Connecting SAP Enterprise Services with Flex Controls

Applies to:

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

Summary

This paper describes a novel four-step approach to developing Enterprise Service-based applications with an Adobe Flex User Interface. This approach allows application developers to quickly bind Enterprise Services to Flex Controls without implementing any code. It comes with an ActionScript Proxy Generator for Enterprise Services, a generator for Flex User Interfaces and a Javadoc-like documentation generator. The four steps are as follows:

1. Select an Enterprise Service from the SAP NetWeaver Enterprise Service Repository.
2. Using an Eclipse plug-in, generate ActionScript proxy code and a raw Flex user interface for the selected Enterprise Service.
3. Modify the user interface to your liking.
4. Configure the application so that it uses a production Enterprise Service and deploys the application on a production server.

This approach complements the Proxy Generator offerings for Enterprise Services in SAP NetWeaver for other languages such as Java and the SAP NetWeaver Interactive Forms generator.

Author: Axel Kratel

Company: SAP

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

Introduction

What is a Web service? A year or two ago, that was the most common question asked at any occasion where Web services were discussed. Today, most people in the IT industry as well as our customers have enough of a common understanding of Web services not to have to ask that question anymore. Once we knew what Web services were, we started asking ourselves: What can I do with Web services? What are they good for?

SAP entered that discussion with the Enterprise Services Architecture, a high-level blueprint for how a customer can build a service oriented system landscape, benefiting from Web services technology to increase the value of the IT platform while dramatically reducing the total cost of ownership.

In this paper we will explain how Enterprise Services can easily be bound to Adobe Flex controls, thus, making the development of applications on top of Enterprise Services much more efficient.

This paper is structured as follows:

- **Section 1**: Introduces the topic
- **Section 2**: Describes the business side of the problem we are addressing
- **Section 3**: Provides an overview of the solution
- **Section 4**: Describes the technologies that are used in the solution
- **Section 5**: Explains the technical solution in more detail
- **Section 6**: Describes two example scenarios
- **Section 7**: Provides a summary and an outlook

Statement of the Problem

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.

In summary, while Adobe Flex is a powerful user interface technology it is very hard for application developers to efficiently make use of Flex in their applications, especially when creating Enterprise Service-based applications.

Overview of the Solution

The solution to this problem is as follows:

- 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.

How to Create and Deploy a Web-Service-Based Flex Application

After deploying the Eclipse plug-in into FlexBuilder or the SAP NetWeaver Developer Studio, you create a Flex application using an Enterprise Service in a simple four-step process:

Step	Process
Step 1	Identify the Enterprise Service
Step 2	Invoke the ActionScript Proxy Generator and the Flex generator
Step 3	Review and refine the generated Flex UI by eliminating unnecessary fields, changing properties of UI elements, re-arranging UI elements
Step 4	Overwrite properties, for example, with host and port information pointing to valid Enterprise Services

Technologies Used

The solution comes with the following technologies:

Technology	Description
SAP NetWeaver Enterprise Service Repository	The Enterprise Services Repository (ESR) is the central information repository about enterprise services in SAP's enterprise SOA ecosystem. In technical terms, the ESR is a metadata repository, meaning that it stores data that describes other data. The metadata inside the ESR is then used by development tools, modeling tools, operational management tools, and by other services to help them do their jobs. The ESR has also been given the job of storing descriptions of business objects and models that show how services work together in process components. It is important to remember that services are not implemented in the ESR. The ESR holds descriptions of services that are then implemented in applications that are separate from the repository.
SAP NetWeaver Developer Studio or other Eclipse-based IDE such as Adobe FlexBuilder 2.0	SAP NetWeaver Developer Studio builds on the open-source Eclipse framework. As a starting point for all Java development tools and the integration basis for all infrastructure components, SAP NetWeaver Developer Studio supports efficient development of Web Dynpro, Web services, and Java/J2EE business applications as well as Java projects on a large-scale basis for both SAP technologies and standard technologies.
ActionScript proxy generator Eclipse plug-in provided by SAP	Based on a valid WSDL file, the Web Service Proxy Generator generates a platform-specific proxy. Based on standardized WSDL descriptions, development of a Web service client is very easy. Using the WSDL file as input, a Web service client proxy is generated. The proxy allows the application developer to focus on business functionality, while technical aspects like creating a valid SOAP message are automatically done by the proxy implementation. The Proxy Generator for ActionScript is used to generate ActionScript classes for application programming. The ActionScript Proxy Runtime converts the used ActionScript classes into XML messages. These XML messages are sent to the server which hosts the Enterprise Services using the HTTP protocol. Additionally, the generator generates screens which can be used for invoking the Enterprise Services and displaying their results.

Technologies in Detail

In this section, we go into more detail on the key technologies upon which the solution was built:

- **SAP NetWeaver**
- **SAP NetWeaver Enterprise Services**
- **Adobe Flex**

SAP NetWeaver

SAP NetWeaver is the technological foundation for all SAP products. It is a service-oriented application and integration platform: SAP NetWeaver acts both as the interface between SAP applications and as their runtime environment. It interoperates with and can be extended using Microsoft's .NET or Sun's J2EE. SAP NetWeaver embraces Internet standards such as HTTP, XML, and Web services. It also enables Enterprise Services Architecture (ESA), SAP's blueprint for service-oriented business solutions.

Key parts of SAP NetWeaver used by our solution include:

- **SAP NetWeaver Portal**
  The people-integration layer of SAP NetWeaver that helps create software that brings together all the data and tools needed into one consistent user interface. The Portal supports Visual Composer and is used for connections to enterprise services (for example, BAPIs).
- **SAP Business Intelligence**
  The information-integration layer of SAP NetWeaver, a comprehensive end-to-end data warehouse solution with optimized structures for reporting and analysis.

SAP NetWeaver Enterprise Services

A Web service is any interface that is described by and can be called through the Web services standards. The standards stack is continuously growing, with standards, such as SOAP and WSDL, being widely accepted. In the Enterprise Services Architecture, we assume that all communication between components is based on Web services, and therefore all services are Web services.

An Enterprise Service corresponds to a business event, independent of any applications. The business event is described in business terms and is typically stable over a very long period of time. For example, receiving an order from a customer is a business event that can be described using terms such as customer, order, product, price, quantity, etc.

The user interface of today's Web-based business applications looks very different from the green-on-black terminal screens of yesterday's mainframe applications. However, the way the user interacts with the application has not really changed a lot. In both yesterday's and today's business applications, the interaction is mainly predefined by the system and centered on transactions. Although the data the knowledge worker is handling is essential to the business applications, the design of the transactions usually has little or nothing in common with the way knowledge workers actually perform their tasks.

With the advent of Web services and service-oriented architectures such as the Enterprise Services Architecture, we now have the toolset to actually build new kinds of user interaction components. First of all, since Web services technology enables platform interoperability, we can choose the front-end tool that is appropriate for each user interaction scenario we want to support. Perhaps a professional user who is familiar with an application prefers a powerful transaction implemented in the GUI technology specifically developed for this application with many powerful features, while an occasional user prefers to perform tasks out of a familiar environment such as the email inbox. All popular front-end tools today support or soon will support Web-services-based communication, so technology no longer is a major obstacle.

Because we now will have access to a catalog of available services and these services are defined in such a way that they can be used in different contexts, we can build user interaction components (or service consumers) that are logically independent of the underlying applications (or service providers). This is the definition of loose coupling. It allows us to combine the services we need for a particular task without having to change the back-end application.

State-of-the-art user interaction technologies such as SAP's Web Dynpro fully support the development of user interaction components based on services. Services can be discovered in a registry and proxies are

automatically generated. The services can be orchestrated into the appropriate interaction model, and additional logic added as needed. The resulting user interaction component can be generated for different run-time environments, again giving us the choice of the appropriate technology.

Adobe Flex

Adobe Flex is a presentation-tier framework and server that enables the development and deployment of Rich Internet Applications. These applications combine the richness and responsiveness of desktop applications with the broad reach of Web applications.

Flex overcomes the traditional limitations of HTML-based user interfaces. Flex applications support rich user interface components and direct object manipulation such as drag-and-drop. They provide visual cues and transitions to help the user make sense of state and data changes in the application. Flex applications can also support real-time data push and rich media streaming.

Flex is used to improve the user experience in a wide range of applications, including dashboard, business process automation, self-service, and commerce applications.

Flex applications are stateful. In other words, the state of the application can be maintained at the client-side. Flex also enables client-side data manipulation (for example, sorting and filtering) and caching. These features can significantly minimize the number of server roundtrips. This results in more responsive applications that also reduce the load on the network and on the server.

Rich Internet Applications built on the Flash platform, and their associated benefits, are not new. However, until recently, these applications could only be built using the traditional Flash IDE whose primary focus is not on traditional application development. Flex addresses the specific requirements of enterprise developers by providing a programming model that supports standard software engineering methodologies and design patterns, and that integrates with the existing enterprise infrastructure. Flex enables the development of large scale applications that are easy to develop, debug, and maintain.

The Flex programming model is based on the combination of MXML and ActionScript 3.0. MXML is a declarative, XML language used to define the user interface of the application. ActionScript 3.0 is an ECMAScript-compliant strongly-typed and object-oriented language that is used to implement the non-visual aspects of the application (client-side logic).

The source code of the Flex application (XML documents and ActionScript classes) is compiled into Flash bytecode (SWF file). This allows the Flex application to be delivered to the ubiquitous Flash Player running inside the browser at the client-side using the traditional lightweight, cross-operating system, and cross-browser Web deployment model.

Class Library

Flex features an extensive library of user interface components, including DataGrid, Tree, TabNavigator, Accordion, Menu, media controllers, and a wide variety of charting. Flex components are customizable – using cascading style sheets (CSS). New components can also be created from scratch, by extending existing components (using traditional object-oriented inheritance), or by aggregating other components. Flex components are available as tags in the MXML language, and can also be instantiated programmatically in ActionScript.

Runtime Services

Flex focuses on the presentation tier of the application, and provides data services to connect to the server using SOAP-based Web services, XML over HTTP, and remote method invocation into Java objects. Visual Composer leverages these services to provide transparent connectivity to SAP and non-SAP, OLAP and Relational Data Services. Other runtime services available in Flex include dynamic compilation, caching, as well as integration with the session management and security infrastructure of the underlying application server.

Use Cases

This section describes two use cases that outline the process of developing and modifying an Enterprise Services based Flex application. The first scenario is an artificial application for the sake of simplicity, the second is a more complex, and realistic Enterprise Service based business application. The scenarios demonstrate how the Eclipse plug-in and Enterprise Service could be used by a developer to develop a Flex application incrementally.

Scenario 1

Summary of Scenario

The first scenario is a simple calculator application. It takes two integers as input and calculates the sum. The result is printed out on a second screen. The Web Service Definition in WSDL is shown in Figure 1:

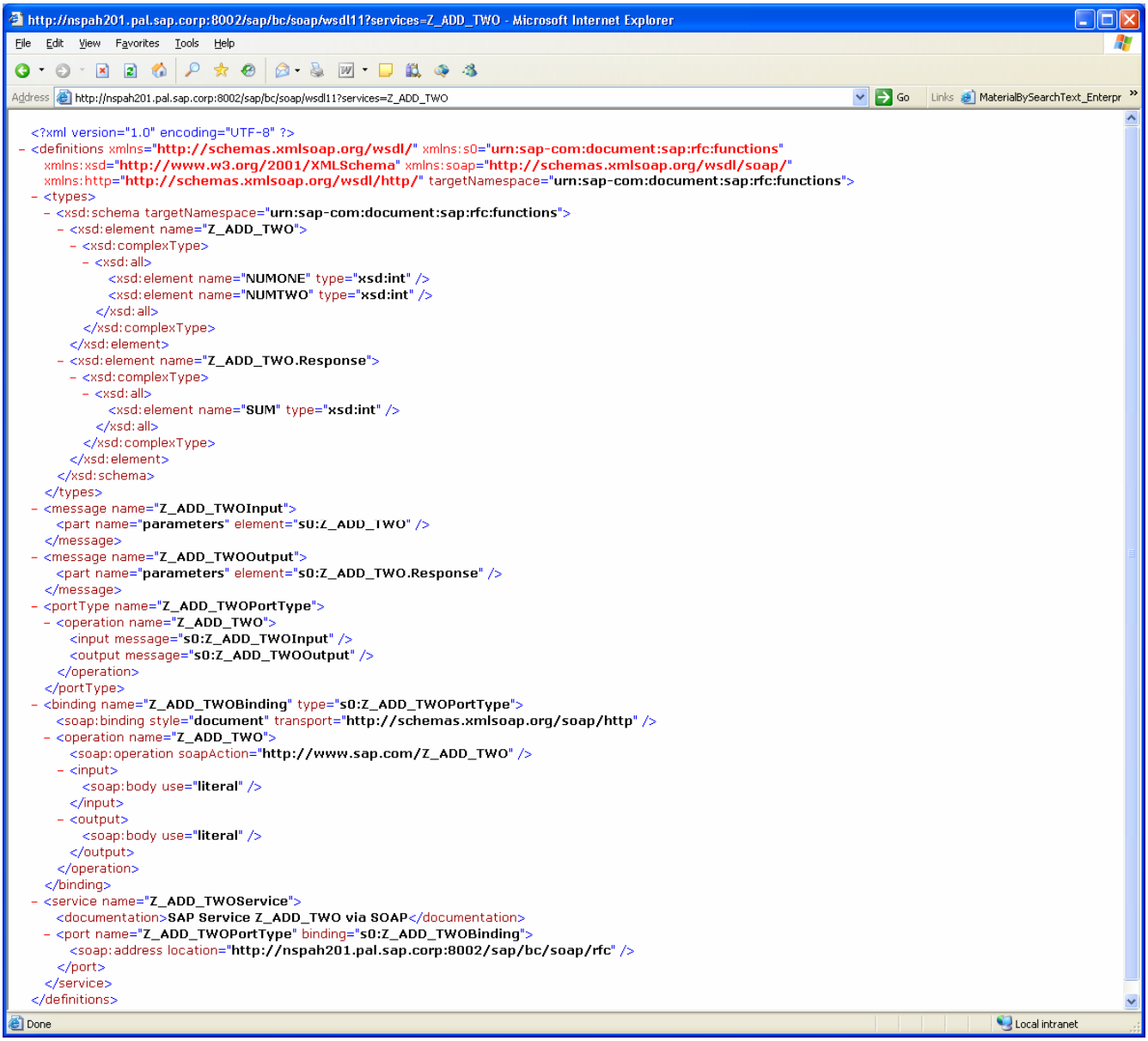


Figure 1 – Simple Calculator Web Service

Implementation of the Scenario

To use the ActionScript proxy generator, the user enters the URL of the WDSL in a wizard in Eclipse. This is shown in Figure 2.

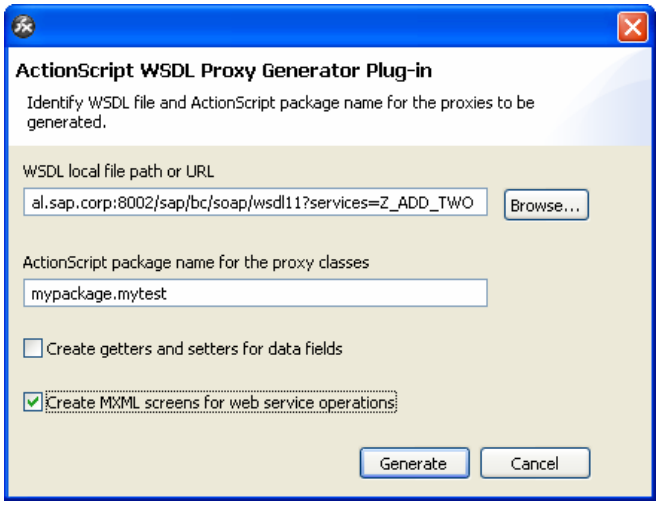


Figure 2 – Entering the Web Service URL in Eclipse

The ActionScript generator, which is implemented as an Eclipse plug-in, generates the raw User Interface for the application which is shown in **Figure 3**.

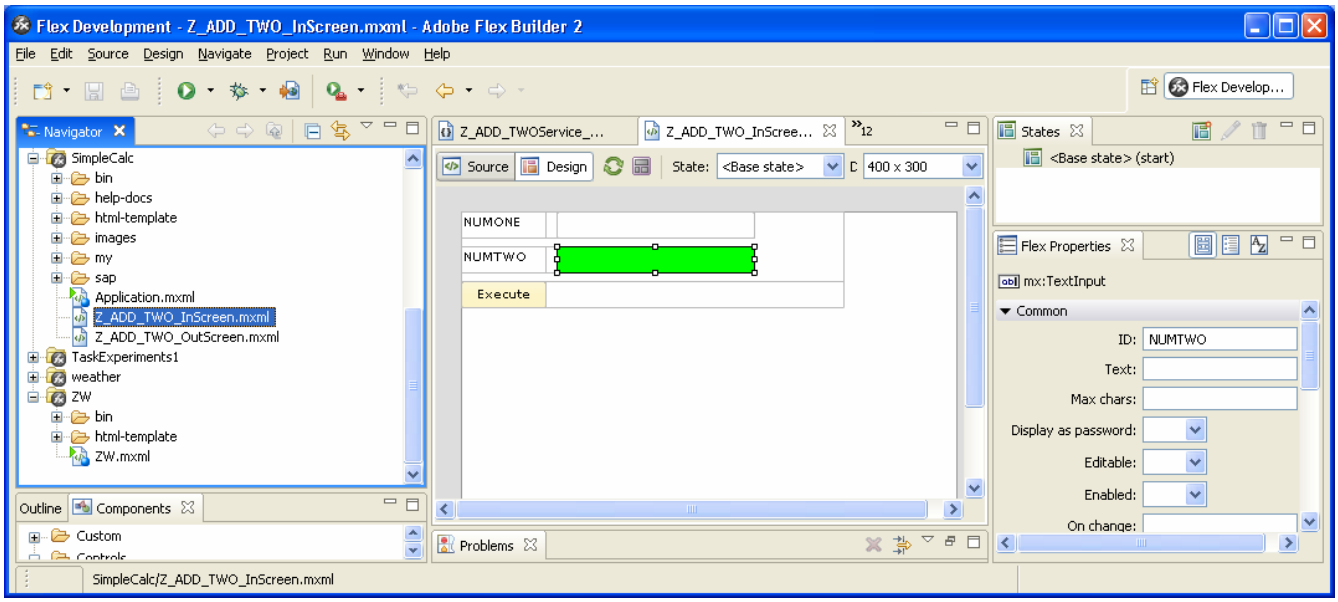


Figure 3 – Generated User Interface for Calculator Web Service

Figure 3 shows how a developer can change the visual appearance of the applications. An application developer, who uses FlexBuilder, can modify properties of screen elements before the application is run.

When all the property changes are entered the application can be run directly from within Eclipse. The finished screen for entering the data and the result screen displaying the result are shown in Figure 4.

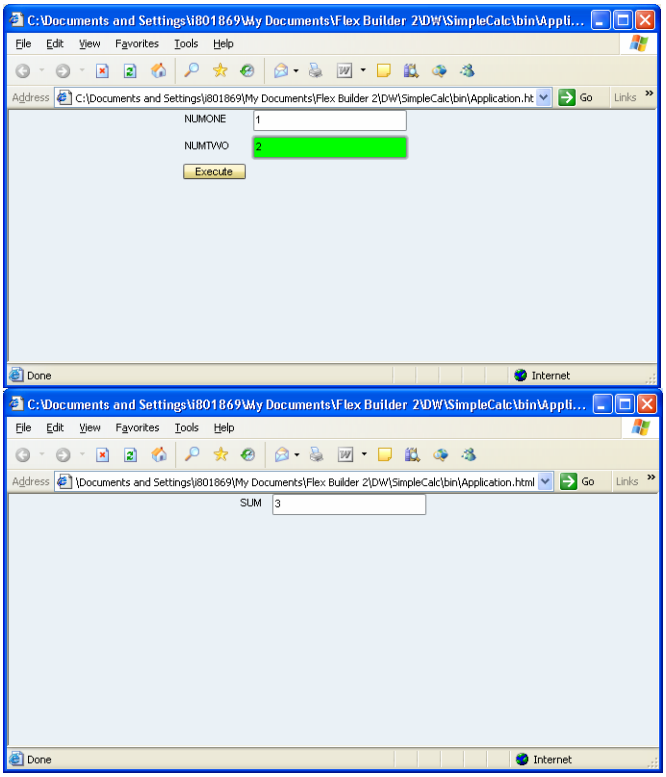


Figure 4 – Running the application – screen 1 is for entering data, screen 2 is for outputting the result of the Web service invocation

Scenario 2

Summary of Scenario

The second application is from the area of Corporate Performance Management (CPM). The application is the CPM Planning Console, which allows for creating and maintaining planning rules. The implementation is based on SAP NetWeaver BI. The functions that are needed for the application are exposed through a Java API which itself is exposed as a set of Enterprise Services. The ActionScript proxy generator is applied to these Enterprise Services so that they can directly be called from a Flex application. An example of a CPM Web Service Definition in WSDL is shown in Figure 5:

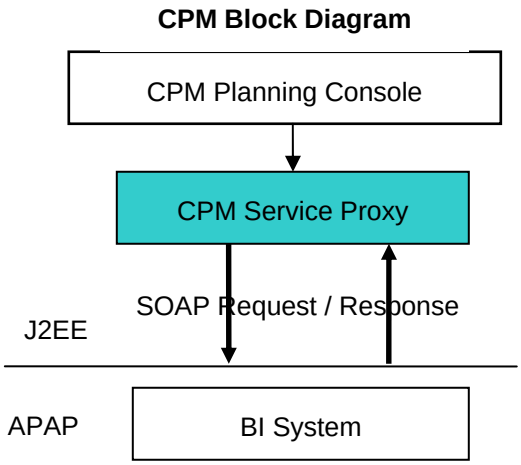
```
<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions targetNamespace="urn:sap-com:document:sap:rfc:functions" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="urn:sap-com:document:sap:rfc:functions"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <wsdl:types>
    - <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="urn:sap-com:document:sap:rfc:functions"
        targetNamespace="urn:sap-com:document:sap:rfc:functions" elementFormDefault="unqualified" attributeFormDefault="qualified">
      + <xsd:simpleType name="char1">
      + <xsd:simpleType name="char2">
      + <xsd:simpleType name="char20">
      + <xsd:simpleType name="char255">
      + <xsd:simpleType name="char30">
      + <xsd:simpleType name="char6">
      + <xsd:simpleType name="char60">
      + <xsd:simpleType name="char70">
      + <xsd:complexType name="CPM_TS_PLANNING_FUNCT_INFO">
      + <xsd:complexType name="CPM_TS_PLANNING_FUNCT_COND">
      + <xsd:complexType name="CPM_TS_SEL_CHARS">
      + <xsd:complexType name="CPM_TS_DIST_TYPE">
      + <xsd:complexType name="CPM_TS_FOX_FORMULA">
      + <xsd:complexType name="CPM_TS_PLANNING_FUNCT_KYFS">
      + <xsd:complexType name="CPM_TT_PLANNING_FUNCT_COND">
      + <xsd:complexType name="CPM_TT_SEL_CHARS">
      + <xsd:complexType name="CPM_TT_DIST_TYPE">
      + <xsd:complexType name="CPM_TT_FOX_FORMULA">
      + <xsd:complexType name="CPM_TT_PLANNING_FUNCT_KYFS">
      + <xsd:element name="CPM_PLANNING_FUNC_CREATE">
      + <xsd:element name="CPM_PLANNING_FUNC_CREATEResponse">
      </xsd:schema>
    </wsdl:types>
  + <wsdl:message name="CPM_PLANNING_FUNC_CREATE">
  + <wsdl:message name="CPM_PLANNING_FUNC_CREATEResponse">
  - <wsdl:portType name="cpm_planning_function_create">
    + <wsdl:operation name="CPM_PLANNING_FUNC_CREATE">
    </wsdl:portType>
  + <wsdl:binding name="cpm_planning_function_createSoapBinding" type="tns:cpm_planning_function_create">
  + <wsdl:service name="cpm_planning_function_createService">
  </wsdl:definitions>
```

Figure 5 – WSDL file representing the CPM application

Implementation of the Scenario

Overview of CPM Planning Console

CPM Planning Console uses Adobe Flex and communicates with SAP BI through Enterprise Services.



CPM Planning Console

The CPM Planning Console (Flex Application) follows Cairngorm framework.

Generated Template/Skeleton

```
public class cpm_planning_function_create_Port_MyApp extends com.sap.cpm.proxy.pfunction.cpm_planning_function_create_Port {
```

Business Delegate

```
public class CreatePlanningFunctionOp extends cpm_planning_function_create_Port {
```

Details of the Scenario

The sequence diagram for the CPM Planning Console is described in the following Figure 6.

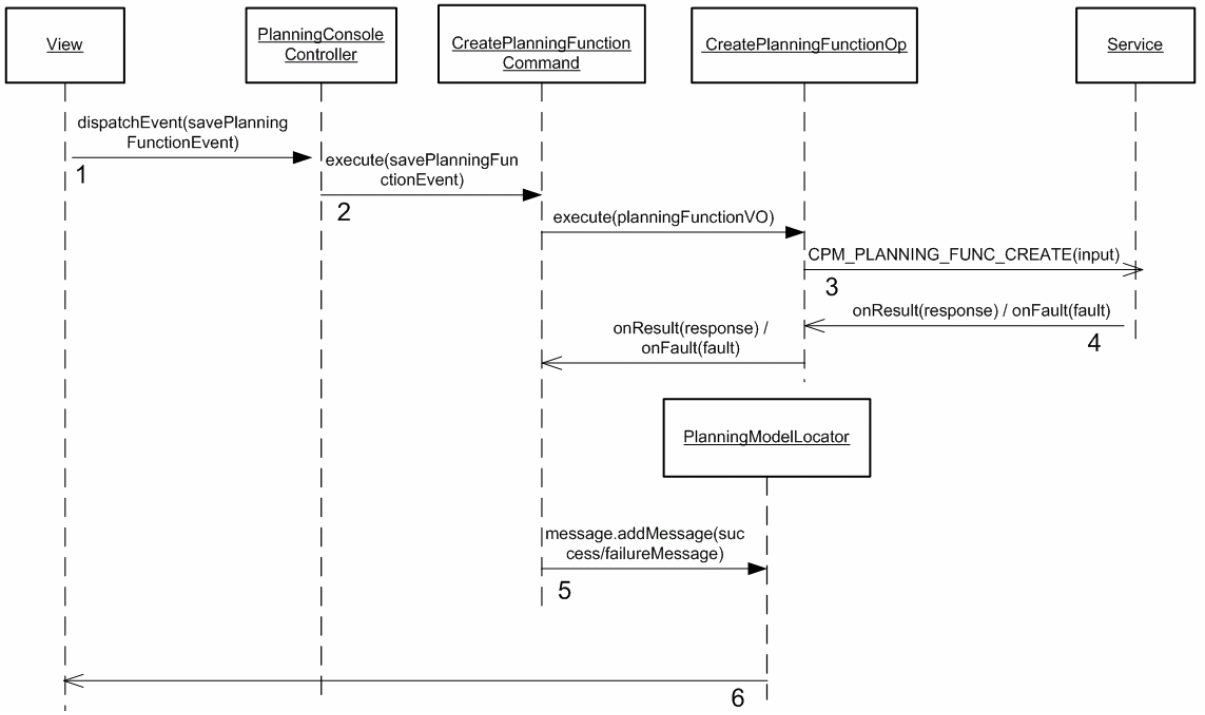


Figure 6 – Sequence Diagram describing the internals of the CPM planning application

1. When the user clicks on "Finish" a Create Planning Function Event is dispatched with Planning Function Value Object in the playload.

2. The Planning Console Controller listens for all registered incoming events. When the Create Planning Function Event is dispatched it determines the corresponding command class (Create Planning Function Command in our scenario) and calls the execute method on it. The command delegates the call to a business delegate (Create Planning Function Op) which extends the *_Port Class (generated by the Action Script Proxy Generator).

3. Create Planning Function Op gets the corresponding service from the Service Class and binds the operation with the Service Call. It then calls the required operation on the service by transforming Planning Function Value Object to the corresponding types that the Service understands.

4. All service calls are asynchronous and on successful execution the onResult function is called and on failure the onFault function is called. The Create Planning Function Op on receipt of the result just delegates the calls to the Create Planning Function Command.

5. OnResult the Command Class updates members on the Model. In this example it updates the message collection in the Model Locator

6. The view is updated with the changes in the model.

- Auto update by making use of Bindable (Flex Inbuilt feature)
- Using the Observe pattern. The view is an observer to a variable in the model. When the variable changes it lets the view know and the view can handle it by calling a handler function

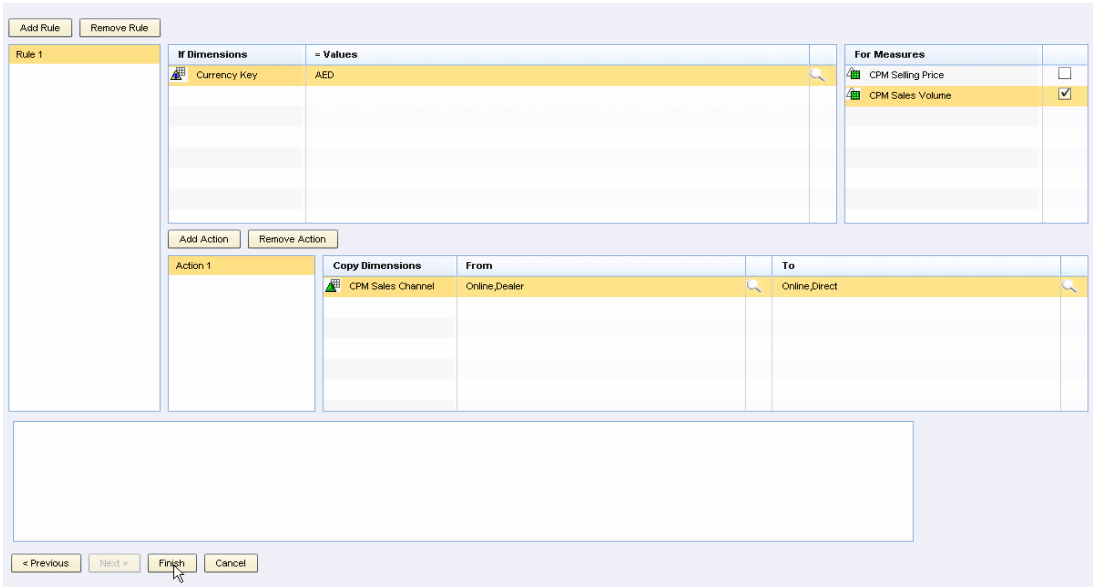


Figure 7 – Running the application – screen of the CPM planning console

Implementation of the Scenario

1. Generate a Web service proxy for CPM_PLANNING_FUNC_CREATE in the CPM Service Proxy library project using the Action Script Proxy Generator.

2. Create a Business Delegate Class:
   i. Extend the cpm_planning_function_create_Port (generated by the Action Script Proxy Generator)
   ii. In the execute() method of this class, request the required service from the Service Locator (Service) and invoke methods on it.
   iii. Delegate the result and fault calls to the responder's onResult and onFault (Typically the Command class). Since the call is asynchronous, the class that implements the Responder Interface (Command class) directly updates the model.

```
public class CreatePlanningFunctionOp extends cpm_planning_function_create_Port {

    private var responder:Responder;

    public function CreatePlanningFunctionOp(responder:Responder) {
        this.responder = responder;
    }

    public function execute(planningFunctionVO:PlanningFunctionVO):void {
        Service.instance().planningFunctionCreateService().getCpm_planning_function_createSoapBinding(this);
        CursorManager.setBusyCursor();
        CPM_PLANNING_FUNC_CREATE(input(planningFunctionVO));
    }

    override public function CPM_PLANNING_FUNC_CREATE_Result(parameters : com.sap.cpm.proxy.pfunction.CPM_PLANNING_FUNC_CREATEResponse) : void {
        responder.onResult(parameters);
        CursorManager.removeBusyCursor();
    }

    override public function CPM_PLANNING_FUNC_CREATE_Fault(fault : com.sap.flex.ws.runtime.WebServiceFault) : void {
        super.CPM_PLANNING_FUNC_CREATE_Fault(fault);
        CursorManager.removeBusyCursor();
    }
```

3. Create a command class.
   i. Implement com.adobe.cairngorm.commands.Command and com.adobe.cairngorm.business.Responder interfaces.
   ii. Implement the function execute() of the Command interface: If there is any information in the event, then cast the CairngormEvent parameter to the CustomEvent to get the required information and delegate the call to the business delegate method.
   iii. Implement onResult(..) and onFault(..) functions of the Responder interface for call back.

```
public class CreatePlanningFunctionCommand implements Command, Responder {

    public function execute(event:CairngormEvent):void {
        new CreatePlanningFunctionOp(this).execute(CreatePlanningFunctionEvent(event).planningFunctionVO);
    }

    public function onResult(event : * = null):void {
        if(CPM_PLANNING_FUNC_CREATEResponse(event).EV_SUBRC == 0)
            PlanningModelLocator.getInstance().messages.addMessage("Planning Function Created");
        else
            PlanningModelLocator.getInstance().messages.addMessage(CPM_PLANNING_FUNC_CREATEResponse(event).EV_MSG);
    }

    public function onFault(event : * = null):void {

    }
}
```

4. Create an Event class
   i. Extend com.adobe.cairngorm.control.CairngormEvent class.
   ii. Define the attributes of the Event class to hold the information (if any) to be passed with the event.

```
public class CreatePlanningFunctionEvent extends CairngormEvent {

    private var _planningFunctionVO:PlanningFunctionVO;

    public function CreatePlanningFunctionEvent(planningFunctionVO:PlanningFunctionVO) {
        super(PlanningFunctionSubController.EVENT_CREATE_PLANNING_FUNCTION);
        _planningFunctionVO = planningFunctionVO;
    }

    public function get planningFunctionVO():PlanningFunctionVO {
        return _planningFunctionVO;
    }
}
```

5. Register the event and the corresponding command in the Planning Console Controller.
6. Dispatch the CustomEvent with its corresponding parameters (if any) when the user clicks on finish.

```
private function savePlanningFunction():void {
    CairngormEventDispatcher.getInstance().dispatchEvent(new CreatePlanningFunctionEvent(PlanningModelLocator.getInstance().planningFunctionVO));
    areaFooter.btnFinish.enabled = false;
}
```

Solution in Detail

Procedural Programming

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)`the callback will be invoked instead.

For each Web service port defined in the WSDL file, the proxy generator creates a proxy class containing three ActionScript methods per each operation of the port: `operation, operation_Result` and `operation_Fault.` The developer should create a callback handler class by:

- sub-classing the 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.

Here is an example of application class invoking a Web service with a method named "Echo" (Echo is the name of a variable in a generated super class which is for brevity not shown below):

```
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
```

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

The first section inside doTest() instantiates an 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 a 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.

User Interface Generator

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 test the generated Flex application it can directly be run without having written a line of code yet.

Let's have a look inside Application.mxml. 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);
```

[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).

```
}
```

The second argument to *switchToScreen* specifies the data structure the screen is to render. This data structure normally is either an input or output WSDL message of Web service operation. When *null* is passed as the second argument to *switchToScreen* it means that the screen controls are to be initialized with blank values.

Plug-in currently uses the following types of controls:

Supported 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 the currently selected row.

TextInput control:

By editing the MXML file the developer can change the control type to Text, TextArea or RichTextEditor, while retaining the same control ID and data binding. The 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 the 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 Controls to Data Fields

The binding of screen controls to data fields is innervated by a handler of UIBinding type which is instantiated inside the MXML file. Bindings are declared inside method *bindScreen()* of the generated MXML file. An instance of the UIBinding class is named *uib* and has the following methods:

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

This method binds data to a control, the optional “controlID “controlID” is used for nested structures.

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

This method binds columns of the DataGrid control to data elements.

```
uib.addPseudoColumn()
```

This method is used for mapping an array of structures to the screen where every field in the structure is complex itself. The proxy generator will create a one-column DataGrid and request it to be populated with dummy records (“Record 1” … “Record N”), one per array

entry, via *addPseudoColumn()* binding.

Future versions of the proxy generator may be enhanced to pick simple subfields nested deeply within complex fields automatically, thus, reducing the need for manual design.

`uib.setButtons(addRowID, deleteRowID)`

This method is used for the declaration of buttons which add or remove records from DataGrid.

`uib.setDisplayName(dataPath, displayName)`

This method is used for displaying user-readable names for given data elements. 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")*.

Processing of Screens and User Input

The 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 the screen receives the control. The actual data structure to be displayed by the screen is passed as *args* argument. The 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 the input side of the operations will have an "Execute" button in their auto-generated MXML files. The "Click" handler of this button binds 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 the actual Web service call. Data is gathered from screen via *uib.fromScreen()*. Note that the 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 the data model embedded in the WSDL file. It will indicate to the user any incorrect data and suggest them to correct it. Validation is implemented only partially at this point. If the validation fails, *uib.fromScreen()* will return *null*.

For overriding the service endpoint address or for providing authentication data the set methods can be uncommented.

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

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

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

If the service invocation was unsuccessful, the *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 an 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.

Summary And Outlook

Adobe Flex is a mature and an easy-to-use technology with which you can create engaging graphical user interfaces. Using Adobe Flex in conjunction with Enterprise applications has already been tried out successfully several times. By enabling Adobe Flex to render SAP Enterprise Services we take this one step further. By putting these two technologies more closely together developer efficiency and quality and end-user friendliness of business applications increase dramatically.

Feature	Flex Out of the Box	SAP NetWeaver Solution
Web service method calls	✓	✓
Action Script proxy generation		✓
Support for simple WS structures		✓
Support for complex WS structures		✓
Flex UI generation based on WS definition		✓
SAP NetWeaver Eclipse based IDE integration		✓
Javadoc-like generated documentation for the generated proxy classes		✓

Related Content

Please include at least three references to SDN documents or web pages.

- Download WSDL to ActionScript Proxy Generator for Adobe FlexBuilder
- How to Develop Flex Applications that Invoke Web Services
- Blog on Developing Flex Applications that Invoke SAP Services

Copyright