

Paper 015-2010

## Flex Your SAS® Muscle

Joe Flynn, SAS Institute Inc., Cary, NC

### ABSTRACT

This paper explains how to integrate the power of SAS® software with the rich and interactive environment provided by Adobe Flex. SAS 9.2 enables you to easily deploy SAS® Stored Processes as open-standard Web services, which can then be called by Adobe Flex applications. This integration enables you to use the proven analytics and power provided by SAS as a viable back-end data source for your Adobe Flex applications.

This discussion, intended for applications developers and SAS programmers, first guides you through the much-improved process of deploying a SAS stored process as a Web service in SAS 9.2. The paper then explains how to call this Web service from Flex, how to interpret the results returned by SAS in Flex, and the different methods in which you can display those results.

The paper also includes an example that shows the code that you need in order to use SAS as a back-end data source to your Flex applications.

### INTRODUCTION

*Rich Internet Applications* (RIA) are Web applications that have the usability of desktop software and give users a more interactive and enriching experience. They are deployable on all major browsers and operating systems, which removes the hassle of performing a client installation. Most systems that run an RIA have Adobe Flash Player installed, which is a prerequisite to running a Flex application. This paper focuses on using Adobe Flex and integrating your SAS data to feed the Flex application. With the enhancements to the SAS 9.2 Intelligence Platform, you can now generate stored processes into Web services easily. After the Web service is generated, Adobe Flex can then call Web services to retrieve your SAS data.

The first section of this paper shows you how to create the stored process using SAS® Enterprise Guide®. Following that section, the discussion focuses on the improved process of deploying your SAS stored process as a Web service. You will learn how to call this Web service from Flex and interpret the results that are returned by SAS in Flex. In addition, you will learn about the different methods that you can use to display those results. The objective is to create an interactive Web application in Flex, which enables you to drill into data. For this discussion, the sample data set is SASHELP.ORSALES, Red Hat JBoss 1.4.2 is the middle tier, and all Flex programming is done with Adobe Flex Builder 3 Professional.

### CREATING THE STORED PROCESS

Before you can use a Web service to call SAS code, first you must create the stored process. The easiest way to create and manage the stored process is within SAS Enterprise Guide 4.2. To do this:

1. Invoke SAS Enterprise Guide.
2. Select **File ► New ► Stored Process** to open the Create New SAS Stored Process Wizard.
3. On the Name and Description page of the wizard, fill out the following essential fields ([Display 1](#)):
  - a. In the **Name** field, enter the name of the stored process. For this example, the stored process is called **orSales**.
  - b. In the **Location** field, choose a location for the SAS folders that will store the metadata object for this stored process. In this example, the location is **/GlobalForum**.
  - c. In the **Keywords (comma separated)** field, type the keywords **XMLA Web Service**, which ensures that this stored process is treated as an XMLA Web service and is not a generated Web service when it is deployed.

1 of 6 Name and Description

Save stored process as:

Name:  
orSales

Location:  
/GlobalForum   
(Example: /BIP Tree/My Folder Name)

Description:

Keywords (comma separated):  
XMLA Web Service

**Display 1. Entering the Name, Location, and Keywords Information**

4. Click **Next** to go to the SAS Code page.
5. Enter your SAS code in the text box as shown in Display 2.

2 of 5 SAS Code

```

1 libname WEBOUT xml;
2 %macro sData;
3
4 proc sql;
5
6 /* If initial execution, create overview
7 of sales based on Product Category */
8
9 %if "&category" = "initial" %then %do;
10 create table WEBOUT.orsales as
11 select Product_Category as Product_Category, sum(profit) as Profit
12 sashelp.orsales group by Product_Category;
13 %end;
14
15 /* If not initial execution, subset data
16 based on Selected Product_Category,
17 grouping now by Product_Group */
18

```

Stored process macros  
 Global macro variables  
 LIBNAME references

**Display 2. Entering SAS Code**

This step requires some planning, because you need to decide what data is streamed back to your Flex application. The data that is returned by this stored process must be in XML, which means that you must stream back your output to the `_WEBOUT` fileref. This example uses the following code:

```
libname _webout xml;
%macro sData;

proc sql;

    /* If this is the initial execution, create an overview
       of sales based on the product category. */

    %if "&category" = "initial" %then %do;
        create table _webout.orsales as
            select Product_Category as Product_Category, sum(profit) as Profit
            from sashelp.orsales group by Product_Category;
    %end;

    /* If this is not the initial execution, subset the data based on
       the selected Product_Category, now grouping by Product_Group. */

    %else %do;
        create table _webout.orsales as
            select product_group as Product_Group, sum(profit) as Profit
            from sashelp.orsales where Product_Category = "&category"
            group by product_group;
    %end
quit;
%mend;

%sData;
```

This code uses macro logic to determine whether the Flex application is requesting the high-level overview of the data (grouped by **Product\_Category**), or a lower-level view of the data, subset by **Product\_Category** and grouped by **Product\_Group**. This is determined by the **category** parameter, which will be set up later.

6. Click the **Include Code for** button and deselect **Stored process macros** ([Display 2](#)) to disable the stored process macros `%STPBEGIN` and `%STPEND`.

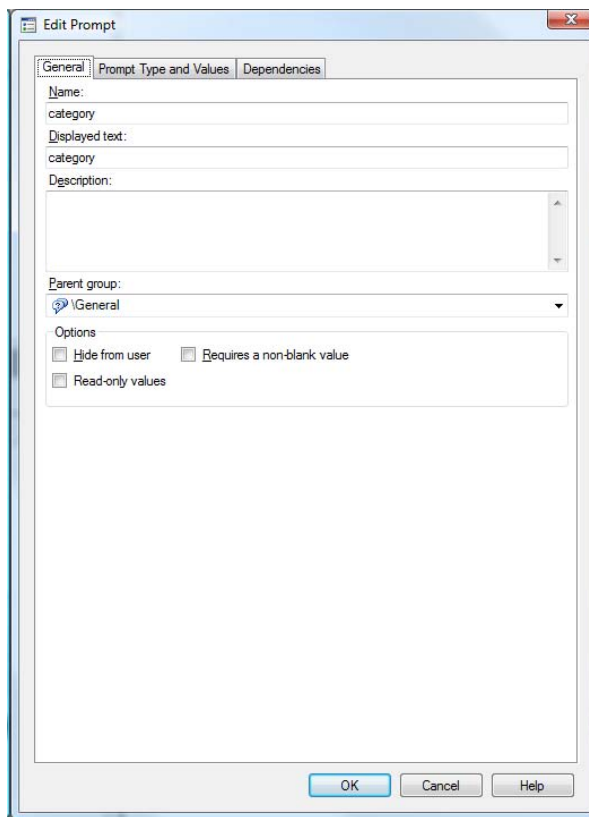
The macros set up ODS statements for the stored process, which are not required when streaming back XML.

7. Click **Next** to go to the Execution Options page.

In this step, you must ensure that the logical stored-process server is set as the execution server. This is required because the XMLA Web service must produce streaming output.

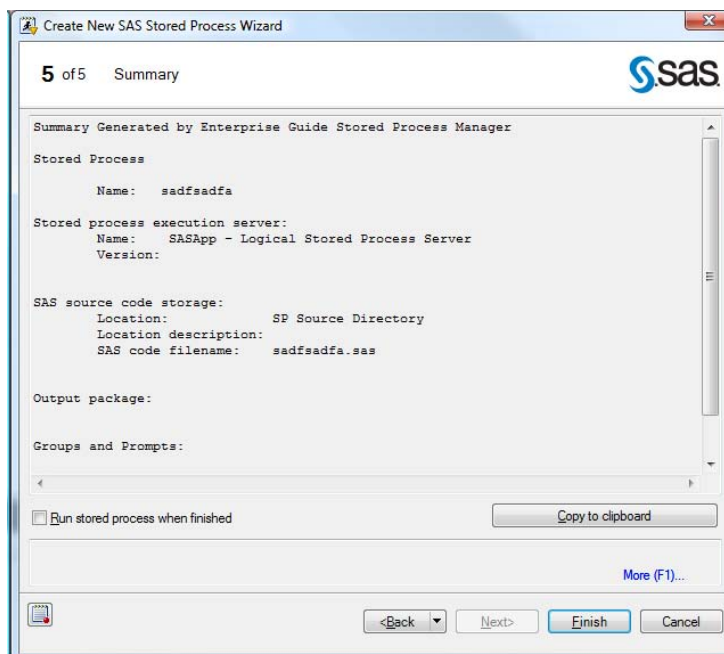
8. In the **Output Options** box, select **Streaming** ([Display 3](#)).





**Display 5. Accepting the Defaults on the Edit Prompt Page**

12. Click **Next** to go to the Summary page.
13. De-select **Run stored process when finished** and click **Finish** (Display 6).



**Display 6. De-selecting Run stored process when finished**

Your stored process is now created and ready to be deployed as a Web service.

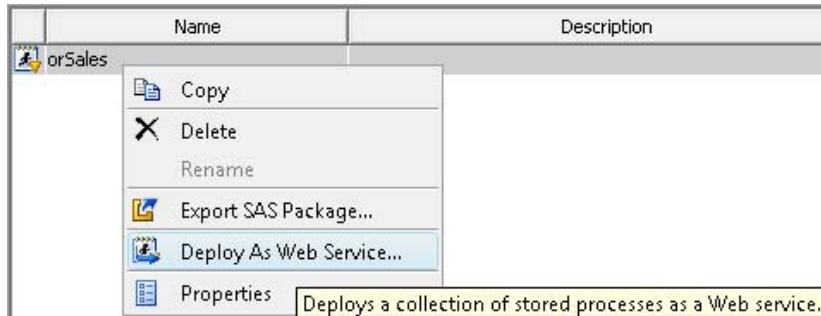
## DEPLOYING THE WEB SERVICE

Deploying the stored process as a Web service enables you to call the stored process directly from Adobe Flex. To deploy your stored process as a Web service:

1. Open SAS® Management Console and navigate to the stored process, **orSales**, that you created in the previous section.

For this example, the stored process is located in **/GlobalForum**.

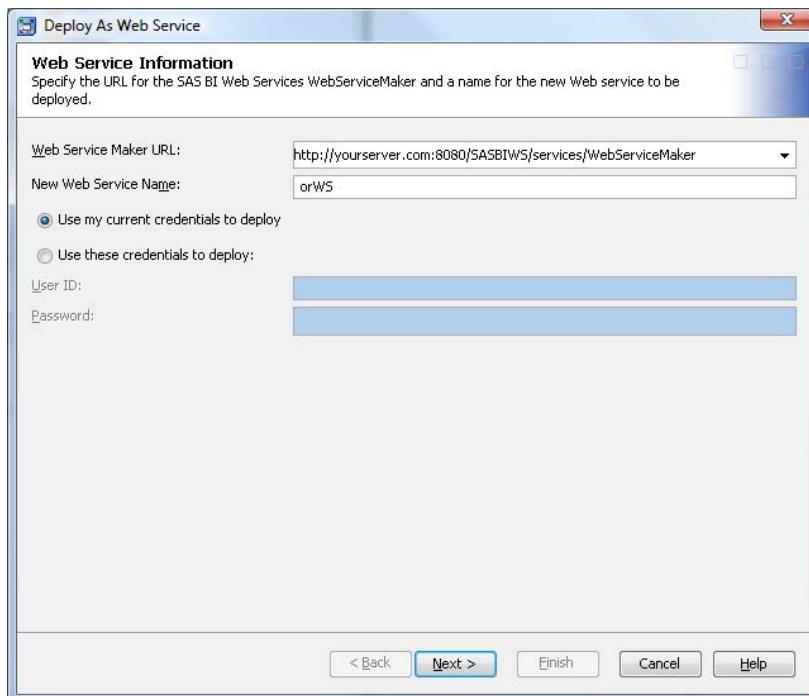
2. Right-click the stored process and click **Deploy as Web Service** (Display 7).



**Display 7. Selecting Deploy As Web Service in the Submenu**

When you deploy a Web service in this manner, you can select more than one stored process. Each stored process that you select corresponds to an operation in the Web service. There can be multiple operations inside the Web service. However, each name must be unique.

3. In the **Web Service Maker URL** field on the Web Service Information page, type the URL that will be the endpoint to the Web Service Maker if it differs from the default endpoint that is shown in Display 8.



**Display 8. Typing the URL for the Endpoint to Web Service Maker**

The *Web Service Maker* is a Web service that is created during the initial installation of the SAS Intelligence Platform. Its purpose is to make Web services.

4. In the **New Web Service Name** field, type the name of your Web service, which is **orWS** in this example (Display 8).  
On this page you can also specify credentials under which to deploy this Web service. Ensure that the user is part of the SAS BI Web Services Users metadata group.
5. Click **Next** to go to the next page.
6. In the **Namespace** field on the Web Services Keywords and Namespace page (Display 9), type the namespace if it differs from the default namespace **http://tempuri.org**.

**Web Service Keywords and Namespace**  
Enter the namespace and keywords associated with this web service.

Namespace:

Keywords:

#### Display 9. Typing the Namespace in the Namespace Field

The default namespace is acceptable for Web services that are under development. If you have a published service, then enter a unique and permanent namespace. This unique namespace is required so that client applications can distinguish it from other Web services. Although namespaces might be in the form of a URL, they do not need to be addresses for actual resources on the Web.

7. In the **Keywords** field, you can enter keywords, which can be used to locate this Web service later. (This example does not use keywords.)
8. Click **Next** to go to the last page, which provides an overview of all of the details that you entered (Display 10).

**Deploy As Web Service**

**Confirm Web Service Deployment**  
Confirm the following parameters which will be passed to the WebServiceMaker to deploy a new Web service. This operation may take several minutes and cannot be cancelled once started.

Web Service Maker URL:

New Web Service Name:

Namespace:

Stored Processes:

/GlobalForum/orSales
----------------------

Keywords:

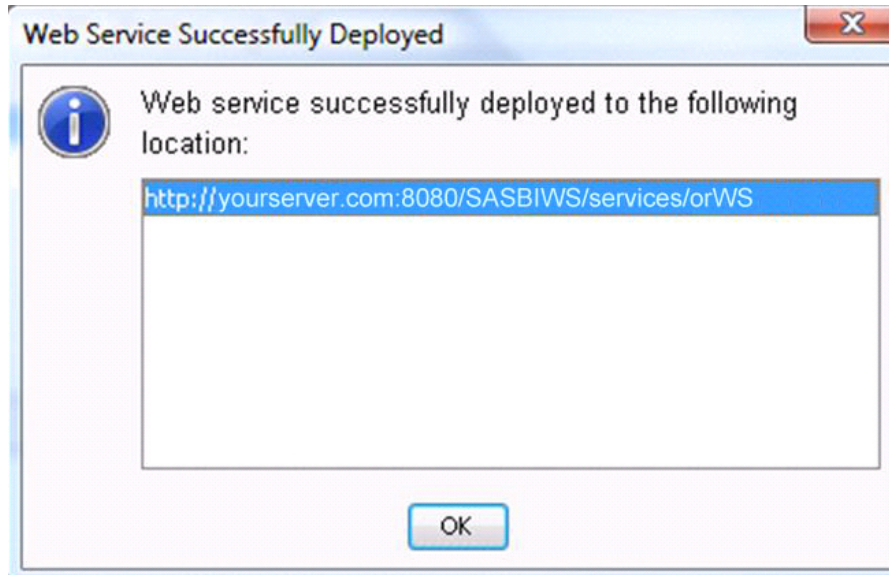
#### Display 10. Overview of the Web Service Details



The details include the Web Service Maker URL, the Web service name, the namespace, a list of the stored processes that are being deployed in this Web service, and any keywords that you entered.

9. Click **Finish**.

If everything works properly, the Web Service Successfully Deployed dialog box displays, which contains the endpoint for your deployed Web service (Display 11).



**Display 11. Dialog Box Indicating That the Web Service Successfully Deployed**

## CALLING AND USING THE WEB SERVICE

Figure 12 shows a simple, high-level overview of the Web service process.



**Figure 12. Overview of the Web Service Process**

The client application communicates directly with the middle-tier Java code (JBoss, in this case), making a standardized call to the Web service. The middle tier then makes an Integrated Object Model (IOM) call to the SAS Application Server and executes the stored process. The results are streamed back to the middle tier and are finally returned to the client application.

Eviware SoapUI is a free and open-source Web service testing tool that enables you to test that the Web service is functioning properly. This tool also enables you to examine the output of your Web service. The client application, which for the purposes of this paper is either SoapUI or Adobe Flex, initially obtains the Web Service Description Language (WSDL) from the Web service. The WSDL is an XML document, which provides all of the information that is necessary to call the Web service. This includes everything that a client application needs to know about this Web service, such as the operations that are available in the Web service, the endpoint, and the format of the XML document that the Web service is expecting. The XML document that is passed into a Web service is transported by a data envelope called a SOAP envelope, which is discussed later.



First, examine the WSDL, as follows:

1. Go to <http://yourserver.com:8080/SASBIWS/> to find the list of Web services that are available.
2. Click the link for the Web service that you created previously (**orWS** for this example) to display the WSDL for this Web service.

The WSDL contains the name of the Web service (**orWS**) and the stored process that you deployed (**orSales**), which is now considered an operation. It also contains **orSalesParameters** and **orSalesResult**, which define the parameters for this operation and the format of the results. The most important information for this example is the URL that is the address for the WSDL.

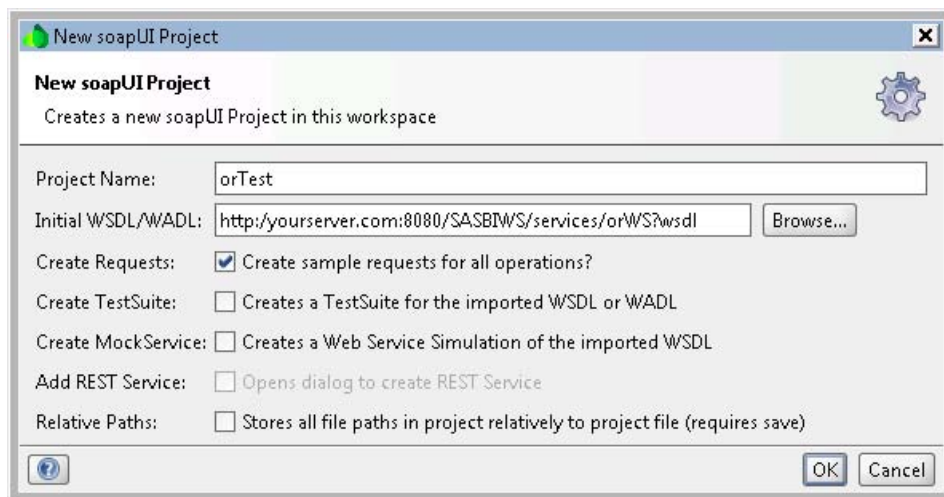
3. Copy the URL that is displayed in the browser to the clipboard, as shown in Display 13.



**Display 13. The Web Browser Showing the URL for the WSDL**

Now you can call this Web service from SoapUI.

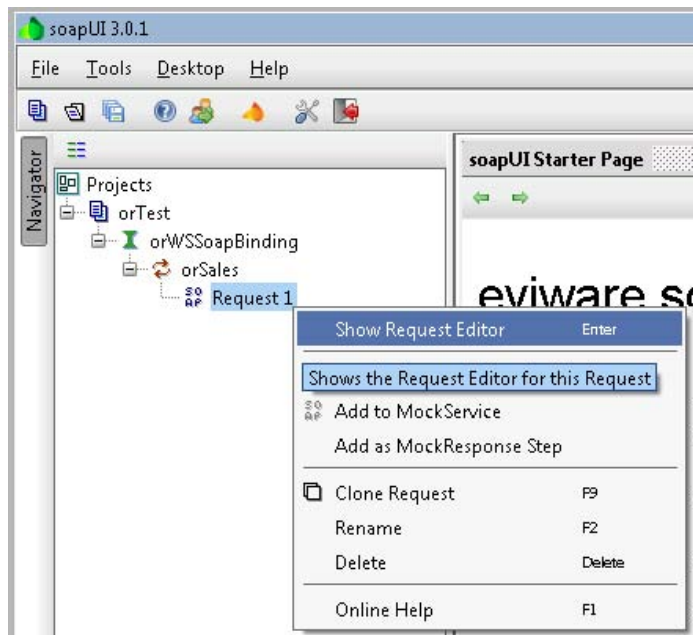
1. Open SoapUI and select **File ► New SoapUI Project**.
2. On the New soapUI Project page, in the **Project Name** text box, type a name for the project (**orTest** for this example) (Display 14).



**Display 14. Typing a Project Name**

3. In the **Initial WSDL/WADL** text box, add the WSDL location that you previously copied from the browser (Display 14).
4. Click **OK**.

In the soapUI window, the **orSales** operation is in the left-hand navigation ([Display 15](#)).



Display 15. The soapUI 3.0.1 Window Showing the orSales Operation


5. Expand **orSales** to see the **Request 1** node. Right-click the **Request 1** node.
6. In the submenu, select **Show Request Editor**.

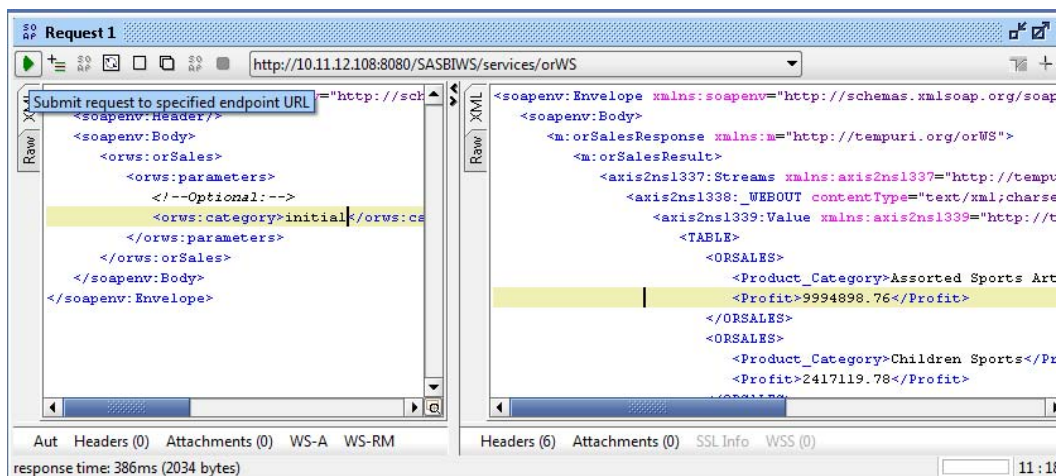
The new Request 1 page appears with a sample SOAP envelope. This was derived from the **orWS** WSDL. In the XML, notice that there is a placeholder for your **category** parameter. This stored process contains the following %IF statement:

```
%if "&category" = "initial" %then %do;
```

7. For testing purposes, pass in the value **initial** for the **category** parameter by changing the XML to the following:

```
<orws:category>initial</orws:category>
```

8. Click the green arrow button  to call your **orWS** Web service for the first time (Display 16).



Display 16. Calling the orWS Web Service

If everything completes successfully, the XML is returned in the pane to the right. The desired result is shown in [Display 16](#).

To test your drill-down capability, you can pass in a product category to ensure that you get back a subset of the products in that category.

- For this example, change the **category** parameter in your SOAP envelope as follows:

```
<orws:category>Children Sports</orws:category>
```

In reviewing each response XML, notice that there are the same six lines in both requests. Also notice the bold names below. You need to use these values in Flex to insert your results into an ArrayCollection.

```
<m:orSalesResult>
  <axis2ns1379:Streams xmlns:axis2ns1379="http://tempuri.org/orWS">
    <axis2ns1380:_WEBOUT contentType="text/xml;charset=windows-1252"
      xmlns:axis2ns1380="http://tempuri.org/orWS">
      <axis2ns1381:Value xmlns:axis2ns1381="http://tempuri.org/orWS">
        <TABLE>
          <ORSALES>
```

After successfully testing the Web services, you are now ready to integrate your SAS data into a Flex application.

## INTEGRATING SAS DATA INTO A FLEX APPLICATION

Flex consists of two languages: MXML and ActionScript. *MXML* is an XML-based language, which incorporates many built-in functions and is generally used to lay out the user interface. When compiled, each MXML tag is generated into ActionScript. *ActionScript* is a powerful object-oriented programming language. This section of the paper provides a brief overview of the important parts of the code that call the previously created Web service. The complete Flex code is included in the [Appendix](#), which provides additional comments about the process. A basic knowledge of Flex programming is required for understanding the content of this section.

To call your Web service, use the **<mx:WebService>** tag. This tag requires the WSDL address (<http://yourserver.com:8080/SASBIWS/services/orWS?wsdl>), the operation name (**orSales**), and any parameters (**category**) that you want to pass into the Web service. Here is an example of how your Web service call should be set up in Flex:

```
<mx:WebService id="webService"
  wsdl="http://yourserver.com:8080/SASBIWS/services/orWS?wsdl"
  showBusyCursor="true">
  <mx:operation name="orSales"
    resultFormat="object"
    result="getResult(event);"
    fault="getFault(event);">
    <mx:request>
      <parameters>
        <category> {parm.toString()}</category>
      </parameters>
    </mx:request>
  </mx:operation>
</mx:WebService>
```

The only fields that are unfamiliar at this point should be in the **<mx: operation name=>** tag, specifically **resultFormat**, **result**, and **fault**. The **resultFormat** function lets the compiler know that the return type of the Web service will be an object. The **result** function executes when the operation is completed, and the **fault** function executes if any problems are encountered. Also, you are using ActionScript to pass the value of your **category** parameter, which enables you to set this dynamically based on user input.

Here is the XML response from your Web service. As you recall, this was detailed in the previous section using SoapUI (the structure of the XML is particularly important, because you only want the **orSales** data in your array).

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <m:orSalesResponse xmlns:m="http://tempuri.org/">
      <m:orSalesResult>
        <axis2ns168:Streams xmlns:axis2ns168="http://tempuri.org/Streams">
          <axis2ns169:_WEBOUT contentTyp="text/xml">
            <axis2ns170:Value xmlns:axis2ns170="http://tempuri.org/Value">
              <TABLE>
                <ORSALES>
                  <Product_Category>A</Product_Category>
                  <Profit>9994898.76</Profit>
                </ORSALES>
                <ORSALES>
                  <Product_Category>B</Product_Category>
                  <Profit>2417119.78</Profit>
                </ORSALES>
              </TABLE>
            </axis2ns170:Value>
          </axis2ns169:_WEBOUT>
        </axis2ns168:Streams>
      </m:orSalesResult>
    </m:orSalesResponse>
  </soapenv:Body>
</soapenv:Envelope>

```

The GetResult function retrieves the results that are generated by the Web service and places them into an ArrayCollection:

```

private function getResult(evt:ResultEvent):void {
    orArray = new ArrayCollection(ArrayUtil.toArray
        (evt.result.Streams._WEBOUT.Value.TABLE.ORSALES.source));
    if (orArray.length == 0)
        orArray = new ArrayCollection(ArrayUtil.toArray
            (evt.result.Streams._WEBOUT.Value.TABLE.ORSALES));
    if (flag == 1)
        sc("Product_Category");
    else
        sc("Product_Group");
}

```

Notice that the object `evt.result.Streams._WEBOUT.Value.TABLE.ORSALES.source` corresponds to the XML that you viewed from SoapUI. You are using an ArrayCollection instead of a plain array in this case. Because you will eventually use this data to populate a data grid, using an ArrayCollection enables you to easily sort the data. Also notice that a %IF statement evaluates a flag variable. This is used to determine whether the Web service is returning the **Product\_Group** or **Product\_Category** level of data and is discussed in more detail in the full code in the [Appendix](#). This is set in alternating executions. Depending on which results are being returned, you need to reset the X axis and the data-grid fields. This is the purpose of the `sc` function.

To call the Web service as the application loads, use `initialize="firstLoad()"` in the `<mx:Application>` tag. Then create a FirstLoad function as shown below which calls your Web service:

```

private function firstLoad():void {
    // Set parameter to initial value
    parm = "initial";
    //Set flag to 1
    flag = 1
    // Execute Web service.
    webService.orSales.send();
}

```

Notice that the call is in the format of `webServiceid.operation.send()`. Upon execution of this application, you should have data returned by your Web service in your ArrayCollection. The data is now ready and can be easily displayed in many different ways.

The following code creates a data grid. You can use the ArrayCollection called `orArray`, which contains the results of your Web service, as the data provider. Because you are initially passing in the `category` parameter with the value of `initial`, the default columns to display are `Product_Category` and `Profit`.

```

<mx:DataGrid id="myGrid" width="426" height="338"
             dataProvider="{orArray}" x="591" y="196">
  <mx:columns>
    <mx:DataGridColumn id="prodDF" dataField="Product_Category" />
    <mx:DataGridColumn dataField="Profit" />
  </mx:columns>
</mx:DataGrid>

```

When you run the project in Flex, the call is made to the Web service and the response is displayed in a data grid much like the one that is shown in Display 17.



Product_Category	Profit
Assorted Sports Articles	9994898.76
Children Sports	2417119.78
Clothes	9208375.41
Golf	3711822.11
Indoor Sports	1481330.76
Outdoors	13400513.2
Racket Sports	2016834.77
Running - Jogging	2300666.19
Shoes	8889545.98
Swim Sports	727868.6
Team Sports	1007238.97
Winter Sports	3928833.99

**Display 17. The DataGrid Display Showing Data for Product\_Category and Profit**

The complete Flex code is included in the [Appendix](#). This code also produces an interactive a bar chart, line plot, and pie chart of this data in which you can click either a column of the bar chart or a slice of the pie chart to drill down into your data. [Display 18](#) shows the final report.



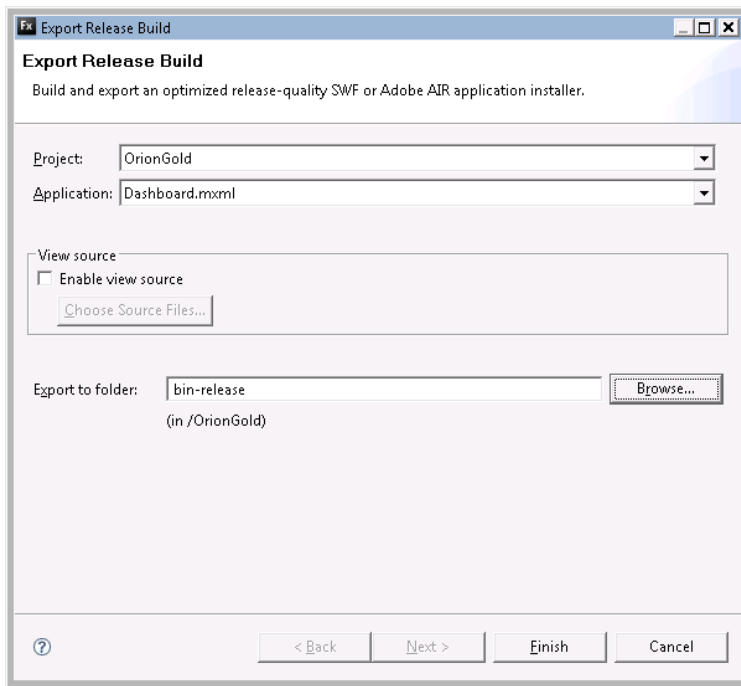
Display18. The Final Report with a Data Grid, Bar Chart, Pie Chart, and Line Plot

### DEPLOYING TO JBOSS

Once the Flex application is working as expected, you can then deploy the report for use on your JBoss application server.

1. In your Flex application, select **File ► Export ► Release Build**.

Make a note of the folder name to which your export displays in the **Export to folder** text box, as shown in [Display 19](#).

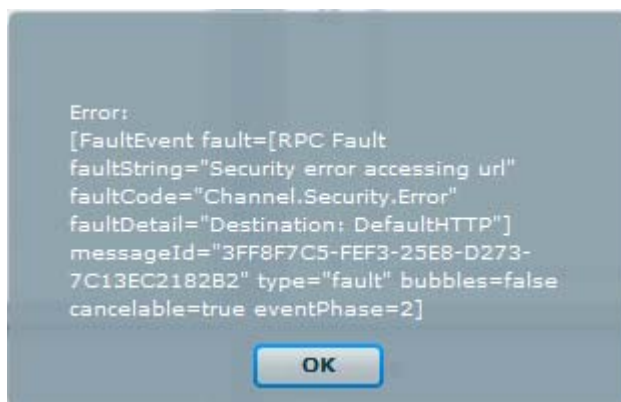


**Display 19. The Export Release Build Window Showing the Folder Name of the Export**

2. Click **Finish**.
3. Use the operating system to browse to the export folder.
4. Copy the files that are in export folder to the `jboss-4.2.0.GA\server\SASServer1\deploy\jboss-web.deployer\ROOT.war\orion` folder of your JBoss installation. (Note that you need to create the **Orion** folder.)

The Flex application should now be available from the URL  
<http://yourserver.com:8080/orion/Dashboard.html>.

You might encounter the following exception (Display 20):



**Display 20. Error Message**

This error is due to the security sandbox restrictions of Flash Player. By default, Flash applications can load data only from the same domain. To automatically give an application access to data on another domain you must use a cross-domain policy file. This file is named `crossdomain.xml` and must be deployed on the root of the Web server



that the SWF file is calling. This error might occur if the Web service is being called using the IP address, rather than the host name. Below is an example of a cross-domain policy file, which would allow an SWF to access the available resources on the Web server where this is located:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
"http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="all"/>
  <allow-http-request-headers-from domain="*" headers="*" secure="false" />
  <allow-access-from domain="*" secure="false" />
</cross-domain-policy>
```

Note that before you deploy any cross-domain policy file like this, it is very important to understand all implications of its presence. Refer to the Adobe resource "Cross-domain policy file specification" ([www.adobe.com/devnet/articles/crossdomain\\_policy\\_file\\_spec.html](http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html)) for more information about cross-domain policy files.

## CONCLUSION

Using SAS BI Web Service is an easy and convenient way to stream your SAS data into Flex applications. This enables you to easily mix the power and analytics of SAS software with the rich and visually stimulating environment that is provided by Flex.

## APPENDIX

### The Flex Code

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
initialize="firstLoad()" width="1098" height="1027">
  <mx:Script>
    <![CDATA[

import mx.collections.ArrayCollection;
import mx.charts.HitData;
import mx.charts.events.ChartItemEvent;
import mx.controls.Alert;
import mx.rpc.events.ResultEvent;
import mx.rpc.events.FaultEvent;
import mx.utils.ArrayUtil;

[Bindable] private var orArray:ArrayCollection;
[Bindable] private var parm:String = "first";
private var flag:int = 0;

private function getResult(evt:ResultEvent):void {
  orArray = new ArrayCollection(ArrayUtil.toArray
    (evt.result.Streams._WEBOUT.Value.TABLE.ORSALES.source));
  if (orArray.length == 0)
    orArray = new ArrayCollection(ArrayUtil.toArray
    (evt.result.Streams._WEBOUT.Value.TABLE.ORSALES));
  // Set the X-Axis and datafield properly.
  if (flag == 1)
    sc("Product_Category");
  else
    sc("Product_Group");
}

private function getFault(evt:FaultEvent):void {
  Alert.show("Error:\n" + evt.toString());
}

]]>
```

```

private function firstLoad():void {
    // Set the parameter to initial value.
    parm = "initial";
    //Set the flag to 1.
    flag = 1;
    // Execute the Web service.
    webService.orSales.send();
}

private function sc(xF:String):void{
    // Change the X axis for the bar chart.
    cSeries1.xField = xF;
    cAxis1.categoryField = xF;

    // Change the column for the datafield.
    prodDF.dataField = xF;

    // Change the pie name for the pie chart.
    ps1.nameField= xF ;

    // Change the X axis for the line plot.
    lSeries1.xField = xF;
    lAxis1.categoryField = xF;
}

private function drillData(e:ChartItemEvent):void {
    // If the flag is 1, you are drilling into the data.
    if (flag == 1) {
        //Set the flag to 0.
        flag = 0;
        // Set the parameter to the value of the item that
        // is clicked.
        parm = e.hitData.item.Product_Category.toString();
        // Execute the Web service.
        webService.orSales.send();
    } else {
        firstLoad();
    }
}

]]>
</mx:Script>

<!--Control the animations for the charts.-->

<mx:SeriesSlide id="slideIn" duration="1000" direction="up"/>
<mx:SeriesSlide id="slideOut" duration="1000" direction="down"/>
<mx:SeriesInterpolate id="pieIn" duration="1000" />

<!--Set up the Web service.-->

<mx:WebService id="webService"
    wsdl="http://vista64-
15938.na.sas.com:8080/SASBIWS/services/orWS?wsdl" showBusyCursor="true">
    <mx:operation name="orSales"
        resultFormat="object"
        result="getResult(event);"
        fault="getFault(event);">
        <mx:request>
            <parameters>
                <category>{parm.toString()}</category>
        </mx:request>
    </mx:operation>
</mx:WebService>

```

```

                </parameters>
            </mx:request>
        </mx:operation>
    </mx:WebService>

<!--Set up the data grid, using orArray as the data provider.
Columns are dynamically set using the sc function in ActionScript.-->

    <mx:Panel title="DataGrid" layout="horizontal" y="79"
horizontalCenter="-291" width="448" height="378">
    <mx:DataGrid id="myGrid" width="426" height="338"
    dataProvider="{orArray}" x="591" y="196">
        <mx:columns>
            <mx:DataGridColumn id="prodDF" dataField="Product_Category" />
            <mx:DataGridColumn dataField="Profit" />
        </mx:columns>
    </mx:DataGrid>
</mx:Panel>

<!--Set up the bar chart, using orArray as the data provider.
The X axis is dynamically set using the sc function in ActionScript.-->

    <mx:Panel title="Bar Chart" layout="horizontal" y="79"
horizontalCenter="175">
        <mx:ColumnChart id="column" height="338" width="432"
itemClick="drillData(event)" paddingLeft="5" paddingRight="5"
showDataTips="true" dataProvider="{orArray}" x="67" y="196">
            <mx:series>
                <mx:ColumnSeries id="cSeries1"
xField="Product_Category" yField="Profit" hideDataEffect="slideOut"
showDataEffect="slideIn"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis id="cAxis1"
categoryField="Product_Category"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
    </mx:Panel>

<!--Set up the pie chart, using orArray as the data provider.
The X axis is dynamically set using the sc function in ActionScript.-->

    <mx:Panel title="Pie Chart" layout="horizontal" y="465"
horizontalCenter="-294" width="454" height="328">
    <mx:PieChart id="pie" itemClick="drillData(event)"
paddingLeft="5" paddingRight="5" showDataTips="true" dataProvider="{orArray}"
x="67" y="196" width="430" height="276">
        <mx:series>
            <mx:PieSeries id="ps1"
            field="Profit"
            nameField="Product_Category"
            labelPosition="callout" hideEffect="pieIn"
showDataEffect="pieIn"
            />
        </mx:series>
    </mx:PieChart>
</mx:Panel>

<!--Set up the Line Plot, using orArray as the dataprovider.
The X axis is dynamically set using the sc function in ActionScript.-->

    <mx:Panel title="Line Plot" layout="horizontal" y="465"
horizontalCenter="174" width="454" height="328">

```

```
        <mx:LineChart x="792" y="586" id="linechart1" width="422"
height="286" dataProvider="{orArray}">
            <mx:series>
                <mx:LineSeries id="lSeries1" displayName="Profit"
xField="Prouct_Group" yField="Profit" hideEffect="pieIn"
showDataEffect="pieIn"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis id="lAxis1"
categoryField="Product_Category"/>
            </mx:horizontalAxis>
        </mx:LineChart>
    </mx:Panel>

</mx:Application>
```

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Joe Flynn  
SAS Institute Inc.  
Cary, NC 27513  
Work Phone:  
E-mail: [support@sas.com](mailto:support@sas.com)  
Web: [support.sas.com](http://support.sas.com)

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.