THE POWER TO KNOW.

SAS® 9.3 BI Web Services
Developer's Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc 2011. *SAS® 9.3 BI Web Services: Developer's Guide*. Cary, NC: SAS Institute Inc.

SAS® 9.3 BI Web Services: Developer's Guide

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st electronic book, July 2011

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/publishing** or call 1-800-727-3228.

Contents

What's New in SAS 9.3 BI Web Services

Overview

SAS 9.3 BI Web Services introduce several new features for programmers that make it easier to consume SAS Stored Processes using popular Web service protocols. New features include support for new transport types, integration with SAS 9.3 Stored Process features, more management capabilities, an engine rewrite for speedier execution and more comprehensive extensions, and a feature that eliminates the need to use the Deploy as Web Service wizard in SAS Management Console to create new generated Web services by exposing stored processes for dynamic execution. In addition, SAS BI Web Services for .NET has been discontinued in SAS 9.3.

General Enhancements

The following general enhancements have been added to SAS BI Web Services:

- The SAS BI Web Services for Java engine has been rewritten to use the Spring Framework. This new engine is backwards compatible with SAS 9.2 generated Web services and the XMLA Web service. You can continue to use any existing client proxy code when invoking migrated SAS 9.2 generated Web services and XMLA proxies should continue to work as they did in SAS 9.2.
- You no longer need to generate Web services using the Deploy as a Web Service wizard in SAS Management Console. As soon as you create a SAS Stored Process, it is available for execution by SAS BI Web Services. You can continue to generate Web services to group multiple stored processes under one endpoint or to publish the intent that these stored processes are to be executed by Web service clients.
- SAS BI Web Services for .NET has been discontinued. SAS BI Web Services for Java will support migrated .NET 9.2 generated Web services in a way that is transparent for clients. In fact, clients should need only to change endpoint addresses (and this step can be omitted if a proxy server is used). For more information, see "Migrating SAS BI Web Services for .NET to SAS BI Web Services for Java" on page 8.

SAS Stored Process Enhancements

The following stored process enhancements have been added to SAS BI Web Services:

- The SAS Workspace Server supports stored processes with output parameters and stored processes with streaming output, except stored processes that use sessions.
- Data tables can be specified as data sources and data targets. Data tables are similar to traditional data sources and targets, but they eliminate the need for stored process authors to hardcode LIBNAME statements in SAS code. Also, data tables enable stored process authors to specify a template table. This template table is used to automatically generate schema for the table in SAS BI Web Service WSDLs.

Transport Type Additions

SAS BI Web Services has always provided SOAP endpoints for XMLA and generated Web services. SOAP is widely used in enterprise scenarios because of the set of WS-* standards available for the protocol, for its use of a Web Service Description Language (WSDL) files, and for its structured and namespaced messages. However, sometimes SOAP is overkill. Many mobile client development libraries lack native SOAP libraries and Web applications typically use client-side asynchronous JavaScript remoting calls where SOAP is not appropriate. Therefore, SAS 9.3 BI Web Services supports plain XML and JSON as transport types. For more information, see Chapter 4, “Using JSON and Plain XML with RESTful Web Services,” on page 49.

Chapter 1

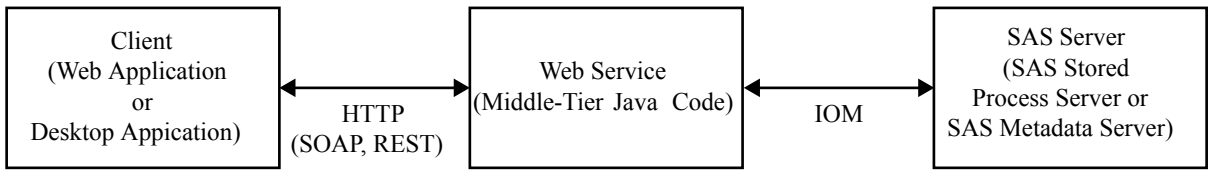
Overview of SAS BI Web Services

What Are SAS BI Web Services?

A Web service is an interface that enables communication between distributed applications. Web services enable cross-platform integration by enabling applications that are written in various programming languages to communicate by using a standard Web-based protocol. This functionality makes it possible for businesses to bridge the gaps between different applications and systems.

The following figure shows how Web services work.

Figure 1.1 *Web Services Communications*



In general, SAS BI Web Services expose SAS Stored Processes for execution by using Web service protocols. Remote clients are then able to specify input parameters, drive execution of SAS code, and obtain results from that execution. Also, Web services make it possible to write clients that perform this act in a myriad of languages and on a variety

of operating systems by using HTTP to exchange messages. Web services can enable a service-oriented, enterprise application approach, or they can support the development of mobile or Web clients, all of which leverage reusable SAS Stored Processes.

There are two core types of SAS BI Web Services: XMLA and structured Web services. Structured Web services can further be divided based on how you access the services and the format of the messages that you send and receive.

SAS BI Web Services expose a single XMLA Web service endpoint with two operations: Discover and Execute. Clients call Discover in order to obtain information from the system, including the list of SAS Stored Processes that are available for execution by the XMLA engine, the inputs and outputs of those stored processes, and other metadata about the stored processes. Clients can then use the Execute operation to specify input, execute a stored process, and obtain the results of that execution. The XMLA Web service is more limited than structured Web services because it provides only a general interface for invoking stored processes. For example, XMLA includes a Web Service Description Language (WSDL) file, but because XMLA can be used to execute any number of stored processes, this WSDL does not actually describe the stored process inputs and outputs. Instead, the WSDL describes the information that the Discover calls will return to the client. This makes XMLA Web services unsuited for use with client Web service libraries where automatic proxy generation and easy service execution are desired. Also, XMLA only supports XML and the SOAP protocol for all operations and does not support many features of SAS Stored Processes.

In SAS 9.3, all stored processes are available individually for execution using Web services without any action required from the user. SAS BI Web Services automatically exposes a WSDL file for each and every stored process that is available in the system. These WSDL files use XML to include detailed information about the inputs and outputs of each stored process using XML schema descriptions. Also, the WSDL file includes the URLs of endpoints to use to invoke these stored processes by using SOAP over HTTP. You can use these WSDL files to automatically generate code in your client framework of choice that can be used to invoke the Web services. SAS BI Web Services exposes these services using a simple mapping from the metadata folder path of the stored process.

You can group multiple stored processes together in a single, named Web service using the Deploy As Web Service wizard in SAS Management Console. In 9.2, these were called generated Web services because the wizard generated a grouping (and because server artifacts were actually generated as well). You can group stored processes for Web service execution to simplify management or to enable generated client code to invoke more than one stored process. The grouping generates a single WSDL file that describes all stored processes together in one document and all stored processes in a generated Web service grouping are invoked at the same, unique endpoint based on the name of the generated service. Stored processes that are a member of a generated Web service are still individually available for execution.

All structured Web services can be invoked by using SOAP over HTTP. SOAP strictly defines message structure, including the envelope containing headers and body. SAS BI Web Services define the content (and namespace) of the payload within the body. In addition, SAS 9.3 BI Web Services support Javascript Simple Object Notation (JSON) and plain XML as message formats for all structured Web services. The format of input XML messages for a structured Web service can be deduced from its WSDL file. The addition of new output resource URL suffixes in conjunction with the new SAS folder path mapping means that SAS BI Web Services now support Representational State (REST) style Web service invocation.

SAS BI Web Services for Java are administered by using JBoss, Oracle WebLogic, or IBM WebSphere. If you install SAS BI Web Services for Java, then you also need to have a Java Virtual Machine for an application server. For more information about

administering and configuring Web services, see the *SAS Intelligence Platform: Web Application Administration Guide*.

Creating SAS BI Web Services

Prerequisites

Before you can use Web services, you need to perform the following steps:

1. Install SAS Integration Technologies, which includes SAS BI Web Services and the SAS Metadata Server.

   Note: When you install SAS 9.3 BI Web Services, you are actually installing several Web services: the XMLA Web service, the WebServiceMaker Web service that is used to create generated Web services, and a Web service that is responsible for handling non-generated, JSON, and REST invocations.

2. Write a SAS program to use as a stored process with Web services. See "Writing SAS Programs for XMLA Web Services" on page 9. See "Writing SAS Programs for Structured Web Services" on page 26.

3. Define a stored process server or workspace server, if one is not already defined.

4. Define a stored process by using SAS Management Console.

Creating XMLA Web Services

If you want to invoke a stored process using the XMLA Web service, you must use **`XMLA Web Service`** as a keyword and **`Stream`** as the stored process output type when defining the stored process. On the client side, perform the following steps to use XMLA Web services:

1. Locate the Web Service Description Language (WSDL) file. You can access the WSDL for a SAS BI Web Service by appending **`'?WSDL'`** onto the service endpoint.

2. Write the code for the client application that uses either the Discover method or the Execute method to call the Web service.

3. Run the code.

For XMLA Web services, the SAS code that implements the Web service, the metadata, and the client code that calls the Web service must all be synchronized. The following table shows how to synchronize these items:

Table 1.1 *Items to Synchronize*

Item	SAS Program	Metadata	Client Code
Name	The name of the file that contains the SAS code.	Associates a SAS Stored Process with the name of the file containing the SAS program. Alternatively, in SAS 9.3, the SAS program can be included directly in metadata.	`<StoredProcess name= 'MyStoredProcess'>`
Input Data	Reads XML from the fileref. `libname instream xml;`	The name of the fileref, which must match the name of the data source. In this case, the name of the fileref is instream.	`<Stream name='instream' XML input stream provided by client... </Stream>`
Input Parameters	Macros. `&tablename`	The parameter name is specified in the metadata. Parameters are validated for proper typing, but they are treated as strings on the server regardless of the type that is specified in the metadata.	`<Parameter name='tablename'> value</Parameter>`
Output Data	Writes output to the _WEBOUT fileref as XML. `libname _WEBOUT xml xmlmeta= &_XMLSCHEMA;`	Designates the output as 'Streaming'.	Uses the XML that is returned.

Creating Generated Web Services

Follow these steps to use generated Web services:

1. Generate a new Web service:
   a. In SAS Management Console, select a set of stored processes and then select **Actions** ⇨ **Deploy As Web Service** to generate a new Web service that can be used to call the selected stored processes.
   b. In the **Web Service Maker URL** field of the Deploy as Web Service wizard, type the endpoint URL or select an existing URL. The user who performs this action should belong to the SAS BI Web Services Users metadata group so that the new Web service can be stored in metadata. The new Web service will contain one operation for each stored process that you selected.
   c. Upon successful deployment, a message displays that tells you the endpoint URL for the newly deployed Web service.

TIP You can use your operating system's keyboard shortcut to copy the URL (for example, on Windows, press control + C).

2. Create clients to call the Web service. Many Web service programming frameworks have utilities to generate client code that can invoke your SAS Web services. Typically, these frameworks will use a Web Service Description Language (WSDL) file to generate these client files. You can access the WSDL of a SAS Web service by appending `?WSDL` or `.wsdl` to the URL of your SAS Web service.

A Web service can be created with multiple operations in it. Each operation corresponds to a stored process, and has the same name as the stored process, unless there is a naming conflict. If the name of the stored process conflicts with another name, then a new operation name is created.

Accessing the Web Service Endpoint for a Stored Process

Any SAS Stored Process that exists in your SAS Metadata Server can be invoked using SAS BI Web Services. This is a convenient alternative to generating Web services when you need to invoke a single stored process. The endpoint URLs for stored processes are not stored anywhere. You can compute the endpoint URL easily by using the type of transport desired and the path of the stored process. See "Accessing SOAP Endpoints for Stored Processes and Generated Web Services" on page 28 for more information about accessing SOAP endpoints for stored processes. See "Accessing RESTful JSON and XML Web Service Endpoints" on page 54 for more information about accessing REST endpoints for stored processes.

Differences between Web Service Types

Differences between XMLA and Structured Web Services

The major differences between XMLA Web services and structured Web services are:

- *Consumption capabilities.* Structured Web services have a WSDL that is customized for each stored process that is in the service. This enables client application developers to create proxies that can create and read the XML documents that are exchanged with the service. XMLA services are described in the Discover call, so proxies must be manually created by the developer for calling the service.
- *Attachments.* XMLA Web services can process XML only. Structured Web Services can read and write binary information by using attachments when using the SOAP protocol and endpoints. For example, this means you can return graphs that are generated by ODS by using structured Web Services and SOAP.
- *Output parameters.* The only allowed output from XMLA is the _WEBOUT stream. Structured Web Services can return output parameters, the _WEBOUT stream, packages, and data targets.
- *Deployment.* To enable a stored process for XMLA execution, you must add the `XMLA Web Service` keyword to the stored process definition in metadata. By comparison, structured Web service access is available for all stored processes automatically by using a RESTful URL mapped to the metadata location of the stored process. Also, you can create new structured Web services by grouping stored processes by using the Deploy as Web Service wizard in the Folder view of SAS Management Console.

Differences among XML, JSON, and SOAP Invocations

Structured Web services can be invoked using XML, JSON, and SOAP messages. Certain features and functionality are available only when using a particular message format. Here are the main differences between the three message formats:

- *Input and output types.* XML and SOAP messages support all stored process input and output types including prompts, XML data sources and targets, generic data sources and targets, data tables, output parameters, streams, and packages. JSON messages only support simple prompt types (ones that can be represented with a string) and output parameters.
- *Endpoint addresses.* The plain XML, JSON, and SOAP versions of a structured Web service are available at three different endpoint URLs.
- *Description files.* Only SOAP Web services expose a WSDL file that strictly defines the inputs and outputs and the endpoints for the Web service. You can use this description file to create plain XML message requests for use with the XML endpoint. JSON services don't have a file that describes their input messages, but an input message can be formed by specifying prompt name/value pairs in JSON.
- *Message format.* SOAP and plain XML services both use XML to convey invocation and result information. However, namespaces might be omitted from plain XML invocations. JSON services use the Javascript Simple Object Notation format for messages.

Overview of Security for Web Services

A default installation of SAS BI Web Services for Java is not highly secure. The default security mechanism is SAS authentication. All requests and responses are sent as clear-Text. If users want to authenticate as a specific user, then they can send a user name and password as clear-Text as part of the WS-Security headers for SOAP services or as HTTP basic authentication headers when using RESTful Web services (plain XML and JSON). Authentication is performed by authenticating client credentials at the SAS Metadata Server. Whenever user names and passwords must be sent as clear-Text, SSL should be enabled to provide transport layer security.

You can configure an anonymous user account to use for Web service invocations when credentials are not provided. The anonymous account is configured during software configuration using the SAS Deployment Wizard. Anonymous users cannot use the Web Service Maker; credentials must always be provided to use the Web Service Maker.

SAS BI Web Services can be secured by using Web authentication. This provides a way for SAS BI Web Services to identify the calling subject as authenticated by the underlying Java application server. This authentication mechanism requires HTTP transport-level security to be enabled.

Note: Web authentication can be used with both XMLA Web services and structured Web services but cannot be used with the Web Service Maker Web service when invoked by SAS Management Console clients because they use SAS one-time passwords.

Consult with your administrator to determine how Web services are configured at your site and how you can invoke them. For more information about setting up Web service security, see the *SAS Intelligence Platform: Web Application Administration Guide*.

Understanding Error Codes

Errors generally fall into one of five categories, and are assigned the appropriate error code for that category. The following table describes these error codes:

Table 1.2 *Error Codes*

Error Code	Description
1000	Specifies an invalid user name or password (the client application might want to re-prompt the user for credentials).
2000	Specifies a client error (the client application might want to pass in different parameters). This error might occur for one of the following reasons: • invalid prompt value • required parameter is missing • invalid request against schema • invalid stored process name (for XMLA Web services only) • no ReadMetadata permission for the stored process
3000	Specifies a SAS error. This error is generated when the stored process generates a SYSCC macro variable that is not listed in the AcceptableSysccList configuration option. An additional attribute is added to indicate the actual error number that SYSCC was set to. The SYSMSG string is also included in the message.
4000	Specifies a configuration error. This indicates a problem that the administrator of the service should be notified about. The administrator should be able to examine logs on the service to determine the cause of this error. This error might occur for one of the following reasons: • invalid default credentials for the anonymous user • invalid trusted credentials • metadata server or stored process server is unreachable • invalid configuration file
5000	Specifies a time-out error. This error occurs if the user configures SAS BI Web Services with a stored process time-out and the execution of a stored process exceeds this time-out.

Note: Before SAS 9.2, XMLA returned an error code of `99` for almost all errors.

The following code is an example of a generated SOAP fault that has an error code of `4000`:

```
<SOAP-ENV:Fault>
   <faultcode>Server</faultcode>
   <faultstring>The XML for Analysis provider encountered an error</faultstring>
   <faultactor>XML for Analysis Provider>XML for Analysis Provider</faultactor>
```

```
   <detail>
      <sas:Fault code="4000">
         <sas:Exception message="The configured credentials are invalid.">
            <sas:Exception message="The config file contains invalid metadata
      credentials."/>
            <sas:Exception message="The user 'anon' is unknown.">
               <sas:Exception message="'anon' is not defined in metadata."/>
            </sas:Exception>
         </sas:Exception>
      </sas:Fault>
   </detail>
</SOAP-ENV:Fault>
```

Migrating SAS BI Web Services for .NET to SAS BI Web Services for Java

SAS 9.3 BI Web Services no longer includes a .NET version. The benefit of Web services is that the technology of the server does not matter to the client. Because of differences in the underlying technologies between previous .NET and Java versions of SAS BI Web Services, the format for input and output messages differed significantly between the two products. This meant that clients created for one version were typically not compatible with the other version. To encourage interoperability and further innovation in SAS BI Web Services, SAS 9.3 does not include a .NET version.

SAS 9.3 BI Web Services for Java is fully backwards compatible with both the Java and .NET versions of SAS 9.2 BI Web Services. When you perform a migration using the SAS Migration Utility and the SAS Deployment Wizard from SAS 9.2 to SAS 9.3, any generated Web services are also migrated. If the generated Web service being migrated was created with SAS 9.2 BI Web Services for .NET, the migrated SAS 9.3 version is fully compatible with clients that were created for the SAS 9.2 .NET version. The migration process sets a flag on the generated Web service metadata that Web services can use when generating and displaying the WSDL and processing a Web service invocation. You need to modify only the endpoint of the generated Web service in the client code to point to the new endpoint in the migrated system. You can retrieve the new endpoint of the migrated service by using the SAS Configuration Manager or by locating the Web service in **/System/Services** in SAS Management Console.

If you regenerate a Web service that was migrated from a .NET installation, then the Web service is no longer compatible with clients that were previously created. You can force .NET backwards compatibility by passing the **dotnetMode=true** query parameter when retrieving WSDLs and invoking Web services. For example, to retrieve a .NET-backwards-compatible WSDL file, use a URL similar to **http://*host*:*port*/SASBIWS/services/*yourServiceName*?wsdl&dotnetMode=true**. To invoke the Web service, send requests to the endpoint **http://*host*:*port*/SASBIWS/services/*yourServiceName*?dotnetMode=true**.

Chapter 2

Writing SAS BI Web Services Using XMLA

Writing SAS Programs for XMLA Web Services

To use the Web service to call your SAS code, you must configure your SAS code as a stored process. A stored process is a SAS program that is stored on a server and can be executed by requesting applications. Any stored process can be deployed as a generated Web service. However, stored processes that are used with XMLA Web services need to conform to rules that enable the Web service to receive data from the client and return data to the client.

You can author a stored process manually by using SAS or a text editor to write the code and then registering the program through SAS Management Console. Alternatively, you can use a program such as SAS Enterprise Guide or another SAS code generator to author a stored process using the point-and-click method. Use the following modifications to make a stored process that can be used with SAS BI Web Services. Keep in mind that XMLA Web services can return XML data only; no attachments can be returned.

Note: XMLA Web services in SAS 9.3 will work only with SAS 9.3 stored process metadata and SAS 9.3 Stored Process Servers.

The following list explains unique details about stored processes that are used with XMLA Web services:

- The data that is returned by the stored process must be XML. Web service stored processes produce streaming results, which means that the SAS program writes output to _WEBOUT, typically by using the following LIBNAME statement:

  ```
  libname _WEBOUT xml xmlmeta=&_XMLSCHEMA;
  ```

- For XMLA Web services, the %STPBEGIN or %STPEND macros are not used in the stored processes. These macros set up Output Delivery System (ODS) statements for the stored process, but XMLA Web services do not use ODS.
- The _XMLSCHEMA macro is unique to XMLA Web services. This macro is passed to the SAS program when it is invoked from the Web service. The _XMLSCHEMA macro is set to one of three values depending on the Content property that gets passed to the Execute method. The possible values for _XMLSCHEMA are Schema, SchemaData (which is the default), Data, or None. For example, the following code causes SAS to write both the XML schema and the data into the libref _WEBOUT:

  ```
  libname _WEBOUT xml xmlmeta=SchemaData;
  ```

  A libref uses a fileref of the same name when a source is not specified in the LIBNAME statement. For example, the following code causes the libref, called _WEBOUT, to read from the fileref called _WEBOUT:

  ```
  libname _WEBOUT xml xmlmeta=_XMLSCHEMA;
  ```

  For XMLA Web services, SAS defines the filerefs for the _WEBOUT output stream as well as for any input streams before invoking the SAS code.

  Note: Applications should not try to write multiple data sets into a library when a schema is being created.

- Data sources are defined when you are registering the stored process metadata. There are three types of data sources:
  - Generic streams, which are most similar to the input streams that were used before SAS 9.2.
  - XML streams, which can be described with or without a schema. If a schema is provided for an XML stream, then that schema is inserted in the WSDL for the service. If no schema is provided, then **`xs:any`** is inserted in the WSDL. Having a schema defined makes it easier for client applications to call a service. The SAS code needs to be written to create XML that is valid according to the schema that is defined in the metadata.
  - Data tables, which are new for SAS 9.3 and describe tabular input and output. Data tables cannot be used with XMLA.

The following example code displays a stored process that is used as a Web service:

```
libname instream xml;
libname _WEBOUT xml xmlmeta=&_XMLSCHEMA;

proc means data=instream.&tablename
   output out=_WEBOUT.mean;
run;
```

The first LIBNAME statement in the sample code defines the data source. This code corresponds to the definition of the data source in the Stored Process Properties dialog box in SAS Management Console. The fileref of the data source is **`instream`**. In this example, the data source provides the data to run PROC MEANS against.

The second LIBNAME statement in the sample code defines the output for the stored process as streaming output, or _WEBOUT. In the Stored Process Properties dialog box, **`Stream`** is specified as the type of output on the **Execution** tab of the Stored Process Properties dialog box.

The **`&tablename`** declaration in the sample code defines a parameter called tablename. In the Stored Process Properties dialog box, this parameter is specified through the New Prompt dialog box, and can be modified using the Edit Prompt dialog box. In this example, tablename is a text parameter that specifies the name of the table to run PROC MEANS against.

Note: The dialog boxes mentioned in the previous example are available from both the Stored Process Properties dialog box and the New Stored Process wizard, which are both part of SAS Management Console. For more information about using SAS Management Console to define metadata for stored processes, see the product Help.

Discover Method

Overview of the Discover Method

The Discover method retrieves information, such as stored process metadata or a list of available data sources, from the SAS Metadata Repository. The Discover method returns a list of all the stored processes that have the keyword "XMLA Web Service" on the SAS Metadata Server. The SAS Stored Process Server is not invoked to service the Discover call.

Here is the syntax for the Discover method:

```
Discover (
   [in] RequestType As EnumString,
   [in] Restrictions As Restrictions,
   [in] Properties As Properties,
   [out] Result  As Rowset)
```

RequestType

Overview of RequestType

RequestType is a required parameter for the Discover method. The value of RequestType is an enumeration value that corresponds to a return rowset. The RequestType parameter specifies the type of information to be returned by the Discover request.

There are two main request types that are normally used with SAS BI Web Services: DISCOVER_DATASOURCES and STOREDPROCESS_PARAMETERS. DISCOVER_DATASOURCES and STOREDPROCESS_PARAMETERS both return a list of the stored processes that can be invoked. DISCOVER_DATASOURCES is a standard XMLA request type that returns a list of available data sources for the server or Web service so that you can select a data source with which to connect. The information that is returned by the DISCOVER_DATASOURCES request type includes the following information:

- the name and a description of the data source
- a URL to connect to the data source, the name, and data type of the provider

- the type of security mode that the data source uses, as well as any additional information that is needed to connect to the data source

STOREDPROCESS_PARAMETERS is a request type that is specific to SAS. This request type returns a list of all the available stored processes along with a list of the parameters that are specified in each stored process.

Other request types that might be useful with SAS BI Web Services are DISCOVER_PROPERTIES and DISCOVER_SCHEMA_ROWSETS. DISCOVER_SCHEMA_ROWSETS returns a list of all the available request types along with their enumeration values and other information. For more information about what the DISCOVER_PROPERTIES request type returns, see "Properties" on page 14.

Note: Although the SAS XMLA Stored Process provider supports the DISCOVER_KEYWORDS,DISCOVER_LITERALS, and DISCOVER_ENUMERATORS request types, these request types are not useful for calling stored processes.

DISCOVER_DATASOURCES

The SAS BI Web Service returns one data source for each stored process that has been defined in the metadata for use with Web services.

For each returned stored process, the returned rowset contains:

DataSourceName
specifies the name of the stored process, as specified in SAS Management Console. For example,

```
/Samples/Stored Processes/
   Sample: MEANS Procedure Web Service
```

DataSourceDescription
specifies the description of the stored process, as specified in SAS Management Console. For example,

```
(PROC MEANS Stored Process that can be invoked by
   the SAS BI Web Services for Java middle tier.)
```

URL
specifies the URL to invoke the XMLA methods. This is usually the same as the URL that is used to invoke this Discover method. For example,

```
http://host:port/SASBIWS/services/XMLA
```

DataSourceInfo
specifies which data source to use. The SAS Stored Process Server data source is "Provider=SASSPS;".

ProviderName
specifies the provider behind the data source. For the SAS Stored Process Server, this is the SAS XML for Analysis Stored Process Provider.

ProviderType
specifies the type of provider that is behind the data source. The Stored Process Service supports only Tabular Data Provider.

AuthenticationMode
specifies the authentication required for the given data source (that is, indicates whether a user name and password are required). SAS BI Web Services for Java always return "Authenticated," meaning that you are required to authenticate to the SAS Metadata Repository whether you pass in credentials or use only default credentials that are configured by the administrator.

STOREDPROCESS_PARAMETERS

STOREDPROCESS_PARAMETERS is a custom request type that is used by the SAS Stored Process Service provider only. It returns metadata describing the parameters that are necessary to call the stored process. A stream parameter is always a required parameter and it never has a default. This does not mean that you are required to have a stream parameter for each stored process, but it means that any stream parameters that are defined for the stored process must be provided when the stored process is called using the Execute method.

For each returned stored process, the returned rowset contains:

StoredProcessName
: specifies the name of the stored process.

Parameters
: specifies a container that includes all of the parameters for the stored process.

Parameter
: specifies a container that includes all of the details for a stored process parameter.

Name
: specifies the name of the stored process parameter.

Description
: specifies the description of the stored process parameter.

Type
: specifies the parameter type. The possible parameter types for XMLA Web services are string, multi-line text, Boolean, integer, float, color, time, timestamp, and date. (XMLA Web services do not support advanced prompt types such as data source, data source item, OLAP member, data library, ranges, and prompts with multiple value types.) Note that all parameters are passed to SAS as macro variables, so the SAS program does not know the parameter type that is specified in the metadata. For more information about how to format parameter values, see "Using Prompts with Generated Web Services" on page 31.

Required
: specifies whether the stored process parameter is required.

Default
: specifies a default value for the stored process parameter.

Streams
: specifies a container that includes all of the data sources for the stored process.

Stream
: specifies a container that includes all of the details for a stored process data source.

The following is an example of a STOREDPROCESS_PARAMETERS response for a stored process that takes a single string and a single stream as input:

```
<row xmlns="urn:schemas-sas-com:xml-analysis:rowset">
   <StoredProcessName>
      /BIP Tree/copyintoout</StoredProcessName>
   <Parameters>
      <Parameter>
         <Name>inputname</Name>
         <Description>A simple string that we are
            passing as a parameter.</Description>
         <Required>true</Required>
         <Default />
         <Type>String</Type>
```

```
            </Parameter>
         </Parameters>
         <Streams>
            <Stream>
               <Name>instream</Name>
               <Description>This stream does allow
                  multi-pass reads, so you do not have to
                  use an XMLMap.</Description>
            </Stream>
         </Streams>
      </row>
```

Restrictions

You can use the Restrictions parameter to filter which results get returned from a call to the Discover method. The restriction name specifies a column in a rowset that you restrict. The restriction value specifies which data to restrict in the column. Use the DISCOVER_SCHEMA_ROWSETS request type to get restriction information about the rowsets that are available in each request type. The DISCOVER_SCHEMA_ROWSETS request type returns a list of all the request types that are supported by the provider, along with restriction information and descriptions for each request type.

The Restrictions parameter is required in the Discover method, but it can be empty. Invalid values for restrictions are ignored.

The following RestrictionList element restricts a call to Discover STOREDPROCESS_PARAMETERS based on the name of the stored process:

```
<RestrictionList
   xmlns="urn:schemas-microsoft-com:xml-analysis">
   <StoredProcessName>
      /Samples/Stored Processes/
         Sample: MEANS Procedure Web Service
   </StoredProcessName>
</RestrictionList>
```

Properties

The Properties parameter enables you to specify properties of the Discover method, such as the return format of the result set or the time-out value.

Use the DISCOVER_PROPERTIES request type to get information about properties that are available for each request type and the values that are associated with those properties. The DISCOVER_PROPERTIES request type returns information about both standard and provider-specific properties. The returned information includes the name, description, data type, access, and current value of each property. The information also shows whether each property is required.

The following table contains a list of properties and property information, including sample values, that the DISCOVER_PROPERTIES request type returns. The value of PropertyType for each of these properties is **`string`**.

Table 2.1 *Values for the Properties Parameter*

PropertyName	PropertyDescription	PropertyAccessType	Value
Content	Specifies the content of the XML result: None, Schema, Data, or Both.	ReadWrite	SchemaData
UserName	Specifies the user name to use for metadata authentication.	ReadWrite	
Password	Specifies the password to use for metadata authentication.	Write	
Domain	Specifies the domain to use for metadata authentication.	ReadWrite	
ProviderName	Specifies the name of the XML for Analysis provider.	Read	SAS XML for Analysis StoredProcess Provider
ProviderVersion	Specifies the version of the XML for Analysis provider.	Read	1.0
Format	Specifies the format of the XML result: Tabular or Multidimensional.	Read	Tabular
DataSourceInfo	Specifies the identifying information that is required to retrieve data from a data source.	ReadWrite	Provider=SASSPS

You can list properties in any order. The Properties parameter is required in the Discover method. The only call to the Discover method that can have empty properties is DISCOVER_DATASOURCES. All other request types require at least DataSourceInfo to be specified, such as:

```
<PropertyList
   xmlns="urn:schemas-microsoft-com:xml-analysis">
   <DataSourceInfo>
      Provider=SASSPS
   </DataSourceInfo>
</PropertyList>
```

To cause a call to Discover to execute under a specific user's identity, a UserName and Password property can be included in the PropertyList element, such as:

```
<PropertyList xmlns="urn:schemas-microsoft-com:xml-analysis">
   <DataSourceInfo>
      Provider=SASSPS
   </DataSourceInfo>
```

```
   <UserName>username</UserName>
   <Password>password</Password>
</PropertyList>
```

If you choose to include the UserName or Password properties, it is important to ensure that access to your Web service is secure and encrypted. For more information, see the *SAS Intelligence Platform: Web Application Administration Guide*.

Result

The Result parameter is required. This parameter specifies the result set that the provider returns. The information that is returned can vary according to which values are used in the RequestType, Restrictions, and Properties parameters.

Execute Method

Overview of the Execute Method

Client applications of the Web service call the Execute method to run a SAS Stored Process.

When an application calls the Execute method, the Web service performs the following actions:

- receives the call and validates the SOAP request against the WSDL.
- validates the command against the command schema.
- searches in the SAS Metadata Server to find the SAS server to connect to that can service the request. If the user name and password are provided in the Properties parameter, then they are used to connect to the SAS Metadata Server. The credentials to use when connecting to the SAS application server are obtained from the metadata.
- invokes the SAS code that represents the stored process on the SAS application server.
- checks the value of the SYSCC macro in SAS. If the SYSCC macro has a nonzero value, then the Web service throws a SOAP fault and includes the value of SYSMSG in the fault.
- returns all data that was written to _WEBOUT.

Here is the syntax for the Execute method:

```
Execute (
   [in] Command As Command,
   [in] Properties As Properties,
   [out] Result As Resultset)
```

Command

The Execute method takes the Command and Properties parameters as input. Both of these parameters are in XML.

The following code shows the command passed to the Execute method:

```
<StoredProcess name="MyStoredProcess">
   <Stream name="instream">
      <TABLE>
         <Class>
            <Name>Alfred</Name>
            <Sex>M</Sex>
            <Age>14</Age>
            <Height>69</Height>
            <Weight>112.5</Weight>
         </Class>
         <Class>
            <Name>Alice</Name>
            <Sex>F</Sex>
            <Age>13</Age>
            <Height>56.5</Height>
            <Weight>84</Weight>
         </Class>
         ...
      </TABLE>
      </Stream>
   <Parameter name="inputname">myName</Parameter>
</StoredProcess>
```

When the previous code is passed to the Execute method, the SAS code has a macro defined whose name corresponds to the String parameter:

```
%LET inputname=myName
```

The SAS code also has a fileref assigned that corresponds to the name of the Stream parameter:

The SAS program should write output to the pre-assigned fileref _WEBOUT. Most applications do this by using the XML LIBNAME engine, as follows:

```
libname instream xml;
libname _WEBOUT xml xmlmeta=&_XMLSCHEMA;

proc copy in=instream out=_WEBOUT;
run;

libname instream clear;
libname _WEBOUT clear;
```

Properties

The Properties parameter enables you to specify properties of the Execute method. Properties describe how to invoke the Command parameter. Calling applications specify the SAS Stored Process Service Provider to be used in DataSourceInfo, as shown in the following example:

```
<PropertyList>
   <DataSourceInfo>
      Provider=SASSPS;
   </DataSourceInfo>
</PropertyList>
```

Use the DISCOVER_PROPERTIES request type in the Discover method to get information about properties that are available for each request type and the values that are associated with those properties. The DISCOVER_PROPERTIES request type

returns information about both standard and provider-specific properties. The returned information includes the name, description, data type, access, and current value of each property. The information also shows whether each property is required.

You can list properties in any order. The Properties parameter is required in the Discover method, but it can be empty. The Properties parameter must be specified for the Execute method, and must include at least the DataSourceInfo property.

Note: After you have selected a data source from the DISCOVER_DATASOURCES rowset, set the DataSourceInfo property in the Properties parameter, which is sent to the server using the Command parameter by the Execute method. Do not attempt to write your own value for the DataSourceInfo property. Use a value only from the DISCOVER_DATASOURCES rowset.

To cause the execute method to run under a specific user's identity, a UserName and Password property can be included in the PropertyList element, such as:

```
<PropertyList xmlns="urn:schemas-microsoft-com:xml-analysis">
   <DataSourceInfo>
      Provider=SASSPS
   </DataSourceInfo>
   <UserName>username</UserName>
   <Password>password</Password>
</PropertyList>
```

If you choose to include the UserName or Password properties, it is important to ensure that access to your Web service is secure and encrypted. For more information, see the *SAS Intelligence Platform: Web Application Administration Guide*.

Result

The Result parameter is required. This parameter specifies the result set that the provider returns. The information that is returned can vary according to which values are used in the Command and Properties parameters.

Sample PROC MEANS Using SAS BI Web Services

Sample Overview

This sample shows how to write, define, and invoke a sample stored process that can be used with SAS BI Web Services. This example is for an XMLA Web service. You can access other sample Web services in the samples database at **`support.sas.com`**.

Write the Stored Process

The following SAS code is a sample stored process called **`stpwsmea.sas`**. This program is installed with SAS Integration Technologies; by default it is located in **`<SASHOME>\SASFoundation\9.3\inttech\sample`**.

```
%put &tablename

libname _WEBOUT xml xmlmeta = &_XMLSCHEMA;
libname instream xml;
```

```
proc means data=instream.&tablename
   output out=_WEBOUT.mean;
run;

libname _WEBOUT clear;
libname instream clear;
```

Define the Metadata

The stored process must be defined on a SAS Metadata Server that is used by SAS BI Web Services in order to determine how and where to run the stored process. Stored process metadata is defined by using SAS Management Console. The tables in this section show the values for each field in the New Stored Process wizard in SAS Management Console.

Note: If you have previously installed the SAS Stored Process sample metadata as part of the SAS Deployment Wizard or the Web Infrastructure Platform installation, then you might not need to re-create the metadata for the "Sample: MEANS Procedure Web Service" sample stored process. The sample metadata should already be available from the **`/Products/SAS Intelligence Platform/Samples`** folder. If you do not have the sample metadata, you can define the metadata for the stored process on your SAS Metadata Server by performing the following steps.

1. Open SAS Management Console and connect to the appropriate metadata server.
2. From the SAS Management Console navigation tree, select the folder under which you would like to create the new stored process. (If you would like to create a new folder, you can select the location in the navigation tree in which you want to add the new folder, and then select **Actions** ⇨ **New** ⇨ **Folder** from the menu bar to open the New Folder wizard. Follow the wizard instructions to create the new folder.)
3. After you select the folder in which you want to add a new stored process, select **Actions** ⇨ **New** ⇨ **Stored Process** from the menu bar. The New Stored Process wizard displays.
4. On the first page of the New Stored Process wizard, enter the following values in their corresponding fields for the sample Web service:

Table 2.2 *Field Values for the New Stored Process Wizard*

Field	Value
Name	Sample: MEANS Procedure Web Service
Keywords	XMLA Web Service

Note: To add the keyword, click **Add** to open the Add Keyword dialog box, and then enter the name of the keyword. Click **OK**. Adding a description and roles for the stored process are optional.

5. Click **Next**.
6. Enter the following values in their corresponding fields for the sample Web service:

Table 2.3 *Values for the Sample Web Service*

Field	Value
Application server	SASApp
Server type	Stored process server only
Source code location and execution	Allow execution on selected application server only Store source code on application server
Source code repository	<SASHOME>\SASFoundation\9.3\inttech \sample
Source code file	stpwsmea.sas
Results	Stream

Click **Next**.

7. Click **New Prompt** to add an input parameter to the stored process.
8. On the **General** tab, enter the following values in their corresponding fields for the sample Web service:

Table 2.4 *Values for the Prompt*

Field	Value
Name	tablename
Displayed text	tablename

9. Select the **Requires a non-blank value** check box. Entering a description is optional.
10. On the **Prompt Type and Values** tab, enter the following values in their corresponding fields for the sample Web service:

Table 2.5 *Values for the Prompt*

Field	Value
Prompt type	Text
Method for populating prompt	User-entered value
Number of values	Single value
Text type	Single line
Default value	InData

11. Click **Next**.
12. Click **New** to open the New Data Source dialog box, where you must define the data source.
    a. Enter the following values in their corresponding fields for the sample Web service:

    Table 2.6 *Values for the New Data Source*

Field	Value
Type	XML Data Source
Label	instream
Fileref	instream
Expected content type	text/xml

    b. You must also select the **Allow rewinding stream** check box in the New Data Source dialog box. Otherwise, an XMLMap would need to be specified on the XML LIBNAME statement to define the XML schema for **`instream`**.
    c. Click **OK** to save the data source definition.
13. Review your stored process information, and click **Finish** to define the metadata for the stored process.

Invoke the Stored Process

SOAP Request

The stored process that we just created can be invoked by SAS BI Web Services for Java middle-tier clients. A Web service client invokes the middle-tier Web service with an **`Execute()`** command. The SOAP request body, or client code, follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:sas="urn:schemas-microsoft-com:xml-analysis">
   <soapenv:Header/>
   <soapenv:Body>
      <sas:Execute>
         <sas:Command>
            <StoredProcess name="/Products/SAS Intelligence Platform/Samples/
               Sample: MEANS Procedure Web Service">
               <Parameter name="tablename">InData</Parameter>
               <Stream name="instream">
                  <Table>
                     <InData>
                        <Column1>1</Column1>
                        <Column2>20</Column2>
                        <Column3>99</Column3>
                     </InData>
                     <InData>
                        <Column1>50</Column1>
                        <Column2>200</Column2>
```

```
                        <Column3>9999</Column3>
                     </InData>
                     <InData>
                        <Column1>100</Column1>
                        <Column2>2000</Column2>
                        <Column3>1000000</Column3>
                     </InData>
                  </Table>
               </Stream>
            </StoredProcess>
         </sas:Command>
         <sas:Properties>
            <PropertyList>
               <DataSourceInfo>Provider=SASSPS;</DataSourceInfo>
            </PropertyList>
         </sas:Properties>
      </sas:Execute>
   </soapenv:Body>
</soapenv:Envelope>
```

SOAP Response

After you run the client code, the resulting SOAP response body is as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <n:ExecuteResponse xmlns:n="urn:schemas-microsoft-com:xml-analysis">
         <n:return>
            <TABLE>
               <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
                  <xs:element name="TABLE">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element ref="MEAN" minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
                  <xs:element name="MEAN">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="_TYPE_" minOccurs="0" type="xs:double"/>
                           <xs:element name="_FREQ_" minOccurs="0" type="xs:double"/>
                           <xs:element name="_STAT_" minOccurs="0">
                              <xs:simpleType>
                                 <xs:restriction base="xs:string">
                                    <xs:maxLength value="8"/>
                                 </xs:restriction>
                              </xs:simpleType>
                           </xs:element>
                           <xs:element name="COLUMN3" minOccurs="0" type="xs:double"/>
                           <xs:element name="COLUMN2" minOccurs="0" type="xs:double"/>
                           <xs:element name="COLUMN1" minOccurs="0" type="xs:double"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:schema>
               <MEAN>
```

```
                        <_TYPE_>0</_TYPE_>
                        <_FREQ_>3</_FREQ_>
                        <_STAT_>N</_STAT_>
                        <COLUMN3>3</COLUMN3>
                        <COLUMN2>3</COLUMN2>
                        <COLUMN1>3</COLUMN1>
                     </MEAN>
                     <MEAN>
                        <_TYPE_>0</_TYPE_>
                        <_FREQ_>3</_FREQ_>
                        <_STAT_>MIN</_STAT_>
                        <COLUMN3>99</COLUMN3>
                        <COLUMN2>20</COLUMN2>
                        <COLUMN1>1</COLUMN1>
                     </MEAN>
                     <MEAN>
                        <_TYPE_>0</_TYPE_>
                        <_FREQ_>3</_FREQ_>
                        <_STAT_>MAX</_STAT_>
                        <COLUMN3>1000000</COLUMN3>
                        <COLUMN2>2000</COLUMN2>
                        <COLUMN1>100</COLUMN1>
                     </MEAN>
                     <MEAN>
                        <_TYPE_>0</_TYPE_>
                        <_FREQ_>3</_FREQ_>
                        <_STAT_>MEAN</_STAT_>
                        <COLUMN3>336699.333</COLUMN3>
                        <COLUMN2>740</COLUMN2>
                        <COLUMN1>50.3333333</COLUMN1>
                     </MEAN>
                     <MEAN>
                        <_TYPE_>0</_TYPE_>
                        <_FREQ_>3</_FREQ_>
                        <_STAT_>STD</_STAT_>
                        <COLUMN3>574456.555</COLUMN3>
                        <COLUMN2>1094.89726</COLUMN2>
                        <COLUMN1>49.5008417</COLUMN1>
                     </MEAN>
                  </TABLE>
               </n:return>
            </n:ExecuteResponse>
         </soapenv:Body>
      </soapenv:Envelope>
```

Chapter 3

Using Structured Web Services

What Are Structured Web Services?

In SAS 9.3, all stored processes are available individually for execution using SAS BI Web Services without any action required on your part. You no longer have to run the Deploy as Web Service wizard in SAS Management Console. For every stored process, you can obtain a description of the structure of input and output Web service messages that can be used to invoke the stored process. The document that describes this structure is called a Web Service Description Language (WSDL) file. SAS BI Web Services automatically exposes a WSDL file for each and every stored process in your system. These WSDL files use XML to include detailed information about the inputs and outputs of each stored process using XML schema descriptions. Also, the WSDL file includes the URLs of endpoints to use to invoke these stored processes by using the SOAP protocol over HTTP. Typically, you use these WSDL files to automatically generate code in your client framework that can be used to invoke the Web services. Structured Web services are all SAS BI Web Services that can expose a WSDL file that describes their inputs and outputs. Structured Web services are all non-XMLA SAS BI Web Services.

Having all the type information in the WSDL is better suited to most client applications, and also makes things simpler for you. Making the WSDL more specific to the actual parameters instead of having a generic interface enables you to simplify the request XML. Making the WSDL more specific also makes it easier to consume the Web service with standard Web service client applications such as BizTalk, InfoPath, Word,

SharePoint, Excel, AJAX, Oracle WebLogic, and WebSphere, or in general any framework that can generate proxies from WSDL and schema.

In addition to being able to access stored processes directly from a Web service endpoint, you can also group multiple stored processes together into a single Web service. These Web services are called generated Web services. You might choose to use the grouping mechanism because it might reduce the amount of generated client code in your application. Many Web service client frameworks produce a proxy object for each Web service, and grouping stored processes into a single Web service could reduce the number of generated proxy objects to one.

All structured Web services support SOAP over HTTP and RESTful invocation using JSON and plain XML. Depending on the message format - SOAP, JSON, or plain XML - different features of structured Web services are available for you to use. SOAP is a recognized enterprise format with many existing standards that describe how to interact with Web services of this type. SAS BI Web Services support WS-Security for securing messages and the WS-I Attachments Profile for attaching binary content to request and response messages when using the SOAP transport. SOAP endpoints also expose a WSDL file that describes the inputs and outputs of the request and response messages, respectively. This enables you to use client frameworks to generate proxies that invoke these Web services and reduces the amount of code that you have to write to integrate with SAS Stored Processes.

Starting in SAS 9.3, SAS BI Web Services also expose all structured Web services as RESTful resources that can be invoked by using either JSON or plain XML inputs and outputs. JSON is a popular message format typically used by AJAX Web applications because JSON is a native data representation in JavaScript. However, JSON can be used from any client. Plain XML is an ideal message format for situations where SOAP libraries are not available or the complexity and features of SOAP are not desired. When using plain XML to communicate with SAS BI Web Services, use the same XML format as you would use with SOAP, but SOAP headers, SOAP elements, and namespaces can all be omitted.

See “Creating SAS BI Web Services” on page 3 for more information about creating and accessing Structured Web services.

Writing SAS Programs for Structured Web Services

In general, SAS programs for structured Web services can use all the functionality supported by SAS Stored Processes.

Consuming Input in SAS Programs

SAS BI Web Services can provide input to your SAS programs by using filerefs and macro variables. Filerefs enable you to stream arbitrary content such as raw XML or binary files from your Web service client to be used in your SAS code. Filerefs can be associated to librefs if the content that you send (such as XML) can be read by a SAS LIBNAME engine. The stored process engine automatically assigns filerefs for your stored process data sources so you do not need to explicitly include a FILENAME statement in your SAS code. For example, the following SAS program snippet simply reads from the instream fileref and prints the contents to the SAS log:

```
data _null_;
      infile instream;
```

```
      INPUT;
      PUT _INFILE_;
run;
```

No other SAS code is needed in this program; you have only to define a data source named instream in your stored process definition. See "Using Attachments with Web Services" on page 29 for more information about how to define your stored processes so that you can supply input data to your SAS programs.

You can include macro variables in your SAS program, and SAS BI Web Services set these macro variables based on the parameters that you supply during Web service invocation. For example, you can use this simple SAS program as a stored process to add two numbers together:

```
%let sum = %sysevalf(&num1 + &num2);
```

Define prompts with the stored process metadata to enable the passing of parameters to macro variables. The prompting framework provides additional information about the type of input macro variable and allows for additional validation before execution. See "Using Prompts with Generated Web Services" on page 31 for more information about defining prompts.

Retrieving Output Values from the SAS Program

You can use filerefs and macro variables as output just like you can with input. Output filerefs are automatically assigned by the stored process engine if the SAS code writes to the fileref. Expanding on the previous example, the following SAS program snippet copies from an input fileref to an output fileref:

```
data _null_;
      infile instream;
      file otstream;

      INPUT;
      PUT _INFILE_;
run;
```

Define the input data source instream and the output data target outstream in the stored process metadata. See "Using Attachments with Web Services" on page 29 for more information about how to define your stored processes so that you can retrieve output data from your SAS programs.

In the following example, the value of the Sum macro variable is automatically retrieved by the stored process engine when you define an output parameter in the stored process metadata:

```
%let sum = %sysevalf(&num1 + &num2);
```

Output parameters are similar to prompts, but there are fewer types of output parameters.

Note: When using the JSON message format, you are limited to prompts that have a simple string representation for input and you can retrieve values only from stored process output parameters. You cannot supply any stored process data sources when invoking SAS BI Web Services using JSON. If your stored process outputs data targets, packages, or streams to _WEBOUT, you cannot access these resources when using JSON. Remember this when you author SAS programs that you intend to use with JSON.

Accessing SOAP Endpoints for Stored Processes and Generated Web Services

Every structured Web service can be invoked by using SOAP. The full range of stored process input and output types are supported when using SOAP, as are attachments and WS-Security headers.

SOAP Endpoints for Stored Processes

To access the SOAP endpoint for a particular stored process, use the following pattern: **http://*host*:*port*/SASBIWS/services/*stored_process_path***. Replace *host* and *port* with the host name and port number where your SAS middle tier and SAS BI Web Services are running. If you do not know the host and port of your middle tier, you can find it by using the Configuration Manager plug-in for SAS Management Console or by asking your SAS administrator. Replace *stored_process_path* with the full metadata path of the stored process that you want to access. If there are any special characters that appear in the stored process name or path, you must URL encode those special characters (many Web service frameworks do this automatically). For example, replace any spaces in the path or name with the value %20.

For example, if your SAS middle tier is hosted on **my.company.com** on port 8080 and you want to access the endpoint for the stored process named **Sample: Hello World** that is stored in the metadata folder **/Products/SAS Intelligence Platform/Samples**, construct the following URL: **http://my.company.com:8080/SASBIWS/services/Products/SAS%20Intelligence%20Platform/Samples/Sample%3A%20Hello%20World**. You can access the WSDL files for these SOAP endpoints by appending **?WSDL** to the endpoint URL (for example, **http://my.company.com:8080/SASBIWS/services/Products/SAS%20Intelligence%20Platform/Samples/Sample%3A%20Hello%20World?WSDL**).

Note: By default, WSDLs are cached. The WSDL that is returned for the first request is also returned for all subsequent requests until you reset the WSDL cache. You can clear the cache by using the WsdlCache MBean, by reloading the SAS BI Web Services WAR file, or by restarting the application server. When the cache is cleared, it is cleared for all Web services and stored processes. You can reload a WSDL for an individual service or stored process by specifying **reload=true** as a query parameter when accessing the WSDL (for example, **http://localhost/SASBIWS/services/myService?wsdl&reload=true**).

SOAP Endpoints for Generated Web Services

When generating Web services using the Deploy as Web Service wizard in SAS Management console, the endpoint URL of the new service is shown after the wizard completes. This URL will be of the form **http://*host*:*port*/SASBIWS/services/*serviceName***. You can find the URL of a generated Web service endpoint at any time by using the Configuration Manager in SAS Management Console.

Using Attachments with Web Services

Streaming attachments can be defined in metadata as data sources (input attachments) and data targets (output attachments). Three types of streaming attachments are available:

XML stream
: specifies an attachment that is in-lined in the payload of the request or response as XML. You can also specify a schema for this data. The XML LIBNAME engine can generate schemas, and you can use the XML Mapper to map existing schema to the data. See the *SAS XML LIBNAME Engine: User's Guide* for more information. The schema is included in the generated WSDL for SOAP endpoints.

  Note: You can specify single streaming output the same way you do with the XMLA Web service by selecting **`Stream`** as the result capability. However, using data targets provides more flexibility because you can provide the name of the attachment as well as provide a schema that matches your expected data.

generic stream
: specifies an out-of-band binary attachment that is included with the request or response in one of the following ways:

  - If the attachment data is small, it can be included directly in the payload and encoded as Base64 binary data. This is the only means available for supplying attachments when using the plain XML message format.
  - If the attachment data is not small, then it is included out-of-band from the payload as a MIME multi-part related attachment where it is referenced from the payload via MTOM XOP/Include or SOAP with Attachments references (swaRef) when using the SOAP message format. When using RESTful access to BI Web Services, individual data output streams can be retrieved as stand-alone resources. For more information, see "Supported Types of Input and Output for XML and JSON Messages" on page 51.

  Note: With MTOM attachments, you can set the AttachmentOptimizedThreshold configuration setting to control when generic streams are Base64 encoded or optimized as XOP/Include attachments.

data table
: specifies that the input or output represents tabular data that is consumed in your SAS program by a LIBNAME engine.

  Data tables are a new feature in SAS 9.3 Stored Processes. They can remove the need to write LIBNAME statements in your stored process code and enable that code to be more flexible and reusable. You can specify a template table in the stored process data table definition. A template table points to a table that is registered in metadata. When a stored process has a template for a source data table, that table's structure is examined in order to generate very specific input schema automatically for the SOAP endpoint's WSDL file. The generated schema describes precisely what columns clients need to supply for each row of input data in a **`<ClientTable>`** element. Clients can choose to supply a LIBNAME and MEMBERNAME pair in a **`<ServerTable>`** element for a source data table. The stored process engine then automatically assigns this LIBNAME for you.

  Target data tables are very similar to source data tables but they return tabular data to the client. If you specify that a target data table is a **`<ServerTable>`** when invoking a Web service, then the output that is sent to that target in your SAS code is

automatically stored in the LIBNAME and MEMBERNAME that you specify in the **<ServerTable>** element. If you specify that a target data table is a **<ClientTable>** when invoking a Web service, then the output that is sent to that target in your SAS code is sent back to the Web service client. Also, if you specify a template for a target data table, then the generated schema for that Web service precisely describes the structure of the tabular output XML returned from the Web service in a **<ClientTable>**. If you use a template with a target data table and specify that the output is a **<ServerTable>** when invoking the service, then the output that is sent to that target in your SAS code is automatically written to the specified template table. See the *SAS Stored Processes: Developer's Guide* for more information about writing SAS programs that use data tables.

The following code is an example of a schema definition for a Web service that expects one generic (binary) stream as an output response:

```
<element name="stpAllParm1Response">
   <complexType>
      <sequence>
         <element name="stpAllParm1Result">
            <complexType>
               <sequence>
                  <element maxOccurs="1" minOccurs="0" name="Streams">
                     <complexType>
                        <sequence>
                           <element maxOccurs="1" minOccurs="0" name="myAttachment">
                              <complexType>
                                 <sequence>
                                    <element name="Value" type="base64Binary"/>
                                 </sequence>
                                 <attribute name="contentType" type="string"/>
                              </complexType>
                           </element>
                        </sequence>
                     </complexType>
                  </element>
               </sequence>
             </complexType>
         </element>
      </sequence>
   </complexType>
</element>
```

In this generated schema, **myAttachment** is the name of the element that represents the output attachment. This name is defined by the user in metadata. This element is a container for the actual value of that attachment. The content type of the attachment can be returned as an attribute to further clarify the content of the data within the attachment.

Package type output is also supported. This type of output produces one or more attachments and packages them together as a single entity. To enable this type of output, select **Package** as the result capability for the stored process. The SAS code needs to produce packages to take advantage of this feature. There are several ways to produce packages in SAS code, including ODS and the Publishing Framework. See the *SAS Stored Processes: Developer's Guide* for more information about using packages in your stored processes.

Attachment definitions in metadata provide a means to establish a contract between all parties involved in a Web service request. SAS BI Web Services generates a WSDL and schema based on metadata definitions that provide a contract between the client and

Web service. The Web service enforces that all required attachments are sent in the request. The SAS code that executes on the SAS server must be written in accordance to the metadata definitions that it is representing. Otherwise, problems might occur (for example, not reading the correct stream) resulting in SAS errors. If a SAS error occurs, the Web service returns a SOAP fault to the client when using SOAP endpoints or an HTTP error when using plain XML or JSON in RESTful invocations.

Using Prompts with Generated Web Services

When defining stored process parameters, the prompt type definitions are mapped to Web service schema. These schema mappings appear in the WSDL files for SOAP endpoints, but the type mappings still apply for other endpoints. You should use the same input XML for the plain XML endpoints as you do for the SOAP endpoints, however you can omit the namespace prefixes and declarations for the plain XML endpoints. See "Supported Types of Input and Output for XML and JSON Messages" on page 51 for more information about using prompts with these endpoint types.

The mapping of stored process prompt types to XML schema types is as follows:

Table 3.1 *How Prompt Types Map to Web Service Schema*

Prompt Type	Parameter Type in Generated WSDL
Text, Date, Time, Color, Data source, File or directory, Data library	**`xs:string`**
Numeric	**`xs:int`** or **`xs:double`**
Ranges have **`lowerBound`** and **`upperBound`** elements	**`xs:`***type* (where *type* is the appropriate value, such as **`int`** or **`dateTime`**, from this table) **`xs:string`** (for **`lowerBound`** and **`upperBound`** elements)
Timestamp	**`xs:dateTime`**
Data source item has **`path`** and **`itemName`** elements	**`xs:string`** (for **`path`** and **`itemName`** elements)
OLAP member has **`label`** and **`uniqueName`** elements	**`xs:string`** (for **`label`** and **`uniqueName`** elements)

The **`xs:`** prefix in these values is an abbreviation for the namespace that is being used. This particular abbreviation stands for the standard XML schema namespace, **`http://www.w3.org/2001/XMLSchema`**. For more information about the XML schema, see **`http://www.w3.org/2001/XMLSchema`**.

For generated Web services, the WSDL that is generated uses facets and restrictions that are based on prompt constraints. Values are validated against the constraints that are defined in metadata.

The following table explains how to format values for the various prompt types:

Table 3.2 *Guidelines for Entering Prompt Values (U.S. English Locale)*

Prompt Type	Guidelines	Examples
Text	Enter any character value. Blank spaces and nonprintable characters can be used, but the value cannot consist completely of these characters. Trailing blanks are stored as part of the value and are included when the value is validated against the minimum and maximum length requirements.	• `you are here` • `eighty-five` • `Bob`
Numeric	Enter a standard numeric value. • If you are working with an integer prompt, then do not use values with decimal places. If you use a value with zeros after the decimal point (for example, `1.00`) for an integer prompt, then the zeros and the decimal point will be removed before the value is stored (for example, `1.00` will be stored as `1`). • For prompts that allow floating-point values, the unformatted prompt value can contain up to 15 significant digits. Values with more than 15 significant digits of precision are truncated. Note that formatted values can have more than 15 significant digits.	• `1.25` • `6000` • `2222.444`

Prompt Type	Guidelines	Examples
Date	For dates of type Day, enter values in one of the following formats: • *ddmmmyyyy* • *ddmonth-nameyyyy* (Java only) • *mm/dd/yy<yy>* • *mm.dd.yy<yy>* • *mm-dd-yy<yy>* • *mmm/dd/yy<yy>* • *mmm.dd.yy<yy>* • *mmm-dd-yy<yy>* • *mmm dd, yyyy* (Java only) • *month-name/dd/yy<yy>* (Java only) • *month-name.dd.yy<yy>* (Java only) • *month-name-dd-yy<yy>* (Java only) • *month-name dd, yyyy* • *day-of-week, mmm dd, yy* (Java only) • *day-of-week, mmm dd, yyyy* (Java only) • *day-of-week, month-name dd, yy* (Java only) • *day-of-week, month-name dd, yyyy* • *yyyy/mm/dd* (Java only) • *yyyy.mm.dd* (Java only) • *yyyy-mm-dd* (Java only) • *yyyy.mmm.dd* (Java only) • *yyyy-mmm-dd* (Java only) • *yyyy.month-name.dd* (Java only) • *yyyy-month-name-dd* (Java only) Here is an explanation of the syntax: *day-of-week* specifies either the first three letters of the day of the week or the full name of the day of the week. This value is not case sensitive. *dd* specifies a one-digit or two-digit integer that represents the day of the month. *mm* specifies a one-digit or two-digit integer that represents the month of the year. *mmm* or *month-name* specifies the first three letters of the full name of the month, or the full name of the month, respectively. This value is not case sensitive. *yy* or *yyyy* specifies a two-digit or four-digit integer that represents the year. To refer to a year that is more than 80 years in the past or 20 years in the future, use four digits. Valid values for a four-digit year range from 1600 to 2400.	• `4APR1860` • `14January1918` • `12/14/45` • `02.15.1956` • `1-1-60` • `Oct/02/08` • `JUL.20.13` • `MAY-13-1924` • `Oct 05, 2006` • `February/10/00` • `March.1.2004` • `DECEMBER-25-08` • `SEPTEMBER 20, 2010` • `FRI, Jan 3, 20` • `Tuesday, Jan 15, 2008` • `Monday, January 16, 40` • `FRIDAY, JANUARY 04, 2008` • `2041/5/13` • `2050.07.25` • `2100-1-1` • `2009.NOV.02` • `2400-Aug-8` • `2101.December.31` • `1919-APRIL-20`

Prompt Type	Guidelines	Examples
Date (cont'd.)	For dates of type Week, enter values in one of the following formats: • **W***ww yy* • **W***ww yyyy* (Java only) • **Week***ww yyyy* Here is an explanation of the syntax: *ww* specifies a one-digit or two-digit integer that represents the week of the year. Valid values range from 1 to 52. *yy* or *yyyy* specifies a two-digit or four-digit integer that represents the year. To refer to a year that is more than 80 years in the past or 20 years in the future, use four digits. Valid values for a four-digit year range from 1600 to 2400.	• **W1 08** • **W52 1910** • **Week 20 2020** • **Week 5 2048**
	For dates of type Month, enter values in one of the following formats: • *mm/yy<yy>* • *mm.yy<yy>* • *mm-yy<yy>* • *mmm yy<yy>* (Java only) • *mmm/yy<yy>* • *mmm.yy<yy>* • *mmm-yy<yy>* • *month-name yy* (Java only) • *month-name yyyy* • *month-name/yy<yy>* (Java only) • *month-name.yy<yy>* (Java only) • *month-name-yy<yy>* (Java only) Here is an explanation of the syntax: *mm* specifies a one-digit or two-digit integer that represents the month of the year. *mmm* or *month-name* specifies the first three letters of the full name of the month, or the full name of the month, respectively. This value is not case sensitive. *yy* or *yyyy* specifies a two-digit or four-digit integer that represents the year. To refer to a year that is more than 80 years in the past or 20 years in the future, use four digits. Valid values for a four-digit year range from 1600 to 2400.	• **12/1828** • **06.65** • **7-76** • **Jul 08** • **JUN/2010** • **SEP.20** • **Oct-2050** • **August 20** • **OCTOBER 1975** • **MARCH/1970** • **May.13** • **November-18**

Prompt Type	Guidelines	Examples
Date (cont'd.)	For dates of type Quarter, enter values in the following format: • *quarter-name* **`quarter`** *yy<yy>* Here is an explanation of the syntax: *quarter-name* specifies the quarter of the year. Valid values are **`1st`**, **`2nd`**, **`3rd`**, and **`4th`**. *yy* or *yyyy* specifies a two-digit or four-digit integer that represents the year. To refer to a year that is more than 80 years in the past or 20 years in the future, use four digits. Valid values for a four-digit year range from 1600 to 2400.	• **`1st quarter 1900`** • **`2nd quarter 50`** • **`3rd quarter 12`** • **`4th quarter 2060`**
	For dates of type Year, enter values in one of the following formats: • *yy* (Java only) • *yyyy* Here is an explanation of the syntax: *yy* or *yyyy* specifies a two-digit or four-digit integer that represents the year. To refer to a year that is more than 80 years in the past or 20 years in the future, use four digits. Valid values for a four-digit year range from 1600 to 2400.	• **`1895`** • **`86`** • **`08`** • **`2035`**

Prompt Type	Guidelines	Examples
Time	Enter time values in the following format: • *hh:mm<:ss>* <**AM** | **PM**> Here is an explanation of the syntax: *hh* specifies a one-digit or two-digit integer that represents the hour of the day. Valid values range from 0 to 24. *mm* specifies a one-digit or two-digit integer that represents the minute of the hour. Valid values range from 0 to 59. *ss* (optional) specifies a one-digit or two-digit integer that represents the second of the minute. Valid values range from 0 to 59. If this value is not specified, then the value defaults to 00 seconds. **AM** or **PM** (optional) specifies either the time period 00:01 to 12:00 noon (AM) or the time period 12:01 to 12:00 midnight (PM). If this value is not specified and you are using the 12-hour system for specifying time, then the value defaults to **AM**. Do not specify **AM** or **PM** if you are using the 24-hour system for specifying time.	• **1:1** • **1:01 AM** • **13:1:1** • **01:01:01 PM** • **22:05**

Prompt Type	Guidelines	Examples
Timestamp	Enter timestamp values in the following format: • *yyyy-mm-dd*T*hh:mm:ss* Here is an explanation of the syntax: *yyyy* specifies a four-digit integer that represents the year. Valid values for a four-digit year range from 1600 to 2400. *mm* specifies a one-digit or two-digit integer that represents the month of the year. *dd* specifies a one-digit or two-digit integer that represents the day of the month. *hh* specifies a one-digit or two-digit integer that represents the hour of the day. Valid values range from 0 to 24. *mm* specifies a one-digit or two-digit integer that represents the minute of the hour. Valid values range from 0 to 59. *ss* specifies a one-digit or two-digit integer that represents the second of the minute. Valid values range from 0 to 59.	• **`2012-11-23T15:30:32`** • **`2008-09-09T01:01:01`**
Color	Enter color values in one of the following formats: • **`CX`***rrggbb* • **`0x`***rrggbb* • **`#`***rrggbb* Here is an explanation of the syntax: *rr* specifies the red component. *gg* specifies the green component. *bb* specifies the blue component. Each component should be specified as a hexadecimal value that ranges from 00 to FF, where lower values are darker and higher values are brighter.	Bright red • **`CXFF0000`** • **`0xFF0000`** • **`#FF0000`** Black • **`CX000000`** • **`0x000000`** • **`#000000`** White • **`CXFFFFFF`** • **`0xFFFFFF`** • **`#FFFFFF`**

Prompt Type	Guidelines	Examples
Data source	Enter the name and location of a data source in the following format: • */folder-name-1/<.../folder-name-n/>data-source-name(type)* Here is an explanation of the syntax: */folder-name-1/<.../folder-name-n/>* specifies the location of the data source. *data-source-name* specifies the name of the data source. *type* is the type of data source. The following values are valid unless otherwise noted: **Table**, **InformationMap**, and **Cube**. Use **InformationMap** for specifying either relational or OLAP information maps.	• **/Shared Data/Tables/OrionStar/Customers(Table)** • **/Users/MarcelDupree/My Folder/My Information Map(InformationMap)** • **/MyCustomRepository/More Data/Order_Facts(Table)**
File or directory	Enter the name and location of a file or directory in the following format: • *directory-specification<filename>* Here is an explanation of the syntax: *directory-specification* specifies the location of the file or directory in the file system of a SAS server. *filename* specifies the name of the file. This value is required only if the prompt is a file prompt. Depending on the operating environment that the SAS server runs in, you might need to put a forward slash (/) or a backslash (\) between the directory specification and the name of the file.	• **C:\Documents and Settings\All Users\Documents\myfile.txt**
Data library	Enter the name and location of a data library in the following format: • */folder-name-1/<.../folder-name-n/>library-name* **(Library)** Here is an explanation of the syntax: */folder-name-1/<.../folder-name-n/>* specifies the location of the library. *library-name* specifies the name of the library.	• **/Data/Libraries/Customer Data Library(Library)** • **/MyCustomRepository/More Data/OracleData(Library)**

Prompt Type	Guidelines	Examples
Data source item	For the path element, enter the path for a data source item in the following format: • */folder-name-1/<.../folder-name-n/>data-source-name(type)* Here is an explanation of the syntax: */folder-name-1/<.../folder-name-n/>* specifies the location of the data source. *data-source-name* specifies the name of the data source. *type* is the type of data source. The following values are valid unless otherwise noted: **`Table`** or **`InformationMap`**. Use **`InformationMap`** for specifying either relational or OLAP information maps. For the itemName element, enter the name for the data source item in the following format: • *item-name* Here is an explanation of the syntax: *item-name* specifies the name of the data source item. This is the name of a column in a table or a data item in an information map.	path • **`/Shared Data/Tables/MYDATA(Table)`** itemName • **`Year`**
OLAP member	For the uniqueName element, enter the name of the OLAP member. For the label element, enter the label for the OLAP member.	uniqueName • **`PRICEAVG`** label • **`Average Price`**

Note: An anonymous user cannot launch workspace servers. Dynamic prompt validation requires use of workspace servers if the user has been defined with an internal account. Thus, internal anonymous users will not be able to use all stored processes.

Prompt definitions in metadata provide a means to establish a contract between all parties that are involved in a Web service request. SAS BI Web Services generate a WSDL and schema based on metadata definitions that provide a contract between the client and generated Web service. Mature client programming tools can help assist clients in formulating valid SOAP requests. Web service endpoints also validate client input values via the SAS prompting framework. The SAS code that executes on the SAS server must be written in accordance to the metadata definitions that it represents. Otherwise, problems occur (for example, reading the wrong input parameter or expecting a different type) that result in SAS errors. If a SAS error occurs, then the generated Web service returns a SOAP fault to the client for SOAP endpoints or HTTP errors for JSON and plain XML endpoints.

Sample WSDLs

Sample Parameters

The following table contains names, prompt types, and restrictions for sample parameters for a stored process.

Table 3.3 *Sample Stored Process Parameters*

Prompt Name	Prompt Type	Restrictions
top_level	Text	Single value
fixed	Text	Read-only values Single value Default value: **`fixed default`**
simple_string	Text	Single value
invisible	Text	Hide from user Single value Default value: **`hidden val`**
default	Text	Single value Default value: **`def val`**
static_list	Text	Multiple ordered values
max_length	Text	Single value Maximum length: **`6`**
mult_entry	Text	Multiple values Maximum value count: **`5`**
text_range	Text range	Default range from: **`aaa`** Default range to: **`zzz`**
req_string	Text	Requires a non-blank value Single value
simple_int	Numeric (integer)	Single value Allows only integer values

Prompt Name	Prompt Type	Restrictions
fixed_int	Numeric (integer)	Read-only values Single value Allows only integer values Default value: `12345`
def_int	Numeric (integer)	Single value Allows only integer values Default value: `12345`
int_list	Numeric (integer)	Multiple ordered values Allows only integer values
int_mult	Numeric (integer)	Requires a non-blank value Multiple values Allows only integer values Minimum value count: `1` Maximum value count: `5`
lim_int	Numeric (integer)	Single value Allows only integer values Minimum value allowed: `1` Maximum value allowed: `99`
req_int	Numeric (integer)	Requires a non-blank value Single value Allows only integer values Default value: `9999`
simple_float	Numeric (double)	Single value
def_float	Numeric (double)	Single value Minimum number of decimal places displayed: `1` Maximum number of decimal places displayed:`3` Minimum value allowed: `1.0` Maximum value allowed: `100.0` Default value: `99.99`

Prompt Name	Prompt Type	Restrictions
float_list	Numeric (double)	Multiple values Minimum number of decimal places displayed: **1** Maximum number of decimal places displayed: **3**
float_mult	Numeric (double)	Multiple values Maximum number of decimal places displayed: **4** Maximum value count: **5** Maximum value allowed: **999999.0**
lim_float	Numeric (double)	Single value Minimum value allowed: **10.0** Maximum value allowed: **20.0**
req_float	Numeric (double)	Requires a non-blank value Single value Default value: **99.0**
simple_color	Color	
def_color	Color	Default value: **CXFF0000**
fixed_color	Color	Read-only values Default value: **CX0000FF**
req_color	Color	Requires a non-blank value Default value: **CXFFFF00**
simple_date	Date	Single value
def_date	Date	Single value Default value: **Today**
date_list	Date	Multiple values Minimum value allowed: **October 01, 2007** Maximum value allowed: **N days from now (200)**
date_range	Date range	Minimum value allowed: **October 01, 2007** Maximum value allowed: **N days from now (300)**

Prompt Name	Prompt Type	Restrictions
req_date	Date	Requires a non-blank value Single value Include special values: Missing values Default value: **Week 50 2007**
simple_time	Time	
fixed_time	Time	Read-only values Default value: **Current hour**
def_time	Time	Minimum value allowed: **N hours ago (1)** Maximum value allowed: **N hours from now (1)** Default value: **Current hour**
timerange	Time range	Default range type: Custom Default range from: **N hours ago (10)** Default range to: **N hours from now (1)**
file1	File or directory	
data_source	Data source	Default value: **/Stored Processes/ CARS(Table)**
data_source_item	Data source item	Single value Default value: **Make [Make] [/Stored Processes/CARS]**
data_library	Data library	Default value library: **/ Stored Processes/ WsmSASHelp(Library)** Default value libref: **myref**
olap_member	OLAP member	Single value

Generated WSDL for Java

If a Web service is generated for a stored process with these sample parameters, the following WSDL is generated for Java:

```
<types>
    <schema elementFormDefault="qualified" targetNamespace=
      "http://tempuri.org/AllPromptTypes"
      xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns=
      "http://tempuri.org/AllPromptTypes">
         <annotation>
            <documentation>SAS BI Web Services generated schema</documentation>
         </annotation>
         <element name="stpAllParm1">
            <complexType>
               <sequence>
                  <element name="parameters" type="tns:stpAllParm1Parameters"/>
               </sequence>
            </complexType>
         </element>
         <complexType name="stpAllParm1Parameters">
            <sequence>
               <element maxOccurs="1" minOccurs="0" name="top_level" type="string"/>
               <element maxOccurs="1" minOccurs="0" name="simple_string" type="string"/>
               <element default="def val" maxOccurs="1" minOccurs="0" name="default"
                  type="string"/>
               <element maxOccurs="1" minOccurs="0" name="static_list">
                  <complexType>
                     <sequence>
                        <element maxOccurs="unbounded" minOccurs="0" name="Item"
                           type="string"/>
                     </sequence>
                  </complexType>
               </element>
               <element maxOccurs="1" minOccurs="0" name="max_length">
                  <simpleType>
                     <restriction base="string">
                        <maxLength value="6"/>
                     </restriction>
                  </simpleType>
               </element>
               <element maxOccurs="1" minOccurs="0" name="mult_entry">
                  <complexType>
                     <sequence>
                        <element maxOccurs="5" minOccurs="0" name="Item" type="string"/>
                     </sequence>
                  </complexType>
               </element>
               <element maxOccurs="1" minOccurs="0" name="text_range">
                  <complexType>
                     <sequence>
                        <element name="LowerBound" type="string"/>
                        <element name="UpperBound" type="string"/>
                     </sequence>
                  </complexType>
               </element>
               <element maxOccurs="1" minOccurs="1" name="req_string" type="string"/>
               <element maxOccurs="1" minOccurs="0" name="simple_int" type="int"/>
               <element default="12345" maxOccurs="1" minOccurs="0" name="def_int"
                  type="int"/>
               <element maxOccurs="1" minOccurs="0" name="int_list">
```

```
   <complexType>
      <sequence>
         <element maxOccurs="unbounded" minOccurs="0" name="Item"
            type="int"/>
      </sequence>
   </complexType>
</element>
<element maxOccurs="1" minOccurs="1" name="int_mult">
   <complexType>
      <sequence>
         <element maxOccurs="5" minOccurs="1" name="Item" type="int"/>
      </sequence>
   </complexType>
</element>
<element maxOccurs="1" minOccurs="0" name="lim_int">
   <simpleType>
      <restriction base="int">
         <minInclusive value="1"/>
         <maxInclusive value="99"/>
      </restriction>
   </simpleType>
</element>
<element default="9999" maxOccurs="1" minOccurs="1" name="req_int"
   type="int"/>
<element maxOccurs="1" minOccurs="0" name="simple_float" type="double"/>
<element default="99.99" maxOccurs="1" minOccurs="0" name="def_float">
   <simpleType>
      <restriction base="double">
         <minInclusive value="1.0"/>
         <maxInclusive value="100.0"/>
      </restriction>
   </simpleType>
</element>
<element maxOccurs="1" minOccurs="0" name="float_list">
   <complexType>
      <sequence>
         <element maxOccurs="unbounded" minOccurs="0" name="Item"
            type="double"/>
      </sequence>
   </complexType>
</element>
<element maxOccurs="1" minOccurs="0" name="float_mult">
   <complexType>
      <sequence>
         <element maxOccurs="5" minOccurs="0" name="Item">
            <simpleType>
               <restriction base="double">
                  <maxInclusive value="999999.0"/>
               </restriction>
            </simpleType>
         </element>
      </sequence>
   </complexType>
</element>
<element maxOccurs="1" minOccurs="0" name="lim_float">
   <simpleType>
```

```
      <restriction base="double">
         <minInclusive value="10.0"/>
         <maxInclusive value="20.0"/>
      </restriction>
   </simpleType>
</element>
<element default="99.0" maxOccurs="1" minOccurs="1" name="req_float"
   type="double"/>
<element maxOccurs="1" minOccurs="0" name="simple_color" type="string"/>
<element default="cxff0000" maxOccurs="1" minOccurs="0" name="def_color"
   type="string"/>
<element default="cxffff00" maxOccurs="1" minOccurs="1" name="req_color"
   type="string"/>
<element maxOccurs="1" minOccurs="0" name="simple_date" type="string"/>
<element default="D0D" maxOccurs="1" minOccurs="0" name="def_date"
   type="string"/>
<element maxOccurs="1" minOccurs="0" name="date_list">
   <complexType>
      <sequence>
         <element maxOccurs="unbounded" minOccurs="0" name="Item">
            <simpleType>
               <restriction base="string">
                  <enumeration value="October 05, 2007"/>
                  <enumeration value="October 31, 2007"/>
               </restriction>
            </simpleType>
         </element>
      </sequence>
   </complexType>
</element>
<element maxOccurs="1" minOccurs="0" name="date_range">
   <complexType>
      <sequence>
         <element name="LowerBound" type="string"/>
         <element name="UpperBound" type="string"/>
      </sequence>
   </complexType>
</element>
<element default="Week 50 2007" maxOccurs="1" minOccurs="1"
   name="req_date" nillable="true">
   <complexType>
      <simpleContent>
         <extension base="string">
            <attribute name="missing">
               <simpleType>
                  <restriction base="string">
                     <pattern value="[_.A-Z]"/>
                  </restriction>
               </simpleType>
            </attribute>
         </extension>
      </simpleContent>
   </complexType>
</element>
<element maxOccurs="1" minOccurs="0" name="simple_time" type="string"/>
<element default="H0H" maxOccurs="1" minOccurs="0" name="def_time"
```

```
            type="string"/>
         <element maxOccurs="1" minOccurs="0" name="timerange">
            <complexType>
               <sequence>
                  <element name="LowerBound" type="string"/>
                  <element name="UpperBound" type="string"/>
               </sequence>
            </complexType>
         </element>
         <element maxOccurs="1" minOccurs="0" name="file1" type="string"/>
         <element maxOccurs="1" minOccurs="0" name="data_source" type="string"/>
         <element maxOccurs="1" minOccurs="0" name="data_source_item">
            <complexType>
               <sequence>
                  <element maxOccurs="unbounded" name="DataSourceItem">
                     <complexType>
                        <sequence>
                           <element maxOccurs="1" minOccurs="1" name="Path"
                              type="string"/>
                           <element maxOccurs="1" minOccurs="1" name="ItemName"
                              type="string"/>
                        </sequence>
                     </complexType>
                  </element>
               </sequence>
            </complexType>
         </element>
         <element maxOccurs="1" minOccurs="0" name="data_library" type="string"/>
         <element maxOccurs="1" minOccurs="0" name="olap_member">
            <complexType>
               <sequence>
                  <element maxOccurs="unbounded" name="OlapMember">
                     <complexType>
                        <sequence>
                           <element maxOccurs="1" minOccurs="1" name="UniqueName"
                              type="string"/>
                           <element maxOccurs="1" minOccurs="0" name="Label"
                              type="string"/>
                        </sequence>
                     </complexType>
                  </element>
               </sequence>
            </complexType>
         </element>
      </sequence>
   </complexType>
   <element name="stpAllParm1Response">
      <complexType>
         <sequence>
            <element name="stpAllParm1Result">
               <complexType>
                  <sequence>
                     <element maxOccurs="1" minOccurs="0" name="Streams">
                        <complexType>
                           <sequence>
                              <element maxOccurs="1" minOccurs="0" name="_WEBOUT">
```

```
                                                <complexType>
                                                   <sequence>
                                                      <element name="Value" type="base64Binary"/>
                                                   </sequence>
                                                   <attribute name="contentType" type="string"/>
                                                </complexType>
                                             </element>
                                          </sequence>
                                       </complexType>
                                    </element>
                                 </sequence>
                              </complexType>
                           </element>
                        </sequence>
                     </complexType>
                  </element>
               </schema>
            </types>
```

Chapter 4

Using JSON and Plain XML with RESTful Web Services

What Are REST and JSON?

REST

Starting in SAS 9.3, SAS BI Web Services also exposes all structured Web services as RESTful resources that can be invoked using either JSON or plain XML inputs and outputs. REST stands for Representational State Transfer and describes a pattern for interacting with content on remote systems, typically using HTTP. REST describes a way that you can access and modify existing content and also how to add content to a system. RESTful HTTP Web services use the standard set of HTTP verbs to indicate the action the user wants to perform. For example, if you wanted to retrieve a value from a RESTful HTTP Web service, you would use the HTTP verb GET. If you wanted to send some data to a RESTful HTTP Web service, you would use the HTTP verb POST. Because REST is an architectural concept and not a standard, there are no set rules for how a client and service should use the set of HTTP verbs, so it is ultimately up to each service to decide how to implement a RESTful architecture. SAS BI Web Services expects clients to use GET when invoking stored processes that require no input and POST when invoking stored processes that require prompt or stream input. Conceptually, you can think of stored processes that require no input as a resource that you simply want to retrieve, and that is why you use GET. Conversely, stored processes that need input require that you POST that input to the resource.

In addition to using standard HTTP verbs, RESTful HTTP Web services emphasize the representation of resources in the form of URLs. Therefore, instead of exposing multiple operations and resources through a single endpoint as SOAP services typically do, RESTful Web services delineate resources by making them available at different URLs.

This means that you can access all the results of a stored process invocation at a URL with a suffix such as **`/rest/storedProcesses/path/to/stored/process`**, but you can also access specific output resources at URLs with a suffix such as **`/rest/storedProcesses/path/to/stored/process/parameters/myOutputParameter`** and **`/rest/storedProcesses/path/to/stored/process/streams/myOutputStream`**. SAS BI Web Services treat output parameters, output streams (also known as data targets), and packages as distinct resources that can be retrieved individually and separately from any other output. Note that even though you can request a distinct output resource, every time you do so the stored process is executed and all results are retrieved by SAS BI Web Services. This means that requesting specific resources will not make execution any quicker or more efficient in this version of the release (but it makes the client code simpler). When you request a distinct output resource using a RESTful URL, only that specific resource is returned to your client. For example, if your stored process writes a PDF to a data target and you request that specific output stream resource using a RESTful URL, then the data returned to the client is the actual PDF binary contents and the HTTP response includes the proper HTTP content type headers. For more information about the output resources available from RESTful Web services and information about how to access RESTful Web services, see "Accessing RESTful JSON and XML Web Service Endpoints" on page 54.

REST does not prescribe a specific message format to use during HTTP resource exchanges. It is up to the RESTful Web service to choose which formats are supported. XML and JavaScript Object Notation (JSON) are two of the most popular formats used by RESTful Web services. SAS BI Web Services supports both of these message formats.

RESTful Message Formats

XML

The format of input and output XML messages when using RESTful SAS BI Web Service endpoints mimics the SOAP format for a given stored process or generated Web service. The only differences are:

- None of the SOAP XML elements should be present in requests or responses.
- Namespaces are optional in requests. If they are used in the request, then they are used in the response. If they are used with the REST endpoint for a generated Web service, then they must match the namespace of the generated Web service. Generally, it is advisable to avoid using namespaces for the plain XML message format.
- Binary content is Base64 encoded and inlined.

The similarities between the plain XML format and the SOAP message format make it easier to write client code if there are no tools that can automatically generate client stubs.

JSON

JSON is a simple text-based message format that is often used with RESTful Web services. Like XML, it is designed to be readable, and this can help when debugging and testing. JSON is derived from JavaScript, and therefore is very popular as a data format in Web applications. Because JSON has extensive support in JavaScript, it is often used in AJAX Web applications for creating rich, dynamic user experiences that incorporate remote data and service execution. However, JSON can be read and written by many programming languages.

JSON has only a limited set of basic, built-in data types. Therefore, SAS BI Web Services supports only simple prompt types and output parameters when using JSON.

Supported Types of Input and Output for XML and JSON Messages

When using the JSON and XML message formats with RESTful Web service endpoints, the type and format of inputs and outputs you can use is limited compared to the features of SOAP endpoints. Stored processes can accept two types of input: prompts and data sources. They can produce three types of output: output parameters, data targets, and packages. Prompts enable you to supply simple parameters for stored process execution. The SAS prompting framework enables you to create prompt definitions that describe the type and format of your stored process's input parameters. When you invoke a Web service, the prompting framework validates the values that you supplied for the stored process prompts. The prompting framework translates these values into a macro variable that can be used in your stored process. Data sources enable you to supply arbitrary streams of data to your stored process to be processed. These data streams can be text such as XML or raw binary content. The data that you supply in a data source is streamed to a fileref in your stored process. Output parameters contain the values from macro variables from your stored process after it is finished running. This enables you to return simple values from your stored process. Data targets are like data sources in that they are streams of arbitrary data. However, data targets are produced by the stored process. Stored processes can use the SAS Publishing Framework to create a package during stored process execution. A package is a collection of assets that can be returned to the stored process client (in the case of SAS BI Web Services, the client is the Web service client). Packages can contain binary and textual data and are a convenient way to package complex reports that contain multiple images and text or HTML content produced by SAS. If a stored process returns a package, that package can be returned to the Web service client as a list of entries and the contents of those entries.

Supported Input and Output for XML Messages

When using RESTful XML SAS BI Web Services endpoints, all types of input and output are supported. You should send XML requests using the same format as for SOAP messages. Always omit SOAP elements from requests to the RESTful XML endpoints, and you can also omit namespaces.

Table 4.1 *SOAP Message Versus Plain XML Message*

SOAP Message	Plain XML Message
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.or xmlns:biw="http://www.sas.com/xml/namesp <soapenv:Header/> <soapenv:Body> <biw:addfloats> <biw:parameters> <biw:num1>2.3</biw:num1> <biw:num2>4.2</biw:num2> </biw:parameters> </biw:addfloats> </soapenv:Body> </soapenv:Envelope>	<addfloats> <parameters> <num1>2.3</num1> <num2>4.2</num2> </parameters> </addfloats>

The other difference between SOAP and plain XML messages is how binary content is handled. Data sources, data targets, and packages can all contain binary data. SOAP endpoints use WS-* standards to handle binary content. Specifically, SOAP can use attachments to include the binary content out-of-band of the SOAP envelope (but still within the same HTTP request or response) using multi-part responses. Plain XML messages contain Base64-encoded data inline in the XML request or response. This is less efficient than SOAP attachments because serializing the binary content to Base64 takes extra processing time and also bloats the size of the binary data compared to its original size. Therefore, avoid using plain XML messages when using multiple large binary resources.

If you need to retrieve only a single output resource (binary or otherwise), you can access the specific output resource that you need by using the appropriate resource suffix on your Web service endpoint URL. When accessing a single binary output resource with resource suffixes, the binary data is not transcoded to Base64 data. See “Accessing RESTful JSON and XML Web Service Endpoints” on page 54 for more information about how to access these resources using suffixes.

Supported Input and Output for JSON Messages

Because JSON supports only a limited number of native data types, JSON SAS BI Web Service endpoints accept only simple values for stored process prompts and can return only results from output parameters (not data targets or packages). The JSON SAS BI Web Service endpoints support prompt values that are supplied using application/x-www-form-urlencoded encoding in the HTTP request body. You must use HTTP POST when supplying values to JSON endpoints. You can use HTTP GET when your stored process does not require any input. For example, a JSON invocation of a stored process that adds two numbers, num1 and num2, might look like the following:

Table 4.2 *JSON Invocation of a Stored Process*

Request	Response

`num1=2.0&num2=3.5`	`{`
	`  "outputParameters":`
	`  {`
	`  "Sum": "5.5"`
	`  }`
	`}`

Be sure to specify the Content-type HTTP header as application/x-www-form-urlencoded when you use POST. The following table lists supported prompt types for JSON messages (note that no multi-value prompt types, including ranges and selections, are supported when using JSON):

Table 4.3 *Supported Prompt Types for JSON Messages*

Prompt Type	Supported	Notes
Text	yes	
Numeric	yes	
Color	yes	Supports hexadecimal values only.
Date	yes	Supply a text value that matches the date type in the prompt definition.
Time	yes	
File	yes	
Data source	yes	
Data library	yes	Supply this in the form **`libraryPath :: libref`**. For example, **`/Products/SAS Intelligence Platform/Samples/STP Samples(Library) :: stpsamp`** is a valid data library value.

Prompt Type	Supported	Notes
Data source item	yes	Supply this in the form **dataSourceLocation :: dataSourceType :: columnName :: columnLabel :: columnType**.
OLAP member	yes	
Ranges	no	
Multi-value prompts	no	

Accessing RESTful JSON and XML Web Service Endpoints

In SAS 9.3, all structured SAS BI Web Services expose a JSON, XML, and SOAP endpoint. This is true whether the Web service represents a single stored process or is a generated Web service. The URL for each of these endpoints depends on the location and type of Web service. For more information about accessing SOAP endpoints, see "Accessing SOAP Endpoints for Stored Processes and Generated Web Services" on page 28.

Accessing RESTful Web Service Endpoints

RESTful plain XML endpoints are available from the server resource **/SASBIWS/rest/**. When accessing endpoints for stored processes, append the SAS folder path for the stored process to the resource **/SASBIWS/rest/storedProcesses/** (for example, **/SASBIWS/rest/storedProcesses/stored/process/path**). Generated Web services have a unique name and contain either a single or multiple named stored processes. To access the RESTful plain XML endpoint for generated Web services, append the generated Web service name to the resource **/SASBIWS/rest/** (for example, **/SASBIWS/rest/generatedServiceName**). Because generated Web services can contain multiple stored processes, you need to identify which stored process that you want to invoke. You can do this in one of two ways:

- Use the stored process name as the payload root in your request message (assuming that the message is an HTTP POST). In this case, the input XML is very similar to the SOAP input XML.
- Append the stored process name (omitting any spaces) to the resource **/SASBIWS/rest/generatedServiceName** (for example, **/SASBIWS/rest/generatedServiceName/storedProcessName**).

For both generated and stored process Web services, you can tell SAS BI Web Services that you are interested only in a single specific aspect of the stored process output. You can do this by accessing the Web service at a specific URL. When you do this, the Web service returns only that specific output resource. Output parameters, output streams (also called data targets), and packages are all supported output resources.

To access a specific output parameter, append the resource **/parameters/*parameterName*** to the RESTful URL for your Web service, replacing *parameterName* with the name of the actual parameter. When you access the **/parameters/** resource of a RESTful Web service, the HTTP response body contains only the string value of that parameter and no additional XML.

To access a specific output stream, append the resource **/streams/*streamName*** to the RESTful URL for your Web service, replacing *streamName* with the name of the actual stream. When you access the **/streams/** resource of a RESTful Web service, the HTTP response is the exact output that your stored process sends to that stream and the HTTP response headers include an appropriate content type if it is available.

A stored process can produce an output package during execution that can contain any number of entries. You can access an individual entry within a package by appending the resource **/packages/*entryNum*** to the RESTful URL for your Web service, replacing *entryNum* with the index of the package entry. Entries in packages are not always named, so you must use the package entry index. The index starts at 0 for the first entry in the package.

You cannot access more than one output resource at a time by appending multiple **/parameters/**, **/streams/**, and **/packages/** resources. You can use only one at a time. You use this URL resource form whether invoking a service with an HTTP GET (when no input parameters or streams are required) or HTTP POST (when sending input prompt values or stream values). See "Invoking RESTful Web Services" on page 55 for examples that use various output parameter resources.

Accessing RESTful JSON Web Service Endpoints

JSON Web service endpoints are available from the server resource **/SASBIWS/json/**. When accessing endpoints for stored processes, append the SAS folder path for the stored process to the resource **/SASBIWS/json/storedProcesses/** (for example, **/SASBIWS/json/storedProcesses/stored/process/path**). Generated Web services have a unique name and contain either a single or multiple named stored processes. To access the JSON endpoint for a generated Web service, append the generated Web service name and the stored process name to the resource **/SASBIWS/json/** (for example, **/SASBIWS/json/generatedServiceName/storedProcessName**). JSON endpoints do not support output resource specifications, so the response is always JSON data.

Invoking RESTful Web Services

RESTful SAS BI Web Services are invoked by sending messages of a particular format to a Web service endpoint in the SAS middle tier. You send request messages to different endpoints depending on which response format you prefer. See "Accessing RESTful JSON and XML Web Service Endpoints" on page 54 for more information about how to determine where to send your Web service requests.

When you invoke a RESTful SAS BI Web service, the HTTP headers that you send are important. The Content-type HTTP header tells SAS BI Web Services what type of content you are sending when you perform an HTTP POST. If you are invoking a JSON endpoint, then your Content-type must be set as **application/x-www-form-urlencoded** and your content must be encoded using this format. If you are invoking an XML endpoint, then your Content-type must be set to **application/xml** and your content must be XML. The HTTP Accept header tells SAS BI Web Services what type of content your client can accept as output. The Accept header must be set to

`application/json` for JSON endpoints and can be set to **`application/xml`** for XML endpoints (this header is not required for XML endpoints). When retrieving binary output resources from an XML endpoint, you can follow normal HTTP procedures for other HTTP headers such as Content-length, Host, and HTTP method.

Table 4.4 *SOAP, RESTful XML, and RESTful JSON Usage*

	Stored process (no input required) /programs/timeAndDate	**Stored process (input required) /programs/addfloats**	**Generated Web service addfloatsWS contains stored process /programs/addfloats**
SOAP Usage			
SOAP endpoint	`http://host:port/SASBIWS/services/programs/timeAndDate`	`http://host:port/SASBIWS/services/programs/addfloats`	`http://host:port/SASBIWS/services/addfloatsWS`
WSDL location	`http://host:port/SASBIWS/services/programs/timeAndDate?wsdl`	`http://host:port/SASBIWS/services/programs/addfloats?wsdl` or `http://host:port/SASBIWS/services/programs/addfloats.wsdl`	`http://host:port/SASBIWS/services/addfloatsWS?wsdl` or `http://host:port/SASBIWS/services/addfloatsWS.wsdl`

	Stored process (no input required) /programs/timeAndDate	**Stored process (input required) /programs/addfloats**	**Generated Web service addfloatsWS contains stored process /programs/addfloats**
Sample SOAP request	POST http://*host*:*port*/SASBIWS/services/programs/timeAndDate HTTP/1.1 Content-Type: text/xml; charset=UTF-8 SOAPAction: "http://www.sas.com/xml/namespace/biwebservices/timeAndDate" Host: *host*:*port* <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:biw="http://www.sas.com/xml/namespace/biwebservices"> <soapenv:Header/> <soapenv:Body> <biw:timeAndDate> <biw:parameters /> </biw:timeAndDate> </soapenv:Body> </soapenv:Envelope>	POST http://*host*:*port*/SASBIWS/services/programs/addfloats HTTP/1.1 Content-Type: text/xml; charset=UTF-8 SOAPAction: "http://www.sas.com/xml/namespace/biwebservices/addfloats" Host: *host*:*port* <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:biw="http://www.sas.com/xml/namespace/biwebservices"> <soapenv:Header/> <soapenv:Body> <biw:addfloats> <biw:parameters> <biw:num1>2.3</biw:num1> <biw:num2>4.2</biw:num2> </biw:parameters> </biw:addfloats> </soapenv:Body> </soapenv:Envelope>	POST http://*host*:*port*/SASBIWS/services/addfloatsWS HTTP/1.1 Content-Type: text/xml; charset=UTF-8 SOAPAction: "http://tempuri.org/addfloatsWS/addfloats" Host: *host*:*port* <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:add="http://tempuri.org/addfloatsWS"> <soapenv:Header/> <soapenv:Body> <add:addfloats> <add:parameters> <add:num1>2.3</add:num1> <add:num2>4.2</add:num2> </add:parameters> </add:addfloats> </soapenv:Body> </soapenv:Envelope>

	Stored process (no input required) /programs/timeAndDate	**Stored process (input required) /programs/addfloats**	**Generated Web service addfloatsWS contains stored process /programs/addfloats**
Sample SOAP response	HTTP/1.1 200 OK SOAPAction: "" Accept: text/xml Content-Type: text/xml;charset=UTF-8 Transfer-Encoding: chunked <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/ soap/envelope/"> <soapenv:Body> <n:timeAndDateResponse xmlns:n="http://www.sas.com/xml/namespace/ biwebservices"> <n:timeAndDateResult> <n:Parameters> <n:time>13:24:20</n:time> <n:date>06Mar2011</n:date> </n:Parameters> </n:timeAndDateResult> </n:timeAndDateResponse> </soapenv:Body> </soapenv:Envelope>	HTTP/1.1 200 OK SOAPAction: "" Accept: text/xml Content-Type: text/xml;charset=UTF-8 Transfer-Encoding: chunked <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/ soap/envelope/"> <soapenv:Body> <n:addfloatsResponse xmlns:n="http://www.sas.com/xml/namespace/ biwebservices"> <n:addfloatsResult> <n:Parameters> <n:Sum>6.5</n:Sum> </n:Parameters> </n:addfloatsResult> </n:addfloatsResponse> </soapenv:Body> </soapenv:Envelope>	HTTP/1.1 200 OK SOAPAction: "" Accept: text/xml Content-Type: text/xml;charset=UTF-8 Transfer-Encoding: chunked <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/ soap/envelope/"> <soapenv:Body> <n:addfloatsResponse xmlns:n="http://tempuri.org/addfloatsWS"> <n:addfloatsResult> <n:Parameters> <n:Sum>6.5</n:Sum> </n:Parameters> </n:addfloatsResult> </n:addfloatsResponse> </soapenv:Body> </soapenv:Envelope>
RESTful XML Usage			
RESTful XML endpoint	**http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/timeAndDate**	**http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/addfloats**	**http://*host*:*port*/SASBIWS/rest/addfloatsWS/addfloats**

	Stored process (no input required) / programs/timeAndDate	**Stored process (input required) /programs/addfloats**	**Generated Web service addfloatsWS contains stored process /programs/addfloats**
Sample RESTful XML request	GET http://*host*:*port*/SASBIWS/rest/ storedProcesses/programs/timeAndDate HTTP/1.1 Accept: application/xml Host: *host*:*port*	POST http://*host*:*port*/SASBIWS/rest/ storedProcesses/programs/addfloats HTTP/1.1 Accept: application/xml Content-Type: application/xml User-Agent: Jakarta Commons-HttpClient/3.1 Host: *host*:*port* <addfloats> <parameters> <num1>2.3</num1> <num2>4.2</num2> </parameters> </addfloats>	POST http://*host*:*port*/SASBIWS/rest/ addfloatsWS/addfloats HTTP/1.1 Accept: application/xml Content-Type: application/xml Host: *host*:*port* <addfloats xmlns="http://tempuri.org/ addfloatsWS"> <parameters> <num1>2.3</num1> <num2>4.2</num2> </parameters> </addfloats>
Sample RESTful XML response	HTTP/1.1 200 OK Content-Type: text/xml Transfer-Encoding: chunked <timeAndDateResponse> <timeAndDateResult> <Parameters> <time>13:24:20</time> <date>06Mar2011</date> </Parameters> </timeAndDateResult> </timeAndDateResponse>	HTTP/1.1 200 OK Content-Type: text/xml Transfer-Encoding: chunked <addfloatsResponse> <addfloatsResult> <Parameters> <Sum>6.5</Sum> </Parameters> </addfloatsResult> </addfloatsResponse>	HTTP/1.1 200 OK Content-Type: text/xml Transfer-Encoding: chunked <n:addfloatsResponse xmlns:n="http://tempuri.org/addfloatsWS"> <n:addfloatsResult> <n:Parameters> <n:Sum>6.5</n:Sum> </n:Parameters> </n:addfloatsResult> </n:addfloatsResponse>
RESTful JSON Usage			
RESTful JSON endpoint	**http://*host*:*port*/SASBIWS/json/storedProcesses/programs/timeAndDate**	**http://*host*:*port*/SASBIWS/json/storedProcesses/programs/addfloats**	**http://*host*:*port*/SASBIWS/json/addfloatsWS/addfloats**

	Stored process (no input required) / programs/timeAndDate	Stored process (input required) /programs/addfloats	Generated Web service addfloatsWS contains stored process /programs/addfloats
Sample RESTful JSON request	GET http://*host*:*port*/SASBIWS/json/storedProcesses/programs/timeAndDate HTTP/1.1 Accept: application/json Host: *host*:*port*	POST http://*host*:*port*/SASBIWS/json/storedProcesses/programs/addfloats HTTP/1.1 Accept: application/json Content-Type: application/x-www-form-urlencoded Host: *host*:*port* num1=2.3&num2=4.2	POST http://*host*:*port*/SASBIWS/json/addfloatsWS/addfloats HTTP/1.1 Accept: application/json Content-Type: application/x-www-form-urlencoded Host: *host*:*port* num1=2.3&num2=4.2
Sample RESTful JSON response	HTTP/1.1 200 OK Content-Type: application/json Transfer-Encoding: chunked {"outputParameters":{"date":"06Mar2011", "time":"13:24:20"}}	HTTP/1.1 200 OK Content-Type: application/json Transfer-Encoding: chunked {"outputParameters":{"Sum":"6.5"}}	HTTP/1.1 200 OK Content-Type: application/json Transfer-Encoding: chunked {"outputParameters":{"Sum":"6.5"}}

Table 4.5 *RESTful Output Resource Access*

	Stored process (no input required) /programs/outputTest	**Stored process (input required) /programs/ioTest**	**Generated Web service outputTestWS contains stored process /programs/outputTest**
Original RESTful URL	**http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/outputTest**	**http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/ioTest**	**http://*host*:*port*/SASBIWS/rest/storedProcesses/outputTestWS/outputTest**
Accessing an output parameter named **result** that returns a value **success**			
Endpoint	**http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/outputTest/parameters/result**	**http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/ioTest/parameters/result**	**http://*host*:*port*/SASBIWS/rest/outputTestWS/outputTest/parameters/result**

Sample request	`GET http://host:port/SASBIWS/rest/storedProcesses/programs/outputTest/parameters/result`
	`HTTP/1.1`
	`Host: host:port`
		`POST http://host:port/SASBIWS/rest/storedProcesses/programs/outputTest/parameters/result`
		`HTTP/1.1`
		`Host: host:port`
		
		`<outputTest>`
		`<sampleInput>`
		`Foo`
		`</sampleInput>`
		`</outputTest>`
			`GET http://host:port/SASBIWS/rest/outputTestWS/outputTest/parameters/result`
			`HTTP/1.1`
			`Host: host:port`

Sample response	`HTTP/1.1 200 OK`
	`Content-Type: text/plain;charset=ISO-8859-1`
	`Content-Length: 7`
	
	`success`		

Accessing a data target named **strOut** that returns XML			

	Stored process (no input required) / programs/outputTest	Stored process (input required) /programs/ioTest	Generated Web service outputTestWS contains stored process /programs/outputTest
Endpoint	http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/outputTest/streams/strOut	http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/ioTest/streams/strOut	http://*host*:*port*/SASBIWS/rest/outputTestWS/outputTest/streams/strOut
Sample response	HTTP/1.1 200 OK Content-Type: text/xml;charset=utf-8 <streamOutput>contents</streamOutput>		
Accessing a data target named **pdfOut** that returns a PDF (binary data)			
Endpoint	http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/outputTest/streams/pdfOut	http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/ioTest/streams/pdfOut	http://*host*:*port*/SASBIWS/rest/outputTestWS/outputTest/streams/pdfOut
Sample response	HTTP/1.1 200 OK Content-Type: application/pdf %PDF-1.3 %Ãåòåë§ó ÐÄÆ 4 0 obj << /Length 5 0 R /Filter /FlateDecode >> stream x›ÝŽ7…ïû)x…		
Accessing the third entry in package output which is a PDF file			
Endpoint	http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/outputTest/packages/2	http://*host*:*port*/SASBIWS/rest/storedProcesses/programs/ioTest/packages/2	http://*host*:*port*/SASBIWS/rest/outputTestWS/outputTest/packages/2

	Stored process (no input required) `/programs/outputTest`	Stored process (input required) `/programs/ioTest`	Generated Web service outputTestWS contains stored process `/programs/outputTest`

Sample response	`HTTP/1.1 200 OK`
	`Content-Type: application/pdf`
	
	`%PDF-1.3`
	`%Äåòåë§ó ÐÄÆ`
	`4 0 obj`
	`<< /Length 5 0 R /Filter /FlateDecode >>`
	`stream`
	`x›ÝŽ7…ïû)x…`		

Index

W

X