IBM Cúram Social Program Management

Cúram Web Services Guide

Version 6.0.4

Note

Before using this information and the product it supports, read the information in Notices at the back of this guide.

This edition applies to version 6.0.4 of IBM Cúram Social Program Management and all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this guide is to provide instructions on how to connect *IBM® Cúram Social Program Management* to external applications that have a web service interface, how to make business logic available as web services and how to secure those web services.

1.2 Audience

This guide is intended for developers that are responsible for the interoperability between enterprise applications using web services. It covers all aspects of *IBM Cúram Social Program Management* web service development including modeling, building, securing, deploying, and troubleshooting.

1.3 Prerequisites

The reader should be familiar with web service concepts and their underlying technologies (for instance see Appendix A, *Glossary*), modeling (as described in the *Cúram Modeling Reference Guide*), and developing in an *IBM Cúram Social Program Management* environment (as described in the *Cúram Server Developer's Guide*).

IBM Cúram Social Program Management web services are based on *Apache Axis2*. The following is a starting point if you require more information: http://axis.apache.org/axis2/java/core/index.html. This site contains a wealth of information including references to underlying technologies such as SOAP and WSDL (see Appendix A, *Glossary*) as well as *Axis2* documentation and links to outside articles, etc.

Chapter 2

Using Web Services

2.1 Overview of Web Services

The term web services describes a standardized way of integrating web-based applications. They allow different applications from different sources to communicate with each other and because all communication is in XML, web services are not tied to any one operating system or programming language. This application-to-application communication is performed 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 *IBM Cúram Social Program Management* infrastructure for development and deployment as described below:

Outbound Web Service Connector

An outbound web service connector allows the *IBM Cúram Social Program Management* application to access external applications that have exposed a web service interface. The WSDL file used to describe this interface is used by the web service connector functionality in *IBM Cúram Social Program Management* 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. See Chapter 3, *Outbound Web Service Connectors* for details on developing outbound web service connectors.

Inbound Web Service

Some functionality within the *IBM Cúram Social Program Management* application can be exposed to other internal or external applications within the network. This can be achieved using an inbound web service. The *IBM Cúram Social Program Management* infrastructure will generate the necessary deployment artifacts and package them for deployment. Once the application EAR file is deployed any application that wishes to communicate with the *IBM Cúram Social Program Management* application will have to 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. See Chapter 4, *Inbound Web Services* for details on developing *IBM Cúram Social Program Management* inbound web services.

2.2 Web Service Platforms

The platforms (a.k.a. stacks) supported for web services are *Apache Axis2* and *Apache Axis* 1.4, for legacy *IBM Cúram Social Program Management* web services.

Legacy web services represent an older generation of web services support in *IBM Cúram Social Program Management*, that is no longer actively maintained by Apache and is thus not viable as a technology base going forward. This feature should only be used if you have a pre-existing *IBM Cúram Social Program Management* web service that utilizes *Axis* 1.4.

Legacy web services are still supported in *IBM Cúram Social Program Management*, but since Apache is not actively maintaining this older *Axis* 1.4 software it is strongly recommended that you begin using the new *Axis2*-based infrastructure for your web services and begin converting any existing legacy web services. Unfortunately, Apache does not provide any specific migration path between their older and newer *Axis2* web service platforms.

There are also a number of other web service platforms available besides *Axis2* (or *Axis* 1.4) that you could potentially adapt for use with *IBM Cúram Social Program Management* ; however, some of the benefits of *Axis2* web services include:

- Complete redesign of Apache *Axis* 1.4 - *Axis2* represents a complete redesign of the Apache *Axis* 1.4 web service engine, which allows for significant improvements in flexibility due to the new architecture and improved performance. Performance improvements come, in part, from a change in XML parser changes using the StAX API, which gives greater speed than SAX event-based parsing that is used in the previous web services implementation.
- New message types available - This third generation of web service support makes new message exchange patterns (MEPs) available. Rather than just in-out processing, in-only (a.k.a. fire-and-forget) and other

MEPs are now available.

- Support for new and updated standards such as SOAP (1.2 & 1.1) and WSDL (2.0 & 1.1) - you will see that the *Axis2* distribution included with *IBM Cúram Social Program Management* includes many new and updated jar files.

2.3 Types of Web Services

Web services can be categorized in a number of ways, one of the main groupings is the web service style and use, which determines the way web service operation parameters are handled. The following table summarizes the *Axis2* (and *Axis* 1.4) offerings in this area.

The `style` option (as per the WSDL specification) determines the structure of the SOAP message payload, which 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 very flexible and can process parameters and return data, or via *IBM® Rational® Software Architect* modeling, can be a W3C Document 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; i.e., <soap:body> and may only contain one element, named after the operation, and all parameters must be represented as sub-elements of this wrapper element. Typically this would be parameters and return values.

Regardless of the choice of style the contents of the SOAP message payload could 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 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> 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, etc. should 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>
```

In general the `encoded` use option has been 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 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, of course, the style must be "document" in the WSDL binding section. Based on the capabilities of Apache *Axis2* (and *Axis* 1.4) only the "wrapped" pattern is supported[1]; however, it is not supported by WSDL 2.0. The following WSDL fragment illustrates this pattern 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>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:operation>
...
```

The following table shows the various style and use combinations supported with *IBM Cúram Social Program Management.*

Style/Use	Cúram with *Axis2*	Cúram with *Axis* 1.4
RPC/Encoded	Not supported (not supported by *Axis2*; not WS-I compliant)	Supported
RPC/Literal	Supported	Not supported
Document/Encoded	Not supported (not WS-I compliant)	Not supported (not WS-I compliant)
Document/Literal (wrapped)	Supported	Supported

Table 2.1 Summary of Web Service Style and Use Support

Of the supported style and use combinations there are a number of relative strengths and weaknesses to be considered when defining your web services. These are summarized in the following table.

Style/Use	Strengths	Weaknesses
Document/Literal (wrapped)	• WS-I compliant • No type encoding information • Can validate in a standard way • Operation name in SOAP message	• Very complex WSDL
RPC/Literal (*Axis2* only)	• WS-I compliant • WSDL is straight-forward • Operation name is included in the WSDL • No type encoding information	• Hard to validate the message
RPC/Encoded (legacy only)	• WSDL is straight-forward • Operation name is included in the WSDL	• Not WS-I compliant

Table 2.2 Summary of Web Service Style and Use Strengths and Weaknesses

2.4 Web Services Security

Web service security is an important consideration in your planning, implementation and runtime support of web services to ensure your valuable and sensitive enterprise data remains safe. This security is implemented entirely by the facilities integrated with *Axis2* (or *Axis* 1.4), which includes WS-Security, wss4j, etc. However, with the support of web services with *Axis2* there is now the option (recommended and on by default) of requiring that clients of inbound web services provide credentials via *IBM Cúram Social Program Management* custom SOAP headers.

2.5 Summary

In this chapter some basics of Apache *Axis2* (and *Axis* 1.4) web services have been introduced and how *IBM Cúram Social Program Management* web services correspond to this web service functionality. Remember that while legacy web services are supported for customers who have already deployed them any new development should be done using the new web service functionality now available. As the basis for the latest generation of web service standards, *Axis2* brings improved architecture, performance, and standards support to your web services.

The following chapters provide the details necessary to enable access to web services externally deployed (outbound) and model, build, customize, secure, and deploy business logic as a web service (inbound).

Notes

[1] Since 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.

Chapter 3

Outbound Web Service Connectors

3.1 Overview

A *IBM Cúram Social Program Management* outbound web service connector allows the application to access external applications that have exposed a web service interface. The WSDL file used to describe this interface is used by the web service connector functionality in *IBM Cúram Social Program Management* to generate the appropriate client code (stubs) to connect to the web service.

In this chapter you will learn how to create new and legacy *IBM Cúram Social Program Management* web services:

- Include the WSDL file in your components file system;
- Add the WSDL file location to the outbound web services file;
- Generate the web service stubs;
- Create a client and invoke the web service.

3.2 Getting Started

The process for building outbound connectors is briefly:

1. Include the WSDL file(s) in your components file system

You must have a WSDL file in order to generate client stubs. Once you have the necessary WSDL file(s) you need to store it within the file system of your `EJBServer/components/custom` directory as shown in Example 3.1, *File System Usage For Outbound Web Services*. These WSDL files will be referenced in the following step.

2. Add the WSDL file location(s) to the component `ws_outbound.xml` file

For each component you wish to have outbound web service connectors built you must place a `ws_outbound.xml` file in the `EJBServer/components/custom/axis` directory. The format of this file is described in Section 3.3.2, *Adding the WSDL File Location to the Outbound Web Services File*.

3. Generate stubs

You are now ready to generate the web service stubs by invoking the following build script: `build wsconnector2`

4. Create a client and invoke the web service

To invoke the web service you must create and build a client (e.g. a *Java*® main program) that utilizes the generated stubs to prepare arguments, call the web service, and process the return results.

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
      + 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 as
                                         below
        + <service_name>
          + <service_name>.wsdl        - where you might copy a
                                         WSDL file as pointed to
                                         by ws_outbound.xml
```

Example 3.1 File System Usage For Outbound Web Services

3.3 Building an Outbound Web Service Connector

3.3.1 Including the WSDL File in Your Components File System

Once you have the WSDL file(s) representing the service you wish to access

place them in the file system (usually under source control). You should place the WSDL file(s) in the custom folder under the location represented by your SERVER_DIR environment variable (and that location is specified in ws_outbound.xml, below). Placing your WSDL within this structure will ensure your web services are isolated from *IBM Cúram Social Program Management*-shipped web services. This is shown in Example 3.1, *File System Usage For Outbound Web Services*. The base name of the (root) WSDL file must use the service name.

3.3.2 Adding the WSDL File Location to the Outbound Web Services File

Once your WSDL file(s) is in your file system you need to create (if not already in existence) a ws_outbound.xml file in your component axis directory and update it. The recommended location for this file is: components/custom/axis/ws_outbound.xml.

In that file you identify the location of the WSDL file(s); for example:

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

Example 3.2 Sample ws_outbound.xml File

In the ws_outbound.xml file there is one service entity for each web service, specifying the service name (matching the WSDL file base name) and location (relative to the SERVER_DIR environment variable).

3.3.3 Generating the Web Service Stubs

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 any files in your components/custom/axis directories. See the example file system structure in Example 3.1, *File System Usage For Outbound Web Services*.

When you invoke the *IBM Cúram Social Program Management* build script:

```
build wsconnector2
```

each WSDL file 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.

3.4 Creating a Client and Invoking the Web Service

Invoking the web service and utilizing the generated code depends on your development environment; but, for example, it might include the following steps, assuming the web service has already been deployed and tested:

1. Copy or reference the generated source and class files; e.g. reference in *Eclipse*;
2. Code your client; e.g. *Java* main program. Typically your steps here will include:
   - Instantiate the generated stub class;
   - Optionally, increase the client timeout threshold (especially for a client that might run first after the application server starts);
   - Setup the credentials in the custom SOAP header (see Section 5.3, *Custom SOAP Headers* for more details);
   - Call the stub methods to instantiate objects and set their values for passing to the service;
   - Invoke the service operation;
   - Check the response;
3. Build and test.

Typically the generated stub code provides a number of options to invoke the web service. Following are some sample code fragments to help illustrate that.

The following fragment calls a service 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.
stub._getServiceClient().getOptions().setProperty(
  HTTPConstants.SO_TIMEOUT, new Integer(180000));
stub._getServiceClient().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();
```

Example 3.3 Sample Web Service 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 using a "hand-built" SOAP message, which in this case takes a simple String argument as input:

```
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(); // Avoid exhausting connection pool
 client.cleanup();
```

Example 3.4 Sample Web Service Client Using Generated Stub and Custom Code

3.5 Legacy Outbound Web Service Connectors

3.5.1 Introduction

This section describes legacy outbound web service connectors, which are defined in section Section 2.1, *Overview of Web Services*.

Warning

The use of legacy web services, while still supported, should only be used for existing web services. This is because the underlying implementation, *Axis* 1.4, is not actively maintained by Apache. Legacy web service support will be removed at some point in the future

and you should convert any legacy web services as soon as possible.

3.5.2 Building an Outbound Web Service Connector

Downloading the WSDL Files

WSDL files are treated as source code which is required to build the application. Consequently, the files should be stored locally (preferably version controlled) with the rest of the source code.

The WSDL files must be manually downloaded (or otherwise obtained), as the web service connector functionality does not support accessing the WSDL definition via a remote access mechanism such as UDDI (Universal Description Discovery and Integration) or HTTP. The downloaded files must be placed in the appropriate build structure folder:

`<SERVER_DIR>/components/<component_name>/wsdl/`

Each *IBM Cúram Social Program Management* component may have its own set of web service connectors, so in the above path `<component_name>` should be the name of the component for which the connector is being deployed. It is considered good practice (though not required) to separate different web services into sub directories within `<SERVER_DIR>/components/<component_name>/wsdl/`.

For example:

`<SERVER_DIR>/components/<component_name>/wsdl/account_service/`

`<SERVER_DIR>/components/<component_name>/wsdl/revenue_service/`

A WSDL definition can be spread over several files that reference each other, possibly in some arbitrary directory structure. These references can be resolved as long as the references are relative and the top level directory is under the `<SERVER_DIR>/components/<component_name>/wsdl/` directory.

Registering a Web Service

Each web service for which outbound connectors should be generated must be registered. Registration is a simple process of updating the following file:

`<SERVER_DIR>/project/config/webservices_config.xml.`

The sample `webservices_config.xml` shown in Example 3.5, *Sample webservices_config.xml* below illustrates how to register a web service:

```
<services>
  <service
   location=
     "components/<component_name>/wsdl/some_service/TopLevel.wsdl"
```

```
  />
</services>
```

Example 3.5 Sample webservices_config.xml

The location attribute represents the location of the WSDL file relative to the <SERVER_DIR> directory. Where the WSDL definition is spread over several files in a hierarchical structure, the web service is registered by referencing the top level WSDL definition file in the webservices_config.xml file.

This registration process also provides the ability to turn a particular web service connector on and off (bearing in mind that business code that accesses the connector would obviously be affected by this), by simply adding or removing the entry as required and rebuilding.

The empty webservices_config.xml file shown in Example 3.6, *An empty webservices_config.xml* below is also valid:

```
<services>
  </services>
```

Example 3.6 An empty webservices_config.xml

Note

If your project does not have a <SERVER_DIR>/project/config/webservices_config.xml file, you may create one by following the structure shown above.

Building the Client Stubs

Once the web service has been registered the server build scripts take care of the rest. The *Axis* 1.4 WSDL-to-Java tool generates client stubs based on the registered WSDL files. These *Java* classes will be compiled as part of the server code.

3.5.3 Creating a Client and Invoking the Web Service

Following web service registration and stub generation, developers can access the web service by utilizing the classes produced by the WSDL-to-Java tool. These include the following:

- For each service element in the WSDL file, an interface class (suffixed with Service) and an implementing service locator class (suffixed with ServiceLocator) are generated. The ServiceLocator class creates an instance of the stub class described next.
- For each portType element in the WSDL file, an interface class (suffixed with _PortType) and an implementing web service stub class (suffixed with SoapBindingStub) are generated. The Soap-

`BindingStub` (instantiated by the `ServiceLocator`) provides access to the external web service via invocation of its appropriate methods.

- A *Java* class is also generated for each schema element `type` in the WSDL file, that represents a parameter or a return type, for each web service.

The following example describes how a very simple web service can be invoked from *Java* code.

The following listing is a WSDL extract of a web service which allows the client to query the price of a shoe by providing its size.

```
<wsdl:types>
    <schema targetNamespace="http://DefaultNamespace"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:impl="http://DefaultNamespace"
      xmlns:intf="http://DefaultNamespace"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

     <element name="askShoePriceResponse">
      <complexType>
       <sequence/>
      </complexType>
     </element>

     <element name="askShoePrice">
      <complexType>
       <sequence>
        <element name="myShoeSize" type="xsd:int"/>
        <element name="shoePrice" nillable="true"
          type="impl:ShoePrice"/>
       </sequence>
      </complexType>
     </element>

     <complexType name="ShoePrice">
      <sequence>
       <element name="priceInCents" type="xsd:int"/>
      </sequence>
     </complexType>
    </schema>
   </wsdl:types>


     <wsdl:message name="askShoePriceResponse">
        <wsdl:part element="impl:askShoePriceResponse"
          name="parameters"/>
     </wsdl:message>

     <wsdl:message name="askShoePriceRequest">
        <wsdl:part element="impl:askShoePrice"
          name="parameters"/>
     </wsdl:message>

     <wsdl:portType name="ShoeShop">
        <wsdl:operation name="askShoePrice">
           <wsdl:input message="impl:askShoePriceRequest"
             name="askShoePriceRequest"/>
           <wsdl:output message="impl:askShoePriceResponse"
             name="askShoePriceResponse"/>
        </wsdl:operation>
     </wsdl:portType>

     <wsdl:binding name="yourhost9082SoapBinding"
       type="impl:ShoeShop">
```

```
      <wsdlsoap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="askShoePrice">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="askShoePriceRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="askShoePriceResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="ShoeShopService">
      <wsdl:port binding="impl:yourhost9082SoapBinding"
        name="yourhost9082">
         <wsdlsoap:address location="http://yourhost:9082"/>
      </wsdl:port>
   </wsdl:service>
```

Example 3.7 WSDL for a sample web service

This WSDL will result in a number of *Java* classes being generated.

The following generated *Java* interface represents the client side of the web service. This interface is implemented by the generated `ShoeShopSoapBindingStub`, which the developer will use to invoke the web service.

```
public interface ShoeShop_PortType extends java.rmi.Remote {
    public ShoePrice askShoePrice(int myShoeSize)
      throws java.rmi.RemoteException;
}
```

Example 3.8 Java client stub interface for the web service

Instances of the `ShoeShopSoapBindingStub` class are created by the generated `ShoeShopServiceLocator` class.

The following code shows how this web service can be invoked from *Java* code:

```
  // The service locator gets instances of the web service.
  final ShoeShopServiceLocator serviceLocator =
    new ShoeShopServiceLocator();

  // get an instance of client stub by casting from interface:
  final ShoeShopSoapBindingStub shoeShop =
    (ShoeShopSoapBindingStub) serviceLocator.getShoeshop();

  // this is our in parameter:
  int myShoeSize = 8;

  // ShoePrice is a generated type wrapper:
  final ShoePrice shoePrice = new ShoePrice();

  // invoke the web service:
  shoePrice =
    shoeShop.askShoePrice( myShoeSize );

  // examine the value we got back:
  final int priceInCents = shoePrice.getPriceInCents();
```

Example 3.9 Invoking the web service from Java code

This is a very simple example to illustrate how web services can be accessed by *IBM Cúram Social Program Management* applications. For working with more complex web services the developer should consult the web service WSDL, and refer to the documentation on the Apache Axis website [http://ws.apache.org/axis].

Addressing anyType Serialization/Deserialization Errors

As per Section D.4, *Avoid Use of 'anyType'* you should avoid the use of anyType. This is due to inconsistencies with, or lack of support for, this feature depending on the environment, which may make your web service non-portable. The following information has been found to be effective for addressing one of the issues related to the use of anyType.

With some web services clients the use of the WSDL `anyType` type may cause an `org.xml.sax.SAXException: No deserializer for anyType` error. One way to resolve this error is to register a serializer and deserializer for the `anyType` type. This is illustrated in the code fragment in Example 3.11, *Java client fragment for serialization/deserialization of anyType* where the underlying definition, shown in the WSDL fragment in Example 3.10, *WSDL fragment illustrating use of anyType* is mapped to a *Java* String type. In summary, the processing steps involve:

- Obtaining a reference to the *Axis* 1.4 client configuration engine;
- Passing this reference into the service locator constructor;
- Getting the type mappings registry from the configuration engine;
- Creating a type mapping implementation for `anyType` and adding it to the mappings registry.

```
...
  <xs:element name="SomeRequest" type="com:SomeResult"/>
    <xs:complexType name="SomeResult">
      <xs:sequence>
        <xs:element minOccurs="0"
                        name="anId"
                        type="xs:anyType">
          </xs:element>
      </xs:sequence>
    </xs:complexType>
...
```

Example 3.10 WSDL fragment illustrating use of anyType

```
...
// Get the client engine configuration from the locator
// to enable registration of mappings for this client.
```

```
final AxisEngine engine = locator.getEngine();
final EngineConfiguration clientEngineConfig = engine.getConfig();

// Instantiate a simple serializer and deserializer to
// map between 'anyType' and String.
final QName qnAnyType =
      new QName("http://www.w3.org/2001/XMLSchema", "anyType");
final SimpleSerializerFactory serFact =
      new SimpleSerializerFactory(String.class, qnAnyType);
final SimpleDeserializerFactory deserFact =
      new SimpleDeserializerFactory(String.class, qnAnyType);

// Now register these serializers in the client engine
// configuration. (Note that the engine config will not
// return a valid typeMapping registry until after the
// locator has created a service.)
final TypeMappingRegistry tmReg =
      clientEngineConfig.getTypeMappingRegistry();
final TypeMapping typeMapping = tmReg.getOrMakeTypeMapping("");
typeMapping.register(String.class, qnAnyType,
                     serFact, deserFact);
tmReg.register("", typeMapping);
// The service is now able to handle the anyType data type.

// The remainder of the client method would simply
// invoke the service normally.
...
```

Example 3.11 Java client fragment for serialization/deserialization of anyType

Chapter 4

Inbound Web Services

4.1 Overview

An inbound web service is *IBM Cúram Social Program Management* application functionality that is exposed to other internal or external applications within the network. This chapter describes the infrastructure for supporting these services and the steps necessary to exploit it.

In this chapter you will learn how to create new and legacy *IBM Cúram Social Program Management* web services:

- Model and implement an inbound web service;
- Build and package web services;
- Provide security data for web services;
- Provide web service customizations.

4.2 Getting Started

The process for developing inbound web services is briefly:

1. Model your web service and provide implementation code

You need to define the classes (WS Inbound) and operations in *Rational Software Architect* that you will be implementing to provide the functionality you wish to expose as web services.

As with any *IBM Cúram Social Program Management* process class you need to provide the implementation for the classes and operations you model as per the *Cúram Modeling Reference Guide*.

2. Build your web services and the web services EAR file

The *IBM Cúram Social Program Management* build system will build and package your web services. Use the server and EAR file build tar-

gets as described in the *Cúram Server Developer's Guide* and the deployment guide appropriate to your platform.

3. Provide security data for your web services

By default your web services are not accessible until you: a) Provide security data (see Section 4.5, *Providing Security Data for Web Services*) 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 Section 5.3, *Custom SOAP Headers* (unless you choose to disable this security functionality; see Section 4.6.3.1, *Custom Credential Processing*).

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

Example 4.1 File System Usage For Inbound Web Services

4.3 Modeling and Implementing an Inbound Web Service

See *Working with the Cúram Model in Rational Software Architect* for more information on using the *Rational Software Architect* tool with the Cúram model. Based on your design decisions you will need to model the necessary classes and operations and set the appropriate properties in the Cúram model. As per the normal *IBM Cúram Social Program Management* development process documented in the *Cúram Server Developers Guide* you must also code your web service implementation classes.

When you model your web services consider:

- 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; see Section 2.3, *Types of Web Services* for more information.

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

4.3.1 Creating Inbound Web Service Classes

In *Rational Software Architect* to add an *Axis2* inbound web service class to a package, select Add Class, `WS Inbound` from the right-click context menu and name the class.

Note

In *IBM Cúram Social Program Management* web service names are based on the class name specified in the *Rational Software Architect* model and must be unique within the environment.

If you require passing and returning a W3C Document instead of *IBM Cúram Social Program Management* 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 Section 4.6.4, *Providing Schema Validation*);
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 *IBM Cúram Social Program Management* 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 gener-

ated WSDL would need to be handled in a separate modeled class.

4.3.2 Adding Operations to Inbound Web Service Classes

In *Rational Software Architect* operations are added to *Axis2* inbound web service classes via the right-click context menu. To 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 modeling inbound web services:

- Certain method names on inbound web services will not operate as expected, due to the fact that when handling an inbound web service call *Java* reflection is used to find and invoke methods in your application. The *Axis2* reflection code identifies methods by name only (i.e., not by signature), which 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—i.e., a stateless session bean—to be generated.

  So, in addition to your application methods, this class also contains methods inherited from `javax.ejb.EjbObject`, and possibly others generated by your application server tooling; e.g.: `remove`, `getEJBHome`, `getHandle`, etc.

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

- *Axis2* web services may not use certain operation names that conflict with method names in the `java.lang.Object` or `javax.ejb.EJBObject` classes; e.g. 'remove', 'notifyAll', etc. Because of this behavior the *Axis2* listServices web app page (e.g. http://localhost:9082/CuramWS2/services/listServices) sometimes includes a process, setSessionContext, that is not part of the WSDL or implementation. This operation name comes from `org.apache.axis2.context.MessageContext.setSessionContext(SessionContext)`.

4.3.3 Adding Arguments and Return Types to Inbound Web Service Operations

Arguments and return types are added to inbound web service operations in the same manner as they are added to process and facade classes. However, they are only relevant for classes that don't specify support for W3C Docu-

ments (WS_Is_XML_Document property). For more information on how to add arguments and return types to process classes refer to the relevant sections of: *Working with the Cúram Model in Rational Software Architect*.

4.3.4 Processing of Lists

An operation is said to use *IBM Cúram Social Program Management* lists if its return value or any of its parameters utilize a struct which aggregates another struct using 'multiple' cardinality.

In the UML metamodel, it is possible to model a <<WS_Inbound>> operation that uses parameters containing lists (i.e., a struct that aggregates another struct(s) 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 (i.e. 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.

4.3.5 Data Types

The *IBM Cúram Social Program Management* data types except Blob (SVR_BLOB) can be used in *Axis2* inbound web service operations. The mappings between *IBM Cúram Social Program Management* and WSDL data types are shown in the following table:

Cúram data type	WSDL data type
SVR_BOOLEAN	xsd:boolean
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: yyyymmdd)
SVR_DATETIME	xsd:string (Format: yyyymmddThhmmss)
SVR_FLOAT	xsd:float
SVR_DOUBLE	xsd:double

Cúram data type	WSDL data type
SVR_MONEY	xsd:float

Table 4.1 Cúram to WSDL data types for Axis2

In conjunction with the supported data types shown in Table 4.1, *Cúram to WSDL data types for Axis2*, 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 such 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 consult the documents: *Working with the Cúram Model in Rational Software Architect* and *Cúram Server Modeling Guide*.

Note

Passing or returning the "raw" *IBM Cúram Social Program Management* data types (i.e., "Date", "DateTime", "Money") as an attribute to an *Axis2* web service is restricted. *IBM Cúram Social Program Management* data types must be wrapped inside a struct before passing them as attributes to a web service.

4.4 Building and Packaging Web Services

This section discusses the targets (`websphereWebServices` and `weblogicWebServices`) for building the web services EAR file.

The steps in this build process are:

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) created by the generator:
   - Process the properties in the following order: custom, generator, defaults (see Section 4.6.1, *Inbound Web Service Properties File* for more information);
   - Generate the `services.xml` descriptor file, unless a custom `services.xml` has been provided (see Section 4.6.2, *Deployment Descriptor File* for more information);
   - Package the web service directory.

The following properties and customizations are available:

- Generation of the `webservices2.war` can be turned off by setting 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 either of the following properties:
  - `axis2.include.location` - that points to a directory containing a structure mapping to the 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.

In conjunction with either of the two properties above, setting the `axis2.include.overwrite` property will cause these contents to override the *IBM Cúram Social Program Management*-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 would be to include the sample Version service to enable *Axis2* to successfully validate the environment (see Section D.2.1, *Axis2 Environment Validation*).

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.overwrite`) and packaged into the `webser-`

vice2.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.)

4.5 Providing Security Data for Web Services

In *IBM Cúram Social Program Management* web services are not automatically associated with a security group. This is to ensure that web services are not vulnerable to a security breach. You have to provide security data in order to make your web service usable. As part of your development process you need to ensure that the appropriate security database entries are created. For instance:

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

The contents of the *IBM Cúram Social Program Management* security tables are explained further in the security chapter of *Cúram Server Developer's Guide*.

4.6 Providing Web Service Customizations

Providing customizations at build-time impacts the security and behavior of your web service at runtime. With the default configuration the web services EAR file build will:

- Assign the appropriate *IBM Cúram Social Program Management* message receiver for struct and domain types, for argument and operation return values, or for W3C Documents, based on how you set the WS_Is_XML_Document property in *Rational Software Architect* for the "WS Inbound" (stereotype: <<wsinbound>>) class.
- Expect the web service client to pass a custom SOAP header with authentication credentials in order to invoke the web service.

To change the above default behaviors you will require a custom receiver (see Section 4.6.3, *Customizing Receiver Runtime Functionality* for more information). Additionally, customizations may be necessary for:

- Implementing Web Services Security (*Apache Rampart*) (see Chapter 5, *Secure Web Services* for more information);
- Providing external, non-*IBM Cúram Social Program Management* functionality such as the *Axis2* Admin application and Apache *Axis2* Monitor (see Appendix F, *Including the Axis2 SOAP Monitor in Your Web Services WAR File* for more information);
- Providing other custom parameters for the deployment descriptor (ser-

vices.xml); e.g. doclitBare, mustUnderstand, etc. See the *Apache Axis2* documentation for more information (Apache Axis2 Configuration Guide [http://axis.apache.org/axis2/java/core/docs/axis2config.html]).

In order to be able to effectively customize your web services you should be aware of how *IBM Cúram Social Program Management* processes web services at build time, which is explained in the following sections.

4.6.1 Inbound Web Service Properties File

Based on the web service classes modeled with *Rational Software Architect* the generator creates a folder in the build/svr/gen/wsc2 directory for each web service class modeled. This is shown in Example 4.1, *File System Usage For Inbound Web Services*. (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 we suggest you 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. See Appendix B, *Inbound Web Service Properties - ws_inbound.xml* for details of the property settings in this file.

4.6.2 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 as per Appendix B, *Inbound Web Service Properties - ws_inbound.xml*. 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 we suggest you start with the generated copy that you will find in the build/svr/wsc2/<service_name> directory after an initial build of the web services WAR/EAR file. This is illustrated in Example 4.1, *File System Usage For Inbound Web Services*. Place your custom services.xml file in your components/custom/axis/<service_name> directory (usually under source control). (See Appendix C, *Deployment Descriptor File - services.xml* for details of the contents of this file.) 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-buildingservices.html).

4.6.3 Customizing Receiver Runtime Functionality

The default receivers provided with *IBM Cúram Social Program Management* should be sufficient for most cases; but, you can provide overrides for the following functionality:

- Credentials processing;
- Application server-specific provider URL and context factory parameters;
- SOAP factory provider for W3C Document processing.

These are explained in more detail in the following sections.

Custom Credential Processing

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

By default, *IBM Cúram Social Program Management* web services are built to expect the client to provide credentials via a custom SOAP header and these credentials are then used in invoking the service class operation. The default processing flow is:

- 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 specified in the `services.xml` descriptor file, default credentials are used.

  However, there is no security data generated for web services, so the defaults credentials on their own won't be adequate to enable access to the service (see Section 4.5, *Providing Security Data for Web Services* for information on providing this data).

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 containing 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;
    }
```

Example 4.2 Sample getAxis2Credentials Method

See Section 4.6.3.4, *Building Custom Receiver Code* on how to specify and build this custom class for this method.

Custom Application Server-Specific Parameters

The `app_webservices2.xml` script will generate correct application server-specific provider URL and context factory parameters; however, you may find it convenient if you are supporting multiple environments to derive one or more of these values in your own custom code.

If so, you can provide your own `getProviderURL()` and/or `getContextFactoryName()` method(s) by overriding class `curam.util.connectors.axis2.CuramMessageReceiver`. Both methods return a string representing the provider URL and context factory name, respectively. See Section 4.6.3.4, *Building Custom Receiver Code* on how to specify and build this custom class for these methods.

Custom SOAP Factory

Generally, the default SOAP factory, `org.apache.axiom.soap.SOAPFactory`, should be adequate for processing your web services that process W3C Documents. But, if necessary 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

For any of the above cases of providing custom receiver code you must:

- Extend the appropriate class (e.g. `public class MyReceiver extends ) curam.util.connectors.axis2.CuramMessageReceiver`. (See Section 4.6.2, *Deployment Descriptor File* for the list of receiver classes and their usage.)
- Specify a package name of webservice in your custom *Java* program (e.g.: `package webservice;`).
- Place your custom source code in your components `source/webservice` directory (e.g. `components/mycomponents/source/webservice`). The `server` build target will then build and package this custom receiver code.
- Create a custom `services.xml` descriptor file for each service class to be overridden by your custom behavior. See Section 4.6.2, *Deployment Descriptor File* and Example 4.3, *Sample services.xml Descriptor File Entry for a Custom Receiver* below.

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

Example 4.3 Sample services.xml Descriptor File Entry for a Custom Receiver

The webservices build (implemented in `app_webservices2.xml`) will package these custom artifacts into a WAR file.

4.6.4 Providing Schema Validation

When using web services that pass and return a W3C Document object you may want to use schema validation to verify the integrity of the document you are processing. Whether you choose to do this might depend on factors such as:

- The CPU cost of performing such validation, which is dependent on the volume of transactions your system will encounter;
- 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 some-

where 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 below (Example 4.4, *Sample Illustrating Schema Validation*) illustrates how this 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.
    AppException ae = new
      AppException(SOME_MESSAGES.ERR_SCHEMA_VALIDATION_ERROR,
      e);
  }

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

  return result;
```

```
    }
```

Example 4.4 Sample Illustrating Schema Validation

4.7 Legacy Inbound Web Services

4.7.1 Introduction

This section describes *IBM Cúram Social Program Management* legacy inbound web services, which are defined in section Section 2.1, *Overview of Web Services*.

Warning

The use of legacy web services, while still supported, should only be used for existing web services. This is because the underlying implementation, *Axis* 1.4, is not actively maintained by Apache. Support for legacy web services will be removed at some point in the future and you should convert any legacy web services as soon as possible.

4.7.2 Web Service Styles

The *IBM Cúram Social Program Management* inbound web service functionality supports the generation of RPC-style (Remote Procedure Call) web services and document-oriented web services (DOWS). In both cases:

- The request and response XML messages are transported using SOAP over HTTP.
- Every web service is described using a Web Services Description Language (WSDL) file.
- The invocation scope for all *IBM Cúram Social Program Management* web services is `Request Scope`, the default. For each request to the web service, a new implementation instance is created to handle the request. The service instance will be removed after the request is complete.

4.7.3 SOAP Binding

Web Services are based on an exchange of SOAP XML messages. A SOAP XML message consists of an envelope that contains a `header` and a `body`:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header>
    <!-- header element(s) here -->
```

```
  </soap:Header>
  <soap:Body>
    <!-- body element(s) here -->
  </soap:Body>
</soap:Envelope>
```

The `binding` element of a `WSDL` file describes how the service is bound to the SOAP messaging protocol. There are two possible SOAP binding styles: RPC and Document.

A SOAP binding can also have an encoded use, or a literal use. The use attribute is concerned with how types are represented in the XML messages. It indicates whether the message parts are encoded using some encoding rules, or whether the parts define the concrete schema of the message.

This means there are four potential style/use models. It is generally accepted that it is best practice to avoid using `RPC/Literal` or `Document/Encoded`. Therefore, the following are supported:

- `RPC/Encoded`
- `Document/Literal`

Each style has particular features that dictate how to create a `WSDL` file, and how to translate a WSDL binding to a SOAP message. Essentially these result in different formatting of the SOAP messages.

It is worth noting that document-oriented web services (DOWS) are regarded as a crucial enabling technology for developing solutions incorporating a Service Oriented Architecture (SOA). Document/Literal web services have emerged as the preferred style by bodies such as the Web Services Interoperability Organization (WS-I), an open industry group chartered to promote Web Services interoperability across platforms, applications, and programming languages. DOWS are self-describing web services (conform to XML Schemas) with no reliance on an external encoding (as with RPC). These features promote interoperability between heterogeneous applications, which is a central component for SOA.

4.7.4 Selecting Web Service Style

The developer can decide whether each process class is exposed as a web service using RPC or document style.

The decision of which web service style to employ should be made based on the particular business requirements for the service under development.

From the developer's perspective they are shielded from being concerned with the creation of WSDL files, the binding between XML and *Java*, or the creation of SOAP request and response messages.

Regardless of the style chosen the same programming paradigm will be followed. The underlying code which implements the business logic of the web service will be a typical *IBM Cúram Social Program Management* process

class, with methods that accept struct arguments and return structs.

RPC

RPC is the default web service style in *IBM Cúram Social Program Management*. RPC services follow the SOAP RPC and encoding rules. *Axis* 1.4 is employed to deserialize XML requests messages into *Java* object(s) which are passed to the process class as method arguments, and will serialize the returned *Java* object(s) into XML.

Document (DOWS)

As described above in Section 4.7.3, *SOAP Binding*, the document style does not use SOAP encoding; it's simply based on XML schema. In order to aid effective web service development provision is made for two DOWS types depending on how the operation should process input parameters and return values:

1. "Method parameters" - Where *Axis* 1.4 is used to deserialize the SOAP message for input parameters into *Java* object(s) and to serialize the returned *Java* object(s) into XML.
2. "XML document" - Where the developer has direct access to the XML message contained in the SOAP body, instead of turning it into *Java* objects. In this case an XML document is the input type and return type. An XML schema is exposed in the WSDL to describe the expected input message. Support is also provided for validating SOAP request messages against the specified schema. This is also referred to as a message-style web service.

See Section 4.7.5.1, *Modeling Legacy Web Service Classes in Rational Software Architect* for details on how to model these web service styles.

4.7.5 Creating Inbound Web Services

Legacy web services should only be used by existing *IBM Cúram Social Program Management* customers who have not yet migrated these web services to *Axis2*. Apache has stabilized *Axis* 1.4 and it is not actively maintained.

Modeling Legacy Web Service Classes in Rational Software Architect

A web service class can be created by creating a WebService class (stereotype: <<webservice>>) via the *Rational Software Architect* user interface. For more information on working with the model in *Rational Software Architect* see the *Working with the Cúram Model in Rational Software Architect* document.

A WebService class will:

- Generate DDL that causes the methods of the class to become callable by user 'WEBSVCS'.
- Generate an *Axis* 1.4 configuration file that makes the class available as a web service.

To add an inbound web service class to a package, select Add Class, Web-Service from the right-click context menu and name the class.

A WebService class can support one of the two styles of web services supported by specifying the Document_Type property on the *Curam* property tab for the class:

- RPC - specify the Document_Type property value as 2 - no (which is the default when 0 - unspecified is the value)
- Document-oriented web service (DOWS) - specify the Document_Type property value as 1 - yes

When creating a DOWS you control the processing of operation arguments and return values by specifying the property XML_Document:

- "Method parameters" - specify the XML_Document property value as 2 - no (which is the default when 0 - unspecified is the value)
- "XML document" - specify the XML_Document property value as 1 - yes

Since the above properties apply uniformly to all operations of that class you will need to model different classes when you have web services with different style requirements.

Adding Operations to Legacy Inbound Web Service Classes

Operations are added to inbound web service classes via the right-click context menu. To 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.

Adding Arguments and Return Types to Inbound Web Service Operations

Add arguments and a return type to an inbound web service to utilize an RPC-style web service; otherwise, for a document-style web service, which passes and returns a W3C XML Document, specify no arguments or return type. The interface with an XML Document argument and return type set is automatically generated for each operation, as illustrated in Example 4.5, *Sample Generated DOWS XML Document Interface*.

When adding arguments and return types to RPC-style inbound web service operations this is done in the same manner as with process and facade

classes as documented in *Working with the Cúram Model in Rational Software Architect*.

Note

Only operations which do not have operation arguments or a return type set will be exposed as a DOWS XML Document service. The interface with an XML Document argument and return type set is automatically generated for each operation, as illustrated in the following Example 4.5, *Sample Generated DOWS XML Document Interface*:

```
public interface DOWSXMLDocTestBPO
{
  public org.w3c.dom.Document processDocument
    (org.w3c.dom.Document xmlMessage)
    throws curam.util.exception.AppException,
    curam.util.exception.InformationalException;

  public org.w3c.dom.Document echoDocument
    (org.w3c.dom.Document xmlMessage)
    throws curam.util.exception.AppException,
    curam.util.exception.InformationalException;
}
```

Example 4.5 Sample Generated DOWS XML Document Interface

To specify a schema in the *Curam* properties tab for the WebService class (where you've set property `XML_Document`):

- Select the `XML_Schema` property and click the edit (`...`) button. In the edit window specify the XML schema filename, relative to your EJBServer directory. The schema identified by this filename will be included in the web service WSDL generated at run time by *Axis* 1.4. This feature provides a way to publish a description of the XML message expected by the service.

  Schema files must be stored in the appropriate build structure folder: `<SERVER_DIR>/components/custom/webservices/`. Each component may have its own set of web services, so in the above path `custom` should be the name of the component for which the web service is being deployed. For example:

  `<SERVER_DIR>/components/testComponent/webservices/sampleSchema.xsd`

- Set the `Validate_Request` property in the *Curam* properties tab for the WebService class to `1 - yes`. SOAP body request messages will then be validated against the specified XML schema before forwarding the request to the process class. If the message does not conform to the schema the process class will not be invoked, and a fault message will be returned to the client stating that the request does not conform to the schema.

Note

If the declared elements are referenced in the XML schema then any references to these elements should be qualified with the namespace (in the example Example 4.6, *XML schema* the references are prefixed with `cns`). This is required to avoid the name clashes as the XML schema will be included in the web service WSDL.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ws.curam/FinancialUpdateWS"
  xmlns:cns="http://ws.curam/FinancialUpdateWS"
  elementFormDefault="qualified">
  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1"
        ref="cns:msgT_financialUpdate"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="msgT_financialUpdate">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="cns:row"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="row">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="1"
        ref="cns:EXTERNALILIID"/>
        <xs:element minOccurs="1" maxOccurs="1"
        ref="cns:DOCUMENTSTATUSCODE"/>
        <xs:element minOccurs="1" maxOccurs="1"
        ref="cns:OPENAMOUNT"/>
        <xs:element minOccurs="0" maxOccurs="1"
        ref="cns:EXTERNALINVOICEID"/>
        <xs:element minOccurs="0" maxOccurs="1"
        ref="cns:EXTERNALPAYMENTLIST"/>
        <xs:element minOccurs="0" maxOccurs="1"
        ref="cns:LOCKLIST"/>
        <xs:element minOccurs="1" maxOccurs="1"
        ref="cns:TIMESTAMP"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EXTERNALILIID" type="xs:integer"/>
  <xs:element name="DOCUMENTSTATUSCODE" type="xs:string"/>
  <xs:element name="OPENAMOUNT" type="xs:double"/>
  <xs:element name="EXTERNALINVOICEID" type="xs:integer"/>
  <xs:element name="EXTERNALPAYMENTLIST">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
        ref="cns:EXTERNALPAYMENTID"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="EXTERNALPAYMENTID" type="xs:integer"/>
  <xs:element name="LOCKLIST">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
        ref="cns:LOCKREASON"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="LOCKREASON" type="xs:string"/>
  <xs:element name="TIMESTAMP" type="xs:integer"/>
</xs:schema>
```

Example 4.6 XML schema

An example of an XML document that conforms to the schema in Example 4.7, *XML document*:

```
<msgT_financialUpdate
  xmlns="http://ws.curam/FinancialUpdateWS"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ws.curam/FinancialUpdateWS
  FinancialUpdate.xsd">
 <row>
  <EXTERNALILIID>1</EXTERNALILIID>
  <DOCUMENTSTATUSCODE>C04</DOCUMENTSTATUSCODE>
  <OPENAMOUNT>0</OPENAMOUNT>
  <EXTERNALINVOICEID>000000000000</EXTERNALINVOICEID>
  <EXTERNALPAYMENTLIST>
   <EXTERNALPAYMENTID>233</EXTERNALPAYMENTID>
  </EXTERNALPAYMENTLIST>
  <TIMESTAMP>20080229094755</TIMESTAMP>
 </row>
 <row>
   <EXTERNALILIID>2</EXTERNALILIID>
   <DOCUMENTSTATUSCODE>C05</DOCUMENTSTATUSCODE>
   <OPENAMOUNT>0</OPENAMOUNT>
   <EXTERNALINVOICEID>000000000000</EXTERNALINVOICEID>
   <EXTERNALPAYMENTLIST>
    <EXTERNALPAYMENTID>3455</EXTERNALPAYMENTID>
   </EXTERNALPAYMENTLIST>
   <TIMESTAMP>20080229094744</TIMESTAMP>
 </row>
</msgT_financialUpdate>
```

Example 4.7 XML document

Note

A web service cannot be called if it has validation schema enabled and the specified validation schema xsd file imports other xsd files. This is illustrated by the following examples where validation schema "`Determination.xsd`" imports "`businesstypes.xsd`".

```
<?xml version="1.0"?>

<!-- root element, namespace and form definitions -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:bt="businesstypesURI"
      xmlns:dt="determinationURI"
  elementFormDefault="qualified"
    attributeFormDefault="unqualified"
  targetNamespace="determinationURI">

  <!-- import the businesstypes schema -->
  <xs:import namespace="businesstypesURI"
    schemaLocation="businesstypes.xsd" />

  <...>

</xs:schema>
```

Example 4.8 Determination.xsd

The schema "businesstypes.xsd", which is imported in "Determination.xsd" above:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- The contents of this file are fixed, by the SDEJ.
     It could easily be shipped with the SDEJ -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="curamtypesURI">

  <xs:simpleType name="date">
    <xs:annotation>
      <xs:documentation>Cúram builtin type date.
      </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:date"/>
  </xs:simpleType>

    <...>

</xs:schema>
```

Example 4.9 businesstypes.xsd

When calling a web service having a validation schema which imports another schema as shown in the example above an error "src-resolve: Cannot resolve the name <...> to a type definition component" will be thrown.

There are two workarounds to solve this issue:

- Disable schema validation on a web service.
- If a web service's schema validation still needs to be specified avoid using schema imports within the schema, and define the contents of the imported schema instead. If we take the example given above, the schema "Determination.xsd" shown below no longer imports "businesstypes.xsd", but now has its contents inline:

```
<?xml version="1.0"?>

 <!-- root element, namespace and form definitions -->
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:dt="determinationURI"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified"
   targetNamespace="determinationURI">

   <!-- import the businesstypes schema -->
   <!-- BEGIN businesstypes.xsd -->

     <xs:simpleType name="date">
       <xs:annotation>
         <xs:documentation>
           Curam builtin type date.
         </xs:documentation>
       </xs:annotation>
       <xs:restriction base="xs:date"/>
```

```
    </xs:simpleType>

    <...>

   <!-- END businesstypes.xsd -->

    <...>

</xs:schema>
```

Example 4.10 Determination.xsd

Example 4.11, *Sample DOWS XML Document Implementation Class* illustrates a skeleton implementation. This simple example shows an implementation of class `DOWSXMLDocTestBPO` that consists of two operations, `echoDocument` and `processDocument`, which are exposed as two web services that echo and process the request SOAP body XML message respectively.

```
package webservice.impl;

import curam.util.exception.AppException;
import curam.util.exception.InformationalException;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

public class DOWSXMLDocTestBPO extends
    webservice.base.DOWSXMLDocTestBPO {

  /**
   * Sample method for echoing an XML message.
   *
   * @param xmlMessage The request message
   * @return the response message.
   *
   * @throws AppException
   * @throws InformationalException
   */
  public Document echoDocument
    (final Document xmlMessage)
    throws AppException, InformationalException {

    Document responseMessage = null;
    try {
      responseMessage = xmlMessage;
    } catch (Exception ex) {
      .....
    }
    return responseMessage;
  }

  /**
   * Sample method for processing an XML message.
   *
   * @param xmlMessage The request message
   * @return the response message.
   *
   * @throws AppException
   * @throws InformationalException
   */
  public Document processDocument
    (final Document xmlMessage)
```

```
    throws AppException, InformationalException {

    Document responseMessage = null;
    try {
      responseMessage =
        doXMLDocumentProcessing(xmlMessage);
    } catch (Exception ex) {
      ....
    }
    return responseMessage;
  }

  /**
   * Do processing of XML and return an XML message.
   * @param bodyXML The message to be processed.
   * @return The processed message.
   */
  private Document doXMLDocumentProcessing
    (final Document bodyXML) {

    final Document doc = bodyXML;

    /** business logic implemented here... **/

    return doc;
  }

}
```

Example 4.11 Sample DOWS XML Document Implementation Class

4.7.6 Build and Deployment

The EAR file containing the web service wrapper is built by target **webSphereEAR** (for *WebSphere*) or **weblogicEAR** (for *WebLogic Server*). The resulting EAR file is deployed in the same way as the normal application EAR file.

Note

The web services EAR file may be deployed into a different server to the application server. This server does not require any database or JMS resources.

In order for a third party to use an *IBM Cúram Social Program Management* web service, they need the WSDL which describes the web service. A WSDL document is produced for each webservice process class. The WSDL is generated at run time, and is therefore available only once the web service EAR has been deployed onto an application server. The WSDL is obtained via HTTP from the server. The URL of the WSDL is determined by the following factors:

- The HTTP port of the server onto which the EAR file has been deployed;
- The application name;
- The process class name.

For example, if the application is named Curam and a class named My-

WebServiceBPO has been deployed as a web service, and the web service server is listening on port 9082 on server testserver then the WSDL for this web service can be obtained from http://testserver:9082/CuramWS/services/MyWebServiceBPO?wsdl. In addition, a list of all web services available on this server can be seen by going to http://testserver:9082/CuramWS/services.

Note

The WSDL for a web service is not available at development time. It is generated at run time by *Axis* 1.4 once the web service has been deployed.

4.7.7 Data Types

All the *IBM Cúram Social Program Management* data types except Blob (SVR_BLOB) can be used in RPC and DOWS Method Parameters web services. The mappings between *IBM Cúram Social Program Management* and WSDL data types are shown in the following table:

Cúram data type	WSDL data type
SVR_BOOLEAN	xsd:boolean
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: yyyymmdd)
SVR_DATETIME	xsd:string (Format: yyyymmddThhmmss)
SVR_FLOAT	xsd:float
SVR_DOUBLE	xsd:double
SVR_MONEY	xsd:float

Table 4.2 Cúram to WSDL Data Types (Legacy)

In conjunction with the supported data types shown in Table 4.2, *Cúram to WSDL Data Types (Legacy)*, 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 such 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 an application model. Outbound web services are not affected by this issue. For more details on related modeling topics consult the documents: *Working with the Cúram Model in Rational Software Architect* and *Cúram Server Modeling Guide*.

Note

Passing or returning the "raw" *IBM Cúram Social Program Management* data types (i.e., "Date", "DateTime", "Money") as an attribute to a web service is restricted. *IBM Cúram Social Program Management* data types must be wrapped inside a struct before passing them as attributes to a web service.

Processing of Lists

An operation is said to use *IBM Cúram Social Program Management* lists if its return value or any of its parameters utilize a struct which aggregates another struct using 'multiple' cardinality.

In the UML metamodel, it is possible to model an operation which uses parameters containing lists. All operations which 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 (i.e. parameters which are domain definitions);
- Non-struct operation return types.

In these cases, the web client ignores the operations which it does not support, but these operations can be used as normal as an inbound web service.

When using lists with a document-oriented inbound web service SOAP messages corresponding to the list structs do not match the WSDL corresponding to these types. This will manifest itself as a runtime error when SOAP messages are being serialized or de-serialized. The recommended workaround is to either:

- Use RPC instead of DOWS web services; or,
- Ensure that your DOWS methods do not use list structs as their parameter or return types.

4.7.8 Security Considerations

Once a BPO has been assigned to a `webservice` server it is callable by anybody as a web service without any authentication. All web service calls are automatically logged in and invoked using default credentials. The default user, `WEBSVCS`, automatically gets permission to invoke all methods

of a class which is assigned to a `webservice` server.

Therefore caution is advised when making a class visible as a web service.

4.7.9 Customizations

The *Axis* 1.4 toolkit used operates by listening for SOAP messages on HTTP, and using them - in conjunction with generated parameter structs - to make EJB invocations to the server. To facilitate customization of this behavior, it is possible for the developer to implement a hook which gets called during the process and which has access to the SOAP message. This gives the developer flexibility to do things like perform additional processing of the SOAP message, authenticate with different credentials, specify a locale etc.

By default RPC and DOWS *Method Parameters* web services use the class `curam.util.connectors.webservice.CuramEJBMethodProvider`, and DOWS *XML document* web services use the class `curam.util.connectors.webservice.CuramMsgStyleEJBMethodProvider`. These classes perform some of the processing on the SOAP message and connect to the application using default credentials. When you specify a custom provider for your RPC or DOWS Method Parameters web service class, you must provide an implementation class which extends one of the above classes.

The following rules apply:

- The name of the custom provider class is specified using the `Provider_Name` property in *Rational Software Architect*.
- If a value for this property is specified, you must provide a *Java* implementation of the class. For example, if you set this property to '`MyProvider`' then you must implement a class named `MyProvider` which extends the class `curam.util.connectors.webservice.CuramEJBMethodProvider` or `curam.util.connectors.webservice.CuramMsgStyleEJBMethodProvider`.
- By overriding methods of this class the developer can gain access to the SOAP message and perform additional processing on it. In most cases the developer should also call the `super` version of the overridden method to ensure that the underlying *Axis* 1.4 web service framework continues to work as normal.
- Your provider class implementation must be in a package named `webservice`.
- If you specify a custom provider class for a web service class, the web service no longer automatically connects using the default credentials. Therefore your provider must provide the credentials. Typically these will be obtained from the SOAP message.

- Since the custom provider implementation resides in a different EAR file to the application, its *Java* source must reside in a separate location to the other *Java* source files, e.g. `EJBServer/components/core/source/webservice/MyProvider.java`
- Since the Ear file for web services can be deployed into a dedicated web services server it may not have access to the same services as the main application, such as a database, JMS, etc. However it does have access to the same infrastructure classes such as `curam.util.type.DateTime`, etc.
- The locale for the web service call can be set by setting the `locale` property in the `MessageContext` object for the call. If this property is not set, the locale defaults to that of the user under whose credentials the call is made.

Sample RPC-Style Customizations

In the following code sample, a SOAP message is parsed to extract the username and password credentials, these values are applied to the `MessageContext` class for the invocation (and in the case of *WebSphere*, a login is performed), and control is returned to the superclass. Also a locale is specified for the call by setting property `locale` in the `MessageContext` object. The code sample is followed by an example of a SOAP message which would be processed by the code.

```
package webservice;

import curam.util.connectors.webservice.CuramEJBMethodProvider;
import curam.util.resources.Configuration;
import curam.util.resources.EnvironmentConstants;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.security.PrivilegedAction;
import java.util.Iterator;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginContext;
import javax.xml.soap.Name;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPHeader;
import org.apache.axis.AxisFault;
import org.apache.axis.Message;
import org.apache.axis.MessageContext;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.providers.java.EJBProvider;


/**
 * A web services hook which extends the Axis EJB provider to
 * enable the developer to access the SOAP message. In this case
 * it takes the username and password from the SOAP header and
 * sets them in the method call.
 *
 */
public class TestmodelProvider extends CuramEJBMethodProvider {

  /** The name of an XML attribute in a multi ref element. */
```

```
private static final String kNameOfIdAttribute = "id";

/** The name of an XML element in the SOAP body. */
private static final String kMultiRefElementName = "multiRef";

/**
 * The name of the attribute containing a `href` to another
 * element.
 */
private static final String kHrefAttributeName = "href";

/**
 * The name of the header element as defined in the WSDL file.
 */
private static final String kNameOfHeaderElement = "inHeader";

/** The name of the element containing the user name field. */
private static final String kUsernameFieldName = "userName";

/** The name of the element containing the password field. */
private static final String kPasswordFieldName = "password";

/** Cached Do As Method instance. */
private static Method stSubjectDoAsMethod;

/**
 * Hook which gets the credentials from the header of the soap
 * message and sets them in the message context for the call
 * before delegating back to the superclass method.
 *
 * @param msgContext The message context for the call.
 *
 * @throws AxisFault Generic Axis exception.
 */
public void invoke(final MessageContext msgContext)
throws AxisFault {

  final Message requestMessage = msgContext.getRequestMessage();
  final SOAPEnvelope envelope =
    requestMessage.getSOAPEnvelope();
  final SOAPElement element =
    getSoapElement(envelope, kNameOfHeaderElement);

  String userName = null;
  String password = null;
  try {
    // Get parameters from the SOAP header and set them in the
    // message context for the call.
    userName = getSubElementValue(element, envelope.createName(
      kUsernameFieldName));
    password = getSubElementValue(element, envelope.createName(
      kPasswordFieldName));

    // Check the soundness of our SOAP header processing before
    // we attempt to use the data for real. Otherwise bad or
    // missing data in these variables will simply manifest
    // itself misleadingly as a security configuration problem.
    if ((userName == null) || (userName.length() == 0)
      || (password == null) || (password.length() == 0)) {

      final AxisFault e = new AxisFault(
        "Bad username/password in SOAP" + " header '" + userName
        + "'/'" + password + "'");
      throw e;
    }

    msgContext.setUsername(userName);
    msgContext.setPassword(password);

    // Specify an absolute locale for the invocation:
    final String localeFrenchCanada = "fr_CA";
    msgContext.setProperty("locale", localeFrenchCanada);
```

```
    } catch (SOAPException e) {
      throw new AxisFault(e.getMessage(), e);
    }


    if (isRunningInWebSphere()) {
      // A WebSphere limitation means that it will not
      // automatically see the credentials we have just set, we
      // must also perform our login here.
      try {
        final Method doAsMethod = getDoAsMethod();
        final LoginContext loginContext =
          getLoginContext(userName, password);
        loginContext.login();
        final Subject subject = loginContext.getSubject();

        // Create a privileged action class which includes all the
        // information about this call.
        final PrivilegedAction action =
          new ProviderPrivilegedAction(this, msgContext);
        final Object[] parameterValues = {subject, action};

        // invoke the rest of the call under the new credentials.
        final Object axisFault =
          doAsMethod.invoke(null, parameterValues);

        // Exceptions cannot be thrown from the above invocation,
        // they are returned instead. If one was returned then
        // throw it now.
        if (axisFault != null) {
          throw new AxisFault("" + axisFault,
            (Exception) axisFault);
        }

      } catch (Exception e) {
        throw new AxisFault(e.getMessage(), e);
      }
    } else {
      // Not in WebSphere. Simply delegate straight through.
      super.invoke(msgContext);
    }

  }

  /**
   * An accessor for the invoke method in the superclass. Required
   * because it must be invoked by another class - the inner class
   * within this one.
   *
   * @param msgContext The message context object for this call.
   *
   * @throws AxisFault Generic Axis fault handler.
   */
  private void superInvoke(final MessageContext msgContext)
  throws AxisFault {

    super.invoke(msgContext);
  }

  /**
   * Indicates whether we are running within WebSphere in which
   * case we must delegate the rest of the call as a privileged
   * action.
   *
   * @return True if running under WebSphere, false otherwise.
   */
  private boolean isRunningInWebSphere() {
    final String vendorName = System.getProperty("java.vendor");
    return vendorName.startsWith("IBM");
  }

  /**
   * Gets a named element from the SOAP message, searching both
```

```
   * the body and header.
   *
   * @param envelope The SOAP envelope.
   * @param elementNameString The name of the element to get.
   *
   * @return The required element or null if it was not found.
   */
  private SOAPElement getSoapElement(final SOAPEnvelope envelope,
    final String elementNameString)  {

    SOAPElement result = null;
    try {
      final Name elementName =
        envelope.createName(elementNameString);
      final Name hrefAttributeName =
        envelope.createName(kHrefAttributeName);
      final SOAPHeader sh = envelope.getHeader();
      final SOAPBody sb = envelope.getBody();

      // first search the header.
      SOAPElement candidateElement = null;
      final Iterator headerIterator =
        sh.getChildElements(elementName);
      if (headerIterator.hasNext()) {
        candidateElement = (SOAPElement) headerIterator.next();
      }

      // search the body, if necessary.
      if (candidateElement == null) {
        final Iterator bodyIterator =
          sb.getChildElements();
        if (bodyIterator.hasNext()) {
          candidateElement = (SOAPElement) bodyIterator.next();
        }
      }

      // Now we need to check if this is literal or encoded
      // element. A literal one is embedded directly, an encoded
      // one means that this element is simply a pointer to an
      // element elsewhere in the message.
      if (candidateElement != null) {
        final String hrefValue =
          candidateElement.getAttributeValue(hrefAttributeName);
        if ((hrefValue != null) && (hrefValue.length() > 0)) {
          // it points to a multi ref, so get this instead.
          result = getMultiRefElement(envelope, hrefValue);
        } else {
          // It's literal so return it directly.
          result = candidateElement;
        }
      }
    } catch (SOAPException e) {
      e.printStackTrace();
    }
    return result;
  }

  /**
   * Gets a multi ref element from a SOAP message.
   *
   * @param envelope The SOAP envelope.
   * @param idStringWithPrefix The identifier of the multi ref
   * element.
   *
   * @return The matching element, or null if it was not found.
   *
   * @throws SOAPException If any SOAP error occurs.
   */
  private SOAPElement getMultiRefElement(
    final SOAPEnvelope envelope, final String idStringWithPrefix)
    throws SOAPException {

    SOAPElement result = null;
```

```
    // Remove the hash character:
    final String idString = idStringWithPrefix.substring(1);
    final Name idName = envelope.createName(kNameOfIdAttribute);

    final SOAPBody body = envelope.getBody();
    final Iterator multiRefIterator = body.getChildElements(
      envelope.createName(kMultiRefElementName));
    while (multiRefIterator.hasNext()) {
      final Object o = multiRefIterator.next();

      final SOAPElement currentElement = (SOAPElement) o;
      final String currentId =
        currentElement.getAttributeValue(idName);
      if (currentId.equals(idString)) {
        result = currentElement;
        break;
      }
    }
    return result;
  }

  /**
   * Gets the value of a specified element within the given
   * element. If multiple occurrences are present, the first one
   * is returned.
   *
   * @param element The element containing the required one.
   * @param elementName The name of the required element.
   *
   * @return The string value of the element, or null if the
   * specified sub element does not exist.
   */
  private String getSubElementValue(final SOAPElement element,
    final Name elementName) {

    String result = null;

    final Iterator elementIterator =
      element.getChildElements(elementName);
    if (elementIterator.hasNext()) {
      final SOAPElement subElement =
        (SOAPElement) elementIterator.next();
      result = subElement.getValue();
    }
    return result;
  }

  /**
   * Gets the hidden implementation class for the Login Context.
   *
   * @param userName The user name to login with.
   * @param password The password to login with.
   *
   * @return class for implementation
   *
   * @throws Exception if an error occurs getting an instance of
   * the LoginContext class
   */
  private LoginContext getLoginContext(
    final String userName, final String password)
    throws Exception {

    final LoginContext resultLoginContext;

    // Initialize WebSphere specific callback handler. Use
    // reflection to avoid a build time dependency on an IBM
    // class.
    final Class wsCallbackHandlerClass = Class.forName(
      EnvironmentConstants.kWSCallbackHandlerImplClassName);
    final Class[] parameters = { String.class, String.class };
    final Constructor constructor =
      wsCallbackHandlerClass.getConstructor(parameters);
    final Object[] parameterValues = {userName, password};
```

```
  // The WebSphere login
  resultLoginContext =
    new LoginContext(
      EnvironmentConstants.kWSLogin,
      (CallbackHandler) constructor.newInstance(
        parameterValues));

  return resultLoginContext;
}

/**
 * Gets the cached Do As method, initializing it if necessary.
 *
 * @return The Do As method for this server.
 *
 * @throws Exception If the method could not be obtained for
 * any reason.
 */
private Method getDoAsMethod() throws Exception {

  if (stSubjectDoAsMethod != null) {
    return stSubjectDoAsMethod;
  }

  final Class wsSubjectClass =
    Class.forName(EnvironmentConstants.kWSSubjectClassName);
  final Class[] moreParameters =
    { Subject.class, PrivilegedAction.class };
  stSubjectDoAsMethod =
    wsSubjectClass.getDeclaredMethod(
      EnvironmentConstants.kDoAsMethodName,
      moreParameters);
  return stSubjectDoAsMethod;
}

/**
 *
 *
 */
private class ProviderPrivilegedAction
implements PrivilegedAction {

  /** The message context for the call. */
  private final MessageContext msgContext;

  /** The class whose method we must invoke. */
  private final TestmodelProvider ownerObject;

  /**
   * Constructor which initializes the fields.
   *
   * @param newOwnerObject The class whose method we will
   * invoke.
   * @param newMsgContext The message context for the call.
   */
  public ProviderPrivilegedAction(
    final TestmodelProvider newOwnerObject,
    final MessageContext newMsgContext) {

    ownerObject = newOwnerObject;
    msgContext = newMsgContext;
  }

  /**
   * Runs the privileged action using the fields of this class.
   *
   * @return The exception resulting from the call, or null if
   * none was thrown.
   */
  public Object run() {
    Object resultFault = null;
    try {
```

```
        ownerObject.superInvoke(msgContext);
      } catch (Exception e) {
        resultFault = e;
      }
      return resultFault;
    }

  }

}
```

The text below shows an actual SOAP message (with some formatting for readability) which is processed by the *Java* code above. Note that the SOAP header refers to a parameter named `*inCred*` which contains the username and password credentials. The actual data is not stored literally in the header but in a `multiRef` element in the message body.

```
<soapenv:Envelope xmlns:soapenv=
  "http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <soapenv:Header>
    <inCred href="#id0" xmlns=""/>
  </soapenv:Header>

  <soapenv:Body soapenc:encodingStyle=
  "http://schemas.xmlsoap.org/soap/encoding/">
    <opDemo xmlns="http://remote.feature">
      <in1 href="#id1" xmlns=""/>
    </opDemo>

    <multiRef id="id1" soapenc:root="0" soapenv:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/"
      xsi:type="ns-905576305:PersonDetailsWrapper"
      xmlns:ns-905576305="http://feature/struct/" xmlns="">

      <firstName xsi:type="xsd:string">Jimmy</firstName>
      <idNumber xsi:type="xsd:string">0000361i</idNumber>
      <surname xsi:type="xsd:string">Client</surname>
    </multiRef>

    <multiRef id="id0" soapenc:root="0" soapenv:encodingStyle=
     "http://schemas.xmlsoap.org/soap/encoding/"
      xsi:type="ns-905576305:CredentialsWrapper"
      xmlns:ns-905576305="http://feature/struct/" xmlns="">

      <password xsi:type="xsd:string">password</password>
      <userName xsi:type="xsd:string">superuser</userName>
    </multiRef>
  </soapenv:Body>

</soapenv:Envelope>
```

Sample Document-Style Customizations

In the following code sample a document-oriented web service passing an XML document (message-style) web service is processed by a custom provider to allow for default credentials to be set based on the operation name.

```
package webservice;
```

```
import
  curam.util.connectors.webservice.CuramMsgStyleEJBMethodProvider;
import java.util.Vector;
import javax.xml.namespace.QName;
import org.apache.axis.AxisFault;
import org.apache.axis.Handler;
import org.apache.axis.MessageContext;
import org.apache.axis.description.OperationDesc;
import org.apache.axis.description.ServiceDesc;
import org.apache.axis.handlers.soap.SOAPService;
import org.apache.axis.i18n.Messages;
import org.apache.axis.message.MessageElement;
import org.apache.axis.message.SOAPEnvelope;

/**
 * A web services hook which extends the Curam message style
 * provider to enable the developer to do custom processing for
 * properties and then pass control to the Curam handler.
 */
public class CustomMsgStyleMethodProvider
  extends CuramMsgStyleEJBMethodProvider {

  /**
   * Process the message: Ensure the interface method matches
   * the supported Curam message style signature:
   * [public Document doSomething (Document xmlMessage)].
   *
   * Parse the SOAP body XML message into a Document object,
   * strip the root wrapper element, do the actual invocation
   * and restore the root wrapper element before responding.
   *
   * @param msgContext the active MessageContext
   * @param reqEnv the request SOAPEnvelope
   * @param resEnv the response SOAPEnvelope
   * @param obj the service target object
   * @throws Exception exception
   */
  @Override
  public void processMessage(final MessageContext msgContext,
    final SOAPEnvelope reqEnv, final SOAPEnvelope resEnv,
    final Object obj)
    throws Exception {

    try {

      OperationDesc operation = msgContext.getOperation();
      final SOAPService serviceHandler = msgContext.getService();
      final ServiceDesc serviceDesc =
        serviceHandler.getServiceDescription();
      QName opQName = null;

      // If operation not in the context extract from the
      // SOAP envelope.
      if (operation == null) {
        final Vector bodyElements = reqEnv.getBodyElements();
        if (bodyElements.size() > 0) {
          final MessageElement element =
            (MessageElement) bodyElements.get(0);
          if (element != null) {
            opQName = new QName(
              element.getNamespaceURI(), element.getLocalName());
            operation =
              serviceDesc.getOperationByElementQName(opQName);
          }
        }
      }

      // Cannot proceed without an operation name.
      if (operation == null) {
        throw new
          AxisFault(Messages.getMessage("noOperationForQName",
```

```
          opQName == null ? "null" : opQName.toString()));
      }

      // If this is a "public" operation we ensure
      // default credentials are supplied.
      if ("public_operation".equals(opQName.toString())) {

        final String jndiUserName = "jndiUser";
        String jndiUser = (serviceHandler != null)
          ? (String) serviceHandler.getOption(jndiUserName)
          : (String) getOption(jndiUserName);
        if (jndiUser == null || jndiUser.length() == 0) {
          serviceHandler.setOption(jndiUserName, "default");

          final String jndiPasswordName = "jndiPassword";
          serviceHandler.setOption(jndiPasswordName, "password");
        }
      }

      msgContext.setService(serviceHandler);

      // Process the message
      super.processMessage(msgContext, reqEnv, resEnv, obj);
    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }
  }
}
```

Example 4.12 Example Message Style Provider Override

A custom message-style handler could also be used to intercept the (W3C) Document in the SOAP message to inspect and modify as the following illustrates:

```
package webservice;

import curam.util.exception.AppException;
import curam.util.message.INFRASTRUCTURE;
import curam.util.webservices.MessageProcessor;
import java.lang.reflect.Method;
import java.util.Vector;
import javax.xml.namespace.QName;
import org.apache.axis.AxisFault;
import org.apache.axis.MessageContext;
import org.apache.axis.description.OperationDesc;
import org.apache.axis.description.ServiceDesc;
import org.apache.axis.handlers.soap.SOAPService;
import org.apache.axis.i18n.Messages;
import org.apache.axis.message.MessageElement;
import org.apache.axis.message.SOAPBodyElement;
import org.apache.axis.message.SOAPEnvelope;
import org.w3c.dom.Document;
import org.w3c.dom.Node;

/**
 *
 */
public class CustomMsgStyleMethodProvider
extends CuramMsgStyleEJBMethodProvider {
  /**
   * Process the message: Ensure the interface method matches
   * the supported Curam message style:
   * [public Document doSomething (Document xmlMessage)].
   *
   * Parse the SOAP body XML message in to a Document object,
   * strip the root wrapper element, do the actual invocation
```

```
   * and restore the root wrapper element before responding.
   *
   * @param msgContext the active MessageContext
   * @param reqEnv the request SOAPEnvelope
   * @param resEnv the response SOAPEnvelope
   * @param obj the service target object
   * @throws Exception exception
   */
  @Override
  public void processMessage(final MessageContext msgContext,
    final SOAPEnvelope reqEnv, final SOAPEnvelope resEnv,
    final Object obj)
    throws Exception {


    OperationDesc operation = msgContext.getOperation();
    final SOAPService service = msgContext.getService();
    final ServiceDesc serviceDesc =
      service.getServiceDescription();
    QName opQName = null;

    // If operation not in the context extract from the
    // SOAP envelope.
    if (operation == null) {
      final Vector bodyElements = reqEnv.getBodyElements();
      if (bodyElements.size() > 0) {
        final MessageElement element =
          (MessageElement) bodyElements.get(0);
        if (element != null) {
          opQName = new QName(
            element.getNamespaceURI(), element.getLocalName());
          operation =
            serviceDesc.getOperationByElementQName(opQName);
        }
      }
    }

    // Cannot proceed without an operation name.
    if (operation == null) {
      throw new
        AxisFault(Messages.getMessage("noOperationForQName",
        opQName == null ? "null" : opQName.toString()));
    }

    final Method method = operation.getMethod();
    final int methodType = operation.getMessageOperationStyle();

    if (methodType == OperationDesc.MSG_METHOD_DOCUMENT) {
      // Dig out the body XML and invoke method.
      final Vector bodies = reqEnv.getBodyElements();

      Document doc =
        ((SOAPBodyElement) bodies.get(0)).getAsDocument();

      // Preserve wrapper root element, and then remove it.
      final Node root = doc.getDocumentElement();
      doc = MessageProcessor.removeWrapperElement(doc);


      // ******************************
      // Custom Document processing here.
      // ******************************
      // ...


      // Add XML to arg for invocation.
      final Object[] argObjects = new Object[1];
      argObjects[0] = doc;

      // Do the actual invocation of EJB method.
      // TODO: instantiate your own provider instead
      //       of CuramEJBMethodProvider.
```

```
      final CuramEJBMethodProvider ejbMP =
        new CuramEJBMethodProvider();
      Document resultDoc =
        (Document) ejbMP.
        invokeMethod(msgContext, method, obj, argObjects);

      // Add return XML to SOAP response
      if (resultDoc != null) {
        // Restore the wrapper root element that
        // was removed above.
        resultDoc =
          MessageProcessor.
          restoreWrapperElement(root, resultDoc);

        resEnv.addBodyElement(
          new SOAPBodyElement(resultDoc.getDocumentElement()));
      }
      return;
    }
  }
}
```

Sample Facade Bean Invocation

Here is an example of invoking a facade bean that is created by the Cúram modeling and build environments. This could be utilized in a context where you are implementing an alternative web services implementation. In this particular example the code is specific to a *WebLogic Server* application server, but could easily be modified for *WebSphere* as shown in the commented section.

```
package webservice;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.security.AccessController;
import java.security.PrivilegedAction;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;


public class FacadeUsageDemo {

  /**
   * Illustrates how a Curam facade can be invoked from a client.
   */
  private void invokeFacade() throws NamingException,
    ClassNotFoundException, SecurityException,
    NoSuchMethodException, IllegalArgumentException,
    IllegalAccessException, InvocationTargetException {

    final String sUser = "tester";
    final String sPassword = "password";

    // TODO: Change for non-WebLogic application server.
    final String initialCtxFactory =
      "weblogic.jndi.WLInitialContextFactory";
    final String providerUrl = "t3://localhost:7001";

    // Authenticate and get an initial context.
    final Properties properties = new Properties();
    properties.setProperty(Context.SECURITY_PRINCIPAL, sUser);
    properties.setProperty(Context.SECURITY_CREDENTIALS,
```

```
      sPassword);
    properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
      initialCtxFactory);
    properties.setProperty(Context.PROVIDER_URL, providerUrl);
    final Context initialContext =
      new InitialContext(properties);

    // Lookup the facade bean.
    final Object o =
      initialContext.
      lookup("java:comp/env/curamejb/MyFacadeBeanClass");


    // Load the home interface class so that we can narrow to it.
    final Class<?> cls = getContextClassLoader().loadClass(
      "my.custom.remote.MyFacadeHome");
    final Object ehome =
      javax.rmi.PortableRemoteObject.narrow(o, cls);

    // Get and invoke the 'create' method of the home interface
    // to give us a reference to the facade bean.
    final Method createMethod =
      cls.getMethod("create", new Class[0]);
    final Object facadeObj =
      createMethod.invoke(ehome, new Object[0]);

    // Use reflection to get and invoke the method
    // 'myMethod' of our interface.
    final Class<?> facadeObjClass = facadeObj.getClass();
    final Method myMethod =
      facadeObjClass.getMethod("myMethod", new Class[0]);
    // Pass arguments to invoke based on the method signature:
    myMethod.invoke(facadeObj, new Object[0]);
    initialContext.close();

    // Note: once the initialContext object has been obtained,
    // the remainder of this method could be written like this.
    // However due to problems experienced with loading classes
    // in WebLogic, the above workaround is necessary so that a
    // classloader could be explicitly specified.
    //try {
    //  final custom.webservice.remote.MyFacadeBeanClassHome
    //    ehome =
    //      (custom.webservice.remote.MyFacadeBeanClassHome)
    //      javax.rmi.PortableRemoteObject.narrow(
    //      o,
    //    custom.webservice.remote.MyFacadeBeanClassHome.class);
    //  custom.webservice.remote.MyFacadeBeanClass facadeObj =
    //    ehome.create();
    //  // Assumes a void method here; otherwise, arguments
    //  // would be needed:
    //  facadeObj.myMethod();
    //  initialContext.close();
    //} catch (Throwable t) {
    //  t.printStackTrace();
    //}

  }

  /**
   * Gets the class loader for the context.
   *
   * @return ClassLoader for the context.
   */
  private ClassLoader getContextClassLoader() {
    return AccessController.doPrivileged(
      new PrivilegedAction<ClassLoader>() {
        /**
         * PrvilegedAction subclass.
         *
         * @return ClassLoader for the context.
         */
```

```
        public ClassLoader run() {
          return Thread.currentThread().getContextClassLoader();
        }
      }
    );
  }

}
```

Chapter 5

Secure Web Services

5.1 Overview

Web service security is an important, but optional, part of your web services implementation. Existing and legacy web service security is described in this chapter. For *Rampart* and *Axis2* web services security you will learn about:

- Using custom SOAP headers with *Axis2* and encrypting them;
- Using and setting up *Rampart*;
- Using HTTPS/SSL to secure web service network traffic.

For legacy web services you will learn about the following, some of which can be utilized with *Rampart*:

- *IBM Cúram Social Program Management* modeling requirements for using secure web services;
- Coding password callback handlers (also applicable to *Axis2* if your policy specifies a password callback handler);
- Setting up the client environment;
- Creating keystore files (also applicable to *Axis2* if your environment requires these steps for supporting HTTPS/SSL).

5.2 Axis2 Security and Rampart

Rampart is the security module of *Axis2*. With the *Rampart* module you can secure web services for authentication (but see below), 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 *IBM Cúram Social Program Management* applications is that *Rampart* Authentication cannot be used. This is due to the requirements of *IBM Cúram Social Program Management* 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 Section 5.3, *Custom SOAP Headers* 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 *IBM Cúram Social Program Management* infrastructure changes using *Rampart* and WS-Security Policy definitions. A WS-Security Policy document can be embedded in a custom `services.xml` descriptor (see Section 4.6.2, *Deployment Descriptor File*). 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 Section 5.4, *Encrypting Custom SOAP Headers*). 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 publically? Are multiple levels of encryption necessary; for instance, do you really need both *Rampart* encryption and HTTP/SSL encryption?

5.3 Custom SOAP Headers

Credential checking is enforced in *IBM Cúram Social Program Management* for web service invocations based on the default expectation that a client invoking a web service will provide a custom SOAP header. This topic was introduced in Section 4.6, *Providing Web Service Customizations* insofar as you need to plan specific customizations if you choose to bypass this security checking. 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 *IBM Cúram Social Program Management* access to the web service. This section explains how your clients can provide these SOAP headers.

The following is an example of the *IBM Cúram Social Program Management* 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>
```

Example 5.1 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);
}
```

Example 5.2 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 failing 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>
```

Potential Security Vulnerability

Be aware that by default custom SOAP headers containing credentials for authentication pass on the wire in plain-text! This is an unsecure situation and you must encrypt this traffic to prevent your credentials from being vulnerable and your security from being breached. See Section 5.4, *Encrypting Custom SOAP Headers* and/or Section 5.6, *Securing Web Service Network Traffic With HTTPS/SSL* on how you might rectify this.

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>
```

Example 5.3 Sample Custom SOAP Header

5.4 Encrypting Custom SOAP Headers

Since SOAP data (e.g. the headers above in Section 5.3, *Custom SOAP Headers*) travels across the wire, by default, as plain text, using *Rampart* to encrypt your *IBM Cúram Social Program Management* custom SOAP headers is one way to help ensure the security of these credentials. Of course, you should plan a security strategy and implementation for all of your web services and related data based on your overall, enterprise-wide requirements, environment, platforms, etc. The information in this section is just one small part of your overall security picture.

There is additional information on coding your web service clients for *Rampart* security in Section 5.5, *Using Rampart With Web Services* that will help provide context for the following.

The steps to encrypt these headers are:

1. Add the following to your client descriptor file:

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

(See Section 5.5.1, *Defining the Axis2 Security Configuration* 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 Section 5.5.1, *Defining the Axis2 Security Configuration* for more information on the contents of this file.)

2. Engage and invoke *Rampart* in your client code as per Section 5.5, *Using Rampart With Web Services*.

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 Example 5.3, *Sample Custom SOAP Header*:

In the following example encryptedParts was used to encrypt the *IBM Cúram Social Program Management* 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>
```

Example 5.4 Example Encrypted Custom SOAP Header

5.5 Using Rampart With Web Services

There are a number of parts to *Rampart* security, as indicated in Section 5.1, *Overview*, and covering these in detail is outside the scope of this document;

but, the following is provided to give you a high-level view on utilizing *Rampart* with your *IBM Cúram Social Program Management Axis2* web services.

Broadly, there 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.

5.5.1 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 (deprecated). On the server side you can define the security configuration via the service descriptor file (`services.xml`) or via a *Rampart* policy 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.curamsoftware.com}Credentials
      </encryptionParts>

    </action>
  </parameter>
...
```

Example 5.5 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>
```

Example 5.6 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
```

Example 5.7 Axis2 Client File System Structure

The equivalent specification to the parameters in Example 5.5, *Sample Client Descriptor Settings (Fragment)* and Example 5.6, *Sample Server Security Settings (services.xml Fragment)* via a *Rampart* policy file would be as follows:

```
...
<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:encryptionCypto>
    <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:encryptionCypto>
</ramp:RampartConfig>
...
```

Example 5.8 Sample Rampart Policy (policy.xml Fragment)

5.5.2 Providing the Security Data and Code

The example configurations in Section 5.5.1, *Defining the Axis2 Security Configuration* specify an encryption property file and password call back routine, which would be used in the process of encrypting 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 below in Example 5.9, *Example Rampart server-crypto.properties File*. When using a *Rampart* policy file, as shown in Example 5.8, *Sample Rampart Policy (policy.xml Fragment)*, these property files are not used 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
```

Example 5.9 Example Rampart server-crypto.properties File

The `client-crypto.properties` file would have 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 Section 5.7.4, *Keystore File Creation*.

When configuring a secure web service the server signature property file and keystore file (`server-crypto.properties` and `server.keystore`) must be placed 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 specified in the `passwordCallbackClass` parameter entities are illustrated in Example 5.17, *ServerPWCallback.java* and Example 5.20, *ClientPWCallback.java*.

5.5.3 Coding the Client

The following code snippets illustrate what's needed to add to the basic client examples in Example 3.4, *Sample Web Service Client Using Generated Stub and Custom Code* to utilize the preceding security illustrations.

To utilize a client `axis2.xml` descriptor file you would need to make the following API call where `C:/Axis2/client` also contains the *Axis2* modules directory as indicated in Example 5.7, *Axis2 Client File System Structure*:

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

Example 5.10 Identifying Axis2 Client Rampart Configuration

To utilize a *Rampart* policy file you would need to create a context as above, 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 utilizing an *Axis2* configuration that specifies the necessary modules (as shown in Example 5.7, *Axis2 Client File System Structure*) you will need to explicitly engage the necessary module(s) prior to invoking the service. The specific modules required will depend on the security features and configuration you are using; for example:

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

Failing to do this will result in a server-side error; e.g.:

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

To utilize a *Rampart* policy you would need to 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, etc. can manifest in the client and/or the server. Often an error in the client cannot be debugged without access to the *log4j* trace from the server. For instance, the error when the proper module(s) has not been engaged (discussed earlier) may appear in the client as:

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

Here is an example that combines the fragments above, illustrating providing a *IBM Cúram Social Program Management* custom SOAP header and 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.getPolicy(builder.getDocumentElement());
options.setProperty(RampartMessageData.KEY_RAMPART_POLICY,
  policy);
client.setOptions(options);


// Because we are not using an axis2.xml client
// configuration file we MUST explicitly load
// Rampart.
client.engageModule("rampart");


// Setup the SOAP message.
// For this example three integers are to be summed.
final OMFactory factory = OMAbstractFactory.getOMFactory();
final OMNamespace ns = factory.
  createOMNamespace("http://remote.custom.util.curam", "ns1");
final OMElement element = factory.
  createOMElement("myOperation", ns);

final OMElement childElem1 = factory.
  createOMElement("args0", null);
childElem1.setText("One");
element.addChild(childElem1);

final OMElement childElem2 = factory.
  createOMElement("args1", null);
childElem2.setText("Two");
element.addChild(childElem2);

final OMElement childElem3 = factory.
  createOMElement("args2", null);
childElem3.setText("Three");
element.addChild(childElem3);


// Invoke the service.
final OMElement response =
  client.sendReceive(element);

// Process the return data.
final String sData = response.getFirstElement().getText();
```

```
  System.out.println("Service returned: " + sData);
}
```

Example 5.11 Sample Client Code to Encrypt a Custom SOAP Header

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 Example 5.11, *Sample Client Code to Encrypt a Custom SOAP Header*, 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);
```

Example 5.12 Sample Client Code (Deprecated) for Setting the Client Security Configuration

5.6 Securing Web Service Network Traffic With HTTPS/SSL

The use of HTTPS/SSL may be a part of your web services security strategy and details about setting this up are beyond the scope of this document; but, be aware that the use of HTTPS/SSL can be established in either of the following ways:

- Application server environment - Setting this up is very specific to your particular application server, but essentially inolves exporting the appropriate server certificates and making them available to your client envir-

onment.

- *Rampart* WS-Security policy - There are a number of articles, etc. available on the Internet that cover this in more detail.

For client access the end point needs to reflect the protocol and port change, which can be done dynamically at runtime. For instance, client code like this 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()));
```

Example 5.13 Example of Dynamically Changing the Web Service End Point

Your client will need to identify the keystore and password that contains the necessary certificates; e.g.:

```
      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 may experience this error:

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

And you should be able to resolve this error with code like this:

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

5.7 Legacy Secure Web Services

5.7.1 Objective

In this chapter you will learn how to apply security to web services using WS-Security (Web Services Security).

The WS-Security specification defines a set of SOAP header extensions for end-to-end SOAP messaging security. It supports message integrity and confidentiality by allowing communicating partners to exchange signed and encrypted messages in a web services environment.

5.7.2 Modeling Secure Web Services

To enable security for an *Inbound Web Service* you need to add a request and a response handler for each web service.

The request security handler processes the incoming SOAP messages and is specified using the *IBM Cúram Social Program Management* `Request_Handlers` property in *Rational Software Architect*. An example of this property's value is shown in Example 5.14, *Request Handler for a web service*).

```
<handler type=
  \"java:org.apache.ws.axis.security.WSDoAllReceiver\">
  <parameter name=\"passwordCallbackClass\"
             value=\"webservice.ServerPWCallback\"/>
  <parameter name=\"action\"
             value=\"UsernameToken Signature Encrypt\"/>
  <parameter name=\"passwordType\"
             value=\"PasswordText\" />
  <parameter name=\"signaturePropFile\"
             value=\"server-crypto.properties\" />
  <parameter name=\"signatureParts\"
             value=\"{Element}{http://docs.oasis-open.org/wss/
2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd}UsernameToken;
{Content}{}Body\" />
  <parameter name=\"encryptionParts\"
             value=\"{Element}{http://docs.oasis-open.org/wss/
2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd}UsernameToken;
{Content}{}Body\" />
</handler>
```

Example 5.14 Request Handler for a web service

The response security handler processes the outgoing SOAP messages and is specified using the *IBM Cúram Social Program Management* `Response_Handlers` property in *Rational Software Architect*. An example of this option's value is shown in Example 5.15, *Response Handler for a web service*.

```
<handler type=
\"java:org.apache.ws.axis.security.WSDoAllSender\">
  <parameter name=\"passwordCallbackClass\"
             value=\"webservice.ServerPWCallback\"/>
  <parameter name=\"action\"
```

```
            value=\"Signature Encrypt\"/>
  <parameter name=\"signaturePropFile\"
            value=\"server-crypto.properties\" />
  <parameter name=\"user\" value=\"curam-sv\"/>
  <parameter name=\"encryptionUser\" value=\"curam\"/>
</handler>
```

Example 5.15 Response Handler for a web service

The use of these options results in the addition of the security handler information to the request flow and response flow of handlers in the `server-config.wsdd` deployment descriptor file for the web service.

In the examples above the `action` value *UsernameToken* directs the handler to insert a *UsernameToken* token, containing the username and password, into the SOAP request. The values *Signature* and *Encrypt* define that the SOAP message should be signed and encrypted. This results in the handler first signing and then encrypting the data. The value of `encryptionParts` and `signatureParts` specifies to sign and encrypt the SOAP message's security header and body elements.

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. The example of crypto property file is shown in Example 5.16, *Example server-crypto.properties File*:

```
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
```

Example 5.16 Example server-crypto.properties File

When configuring a web service the signature property file and keystore file (`server-crypto.properties` and `server.keystore`) must be placed in the `%SERVER_DIR%/project/config/wss/` directory. Section 5.7.4, *Keystore File Creation* describes how to create file `server.keystore`.

The security handler (`passwordCallbackClass`), `webservice.ServerPWCallback` in Example 5.14, *Request Handler for a web service* and Example 5.15, *Response Handler for a web service* above, provides a password callback mechanism and should be implemented by the developer. The example of an implementation for `webservice.ServerPWCallback` is shown in Example 5.17, *ServerPWCallback.java*:

```
package webservice;
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

/**
 * Implementation of password callback class.
```

```
 */
public class ServerPWCallback implements CallbackHandler {

  /**
   * Retrieve or display the information requested in the
   * provided Callbacks.
   *
   * @param callbacks an array of Callback objects provided by
   * an underlying security service which contains the
   * information requested to be retrieved or displayed.
   *
   * @throws IOException if an input or output error occurs.
   * @throws UnsupportedCallbackException if the implementation
   * of this method does not support one or more of the Callbacks
   * specified in the callbacks parameter.
   */
  public void handle(final Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {

    for (int i = 0; i < callbacks.length; i++) {
      if (callbacks[i] instanceof WSPasswordCallback) {
        final WSPasswordCallback pc =
          (WSPasswordCallback) callbacks[i];
        /*
         * Here call a method to lookup the password for
         * the given identifier (e.g. a user name or key
         * store alias), e.g. pc.setPassword(
         * passStore.getPassword(pc.getIdentfifier))
         * for testing we supply a fixed name/fixed key here.
         */
        if ("beantester".equals(pc.getIdentifer())) {
            pc.setPassword("password");
        } else if ("curam-sv".equals(pc.getIdentifer())) {
            pc.setPassword("password");
        } else if ("curam".equals(pc.getIdentifer())) {
            pc.setPassword("password");
        }

      } else {
        throw new UnsupportedCallbackException(callbacks[i],
          "Unrecognized Callback");
      }
    }
  }
```

Example 5.17 ServerPWCallback.java

5.7.3 Client Side Configuration

To provide security the web service client can be configured either programmatically or using a deployment descriptor file. The web services java client example (see Example 5.21, *WebServiceTest.java*) details how to create a deployment descriptor programmatically by adding *UsernameToken* token and configuring the client to sign and encrypt the incoming soap request.

The example of deployment descriptor file is shown in Example 5.18, *client_config.wsdd*:

```
<?xml version="1.0"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <transport name="http"pivot=
"java:org.apache.axis.transport.http.HTTPSender"/>
  <globalConfiguration>
    <requestFlow>
```

```
      <handler type=
"java:org.apache.ws.axis.security.WSDoAllSender"/>
    </requestFlow>

    <responseFlow>
     <handler type=
"java:org.apache.ws.axis.security.WSDoAllReceiver">
      <parameter name="passwordCallbackClass"
        value="test.ClientPWCallback"/>
      <parameter name="action"
                 value="Signature Encrypt"/>
      <parameter name="signaturePropFile"
                 value="client-crypto.properties" />
     </handler>
    </responseFlow>

  </globalConfiguration>
</deployment>
```

Example 5.18 client_config.wsdd

The deployment descriptor file (Example 5.18, *client_config.wsdd*) contains the request flow and response flow specified for incoming and outgoing SOAP messages.

As on the server side, the request flow will contain `action` *UsernameToken Signature Encrypt*.

Note that request flow does not specify any actions or security configuration in the deployment descriptor. In Example 5.21, *WebServiceTest.java* below you will see that it can be added programmatically instead of specifying in deployment descriptor file.

The `action` *UsernameToken* is not specified in the response flow. This means that the *UsernameToken* will not be included into SOAP message.

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. A sample crypto property file is shown in Example 5.19, *client-crypto.properties*.

```
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
```

Example 5.19 client-crypto.properties

The value of `org.apache.ws.security.crypto.merlin.file` in Example 5.19, *client-crypto.properties* specifies the client's keystore file. Section 5.7.4, *Keystore File Creation* how to create file `client.keystore`.

The security handler also requires a callback class, `test.ClientPWCallback`, to be implemented. The example of an implementation for `test.ClientPWCallback` is shown in Example 5.20, *ClientPWCallback.java*:

```
package test;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;


/**
 * Implementation of password callback class.
 */
public class ClientPWCallback implements CallbackHandler {

  /**
   * Retrieve or display the information requested in the
   * provided Callbacks.
   *
   * @param callbacks an array of Callback objects provided
   * by an underlying security service which contains the
   * information requested to be retrieved or displayed.
   *
   * @throws IOException if an input or output error occurs.
   * @throws UnsupportedCallbackException if the implementation
   * of this method does not support one or more of the Callbacks
   * specified in the callbacks parameter.
   */
  public void handle(final Callback[] callbacks
    throws IOException, UnsupportedCallbackException {

    for (int i = 0; i < callbacks.length; i++) {
      if (callbacks[i] instanceof WSPasswordCallback) {
        final WSPasswordCallback pc =
          (WSPasswordCallback) callbacks[i];
        if ("beantester".equals(pc.getIdentifer())) {
          pc.setPassword("password");
        } else if ("curam-sv".equals(pc.getIdentifer())) {
          pc.setPassword("password");
        }
      } else {
        throw new UnsupportedCallbackException(callbacks[i],
          "Unrecognized Callback");
      }
    }
  }
}
```

Example 5.20 ClientPWCallback.java

The request flow may be configured programmatically before invoking a web service. Example 5.21, *WebServiceTest.java* shows how to configure the request flow for the `SecureWebService` programmatically, and then invoke the web service.

```
package test;

import org.apache.axis.EngineConfiguration;
import org.apache.axis.configuration.FileProvider;
import org.apache.ws.security.handler.WSHandlerConstants;
import wsconnector.SecureWebServiceInput;
import wsconnector.SecureWebServiceServiceLocator;
import wsconnector.SecureWebServiceSoapBindingStub;

public class WebServiceTest {

  /**
   * Configures web service's request flow and call
   * secure web service.
```

```
   */
  public void callSecureWebservice() {

    final EngineConfiguration config = new FileProvider(
      ClassLoader.getSystemResourceAsStream(
        "client_config.wsdd"));

    final SecureWebServiceServiceLocator locator =
      new SecureWebServiceServiceLocator(config);

    final SecureWebServiceSoapBindingStub call =
      (SecureWebServiceSoapBindingStub)
        locator.getSecureWebService();

    call._setProperty(WSHandlerConstants.ACTION,
      WSHandlerConstants.USERNAME_TOKEN + " "
      + WSHandlerConstants.SIGNATURE + " "
      + WSHandlerConstants.ENCRYPT + " ");
    call._setProperty(WSHandlerConstants.PASSWORD_TYPE,
      "PasswordText");
    call._setProperty(WSHandlerConstants.SIGNATURE_PARTS,
      "{Element}{http://docs.oasis-open.org/"
      + "wss/2004/01/oasis-200401-wss-"
      + "wssecurity-secext-1.0.xsd}UsernameToken; "
      + "{Content}{}Body");
    call._setProperty(WSHandlerConstants.ENCRYPTION_PARTS,
      "{Element}{http://docs.oasis-open.org/"
      + "wss/2004/01/oasis-200401-wss-"
      + "wssecurity-secext-1.0.xsd}UsernameToken; "
      + "{Content}{}Body");
    call._setProperty(WSHandlerConstants.USER,
      "beantester");
    call._setProperty(WSHandlerConstants.PW_CALLBACK_CLASS,
      "test.ClientPWCallback");
    call._setProperty(WSHandlerConstants.SIG_PROP_FILE,
      "client-crypto.properties");
    call._setProperty(WSHandlerConstants.ENC_KEY_ID,
      "X509KeyIdentifier");
    call._setProperty(WSHandlerConstants.ENCRYPTION_USER,
      "curam");

    final SecureWebServiceInput details =
      new SecureWebServiceInput();
    details.setIntValue(47277);

    call.oper(details);

  }
}
```

Example 5.21 WebServiceTest.java

5.7.4 Keystore File Creation

This section describes how to create the `server.keystore` and `client.keystore` keystore files for secure web service configuration, as used in examples Example 5.16, *Example server-crypto.properties File* and Example 5.19, *client-crypto.properties*:

- 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 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 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

Appendix A

Glossary

A.1 Definitions

document-oriented web services (DOWS)

There is no formal, industry-accepted definition for DOWS, it is generally accepted that these are defined by the use of `style="document"` attribute in the WSDL binding section. In the context of Cúram the concept of DOWS also includes the option to model Cúram operations (web services) to process XML documents directly.

inbound web services

Inbound web services refers to web services that you implement and are accessed by external clients.

outbound web services

Outbound web services refers to web services that you would access, hosted externally.

SOA

Service-Oriented Architecture

As defined by IBM® (http://www-01.ibm.com/software/solutions/soa/): "Service Oriented Architecture (SOA) is a business-centric IT architectural approach that supports integrating your business as linked, repeatable business tasks, or services." Typically, web services take a significant role in a SOA implementation.

SOAP

Simple Object Access Protocol

SOAP is an XML-based protocol for processing web services over HTTP. For a more comprehensive definition see: http://www.w3.org/TR/soap/.

stub

A stub refers to the *Java* code that's generated at build time by the Cúram scripts by invoking *Axis2* (or *Axis* 1.4) tooling; they are stubs in that the code does not stand alone: you must code a *Java* main program, or provide some other context, where you instantiate its various objects and call their methods to invoke (outbound) the web service, process results, etc.

WSDL

Web Services Description Language

WSDL is an XML-based format for describing web services. For a more comprehensive definition see: http://www.w3.org/TR/wsdl.

WS-I (Web Services Interoperability Organization)

The WS-I is part of OASIS (Organization for the Advancement of Structured Information Standards) standards organization and as their website (`http://www.oasis-ws-i.org/`) their mission is to advance: "... Best Practices for Web services standards across platforms, operating systems, and programming languages."

Appendix B

Inbound Web Service Properties - ws_inbound.xml

B.1 Property Settings

The following details 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 can also be customized as described in Section 4.6, *Providing Web Service Customizations*.

These are the default properties produced by the *IBM Cúram Social Program Management* generator:

classname

The fully qualified name of the web service class, from the *Rational Software Architect* 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* 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* 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 in Example B.1, *Sample Generated ws_inbound.xml Properties File*.

```
<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>
```

Example B.1 Sample Generated ws_inbound.xml Properties File

The following are the properties that can be provided and/or customized via 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* 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 *IBM Cúram Social Program Management* 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 *IBM Cúram Social Program Management* 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 be-

ing built and thus exposed.

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

An example, custom `ws_inbound.xml` property file is shown in Example B.2, *Sample Custom ws_inbound.xml Properties File*.

```
<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>
```

Example B.2 Sample Custom ws_inbound.xml Properties File

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.

Appendix C

Deployment Descriptor File - services.xml

C.1 Descriptor File Contents

Each web service class requires its own *Axis2* deployment descriptor file (`services.xml`). The Cúram build automatically generates a suitable deployment descriptor for the default settings described in Section 4.6.1, *Inbound Web Service Properties File* and Appendix B, *Inbound Web Service Properties - ws_inbound.xml*. 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.

Based on the settings from the `ws_inbound.xml` property file(s) 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.

An example `services.xml` descriptor file that would be generated is shown in Example C.1, *Sample Generated services.xml Descriptor File*.

```
<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>
```

Example C.1 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 Section 4.6.1, *Inbound Web Service Properties File*). 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.)

Appendix D

Troubleshooting

D.1 Introduction

This appendix discusses some techniques for troubleshooting *Axis2* and *Axis* 1.4 web services. It covers:

- Initial server validation and troubleshooting;
- Tools and techniques for troubleshooting *Axis2* or *Axis* 1.4 errors;
- Avoid use of 'anyType'.

Having modelled your web service(s), developed your server code, built and deployed your application and web service EAR files, you are now ready to begin testing and finally delivering your web service.

Axis2 and *Axis* 1.4 represent a complex set of software and third-party products, especially when viewed from the perspective of running in an application server environment. While the *IBM Cúram Social Program Management* environment simplifies many aspects of web service development the final steps of testing and debugging your services can prove daunting. The various tips and techniques discussed here are neither new nor comprehensive, but are here to help you consider options and ways of increasing your effectiveness.

D.2 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. So, once deployed you want to first focus your testing on the server side to ensure everything there is working properly and then introduce your client development and testing so that you will better know where to focus for resolving errors.

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

If your application has started successfully the next step is to ensure your service is available. This is done differently for *Axis2* and *Axis* 1.4. 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* and *Axis* 1.4 environments are in the sections following.

D.2.1 Axis2 Environment Validation

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 resultant 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 nice sanity check) by including that service as external content when you build your *Axis2* web services WAR/EAR file; see Section 4.4, *Building and Packaging Web Services* for more information on doing this.

Having successfully validated your *Axis2* environment you should click the "Back Home" link on that page and select the Services link on the "Welcome!" page. The resulting "Available services" page will list all available services (classes) and their operations. If there is any invalid service (e.g. due to a missing implementation class) it will be flagged here 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 will generate and display the WSDL for that service. This verifies your deployed service(s) and it should now be available for invocation.

To Be Aware Of

- On the "Available services" page you may see the operation "setSessionContext", which you did not model and code. This behavior is an aspect of the issue described in Section 4.3, *Modeling and Implementing an Inbound Web Service* and in the *Cúram Release Notes*. It has no impact and can be ignored.
- The WSDL generated from the "Available services" link(s) is not equivalent to the WSDL generated by the *Axis2* Java2WSDL tool and the latter should be used for development of outbound web services and can be found in the `build/svr/wsc` directory of your development environment following a web services EAR file build.

- There are a number of issues with dynamic WSDL generation (e.g. *AXIS2-4541*) at this level of Apache *Axis2* (1.5.1); see http://issues.apache.org/jira/browse/AXIS2 for more information.
- *Axis2* has additional capabilities for checking, investigating, etc. your environment via its external administration web application (the "Administration" link on the "Welcome!" page). See Appendix E, *Including the Axis2 Admin Application in Your Web Services WAR File* for details on including this application in your environment. If you don't explicitly build/include this application the functionality won't be available.

D.2.2 Axis 1.4 Environment Validation

The equivalent validation and service check for *Axis* 1.4 is performed by invoking the "services" page via your web services application root context and application server port configuration. For instance, this URL might look like: `http://localhost:9099/CuramWS/services`. This page generates a page entitled "And now... Some Services" that lists your services and their operations. For each service there is a link to generate and view its WSDL. This verifies your deployed service(s) and it should now be available for invocation.

D.2.3 Using an External Client to Validate and Troubleshoot

You should begin validating the service on the server side first by using an external client because unless the web service class exists, deployment is setup properly, etc. a client failure may not be clearly distinguishable. To keep the path length and areas you may have to investigate for possible errors as small as possible you should use a known, working client to invoke 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 setup 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* and *Axis* 1.4:

- 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 will create and open 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 "Re-

quest 1" (default name) request. Double-clicking the request will open 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 Section D.3, *Tools and Techniques for Troubleshooting Axis2 and Axis 1.4 Errors* 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 Section 3.4, *Creating a Client and Invoking the Web Service*), but for *soapUI* you can just re-invoke the service and the subsequent request will likely not timeout; otherwise, see Section D.3, *Tools and Techniques for Troubleshooting Axis2 and Axis 1.4 Errors* for further techniques for resolving and addressing general web services errors.

D.3 Tools and Techniques for Troubleshooting Axis2 and Axis 1.4 Errors

The following highlight possible tools and techniques you might use in troubleshooting errors with *Axis2* or *Axis* 1.4 web services, but is not an exhaustive list. Also, the tools available to you may vary by platform and application server environment.

When trying to understand why a service has failed the following should be considered:

- Use a monitoring tool (e.g. *Apache TCPMon* or *SOAP Monitor*) to view the SOAP message traffic. It's easier to setup *TCPMon* (download from http://ws.apache.org/commons/tcpmon, unzip, & run; also available within *soapUI*), but it requires changing your client end points or your server port. Once setup, *SOAP Monitor* doesn't require any client or server changes, but does require special build steps for your WAR/EAR files. Apache ships *SOAP Monitor* as an *Axis2* module and see Appendix F, *Including the Axis2 SOAP Monitor in Your Web Services WAR File* 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. Here is an example *Apache log4j* properties file that would log verbosely in a `C:\Temp\axis2.log` file, you can adjust these settings to suit your requirements.

```
# Set root category
log4j.rootCategory=DEBUG, CONSOLE, LOGFILE

# Set the enterprise logger priority to FATAL
log4j.logger.org.apache.axis2.enterprise=FATAL
log4j.logger.de.hunsicker.jalopy.io=FATAL
log4j.logger.httpclient.wire.header=FATAL
log4j.logger.org.apache.commons.httpclient=FATAL

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=[%p] %m%n

# LOGFILE is set to be a File appender using a PatternLayout.
log4j.appender.LOGFILE=org.apache.log4j.FileAppender
log4j.appender.LOGFILE.File=c:/temp/axis2.log
log4j.appender.LOGFILE.Append=true
log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
log4j.appender.LOGFILE.layout.ConversionPattern=
  %d [%t] %-5p %c %x - %m%n
```

  You need to place the `log4j.properties` somewhere in the classpath of the *Axis2* or *Axis* 1.4 WAR file.
- Check the application server logs for more information.
- Turn on *log4j* tracing for *Axis2* or *Axis* 1.4 as this will most likely give you the most detailed picture of the web service processing or error at the time of the failure. This can be quite voluminous so use it with care.
- Turn on the *IBM Cúram Social Program Management* application *log4j* trace as this will also help to give you further context for the failure.
- Consider remote debugging the service running on the application server 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

and/or server are a high probability and appropriate steps should be taken; for the client see Section 3.4, *Creating a Client and Invoking the Web Service* 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. This is because the SDEJ employs a proxy wrapper object for ultra verbose (`trace_ultra_verbose`) tracing in order 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.

D.4 Avoid Use of 'anyType'

In general, it is best to avoid using `anyType` within your WSDL as it makes interoperability difficult at best, since both the service platform(s) and any client platforms must be able to map, or serialize/deserialize the underlying object.

WSDL will typically get generated with anyType when the underlying data type (e.g. object) cannot be resolved.

You may find with *Axis2* or *Axis* 1.4 that your WSDL will work 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.

You should begin generating 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.

Appendix E

Including the Axis2 Admin Application in Your Web Services WAR File

E.1 Introduction

This appendix shows you how to setup your *Axis2* web services build to include the *Axis2* Admin web application, which provides useful functionality for working with your *Axis2* environment.

Warning

The dynamic functionality of *Axis2* (e.g. hot deployment) isn't intended for production application server environments such as *WebSphere Application Server* and *WebLogic Server* and this functionality should not be attempted in these environments.

E.2 Steps for Building

While it is not recommended to use this application to dynamically modify a production environment the *Axis2* admin application can be useful for validating settings, viewing services, modules, etc. 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 (1.5.1) and unload it to your hard disk (e.g. `C:\Downloads\Axis2`).
- Create a location on your disk to contain the necessary *Axis2* artifacts; e.g.:

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

- Put the class files, `AdminAgent.class` & `AxisAdminServlet.class`, in the `C:\Downloads\Axis2\webapp\WEB-INF\classes\org\apache\axis2\webapp\` (based on the sample location above) directory from your *Axis2* binary download location into a jar file that you will place into the `WEB-INF\lib` directory in your newly created `C:\Axis2-includes` location (as above); e.g.:

```
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 may 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 shipped `axis2.xml` file to this location; e.g.:

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

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

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

  - Of course, for this to be secure the `axis2.xml` file would have to be secured in your development and deployed environments without access in the runtime environment to the *Axis2* configuration.

- Then, use the following properties when you invoke your web services ear target (see Section 4.4, *Building and Packaging Web Services*):

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

- Upon deployment you should then be able to access the Administration link via the *Axis2* "Welcome!" page menu (e.g. `http://localhost:9082/CuramWS2/axis2-web/index.jsp`).

Appendix F

Including the Axis2 SOAP Monitor in Your Web Services WAR File

F.1 Introduction

This appendix shows you how to setup your *Axis2* web services build to include the *Axis2 SOAP Monitor* module in your *Axis2* web services WAR file. The *SOAP Monitor* provides the ability to view SOAP message requests and responses, which can be useful in debugging.

F.2 Steps for Building

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 shipped with the `webservices2.war` has the necessary entries to support the *SOAP Monitor*. Beyond this the following additional steps are needed to enable this functionality:

1. Create a location on your disk to contain the necessary *Axis2* artifacts; e.g.

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

2. As per 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-1.5.1.mar
  org/apache/axis2/soapmonitor/applet/
```

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

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

4. The shipped `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 prior to deployment or for *WebSphere* in the deployed filesystem.

5. Then to access the SOAPMonitor you would use a URL like this: `http://localhost:9082/CuramWS2/SOAPMonitor`.

6. Unfortunately the applet doesn't give much information when there is an issue. If you see the error: `"The SOAP Monitor is unable to communicate with the server."`:

   - Ensure there is not a port conflict; the default as set in `web.xml` is 5001—if so, change that port.
   - This error may occur if you use *Microsoft® Internet Explorer 6*; if so, use a more current browser version.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing

Legal and Intellectual Property Law.

IBM Japan Ltd.

1623-14, Shimotsuruma, Yamato-shi

Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typograph-

ical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Dept F6, Bldg 1

294 Route 100

Somers NY 10589-3216

U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources.

IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products

should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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 the names and addresses used by an actual business enterprise is entirely coincidental.

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 IBM, 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. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication documents intended programming interfaces that allow the customer to write programs to obtain the services of IBM Cúram Social Pogram Management.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/us/en/copytrade.shtml.

Apache is a trademark of Apache Software Foundation.

Microsoft, Windows 7, Windows XP, Windows NT, Windows Server 2003, Windows Server 2008, Windows Explorer, Internet Explorer, Word, Excel, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Oracle, WebLogic Server, Java and all Java-based trademarks and logos are registered trademarks of Oracle and/or its affiliates.

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