How to Develop Flex Applications that Invoke Web Services

Applies to:

SAP NetWeaver CE, FlexBuilder 2.0, Windows XP; may work with other platforms

Summary

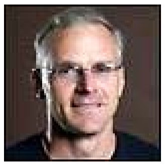
You can now develop your favorite Flex based client application that consumes SAP services. SAP has provided an Eclipse plug-in that generates ActionScript proxy classes for the invocation of a Web service represented by a WSDL file. This plug-in will generate classes that represent data types, operations, ports and services as defined in the WSDL file. The Eclipse plug-in can be deployed into Adobe Flex Builder, SAP NetWeaver Developer Studio, or any other Eclipse-based IDE.

Author: Axel Kratel

Company: SAP Labs, Inc.

Created on: 17 December 2007

Author Bio



Axel Kratel is a product manager specializing in the area of SAP NetWeaver Composition Environment and Java Development. Prior to working as a product manager, Axel worked as a developer and architect for the SAP HCM applications. He also spent several years away from SAP as the Product Manager for Borland's JBuilder development environment for J2EE. Axel has also served on the Executive Committee (EC) for the Java Community Process (See Executive Committee (EC) Members and contributed in the JSR 168 expert group in the definition of The Portlet Specification for Java. Axel has a PhD in Physics from the California Institute of Technology.

Table of Contents

Abstract

As of spring 2007, Adobe provides no adequate tools for the Flex application developer to invoke Web services, including SAP Enterprise Services. The current offering in the Flex/ActionScript environment is limited to dealing manually with raw SOAP messages. While this approach is acceptable for calling very simple Web services that take a couple of strings as an input and produce equally simplistic output, it is not suitable for invoking those services whose interfaces involve complex data structures, arrays and other complex constructs, as most real-world Enterprise Services do.

Adobe also provides an alternate mechanism for calling Web services via Flash Remoting MX. However, the major problem with Flash Remoting MX is that its interface is a black box. Flash Remoting MX consumes a WSDL file dynamically without any indication to the developer of what the structure of input data should be or what the structure of output data will be. This is left up to guesswork by the developer. While it may be possible to make such guesses for extremely simplistic services, using this approach for real-world Enterprise Services is not feasible.

This document tells you how to get started with the solution, which provides the following:

- There is an Eclipse plug-in that generates ActionScript proxy classes for the invocation of a Web service represented by a WSDL file. The plug-in generates classes that represent data types, operations, ports and services defined in the WSDL file. The Eclipse plug-in can be deployed into Adobe Flex Builder, SAP NetWeaver Developer Studio, or any other Eclipse-based IDE. The generated ActionScript code serves as the runtime for the execution of generated proxies.
- The Eclipse plug-in also generates MXML screens (i.e. data forms) to represent input arguments and output results for each operation defined by the selected WSDL file. Two MXML files are generated per each operation of a target Web service: one screen to represent the operation’s input arguments, another screen to represent the operation’s output results. These screens can be used to execute Web service calls right away, without extra coding, and can also be used by the developer as a starting point for further polishing, mending and composing them into a finished application.
- The plug-in also generates Javadoc-like documentation for generated classes that represent operations and data types defined by the selected Web service. This documentation is orders of magnitude more convenient to use for a developer than digging into raw WSDL file.
- Additionally, there is a generated runtime for binding screen elements to data structures. This runtime glues to the generated screens and allows interactive creation and editing of complex nested data structures as well as interactive browsing of such structures without a developer having to write a single line of code for that. Generated screens also include pre-cooked code for the invocation of Web services when the “Execute” button is clicked. Whenever a response from a Web service is received, the pre-cooked code in the screen automatically displays the output screen in the application window and displays received response data within this screen. This code can be overwritten by developers according to specific needs.

Getting Started: Creating Your First Project

Step 1. *Add proxy generator plug-in to IDE (for example, SAP NetWeaver Developer Studio, Eclipse, or FlexBuilder)*

- Unpackage **ASProxyGeneratorPluginUpdate.zip** to local file system.
- In your IDE (for example, SAP NetWeaver Developer Studio), choose Help --> Software Updates --> Find and Install --> “Search for new features to install” --> “New Archived Site” --> navigate to ASProxyGeneratorPluginUpdate.zip --> select it in “Update sites to visit” view --> Finish --> proceed to the dialogs installing the plug-in.
- Restart the IDE. Verify that ActionScript proxy generator plug-in has been installed. Help --> Product Details --> Plug-in Details. Scroll down to provider = SAP, locate the ActionScript WSDL Proxy Generator Plug-in there (plug-in id: com.sap.flex.ws.proxygen.ASProxyGeneratorPlugin).

Step 2. *Add runtime project to your workspace.*

The project is an ActionScript library. Unpackage **runtime.zip** and import it into your workspace. Open project properties, “Flex Library Build Path” category, “Classes” tab. Make sure that all classes are marked in “classes to include in the library” pane.

Step 3. *Configure HTTP proxy server in IDE.*

Proxy generator can read WSDL files from the Web (http server) or from local drive. If WSDL file for the Web service you will be using resides on the Web, or if it imports shared WSDL file from the Web, make sure your IDE has properly configured HTTP proxy settings. Go to Windows --> Preferences --> Internet --> Proxy Settings and configure the HTTP proxy server as needed.

Note: Flex Builder 2 uses an older version of Eclipse that does not have Proxy Settings in preferences. A workaround is to add proxy settings to the command line of Flex Builder, for instance:

```
FlexBuilder.exe -vmargs -Dhttp.proxyHost=proxy.wdf.sap.corp -
Dhttp.proxyPort=8080
-Dhttp.nonProxyHosts=127.0.0.1;localhost;sap.com;sap-ag.de;sap.corp
```

Step 4. *Create project for ActionScript proxies (optional).*

This step is optional. You can generate proxies right into your application project instead if you prefer. It may be more trouble-proof though to keep hand-written code (application project) separate from auto-generated code (proxies project) to avoid confusion and accidental overwriting in case proxies are regenerated.

Create Flex library project that will contain generated ActionScript proxies (File --> New --> Flex Library Project). Leave “main source folder” blank. Leave project initially empty. Open its properties, category “Flex Library Build Path”, click “Add Project”, add “runtime” project to build path.

Step 5. *Build ActionScript proxies.*

Right click on (by now, empty) empty proxies project. Select “Generate ActionScript proxies from WSDL” from the context menu. Enter local file path or URL for the WSDL file. Enter ActionScript package name for the proxies, e.g. “com.sap.mycomponent.ws.proxy”. Click “Generate”. Project will be populated with generated ActionScript classes and help files. If you do not see them, try refreshing the project. Open project properties, category “Flex Library Build Path”, tab “Classes”. Select all classes in “Classes to include in the library” pane. Build the project.

The plug-in also creates Javadoc-like documentation for the generated classes. To browse the documentation, open index.html file as the starting point. Documentation describes classes representing Web services, ports, operations and data types defined in the WSDL file, and also indicates for each operation names of generated input and output MXML screens.
Step 6. *Create application project.*

File --> New --> Flex Project. Open project properties, “Flex Build Path” category, “Library path” tab. Add “runtime” project and proxies project to the list of referenced projects.

- Now you are set up for developing your Flex application invoking Web services.

Developing the Application

Procedural Programming

Find the Web service you need to call in the “Services” section of the generated HTML documentation. Within the service description find the port that contains the operation(s) you want to invoke.

The Flex programming model allows only asynchronous invocation of outside components, such as Web services. Synchronous calls are discouraged and, in fact, are prohibited on technical level. For this reason proxy generator generates *asynchronous* proxies. For each operation defined by web service, three methods are generated in the class representing service port:

`*operation(*arguments) : void`

`*operation*_Result(results) : void`

`*operation*_Fault(fault) : void`

Instead of coding

`results = port.*operation*(arguments)`

...as one would do in a synchronous invocation model, you should instead invoke `port.*operation*(arguments)` that does not return any results right to the caller having signature of “void” type. Instead it places asynchronous call to Web service and returns to the caller without providing any results immediately. Some time later a response arrives and gets decoded by the runtime that would in turn invoke `*operation*_Result(results)` callback. If call results in a fault, `*operation*_Fault(fault)` callback will be invoked instead.

For each Web service port defined in the WSDL file, the proxy generator creates proxy class containing three ActionScript methods per each operation of the port: `*operation*, *operation*_Result and *operation*_Fault.` You should create callback handler class by:

- subclassing proxy class of the port
- overriding `*operation*_Result and *operation*_Fault` methods for the operations he intends to use within the created callback handler class.

To simplify this, the proxy generator creates a template with overridden methods. This template has file name `*portname*_MyApp.as` and can be used as a starting point. Simply copy the template to your project, rename it and edit it. DO NOT edit the template inside the proxy project as it will be over-written if you re-run the generator and all your added code will be lost.

Here is an example of application class invoking Web service with method named “Echo”.

```
package my.test
{
import my.test.proxy.*;

public class MyClient
{
    public function doTest() : void
    {
        var service : WSASimpleRPC_Service = new WSASimpleRPC_Service();


service.setAuthenticationMethod(com.sap.flex.ws.runtime.Service.AUTH_WSSE);
        service.setUsername("landadmin");
        service.setPassword("sdc123");
        myCallbackHandler = new MyCallbackHandler();
        service.getWSAPort(myCallbackHandler);

        var inpart2 : ELEM_A = new ELEM_A("a", "b", 123);
        var inpart1 : TypeD = new TypeD();
        inpart1.dt = new Date();
        inpart1.c1 = new TypeC1("s1", "b", "c", 123);
        inpart1.c2 = new TypeC2("s1", "b", "c", 123, "d", "e", 123);
        inpart1.c = new Array();
        var vc : TypeC;

        vc = new TypeC;
        vc.s = "s[1]";
        vc.as_1 = ["as11", "as12", "as13"];
        vc.i = 123;
        vc.ai = [11, 12, 13];
        vc.s2 = "s2[1]";
        inpart1.c.push(vc);

        vc = new TypeC;
        vc.s = "s[2]";
        vc.as_1 = ["as21", "as22", "as23"];
        vc.i = 223;
        vc.ai = [21, 22, 23];
        vc.s2 = "s2[2]";
        inpart1.c.push(vc);

        myCallbackHandler.Echo(inpart1, inpart2);
    }
} // end class declaration
} // end package declaration



package my.test
{
import com.sap.flex.ws.runtime.WebServiceFault;
import my.test.proxy.*;

public class MyCallbackHandler extends my.test.proxy.WSAPort_Port
{
    override public function Echo_Result(outpart1 : my.test.proxy.TypeD) : void
    {
        Alert.show(“Value of response.c1.b: ” + outpart1.c1.b, “MyClient received
web service response”);
    }

    override public function Echo_Fault(fault :
com.sap.flex.ws.runtime.WebServiceFault) : void
    {
        super.Echo_Fault(fault);
    }

} // end class declaration
} // end package declaration
```

Class MyCallbackHandler subclasses web service port proxy class WSAPort_Port and was initially cloned from template class WSAPort_Port_MyApp.

First section inside doTest() instantiates object representing a connection to the service.

```
var service : WSASimpleRPC_Service = new WSASimpleRPC_Service();

 service.setAuthenticationMethod(com.sap.flex.ws.runtime.Service.AUTH_WSSE);
 service.setUsername("landadmin");
```

```
        service.setPassword("sdc123");
```

Service object contains connection context, including authentication data and method, endpoint address, call timeout duration and SOAPAction value (in case default value specified in WSDL file is overridden by application developer). Once set, these values should be kept immutable. If you need to invoke the same service using different sets of described values, create multiple instances of service object.

Three authentication methods are implemented now. AUTH_NONE performs no authentication. AUTH_WSSE sends authentication data inside SOAP request header[1]. AUTH_HTTP_BASIC uses basic HTTP authentication; you must set up Flex proxy server in order to use AUTH_HTTP_BASIC.

Next, create callback handler object and associate it with the service connection:

```
        myCallbackHandler = new MyCallbackHandler();
        service.getWSAPort(myCallbackHandler);
```

You are now ready to invoke the operation. Create arguments (inpart1 and inpart2, in the example), fill in the values for them, and place a call:

```
        myCallbackHandler.Echo(inpart1, inpart2);
```

Eventually a response will be received from web service. If call was successful, method Echo_Result will be invoked. If call failed, Echo_Fault will be invoked instead. In the latter case you can process the response as you see fit, but the default handler will display error message with information contained in the fault object.

Screens Programming

Besides generating ActionScript proxy classes for Web services defined in the WSDL file and documentation for those classes and services, the plug-in also generates UI screens that represent input and output of each operation.

Two MXML files are generated per each operation of each target Web service: one screen to represent operation's input arguments, another screen to represent operation's output results. These screens can be used to execute Web service calls right away, without extra coding, and can also be used by the developer as a starting point for further polishing, mending and composing provided screens into a finished application.

To create those screens, check "Create MXML screens for web service operations" checkbox in plug-in's dialog box.

After clicking "Generate" you will see a bunch of MXML files appearing in your target project.

If you generated proxies into a library project separate from your application project, you would need at this point to move the following files and folders from the library project into main application project:

folder *images* – this folder contains icons for the buttons used by the generated UI and other similar elements

folder *sap* – this folder contains SAP style elements referenced from mx:Style header of Application.mxml

**.mxlm* files

There will be two MXML files per each operation (except in the rare case of one-way operations that do not produce output) and one main file called *Application.mxml*. *Application.mxml* contains screen dispatcher code. Right-click on *Application.mxml* and select "Set as Default Application".

Now select *Application.mxml* by clicking on it. You are ready to run the application.

To run it, click big green button with white arrow on main toolbar or select Run -> Run Application from main menu.

First screen will come up. You can invoke a Web service without having written a line of the code yet.

[1] WSSE authentication requires SAP J2EE 710 SP1 Patch 6 or later (broken in earlier 710 releases). Has not been tested against NW04 (640) or NW04s (645).

Hello, Application!

Let's have a look inside Application.xml. At the top you will notice blocks referencing each defined screen, in the form:

```
<mx:VBox id="ScreenName_Container" visible="false" width="0" height="0">
    <local:ScreenName id="ScreenName_var"/>
</mx:VBox>
```

Remove references to those screens you do not intend to use and delete their MXML files.

Method *onInit* defines what screen is to be displayed when application starts:

```
public function onInit() : void
{
    switchToScreen("ScreenName", null);
}
```

Second argument to *switchToScreen* specifies data structure that screen is to render. This data structure normally is either an input or output WSDL message of web service operation. Check generated help file for web service port, find the operation description there, and under this description look up for "WSDL input message class" and "WSDL output message class" links that point out to the data types expected by the input/output screens for the given operations. ActionScript classes for these data types have already been generated by the plug-in. When *null* is passed as the second argument to *switchToScreen* it means that screen controls are to be initialized with blank values.

Now let's have a look at the structure of each particular screen.

Plug-in currently uses the following types of controls:

DataGrid control:

DataGrid is used to represent arrays – either:

- arrays of simple elements
- or arrays of structures
- or arrays of arrays.

-

For each DataGrid there will also be two buttons generated: one to add rows to the grid, another to delete currently selected row. You can remove those buttons if you want to disallow user to remove or delete records, but if you do so be also sure to modify or remove *uib.setButtons* binding (described below).

TextInput control:

You can edit the MXML file and change control type from TextInput to Text, TextArea or RichTextEditor, while retaining the same control ID and data binding. Runtime will recognize these types and handle data mapping appropriately.

If RichTextEditor is used, there is some (currently, rudimentary) recognition of whether input data is HTML and if so, handling it like that.

CheckBox control:

Boolean (xsd:boolean) fields are mapped to CheckBox controls. CheckBox controls can be substituted with Text, TextInput or TextArea controls by editing MXML file and retaining control ID and its data mapping; in this case value is represented as “true” / “false” text string.

Date control:

Displays data of xsd:date type or derivative types. Can be substituted with Text, TextInput or TextArea controls; in this case SOAP format for time representation as a string will be used.

Data elements typed *xsd:time* and *xsd:dateTime* are also mapped to Date control, but of course the latter is unable to display the time part, and displays only date part. There is currently no solution for this, as required control functionality for time picking is missing.

Besides changing control type (within limits mentioned above), developer can also change control status from editable to read-only through appropriate tags in MXML file; runtime will retain this setting.

Binding of screen controls to data fields is innervated by handler of UIBinding type instantiated inside screen MXML file. Bindings are declared inside method *bindScreen()* of auto-generated MXML file. Instance of UIBinding class is named *uib* and has the following methods:

```
uib.bind(controlID, dataPath, controllerID)
```

“controlID” is the ID of screen control, as defined by the value of “id” attribute in control’s tag in the MXML file.

“dataPath” defines data path from root level of the structure that the whole screen displays to the data element that given control should display. Path elements are separated by dot. For instance, if screen displays input message class for WSDL operation and this class has three elements {a, b, c}, where “a” and “b” are scalar elements and c is a structure with two fields {x, y}, then the following data paths exist: “a”, “b”, “c.x”, “c.y”. Arrays are not indicated in any special way in data path syntax, i.e. there is no brackets or other constructs to represent indexing. Instead, indexing is presumed implicitly by mapping path component to DataGrid.

“controllerID” points to parent DataGrid. In most cases this will be null. However if data field represents a child of a structure displayed in the data grid, then “controllerID” will pointing to parent DataGrid and will be MXML “id” of that parent DataGrid control.

If you remove a control from the screen, be sure to remove binding statement for it as well.

```
uib.addColumns(columnID, dataPath)
```

Used for DataGrid controls and binds columns of the control to data elements. *addColumn* statements should immediately follow *uib.bind(…)* statement for given DataGrid, they apply to the control declared through the most recent *uib.bind*.

“columnID” is an ID of the column as declared by mx:DataGridColumn tag, dataField attribute of it.

dataPath describes data path of the element to be connected to the column as *relative* to the data path bound to the control itself. It can be blank as well.

For example, if control is bound to data path *“a.b”* via binding declaration *uib.bind(controlID, “a.b”, controllerID)* then:

- “b” must be an array; attempt to bind scalar field to DataGrid will cause an exception
- if “b” is an array of simple scalar values (e.g. strings, numbers, booleans or dates) than these values can be displayed in the column via *addColumn(…, null)*
- if “b” is an array of structures X and that structure has field “c” of some simple scalar type, then element “a.b.c” can be bound to the column via *addColumn(…, “c”)*
- if “b” is an array of structures typed X and that structure has field “c” that in turn is a structure typed Y with field “d” that is simple scalar type, then element “a.b.c.d” can be bound to the column via *addColumn(…, “c.d”)*

`uib.addPseudoColumn()`

When mapping an array of structures to the screen, and every field in the structure is complex itself, proxy generator will create one-column DataGrid and request it to be populated with dummy records (“Record 1” … “Record N”), one per array entry, via *addPseudoColumn()* binding.

In addition, child controls will be created for each field of the structure and will have there controllerID set to parent DataGrid via *uib.bind(controlID, dataPath, dataGridID)*.

You may prefer to refine auto-generated structure as follows:

- pick a set of subfields in nested structures to be displayed in the DataGrid itself, rather than children controls
- remove auto-generated *mx:DataGridColumn* tag (for dataField=“Index”)
- remove *addPseudoColumn()* binding
- add *mx:DataGridColumn* tags to define columns of DataGrid that will display selected subfields
- add *addColumn()* bindings that map each of the declared columns to corresponding data fields

In the future proxy generator may be enhanced to pick simple subfields nested deeply within complex fields automatically and reduce the need for manual design.

`uib.setButtons(addRowID, deleteRowID)`

Declares buttons to add or remove records from DataGrid. This statement must be placed right after *uib.bind()* declaration for DataGrid control since it is applied to the control declared in most recent *uib.bind()* binding.

addRowID is MXML ID of “Add New Record” button. Clicking this button will create new blank record at the bottom of DataGrid and insert it in the underlying data structure as well

deleteRowID is MXML ID of “Remove Record” button. Clicking this button will remove currently selected record from DataGrid and underlying data structure.

In case you decide to delete

`uib.setDisplayName(dataPath, displayName)`

Sometimes runtime may need to display user-readable name for given data element. This may be required, for example, to display messages during data validation. *setDisplayName* allows developer to provide user-readable strings identifying data elements. For instance: *uib.setDisplayName(“patientData.SSN”, “Social Security Number”)*.

Auto-generated MXML file contains the following methods:

```
public function initializeScreen(screenController : Object, args :
WSDL_MESSAGE_CLASS = null) : void

{

    this.screenController = screenController;
```

```
        if (args == null)
            args = new WSDL_MESSAGE_CLASS();

        uib = new com.sap.flex.ws.runtime.screens.UIBinding(this, args);
        bindScreen();

        uib.toScreen();
    }
```

This method is invoked when screen receives control and displayed by the application. Actual data structure to be displayed by the screen is passed as *args* argument. Method creates instance of UIBinding, named *uib* and passes *args* to it; then screen-to-data bindings are defined inside *bindScreen()* method. Finally, *uib.toScreen()* causes screen controls to be initialized from data held in *args*.

Note that *uib* makes private copy of *args*, so original *args* is not modified as user edits the data.

```
    private function bindScreen() : void
```

This method defines screen controls binding to data and contains a set of *uib.bind*, *uib.addColumns*, *uib.addPseudoColumn*, *uib.setButtons* and *uib.setDisplayName* statements as described above.

Screens that implement input side of the operations will have “Execute” button in their auto-generated MXML files. “Click” handler of this button bind to the following method:

```
    private function execute() : void
    {
        var args : WSDL_MESSAGE_CLASS = uib.fromScreen() as WSDL_MESSAGE_CLASS;
        if (args == null)  return;
        var result : WSDL_RESPONSE_CLASS = new WSDL_RESPONSE_CLASS();
        var service : SERVICE_CLASS = new SERVICE_CLASS();
        // service.setTargetEndpointAddress("...");
        // service.setAuthenticationMethod(service.AUTH_WSSE);
        // service.setUsername("...");
        // service.setPassword("...");
        var faults : Array = [...];
        service.invokeOperation("port-name", "port-namespace", "method-name", "how",
                                args, result, faults, this,
                                "onResult", "onFault");
    }
```

This method gathers data from screen and performs actual web service call. Data are gathered from screen via *uib.fromScreen()*. Note that object returned by *uib.fromScreen()* will be of the same *WSDL_MESSAGE_CLASS* as *args* in *initializeScreen()*, but it will be a different instance. *uib.fromScreen()* performs data validation against restrictions specified in data model emedded in service description (WSDL file). It will indicate to the user any incorrect data and suggest them to correct it. Validation is impemented only partially at this point. If validation fails, *uib.fromScreen()* will return *null*.

If you want to override service endpoint address and use address different from one defined in WSDL file, or if you want to provide authentication data, uncomment corresponding service.setXXX methods and fill in the values.

Actual service invocation is performed by *service.invokeOperation*. Invocation is asynchronous. Once a response is received, runtime will invoke *onResult* method:

```
    public function onResult(result : WSDL_RESPONSE_CLASS) : void
    {
        screenController.switchToScreen("output-screen-name", result);
    }
```

This method, by default, requests screen controller to display output screen in place of current input screen, and then display received “result” data in the output screen.

If service invocation was unsuccessful, *onFault* method will be invoked instead of *onResult*:

```
public function onFault(fault : com.sap.flex.ws.runtime.WebServiceFault) : void
{
    com.sap.flex.ws.runtime.Call.defaultFaultHandler(fault);
}
```

By default, this method will display error message.

There are some additional methods in auto-generated MXML file:

```
private function validate(event : Event, kind : String) : void
private function addRow(event : Event, dataGrid : DataGrid) : void
private function deleteRow(event : Event, dataGrid : DataGrid) : void
private function dataGridChange(event : Event) : void

protected function forcedReferences() : Array
```

These methods perform auxiliary functions.

Limitations

Current limitations of the runtime include:

- X509 certificate authentication is not implemented.
- Circular references in the structures being marshaled out are not properly handled (i.e. structure A referring to structure B, but B also referring to A). Runtime will throw exception if such a call is attempted.

Current limitations of proxy generator:

- Only "xsd:sequence" and "xsd:all" are supported in complex type definitions.
  "xsd:choice" is not supported yet.

- Only the following XSD data types are supported:

```
string
integer
int
unsignedInt
unsignedInteger
positiveInt
positiveInteger
negativeInt
negativeInteger
nonPositiveInt
nonPositiveInteger
nonNegativeInt
nonNegativeInteger
byte
unsignedByte
short
unsignedShort
long
unsignedLong
decimal
boolean
date
time
dateTime
double
float
```

```
ID
normalizedString
token
language
anyURI
```

The following data types are not supported:

```
base64Binary
hexBinary
NOTATION
QName
duration
gDay
gMonth
gMonthDay
gYear
gYearMonth
ENTITIES
ENTITY
IDREF
IDREFS
Name
NCName
NMTOKEN
NMTOKENS
QName
```

- xsd:list is not supported in either simple or complex type definition. Use multiplicity instead. (Eclipse/SAPIDE WSDL editor does not support xsd:list either.)
- <element ref="..."> references are not supported.
- MIME tags are not supported (mime:content, mime:mime-xml, mime:multipart-related).
- HTTP binding tags are not supported (http:address, http:binding, http:operation, http:url-encoded, http:url-replacement).
- Various restriction tags are not handled (ignored). This applies to tags that describe data precision, length, range of values, string patterns, enum values etc.
- xsd:union is not supported.
- xsd:attributeGroup is not supported.

Related Content

- Download WSDL to ActionScript Proxy Generator for Adobe FlexBuilder
- Connecting SAP Enterprise Services with Flex Controls
- Blog on Developing Flex Applications that Invoke SAP Services

Copyright