WAR file (*webservices2.war*) as per Axis2 requirements, that is, using this file system structure: *WEB-INF/services/<service_name>/META-INF/services.xml*. See the *Apache Axis2 User's Guide - Building Services* (<http://axis.apache.org/axis2/java/core/docs/userguide-building-services.html>).

Customizing Receiver Runtime Functionality

The default receivers that are provided with Cúram should be sufficient for most cases. However, you can provide overrides for the following functionality.

- Credentials processing
- Accessing the SOAP Message
- Application server-specific provider URL and context factory parameters
- SOAP factory provider for W3C Document processing

Custom Credential Processing

You might need to customize credentials processing, for example, if you want to obtain or validate credentials externally before passing them to the receiver for authentication.

By default, Cúram web services are built to expect the client to provide credentials using a custom SOAP header. These credentials are then used in starting the service class operation. The default processing flow is as follows:

- Unless *curamWSClientMustAuthenticate* is set to *false* in the *services.xml* descriptor for the service, the SOAP message is checked for a header and if present these credentials are used. If the SOAP header is not present, then the invocation of the service fails.
- If *curamWSClientMustAuthenticate* is set to *false* the *services.xml* *jndiUser* and *jndiPassword* parameters are used.
- If there are no *jndiUser* and *jndiPassword* parameters in the *services.xml* descriptor file, default credentials are used.

However, there is no security data generated for web services. In this case, the defaults credentials on their own are not adequate to enable access to the service. For more information on providing this data, see [Providing Security Data for Web Services on page 24](#).

If you require your own credential processing you must code your own `getAxis2Credentials(MessageContext)` method, extending `curam.util.connectors.axis2.CuramMessageReceiver`, to provide these parameters. This method takes a `MessageContext` object as an input parameter and returns a `java.util.Properties` object that contains the Axis2 parameter name and value. For example:

```
public Properties getAxis2Credentials(
    final MessageContext messageContextIn) {

    final Properties loginCredentials = new Properties();

    String sUser = null;
    String sPassword = null;

    <Your processing here...>

    if (sUser != null) {
        loginCredentials.put(
org.apache.axis2.rpc.receivers.ejb.EJBUtil.EJB_JNDI_USERNAME,
        sUser);
    }

    if (sPassword != null) {
        loginCredentials.put(
org.apache.axis2.rpc.receivers.ejb.EJBUtil.EJB_JNDI_PASSWORD,
        sPassword);
    }

    return loginCredentials;
}
```

Figure 2: Sample `getAxis2Credentials` Method

See [Building Custom Receiver Code on page 28](#) on how to specify and build this custom class for this method.

You can use the runtime properties `curam.security.credentials.ws.username` and `curam.security.credentials.ws.password` (encrypted) to specify default web services credentials. Using runtime properties might not be appropriate in a secure production environment; but, could be a useful, for instance, in development for simulating functions that would ultimately be provided by an external security system. For more information on encrypted passwords, see the *Curam Security Guide*.

Accessing the SOAP Message

If you require access to the SOAP message, you can extend the Curam receiver class as shown in the following example.

```
package webservice;

import org.apache.axis2.AxisFault;
import org.apache.axis2.context.MessageContext;
import org.apache.log4j.Logger;

/**
 * Sample SOAP message access.
 */
public class CustomReceiverInOutAccessSOAPMsg
    extends curam.util.connectors.axis2.CuramMessageReceiver {

    /** Class logger. */
    private final Logger log =
        Logger.getLogger(CustomReceiverInOutAccessSOAPMsg.class);

    /**
     * Access the SOAP message and invoke
     * Curam receiver invokeBusinessLogic.
     *
     * @param messageContextIn Input MessageContext.
     * @param messageContextOut Output MessageContext.
     *
     * @throws AxisFault based on called method.
     */
    @Override
    public void invokeBusinessLogic(final MessageContext
        messageContextIn,
        final MessageContext messageContextOut) throws AxisFault {
        if (messageContextIn != null) {
            final org.apache.axiom.soap.SOAPEnvelope inEnv =
                messageContextIn.getEnvelope();
            if (inEnv != null) {
                // Insert custom SOAP processing here.
                log.debug("Sample access of SOAP message: " +
                    inEnv.toString());
            }
        }

        super.invokeBusinessLogic(messageContextIn,
            messageContextOut);
    }
}
```

```
}

```

Figure 3: Sample Custom Receiver to Access the SOAP Message

Note, the invocation of `super.invokeBusinessLogic()` must be made.

See [Building Custom Receiver Code on page 28](#) on how to specify and build this custom class.

Custom Application Server-Specific Parameters

The `app_webservices2.xml` script generates correct application server-specific provider URL and context factory parameters. However, if you are supporting multiple environments, you can derive one or more of these values in your own custom code.

You can provide your own `getProviderURL()` and/or `getContextFactoryName()` methods by overriding class `curam.util.connectors.axis2.CuramMessageReceiver`. Both methods return a string representing the provider URL and context factory name, respectively. For more information about how to specify and build this custom class for these methods, see [Building Custom Receiver Code on page 28](#).

Custom SOAP Factory

Generally, the default SOAP factory, `org.apache.axiom.soap.SOAPFactory`, is adequate for processing your web services that process W3C Documents. However, you can override this behavior by providing your own `getSOAPFactory(MessageContext)` method.

This method takes a `MessageContext` object as an input parameter and returns an `org.apache.axiom.soap.SOAPFactory`.

Building Custom Receiver Code

To build custom receiver code, you must complete the following steps.

1. Extend the appropriate class. For example, `public class MyReceiver` extends `curam.util.connectors.axis2.CuramMessageReceiver`. For the list of receiver classes and their usage, see [Deployment Descriptor File on page 25](#).
2. Specify a package name of webservice in your custom Java program. For example, `package webservice;`
3. Place your custom source code in your components `source/webservice` directory. For example, `components/mycomponents/source/webservice`). The server build target builds and packages this custom receiver code.
4. Create a custom `services.xml` descriptor file for each service class to be overridden by your custom behavior. See [Deployment Descriptor File on page 25](#) and [Building Custom Receiver Code on page 28](#).

```
<messageReceivers>
  <messageReceiver
    mep="http://www.w3.org/2004/08/wSDL/in-out"
    class="webservice.MyReceiver" />
</messageReceivers>
```

Figure 4: Sample `services.xml` Descriptor File Entry for a Custom Receiver

The webservices build that is implemented in `app_webservices2.xml` packages these custom artifacts into a WAR file.

Providing schema validation

When you use web services that pass and return a W3C Document object, you might want to use schema validation to verify the integrity of the document you are processing.

Whether you choose to use schema validation depends on the following factors:

- The CPU cost of performing such validation, which depends on the volume of transactions your system encounters.
- The source of the Documents being passed to your web service, whether that is under your control or public.

The steps for validating an XML Document in an inbound web service are as follows:

1. Include the schema document in the application ear by storing it somewhere within directory `SERVER_DIR/components/**/webservices/**/* .xsd`.
2. Provide code within the implementation code of the BPO method that loads the schema file, and passes it into the infrastructure validator class along with the `org.w3c.Document` class to be validated.

The code example ([Providing schema validation on page 29](#)) illustrates how validation can be implemented.

```
import curam.util.exception.AppException;
import curam.util.exception.InformationalException;
import curam.util.webservices.DOWSValidator;
import java.io.InputStream;
import org.w3c.dom.Document;

. . .

/**
 * A sample XML document web service.
 */
public org.w3c.dom.Document
myWebServiceOperation(final org.w3c.dom.Document docIn)
throws AppException, InformationalException {

    // DOWSValidator is the SDEJ infrastructure class for
    // validating org.w3c.Document classes in web services.
    final curam.util.webservices.DOWSValidator validator =
        new curam.util.webservices.DOWSValidator();

    try {
        // The following is used only for error reporting
        // purposes by DOWSValidator. In your code you can
        // provide a relevant value to help identify the schema
        // in the event of an error.
        final String schemaURL = "n/a";

        // Load the schema file from the .ear file.
        // For example, the source location of
        // 'test1.xsd' was
        // SERVER_DIR/components/custom/webservices.

        final InputStream schemaStream =
            getClass().getClassLoader().
```

```

        getResourceAsStream("schemas/test1.xsd");

// if schema file is in
// SERVER_DIR/components/custom/webservices/test/test1.xsd
schemaStream =
    getClass().getClassLoader().
        getResourceAsStream("schemas/test/test1.xsd");

// Invoke the validator.
validator.validateDocument(docIn, schemaStream,
    schemaURL);

} catch (Exception e) {
    // Schema validation failed. Throw an exception.
    ApplicationException ae = new
        ApplicationException(SOME_MESSAGES.ERR_SCHEMA_VALIDATION_ERROR,
            e);
}

// normal BPO logic goes here.
// ...

return result;
}

```

Figure 5: Sample Illustrating Schema Validation

1.4 Securing Web Services

Web service security is an important part of your web services implementation. Use this information to learn about existing web service security and how to secure your web services.

For Rampart and Axis2 web services security, you will learn how to:

- Use and set up Rampart.
- Use custom SOAP headers with Axis2 and encrypt them.
- Use HTTPS/SSL to secure web service network traffic.

You can also use the following to secure your web services:

- Cúram modeling requirements for using secure web services.
- Code password callback handlers (also applicable to Axis2 if your policy specifies a password callback handler).
- Set up the client environment.
- Create keystore files (also applicable to Axis2 if your environment requires these steps for supporting HTTPS/SSL).

Axis2 Security and Rampart

Rampart is the security module of Axis2. With the Rampart module you can secure web services for authentication, integrity (signature), confidentiality (encryption/decryption) and non-repudiation (timestamp).

Rampart secures SOAP messages according to specifications in WS-Security, using the WS-Security Policy language.

The only specific restriction placed on the use of web service security for Cúram applications is that Rampart Authentication cannot be used. This is due to the requirements of Cúram receivers (this authentication is typically coded in the service code itself, which would be moot by that point as these receivers would have already performed authentication). However, custom SOAP headers provide similar functionality (see [Custom SOAP Headers on page 31](#) for more details).

WS-Security can be configured using the Rampart WS-Security Policy language. The WS-Security Policy language is built on top of the WS-Policy framework and defines a set of policy assertions that can be used in defining individual security requirements or constraints. Those individual policy assertions can be combined using policy operators defined in the WS-Policy framework to create security policies that can be used to secure messages exchanged between a web service and a client.

WS-security can be configured without any Cúram infrastructure changes using Rampart and WS-Security Policy definitions. A WS-Security Policy document can be embedded in a custom *services.xml* descriptor (see [Deployment Descriptor File on page 25](#)). WS-Policy and WS-SecurityPolicy can also be directly associated with the service definition by being embedded within a WSDL document.

Encryption generally incurs costs (e.g. CPU overhead) and this is a concern when using WS-Security. However, there are ways to help minimize these costs and one of these is to set the WS-SecurityPolicy appropriate for each individual operation, message, or even parts of the message for a service, rather than applying a single WS-SecurityPolicy to the entire service (for example, see [Encrypting Custom SOAP Headers on page 34](#)). To apply such a strategy you need to have a clear grasp of your requirements and exposures. Questions you might consider as you plan your overall security strategy and implementation: Can some services bypass encryption if they are merely providing data that is already available elsewhere publicly? Are multiple levels of encryption necessary; for instance, do you really need both Rampart encryption and HTTP/SSL encryption?

Custom SOAP Headers

Cúram enforces credential checking on web service invocations based on the default expectation that a client that is calling a web service has provided a custom SOAP header. This information describes how your clients can provide the required SOAP headers.

This topic was introduced in [Providing Web Service Customizations on page 24](#). If you choose to bypass this security checking, you must plan specific customizations. By default, the provided receivers for Axis2 expect the client invocation of each web service to provide a custom SOAP header that contains credentials for authenticating Cúram access to the web service.

The following is an example of the Cúram custom SOAP header in the context of the SOAP message:

```
<?xml version='1.0' encoding='UTF-8'?>
  <soapenv:Envelope
    xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
    <soapenv:Header>
      <curam:Credentials
        xmlns:curam="http://www.curamsoftware.com">
        <Username>testerID</Username>
        <Password>password</Password>
      </curam:Credentials>
    </soapenv:Header>
    <soapenv:Body>
      <!-- SOAP message body data here. -->
    </soapenv:Body>
  </soapenv:Envelope>
```

Figure 6: Example Custom SOAP Header

The following is a sample client method for creating custom SOAP headers:

```
import org.apache.axis2.client.ServiceClient;
import javax.xml.namespace.QName;
import org.apache.axiom.om.OMAbstractFactory;
import org.apache.axiom.om.OMElement;
import org.apache.axiom.om.OMFactory;
import org.apache.axiom.om.OMNode;
import org.apache.axiom.om.OMNamespace;
import org.apache.axiom.soap.SOAPFactory;
import org.apache.axiom.soap.SOAPHeaderBlock;

...

/**
 * Create custom SOAP header for web service credentials.
 *
 * @param serviceClient Web service client
 * @param userName      User name
 * @param password      Password
 */
void setCuramCredentials(final ServiceClient serviceClient,
    final String userName, final String password)

    // Setup and create the header
    final SOAPFactory factory =
        OMAbstractFactory.getSOAP12Factory();
    final OMNamespace ns =
        factory.createOMNamespace("http://www.curamsoftware.com",
            "curam");
    final SOAPHeaderBlock header =
        factory.createSOAPHeaderBlock("Credentials", ns);
    final OMFactory omFactory = OMAbstractFactory.getOMFactory();

    // Set the username.
    final OMNode userNameNode =
        omFactory.createOMElement(new QName("Username"));
    ((OMElement) userNameNode).setText(userName);
    header.addChild(userNameNode);
```

```

// Set the password.
final OMNode passwordNode =
    omFactory.createOMElement(new QName("Password"));
((OMElement) passwordNode).setText(password);
header.addChild(passwordNode);

serviceClient.addHeader(header);
}

```

Figure 7: Sample Method to Create Custom SOAP Headers

Then a call to the above method would appear as:

```

// Set the credentials for the web service:
MyWebServiceStub stub =
    new MyWebServiceStub();
setCuramCredentials(stub._getServiceClient(),
    "system", "password");

```

By default, the client that failed to provide this custom header will cause the service to not be invoked. And, of course, incorrect or invalid credentials will cause an authentication error. The following is an example of failing to provide the necessary custom SOAP header:

```

<soapenv:Envelope xmlns:
    soapenv="http://www.w3.org/2003/05/soap-envelope">
    <soapenv:Body>
        <soapenv:Fault>
            <soapenv:Code>
                <soapenv:Value>
                    >soapenv:Receiver</soapenv:Value>
                </soapenv:Code>
            <soapenv:Reason>
                <soapenv:Text xml:lang="en-US">
                    No authentication data.
                </soapenv:Text>
            </soapenv:Reason>
            <soapenv:Detail/>
        </soapenv:Fault>
    </soapenv:Body>
</soapenv:Envelope>

```

Warning: Potential Security Vulnerability

By default, custom SOAP headers that contain credentials for authentication pass on the wire in plain-text. This is an insecure situation and you must encrypt this traffic to prevent your credentials from being vulnerable and your security from being breached. For information about how you can rectify this, see [Encrypting Custom SOAP Headers on page 34](#) and/or [Securing Web Service Network Traffic with HTTPS/SSL on page 45](#).

For example, this is what the custom SOAP header looks like in the SOAP message with the credentials visible:

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
  xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  <soapenv:Header>
    <curam:Credentials
      xmlns:curam="http://www.curamsoftware.com">
      <Username>tester</Username>
      <Password>password</Password>
    </curam:Credentials>
  </soapenv:Header>
  <soapenv:Body>
    ...
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 8: Sample Custom SOAP Header

Encrypting Custom SOAP Headers

By default, SOAP headers travel across the wire as plain text. You can use Rampart to encrypt your Cúram custom SOAP headers to help to ensure the security of these credentials.

We recommend that you plan a security strategy and implementation for all of your web services and related data based on your overall, enterprise-wide requirements, environment, platforms, and so on. This information is just one small part of your overall security picture.

There is additional information on coding your web service clients for Rampart security in [Using Rampart With Web Services on page 36](#) that provides more context for the following.

The steps to encrypt these headers are follows:

1. Add the following to your client descriptor file:

```
<encryptionParts>
  {Element}{http://www.curamsoftware.com}Credentials
</encryptionParts>
```

(See [Defining the Axis2 Security Configuration on page 36](#) for more information on the contents of this file.)

Or, add the following to your Rampart policy file:

```
<sp:EncryptedElements
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:sp=
```

```

    "http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
    <sp:XPath xmlns:curam="http://www.curamsoftware.com" >
      /soapenv:Envelope/soapenv:Header/curam:Credentials/Password
    </sp:XPath>
  </sp:EncryptedElements>

```

(See [Defining the Axis2 Security Configuration on page 36](#) for more information on the contents of this file.)

2. Engage and invoke Rampart in your client code as per [Using Rampart With Web Services on page 36](#).

With WS-Security applied as per above the credentials portion of the wsse:Security header will be encrypted in the SOAP message as shown in this example below, which you can contrast with [Figure 8: Sample Custom SOAP Header on page 34](#):

In the following example encryptedParts was used to encrypt the Cúram credentials.

...

```

<?xml version='1.0' encoding='UTF-8'?>
  <soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
    <soapenv:Header>
      <wsse:Security
        xmlns:wsse="http://docs.oasis-open.org/wss/
          2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
        soapenv:mustUnderstand="1">
        <xenc:EncryptedKey
          Id="EncKeyId-A5ACA637487ECDA81713059750729855">
          <xenc:EncryptionMethod
            Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
          <ds:KeyInfo
            xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <wsse:SecurityTokenReference>
          .....
        </wsse:Security>

        <!-- Credential data is then encoded in sections
          that follow as illustrated -->
        <xenc:EncryptedData Id="EncDataId-3"
          Type="http://www.w3.org/2001/04/xmlenc#Element">
          <xenc:EncryptionMethod
            Algorithm="http://www.w3.org/
              2001/04/xmlenc#aes128-cbc" />
          <ds:KeyInfo
            xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <wsse:SecurityTokenReference
              xmlns:wsse="http://..oasis-
                200401-wss-wssecurity-secext-1.0.xsd">
              <wsse:Reference
                URI="#EncKeyId-A5ACA637444e87ECDA81713059750729855" />
            </wsse:SecurityTokenReference>

          </ds:KeyInfo>
          <xenc:CipherData>
            <xenc:CipherValue>

```

```

        eZFRrk6VSncaDanYCjyVD=</xenc:CipherValue>
    </xenc:CipherData>
</xenc:EncryptedData>
    <wsa:Action>urn:simpleXML</wsa:Action>
</soapenv:Header>

```

Figure 9: Example Encrypted Custom SOAP Header

Using Rampart With Web Services

There are a number of parts to Rampart security. Covering them in detail is outside the scope of this information. However, the following gives a high-level view on using Rampart with your Cúram Axis2 web services.

These are the steps for using web services security with Axis2:

1. Define configuration data and parameters for your client and server environments;
2. Provide the necessary data and code specified in your configuration;
3. Code a client to identify and process the configuration.

There is a lot of flexibility in how you fulfill the above steps and the following sections will show some possible ways of doing this.

Defining the Axis2 Security Configuration

While the necessary configuration will depend on what security features you choose to use, the overall set of activities will be similar regardless. On the client side, you can define the security configuration via a client Axis2 descriptor file (*axis2.xml*), Rampart policy file, or programmatically (now deprecated).

On the server side, you can define the security configuration via the service descriptor file (*services.xml*) or through a Rampart policy that is embedded in the service WSDL.

The following examples show the client and server configurations in the context of a client Axis2 descriptor and Rampart policy files and the server configuration via the context of the service descriptor file.

Client configuration:

```

<axisconfig name="AxisJava2.0">
  <module ref="rampart" />

  <parameter name="InflowSecurity">
    <action>
      <items>Signature Encrypt</items>
      <signaturePropFile>
        client-crypto.properties
      </signaturePropFile>
      <passwordCallbackClass>
        webservice.ClientPWCallback
      </passwordCallbackClass>
      <signatureKeyIdentifier>
        DirectReference
      </signatureKeyIdentifier>
    </action>
  </parameter>

```

```

<parameter name="OutflowSecurity">
  <action>
    <items>Signature Encrypt</items>

    <encryptionUser>admin</encryptionUser>
    <user>tester</user>

    <passwordCallbackClass>
      webservice.ClientPWCallback
    </passwordCallbackClass>

    <signaturePropFile>
      client-crypto.properties
    </signaturePropFile>
    <signatureKeyIdentifier>
      DirectReference
    </signatureKeyIdentifier>

    <encryptionParts>
      {Element}{http://www.curamssoftware.com}Credentials
    </encryptionParts>

  </action>
</parameter>
...

```

Figure 10: Sample Client Descriptor Settings (Fragment)

Server configuration:

```

<serviceGroup>
  <service name="SignedAndEncrypted">
    ...

    <module ref="rampart" />

    <parameter name="InflowSecurity">
      <action>
        <items>Signature Encrypt</items>
        <passwordCallbackClass>
          webservice.ServerPWCallback
        </passwordCallbackClass>
        <encryptionUser>admin</encryptionUser>
        <user>tester</user>
        <signaturePropFile>
          server-crypto.properties
        </signaturePropFile>
        <signatureKeyIdentifier>
          DirectReference
        </signatureKeyIdentifier>
      </action>
    </parameter>

    <parameter name="OutflowSecurity">
      <action>
        <items>Signature Encrypt</items>
        <encryptionUser>admin</encryptionUser>
        <user>tester</user>

```

```

        <passwordCallbackClass>
            webservice.ServerPWCallback
        </passwordCallbackClass>
        <signaturePropFile>
            server-crypto.properties
        </signaturePropFile>
        <signatureKeyIdentifier>
            DirectReference
        </signatureKeyIdentifier>
    </action>
</parameter>

...

</service>
</serviceGroup>

```

Figure 11: Sample Server Security Settings (*services.xml* Fragment)

All Rampart clients must specify a configuration context that at a minimum identifies the location of the Rampart and other modules. The following example illustrates this and includes a client Axis2 descriptor file. Later code examples will utilize this same structure assuming it is located in the `C:\Axis2\client` directory.

```

modules/
  addressing-1.3.mar
  rahas-1.5.mar
  rampart-1.5.mar
conf/
  client-axis2.xml

```

Figure 12: Axis2 Client File System Structure

The equivalent specification to the parameters in [Defining the Axis2 Security Configuration on page 36](#) and [Defining the Axis2 Security Configuration on page 36](#) via a Rampart policy file would be as follows:

(*policy.xml* Fragment)

```

...
<ramp:RampartConfig
  xmlns:ramp="http://ws.apache.org/rampart/policy">
  <ramp:user>beantester</ramp:user>
  <ramp:encryptionUser>curam</ramp:encryptionUser>
  <ramp:passwordCallbackClass>
    webservice.ClientPWCallback
  </ramp:passwordCallbackClass>

  <ramp:signatureCrypto>
    <ramp:crypto
      provider="org.apache.ws.security.components.crypto.Merlin">
      <ramp:property

name="org.apache.ws.security.crypto.merlin.keystore.type">
  JKS
    </ramp:property>
    <ramp:property
      name="org.apache.ws.security.crypto.merlin.file">
      client.keystore

```

```

    </ramp:property>
    <ramp:property
      name=
        "org.apache.ws.security.crypto.merlin.keystore.password">
      password
    </ramp:property>
  </ramp:crypto>
</ramp:signatureCrypto>
<ramp:encryptionCrypto>
  <ramp:crypto
    provider="org.apache.ws.security.components.crypto.Merlin">
    <ramp:property
      name="org.apache.ws.security.crypto.merlin.keystore.type">
      JKS
    </ramp:property>
    <ramp:property
      name="org.apache.ws.security.crypto.merlin.file">
      client.keystore
    </ramp:property>
    <ramp:property
      name=
        "org.apache.ws.security.crypto.merlin.keystore.password">
      password
    </ramp:property>
  </ramp:crypto>
</ramp:encryptionCrypto>
</ramp:RampartConfig>
...

```

Figure 13: Sample Rampart Policy

Providing the Security Data and Code

Use the example configurations in the Axis Security Configuration section to specify an encryption property file and password call back routine that is used to encrypt your web service data.

The value of `signaturePropFile` specifies the name of the signature crypto property file to use. This file contains the properties used for signing and encrypting the SOAP message. An example server crypto property file is shown in [Providing the Security Data and Code on page 39](#). When you use a Rampart policy file, as shown in [Defining the Axis2 Security Configuration on page 36](#), these property files are not relevant as the policy itself contains the equivalent settings.

```

org.apache.ws.security.crypto.provider=
org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.file=server.keystore

```

Figure 14: Example Rampart server-crypto.properties File

The `client-crypto.properties` file has similar properties as above, but with client-specific values:

```

org.apache.ws.security.crypto.provider=
org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks

```

```
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.file=client.keystore
```

The creation of the keystore file and the related properties are discussed in [Creating Keystore Files on page 46](#).

When you configure a secure web service, you must place the server signature property file and keystore file (*server-crypto.properties* and *server.keystore*) in the `%SERVER_DIR%/project/config/wss/` directory so that the build will package them and they will be available on the classpath at execution time.

The password callback handlers are specified in the `passwordCallbackClass` parameter entities.

Coding the client

Code snippets show you how to add to the basic client examples.

Code snippets show to add to the basic client examples in [Creating a Client and Starting the Web Service on page 15](#) to use the preceding security illustrations.

To use a client *axis2.xml* descriptor file, make the following API call where `C:/Axis2/client` also contains the Axis2 modules directory as indicated in [Defining the Axis2 Security Configuration on page 36](#):

```
final ConfigurationContext ctx =
    ConfigurationContextFactory.
    createConfigurationContextFromFileSystem(
        // Looks for modules, etc. here:
        "C:/Axis2/client",
        // Axis2 client descriptor:
        "C:/Axis2/client/conf/client-axis2.xml");
```

Figure 15: Identifying Axis2 Client Rampart Configuration

To use a Rampart policy file you would need to create a context as shown in figure 1, but the client Axis2 descriptor is not necessary in this example, just the Axis2 modules directory:

```
final ConfigurationContext ctx =
    ConfigurationContextFactory.
    createConfigurationContextFromFileSystem(
        // Looks for modules, etc. here:
        "C:/Axis2/client",
        null);
```

When not using an Axis2 configuration that specifies the necessary modules (as shown in [Defining the Axis2 Security Configuration on page 36](#)) you must engage the necessary modules that are required before starting the service. The modules depend on the security features and configuration you are using; for example,:

```
client.engageModule("rampart");
```

Failing to do this results in a server-side error, for example:

```
org.apache.rampart.RampartException:
    Missing wsse:Security header in request
```

To use a Rampart policy, create a policy object and set it in the service options properties:

```
final org.apache.axiom.om.impl.builder.StAXOMBuilder builder =
    new StAXOMBuilder("C:/Axis2/client/policy.xml");
final org.apache.neethi.Policy policy =
    org.apache.neethi.PolicyEngine.
        getPolicy(builder.getDocumentElement());
options.setProperty(
    org.apache.rampart.RampartMessageData.KEY_RAMPART_POLICY,
    loadPolicy(policy);
```

Note: Any number of client coding errors, policy specification errors, configuration errors, and so on, can manifest in the client and/or the server. Often an error in the client cannot be debugged without access to the Apache Log4j 2 trace from the server. For instance, the error when the proper modules are not engaged (discussed earlier) and appear in the client as follows:

```
OMEException in getSOAPBuilder
org.apache.axiom.om.OMEException:
com.ctc.wstx.exc.WstxUnexpectedCharException:
Unexpected character 'E' (code 69) in prolog; expected '<'
at [row,col {unknown-source}]: [1,1]
```

The following example combines the fragments above, and show how to provide a Cúram custom SOAP header using Rampart to encrypt it:

```
import wsconnector.MyServiceStub;
import java.io.File;
import java.net.URL;
import org.apache.axiom.om.impl.builder.StAXOMBuilder;
import org.apache.axiom.om.OMAbstractFactory;
import org.apache.axiom.om.OMElement;
import org.apache.axiom.om.OMFactory;
import org.apache.axiom.om.OMNamespace;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;
import org.apache.axis2.context.ConfigurationContext;
import org.apache.axis2.context.ConfigurationContextFactory;
import org.apache.axis2.transport.http.HTTPConstants;
import org.apache.neethi.Policy;
import org.apache.neethi.PolicyEngine;
import org.apache.rampart.RampartMessageData;

...

/**
 * Invoke a web service with encrypted credentials.
 *
 */
public void webserviceClient() {

    final String serviceName = "myService";
    final String operationName = "myOperation";

    // Instantiate the stub.
    final MyServiceStub stub =
        new MyServiceStub();

    // Get the end point of the service and convert it to a URL
    final Options options = stub._getServiceClient().getOptions();
    final EndpointReference eprTo = options.getTo();
    final URL urlOriginal = new URL(eprTo.getAddress());

    // Use that URL,
    // plus our service name to construct a new end point.
    final URL urlNew = new URL(
        urlOriginal.getProtocol(),
        urlOriginal.getHost(),
        urlOriginal.getPort(),
        "/CuramWS2/services/" + serviceName);
    final EndpointReference endpoint =
        new EndpointReference(urlNew.toString());

    // Load configuration.
    final ConfigurationContext ctx = ConfigurationContextFactory.
        createConfigurationContextFromFileSystem(
            "C:/Axis2/client", // Looks for modules, etc. here.
            null); // Configuration provided via API engaging rampart.

    final ServiceClient client = new ServiceClient(ctx, null);

    // Set the credentials - illustrated as an example earlier
    setCuramCredentials(client, "tester", "password");

    // Set the operation in the endpoint.
    options.setAction("urn:" + operationName);
    options.setTo(endpoint);
```

```
// Set client timeout to 30 seconds for slow machines.
options.setProperty(
    HTTPConstants.SO_TIMEOUT, new Integer(30000));
options.setProperty(
    HTTPConstants.CONNECTION_TIMEOUT, new Integer(30000));

// Load the Rampart policy file.
final StAXOMBuilder builder =
    new StAXOMBuilder("C:/Axis2/client" + File.separator
        + "policy.xml");
final Policy policy =
    PolicyEngine.getDocumentElement();
options.setProperty(RampartMessageData.KEY_RAMPART_POLICY,
    policy);
client.setOptions(options);
```

The following shows an equivalent technique for setting the security parameters programmatically, although it is deprecated, it would replace the block of code commented "Load the Rampart policy file" in [Coding the client on page 40](#), above as well as the related policy file:

```
final OutflowConfiguration outConfig =
    new OutflowConfiguration();
outConfig.setActionItems("Signature Encrypt");
outConfig.setUser("tester");
outConfig.
    setPasswordCallbackClass("my.test.ClientPwCallback");
outConfig.
    setSignaturePropFile("client-crypto.properties");
outConfig.setSignatureKeyIdentifier(
    WSSHandlerConstants.BST_DIRECT_REFERENCE);
outConfig.setEncryptionKeyIdentifier(
    WSSHandlerConstants.ISSUER_SERIAL);
outConfig.setEncryptionUser("admin");

final InflowConfiguration inConfig =
    new InflowConfiguration();
inConfig.setActionItems("Signature Encrypt");
inConfig.
    setPasswordCallbackClass("my.test.ClientPwCallback");
inConfig.setSignaturePropFile("client-crypto.properties");

//Set the rampart parameters
options.setProperty(WSSHandlerConstants.OUTFLOW_SECURITY,
    outConfig);
options.setProperty(WSSHandlerConstants.INFLOW_SECURITY,
    inConfig);
```

Figure 17: Sample Client Code (Deprecated) for Setting the Client Security Configuration

Here is an sample working axis2 client descriptor that provides the functionality to send a soap request message using Rampart, with UserNameToken, wsse-Timestamp, Signing, Encryption:

```

Use the following code snippet in the axis2 java client to load the axis2 client
descriptor.
ConfigurationContext ctx = ConfigurationContextFactory
.createConfigurationContextFromFileSystem(
"base directory under which the axis2 modules are present",
// pass the absolute path of your client axis2 descriptor, as this is very important.
"absolute path of your client-axis2.xml");

Sample client-axis2.xml
-----
<?xml version="1.0" encoding="UTF-8"?>

<axisconfig name="AxisJava2.0">

    ref="rampart" />

    <parameter name="OutflowSecurity">
        <action>
            <items>UsernameToken Timestamp Signature Encrypt</items>
            <!-- encryption user is the certificate alias , that is present in
keystore -->
            <encryptionUser>verisignsecondarycert</encryptionUser>
            <!-- the username that is passed in username token-->
            <user>scmca</user>
            <!-- the client password callback class ,
where at runtime, the username and password can be manipulated-->
            <passwordCallbackClass>curam.mm.validation.service.impl.MMClientPWCallback
</passwordCallbackClass>
            <!-- the client crypto property file that provides the keystore
related information for the axis2 client engine-->
            <signaturePropFile>client-crypto.properties</signaturePropFile>
            <signatureKeyIdentifier>DirectReference</signatureKeyIdentifier>
        </action>
    </parameter>

    <parameter name="InflowSecurity">
        <action>
            <items>Signature Encrypt</items>
            <signaturePropFile>client-crypto.properties</signaturePropFile>
            <passwordCallbackClass>
curam.mm.validation.service.impl.MMClientPWCallback</passwordCallbackClass>
            <signatureKeyIdentifier>DirectReference</signatureKeyIdentifier>
        </action>
    </parameter>

    <!-- ===== -->
    <!-- Message Receivers -->
    <!-- ===== -->
    <!--This is the Default Message Receiver for the system ,
if you want to have MessageReceivers for -->
    <!--all the other MEP implement it and add the correct entry
to here , so that you can refer from-->
    <!--any operation -->
    <!--Note : You can override this for particular
service by adding the same element with your requirement-->
    <messageReceivers>
        <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
class="org.apache.axis2.receivers.RawXMLINOnlyMessageReceiver"/>
        <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>
    </messageReceivers>

```

```

    <!-- ===== -->
    <!-- Transport Outs -->
    <!-- ===== -->

```

```

<transportSender name="http"

```

```

class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
    <parameter name="PROTOCOL" locked="false">HTTP/1.1</parameter>
    <parameter name="Transfer-Encoding" locked="false">chunked</parameter>
</transportSender>

```

Securing Web Service Network Traffic with HTTPS/SSL

HTTPS/SSL might be a part of your web services security strategy. Details about setting up HTTPS/SSL are beyond the scope of this document. However, the use of HTTPS/SSL can be established in either of the following ways.

- **Application server environment**

Setting up this environment is specific to your particular application server, but essentially involves exporting the appropriate server certificates and making them available to your client environment.

- **Rampart WS-Security policy**

There are a number of web articles that cover this subject in more detail.

For client access, the end point must reflect the protocol and port change, which can be done dynamically at run time. Client code such as the following example, can change the endpoint:

```
// stub is a previously obtained service stub.
// nHttpsPort is an integer identifying the HTTPS port of
// your application server.
// serviceName is a String identifying the service name.

ServiceClient client = stub._getServiceClient();

// Get the end point of the service and convert it to a URL

final Options options = stub._getServiceClient().getOptions();
final EndpointReference eprTo = options.getTo();
final URL urlOriginal = new URL(eprTo.getAddress());

// Use that URL, plus our service name to construct
// a new end point.

final URL urlNew = new URL("https", urlOriginal.getHost(),
nHttpsPort, "/CuramWS2/services/" + serviceName);
client.setTargetEPR(new EndpointReference(urlNew.toString()));
```

Figure 19: Example of Dynamically Changing the Web Service End Point

Your client needs to identify the keystore and password that contains the necessary certificates, for example:

```
System.setProperty("javax.net.ssl.trustStore",
    "C:/keys/server.jks");
System.setProperty("javax.net.ssl.trustStorePassword",
    "password");
```

Otherwise, client coding for HTTPS is similar to that of HTTP.

Note: In a WebSphere environment, the SSL socket classes are not available by default and you might experience this error:

```
org.apache.axis2.AxisFault: java.lang.ClassNotFoundException:
    Cannot find the specified class
    com.ibm.websphere.ssl.protocol.SSLSocketFactory
```

Resolve this error with code like the following:

```
Security.setProperty("ssl.SocketFactory.provider",
    "com.ibm.jsse2.SSLSocketFactoryImpl");
Security.setProperty("ssl.ServerSocketFactory.provider",
```

```
"com.ibm.jsse2.SSLServerSocketFactoryImpl");
```

Creating Keystore Files

For secure web service configuration, you can create the *server.keystore* and *client.keystore* files.

- Generate the server keystore in file *server.keystore*:

```
%JAVA_HOME%/bin/keytool -genkey -alias curam-sv -dname
"CN=localhost, OU=Dev, O=Curam, L=Dublin, ST=Ireland, C=IRL"
-keyalg RSA -keypass password -storepass password -keystore
server.keystore
```

- Export the certificate from the keystore to an external file *server.cer*:

```
%JAVA_HOME%/bin/keytool -export -alias curam-sv -storepass
password -file server.cer -keystore server.keystore
```

- Generate the client keystore in file *client.keystore*:

```
%JAVA_HOME%/bin/keytool -genkey -alias beantester -dname
"CN=Client, OU=Dev, O=Curam, L=Dublin, ST=Ireland, C=IRL" -
keyalg RSA -keypass password -storepass password -keystore
client.keystore
```

- Export the certificate from the client keystore to external file *client.cer*:

```
%JAVA_HOME%/bin/keytool -export -alias beantester -storepass
password -file client.cer -keystore client.keystore
```

- Import the server's certificate into the client's keystore:

```
%JAVA_HOME%/bin/keytool -import -v -trustcacerts -alias curam -
file server.cer -keystore client.keystore -keypass password -
storepass password
```

- Import the client's certificate into the server's keystore:

```
%JAVA_HOME%/bin/keytool -import -v -trustcacerts -alias curam -
file client.cer -keystore server.keystore -keypass password -
storepass password
```

1.5 Inbound Web Service Properties: *ws_inbound.xml*

Use this information to learn about the name/value pairs in the *ws_inbound.xml* property file, which are used to build *services.xml* descriptor files for a web service. These files are generated by default, but you can also customize them.

Property Settings

The following default properties are produced by the Cúram generator:

- **classname**
The fully qualified name of the web service class, from the IBM® Rational® Software Architect Designer model. This property should never be overridden and should always be provided by the generator.
- **ws_binding_style**
The web service binding style, based on the Rational® Software Architect Designer class property `WS_Binding_Style`. Values: `document` (default) or `rpc`.
- **ws_is_xml_document**
Indicator of a service class whose operations process W3C Documents, based on the Rational® Software Architect Designer class property `WS_Is_XML_Document` property. This property should always be determined by the generator. Values: `true` or `false` (default).

An example `ws_inbound.xml` property file that the generator would create is shown.

```
<curam_ws_inbound>
  <classname>my.util.component_name.remote.WSClass</classname>
  <ws_binding_style>document</ws_binding_style>
  <ws_is_xml_document>>false</ws_is_xml_document>
</curam_ws_inbound>
```

The following are the properties that can be provided and/or customized through a custom `ws_inbound.xml` property file:

- **ws_binding_style**
The web service binding style. This property has no direct dependency on the Rational® Software Architect Designer model. It is used for passing the corresponding argument to the Apache Axis2 Java2WSDL tool. See also the description of the `ws_binding_use` property below.

Values: `document` (default) or `rpc`.
- **ws_binding_use**
The web service binding use. It is used for passing the corresponding argument to the Axis2 Java2WSDL tool.

Values: `literal` (default) or `encoded`.
- **ws_service_username**
A username (see `ws_service_password` below) to be used for authentication by the Cúram receiver. Not set by default as the default is to utilize a custom SOAP header for specifying authentication credentials. If specified, results in the corresponding descriptor parameter in `services.xml` being set.

Values: A valid Cúram user.
- **ws_service_password**
A password (see `ws_service_username` above) to be used for authentication by the Cúram receiver. Not set by default as the default is to utilize a custom SOAP header for specifying authentication credentials. If specified, results in the corresponding Axis2 descriptor parameter in `services.xml` being set.

Values: A valid password for the corresponding Cúram user.

- **ws_client_must_authenticate**
An indicator as to whether custom SOAP headers are to be used for Cúram web service client authentication. Should not be specified with `ws_service_username` and `ws_service_password` (above), but if specified this setting overrides, causing the credentials in those properties to be ignored. If specified, results in the corresponding Axis2 descriptor parameter in `services.xml` being set.

Values: `true` (default) or `false`.

- **ws_disable**
An indicator as to whether this web service should be processed by the build system for generating and packing the service into the WAR file. Typically you would use this to temporarily disable a service from being built and thus exposed.

Values: `true` or `false` (default).

An example, custom `ws_inbound.xml` property file is shown.

```
<curam_ws_inbound>
  <ws_binding_style>document</ws_binding_style>
  <ws_client_must_authenticate>false</ws_client_must_authenticate>
  <ws_service_username>beantester</ws_service_username>
  <ws_service_password>password</ws_service_password>
</curam_ws_inbound>
```

When providing a custom `ws_inbound.xml` properties file place the file in your `components/custom/axis/<service_name>` directory (the `<service_name>` and class name must match). During the build the properties files are combined based on the following precedence order:

- Your custom `ws_inbound.xml` properties file
- The generated `ws_inbound.xml` properties file
- The default values for the properties

1.6 Deployment Descriptor File: `services.xml`

Each web service class requires an Axis2 deployment descriptor file called `services.xml`. Use this information to learn about the `services.xml` file.

Descriptor File Contents

The Cúram build automatically generates a suitable deployment descriptor for the default settings that are described in [Inbound Web Service Properties File on page 25](#) and [1.5 Inbound Web Service Properties: `ws_inbound.xml` on page 46](#). The format and contents of the `services.xml` are defined by Axis2. For more information, see the *Apache Axis2 Configuration Guide* (<http://axis.apache.org/axis2/java/core/docs/axis2config.html>).

Based on the settings from the `ws_inbound.xml` property files the `app_webservices2.xml` script generates a `services.xml` file for each web service class. This descriptor file contains a number of parameters that are used at runtime to define and identify the web service and its behavior.

```
<serviceGroup>
```

```

<service name="ServiceName">

<!-- Generated by app_webservices2.xml -->
<description>
  Axis2 web service descriptor
</description>

<messageReceivers>
  <messageReceiver
    mep="http://www.w3.org/2004/08/wsdl/in-out"
    class=
      "curam.util.connectors.axis2.CuramXmlDocMessageReceiver" />
  <messageReceiver
    mep="http://www.w3.org/2004/08/wsdl/in-only"
    class=
      "curam.util.connectors.axis2.CuramInOnlyMessageReceiver" />
</messageReceivers>

  <parameter
    name="remoteInterfaceName">
    my.package.remote.ServiceName</parameter>
  <parameter
    name="ServiceClass" locked="false">
    my.package.remote.ServiceNameBean</parameter>
  <parameter
    name="homeInterfaceName">
    my.package.remote.ServiceNameHome</parameter>
  <parameter
    name="beanJndiName">
    curamejb/ServiceNameHome</parameter>

  <parameter
    name="curamWSClientMustAuthenticate">
    true</parameter>

  <parameter
    name="providerUrl">
    iiop://localhost:2809</parameter>
  <parameter
    name="jndiContextClass">
    com.ibm.websphere.naming.WsnInitialContextFactory
  </parameter>

  <parameter
    name="useOriginalwsdl">
    false</parameter>
  <parameter
    name="modifyUserWSDLPortAddress">
    false</parameter>

  <!--
NOTE: For any In-Only services (i.e. returning void) you must
explicitly code those operation names here as per:
http://issues.apache.org/jira/browse/AXIS2-4408
For example:
  <operation name="insert">
    <messageReceiver
      class="curam.util.connectors.axis2.

```

```

        CuramInOnlyMessageReceiver" />
    </operation>
-->

</service>
</serviceGroup>

```

Figure 20: Sample Generated services.xml Descriptor File

The following lists the mapping of the *services.xml* parameters to the settings in your build environment:

- **messageReceiver**

Specifies the appropriate receiver class for the MEPs of the service. For Cúram there are three available settings/classes:

- `curam.util.connectors.axis2.CuramXmlDocMessageReceiver` - For service classes that process W3C Documents as arguments and return values.
- `curam.util.connectors.axis2.CuramMessageReceiver` - For service classes that process Cúram classes and use the in-out MEP.
- `curam.util.connectors.axis2.CuramInOnlyMessageReceiver` - For service classes that process Cúram classes and use the in-only MEP.

This value is set by the *app_webservices2.xml* script as per the description above. (Required)

- **remoteInterfaceName, ServiceClass, homeInterfaceName, beanJndiName**

Specify the class names and JNDI name required by the receiver code for invoking the service via the facade bean.

These values are set by the *app_webservices2.xml* script based on the generated classname value in the *ws_inbound.xml* properties file. (Required)

- **curamWSClientMustAuthenticate, jndiUser, jndiPassword**

Specify credential processing and credentials for accessing the operations of the web service class.

These are set by the *app_webservices2.xml* script based on the corresponding properties in *ws_inbound.xml* (see [Inbound Web Service Properties File on page 25](#)). Default for `curamWSClientMustAuthenticate` is `true`, but can be overridden at runtime by custom receiver code. (Optional)

- **providerUrl, jndiContextClass**

Specify the application server-specific connection parameters.

These values are set by the *app_webservices2.xml* script based on your *AppServer.properties* settings for your `as.vendor`, `curam.server.port`, and `curam.server.host` properties. Can be set at runtime by custom receiver code. (Optional)

- **useOriginalwsdl, modifyUserWSDLPortAddress**

Specify the processing and handling of WSDL at runtime.

These are explicitly set to `false` by the *app_webservices2.xml* script due to symptoms reported in, for instance, Apache Axis2 JIRA: *AXIS2-4541*. (Required for proper WSDL handling.)

1.7 Troubleshooting

Troubleshoot Axis2 web services. The tips and techniques are not new or comprehensive, but help you to consider options and ways to increase the effectiveness of your web services.

You modeled your web services, developed your server code, built and deployed your application, and your web service EAR files. Finally, you are now ready to begin testing and delivering your web service.

Axis2 represents a complex set of software and third-party products, especially when viewed from the perspective of running in an application server environment. While the Cúram environment simplifies many aspects of web service development, the final steps of testing and debugging your services can be daunting.

Initial Server Validation and Troubleshooting

Because web services process through many layers, one effective technique for more quickly identifying and resolving problems is to keep the server and client side of your service testing separate.

When deployed, first focus your testing on the server side to ensure everything works properly and then introduce your client development and testing so that you know where to focus to resolve errors.

If this is your first deployment of a web service, did the application server and deployed application EAR or WAR files start without errors? If not, investigate and resolve these errors.

If your application starts successfully the next step is to ensure that your service is available. This is done differently for Axis2. But, in general, it involves entering the web service URL with the `?wsdl` argument to verify that your service can be accessed. Details for validating the Axis2 environment are in the sections following.

Validating Your Axis2 Environment

To use the Axis2 web app, first you must download it from Apache.

Follow the instructions in [1.8 Including the Axis2 Admin Application in Your Web Services WAR File on page 56](#) for details about how to include this application in your environment. Axis2 provides an initial validation step that is provided by its built-in validation check. You invoke this by entering the URL for your Axis2 web service application as defined by your web services application root context and application server port configuration. For instance, this might look like: `http://localhost:9099/CuramWS2/axis2-web/index.jsp`. This page brings up the Welcome! page with an option to validate your environment, which you should select. Out of the box, the only error you should see on the resulting page is in the Examining Version Service section where it warns you about not having the sample Apache Axis2 version web service.

You can rectify this error (which is not really an error, but a good validation check) by including that service as external content when you build your Axis2 web services WAR/EAR file. For more information, see [Building and Packaging Web Services on page 22](#).

After you successfully validate your Axis2 environment, click the Back Home link on that page and select the Services link on the Welcome! page. The resulting Available services page lists all available services (classes) and their operations. If there is an invalid service, (for example, due to a missing implementation class), it is flagged in more detail and you need to use the diagnostics provided to resolve any errors. For all valid services, selecting a service name link from the Available services page generates and displays the WSDL for that service. This verifies your deployed services and it is now be available for invocation.

Key Points to be Aware of

Key points to be aware of when you validate your Axis2 environment.

- On the **Available services** page, you might see the operation `setSessionContext`, which you did not model and code. This behavior relates to the issue that is described in [Modeling and Implementing an Inbound Web Service on page 19](#) and in the *Cúram Release Notes*. It has no impact and can be ignored.
- The WSDL generated from the "Available services" links is not equivalent to the WSDL generated by the Axis2 Java2WSDL tool. Use the Java2WSDL tool to develop outbound web services. You can find the Java2WSDL tool in the `build/svr/wsc` directory of your development environment after a web services EAR file build.
- Axis2 has capabilities for checking, investigating your environment using its external administration web application (the Administration link on the Welcome! page). See [1.8 Including the Axis2 Admin Application in Your Web Services WAR File on page 56](#) for details on including this application in your environment. If you don't explicitly build or include this application, the functionality is not available.

Using an External Client to Validate and Troubleshoot

Begin validating the service on the server side first by using an external client because unless the web service class exists, deployment is set up properly. A client failure might not be clearly distinguishable.

To keep the path length and areas, you might have to investigate for possible errors as small as possible. Use a known, working client to start your service. Common areas of failure that a known, working external client can help validate include: service packaging, receiver processing, security configuration, and implementation processing. An example of an external client you might use is the freely available soapUI client (www.soapui.org), which is relatively easy and fast to set up and begin using. While a detailed treatment of soapUI is beyond the scope of this document the following is an outline of the steps you would use, which are similar for Axis2:

- Download, install, and start soapUI.
- When validating your service(s) (above) save the generated WSDL.
- In soapUI select the File menu -> New soapUI Project and in this dialog specify the location of your saved WSDL and click OK. This creates and opens a new soapUI project from where you can invoke your web services.
- From the soapUI tree control, expand your newly created project and expand the "Soap12Binding" or "Soap11Binding". Under this tree branch you will see your service operations and under each operation a "Request 1" (default name) request. Double-clicking the request opens a request editor. In the left pane you must code your SOAP message (e.g.

parameters, etc.) and a template is provided for doing this. In the right pane is where the result is displayed. Once you've coded your SOAP message click the right green arrow/triangle in the tool bar to execute the service. If you've coded the SOAP message correctly the service output will be displayed in the right pane. However, if an error occurs there will be error information in this pane. In the event of an error verify your SOAP message syntax and content; also see [Using an External Client to Validate and Troubleshoot on page 52](#) for some further techniques for resolving and addressing these.

Note: For Axis2 you must keep in mind the default security behavior and that you must include the custom SOAP header credentials in your request. This would look something like this:

```
<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:rem="http://remote.my.package">
  <soap:Header>
    <curam:Credentials
      xmlns:curam="http://www.curamsoftware.com">
      <Username>beantester</Username>
      <Password>password</Password>
    </curam:Credentials>
  </soap:Header>
  <soap:Body>
  ...
  </soap:Body>
</soap:Envelope>
```

Note: For Axis2 the first access of a web service may timeout due to the large number of jar files and processing done at first initialization. This can easily be mitigated in a Java client (e.g. see [Creating a Client and Starting the Web Service on page 15](#)), but for soapUI you can just re-invoke the service and the subsequent request will likely not timeout; otherwise, see [Troubleshooting Axis2 errors on page 53](#) for further techniques for resolving and addressing general web services errors.

Troubleshooting Axis2 errors

Troubleshoot errors with Axis2 web services. The tools available might vary by operating system and application server environment.

To understand why a service fails, consider the following situations:

- To debug axis2 soap request/response messages effectively, add the following as JVM Arguments in Eclipse (for axis2 Java clients) or pass it in the IBM® WebSphere® Application Server console, JVM Process definition VM Arguments, or as JAVA_OPTS for Oracle

WebLogic Server. A server restart is required if applied in the application server. Use this step for debugging only.

```
-Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
-Dorg.apache.commons.logging.simplelog.showdatetime=true
-Dorg.apache.commons.logging.simplelog.log.httpclient.wire=debug
-Dorg.apache.commons.logging.simplelog.log.org.apache.commons.httpclient=debug
-Dorg.apache.commons.logging.simplelog.log.org.apache.axis2=debug
```

This Java™ system property enables console logging of the soap request that is sent and soap response received. This is another alternative to using soap monitor.

- Use a monitoring tool (Apache TCPMon or SOAP Monitor) to view the SOAP message traffic. It's easier to set up TCPMon (download from <http://ws.apache.org/commons/tcpmon>, extract, and run; also, available within soapUI), but it requires changing your client end points or your server port. When set up, SOAP Monitor doesn't require any client or server changes, but does require special build steps for your WAR/EAR files. Apache includes SOAP Monitor as an Axis2 module and see [1.9 Including the Axis2 SOAP Monitor in Your Web Services WAR File on page 57](#) on how to include this in your built Axis2 environment.
- Look at the failure stack trace and investigate any messages there. Try to understand where in the processing the error occurred. The following example Apache Log4j 2 properties file logs verbosely in a `C:\Temp\axis2.log` file, you can adjust these settings to suit your requirements.

```
# set root logging level
rootLogger.level=DEBUG

appenders=LOGFILE,CONSOLE

# bind appenders to the root logger
rootLogger.appenderRefs=ref1,ref2
rootLogger.appenderRef.ref1.ref=CONSOLE
rootLogger.appenderRef.ref2.ref=LOGFILE

appender.CONSOLE.type = Console
appender.CONSOLE.name = CONSOLE
appender.CONSOLE.layout.type = PatternLayout
appender.CONSOLE.layout.pattern=[%p] %m%n

appender.LOGFILE.type = File
appender.LOGFILE.name = LOGFILE
appender.LOGFILE.fileName=./temp/axis2.log
appender.LOGFILE.layout.type = PatternLayout
appender.LOGFILE.layout.pattern=%d [%t] %-5p %c %x - %m%n
```

You need to place the `log4j2.properties` somewhere in the class path of the Axis2 WAR file.

- Check the application server logs for more information.
- Turn on Apache Log4j 2 tracing for Axis2, as this gives you the most detailed picture of the web service processing or error at the time of the failure. This tracing can be voluminous so use it with care.
- Turn on the Cúram application Apache Log4j 2 trace to gives you more context for the failure.
- Consider remote debugging the service that is running on the application server by using Eclipse. Consult your application server-specific documentation for setting up this kind of an environment. Remember that if you are setting breakpoints in this kind of environment that timeouts in the client or server are a high probability and appropriate steps should be taken; for

the client see [Creating a Client and Starting the Web Service on page 15](#) and for the server consult your application server-specific documentation for setting timer values.

Note: Application verbose tracing (`trace_verbose`) is the highest level of logging available for tracing with web services because the SDEJ employs a proxy wrapper object for ultra verbose (`trace_ultra_verbose`) tracing to provide detailed logging. Due to the fact that the SDEJ uses reflection for forwarding a web service request to the underlying process class, the use of a proxy wrapper object is not compatible with the web services infrastructure.

Avoiding Use of `anyType`

Avoid using `anyType` in your WSDL because it makes interoperability difficult as both service platforms and any client platforms must be able to map, or serialize or deserialize the underlying object.

WSDL is typically generated with `anyType` when the underlying data type (for example, object) cannot be resolved.

You might find with Axis2 that your WSDL works with `anyType` because some vendors/platforms map it to, for instance, `java.lang.Object`, which allows it, if it's XML-compliant, to be processed into a SOAP message, and allows processing from XML to a Java object.

Generate your WSDL as early as possible, checking it for the use of `anyType`. In your development, focus on implementing the overall web service structure first and implement the actual service functionality last. For instance, code your web service operations as stubs that merely echo back with minimal processing the input parameters to ensure they can be processed successfully from end to end.

Axis2 Exceptions

Exceptions in web services are returned to the client as an `AxisFault` exception and the message string from the original exception is retained where possible.

For example, client code might look like this:

```
// Processing
...
} catch (final AxisFault a) {
    System.out.println(a.getMessage());
}
```

The structure and contents of the fault SOAP message vary depending on whether the request is a SOAP 1.1 or SOAP 1.2 request. Also, you need to ensure that, depending on the context of the web service client, the web service provides a meaningful exception message. Otherwise, it might not be possible for the handler of the `AxisFault` exception to react. However, sometimes failures occur unexpectedly and you must resolve them along with the application server logs and the Apache Log4j 2 output from Cúram and/or Axis2.

1.8 Including the Axis2 Admin Application in Your Web Services WAR File

Use this information to learn about how to set up your Axis2 web services build to include the Axis2 Administration web application. The Axis2 Administration web application provides useful functionality for working with your Axis2 environment.

Warning: The dynamic functionality of Axis2, for example, hot deployment is not intended for production application server environments such as WebSphere Application Server and WebLogic Server and this functionality should not be attempted in these environments.

Steps to Build

Do not use Axis2 to dynamically modify a production environment. However Axis2 is useful for validating settings, viewing services, and modules. To build your EAR file to include this application:

- Download the Axis2 binary distribution (<http://axis.apache.org/axis2/java/core/download.cgi>) corresponding to the supported Apache Axis2 version and unload it to your hard disk. For example, `C:\Downloads\Axis2`.
- Create a location on your disk to contain the necessary Axis2 artifacts, for example:

```
cd C:\
mkdir Axis2-includes
```

- Put the class files `AdminAgent.class` and `AxisAdminServlet.class` in the `C:\Downloads\Axis2\webapp\WEB-INF\classes\org\apache\axis2\webapp\` into a JAR file that you place into the `WEB-INF\lib` directory in your newly created `C:\Axis2-includes` location. For example:

```
mkdir C:\Axis2-includes\WEB-INF\lib
cd C:\Downloads\Axis2\webapp\WEB-INF\classes
jar -uvf C:\Axis2-includes\WEB-INF\lib\WebAdmin.jar
org/apache/axis2/webapp/
```

- Additionally, you might want to add a custom `axis2.xml` descriptor file to a `WEB-INF\conf` folder to change the default credentials. You can copy the existing included `axis2.xml` file to this example location:

```
mkdir C:\Axis2-includes\WEB-INF\conf
copy %CURAMSDEJ%\ear\webservices2\Axis2\conf\axis2.xml
C:\Axis2-includes\WEB-INF\conf
```

- Then, change the existing `userName` and `password` parameters, for example:

```
<parameter name="userName">restricted</parameter>
<parameter name="password">special</parameter>
```

- To secure the username and password, the `axis2.xml` file must be secured in your development and deployed environments without access in the runtime environment to the Axis2 configuration.

- Use the following properties when you start your web services ear target (see [Building and Packaging Web Services on page 22](#)):

```
-Daxis2.include.overwrite=true
-Daxis2.include.location=C:\Axis2-includes
```

- On deployment, access the Administration link using the Axis2 Welcome! page menu. For example: `http://localhost:9082/curamWS2/axis2-web/index.jsp`.

1.9 Including the Axis2 SOAP Monitor in Your Web Services WAR File

Use this information to learn about how to set up your Axis2 web services build to include the Axis2 SOAP Monitor module in your Axis2 web services WAR file. You can also exclude SOAP Monitor from being packed into the `webservices2.war` if needed.

Steps for Building

The SOAP Monitor provides the ability to view SOAP message requests and responses, which is useful in debugging. The SOAP Monitor module is included with the binary distribution of Axis2 and its module (`soapmodule.mar`) is included in the packaging of the `webservices2.war lib` directory during the build. The `web.xml` file that is included with the `webservices2.war` has the necessary entries to support the SOAP Monitor. Complete the following steps to enable this functionality:

1. Create a location on your disk to contain the necessary Axis2 artifacts, for example:

```
cd C:\
mkdir Axis2-includes
```

2. As indicated in the Axis2 documentation, you must place the SOAPMonitor applet classes at the root of the WAR file. For example:

```
cd C:\Axis2-includes
jar -xvf %CURAMSDEJ%\ear\webservices2\Axis2\modules\soapmonitor.mar
org/apache/axis2/soapmonitor/applet/
```

3. Then, use the following properties when you start your web services ear target (`websphereWebServices` or `weblogicWebServices`):

```
-Daxis2.include.overwrite=true
-Daxis2.include.location=C:\Axis2-includes
```

4. The included `axis2.xml` file defines the necessary SOAP Monitor phase elements, but to be functional the following entry needs to be added (similarly to other `module` entries):

```
<module ref="soapmonitor"/>
```

This change can be made to the EAR file before deployment or for WebSphere in the deployed file system.

5. To access the SOAPMonitor, use a URL, for example: *http://localhost:9082/CuramWS2/SOAPMonitor*.
6. Unfortunately the applet does not give much information if there is an issue. If you see the error:

```
"The SOAP Monitor is unable to communicate with the server."
```

Ensure that there is no port conflict. The default that is set in *web.xml* is 5001. If so, change that port.

7. You can change the default port by setting *soap.monitor.port* = *<port number>* in the *AppServer.properties*.

Excluding SOAP Monitor

You can exclude SOAP Monitor from the deployed EAR file by setting the property *exclude.soapmonitor* in the *AppServer.properties* to TRUE.

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those

websites are not part of the materials for this Merative product and use of those websites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

The licensed program described in this document and all licensed material available for it are provided by Merative under terms of the Merative Client Agreement.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Merative, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Merative, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. Merative shall not be liable for any damages arising out of your use of the sample programs.

Privacy policy

The Merative privacy policy is available at <https://www.merative.com/privacy>.

Trademarks

Merative™ and the Merative™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Adobe™, the Adobe™ logo, PostScript™, and the PostScript™ logo are either registered trademarks or trademarks of Adobe™ Systems Incorporated in the United States, and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft™, Windows™, and the Windows™ logo are trademarks of Microsoft™ Corporation in the United States, other countries, or both.

UNIX™ is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.