THE POWER TO KNOW®

SAS® Solutions Services 5.1 Customization Guide

Second Edition

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2010. *SAS® Solutions Services 5.1: Customization Guide, Second Edition*. Cary, NC: SAS Institute Inc.

SAS® Solutions Services 5.1: Customization Guide, Second Edition

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

1st electronic book, November 2010

Contents

Chapter 1

About the Customization Guide

What's in This Book

This book contains information about customizing SAS Solutions Services 5.1 and the solutions that use SAS Solutions Services:

- SAS Financial Management 5.1
- SAS Human Capital Management 5.1
- SAS Strategy Management 5.1

It includes the following topics:

- creating custom stored processes
- customizing SAS Financial Management:
  - writing SAS code that accesses the SAS Financial Management application programming interface (API)
  - writing macros in Microsoft Excel that interact with SAS Financial Management objects
  - adding custom cell actions to Microsoft Excel
  - customizing a workflow
- enabling and customizing auditing in SAS Strategy Management
- configuring Secure Sockets Layer (SSL)

Note: This book no longer contains information about alerts or directives, because those features are now part of the SAS Intelligence Platform or the Web Infrastructure Platform. For information about creating row-level security filters, see the *SAS Human Capital Management: Administrator's Guide*. For information about customizing themes, see the *SAS Intelligence Platform: Web Application Administration Guide*.

Required Skills

To use the SAS Financial Management Java API, you must be familiar with both SAS and Java programming. To use the SAS Financial Management Add-In API for Microsoft Excel, you must have an understanding of Microsoft Excel and Microsoft Visual Basic for Applications (VBA).

Documentation Conventions

This book uses the following documentation conventions to identify paths in the solutions configuration:

Path	Refers to	Example
`!sasroot`	Path to the SAS root directory	`C:\Program Files\SAS\SASFoundation\9.2`
SAS-config-dir	Path to the SAS configuration directory	`C:\SAS\Config`
MySQL-install-dir	Path to the MySQL installation directory	`C:\mysql`

Note:

- Your site might have a different configuration directory name or a different level number.
- If your configuration is the result of a migration from the previous release of SAS Solutions Services, the `SASApp` directory might be called `SASMain` instead (for example, `C:\SAS\Config\Lev1\SASMain` rather than `C:\SAS\Config\Lev1\SASApp`). Please make the appropriate substitutions as you read this book.

Additional Documentation

For additional information, see the appropriate versions of the following books. They are available at `http://support.sas.com/documentation/`. From the Products Index, select **SAS Financial Management**, **SAS Strategy Management**, or **SAS Human Capital Management** to see a list of documents that are available for that product.

- *SAS Solutions Services: System Administration Guide*
- *SAS Solutions Services: Data Administration Guide*
- *SAS Solutions Services: Data Model Reference*

- *SAS Performance Management Solutions: Migration Guide*
- The user's guides for SAS Financial Management, SAS Human Capital Management, and SAS Strategy Management
- The administrator's guide for SAS Human Capital Management

In addition, the SAS Intelligence Platform administration guides are available at `support.sas.com/92administration`.

For information about configuring your Web application server, go to `http://support.sas.com/resources/thirdpartysupport/v92/`.

Chapter 2
Working with Stored Processes

Overview: Stored Processes and SAS Solutions Services

A stored process is a SAS program that is stored centrally on a server and is executed via a client application, which then can receive and process the results. Stored processes can access a SAS data source or external file and can create new data sets, files, or other data targets.

A stored process can be defined with parameters, with or without global default values. At run time, client applications can supply parameter values when they invoke the stored process.

Here are some common uses for stored processes, within the context of the solutions:

- **Creating charts.** Parameters are used to select elements such as time, analysis, and product category.
- **Generating quick reports,** such as profit and loss reports. Parameters are used to select product, customer, region, and so on.
- **Validating data.** For example, a stored process can be used to find extraordinary values such as too many returns or extraordinarily high sales in an unlikely area.
- **Verifying data.** One such example is the ETL Job Status report, which is included with the product as a standard report.

- **Loading data.** For example, you might create a stored process to import new data for a forthcoming period, from the SAS Data Integration Studio jobs that load metric tables. Another example is the standard Import Users and Groups stored process, which stores user and group information in the SASSDM database.

Writing a Custom Stored Process for the Solutions

Creating the Stored Process

Stored processes are a standard way to extend and otherwise customize the solutions. For detailed information about writing SAS code for use in stored processes, see the *SAS Stored Processes: Developer's Guide*.

Stored processes that you create should reside on the data tier of your installation. One good location is in the ***`SAS-config-dir`***`\Lev1\SASApp\SASEnvironment\`***`solution-name`***`\SASCode` directory. In the `SASCode` directory, you can create a subdirectory for your code (for example, `C:\SAS\Config\Lev1\SASApp\SASEnvironment\FinancialManagement\SASCode\UserDefined`).

Several macros are available for use in stored processes that are part of SAS Solutions Services and the solutions. For more information, see "Macros for Use with SAS Solutions Services" on page 9.

For examples of stored processes that use these macros, see the stored processes that are included with SAS Solutions Services and the solutions:

- **SAS Solutions Services:** `!sasroot\soltnsdata\sasstp`
- **SAS Financial Management:** `!sasroot\finance\sasstp`
- **SAS Strategy Management:** `!sasroot\scorecard\sasstp`
- **SAS Human Capital Management:** ***`SAS-config-dir`***`\`*`Lev1`*`\AppData\SASHumanCapitalManagement5.1\StoredProcesses`

Making the Stored Process Available

Overview

To register a stored process, log on to SAS Management Console as an administrator and add the stored process to the appropriate folder.

Users can execute a stored process from Document Manager. They can also create or edit a My Favorites portlet or a Collection portlet and add a link to a stored process. With SAS Human Capital Management, users can execute a stored process from the workspace.

With SAS Financial Management, a stored process can be used in a custom cell action or as a workflow customization.

For more information about creating and registering a stored process, see the *SAS Stored Processes: Developer's Guide* (available at `http://support.sas.com/documentation/`).

For information about security for stored processes, see the *SAS Intelligence Platform: Security Administration Guide* (available at `http://support.sas.com/92administration`).

Creating Package Results for SAS Financial Management Reports

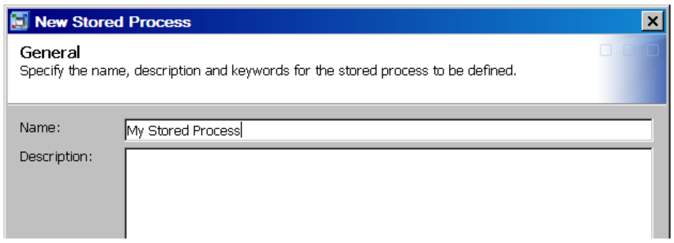
When you register a stored process, you can specify what type of output that stored process can produce. You can specify **Stream**, **Package**, both output types, or neither output type.

The simplest type of output, or result type, is none. The client receives no output from the stored process. The stored process is still able to create or update data sets, external files, or other objects, but this output remains on the server. Streaming output delivers a data stream, such as an HTML page or XML document, to the client. It is supported only on the stored process server.

Package output can be stored in a permanent location, such as a WebDAV repository. For a SAS Financial Management report, your stored process code should begin with a call to %RPTINIT, and your stored process should generate package results. Typically, the results are stored in the user's personal repository. Each user can run the same stored process and generate results that depend on the parameters the user selects and depending on security that is in place.

The following steps define a stored process in a shared folder, with output to a personal repository. (In the screen displays, some dialog boxes are truncated.)

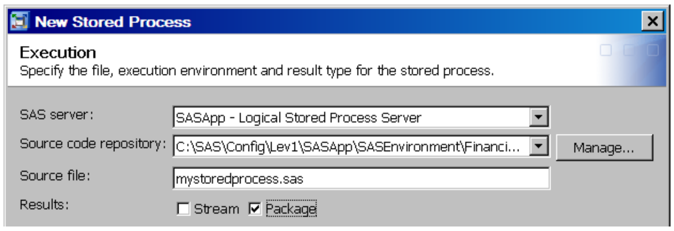
1. Log on to SAS Management Console as an administrator.
2. On the **Folders** tab, right-click a shared folder and select **New Stored Process**. One possible location is the **`/Products/SAS Financial Management`** folder, where you might create a subfolder to hold stored processes for your site.
3. On the General page of the New Stored Process wizard, give the stored process a name.



General
Specify the name, description and keywords for the stored process to be defined.

Name: My Stored Process
Description:

4. On the Execution page of the wizard, select the stored process server and define the path and the name for the stored process. Select the **Package** check box.



Execution
Specify the file, execution environment and result type for the stored process.

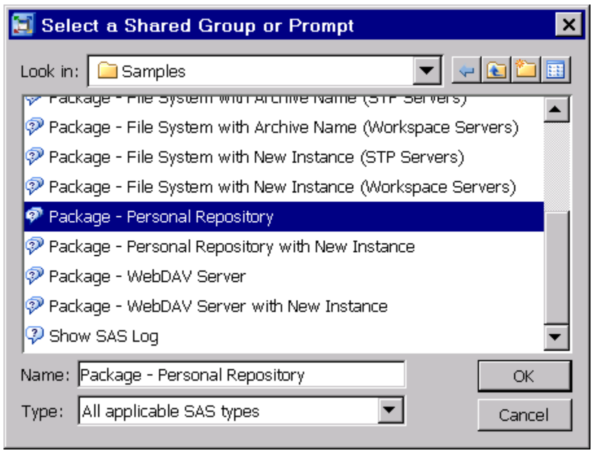
SAS server: SASApp - Logical Stored Process Server
Source code repository: C:\SAS\Config\Lev1\SASApp\SASEnvironment\Financi...
Source file: mystoredprocess.sas
Results: ☐ Stream ☑ Package

5. On the Parameters page, define any input parameters that are required by the stored process.

   For the results options, select WebDAV output, as follows:

   a. Click **Add Shared**.

   b. In the Select a Shared Group or Prompt dialog box, navigate to **`SAS Folders \Products\Intelligence Platform\Samples`**. To store the output in the user's personal folder in the WebDAV repository, select **Package - Personal Repository**.



Select a Shared Group or Prompt

Look in: Samples

- Package - File System with Archive Name (STP Servers)
- Package - File System with Archive Name (Workspace Servers)
- Package - File System with New Instance (STP Servers)
- Package - File System with New Instance (Workspace Servers)
- Package - Personal Repository
- Package - Personal Repository with New Instance
- Package - WebDAV Server
- Package - WebDAV Server with New Instance
- Show SAS Log

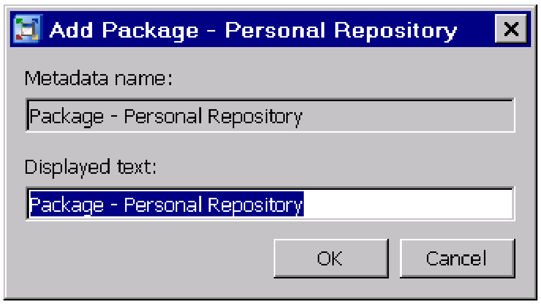
Name: Package - Personal Repository

Type: All applicable SAS types

Note: Do not select **Package - Personal Repository with New Instance**.

c. Click **OK**.

d. In the Add Package dialog box, click **OK**.



Add Package - Personal Repository

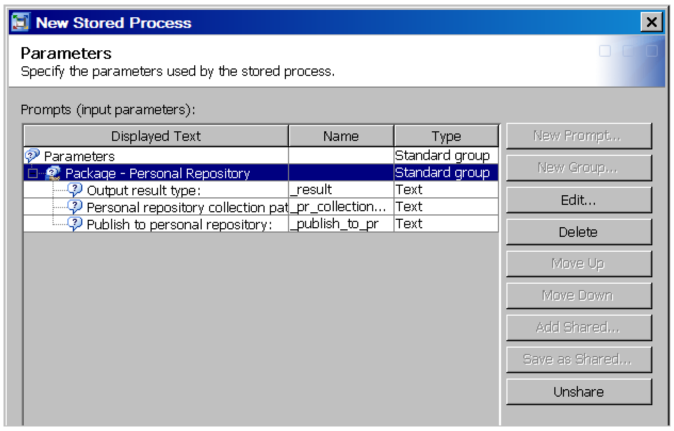
Metadata name: Package - Personal Repository

Displayed text: Package - Personal Repository

6. Select **Package - Personal Repository** and click **Unshare** to unshare the prompts so that you can modify them.

   SAS Management Console displays a warning message and asks whether you want to continue. Click **Yes**.

7. Click the plus sign next to the **Package - Personal Repository** prompt to expand the options.



New Stored Process

Parameters

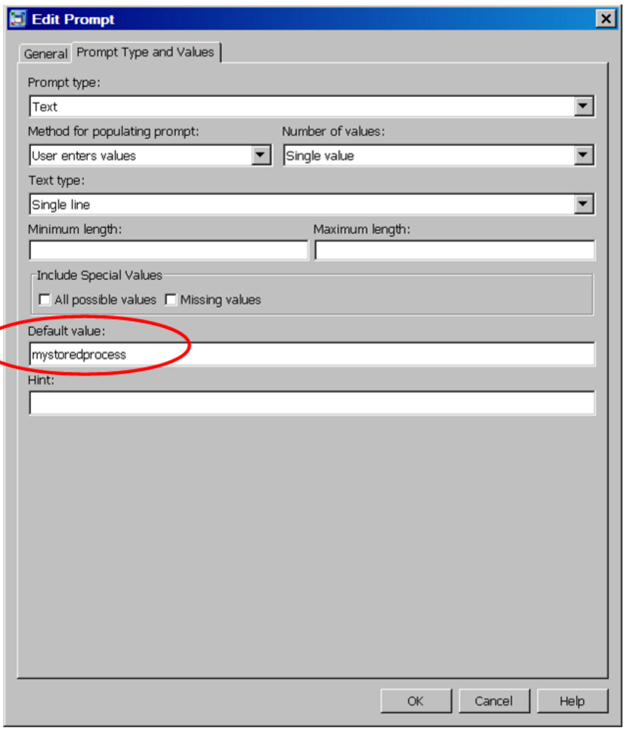
Specify the parameters used by the stored process.

Prompts (input parameters):

Displayed Text	Name	Type
Parameters		Standard group
Package - Personal Repository		Standard group
Output result type:	_result	Text
Personal repository collection pat	_pr_collection...	Text
Publish to personal repository:	_publish_to_pr	Text

8. Select the **Personal repository collection path** prompt and click **Edit**.

9. On the **Prompt Type and Value** tab, type a value into the **Default value** text box.


Edit Prompt

General | Prompt Type and Values

Prompt type: Text

Method for populating prompt: User enters values

Number of values: Single value

Text type: Single line

Minimum length:

Maximum length:

Include Special Values: All possible values, Missing values

Default value: mystoredprocess

Hint:

OK Cancel Help

This value becomes the directory name for storing the output in the WebDAV repository.

10. Keep the defaults for the other results prompts.
11. Click **Next**.
12. On the Data page of the wizard, enter any source or target data sources. Then click **Finish**.

When a user executes a stored process, the results are immediately available in the Web browser. The results are also stored in the WebDAV repository; for example, if the personal repository is selected, results are stored in the **`sasdav/Users/`*****`user-name`*****`/PR/MyResults/`*****`default-value`*** folder. The user can access the stored results (for example, via a WebDAV navigator portlet in the portal).

For an example of a stored process that was created with package output, see the stored processes in the **`/Products/SAS Financial Management/5.1 Standard Reports`** folder. For more information about SAS Content Server and WebDAV content, see the *SAS Intelligence Platform: Web Application Administration Guide*. For more information about WebDAV navigator portlets, see the online Help for the portal.

Macros for Use with SAS Solutions Services

Overview

The following macros are available to use in stored processes that work with SAS Solutions Services or the solutions:

Table 2.1 Macros for Use with SAS Solutions Services

Macro	Description
%BLDVIEW	Creates a view of records that have been filtered for the current user. Available only if you have SAS Human Capital Management.
%GETLSTNR	Locates a designated Event Broker Service from a SAS Metadata Server repository.
%MTRCLOAD	Updates metric data in the SAS Solutions Data Mart.
%RPTINIT	Extends and replaces the standard %STPBEGIN macro. Use this macro for a stored process that is used in SAS Financial Management to create package output.
%SENDEVNT	Sends an event to an event listener that is running in the SAS Solutions middle tier.

The %BLDVIEW Macro

Overview

Creates a view of records to which row-level security has been applied.

Note: This macro is available for use only in SAS Human Capital Management.

Syntax

%BLDVIEW (
  INTABLE=,
    OUTTABLE=
    [, OUTTYPE=TABLE|VIEW]
    [, **LIBREF**=]
    [, **DEFAULT_LIBREF**=_INTBL_]
    [, **DEFAULT_DATAPATH**=]
    [, **DEFAULT_EVENTSERVER**=]
    [, **TABLE_REPOSITORY**=]
    [, **DOALLVARS**=]
    [, **EVENTNAME**=]
    [, **DEFAULT_EVENTNAME**=SAS.Solutions.Service.Requested]
)

INTABLE
: Name of the table that the secured view should be based on. This value should be a one-level name, without any libref. The table must be registered in the metadata repository.

OUTTABLE
: Name of the output table or view. This value should include a libref.

OUTTYPE
: Output type: a VIEW or a data TABLE. If unspecified, this parameter defaults to VIEW.

LIBREF
: A standard SAS libref that applies to the INTABLE.

DEFAULT_LIBREF
: A standard SAS libref that applies to the INTABLE if the LIBREF parameter is empty.

DEFAULT_DATAPATH
: The path to the input table. (Use if there is no libref assigned to the input table.)

DEFAULT_EVENTSERVER
: The default event server to be used by the %SENDEVNT macro. If you do not set this parameter, %BLDVIEW calls the %GETLSTNR macro, which sets the event server name.

TABLE_REPOSITORY
: Name of the metadata repository for the input table.

DOALLVARS
: Flag that determines whether secured columns are visible. If DOALLVARS is missing or has a value of **`Y`**, then %BLDVIEW returns all columns in the table. If the user does not have access to a column, then a missing value is returned for that column (a blank for character data or a period for numeric data).

  If DOALLVARS has a value other than **`Y`**, then %BLDVIEW does not return any columns that are not accessible by the user. If the stored process explicitly references one of those columns, then the stored process server returns an error.

  Note: You can achieve the same effect by setting the BLDVIEW_ALLVARS global variable.

  DOALLVARS and BLDVIEW_ALLVARS apply only to column-level security. If the user does not have access to an entire table, then %BLDVIEW does not return any columns

EVENTNAME
: The event to use. The default is **`SAS.Solutions.Service.Requested`**. To override the default, give the EVENTNAME parameter a value or set a global macro variable named BLDVIEW_EVENTNAME.

DEFAULT_EVENTNAME
: The default event to use. (See the description of the EVENTNAME parameter.)

Details

Use this macro to ensure that your report includes only those records that the user is authorized to view. For information about row-level security, see the *SAS Human Capital Management: Administrator's Guide*.

Example

This example is from the ABSWKDAY stored process, one of the SAS Human Capital Management standard reports:

```
%let dsn=abshmast;
%bldview(intable=&dsn, outtable=&dsn, libref=hcmdata,
   default_libref=hcmdata, table_repository=Foundation);
```

The %GETLSTNR Macro

Overview

Locates a designated Event Broker Service in a metadata repository.

Syntax

%GETLSTNR (

[**METASERVER**=]
[, **METAPORT**=]
[, **METAPROTOCOL**=BRIDGE|]
[, **METAUSER**=]
[, **METAPASS**=]
[, **METAREPOSITORY**=]
[, **SOFTWARECOMPONENTNAME**=Remote Services|]
[, **SOFTWARETREENAME**=Event]
[, **MEMBERNAME**=Event Broker Service]
[, **TRANSFORMATIONSOURCENAME**=HTTP_Transport]

)

METASERVER, METAPORT, METAPROTOCOL, METAUSER, METAPASS, METAREPOSITORY

Values for the metadata repository and server, if they were not specified in SAS Options or if they differ from the Options value.

SOFTWARECOMPONENTNAME, SOFTWARETREENAME, MEMBERNAME, TRANSFORMATIONSOURCENAME

Values for the event listener. These parameters need to be set only if the event listener is stored in a different place in the metadata repository from the default location.

Details

The macro creates a global macro variable named EVENTSERVER that contains the name of the HTTP event server, in the form *server-name:port*. The value of the EVENTSERVER variable can be passed to the EVENTSERVER parameter of the %SENDEVNT macro.

Example

If the metadata-related parameters are already set as SAS options, then the call to this macro is as follows:

```
%getlstnr();
```

Note: Avoid defining a different Event Broker Service.

The %MTRCLOAD Macro

Overview

Updates metric data in the SAS Solutions Data Mart.

Syntax

%MTRCLOAD (

INPUT=
, **DIMFLDS**=

    , **STATFLDS**=
    [, **SOURCE_SYSTEM_CD**=ETL]
    [, **TABLE_DESC**=]
    [,**_STRINGDELIMITER**=|++|]
)

INPUT
: Specifies the two-level table that contains metric information.

  For more information about this table, see "Details" on page 13.

DIMFLDS
: One or more sets of dimension field values, separated by the `|++|` separator. For each dimension, provide the following values, also separated by `|++|`:

  `field-name|++|dimension-code|++|dimension-type-code|++|hierarchy-code`

STATFLDS
: A list of value fields, separated by the `|++|` separator. At least one value is required.

TABLE_DESC
: The table description. If this parameter is omitted, the macro creates it from the dimension list.

SOURCE_SYSTEM_CD
: Source system code, such as **ETL**, **FM**, or **HCM**. The default is **ETL**.

_STRINGDELIMITER
: The delimiter used for the DIMFLDS and STATFLDS parameters. The default is `|++|`.

Details

This macro checks to see whether there is already a metric table of the same structure in the database that is referenced by the SDMMET libref. If not, it adds the table to the database (with a name of METRICTABLE*X*) and registers it in the metadata repository.

The table that is designated in the INPUT parameter should include the following columns:

Column Name	**Description**
MEASURE_NM	Contains a valid measure name derived from the MEASURE table in the Cross Industry Detail Data Store.
DIRECTIVE_TXT	Specifies a directive to use to drill to details on this measure.
MODIFIED_DT	Contains the date on which the record was created.
TIME_PERIOD_ID	Contains a valid time period code from the TIME_PERIOD table in the Cross Industry Detail Data Store.
value columns	Contains one or more columns that are supplied as values to the **STATFLDS** parameter in the %MTRCLOAD macro. Examples: VALUE, TARGET_VALUE.
dimension columns	Contains one or more columns that are supplied as values to the **DIMFLDS** parameter in the %MTRCLOAD macro. Example: INTERNAL_ORG_ID.

Note: When you create a metric table, do not use dimension codes that are also reserved words in MySQL. For a list of these reserved words, see **http://dev.mysql.com/doc/refman/5.0/en/reserved-words.html**.

Example

This example is from an ETL job:

```
%let dimflds = %nrquote(TIME_PERIOD_ID|++|TIME|++|TIME|++|TIME_MR|++|
   INTORG_HR_ID|++|ORG|++|INTORG|++|INTORG_HR);
%let statflds = %nrquote(value|++|mean_val|++|median_val|++|p10_val|++|
   p25_val|++|p75_val|++|p90_val);
%let table_desc = %nrquote(HCM Metric Table);

%mtrcload(input=&syslast,
          dimflds=&dimflds,
          statflds=&statflds,
          table_desc=&table_desc);
```

The %RPTINIT Macro

Overview

This macro extends and replaces the standard %STPBEGIN macro when it is used in a stored process for SAS Solutions Services reporting. It is used in SAS Financial Management stored processes.

Syntax

%RPTINIT (

[**STYLE**=]
[, **DEVICE**=]

)

STYLE
Name of the ODS style to define for output by setting the _ODSSTYLE variable; defaults to **sasweb**.

DEVICE
SAS/GRAPH DEVICE option for generating graphical output. Defaults to **gif**.

Details

%RPTINIT performs the following tasks:

- sets the image path to obtain output assets from the appropriate WebDAV path
- calls the %STPBEGIN macro

Example

```
/* Simple Stored Process for SAS Solutions */
%rptinit;
goptions hpos=45
         vpos=25
         ftext=
         colors=(blue red green);
proc gtestit;
run;
%stpend;
```

The %SENDEVNT Macro

Overview

Sends an event to an event listener that is running in the middle tier.

Syntax

%SENDEVNT (
EVNTNAME=,
EVENTSERVER=
[, **FILEREF**=]
[, **FILENAME**=]
[, **RESULT_URL**=]
[, **RESULT_FILEREF**=]
[, **SENTBY**=]
[, **PROPS**=, **VALS**=]
[, **RESPONSE**=NONE]
[, **PRIORITY**=]
[, **VALS_DELIMITERS**=%str( )]
[, **ETL_GROUP_NM**=]
[, **SENDEVNT_RC_VAR**=SENDEVNT_RC]
)

EVNTNAME
(Required) Name of the event to send.

EVENTSERVER
(Required) Name of the event server to which the event is sent. Use %GETLSTNR to obtain the appropriate event listener.

FILEREF
SAS fileref that points to an XML file that is sent with the event. Some events read an XML file that contains additional information about what the event should do. For these events, you can supply either a SAS fileref or filename that points to this XML file. If values for both the FILEREF and FILENAME parameters are specified, then the filename takes precedence.

FILENAME
Name of an XML file that is sent with the event. (See FILEREF, above.)

RESULT_URL
URL to hold the response XML. This value takes priority over the RESULT_FILEREF, if both parameters are specified.

RESULT_FILEREF
A fileref that points to a file to hold the response XML.

SENTBY
Optional sender information that is sent with the event. Its value is usually a name or user ID.

PROPS, VALS
Additional properties (separated by spaces or commas) to send with the event. There is a one-to-one match between the values defined in PROPS and the values defined in VALS.

RESPONSE
Indicates whether the event includes a response; the default is NONE.

PRIORITY
: The Java priority level for the event, with values that range between 1 and 10. The default, which typically does not need to be changed, is **10**.

VALS_DELIMITERS
: Specifies the delimiter character that is used to separate the values in the VALS field. Normally this value should not be changed; the default is the pound character (**#**).

ETL_GROUP_NM
: Used only by ETL jobs.

SENDEVNT_RC_VAR
: The name of a macro variable to receive the return code (RC) of the DATA step that publishes the event. A nonzero value indicates an error. Proper usage is to define the receiving macro outside of this macro, because the %SENDEVNT macro does not define it as either global or local.

Example

```
%sendevnt(evntname=&eventName,
          priority=&priority,
          eventserver=&EVENTSERVER,
          sentby=&sentby,
          props=%bquote(&props),
          vals=%bquote(&prop_vals));
```

Troubleshooting Stored Processes

If a stored process does not run correctly, view the stored process log file that is located in the ***SAS-config-dir*\Lev1\SASApp\StoredProcessServer\Logs** directory on the machine where the stored process server is running. For information about configuring the log files for the stored process server, see “Administering Logging for SAS Servers” in the *SAS Intelligence Platform: System Administration Guide*.

For standard reports that are a part of SAS Financial Management, you can configure an additional log file to provide more information. For details, see “Viewing and Configuring the Log Files” in the *SAS Solutions Services: System Administration Guide*.

Chapter 3

The SAS Financial Management Java API

Using the SAS Financial Management Java API

About the SAS Financial Management Java API

The SAS Financial Management application programming interface (API) is a set of Java classes that are available to SAS code for accessing SAS Financial Management data. Among other tasks, the API can be used to:

- customize a workflow (for example, to validate user input in a data entry form).
- perform a query against a model and store the results in a data set that can be accessed by other code.
- create a custom audit report.
- run a custom query against SAS Financial Management data.

Most of the classes apply only to financial planning. However, the AuditHistory and Metadata classes can also be used for operational planning.

Note: For information about the terminology in this chapter, see the *SAS Financial Management User's Guide*.

Instantiating an Object

The API uses the Javaobj interface, a mechanism that is similar to the Java Native Interface (JNI) for instantiating Java classes and accessing their methods and fields. The DATA step that includes a Javaobj declaration must include the following option:

```
/picklist='finance/finance.txt'
```

The picklist option is necessary so that the Javaobj can access the necessary JAR files.

To instantiate an object, you declare a Javaobj object using the following syntax:

```
dcl javaobj object-name (classname, constructor-arguments);
```

Parameters are as follows:

object-name
: The handle to the Java object that is returned. You use this handle to access the object's methods.

classname
: A string that contains the fully qualified name of the Java class that you are instantiating, such as **`com/sas/solutions/finance/api/Form`**.

constructor-arguments
Any arguments that are required by the constructor.

Authenticating the User

Authentication Using the METADATA_PASSID Function

In order to access SAS Financial Management data, the user must be authenticated on the middle tier. The recommended approach is to call the object's setEnvironment method and then call the METADATA_PASSID function in the DATA step. For example:

```
data _null_ /picklist='finance/finance.txt';
dcl javaobj j("com/sas/solutions/finance/api/AuditHistory");
j.ExceptionDescribe(1);
j.callVoidMethod("setEnvironment", "default");
call METADATA_PASSID("j", "");
```

This function creates a one-time user/password combination and authenticates the user on the middle tier.

In a stored process, the METADATA_PASSID function has access to the user ID and password. In an interactive SAS session, the user is asked for the user ID and password to be used for authentication on the middle tier. If the authentication fails, check the stored process log or the SAS log.

Note: Document Manager no longer passes the session context to a stored process. Consequently, you cannot use a constructor such as **`Model(`*`entityKey`*`)`** in a stored process that is called from Document Manager. (You can use such a constructor in a stored process that is called from a workflow.)

Authentication Via User ID and Password

An alternative approach is to pass the user ID, password, and environment (also called domain) to the constructor. For example:

```
data _null_ /picklist='finance/finance.txt';
   dcl javaobj oAdmin("com/sas/solutions/finance/api/AdminQuery",
      "sasdemo", "DemoDemo1", "default");
```

We recommend encoding or encrypting the password, rather than using a plain-text password. For more information, see the *SAS Intelligence Platform: Security Administration Guide*.

Authentication from a Workflow

For a stored process that is called from a workflow, you must get the session context from the FM_SP_SECKEY variable and pass it to the constructor for a Java class. For more information about using a stored process in a workflow, see Chapter 4, “Customizing a Workflow,” on page 57.

Specifying the SAS Solutions Environment

The environment argument should be the same value a user would specify when logging on to the middle tier from Microsoft Excel. If you did not already do so at installation time, add the JREOPTIONS option to the sasV9_usermods.cfg file located in each SAS application server context directory that you use. The JREOPTIONS should point to a network-accessible copy of the EnvironmentFactory.xml file. By default, this file is made available as follows:

```
-JREOPTIONS=(-Denv.factory.location=
   http://hostname:port/SASConfig/EnvironmentFactory.xml)
```

Note: Line break inserted for readability. *hostname* is the name of the host machine for the middle tier, and *port* is the port number of the managed server to which you deployed SAS Solutions Services. Typically, this port number is 7201.

If the environment files are published to an HTTP server, the URL would resemble the following: **`http://myhttpserver:`*`port`*`/EnvironmentFactory.xml`**. See "About the SAS Environment Files" on page 20.

About the SAS Environment Files

The SAS environment file (sas-environment.xml) and the SAS Solutions environment file (EnvironmentFactory.xml) contain information about the resources that are necessary to run the client applications (for example, the addresses of services used by the applications).

You can use the original configuration of these files to validate an installation. However, after validation, the SAS environment files should be published to an HTTP server, as described in the "Installing the Client Applications" chapter of the *SAS Solutions Services: System Administration Guide*.

Note: .

Calling an Object's Methods

With a handle to the Java object, you can call its methods. This code calls the Form object's getState method, which returns a String value:

```
oForm.callStringMethod("getState", state);
```

In this example, OFORM represents the handle to the Form object. The call statement matches the method's return type (for example, CALLSTRINGMETHOD, CALLDOUBLEMETHOD, CALLINTMETHOD, CALLBOOLEANMETHOD, or CALLVOIDMETHOD).

The first parameter is always the method name, and the last parameter always contains the return value (if any). The remaining parameters are the parameters that the Java method requires. In the example above, the getState method has no parameters.

Deleting the Javaobj

To avoid memory leaks, all instantiations of a Javaobj should be terminated by a call to the DELETE method. Call the object's logout method before deleting the object, as in this example:

```
dcl javaobj oAudit("com/sas/solutions/finance/api/AuditHistory");
oAudit.ExceptionDescribe(1);
oAudit.callVoidMethod("setEnvironment", "default");
call METADATA_PASSID("oAudit", "");
...
oAudit.callVoidMethod ("logout");
oAudit.delete();
```

Retrieving Error Messages

Many methods return a Boolean value indicating whether the action was successful. Because SAS does not have a true Boolean type, the return code is either **`0`** (failure) or **`1`** (success). When the return code is `0`, the getErrorMessage method can be used to retrieve the pertinent error message, as in this example:

```
if rc le 0 then do;
    oForm.callStringMethod("getErrorMessage", msg);
end;
```

Configuring a Log File

In addition to calling getErrorMessage, you can generate a more detailed log by creating a log4j.properties file. For more information, see “Configure a Log File for the SAS Financial Management Reports” in the *SAS Solutions Services: System Administration Guide*.

Handling Exceptions

The EXCEPTIONCHECK method can be used to determine whether an exception has been thrown. The EXCEPTIONCLEAR method clears any existing exceptions. Here is an example:

```
/* clear any existing stored exception */
oModel.ExceptionClear();
oModel.callvoidmethod("getModelHierarchies", "Default_Model", "FMSData",
   "TstHierarchies");
/* check to see if an exception has been thrown */
rc = oModel.ExceptionCheck(exception);
if (exception) then
    put 'Exception occurred,Please check the log for more information';
oModel.callVoidMethod("logout");
```

Note: The EXCEPTIONCHECK method cannot be used to detect exceptions that are thrown when constructing an object.

Summary of Classes

Table 3.1 *Summary of Classes*

Class	Description
AdminQuery	Contains methods for running queries on the Base Facts data of SAS Financial Management. Applies only to financial planning.
AuditHistory	Contains methods for running queries on the AuditHistory data of SAS Financial Management.
BaseApi	Serves as the base class for the SAS Financial Management Java API.
BaseQuery	Contains methods for running queries. This class is extended by the AuditHistory, AdminQuery, and CycleQuery classes.
CycleQuery	Contains methods for extracting facts from a cycle. Applies only to financial planning.

Class	Description
Form	Contains methods for running queries on the properties of a planning form from SAS Financial Management. Applies only to financial planning.
Metadata	Contains methods for retrieving metadata about SAS Financial Management.
Model	Contains methods for retrieving information about SAS Financial Management models and for running queries against a model. Applies only to financial planning.

The AdminQuery Class

Overview

The AdminQuery class contains methods for running queries on the Base Facts data. It extends the com.sas.solutions.finance.api.BaseQuery class. The AdminQuery class applies only to financial planning.

For an example of using the AdminQuery class, see the Facts stored process (**`!sasroot\finance\sasstp\facts.sas`**). This stored process lists data records that are associated with a specified financial model. You can limit a Facts report to a time period or an analysis member, and in several other ways.

Method Summary

Table 3.2 *AdminQuery Class Method Summary*

Method	Description
`AdminQuery()`	Constructor. Throws: **`java.lang.Exception`**
boolean **`executeQuery`**()	Executes the query using the filters and any other parameters that have been previously specified. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: **`java.lang.Exception`**
java.lang.String **`getQueryColNames`** (java.lang.String queryType)	Gets the list of column names for a specific query and model. This method can be executed before running a query. However, you must set the model to be used in the query before calling getQueryColNames. Parameters: queryType: type of query to execute. For a list of possible values, see the definition of the setQueryType method. Returns: column names, separated by commas

Method	**Description**
java.lang.String **`getQueryColNames`** (java.lang.String queryType, java.lang.String modelCode)	Gets the list of column names for a specific query and model. This method can be executed before running a query. Parameters: • queryType: the type of query to execute. For a list of possible values, see the definition of the setQueryType method. • modelCode: the model code to be used in the query. Returns: column names, separated by commas
java.lang.String **`getQueryColNamesWithSeparator`** (java.lang.String queryType, java.lang.String modelCode, java.lang.String separator)	Gets the list of column names for a specific query and model. This method can be executed before running the query. Parameters: • queryType: the type of query to execute. For a list of possible values, see the definition of the setQueryType method. • modelCode: the model code to be used in the query. • separator: the text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
java.lang.String **`getQuerySASNames`** (java.lang.String queryType)	Gets a list of column names (in SAS valid name format) for a specific query and model. This method can be executed before running a query. However, you must first set the model to be used in the query. Parameters: queryType: the type of query to execute. For a list of possible values, see the definition of the setQueryType method. Returns: column names, separated by commas
java.lang.String **`getQuerySASNames`** (java.lang.String queryType, java.lang.String modelCode)	Gets a list of column names (in SAS valid name format) for a specific query and model. This method can be executed before running the query. It returns column names, separated by commas. Parameters: • queryType: the type of query to execute. For a list of possible values, see the definition of the setQueryType method. • modelCode: the model code to be used in the query. Returns: column names, separated by commas

Method	Description
java.lang.String **`getQuerySASNamesWithSeparator`** (java.lang.String queryType, java.lang.String modelCode, java.lang.String separator)	Gets the list of column names (in SAS valid name format) for a specific query and model. This method can be executed before running the query. Parameters: • queryType: the type of query to execute. For a list of possible values, see the definition of the setQueryType method. • modelCode: the model code to be used in the query. • separator: the text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
java.lang.String **`getReportingCurrency()`**	Gets the reporting currency member code. Returns: If the reporting currency has been set (via the setReportingCurrency method), then that member code is returned. Otherwise, the default reporting currency is returned. Throws: java.lang.Exception
boolean **`setDimFilter`** (java.lang.String code, java.lang.String value)	Sets a filter on a dimension; to filter on multiple values, call the method for each value. Parameters: • code: the dimension code • value: the member code to be used as the filter value Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean **`setDimFilterID`** (java.lang.String dimID, java.lang.String memID)	Sets a filter on a dimension; to filter on multiple values, call the method for each value. Parameters: • dimID: the dimension ID • memID: the member reference ID to be used in the filter Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean **`setDimTypeFilter`** (java.lang.String code, java.lang.String value)	Sets a filter on a dimension type; to filter on multiple values, call the method for each value. Parameters: • code: the dimension type code. • value: the member code to be used in the filter. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean **`setFactsParms`** (java.lang.String otid, java.lang.String oid, java.lang.String ssid)	Deprecated. Use setParms instead.

Method	Description
boolean **`setModel`** (java.lang.String name)	Sets the model to be used in a query. Parameters: name: the model name. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean **`setModelCode`** (java.lang.String code)	Sets the model to be used in a query. Parameters: code: the model code. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean **`setModelID`** (java.lang.String ID)	Sets the model to be used in a query. Parameters: ID: the model ID. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception
boolean **`setParms`** (java.lang.String otid, java.lang.String oid, java.lang.String convert)	Sets the parameters for the query. Parameters: • otid: the object type ID, which must be a string containing one of these values: **`adjustmentsequence`**, **`attachment`**, **`cashinfusiontransaction`**, **`compositeresult`**, **`cycle`**, **`dataload`**, **`differentialwritedown`**, **`disposaltransaction`**, **`dividendtransaction`**, **`equityassignment`**, **`form`**, **`formset`**, **`formtemplate`**, **`holding`**, **`holdingmethodaccounts`**, **`lineitem`**, **`manualadjustment`**, **`measureexport`**, **`othercpolineitem`**, **`othercpotransaction`**, **`ownershipchangetransaction`**, **`period`**, **`pocconsolidationmethod`**, **`pocholdingfact`**, **`purchaseadjustment`**, **`purchasedifferential`**, **`purchasetransaction`**, **`result`**, **`rule`**, **`standaloneparent`**, or **`balsheet_reversal`**. • oid: the object ID. Typically, this value is an empty string (**`""`**). • convert: the currency conversion flag. A value of **`Y`** specifies that currency values should be converted from their functional currencies to a presentation currency. A value of **`N`** specifies that conversion should not take place. Returns: **`true`** if the parameter values are valid; otherwise, **`false`**

Method	Description
boolean **setQueryType** (java.lang.String queryType)	Sets the type of query to execute. Parameters: queryType: the type of query to execute, which must be a string containing one of these values: **ELIMINATIONS**, **NONLEAF**, **DATAENTRY**, **TRIALBALANCE**, **INTERCOMPANY**, **NONINTERCOMPANY**, **RULESFACTS**, **RULE**, **MANUALADJUSTMENTS**, **FACTS**, **OWNERSHIP**, **ICACCOUNTS**, **FACTSR**, **DETAILS**, **OWNERSHIPTRANSACTIONS**, or **OWNERSHIPMETHODS** Returns: **true** if the query type value is valid; otherwise, **false**
boolean **setReportingCurrency** (java.lang.String code)	Sets the currency to be used for reporting values. Parameters: code: a currency code, such as **EUR**. Returns: **true** if the action succeeded; otherwise, **false** Throws: java.lang.Exception
boolean **setRule** (java.lang.String name)	Sets the rule by name (required only by the RULE query). Parameters: name: the name of a rule. Returns: **true** if the action succeeded; otherwise, **false** Throws: java.lang.Exception
boolean **setRuleID** (java.lang.String id)	Sets the rule by ID (required only by the RULE query). Parameters: id: the ID of a rule. Returns: **true** if the action succeeded; otherwise, **false** Throws: java.lang.Exception
boolean **setVCubeID** (java.lang.String ID)	Sets the model using the ID of a virtual cube (vcube). Parameters: ID: the ID of a virtual cube. Returns: **true** if the action succeeded; otherwise, **false** Throws: java.lang.Exception

Methods inherited from class com.sas.solutions.finance.api.BaseQuery: getColumnName, getColumnSASName, getColumnType, getMaxRowsMessage, getNumberOfColumns, getNumericValue, getQueryColNames, getQueryColNamesWithSeparator, getQueryRecordsNumber, getQuerySASNames, getQuerySASNamesWithSeparator, getRecord, getValue, and setMaxRows.

Methods inherited from class com.sas.solutions.finance.api.BaseApi: authenticate, buildExceptionMessageString, getErrorMessage, getMessage, setEnvironment, setLocale, and trim.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

The AuditHistory Class

Overview

The AuditHistory class contains methods for running queries on AuditHistory data from SAS Financial Management. It extends the com.sas.solutions.finance.api.BaseQuery class.

For an example of using the AuditHistory class, see the Audit stored process (**`!sasroot\finance\sasstp\audit.sas`**). This stored process extracts audit and history data that is filtered by three optional parameters: a user, an action type, and a date range.

Method Summary

Table 3.3 *AuditHistory Class Method Summary*

Method	**Description**
boolean **`executeQuery()`**	Executes the query using the filters and any other parameter previously specified. The query generates records with the following columns (all are character data): USERNAME, ACTION_TYPE_ID, TIMESTAMP_TS, OBJECT_CLASS_ID, OBJECT_ID, SOLUTION_ID, TRANSACTION_ID, AUDIT_ID, PROPERTY_NM, OLD_VALUE, and NEW_VALUE. You can call the getValue method to retrieve these values. Returns: **`true`** if the action succeeded; otherwise, **`false`**
java.lang.String **`getQueryColNames()`**	Gets the list of column names that were returned by the AuditHistory query. This method can be called only after running the query. Returns: column names, separated by commas
java.lang.String **`getQueryColNames`** (java.lang.String separator)	Gets the list of column names that were returned by the AuditHistory query. This method can be called only after running the query. Parameters: • separator: the text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
void **`setDateFormat`** (java.lang.String format)	Sets the desired format for passing the dates when calling setDateRange.
boolean **`setDateRange`** (java.lang.String from, java.lang.String to)	Sets a date range. Dates are expected to be in the format *mm/dd/yyyy* unless they are otherwise specified by a call to setDateFormat.

Method	Description
boolean **`setFilter`** (java.lang.String name, java.lang.String value)	Sets a filter on a column. To filter on multiple values, call the method for each value. Parameters: • name: the column name. For a list of valid column names, see the description of the executeQuery method. • value: the value for the filter. For example, if you wanted to see audit records for the sasdemo, you would use a call like this: **`oAuditHistory.callBooleanMethod("setFilter", "username", "sasdemo", rc);`**

Methods inherited from class com.sas.solutions.finance.api.BaseQuery: getColumnName, getColumnSASName, getColumnType, getMaxRowsMessage, getNumberOfColumns, getNumericValue, getQueryColNames, getQueryColNamesWithSeparator, getQueryRecordsNumber, getQuerySASNames, getQuerySASNamesWithSeparator, getRecord, getValue, and setMaxRows.

Methods inherited from class com.sas.solutions.finance.api.BaseApi: authenticate, buildExceptionMessageString, getErrorMessage, getMessage, setEnvironment, setLocale, and trim.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

The BaseApi Class

Overview

The BaseApi class is extended by the BaseQuery, Form, Metadata, and Model classes.

Note: BaseApi methods should be called only by one of its subclasses.

Method Summary

Table 3.4 *BaseApi Class Method Summary*

Method	Description
`BaseApi()`	Constructor. Throws: java.lang.Exception

Method	Description
boolean **`authenticate`** (java.lang.String entityKey)	Authenticates the user on the middle tier. This method can be called only from a stored process that is part of a workflow. Parameters: • entityKey: the security key that contains the session context information for the current user. See "Authentication from a Workflow" on page 19. Returns: **`true`** if the authentication succeeded; otherwise, **`false`** Throws: java.lang.Exception
java.lang.String **`getErrorMessage()`**	Gets the localized error message from the last action. If the setLocale method was called, the specified locale is used. Otherwise, the system default locale is used. Returns: a localized message string

java.lang.String **`getMessage`** (java.lang.String message)	Gets the localized message that corresponds to a message code. If the setLocale method was called, the specified locale is used. Otherwise, the system default locale is used.
	Parameters:
	• message: the identifier for a localized message string.
	For a list of valid message codes, see the Resources_*language-code*.properties files in the sas.solutions.finance.api.nls.jar file.
	To locate the correct JAR file, open the **`!sasroot\picklist\finance\finance.txt`** file and find the following name: **`sas.solutions.finance.api`**. Make a note of the version that corresponds to this name. The JAR file is in the ***`SAS-install-dir`*** **`\SASVersionedJarRepository\`*****`version`*** directory.
	Returns: a localized message string
	Example:
	`j.callStringMethod("getMessage",`
	`   "Api.QueryReturnedNoFacts.txt", msg);`
	` call symput('msg', msg);`

void **`logout()`**	Logs the user off the middle tier and releases any resources allocated for the user. *Note:* The login method is no longer a public method.

Method	Description
boolean **`setLocale`** (java.lang.String l)	Sets the locale. (The default locale is the system default locale.) Parameters: • l: a locale that is specified as *language-code_country-code*, such as **`en_US`** or **`es_SP`**. The *language-code* is a valid ISO language code in the form of a lowercase, two-character string, and the *country-code* is a valid ISO country code in the form of an uppercase, two-character string. Returns: **`true`** if the action succeeded; otherwise, **`false`**
java.lang.String **`trim`** (java.lang.String s)	Returns the value passed in, with trailing blanks removed.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

The BaseQuery Class

Overview

The BaseQuery class is extended by the AdminQuery, AuditHistory, and CycleQuery classes. It contains methods for retrieving the results of a query.

Note: The methods of the BaseQuery class should be called only from one of its subclasses.

Method Summary

Table 3.5 *BaseQuery Class Method Summary*

Method	Description
`BaseQuery()`	Constructor. Throws: java.lang.Exception
java.lang.String **`getColumnName`** (double n)	Gets the name of the *n*th column.
java.lang.String **`getColumnSASName`** (double n)	Gets the SAS name of the *n*th column.
java.lang.String **`getColumnType`** (double n)	Gets the column type (numeric or character) of the *n*th column.
java.lang.String **`getMaxRowsMessage()`**	Gets the maximum number of rows that a query can return. The default is 10,000 rows. If the query returns fewer than this maximum number of rows, the getMaxRowsMessage method returns an empty string. Otherwise, it returns a localized message with this string: **`Showing the first n rows`**, where *n* is the maximum number of rows that were requested.

Method	**Description**
int **`getNumberOfColumns()`**	Gets the number of columns returned by the query. This method can be executed only after a query has run.
double **`getNumericValue`** (double n, double m)	Gets the numeric value of the *n*th column of the *m*th record.
java.lang.String **`getQueryColNames()`**	Gets the list of column names that were returned by a query. This method can be executed only after the query has been run. Returns: column names, separated by commas
java.lang.String **`getQueryColNamesWithSeparator`** (java.lang.String separator)	Gets the list of column names that were returned by a query. This method can be executed only after the query has been run. Parameters: • separator: the text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
int **`getQueryRecordsNumber()`**	Gets the number of records (facts) that were returned by the query.
java.lang.String **`getQuerySASNames()`**	Gets a list of column names (in SAS valid name format) that were returned by a query. This method can be executed only after the query has been run. Returns: column names, separated by commas
java.lang.String **`getQuerySASNamesWithSeparator`** (java.lang.String separator)	Gets the list of column names (in SAS valid name format) that were returned by a query. This method can be executed only after the query has been run. Parameters: • separator: the text (such as a comma) to be used to separate column names in the list that is returned. Returns: column names, separated by the separator text
java.lang.String **`getRecord`** (double n)	Gets the *n*th record. Parameters: • n: the index of a record in the query results. Returns: record values, separated by commas
java.lang.String **`getRecord`** (double n, java.lang.String separator)	Gets the *n*th record. Parameters: • n: the index of a record in the query results. • separator: the text to be used as a separator, such as a comma. Returns: record values, separated by the separator text
java.lang.String **`getValue`** (double n, double m)	Gets the value of the *n*th column of the *m*th record.

Method	Description
boolean **`setMaxRows`** (java.lang.String s)	Sets the maximum number of records (or facts) a query can return. The default is 10,000 rows. Parameters: • s: the maximum number of rows. A value of `0` specifies no limit. Returns: **`true`** if the action succeeded; otherwise, **`false`**

Methods inherited from class com.sas.solutions.finance.api.BaseApi: authenticate, buildExceptionMessageString, getErrorMessage, getMessage, setEnvironment, setLocale, and trim.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

The CycleQuery Class

Overview

The CycleQuery class contains methods for extracting facts from a cycle. It extends the com.sas.solutions.finance.api.BaseQuery class.

The CycleQuery class is similar to the AdminQuery class. For an example of its use, see the ETL Facts stored process (**`!sasroot\finance\sasstp\etlfacts.sas`**). This stored process lists data records that have been loaded from SAS Data Integration Studio to a specified time period and analysis member within a specified financial cycle, and (optionally) a specified organization.

Method Summary

Table 3.6 *CycleQuery Class Method Summary*

Method	Description
`CycleQuery()`	Constructor.
boolean **`getETLFacts()`**	Gets the ETL facts for the specified cycle and filters. Returns: **`true`** if the action succeeded; otherwise, **`false`**
java.lang.String **`getQueryColNames`** (java.lang.String cycleName, java.lang.String separator)	Gets the list of column names for the query. This method can be called before running the query. Parameters: • cycleName: the name of a cycle • separator: the text, such as a comma, to be used as a separator Returns: a list of column names, separated by the separator text.

Method	**Description**
java.lang.String **`getQueryColNames`** (java.lang.String cycleID, java.lang.String separator)	Gets the list of column names for the query. This method can be called before running the query. Parameters: • cycleID: the ID of a cycle. • separator: the text, such as a comma, to be used as a separator. Returns: a list of column names, separated by the separator text
java.lang.String **`getQuerySASNames`** (java.lang.String cycleName, java.lang.String separator)	Gets a list of column names (in SAS valid name format) for a specific query and cycle. This method can be executed before running the query. Parameters: • cycleName: the cycle name. • separator: the text, such as a comma, to be used as a separator. Returns: a list of column names, separated by the separator text
java.lang.String **`getQuerySASNamesByID`** (java.lang.String cycleID, java.lang.String separator)	Gets a list of column names (in SAS valid name format) for a specific query and cycle. This method can be executed before running the query (after setting the cycle to be used in the query). Parameters: • cycleID: the cycle ID. • separator: the text, such as a comma, to be used as a separator. Returns: a list of column names, separated by the separator text
boolean **`setCycleByID`** (java.lang.String ID)	Sets the cycle for the query by ID. Returns: **`true`** if the action succeeded; otherwise, **`false`**
boolean **`setCycleByName`** (java.lang.String name)	Sets the cycle for the query by name. Returns: **`true`** if the action succeeded; otherwise, **`false`**
boolean **`setDimTypeFilter`** (java.lang.String code, java.lang.String value)	Sets a filter on a dimension type; to filter on multiple values, call the method for each value. Parameters: • code: the dimension type code. • value: the member code to be used in the filter. Returns: **`true`** if the action succeeded; otherwise, **`false`** Throws: java.lang.Exception

Method	Description
boolean **`setParms`** (java.lang.String otid, java.lang.String oid)	Sets the parameters for the query. Parameters: • otid: the object type ID. Possible values are **`adjustmentsequence`**, **`attachment`**, **`cashinfusiontransaction`**, **`compositeresult`**, **`cycle`**, **`dataload`**, **`differentialwritedown`**, **`disposaltransaction`**, **`dividendtransaction`**, **`equityassignment`**, **`form`**, **`formset`**, **`formtemplate`**, **`holding`**, **`holdingmethodaccounts`**, **`lineitem`**, **`manualadjustment`**, **`measureexport`**, **`othercpolineitem`**, **`othercpotransaction`**, **`ownershipchangetransaction`**, **`period`**, **`pocconsolidationmethod`**, **`pocholdingfact`**, **`purchaseadjustment`**, **`purchasedifferential`**, **`purchasetransaction`**, **`result`**, **`rule`**, **`standaloneparent`**, or **`balsheet_reversal`**. • oid: the object ID. Returns: **`true`** if the parameter values are valid; otherwise, **`false`**

Methods inherited from class com.sas.solutions.finance.api.BaseQuery: getColumnName, getColumnSASName, getColumnType, getMaxRowsMessage, getNumberOfColumns, getNumericValue, getQueryColNames, getQueryColNamesWithSeparator, getQueryRecordsNumber, getQuerySASNames, getQuerySASNamesWithSeparator, getRecord, getValue, and setMaxRows.

Methods inherited from class com.sas.solutions.finance.api.BaseApi: authenticate, buildExceptionMessageString, getErrorMessage, getMessage, setEnvironment, setLocale, and trim.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

The Form Class

Overview

The Form class contains methods for running queries on the properties of a planning form from SAS Financial Management. It extends the com.sas.solutions.finance.api.BaseApi class. This class applies only to financial planning.

For an example of using the Form class, see “Data Validation Example” on page 63.

Method Summary

Table 3.7 *Form Class Method Summary*

Method	Description
`Form` (int formId, java.lang.String entityKey)	Constructor. This constructor can be used only in a stored process that is used in a workflow. Both the form ID and the security key (entityKey) are available as environment variables that are set by the workflow. Throws: java.lang.Exception
`Form` (java.lang.String sFormId, java.lang.String userId, java.lang.String password, java.lang.String environment)	Constructor. Throws: java.lang.Exception
java.lang.String **`getAuthors`** (java.lang.String delimiter)	Returns the user IDs of all authors of a specified form, separated by the *delimiter* text if more than one author was found. Parameters: • delimiter: the text (such as a space or semi-colon) that is used to separate author names in the return string.
java.lang.String **`getDescription()`**	Returns the form description.
java.lang.String **`getDueDate()`**	Returns the due date of the form.
java.lang.String **`getFormSetDescription()`**	Returns the description of the form set to which the form belongs.
int **`getFormSetId()`**	Returns the ID of the form set to which the form belongs.
java.lang.String **`getFormSetName()`**	Returns the name of the form set to which the form belongs.
java.lang.String **`getId()`**	Returns the form ID as a string.
java.lang.String **`getInfo()`**	Returns a formatted string with key information about the form.
java.lang.String **`getName()`**	Returns the form name.
java.lang.String **`getPlanningAdministrators`** (java.lang.String delimiter)	Returns a list of users with the role of Finance Process Administrator. Parameters: • delimiter: the text that is used to separate names in the return string.
java.lang.String **`getReviewers`** (java.lang.String delimiter)	Returns all reviewers of a specified form. The reviewers are separated by the *delimiter* text if more than one reviewer was found. Parameters: • delimiter: the text that is used to separate names in the return string.

Method	Description
java.lang.String **getState()**	Returns the form state.
java.lang.String **getTargetDimensionCode()**	Returns the code of the target dimension of the form set to which the form belongs.
java.lang.String **getTargetDimensionDescription()**	Returns the description of the target dimension of the form set to which the form belongs.
java.lang.String **getTargetDimensionName()**	Returns the name of the target dimension of the form set to which the form belongs.
java.lang.String **getTargetMemberCode()**	Returns the target member code of the form.
java.lang.String **getTargetMemberDescription()**	Returns the description of the target member of the form.
int **getTargetMemberId()**	Returns the target member ID of the form.
java.lang.String **getTargetMemberName()**	Returns the name of the target member of the form.
boolean **isLocked()**	Returns **true** if the form is locked by some process.

Methods inherited from class com.sas.solutions.finance.api.BaseApi: authenticate, buildExceptionMessageString, getErrorMessage, getMessage, setEnvironment, setLocale, and trim.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

The Metadata Class

Overview

The Metadataclass contains methods for looking up SAS Financial Management metadata. It extends the com.sas.solutions.finance.api.BaseApi class.

For an example of using the Metadata class, see Chapter 5, “Creating a Custom Cell Action,” on page 69.

Method Summary

Table 3.8 *Metadata Class Method Summary*

Method	**Description**
`Metadata ()`	Constructor. Throws: java.lang.Exception
java.lang.String **`getDimensionCode`** (java.lang.String dimID)	Gets the dimension code. Parameters: • dimID: the dimension ID. Returns: the dimension code that corresponds to the dimension ID
java.lang.String **`getMemberCode`** (java.lang.String dimID, java.lang.String memID)	Gets the member code. Parameters: • dimID: the dimension ID. • memID: the member ID. Returns: the member code that corresponds to the specified dimension ID and member ID.

Methods inherited from class com.sas.solutions.finance.api.BaseApi: authenticate, buildExceptionMessageString, getErrorMessage, getMessage, setEnvironment, setLocale, and trim.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

The Model Class

Overview

The Model class contains methods for retrieving information about SAS Financial Management models and for running queries against a model. It extends the com.sas.solutions.finance.api.BaseApi class. This class applies only to financial planning.

Note: For performing queries against a model, use the %FMQUERY macro, which supports both MDX and non-MDX queries. See "Performing Queries with the %FMQUERY Macro" on page 44.

The Model class is not designed for interactive use. It is intended to be used by administrators or power users to export data to an external data set, for viewing (for example, in Microsoft Excel) or for use by another application or process. The security that is applied to a query is the security for the user who is running the query. (Keep that in mind if you make the query results available to other users.)

In all query methods in the Model class, the SAS library that is used for the result data set must be registered in the metadata. Before executing the call, the SAS program is responsible for ensuring that *resultDataSetName* does not exist (unless you want to write over an existing data set). The SAS program is also responsible for cleanup of these tables after the call. If *resultDataSetName* does exist before the query and the query fails (for example, because the user lacks sufficient privileges), the data set is not deleted.

Method Summary

Table 3.9 *Model Class Method Summary*

Method	Description
`Model`	Constructor.
`Model` (java.lang.String storedProcessEntityKey)	Constructor. This constructor can be used only in a stored process that is part of a workflow. Parameters: • storedProcessEntityKey: the security key that is passed from the workflow. Throws: java.lang.Exception
`Model` (java.lang.String userId, java.lang.String password, java.lang.String environment)	Constructor. Parameters: • userId: the user ID for logging on to the middle tier. • password: the password for this user. • environment: the environment for logging on to the middle tier. Throws: java.lang.Exception
void **`executeQuery`** (java.lang.String sasLibraryName, java.lang.String modelCode, java.lang.String queryDataSetName, java.lang.String resultDataSetName, double filterOptions)	Performs a query against a model. Do not use this method. Instead, use the %FMQUERY macro. See “Performing Queries with the %FMQUERY Macro” on page 44.

Method	**Description**
int **`generateFormulaFacts`** (java.lang.String cycleName, java.lang.String formSetName)	Computes and stores all driver formula output values for crossings in the selected form set. This method corresponds to the **Run driver formulas** option of SAS Financial Management Studio, which is used to make sure that the stored output values of all driver formulas are current. Parameters: • cycleName: the name of the cycle to use. • formSetName: the name of the form set to use. Returns an integer containing the status code: • `0`: SUCCESS • `1`: OBJECT NOT FOUND • `2`: FORM SET IS LOCKED • `3`: GENERIC ERROR Before calling this method for an imported form set, save the form set template (in Microsoft Excel). Otherwise, the method returns an error. This note also applies when you select **Run driver formulas** in SAS Financial Management Studio.
void **`getAllModels`** (java.lang.String sasLibraryName, java.lang.String resultDataSetName)	Retrieves the available models for the default locale and creates a result set with these columns: MODEL_CD, MODEL_NAME, and MODEL_DESCRIPTION. The default locale refers to the system default locale on the machine where the SAS session is running. Parameters: • sasLibraryName: the libref for the SAS library that will hold the result set. • resultDataSetName: the name of the result set. Throws: FinanceClientException
void **`getAllModels`** (java.lang.String language, java.lang.String country, java.lang.String variant, java.lang.String sasLibraryName, java.lang.String resultDataSetName)	Retrieves the available models for a specified locale and creates a result set with these columns: MODEL_CD, MODEL_NAME, and MODEL_DESCRIPTION. Parameters: • language: a valid ISO language code in the form of a lowercase, two-character string, such as **`en`** or **`es`**. • country: a valid ISO country code in the form of an uppercase, two-character string, such as **`US`** or **`SP`**, or an empty string. • variant: this parameter is not used and should be set to an empty string (**`""`**). • sasLibraryName: the libref for the SAS library that will hold the result set. • resultDataSetName: the name of the result set. Throws: FinanceClientException

Method	**Description**
double **`getCellValue`** (java.lang.String resultCode, java.lang.String[] dimensionCodes, java.lang.String[] memberCodes)	Gets the value of a crossing. Parameters: • resultCode: the code of the results model. • dimensionCodes: the list of dimension codes that define the crossing. • memberCodes: a matching list of member codes that define the crossing. Returns: the value of the specified crossing Throws: java.lang.Exception

void **`getModelHierarchies`** (java.lang.String modelCode, java.lang.String sasLibraryName, java.lang.String resultDataSetName)	Retrieves a model's hierarchy for the default locale and creates a result set with these columns: DIMENSION_TYPE_CD, DIMENSION_CD, DIMENSION_NAME, DIMENSION_DESCRIPTION, HIERARCHY_CD, HIERARCHY_NAME, and HIERARCHY_DESCRIPTION.
	Parameters:
	• modelCode: the code that identifies the model for which you want to retrieve the hierarchy.
	• sasLibraryName: the libref for the SAS library that will hold the result set.
	• resultDataSetName: the name for the result set.
	Throws: FinanceClientException
	Example:
	`oModel.callvoidmethod("getModelHierarchies",`
	`   "Default_Model", "FMSData", "TstHierarchies");`

void **`getModelHierarchies`** (java.lang.String modelCode, java.lang.String language, java.lang.String country, java.lang.String variant, java.lang.String sasLibraryName, java.lang.String resultDataSetName)	Retrieves a model's hierarchy for a specified locale and creates a result set with these columns: DIMENSION_TYPE_CD, DIMENSION_CD, DIMENSION_NAME, DIMENSION_DESCRIPTION, HIERARCHY_CD, HIERARCHY_NAME, and HIERARCHY_DESCRIPTION. Parameters: • modelCode: the code that identifies the model for which you want to retrieve the hierarchy. • language: a valid ISO language code in the form of a lowercase, two-character string, such as **`en`** or **`es`**. • country: a valid ISO country code in the form of an uppercase, two-character string, such as **`US`** or **`SP`**, or an empty string. • variant: this parameter is not used and should be set to an empty string (**`""`**). • sasLibraryName: the libref for the SAS library that will hold the result set. • resultDataSetName: the name for the result set. Throws: FinanceClientException

Method	Description
public java.lang.String **`getModelMemberPropeties`** (java.lang.String modelCode, java.lang.String filePrefix, java.lang.String delim)	Creates a file that contains the following properties for the specified models: dimension_type_cd, hierarchy_cd, member_cd, property_cd, property_name, and property_value. The properties are separated by the string specified in the delim parameter. (Instead of calling this method, you can use the %GETMODELPROPERTIES macro. See "Calling the %GETMODELPROPERTIES Macro" on page 54.) The properties are written to a temporary file with a name of *filePrefix* + a random number. The method returns the full path to this file, and the calling routine is expected to read the file contents and store the data elsewhere. (For an example, see the code for the %FMQUERY macro (in the **`!sasroot\finance\sasmacro`** directory.) Parameters: • modelCode: the identifier for the results model to be used • filePrefix: a name for the output file • delim: the delimiter to be used to parse the input (dimension type codes, property codes, and member codes) as well as to separate values in the output file. Returns: the full path to the output file. Throws: FinanceClientException

Method	Description
public String **`getModelMemberPropeties`** (String modelCode, String filePrefix, String language, String country, String variant, String dimTypeCodes, String propertyCodes, String memberCodes, String delim)	Creates a file that contains the following properties for the specified models: dimension_type_cd, hierarchy_cd, member_cd, property_cd, property_name, and property_value. The properties are separated by the string specified in the delim parameter. (Instead of calling this method, you can use the %GETMODELPROPERTIES macro. See “Calling the %GETMODELPROPERTIES Macro” on page 54.) The properties are written to a temporary file with a name of *filePrefix* + a random number. The method returns the full path to this file, and the calling routine is expected to read the file contents and store the data elsewhere. (For an example, see the fmquery.sas code.) Parameters: • modelCode: the identifier for the results model to be used • filePrefix: a name for the output file • language, country, variant: language and country are used to determine the locale for the query; variant is not used and should be set to an empty string (`""`). • dimTypeCodes: a delimited list of unquoted dimension type codes to use in the query. If this parameter is NULL, all dimension type codes are used. • propertyCodes: a delimited list of unquoted property codes to use in the query. If this parameter is NULL, all properties are used. • memberCodes: a delimited list of unquoted member codes to use in the query. If this parameter is NULL, all members are used. • delim: the delimiter to be used to parse the input (dimension type codes, property codes, and member codes) as well as to separate values in the output file. Returns: the full path to the output file. Throws: FinanceClientException

Method	**Description**

void **`getModelMembers`** (java.lang.String modelCode, java.lang.String sasLibraryName, java.lang.String resultDataSetName)	Retrieves a model's members for the default locale and creates a result set with these columns: DIMENSION_TYPE_CD, HIERARCHY_CD, MEMBER_CD, MEMBER_NAME, MEMBER_DESCRIPTION, HIERARCHY_LEVEL, HIERARCHY_ORDER, PARENT_CD, and IS_LEAF. (A value of **`1`** for IS_LEAF signifies that this member is a leaf. Otherwise, the value is **`0`**.)
	
	Parameters:
	• modelCode: the code that identifies the model for which you want to retrieve the members.
	• sasLibraryName: the libref for the SAS library that will hold the result set.
	• resultDataSetName: the name for the result set.
	
	Throws: FinanceClientException
	
	Example:
	`oModel.callvoidmethod("getModelMembers",`
	`   "Default_Model", "FMSData", "TstModelMembers");`

void **`getModelMembers`** (java.lang.String modelCode, java.lang.String language, java.lang.String country, java.lang.String variant, java.lang.String sasLibraryName, java.lang.String resultDataSetName)	Retrieves a model's members for the specified locale and creates a result set with these columns: DIMENSION_TYPE_CD, HIERARCHY_CD, MEMBER_CD, MEMBER_NAME, MEMBER_DESCRIPTION, HIERARCHY_LEVEL, HIERARCHY_ORDER, PARENT_CD, and IS_LEAF (a value of **`1`** signifies that this member is a leaf. Otherwise, the value is **`0`**). Parameters: • modelCode: the code that identifies the model for which you want to retrieve the members. • language: a valid ISO language code in the form of a lowercase, two-character string, such as **`en`** or **`es`**. • country: a valid ISO country code in the form of an uppercase, two-character string, such as **`US`** or **`SP`**, or an empty string. • variant: this parameter is not used and should be set to an empty string (**`""`**). • sasLibraryName: the libref for the SAS library that will hold the result set. • resultDataSetName: the name for the result set. Throws: FinanceClientException

Methods inherited from class com.sas.solutions.finance.api.BaseApi: authenticate, buildExceptionMessageString, getErrorMessage, getMessage, setEnvironment, setLocale, and trim.

Methods inherited from class java.lang.Object: equals, getClass, hashCode, notify, notifyAll, toString, and wait.

Performing Queries with the %FMQUERY Macro

Overview

The %FMQUERY macro performs a query against a model. It creates a temporary file to hold the results of the query and then writes those results to the output data set. Use this macro instead of the executeQuery method of the Model class. The macro returns the same result as a query in Excel, including calculated members.

Note: This macro applies only to financial models.

The %FMQUERY macro supports two types of queries:

- **MDX queries:** Queries that use MDX syntax, which is similar to SQL syntax.

  For information about MDX syntax, see "MDX Reference for SAS Financial Management" on page 48.

- **Non-MDX queries:** Queries that are based on a model code and a data set that contains query parameters.

The SAS library that is used for the query data set (required for non-MDX queries) and the result data set must be registered in the metadata repository. Before calling the macro, the SAS program is responsible for the following:

- ensuring that *queryDataSetName* exists (if applicable).
- ensuring that *resultDataSetName* does not exist (unless you want to write over an existing data set).

  Note: If *resultDataSetName* does exist before the query and the query fails (for example, because the user lacks sufficient privileges), the data set is not deleted.

The SAS program is also responsible for cleanup of these tables after the call.

Syntax

```
%FMQUERY (
  LOCALSASLIBNAME,
    RESULTDATASETNAME,
    MDXSTRING="",
    MODELCODE="",
    SASLIBNAME="",
    QUERYDATASETNAME="",
    FILTEROPTS=0,
    MEMBEROPTS=0,
    TRUSTEDUSERNAME="",
    TRUSTEDPASSWORD="",
    ENVIRONMENT="default",
    DELIM=';'
)
```

LOCALSASLIBNAME
: The libref for the SAS library that holds the result set. This library must be defined during the current SAS session; typically, it is the WORK library.

RESULTDATASETNAME
: The name of the result set to be produced by the query.

The result table contains the following columns:

DIMENSION_TYPE_CD: the member code for each dimension. The calling routine must handle illegal characters in the member codes.

VALUE: the corresponding value. **`NaN`** is represented as a period (.).

Records are filtered according to the *filterOptions*.

MDXSTRING
: The query to be executed. This parameter is required if you are performing an MDX-style query.

Note: For non-MDX-style queries, use the QUERYDATASETNAME parameter instead.

MODELCODE
: The identifier for the results model to be used in the query.

Note: This parameter is not used for MDX-style queries.

SASLIBNAME
: The libref for the SAS library that holds the query data set. This library must be registered in the metadata repository.

QUERYDATASETNAME
: The name of the SAS table that contains the query. This table must exist before you call %FMQUERY. For details, see "The Query Data Set" on page 46.

This parameter is not used for MDX-style queries. Instead, use the MDXSTRING parameter.

FILTEROPTS
: A value that specifies filters to be applied to the result set. Valid options are:

`0`: include all crossings (default)

`1`: exclude missing values

`2`: exclude zero values

`3`: exclude missing and zero values

MEMBEROPTS
: Additional member attributes, including hierarchical ordering, to be printed beside the member codes. The parameter can have any combination of these values:

`0`: include only member code (_CD) columns.

`1`: include member name (_NAME) columns.

`2`: include member description (_DESC) columns.

`4`: include member hierarchy sort (_SORT) columns. The sort values are represented as hierarchical child numbering of the member starting from the root of the hierarchy (such as `1.4.2.5`).

Regardless of other options, the member code column is always printed. The options can be used in any combination. For example, a value of `5` includes the member code (always), the member name, and the member hierarchy sort columns.

TRUSTEDUSERNAME
: The user name for logging on to the middle tier.

TRUSTEDPASSWORD
: The password for logging on to the middle tier. For information about encoding passwords, see the *SAS Intelligence Platform: Security Administration Guide*.

ENVIRONMENT
: An environment (such as **`default`**, **`dev`**, or **`prod`**) refers to an installation of SAS Solutions Services and one or more solutions.

  The environment value is site-specific. For more information, see “Client Installation and Configuration” in the *SAS Solutions Services: System Administration Guide*.

DELIM
: The delimiter used to separate items in the results (for example, to separate member codes, names, and descriptions). The default delimiter is a semicolon.

The Query Data Set

For non-MDX queries, one parameter of the %FMQUERY macro is QUERYDATASETNAME, the name of a table that contains the query. This table must exist before you call the macro, and it must reside in the same library as the result set that is produced by the query.

Note: This parameter is not used for MDX queries.

The table has the following columns:

Table 3.10 *Contents of the Query Data Set*

Column	Description	Data Type
DIMENSION_TYPE_CD	Dimension type code	character
MEMBER_CD	Member code. The dimension type code and member code pair define the root of the subtree to be queried.	character
INCLUDE_MEMBER	`0`: exclude the member `1`: include the member	numeric
INCLUDE_LEAVES	`0`: exclude leaves `1`: include first-level leaves `2`: include all levels of leaves `3`: include first-level leaves and virtual children `4`: include all levels of leaves and virtual children	numeric
INCLUDE_ROLLUPS	`0`: exclude roll-ups `1`: include first-level roll-ups `2`: include all levels of roll-ups	numeric

%FMQUERY Example (Non-MDX)

This example executes a query against a fictitious model that is named TESTING18_MODEL. The query data set name is QUERYPARAMETERS. In this example, the results are written to the NONMDXRESULTDATASETNAME data set in the WORK library.

Example Code 3.1 *Non-MDX Query*

```
LIBNAME stagedds BASE "C:\SAS\Config\Lev1\SASApp\Data\SolutionsServices\stagedds";

data stagedds.queryParameters;
   length DIMENSION_TYPE_CD MEMBER_CD $32;
   DIMENSION_TYPE_CD = "ACCOUNT"; MEMBER_CD = "A8420"; INCLUDE_MEMBER=1;
   INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
   DIMENSION_TYPE_CD = "TIME"; MEMBER_CD = "DEC1997"; INCLUDE_MEMBER=1;
   INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
   DIMENSION_TYPE_CD = "CURRENCY"; MEMBER_CD = "USD"; INCLUDE_MEMBER=1;
   INCLUDE_LEAVES=0; INCLUDE_ROLLUPS=0; output;
run;
%fmquery(modelCode="testing18_model",localSasLibName="Work", sasLibName="stagedds",
   queryDataSetName="queryParameters", resultDataSetName="NONMDXResultDataSetName",
   trustedUserName="sasdemo",trustedPassword="DemoDemo1",
   environment="default")
```

%FMQUERY Example with MDX String

Here is an example of calling %FMQUERY using an MDX string:

Example Code 3.2 *MDX Query*

```
%fmquery("Work", "MDXResultDataSetName",
   mdxString="SELECT {ACCOUNT.A8420} on 0 FROM testing18_model WHERE (TIME.DEC1997, CURRENCY.USD)",
   trustedUsername="sasdemo",
   trustedPassword="DemoDemo1", environment="default")
```

Note: The mdxString cannot include a line break.

For one approach to creating an MDX string, see "Copying an MDX String" on page 47.

For MDX reference information, see "MDX Reference for SAS Financial Management" on page 48.

Note: Currently, MDX queries in SAS Financial Management do not support the equivalent of the INCLUDE_MEMBER, INCLUDE_LEAVES, or INCLUDE_ROLLUPS options (that are available in non-MDX queries). In an MDX query, you must specify each member separately. To include leaves for one or more dimensions, specify those leaf members in the MDX string.

Copying an MDX String

To create an MDX string, one simple approach is to save the string that is created when you insert a Read-only table in Microsoft Excel. Follow these steps:

1. In Microsoft Excel, log on to the middle tier.
2. Insert a Read-only table.
3. Open the table properties.
4. Select the **Dimensions** tab.
5. Click **Query Diagnostics**.

6. Click **Copy ODCS MDX String to Clipboard**.

The MDX string for the Read-only table is available on the Windows clipboard.

MDX Reference for SAS Financial Management

Overview

Via ODCS, SAS Financial Management supports simple MDX queries that extend the capabilities that are available with the standard query parameters.

Previously, complex queries required exploding the cube or running multiple, smaller queries. By stacking multiple dimensions on an axis, MDX allows clients to express the specific query they need.

Only a small subset of MDX functionality is currently supported in ODCS:

- basic queries: `SELECT ... FROM ... WHERE ...`
- basic member functions

More sophisticated features are not currently supported. For example, these features are not currently supported:

- creating or manipulating metadata
- defining calculated members
- more advanced functions, such as filter, aggregate, and non-empty
- anything that is defined on a WITH clause

Members

A member is represented as ***DimensionTypeCode.MemberCode***. For example:

- `CURRENCY.USD`
- `TIME.Jan2001`
- `INTORG.Legal`

Note: Standard MDX and OLAP do not have the concept of dimension types. Instead, they use dimension codes to define members. ODCS uses dimension types, because they make it easier to reuse queries between virtual cubes (vcubes). In this MDX reference, references to dimensions and dimension types are interchangeable.

All codes in ODCS are case sensitive. If a dimension type code or member code includes a non-alphanumeric character, the code must be wrapped in square brackets, as in these examples:

- `INTORG.[R&D]`
- `ANALYSIS.[My Analysis]`
- `PRODUCT.[Hershey's Kisses]`
- `[`***CUSTOM TYPE***`].[My Member]`

A member function can be appended to a member using the following syntax:
DimensionTypeCode.MemberCode.Function

An example is the VC function, a SAS Financial Management function that returns the virtual child of the member:

- `INTORG.Legal.VC`

- `PRODUCT.[Hershey's Kisses].VC`

(In MDX, the virtual child is known as a DataMember.)

Tuples

A tuple is a combination of members from one or more dimensions, with only one member from each dimension. You can think of it as a multidimensional member. The simplest example of a tuple has one member, such as `INTORG.Legal`.

When there are multiple members on a tuple, the members are separated by commas and the entire tuple is wrapped in parentheses, as in these examples:

- `(INTORG.Legal, TIME.Jan2001)`
- `(INTORG.Legal, TIME.Jan2001, PRODUCT.[Hershey's Kisses])`
- `(INTORG.Legal, TIME.Jan2001, PRODUCT.[Hershey's Kisses], CURRENCY.USD, ANALYSIS.Actuals)`

It is important to remember that tuples can have only one member from each dimension. The following tuples are invalid because they have multiple members from the same dimension:

- `(INTORG.Legal, TIME.Jan2001, TIME.Feb2001)`

  Invalid: two members from the TIME dimension.
- `(INTORG.Legal, TIME.Jan2001, INTORG.[R&D])`

  Invalid: two members from the INTORG dimension.

Tuple Sets: { }s

A tuple set is an ordered collection of tuples. A tuple set can have one tuple, multiple tuples, or even zero tuples. Within a set, tuples can be repeated.

Note: This definition differs from the mathematical definition of a set or the Set data structures in Java.)

The tuples in a set can have one or more members. A set is wrapped in curly braces, and the tuples are separated by commas. Here are some examples:

- `{ INTORG.Legal, INTORG.[R&D] }`

  Set with two tuples, each containing one member.
- `{ (INTORG.Legal, TIME.Jan2001) }`

  Set with one tuple (wrapped in parentheses), containing two members.
- `{ (INTORG.Legal, TIME.Jan2001), (INTORG.[R&D], TIME.Feb2001) }`

  Set with two tuples, each tuple containing two members.
- `{ (INTORG.Legal, TIME.Jan2001, ANALYSIS.Actuals), (INTORG.[R&D], TIME.Feb2001, ANALYSIS.Budget) }`

  Set with two tuples, each tuple containing three members.
- `{ (INTORG.Legal, TIME.Jan2001), (INTORG.[R&D], TIME.Feb2001), (INTORG.[R&D], TIME.Feb2001) }`

  Set with three tuples, each tuple containing two members. One tuple is repeated.

All tuples in a set must have the same dimensions represented, and the dimensions must be in the same order. This is called the dimensionality of the tuple. Notice that all of the examples above meet this requirement. The last example has three tuples, each with two members. All three tuples contain the same dimensions and specify the INTORG dimension first and the TIME dimension second. Thus, they have the same dimensionality.

The following sets are invalid because they do not have the same dimensionality:

- `{ (INTORG.Legal, TIME.Jan2001), (INTORG.[R&D], ANALYSIS.Budget) }`

  Invalid: TIME and ANALYSIS are different dimensions.

- `{ (INTORG.Legal, TIME.Jan2001), (TIME.Feb2001, INTORG.[R&D]) }`

  Invalid: tuple dimensions are not in the same order.

- `{ (INTORG.Legal), (ANALYSIS.Actuals) }`

  Invalid: INTORG and ANALYSIS are different dimensions.

- `{ INTORG.Legal, ANALYSIS.Actuals }`

  Invalid: INTORG and ANALYSIS are different dimensions.

  This example might look like a single tuple with two members. However, it is actually a tuple set with two tuples, each containing one member (using the convention of omitting parentheses for a tuple with a single member). Because the members are from different dimensions, the tuple set is invalid.

Basic Query Syntax

The MDX query syntax enables you to define the view of the data that you want returned. Syntactically, it is similar to an SQL query. The basic syntax of a SELECT clause is as follows:

SELECT {*tuple set*} ON COLUMNS, {*tuple set*} ON ROWS

This simple query retrieves data with TIME members on the columns and INTORG members on the rows:

-
  ```
  SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001,
  TIME.Q12001} ON COLUMNS, {INTORG.Legal, INTORG.[R&D]} ON ROWS
  FROM [My VCube]
  ```

Note: The example queries in this chapter contain line breaks only so that they fit on the page. In the %FMQUERY macro, MDX query strings cannot contain a line break. In addition, keywords are shown in upper case. However, MDX queries are not case sensitive.

The results would resemble the following:

	TIME.Jan2001	TIME.Feb2001	TIME.Mar2001	TIME.Q12001
INTORG.Legal	2	6	10	18
INTORG.[R&D]	10	40	20	70

The SELECT clause defines one or more axes, with each axis assigned a position on the table (columns or rows). The example above defines two axes: TIME on columns and INTORG on rows. Notice the curly braces in the row axis definition, denoting a tuple set. Each tuple in the set contains only one member. However, like any tuple set, it can contain multiple members. This feature enables you to stack multiple dimensions on an axis, mixing and matching members between dimensions.

The following example crosses the INTORG members with different ANALYSIS members on the rows:

- **`SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { (INTORG.Legal, ANALYSIS.Actuals), (INTORG.[R&D], ANALYSIS.Budget) } ON ROWS FROM [My VCube]`**

The results would resemble the following:

	TIME.Jan2001	**TIME.Feb2001**	**TIME.Mar2001**	**TIME.Q12001**
INTORG.Legal **ANALYSIS.Actuals**	2	6	10	18
INTORG.[R&D] **ANALYSIS.Budget**	20	60	15	95

WHERE Clause: Defining a Slicer

The previous examples use only two or three dimensions in the queries. For any dimensions in the cube that were not specified (such as CURRENCY, PRODUCT, or ACCOUNT), the default member for the dimension is implicitly used in the query.

What if you want to cross your table with members that are not default members? In MDX, you can use a WHERE clause to define members that apply to the entire table. This clause is known as a slicer. The example below defines a slicer for three dimensions that are not shown on the table:

- **`SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, {INTORG.Legal, INTORG.[R&D]} ON ROWS FROM [My VCube] WHERE (CURRENCY.USD, ANALYSIS.Budget, FREQUENCY.PA)`**

Results would resemble the following:

Slicer: CURRENCY.USD, ANALYSIS.Budget, FREQUENCY.PA				
	TIME.Jan2001	**TIME.Feb2001**	**TIME.Mar2001**	**TIME.Q12001**
INTORG.Legal	4	8	12	24
INTORG.[R&D]	20	60	15	95

Notice that the slicer in the WHERE clause is enclosed by parentheses: it is really just a tuple. Like any tuple, it can contain one or more members, and the members must be from different dimensions. In addition, the slicer in the tuple cannot contain a member from a dimension that is used in one of the axes. The following example is invalid because it uses the TIME dimension on both the rows and the slicer:

- **`SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, {INTORG.Legal, INTORG.[R&D]} ON ROWS FROM [My VCube] WHERE (TIME.Apr2001, ANALYSIS.Budget)`**

SELECT Clause: Defining Axes

So far, all the query examples have used only two axes: columns and rows. However, an MDX query can have anywhere from 0–64 axes. Beyond COLUMNS and ROWS, the axis keywords are PAGES, CHAPTERS, and SECTIONS. Here are examples of queries that use a different number of axes:

- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS, {INTORG.Legal} ON ROWS, {ANALYSIS.Actuals, ANALYSIS.Budget} ON PAGES FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS, {INTORG.Legal} ON ROWS, {ANALYSIS.Actuals, ANALYSIS.Budget} ON PAGES, {FREQUENCY.PTD} ON CHAPTERS FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON COLUMNS, {INTORG.Legal} ON ROWS, {ANALYSIS.Actuals, ANALYSIS.Budget} ON PAGES, {FREQUENCY.PTD} ON CHAPTERS, {PRODUCT.Widgets, PRODUCT.Gadgets} ON SECTIONS FROM [My VCube] WHERE (CURRENCY.USD)`

Instead of using the axis keywords such as COLUMNS or PAGES, you can refer to axes by numbers, beginning with 0 (where 0=COLUMNS, 1=ROWS, 2=PAGES, 3=CHAPTERS, and 4=SECTIONS). Beyond sections, you must use numbers. The following queries are the same as the examples above, except that they use axis numbers instead of keywords:

- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0 FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0, {INTORG.Legal} ON 1, {ANALYSIS.Actuals, ANALYSIS.Budget} ON 2 FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0, {INTORG.Legal} ON 1, {ANALYSIS.Actuals, ANALYSIS.Budget} ON 2, {FREQUENCY.PTD} ON 3 FROM [My VCube] WHERE (CURRENCY.USD)`
- `SELECT {TIME.Jan2001, TIME.Feb2001} ON 0, {INTORG.Legal} ON 1, {ANALYSIS.Actuals, ANALYSIS.Budget} ON 2, {FREQUENCY.PTD} ON 3, {PRODUCT.Widgets, PRODUCT.Gadgets} ON 4 FROM [My VCube] WHERE (CURRENCY.USD)`

Note: You cannot skip axis definitions. For example, you cannot specify `0` and `2` and omit `1`.

Specifying Excluded Members

In an ODCS query, you can specify excluded members (members on an axis that should be ignored while running a query). Because there is no equivalent concept in MDX, ODCS supports an MDX extension for using this functionality in SAS Financial Management. At the end of a query, you can add an EXCLUDE clause to specify the members to be excluded from the query. For each dimension from which you want to exclude members, the EXCLUDE clause contains a tuple set separated by commas, as in these examples:

- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { INTORG.All } ON ROWS FROM [My VCube] EXCLUDE { INTORG.Legal }`

- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { INTORG.All } ON ROWS FROM [My VCube] WHERE ( CURRENCY.USD ) EXCLUDE { INTORG.Legal, INTORG.[R&D] }`
- `SELECT {TIME.Jan2001, TIME.Feb2001, TIME.Mar2001, TIME.Q12001} ON COLUMNS, { INTORG.All } ON ROWS, { PRODUCT.All } ON PAGES FROM [My VCube] WHERE ( CURRENCY.USD ) EXCLUDE { INTORG.Legal, INTORG.[R&D] }, { PRODUCT.Widgets }`

Notice that each set corresponds to a dimension in the query, and each tuple in the set contains only one member.

Supported Member Functions

ODCS supports a limited number of functions:

`.VC`
Uses the virtual child of the member. Examples:

- `INTORG.Legal.VC`
- `INTORG.[R&D].VC`

`.DataMember`
MDX term for the ODCS term “virtual child.” This function is interchangeable with the .VC function. Example:

- `INTORG.Legal.DataMember`

`.Ignore`
Placeholder member that is never calculated. This function is used by Excel to overlay client-side calculations after the MDX table is returned. Only the dimension type code must be valid; the member code is ignored by the server. Here is an example:

- `PRODUCT.MyClientSideCalc.Ignore`

ODCS versus Standard OLAP

The ODCS architecture differs from standard OLAP in a few ways. These differences affect MDX usage and syntax support:

- ODCS supports only a single, numeric measure. Therefore, there is never a need to use the MEASURES keyword in a query.
- Levels are not supported explicitly in ODCS, except for certain dimensions such as TIME. Currently, there is no support for referencing Levels in the query syntax.
- In ODCS, members in the same dimension must have a unique code. Because a cube has only one dimension for each dimension type, a member code is always unique in a given dimension type at query time.

  This requirement provides the shortcut when defining member definitions of ***`DimensionTypeCode.MemberCode`***, such as **`TIME.Jan05`**. If ODCS supported non-unique member codes in a dimension, you would need to follow the MDX standard and specify the ancestors of the member, such as **`TIME.2005.Q1.Jan`**.

Calling the %GETMODELPROPERTIES Macro

Overview

The %GETMODELPROPERTIES macro creates a data set that contains the following properties for the specified models: dimension_type_cd, hierarchy_cd, member_cd, property_cd, property_name, and property_value. The macro is defined within the %FMMODEL macro. Use the following syntax:

```
%FMMODEL
%GETMODELPROPERTIES(...)
```

(See “Example” on page 55.)

Note: You can call this macro instead of using the getModelMemberPropeties method of the Model class.

Syntax

GETMODELPROPERTIES (
  MODELCODE,
    SASLIBNAME,
    OUTPUTDATASETNAME,
    TRUSTEDUSERNAME="",
    TRUSTEDPASSWORD="",
    ENVIRONMENT="",
    DIMTYPECODES="",
    PROPERTYCODES="",
    MEMBERCODES="",
    DELIM=';'
)

MODELCODE
The identifier for the model to be used in the query.

SASLIBNAME
The libref for the SAS library that holds the result set.

OUTPUTDATASETNAME
The name of the result set produced by the query. After the query is executed, the data is written to *sasLibName.outputDataSetName*.

TRUSTEDUSERNAME
The user name for logging on to the middle tier; required for a stored process.

For use in an interactive SAS session, you can omit the TRUSTEDUSERNAME and TRUSTEDPASSWORD. At run time, the user is prompted for the user name and password.

TRUSTEDPASSWORD
The password for logging on to the middle tier.

ENVIRONMENT
The environment for logging on to the middle tier. (See “Specifying the SAS Solutions Environment” on page 19.)

DIMTYPECODES
A delimited list of unquoted dimension type codes to use in the query. If this parameter is omitted, all dimension type codes are used. Example:

```
DIMTYPECODES=”TIME;INTORG”
```

PROPERTYCODES
A delimited list of unquoted property codes to use in the query. If this parameter is omitted, all properties are used.

MEMBERCODES
A delimited list of unquoted member codes to use in the query. If this parameter is omitted, all members are used.

DELIM
The delimiter that was used to separate the DIMTYPECODES, PROPERTYCODES, and MEMBERCODES values. The default is a semi-colon (**`;`**).

Example

This example writes the properties for all members of a model named **`Default_Model`**:

```
%FMMODEL
%GETMODELPROPERTIES ("Default_Model", "FMSData", "TstModelProperties",
   trustedUsername="sasdemo", trustedPassword="DemoDemo1", environment="default")
```

Chapter 4

Customizing a Workflow

About Customizing a Workflow

Note: This chapter applies to both financial form sets and operational form sets.

In SAS Financial Management, a workflow defines the review and approval process used in budgeting, forecasting, and other planning activities. Each workflow consists of a collection of states (such as READY, EDITED, and COMPLETE) and actions (such as PUBLISH and EDIT). At run time, the actions advance the workflow from one state to the next. Each action triggers a corresponding policy file (code that is associated with these actions).

You can customize a workflow by writing a stored process that executes before or after the workflow is advanced. This chapter explains how to add your custom code to a workflow. It also contains a short example of a workflow stored process.

Workflow Types

Overview

SAS Financial Management supports two types of workflows: top-down and bottom-up.

Note: For more information about the terminology that is used in this chapter, see the SAS Financial Management User's Guide or the online Help for the SAS Financial Management Add-In for Microsoft Excel.

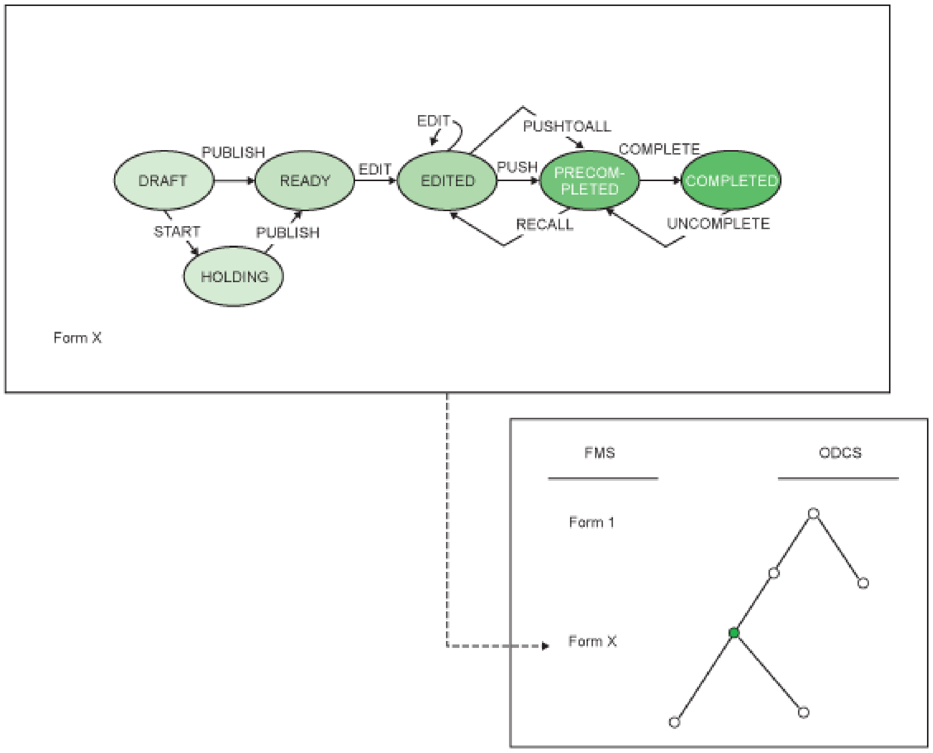
Top-Down Workflow

A top-down workflow enables users at any roll-up point to make bulk updates and adjustments down and across multiple entities and dimensions.

A data-entry project that has a top-down workflow begins when a top-down form set is published from SAS Financial Management Studio. The workflow ends when a Finance Process Administrator applies the COMPLETE action to the form set in SAS Financial Management Studio.

Here is a schematic diagram of a top-down workflow. The applicable states are displayed in ellipses, and the applicable actions are displayed as lines that connect one state to another.

Figure 4.1 *A Top-Down Workflow*



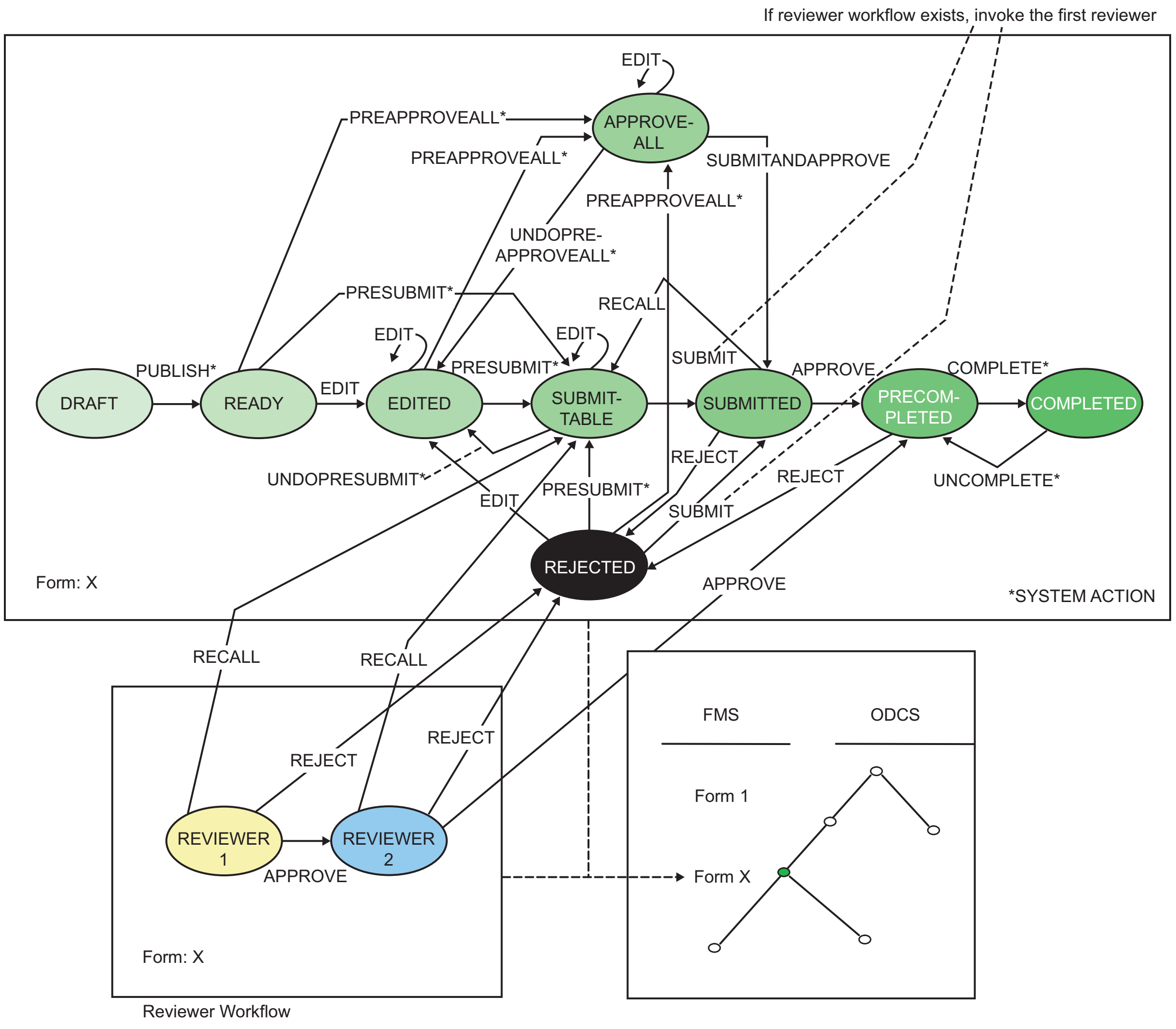
Bottom-Up Workflow

In a bottom-up workflow, forms begin at the lower levels of the hierarchy and are aggregated and reviewed by the organization as they move up an approval hierarchy. (Optional) A bottom-up workflow can be connected to a separate reviewer workflow that supports additional reviewers in the budget approval process.

A bottom-up workflow begins when a bottom-up form set is published from SAS Financial Management Studio. The workflow ends when a Finance Process Administrator applies the COMPLETE action to the form set in SAS Financial Management Studio.

Here is a schematic diagram of a bottom-up workflow, with the applicable states and actions. The diagram also contains a reviewer workflow (for two reviewers) that is attached to the bottom-up workflow.

Figure 4.2 *A Bottom-Up Workflow*



*Notes on system actions:

- Children are moved from PRECOMPLETED to COMPLETED when their parent invokes SUBMIT.
- Children are moved from COMPLETED to PRECOMPLETED when their parent is recalled.
- Children are moved from COMPLETED to PRECOMPLETED when their parent is rejected.
- A parent is moved from SUBMITTABLE to EDITED if any child is recalled and moved to SUBMITTABLE.
- A parent is moved from SUBMITTABLE to EDITED if any child is rejected and moved to that state.
- A parent is moved from EDITED to SUBMITTABLE if all its children are in a PRECOMPLETED state.

Adding Your Custom Code to a Workflow

The Pre and Post Classes

Two Java classes (Pre and Post) form the bridges between the SAS Financial Management workflow system and the SAS stored processes in which the customized code is deployed.

Whenever a policy file is triggered, the Pre.invoke method is called before the policy file is executed, and the Post.invoke method is called after the policy file is executed. These methods call a stored process if one is linked to this part of the workflow.

If the stored process fails (due to exception or error in the customized codes), the workflow does not advance to the next state.

- If the Pre operation fails, the policy file is not executed.
- If the Post operation fails, the workflow is rolled back to its previous state.

However, if the stored process itself makes any changes, such as updating the database, those changes remain.

Steps in Customizing a Workflow

Do not modify the Pre and Post classes directly. To customize the workflow, follow these steps:

1. Write a SAS stored process to perform the necessary business logic.

   The stored process must set the FM_SP_RESULT environment variable. If the operation fails, the program should set FM_SP_RESULT to **`INVALID`** and set the FM_SP_MESSAGE environment variable to an appropriate text message. Otherwise, the stored process should set FM_SP_RESULT to **`VALID`**.

   These environment variables are available on the middle tier. If the value of FM_SP_RESULT is **`INVALID`**, an exception is thrown, the workflow is not advanced to the next state, and the corresponding text message is displayed in a message box in the rich client or in the Web browser.

   For information about writing a stored process, see Chapter 2, “Working with Stored Processes,” on page 5. For an example stored process, see “Data Validation Example” on page 63.

2. On the data tier, save the stored process in a directory such as ***`SAS-config-dir`*** **`\Lev1\SASApp\SASEnvironment\FinancialManagement\SASCode\UserDefined`**. (Create the **`UserDefined`** directory if it does not already exist.)

3. Log on to SAS Management Console as an administrator and register the stored process in the **`/Products/SAS Financial Management/Customized workflow`** folder. (You might need to create this folder.)

4. Create a resource file that links a workflow action to the stored process. If the resource file already exists, update the file with information about the new stored process. See “The Resource File” on page 61.

The Resource File

Update the Resource File

The resource file is an XML file that provides the location of a stored process and associates it with a specific form set and an action. A template for a resource file follows:

```
<SASWorkflowCustomizations>
  <Application name="SAS Financial Management">
    <Object type="FormSet" name="form_set_ID">
      <Action type="action_type">
        <Execute type="execute_type"
          storedProcessFullPath="path_to_stp"/>
      </Action>
    </Object>
  </Application>
</SASWorkflowCustomizations>
```

Replace the italicized strings with the appropriate values:

- *execute_type* specifies when the stored process is called, relative to execution of the policy file. It must have a value of **`pre`** or **`post`**.
- *action_type* is an action such as SUBMIT or REJECT.

  For a list of available action types, see Table 4.1 on page 62. Notice that some actions are available only in a top-down workflow or only in a bottom-up workflow.
- *path_to_stp* is the path to the stored process metadata definition, such as **`/Products/SAS Financial Management/Customized workflow/validation`**.

  You can link the same stored process to more than one form set or action: just create a separate <Object> entry for each form set, action type, and execute type combination.
- *form_set_ID* is the ID of the form set to which the action applies. To look up a form set ID in the SASSDM database, you can use the following SQL query:

  ```
  "select form_set_id from sassdm.sas_form_set where form_set_nm='form-set-name'"
  ```

Here is an example:

Example Code 4.1 *Example Resource File*

```
<SASWorkflowCustomizations>
  <Application name="SAS Financial Management">
    <Object type="FormSet" name="2">
      <Action type="SUBMIT">
        <Execute type="pre" storedProcessFullPath= _
         "/Products/SAS Financial Management/Customized workflow/Customized workflow test"/>
      </Action>
      <Action type="REJECT">
        <Execute type="post"
          storedProcessFullPath= _
           "/Products/SAS Financial Management/Customized workflow/Customized workflow test"/>
      </Action>
    </Object>
  </Application>
</SASWorkflowCustomizations>
```

Note: Line breaks (“_”) added for readability.

Table 4.1 Available Workflow Actions

Action Type	Top-down Workflow	Bottom-up Workflow	Description
PUBLISH	√	√	Moves a form from the DRAFT state to the READY state so that it can be edited.
SUBMIT		√	Submits a form for approval.
EDIT			Opens a form for editing.
REVIEW		√	Opens a form in read-only mode so that it can be reviewed.
REJECT		√	Changes the form's state to REJECTED and notifies the user who submitted the form.
APPROVE		√	Approves a form and copies that form's data to its parent form.
RECALL	√	√	Recalls a form so that it can be further edited and then pushed again or resubmitted.
PUSH	√		Makes a form available to the users who are responsible for the top member's children. The amounts that have been allocated to the children of that member are copied to the forms for those child members. As a result, the users who are responsible for the child members to edit their forms, allocate the pushed amounts to the next level of child members, and then push their forms in turn.
PUSHTOALL	√		Makes a form available to the users who are responsible for all the top member's descendants. The amounts that have been allocated to the descendants of that member are copied to the forms for those descendant members. As a result, the users who are responsible for the descendant members to edit their forms. However, their editing is limited to redistributing amounts within their target member. No other user can push amounts to the next level of child members because PUSHTOALL cascades all the way down the target hierarchy in a single step.
COMPLETE	√	√	Ends the workflow. This action can be performed only by a Finance Process Administrator.
UNCOMPLETE	√	√	Reactivates a form for further work. This action can be performed only by a Finance Process Administrator.
SUBMITAPPROVE		√	Moves a form from the APPROVEALL state to the SUBMITTED state.
PREAPPROVEALL		√	Moves a form from the READY or EDITED state to the APPROVEALL state.

Action Type	Top-down Workflow	Bottom-up Workflow	Description
UNDOPREAPPROVEALL		√	Moves a form from the APPROVEALL state to the EDITED state.

You can associate as many actions with a form set as necessary, but each action can have only one stored process associated with it. On the other hand, you are free to associate the same stored process with multiple actions in multiple form sets, if applicable.

Name the file WorkflowCustomizations.xml and save it on the middle tier, where the Web application server resides. A good location is the following directory: ***SAS-config-dir*** **\Lev1\CustomAppData\FMCustomizedWorkflow**.

Set the JVM Options

To make the resource file available, add the following option to the JVM options for SASServer3 (the managed server to which SAS Financial Management is deployed).

```
-Dsas.workflow.customizations="file:///path-to-resource-file"
```

Here is an example:

```
-Dsas.workflow.customizations=
"file:///C:/SAS/Config/Lev1/CustomAppData/Workflow/WorkflowCustomizations.xml"
```

The option applies when you restart the managed server.

Note: You do not need to restart the managed server when you make updates to the resource file.

Data Validation Example

About the Data Validation Example

Here is an example of cell-based data validation that uses a stored process, an execute type of **`pre`**, and a SUBMIT action. At run time, when a user submits a form in the specified form set, the stored process is automatically triggered. It validates a cell value in the form. If the value is greater than `0`, the SUBMIT succeeds. Otherwise, the SUBMIT fails.

The example makes the following assumptions:

- A form set with ID **`123`** has been created.
- A form template with a result model (called **`tst_model`**) has been saved. It includes the dimensions shown in the Dimension column of Table 4.2 on page 64..
- The form cell whose value is to be validated is defined by the crossing that is exemplified by the codes in the Member Code column of the following table.

The dimensionCodes and memberCodes arrays in the example contain the values from the first and second columns, respectively, of the following table. You do not need to include all the values in the table in the two arrays, but the values of the two arrays must match. During the query, any missing dimension code-member code pairs are filled with default values from the dimensions that are defined for the results model and the default read member that is defined in the hierarchy for each dimension.

***Table 4.2** Example Dimensions and Member Codes*

Dimension	Member Code
ACCOUNT_FM	6232
ANALYSIS_FM	BUDGET
Cost Center	Total
CURRENCY	EUR
fm_INTORG_CODE	WW_SA
TIME_FM	012002
fm_INTORG_CODE_TRADER	EXT
SOURCE	BaseForm
PRODUCT_FM	Jackets

The actual query is carried out in the following code:

```
model.callDoubleMethod("getCellValue", "tst_model", dimensionCodes,
    memberCodes, value);
```

Depending on the return value, the program sets the FM_SP_RESULT and FM_SP_MESSAGE environment variables. If the return value is less than or equal to 0, the program sets FM_SP_RESULT to **`INVALID`** and sets FM_SP_MESSAGE to a text message. Otherwise, the program sets FM_SP_RESULT to **`VALID`**.

This example uses methods from the SAS Financial Management Java API. Most of the classes in this API apply only to financial planning. For details, see Chapter 3, "The SAS Financial Management Java API," on page 17.

Code for the Example

This SAS program retrieves the data from the cell and validates the data.

Note: If you are declaring a Javaobj, the picklist option is required in the DATA step so that the Javaobj can find the necessary JAR files.

***Example Code 4.2** Stored Process for Workflow Customization*

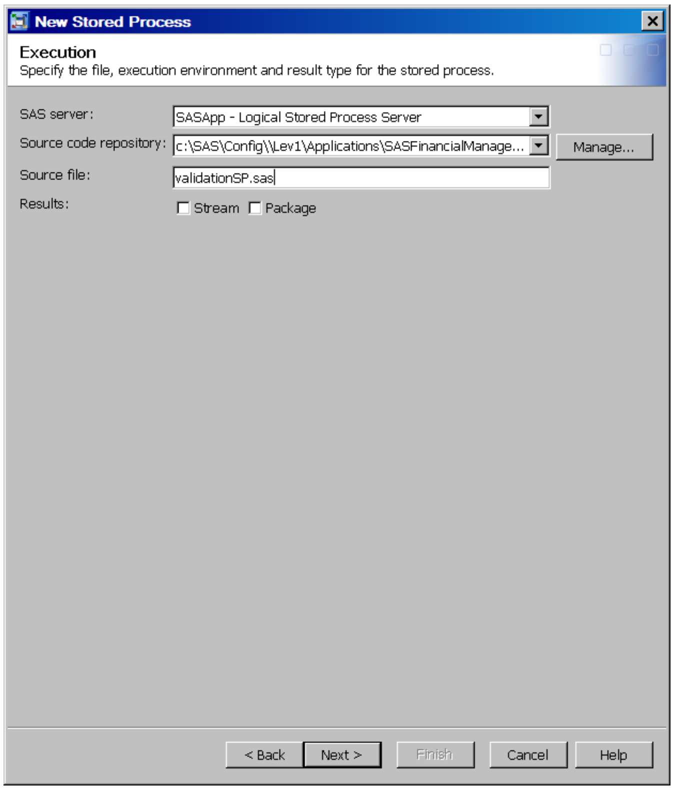
```
data _null_ /picklist='finance/finance.txt';
   put 'This is a data entry validation test';

   /* Read and echo environment variables passed in from the middle tier */
   /* form ID */
   length formId $20;
   formId = symgetc("fm_sp_form_id");
   put formId=;

   /* security key */
   length secKey $200;
   secKey = symgetc("fm_sp_seckey");
```

```
put secKey=;

/* action on the form */
length action $20;
action = symgetc("fm_sp_action");
put action=;

/* user ID */
length userId $20;
userId = symgetc("fm_sp_user_id");
put userId=;

/* user name */
length userName $60;
userName = symgetc("fm_sp_user_name");
put userName=;

/* Instantiate the Form class */
dcl javaobj form("com/sas/solutions/finance/api/Form",formId, trim(secKey));
    form.ExceptionDescribe(1);

/* Call methods of the Form class and echo the results */
/* Get the target member code */
length targetMemberCode $50;
form.callStringMethod("getTargetMemberCode", targetMemberCode);
put targetMemberCode=;

/* Get the target dimension code */
length targetDimensionCode $50;
form.callStringMethod("getTargetDimensionCode", targetDimensionCode);
put targetDimensionCode=;

length cFormInfo $20000;
form.callStringMethod("getInfo",cFormInfo);
put cFormInfo=;

length authors $ 200;
form.callStringMethod("getAuthors", " ", authors);
put authors=;

length admins $30000;
form.callStringMethod("getPlanningAdministrators", " ", admins);
put admins=;

/* Instantiate the Model class */
dcl javaobj model("com/sas/solutions/finance/api/Model", trim(secKey));

/* Set up two arrays, dimensionCodes and memberCodes */
array dimensionCodes[9] $50
     (
         "",
            "ANALYSIS_FM",
            "ACCOUNT_FM",
            "Cost Center",
            "CURRENCY",
            "TIME_FM",
                "fm_INTORG_CODE_TRADER",
                "SOURCE",
                "PRODUCT_FM"
          );
   /* Set target dimension code */
   dimensionCodes[1] = targetDimensionCode;

   array memberCodes[9] $30
        (
            "",
            "BUDGET",
            "6232",
            "Total",
            "EUR",
            "012002",
            "EXT",
            "BaseForm",
            "Jackets"
          );
    /* Set target member code */
    memberCodes[1] = targetMemberCode;

    /* Call getCellValue method */
   length value 8;
   model.callDoubleMethod("getCellValue", "tst_model", dimensionCodes,
       memberCodes, value);
   put value=;

   /* Test for value <= 0  and set environment variables accordingly */
   if value <= 0 then do;
         call symput("fm_sp_result", "INVALID");
         call symput("fm_sp_message",
             "Account 6232 of JAN2002 should be greater than 0.");
   end;
   else do;
         call symput("fm_sp_result", "VALID");
   end;
   form.delete();
   model.delete();
run;
```

Registering the Stored Process

Register the stored process in SAS Management Console. This example uses the recommended location of **`/Products/SAS Financial Management/Customized workflow`**.

Note: For this example, neither **Stream** nor **Package** is selected for the **Results**, because the only output is to the log file.


New Stored Process
Execution
Specify the file, execution environment and result type for the stored process.
< Back
Next >
Finish
Cancel
Help
Manage...

SAS server: SASApp - Logical Stored Process Server
Source code repository: c:\SAS\Config\\Lev1\Applications\SASFinancialManage...
Source file: validationSP.sas
Results: ☐ Stream ☐ Package

Updating the Resource File

The resource file (**`SAS-config-dir\Lev1\CustomAppData\Workflow\WorkflowCustomizations.xml`**) might have an entry as follows:

```
<SASWorkflowCustomizations>
  <Application name="SAS Financial Management">
    <Object type="FormSet" name="123">
      <Action type="SUBMIT">
        <Execute type="pre" storedProcessFullPath=
          "/Products/SAS Financial Management/Customized workflow/validation"/>
      </Action>
    </Object>
  </Application>
</SASWorkflowCustomizations>
```

In this case, the execute type is set to “pre”, which means that the stored process is executed before the workflow policy file.

Chapter 5

Creating a Custom Cell Action

Overview

This chapter explains how to create a custom cell action for use in a read-only table in Microsoft Excel.

When a user selects a cell in the Excel read-only table and clicks the right mouse button, the **Contributing Data** action is available by default. This action enables the user to view the data records that make up the selected cell.

You can add your own custom actions that invoke a stored process that displays its output in a browser window. For example, you might create a custom action that displays the transactions that make up the selected cell. Or you might create a custom action to reconcile adjustments in consensus forecasting.

Note: Viewing read-only tables requires the SAS Financial Management Add-in for Microsoft Excel. Currently, custom cell actions cannot be applied to data-entry tables.

Follow these steps to create a custom action:

1. Write a stored process to run when the action is invoked.

   See "Write the Stored Process" on page 70.

2. In SAS Management Console, define the stored process metadata.

   See "Register the Stored Process" on page 73.

3. Define the custom action in a resource file.

   For the first custom action, you must create this file and set a JVM option that points to the resource file.

See “Update the Resource File” on page 73.

4. The new action is available from a read-only table in Microsoft Excel. When a user right-clicks a cell and selects **Tools**, the new action appears as a selection.

   See “Select the Action” on page 74.

Write the Stored Process

About the Stored Process

Your stored process will most likely use the SAS Financial Management Java API. For information about the classes and methods that make up that API, as well as information about declaring a Javaobj object and authenticating the user, see Chapter 3, “The SAS Financial Management Java API,” on page 17.

Save the stored process code on the data tier, in a location such as the ***SAS-config-dir*** **\Lev1\SASApp\SASEnvironment\FinancialManagement\SASCode** **\UserDefined** directory. Create the **UserDefined** directory if it does not already exist.

Parameters That You Can Expect

At run time, when a user selects a custom action, a URL is built to call the associated stored process. The URL includes the following parameters, which are available to the stored process:

Parameter Name	Value
_model	The model ID for this table
__*dimension-ID*	The member ID for this dimension, for the selected crossing

The parameter names are available in the _APSLIST. For example:

```
_APSLIST=__19, __8,_archive_path,_model,_metaperson, _metauser, ...
```

Dimension IDs and member IDs are represented by parameters beginning with two underscores (__). The parameter name following the underscores is the dimension ID, and the parameter value is the member ID for the selected crossing. The simple example below scans the list for variables beginning with two underscores (such as **__19**) and extracts the dimension IDs and member IDs. With the dimension ID, you can call the getDimensionCode method of the Metadata class to get the associated dimension code. With the dimension ID and member ID, you can call the getMemberCode method to get the associated member code.

Example Code 5.1 *Example Stored Process for Custom Cell Action*

```
/*+------------------------------------------------------------------
| Copyright (c) 2009 by SAS Institute Inc., Cary, NC, USA.
| All rights reserved.
| Name: viewtrans.sas
| Purpose: show DDS transactions that make up a cell value
+-------------------------------------------------------------------+*/
```

```
*Options mprint;
*ProcessBody;
ods path(prepend) sashelp.sasweb2(read);
%rptinit(style=sasweb2);
* extract crossing values from the parameter list;

LIBNAME APITest BASE "C:\sas\API_Test";
run;

Data _null_ /picklist='finance/finance.txt';

 /* Substitute the name of your environment */
dcl javaobj oModel("com/sas/solutions/finance/api/Model");
    oModel.ExceptionDescribe(1);
    oModel.callVoidMethod("setEnvironment","default");
    call METADATA_PASSID("oModel", "");

 /* Substitute the name of your model */
    oModel.callvoidmethod("getModelHierarchies","Default_Model","APITest","HierOut");
    oModel.delete();
run;

data DimType;
    set APITest.HierOut;
        If dimension_type_cd IN ("ACCOUNT", "INTORG", "ANALYSIS", "TIME")
            then call symputx(dimension_type_cd, "Dim_"||dimension_cd);
run;

data _null_ /picklist='finance/finance.txt';
    length parameter $32;
    length value $1000;
    length dimID dim member $200;

dcl javaobj oMetadata("com/sas/solutions/finance/api/Metadata");
    oMetadata.ExceptionDescribe(1);
    oMetadata.callVoidMethod("setEnvironment", "default");
    call METADATA_PASSID("oMetadata", "");
    * get the model and the list of filters ;
    do until(parameter = '');
        i+1;
        parameter = scan("&_APSLIST", i, ",");
        if parameter ne '' then do;
            value = symget(parameter);
            put parameter= value=;
            if substr(parameter,1,2)='__' then do;
                dimID=upcase(substr(parameter,3));
                if dimID ne 'FREQ' then do;
                    put dimid=;
                    oMetadata.callStringMethod("getDimensionCode", trim(dimID),dim);
                    oMetadata.callStringMethod("getMemberCode", trim(dimID),
                    trim(value), member);
                    /* set dimension values, such as ACCOUNT=10020 */
                    call symputx("dim_"||dim, member);
                end;
            end;
        end;
    end;
    oMetadata.delete();
run;

/*assign library (modify path if necessary) */

libname dds 'C:\SAS\Config\Lev1\SASApp\Data\SolutionsServices\DDSData';

proc sql;
    select b.gl_account_id, c.internal_org_id, d.analysis_id,
        e.time_period_id, a.transaction_amt
    from dds.gl_transaction_sum a,
        dds.gl_account b,
        dds.internal_org c,
        dds.analysis d,
        dds.time_period e
    where   a.gl_account_rk=b.gl_account_rk
        AND a.initiating_internal_org_rk=c.internal_org_rk
        AND a.analysis_rk=d.analysis_rk
        AND a.affected_time_period_rk=e.time_period_rk
        AND b.gl_account_id=symget("&ACCOUNT")
        AND c.internal_org_id=symget("&INTORG")
        AND d.analysis_id=symget("&ANALYSIS")
        AND e.time_period_id=symget("&TIME")
        ;
quit;

data _null_;
    if symget('SQLOBS')=0 then do;
        file print;
        put "NOTE: No rows were found";
                    value=symget("&ACCOUNT");
                put "ACCOUNT= " value;
                value=symget("&INTORG");
                put "ORG= " value;
                value=symget("&ANALYSIS");
                put "ANALYSIS= " value;
                value=symget("&TIME");
                put "TIME= " value;
    end;
run;

title;
footnote;


proc printto;
quit;


ods _all_ close;
ods listing;

%stpend;
```

Register the Stored Process

1. Log on to SAS Management Console
2. Create the stored process in a folder such as **`/Products/SAS Financial Management/Custom Cell Actions`**.

   Create the **`Custom Cell Actions`** folder if it does not already exist.
3. Define the stored process with package output.
4. Select **Package - WebDAV Server** as the output location.

   Use the Import Users and Groups stored process as a model. It is located in the **`/Products/SAS Solutions Services/Standard Reports`** folder. In the **Collection URL** properties, on the **Prompt Type and Values** tab, you must supply the full path to the WebDAV location. If necessary, create a new WebDAV folder. (See "Using the SAS Web Administration Console" in the *SAS Intelligence Platform: Web Application Administration Guide*.)
5. Make sure that the Solutions Users group has ReadMetadata and WriteMetadata access to the stored process.

For more information about registering a stored process, see the online Help in SAS Management Console. See also the *SAS Stored Processes: Developer's Guide*.

Update the Resource File

Define the Custom Action

Custom actions are defined in a resource file that is stored on the middle tier, where the Web application server resides. A good location is a directory such as ***`SAS-config-dir`*** **`\Lev1\CustomAppData\FMCustomActions`**. Create the **`FMCustomActions`** directory if it does not already exist.

The resource file is an XML file with the following contents:

```
<?xml version="1.0"?>
<customActions>
  <action name="action-name" onCell="true|false" onRollups="true|false"
      onLabels="true|false" onReadTable="true|false" onWriteTable="true|false">
    <description>description of this stored process</description>
    <url>URL to fallback page</url>
    <path>path to stored process metadata definition</path>
  </action>
</customActions>
```

The *action-name* is the name of the stored process, as defined in the metadata repository. In Microsoft Excel, it appears as the custom action.

The *fallback page* is the page to be displayed if the custom action fails for some reason. SAS Financial Management expects this file (with a name of main.html) to be available from the URL that you define in the resource file. In the example below, the fallback page would be **`http://www.mycompany.com/CustomActions/Error/main.html`**.

Each custom action can have its own page (with its own URL), or you can specify the same URL for multiple actions.

The *path* is the path to the stored process definition in the metadata repository. Do not include a slash (/) before **`Products`**, and do not include the name of the stored process.

Here is an example:

```
<?xml version="1.0"?>
<customActions>
  <action name="View transactions" onCell="true" onRollups="true"
      onLabels="false" onReadTable="true" onWriteTable="false">
    <description>View DDS transactions</description>
    <url>www.mycompany.com/CustomActions/Error</url>
    <path>Products/SAS Financial Management/Custom Cell Actions</path>
  </action>
</customActions>
```

Set the JVM Option

If you have not already done so, tell SAS Financial Management where to find the resource file. Add the following option to the JVM options for the managed server to which SAS Financial Management is deployed (by default, SASServer3):

```
-Dsas.customActions.customizations="file:///path-to-resource-file"
```

For information about configuring your Web application server, go to **`http://support.sas.com/resources/thirdpartysupport/v92/`**.

Here is an example:

```
-Dsas.customActions.customizations=
"file:///C:/SAS/Config/Lev1/CustomAppData/FMCustomActions/CustomActions.xml"
```

The option applies when you restart the managed servers for SAS Financial Management and ODCS (typically, SASServer3, SASServer4, and SASServer5).

Note: If you update the resource file, you must also restart the managed servers.

Select the Action

In Microsoft Excel, right-click a cell in a read-only table and select **Tools** to see the new action.

Chapter 6

The SAS Financial Management Add-In API for Microsoft Excel

Overview of Working with the SAS Financial Management Add-In API for Microsoft Excel

With the SAS Financial Management Add-In for Microsoft Excel and the SAS Financial Management Add-In API for Microsoft Excel, you can use Microsoft Visual Basic for Applications (VBA) to write macros that interact with SAS Financial Management objects. For example, you might perform some of the following tasks:

- Launch a SAS Financial Management report in batch mode, automatically log on to the SAS Financial Management server, and print the report with updated numbers.
- Retrieve SAS Financial Management data and metadata.
- Use the FMMember selection dialog box in a cell data access (CDA) report.
- Execute code that is based on events from the SAS Financial Management objects.
- Apply custom formatting to SAS Financial Management tables.

Note: For information about the terminology that is used in this chapter, see the *SAS Financial Management 5.1: User's Guide* or the online Help for the SAS Financial Management Add-In for Microsoft Excel.

Setup for Using the API

If you have not already done so, load the add-ins for Microsoft Excel that are required by the solutions. See the instructions in the “Installing the Client Applications” chapter of the *SAS Solutions Services: System Administration Guide*.

The API requires a reference to the SASSESExcelAddin.tlb type library. In Microsoft Excel, follow these steps to add the reference:

1. Click the **Developer** tab.
2. Click **Visual Basic**.
3. From the **Tools** menu of the Visual Basic Editor, select **References**.
4. From the list of available references, select **SASSESExcelAddIn**.

   If **SASSESExcelAddIn** is not in the list, click **Browse** to select the file and add it to the list. The file is located in the ***`SAS-install-dir`*** **`\SASFinancialManagementAdd-InforMicrosoftExcel\5.1`** directory.
5. Click **OK**.

Note: If you had an earlier version of the SAS Financial Management Add-In for Microsoft Excel, deselect the check box for **SASSESExcelAddin** on the References page, click **OK**, and exit Excel. Then re-open Excel and add the new TLB file as described above.

General Usage Information

Declaring the FMAddIn Object

In the **`Declarations`** section of the Workbook module, declare the FMAddIn object and other SAS Financial Management objects in code that resembles the following:

```
Public addin As FMAddIn
Public table As FMTable
Public cube As FMCube
Public user As FMUser
```

To use the events framework, the declarations for FMAddin and FMTable should resemble the following code:

```
Public WithEvents addin As FMAddIn
Public WithEvents table As FMTable
```

For more information about the events framework, see "Handling Events" on page 78.

Working with Objects

The FMAddin Object

To get a reference to the FMAddIn object, use code that resembles the following:

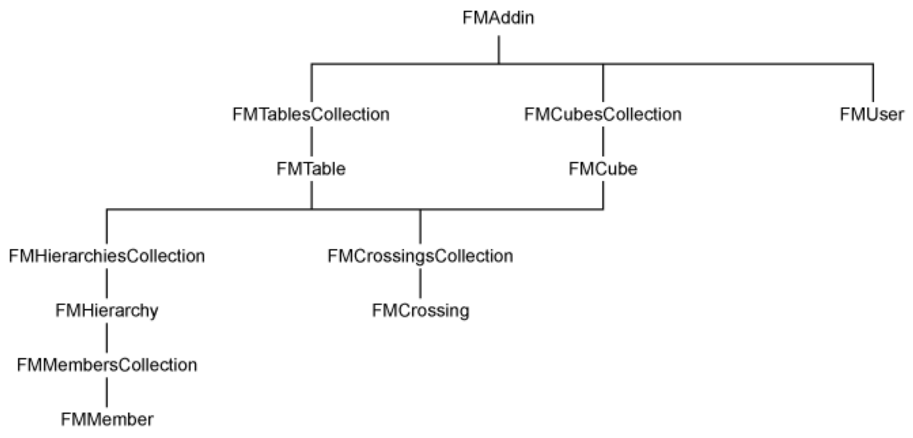
```
Dim conn As Connect
...
Set conn = Application.COMAddIns.Item("SASSESExcelAddIn.Connect").Object
Set addin = conn.FMAddIn
```

Note: For the remainder of this chapter, the code examples assume that you already have a reference (called **addin**) to the FMAddIn object. (Your code should contain only one instance of the FMAddIn object.)

From the FMAddIn object, you can get a reference to the FMTablesCollection object or to an FMCubesCollection object. The tables collection represents all tables in the workbook. Each FMTable object in the collection represents a data entry or read-only table in the current workbook. The cubes collection represents all virtual cubes (results models) on the server. Each FMCube object represents a virtual cube.

This diagram shows classes in the API. It indicates which classes contain references to other classes. (It is not intended to imply any inheritance from one class to another.)

Figure 6.1 *Classes in the SAS Financial Management Add-In API for Microsoft Excel*



Objects in a Collection

To get a reference to an object in a collection, you can specify an index into the collection. For example, **addin.Tables(0)** references the first table in an FMTablesCollection object.

You can also name an object in the collection. To get a reference to an object in the FMCubesCollection, FMHierarchiesCollection, or FMMembersCollection, you specify the code for the cube, hierarchy, or member. For example:

```
Dim cube As FMCube
Set cube = addin.cubes("Default_Model")
```

Table Objects

To get a reference to a table, use the table name (tables do not have codes). For example:

```
Dim table As FMTable
Set table = addin.Tables("NewTable0")
```

In Excel, the location of a table is defined as a named range. When you add the first table, it is automatically named **`NewTable0`**. The next table is named **`NewTable1`**, and so on.

Note: A user might change the name of a table (in the table properties), but the new name is only for display purposes and cannot be used in the code. For more information, see the getTableName method of the FMTable class.

Another approach is to iterate through the collection. This code iterates over a collection of server hierarchies in a cube:

```
For Each hierarchy In cube.ServerHierarchies
    ...
 Next hierarchy
```

Handling Events

About Events

An event is an action that happens in Excel (for example, logging in or refreshing a table). Event handlers are called when the user performs the specified action.

For an event to be captured:

- The object that the event is associated with must be declared using the **`WithEvents`** clause; for example:

  ```
  Public WithEvents addin As FMAddIn
  Public WithEvents table As FMTable
  ```

- There must be an existing reference to the object; for example:

  ```
  Set table = addin.tables("NewTable0")
  ```

Write an Event Handler

To write an event-handling procedure, use code similar to the following example, which is invoked when the user refreshes the worksheet:

```
Public Sub addin_AfterRefresh()
    MsgBox "Refresh event trapped in VBA"
End Sub
```

The name of the procedure is *object-name* + _ + *event-name*.

Be aware that an action can trigger multiple events. For example, if a user selects **SAS Solutions** ⇨ **View** ⇨ **Refresh**, the table refresh event is triggered, followed by the worksheet's refresh event. On the other hand, if a table object's Refresh method is called, or if the user performs an action that affects a single table, then only that table's refresh event is triggered.

Imagine that you want to resize the columns for a table each time the table is refreshed. To ensure that the table columns are always resized correctly, you need to add the resizing code to both the **`table0_AfterRefresh`** event handler and the **`addin_AfterRefresh`** event handler. The code might resemble the following:

Example Code 6.1 *Event Handler*

```
Public WithEvents table0 As FMTable
Public WithEvents addin As FMAddin
...
' Event handler for table0
Private Sub table0_AfterRefresh()
    ' Temporarily disable screen updating
    Application.ScreenUpdating = False

    ' Resize columns to have a uniform width
    startColumn = table0.Position(fmArea_Column, fmType_startColumn)
    endColumn = table0.Position(fmArea_Column, fmType_endColumn)
    For col = startColumn To endColumn
        Columns(col).ColumnWidth = 20
    Next col

    ' Re-enable screen updating
    Application.ScreenUpdating = True
End Sub

' addin object's AfterRefresh event handler
Private Sub addin_AfterRefresh()
    Application.ScreenUpdating = False

   ' Check to be sure this table is in the active worksheet
   If Range(table0.Name).Worksheet.Name = ActiveSheet.Name Then
        ' Resize columns
        startColumn = table0.Position(fmArea_Column, fmType_startColumn)
        endColumn = table0.Position(fmArea_Column, fmType_endColumn)
        For col = startColumn To endColumn
            Columns(col).ColumnWidth = 20
        Next col
    End If

    Application.ScreenUpdating = True
End Sub
```

To handle a table refresh that occurs when the user selects **SAS Solutions** ⇨ **View** ⇨ **RefreshAll**, you would write similar code for the **`addin_AfterRefreshAll`** event handler.

If you wanted to resize the columns of all tables to have a uniform width, then you would write an **`addin_AfterTableRefresh`** event handler, which would be called for each table that was refreshed.

For more information about specific events, see the event summaries for the FMAddin class and the FMTable class.

Activating the Log

The log for the SAS Add-In for Microsoft Office records information about queries generated via the SAS add-ins for Microsoft Office applications, including the SAS Financial Management Add-In for Microsoft Excel. You can write to the log using the traceWrite function of the FMAddin class.

By default, this log is disabled. For information about activating the log, see SAS Usage Note 19846 at **`http://support.sas.com/kb/19/846.html`**. For information about directing the log output to a file, see SAS Usage Note 38063 at `http://support.sas.com/kb/38/063/html`.

Note: To prevent the log file from becoming too long, we recommend that you specify a DebugLevel no higher than `2` in the configuration file.

Summary of Classes

The following table summarizes the classes that make up the API.

Table 6.1 *Summary of Classes*

Class	Description
FMAddIn	The top-level class for manipulating the add-in.
FMCollections	Base class for other collections such as FMCrossingsCollection and FMTablesCollection. Its properties and methods are inherited by these subclasses.
FMCrossing	Provides access to the properties of a crossing in a table or cube.
FMCrossingsCollection	Represents a collection of crossings.
FMCube	Represents a virtual cube (results model).
FMCubesCollection	Represents a collection of cubes.
FMHierarchy	Represents a hierarchy.
FMHierarchiesCollection	Represents a collection of hierarchies.
FMMember	Represents a member of a hierarchy.
FMMembersCollection	Represents a collection of members.
FMTable	Represents a table.
FMTablesCollection	Represents a collection of tables.
FMUser	Represents the user who is currently logged on.

The FMAddIn Class

The FMAddIn class is the top-level class in the API. From the FMAddIn object, you can get a reference to the tables in the workbook, the cubes that are on the server, and the current user.

Table 6.2 *FMAddIn Property Summary*

Property	**Description**
Property **Cubes** As FMCubesCollection	A collection of cubes that are on the server (and that you have access to). Read-only.
Property **isLoggedIn** As Boolean	If **True**, the user is logged on. Read-only.
Property **MessageBoxEnabled** As Boolean	If **False**, pop-up messages are disabled from the SAS Financial Management Add-In. Typically, you would set this property to **False** when you are running in batch mode. The default is **True**. Read-write.
Property **MessageBoxResponseOK** As Boolean	The default response to any suppressed message boxes. This property applies only if MessageBox Enabled is set to **False**. A value of **True** sets the default response to **Yes** or **OK**. A value of **False** sets the default response to **No** or **Cancel**. The default is **True**. Read-write.
Property **Port** As Long	The port number of the middle-tier server. Read-only.
Property **ReadOnly** As Boolean	This property applies if the user is viewing a data-entry form. If **True**, the form cannot be edited.
Property **Secure** As Boolean	**True** if the middle-tier server is using the Secure Sockets Layer (SSL) protocol. Otherwise **False**. Read-only.
Property **Server** As String	The name of the middle-tier server. Read-only.
Property **Tables** As FMTablesCollection	A collection of tables. Read-only.
Property **Url** As String	The URL to the middle-tier server on which SAS Financial Management is running. Read-only.
Property **User** As FMUser	A FMUser object that represents the user who is currently logged on. Read-only.
Property **Version** As String	The name and version number of this software. Read-only.
Property **VersionDate** As String	The date of this version of the software. Read-only.
Property **VersionID** As String	The version number of this software. Read-only.

***Table 6.3** FMAddIn Class Method Summary*

Function **`enumString`** (fmEnum As fmEnums, enumValue As Long) As String	Returns the String equivalent of an enumerated constant—for example, the value returned from a write operation, the name of a role, or an area of the table.
	Parameters:
	• *fmEnum*: the type of enumerated constant. This parameter can be one of the following: **`fmBudgetMode`**, **`fmDisplayMode`**, **`fmRole`**, **`fmType`**, **`fmArea`**, **`fmSelection`**, **`fmCreditsDebitsDisplay`**, or **`fmWriteBackReturn`**.
	• *enumValue*: the value to be converted into a string.
	Returns: a string that corresponds to *enumValue* for the specified type of constant.
	Many methods take enumerated constants as parameters or return them as return values. The Write method returns an enumerated constant (a numeric value). You can declare the variable that you are using for the return value as an enumerated constant and then access its string representation. This code fragment displays a message box for a write operation that failed, with the reason for the failure:
	`Dim rc As fmWriteBackReturn`
	`...`
	`Set crossing = addin.Tables(0).crossing(4, 3)`
	`rc = crossing.Write(111)`
	`If rc <> fmWriteBackReturn_Succeeded Then`
	`   MsgBox "return from write: " & _`
	`     addin.enumString(fmWriteBackReturn, rc)`
	`End If`

Function **`findTable`** (sheetName As String, row As Long, column As Long) As FMTable	Finds the table object that corresponds to the specified sheet and position. Parameters: • *sheetname*: the name of a sheet in the workbook. • *row*, *column*: the position of a table element. Returns: an FMTable object.
Function **`getTableName`** (username As String) As String	Returns the internal name of the table that corresponds to a name in the table properties. By default, the first table a user inserts is named **`NewTable0`**, the second table is **`NewTable1`**, and so on. The user might rename the table in the table properties. However, the new name is only a display name. The code requires the original name, which is available via the getTableName function. Parameters: • *username*: the table name in the table properties. Returns: the original table name.

Function **`Login`** (environment As String, username As String, password As String) As Boolean	Logs the user on to the middle tier. If the user is already logged on, this function returns **`True`** even if the parameter values are incorrect. Parameters: • *environment*, *username*, and *password*: the environment, user name, and password for logging on to the middle tier. These parameters are the same values that you would use to log on from the **SAS Solutions** menu in Excel. The environment value is site-specific. Environments are defined in the EnvironmentFactory.xml file. For more information, see "About the SAS Environment Files" on page 20. We recommend generating an encoded or encrypted password that you can copy and paste into your code, rather than using a plain-text password. For more information, see the *SAS Intelligence Platform: Security Administration Guide*. Returns: **`True`** if the user is already logged on or if the login succeeds; otherwise, **`False`**.
Function **`Logoff`**() As Boolean	Logs the user off the middle tier. Returns: **`True`** if the action succeeded; otherwise, **`False`**.
Function **`Refresh`**() As Boolean	Refreshes the selected worksheet. This action is similar to the **Refresh** action from the **SAS Solutions** menu. Returns: **`True`** if the action succeeded; otherwise, **`False`**.
Function **`RefreshAll`**() As Boolean	Refreshes all open worksheets in the selected file. This action is similar to the **Refresh All** action from the **SAS Solutions** menu. Returns: **`True`** if the action succeeded; otherwise, **`False`**.
Function **`traceWrite`** (traceString As String) As Boolean	Writes the contents of *traceString* to the log for SAS Add-In for Microsoft Office. This method is helpful in debugging your code. Parameters: • *traceString*: the string to write. By default, the log is disabled. See "Activating the Log" on page 79.

Table 6.4 *FMAddin Event Summary*

Event	Description
Event **`AfterLogOff`**()	Triggered after the user logs off from the middle tier. A logoff event occurs when there is a call to the Logoff method of the FMAddin object or when the user selects **Log Off** from the **SAS Solutions** menu.
Event **`AfterLogon`**()	Triggered after the user has logged on. This event occurs when there is a call to the Login method of the FMAddin object, when the user selects **Log On** from the **SAS Solutions** menu, or when the user opens an Excel report from the portal.
Event **`AfterRefresh`**()	Triggered after a refresh action—for example, if there is a call to **`addin.Refresh()`** or if the user selects **SAS Solutions** ⇨ **View** ⇨ **Refresh**.

Event	**Description**
Event **`AfterRefreshAll`**()	Triggered if there is a call to **`addin.RefreshAll()`** or if the user selects **SAS Solutions ⇨ View ⇨ RefreshAll**.

Event **`AfterTableRefresh`** (table As FMTable)	Triggered after a table has been refreshed. This event might occur if there is a call to the **`Refresh`** method of a table object, if the user selects **Refresh** or **RefreshAll** from the **View** menu, or if the user performs some other manual action, such as a pivot, that triggers a refresh.
	
	If the user refreshes a worksheet, the table refresh event and the addin refresh event are triggered, in that order.
	
	Parameters:
	
	• *table*: an FMTable object that represents the table that was refreshed. If you refresh a worksheet that contains multiple tables, the AfterTableRefresh event is triggered multiple times, once for each table.
	
	This event handler displays a message that includes the name of the table that was refreshed:
	
	`Private Sub addin_AfterTableRefresh(ByVal table As FMTable)`
	`   txt = "Addin afterRefresh: " + table.Code`
	`   MsgBox txt`
	`End Sub`

Event **`BeforeLogOff`**()	Triggered when logoff has been requested but before the user logs off. This event handler returns a Boolean. If the return value is **`True`**, the logoff continues. If the return value is **`False`**, the logoff is canceled. Here is an example:
	
	`Private Function addin_BeforeLogOff() As Boolean`
	`   response = MsgBox("Do you really want to log off?", _`
	`      vbOKCancel, "SAS Financial Management")`
	`   If response = vbOK Then`
	`      addin_BeforeLogOff = True`
	`   Else`
	`   ' Cancel the logoff process`
	`      addin_BeforeLogOff = False`
	`   End If`
	`End Function`

Event **`BeforeTableRefresh`** (table As FMTable)	Triggered before a table is refreshed. You might use this event handler to disable screen updating while you are making modifications to the screen. In the AfterTableRefresh event handler, you could re-enable screen updating. Parameters: • *table*: an FMTable object that represents the table that was refreshed. If you refresh a worksheet that contains multiple tables, the BeforeTableRefresh event is triggered multiple times, once for each table.

The FMCollections Class

The FMCollections class is the base class for several other collections: FMCrossingsCollection, FMCubesCollection, FMHierarchiesCollection, FMMembersCollection, and FMTablesCollection. Its properties and methods for manipulating a collection are inherited by these subclasses.

Note: Do not invoke this class directly. Instead, use one of its subclasses.

Table 6.5 *FMCollections Class Property Summary*

Property	**Description**
Property **`Count`** As Long	The number of items in the collection. Read-only.

Table 6.6 *FMCollections Class Method Summary*

Class	**Description**

Sub **`Add`** (item)	Adds a single item to the collection.
	Parameters:
	• *item*: the item to add.
	This example creates a FMMembersCollection object and adds two members of the **`ACCOUNT.AccountType`** hierarchy to the collection:
	`Dim hierarchy As FMHierarchy`
	`Dim excmems As New FMMembersCollection`
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT.AccountType")`
	`Call excmems.Add(hierarchy.Members("StatisticalBalance"))`
	`Call excmems.Add(hierarchy.Members("Equity"))`

Sub **`AddAll`**(item)	Adds a collection of items to the collection.
	Parameters:
	• *item*: a collection of items to add (for example, an FMMembersCollection object that represents a collection of members).
	This example creates a FMMembersCollection object and adds all the members of the **`ACCOUNT.AccountType`** hierarchy to the collection:
	`Dim hierarchy As FMHierarchy`
	`Dim mems As New FMMembersCollection`
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT.AccountType")`
	`Call mems.AddAll(hierarchy.Members)`

Sub **`Clear`**()	Clears the collection.
Function **`Contains`** (item) As Boolean	Returns: **`True`** if the collection contains the specified item. Parameters: • *item*: a single item (for example, an FMMember object if you are searching an FMMembersCollection instance).
Function **`IndexOf`** (item) As Long	Returns: the zero-based index (position) of the specified item in the collection, or **`-1`** if the item is not found. Parameters: • *item*: an object of the collection type (for example, an FMMember object).

Class	Description
Sub **`Insert`** (index As Long, item)	Inserts *item* at the *index* position in the collection. Parameters: • *index*: a (zero-based) index into the collection. • *item*: an object of the collection type.
Sub **`InsertAll`** (index As Long, item)	Inserts a collection at the *index* position in the collection. Parameters: • *index*: a (zero-based) index into the collection. • *item*: an object that represents a collection (for example, an FMMembersCollection object).
Sub **`Remove`** (item)	Removes an object from a collection (if the object is found). Parameters: • *item*: an item in a collection.
Sub **`RemoveAt`** (index As Long)	Removes the item at the *index* position in the collection. Parameters: • *index*: a (zero-based) index into the collection.
Function **`ToString`** () As String	Returns: a string that represents the concatenated codes of all the elements in the collection.

The FMCrossing Class

The FMCrossing class provides access to the properties of a crossing. In a table, a crossing is determined by its position in the table (row and column). In a cube, a crossing is determined by a two-dimensional String array of dimension codes and member codes. For examples, see the FMTable class and the FMCube class.

Table 6.7 *FMCrossing Class Property Summary*

Property	Description
Property **`Code`** As String	An identifier for this crossing. For table crossings, the code is a string that contains information about the row and column for the crossing. For crossings in a cube, the code is a concatenated string of model code, dimension codes, and member codes, such as the following: `DefaultModel_ACCOUNT_NETINCOME_TIME_JAN2003_ANALYSIS_BUDGET...` Read-only.
Property **`Column`** As Long	The column position of this crossing. Applies only to tables. Read-only.
Property **`ColumnRelative`** As Long	The column position of this crossing, relative to the leftmost column of the table (*crossingColumn* - *firstColumn* + 1). Applies only to tables. Read-only.

Property	Description
Property **`DimensionMembers`** As String()	A two-dimensional array of strings that contain the dimensions and members that apply to this crossing, in the form (*dimension*, *member*). Read-only.
Property **`Length`** As Long	The number of dimensions in this crossing. Read-only.
Property **`NewValue`** As Double	For a cube or a table, the NewValue is the value that is written to the server for this crossing when the BatchWrite method is called. Read/write.
Property **`Row`** As Long	The row position of this crossing. Applies only to tables. Read-only.
Property **`RowRelative`** As Long	The row position of this crossing, relative to the topmost row of the table (*crossingRow* - *firstRow* + 1). Applies only to tables. Read-only.
Property **`ScaledValue`** As Double	For tables, this property contains the value of the crossing divided by the current scale of the table. This value is similar to the value that is shown in the table. For cubes, this property contains the value of the crossing (it is identical to the Value property). Read-only.
Property **`Value`** As Double	The value of this crossing, before any table scaling is applied. Read-only.
Property **`Writeable`** As Boolean	If **`True`**, the crossing is writable. Read-only.

Table 6.8 *FMCrossing Class Method Summary*

Method	Description
Function **`GetMember`** (item As String) As FMMember	Returns the member of this crossing for the specified dimension code. Parameters: • *item*: a dimension code, such as **`ACCOUNT`** or **`ORG`**.
Function **`GetMemberCode`** (item As String) As String	Returns the member code in this crossing for the specified dimension code. Parameters: • *item*: a dimension code.
Function **`Write`** (value As Double) As fmWriteBackReturn	Writes *value* to this crossing. Returns: the status of the write operation, which can be one of the following: **`fmWriteBackReturn_Succeeded`**, **`fmWriteBackReturn_FailedCantUpdateForm`**, **`fmWriteBackReturn_FailedReadOnly`**, **`fmWriteBackReturn_FailedNoValueChange`**, **`fmWriteBackReturn_FailedNoValue`**, **`fmWriteBackReturn_FailedNoXRate`**, or **`fmWriteBackReturn_FailedUnknown`**.

The FMCrossingsCollection Class

The FMCrossingsCollection class represents a collection of crossings. This class is a subclass of FMCollections.

Properties inherited from FMCollections: Count.

Methods inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

The FMCube Class

An FMCube object represents a virtual cube (results model). With the FMCube class, you can access metadata from the Solutions data mart. An instance of the FMCube class can be the entry point for metadata about results models, hierarchies, and members. With FMCube methods, you can also read and write facts.

The FMCube class works independently of read-only tables, data entry tables, and CDA expressions. As a result, your code is able to interact with metadata and with facts.

To perform a query for a cube:

1. Get a reference to the cube.
2. Get a reference to the cube's crossings collection, which is empty to begin with.
3. Get the crossings for a particular set of (*dimension code*, *member code*) values, and add them to the cube's crossings collection.

   At this point, you have the metadata for the crossings, but you have no corresponding values.
4. Call the cube's ExecuteQuery method to get the values for each crossing in the collection.

The (*dimension code*, *member code*) values in the cube's crossings collection are the parameters for the query. If you omit a dimension from the set of parameters, the model's default read member for that dimension is used.

This example creates a set of query parameters, performs a query, and displays the results.

Example Code 6.2 *Query on a Cube*

```
Public addin As FMAddIn
Dim crossing As FMCrossing
Dim crs As FMCrossing
Dim crossings As FMCrossingsCollection
Dim cube As FMCube
Dim member As FMMember

Public Sub testCube()
    Set Connection = _
       Application.COMAddIns.Item("SASSESExcelAddIn.Connect").Object
    Set addin = Connection.FMAddIn

    If addin.IsLoggedIn = False Then
```

```
        MsgBox "Please log in..."
        Exit Sub
    End If

    ' Get reference to cube
    Set cube = addin.Cubes("Default_Model")

    ' Specify dimension, member pairs
    ' to be used as default parameters for query
    Dim dm() As String
    ReDim dm(9, 1)
    dm(0, 0) = "ACCOUNT"
    dm(0, 1) = "A6520"
    dm(1, 0) = "TIME"
    dm(1, 1) = "JAN2003"
    dm(2, 0) = "FREQUENCY"
    dm(2, 1) = "PA"
    dm(3, 0) = "ORG"
    dm(3, 1) = "BMRM"
    dm(4, 0) = "ANALYSIS"
    dm(4, 1) = "BUDGET"
    dm(5, 0) = "COUNTRY_D"
    dm(5, 1) = "WW.vc"
    dm(6, 0) = "TRADER"
    dm(6, 1) = "EXT"
    dm(7, 0) = "PERIODS"
    dm(7, 1) = "AP.vc"
    dm(8, 0) = "PRODUCT"
    dm(8, 1) = "B0815"
    dm(9, 0) = "CURRENCY"
    dm(9, 1) = "USD"

    ' Get reference to crossings collection for this cube
    ' (Collection is currently empty.)
    Set crossings = cube.crossings

    ' Get reference to crossings for YR2001 in TIME dimension
    ' and add to crossings collection
    For Each member In cube.Hierarchies("TIME").GetMembers("YR2001", True, False)
        ' Replace member code for TIME dimension in array
        dm(1, 1) = member.Code
        ' Get the crossing for this member
        Set crossing = cube.crossing(dm)
        ' Add this crossing to the collection
        Call crossings.Add(crossing)
    Next member

    ' Execute query to fetch values for each crossing in collection
    cube.ExecuteQuery

    ' Display values for each member of TIME dimension
    For Each crs In cube.crossings
        MsgBox crs.GetMemberCode("TIME") + " = " + Str(crs.Value)
    Next crs
End Sub
```

After you perform a query, the values that the query returns are available locally. Before performing any additional queries, you would call the cube's ClearQuery method and then define the parameters for the new query.

To write values to a cube, you can call the cube's Write method with the crossing and new value as arguments, or you can set the NewValue property for each crossing that you want to affect and then call the cube's BatchWrite method. Here is an example of a batch write for a cube.

Example Code 6.3 *Batch Write for a Cube*

```
Dim dm() As String
ReDim dm(9, 1)
dm(0, 0) = "ACCOUNT"
dm(0, 1) = "A6520"
dm(1, 0) = "TIME"
dm(1, 1) = "JAN2003"
...
dm(9, 0) = "CURRENCY"
dm(9, 1) = "USD"

Set crossings = cube.crossings

' Create new crossings and add to cubes crossings collection
For Each member In cube.Hierarchies("TIME").GetMembers("YR2002", True, False)
   ' One crossing for each month in year 2002
   dm(1, 1) = member.Code
   Set crossing = cube.crossing(dm)
   Call crossings.Add(crossing)
   crossing.newValue = newValue
Next member

cube.ExecuteQuery

' Write new values
rc = cube.BatchWrite(True)
```

Table 6.9 *FMCube Class Property Summary*

Property	Description
Property **`Code`** As String	The code for this cube (for example, **`Default_Model`**). Read-only.
Property **`Crossings`** As FMCrossingsCollection	The collection of crossings in this cube. Read-only.
Property **`CurrencyHierarchy`** As FMHierarchy	An FMHierarchy object that contains the currency hierarchy for this cube. Read-only.
Property **`Description`** As String	The description of this cube. Read-only.

Property	Description
Property **`Hierarchies`** As FMHierarchiesCollection	A collection of hierarchies (both server hierarchies and client attribute hierarchies) in this cube. Read-only. Server hierarchies are hierarchies that are defined on the server and that are being used in the specified results model, either directly (via an FMCube object) or via a virtual cube that is attached to an FMTable object. They are based on the required dimensions (such as **`ACCOUNT`**, **`ANALYSIS`**, and **`TIME`**) and any custom defined dimensions (such as **`PRODUCT`** or **`COSTCENTER`**). Client attribute hierarchies are virtual hierarchies that exist only on the client side. They are based on dimension attributes (both system properties and custom properties).
Property **`Id`** As Long	A unique numeric identifier from the Solutions data mart. Read-only.
Property **`Index`** As Long	The position of this cube within the cubes collection. Read-only.
Property **`Name`** As String	The name of this cube. Read-only.
Property **`ServerHierarchies`** As FMHierarchiesCollection	The collection of server hierarchies that are associated with the cube. Read-only. For more information about server hierarchies, see the description of the Hierarchies property.

Table 6.10 *FMCube Class Method Summary*

Method	Description
Function **`BatchWrite`** (reQuery As Boolean) As fmWriteBackReturn	Writes the accumulated transactions to the server. Parameters: • *reQuery*: the requery flag. If **`True`**, the function performs a requery and repaint after the write operation. If **`False`**, the function performs a repaint only. Returns: **`fmWriteBackReturn_Succeeded`**, **`fmWriteBackReturn_FailedCantUpdateForm`**, **`fmWriteBackReturn_FailedReadOnly`**, **`fmWriteBackReturn_FailedNoValueChange`**, **`fmWriteBackReturn_FailedNoValue`**, **`fmWriteBackReturn_FailedNoXRate`**, or **`fmWriteBackReturn_FailedUnknown`**. *Note:* The BatchWrite method does not honor driver formulas. As an alternative, you can include the driver formula calculations in your code or execute the **Run driver formula** function on the form set. See the online Help for SAS Financial Management Studio, or the description of the generateFormulaFacts method in "The Model Class" on page 37.
Function **`BatchWriteNew`** (reQuery As Boolean) As fmWriteBackReturn	The BatchWriteNew function behaves like the BatchWrite function. The difference is that BatchWriteNew returns failure codes for each cell write failure, whereas BatchWrite returns only the last failure code.
Sub **`ClearQuery`** ()	Clears the query definitions on the cube.

Method	Description
Function **`Crossing`** (item) As FMCrossing	Returns the crossing that is represented by *item*. Parameters: • *item*: an array of *dimension code*/*member code* value pairs.
Sub **`ExecuteQuery`** ()	Queries the server for all crossings that are defined for the cube.
Function **`isWriteable`**() As Boolean	Returns **`True`** if the cube is writeable.
Function **`Write`** (crossing As FMCrossing, newValue As Double) As fmWriteBackReturn	Writes *newValue* to the specified *crossing*. Parameters: • *crossing*: an FMCrossing object. • *newValue*: the value to write. Returns: **`fmWriteBackReturn_Succeeded`**, **`fmWriteBackReturn_FailedCantUpdateForm`**, **`fmWriteBackReturn_FailedReadOnly`**, **`fmWriteBackReturn_FailedNoValueChange`**, **`fmWriteBackReturn_FailedNoValue`**, **`fmWriteBackReturn_FailedNoXRate`**, or **`fmWriteBackReturn_FailedUnknown`**.

The FMCubesCollection Class

The FMCubesCollection class represents a collection of cubes that are available on the server (and that the user has permission to access). This class is a subclass of FMCollections.

Properties inherited from FMCollections: Count.

Methods inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

The FMHierarchiesCollection Class

The FMHierarchiesCollection class represents a collection of hierarchies. This class is a subclass of FMCollections.

Properties inherited from FMCollections: Count.

Methods inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

The FMHierarchy Class

The FMHierarchy class has properties and methods for accessing members of a hierarchy. For more information about types of hierarchies, see the Hierarchies property of the FMCube class or the FMTable class.

One method to note in the FMHierarchy class is the ShowMemberSelectionDialog method. This method displays a dialog box from which users can select a member of a specified hierarchy. The following example presents a dialog box from which the user can select a member of the TIME hierarchy.

Example Code 6.4 *Selecting a Hierarchy Member*

```
Dim cube As FMCube
Dim hier As FMHierarchy
Dim selmem As FMMember
Dim premem As FMMember
Dim exclude As FMMembersCollection
...
' Get an instance of the results model /cube + hierarchy
Set cube = addin.Cubes("Default_Model")
Set hier = cube.Hierarchies("TIME")

' Set a preselected member
Set premem = hier.Members("YR2005")
' ... or use the default
' Set selmem = Nothing

' Prepare a list of members to exclude from the dialog
Set exclude = New FMMembersCollection

' Add YR1997 and all descendants of YR1997 to the list
Call exclude.Add(hier.Members("YR1997"))
For Each member In hier.GetMembers("YR1997", True, False)
    Call exclude.Add(member)
Next member

' Display the dialog with preselected member and exclusion list
Set selmem = hier.ShowMemberSelectionDialog(premem, exclude, _
    fmDisplayMode_CodeAndDescription)
' ...or use default member and no exclusion list
' Set selmem = hier.ShowMemberSelectionDialog(Nothing, Nothing, _
    fmDisplayMode_CodeAndDescription)
```

The ShowMemberSelectionDialog method displays the Select Member dialog box. In this case, **`YR2005`** would be pre-selected, and **`YR1997`** and its descendants would be excluded.

The user selects a member of the hierarchy and clicks **OK**. The return value is the selected member.

Table 6.11 FMHierarchy Class Property Summary

Property	Description

Property **`Asof`** As Double	The as-of date for this hierarchy. Read-only.
	To view this date as an Excel date, store it in a Date field. For example:
	`Dim hier as FMHierarchy Dim dt as Date`
	`...`
	`dt = hier.Asof MsgBox Str(dt)`

Property **`AvailableMembers`** As FMMembersCollection	A list of the hierarchy members that are available after the member selection rules have been applied. Read-only.
Property **`Code`** As String	The dimension code that applies to this hierarchy. Read-only.
Property **`Description`** As String	The description of this hierarchy. Read-only.
Property **`DimensionCode`** As String	The dimension code that applies to this hierarchy. Read-only.
Property **`DimensionDescription`** As String	The dimension description that applies to this hierarchy. Read-only.
Property **`DimensionId`** As Long	The dimension ID that applies to this hierarchy. Read-only.
Property **`DimensionName`** As String	The dimension name that applies to this hierarchy. Read-only.
Property **`DimensionTypeCode`** As String	The dimension type code that applies to this hierarchy. Read-only.
Property **`DimensionTypeDescription`** As String	The dimension type description that applies to this hierarchy. Read-only.
Property **`DimensionTypeID`** As Long	A unique numeric identifier from the Solutions data mart. Read-only.
Property **`DimensionTypeName`** As String	The dimension type name that applies to this hierarchy. Read-only.

Property **`DisplayedMembers`** As FMMembersCollection	A collection of the hierarchy members that are currently being displayed. Read-only.
	This example creates a collection of the members of the ACCOUNT hierarchy that are currently displayed in the specified table and displays the results in a message box:
	`Dim txt as String`
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT")`
	`txt = "All displayed members in " & hierarchy.Description _`
	`   & Chr$(10)`
	`For Each member In hierarchy.DisplayedMembers`
	`   txt = txt & " " & member.Code`
	`Next member`
	`MsgBox txt`

Property	Description
Property **`DisplayMode`** As fmDisplayMode	The labeling method for this hierarchy, which specifies the way displayed members are identified. The value can be one of the following: **`fmDisplayMode_Code`**, **`fmDisplayMode_Name`**, **`fmDisplayMode_Description`**, **`fmDisplayMode_CodeAndName`**, or **`fmDisplayMode_CodeAndDescription`**. Read/write.
Property **`HierarchyCode`** As String	The code for this hierarchy. Read-only.
Property **`HierarchyIndex`** As Long	For a table, this value represents the index of this hierarchy in the set of dimensions that make up the query for the table. For cubes, this value is always **`-1`**. Read-only.
Property **`ID`** As Long	A unique numeric identifier from the Solutions data mart. Read-only.
Property **`LeafMembers`** As FMMembersCollection	A collection of the leaf members of this hierarchy. Read-only.
Property **`Members`** As FMMembersCollection	A collection of the members of this hierarchy. Read-only.
Property **`Name`** As String	The name of this hierarchy. Read-only.
Property **`Position`** As Long	The position of this hierarchy within its section. The section is determined by the Role property. (See below.) If the hierarchy is in the **Available** list, its position is **`-1`**. Read/write.
Property **`ReadableMembers`** As FMMembersCollection	A collection of hierarchy members that are readable by the current user. Read-only.
Property **`ReadDefaultMember`** As FMMember	The default Read member for this hierarchy. Read/write.

Property **`Role`** As fmRole	The role of this hierarchy. Read/write.
	The role determines the section in which the hierarchy appears. It can have one of the following values:
	• **`fmRole_Row`**: row
	• **`fmRole_Column`**: column
	• **`fmRole_Slicer`**: slicer
	• **`fmRole_Available`**: available for use in a row, column, or slicer
	This example performs a pivot of a table by changing the Role and Position properties of a Hierarchy object. When the code has been executed, the ACCOUNT hierarchy is the first column heading in the table.
	`Set table = addin.Tables("NewTable0")`
	`Set hierarchy = table.Hierarchies("ACCOUNT")`
	`hierarchy.role = fmRole_Column hierarchy.Position = 0`
	`' Refresh with a requery`
	`table.Refresh (True)`

Property	Description
Property **`TargetMember`** As FMMember	The hierarchy member that a form is assigned to. Read-only.
Property **`VCFilter`** As Boolean	If **`True`**, the table or cube is filtered so that virtual children are not included. If **`False`**, virtual children are included. Read/write.
Property **`WriteDefaultMember`** As FMMember	The default Write member for this hierarchy. Read/write.

Table 6.12 *FMHierarchy Class Method Summary*

Method	Description

Function **`ChangeSlicer`** (item) As Boolean	Changes the hierarchy member that is used as the slicer. For example, if you are using a member of the TIME hierarchy as a slicer, you might change from one year to another. (The hierarchy must already be functioning as a slicer. In other words, if the hierarchy is being used in a column or row or is simply available for use, the ChangeSlicer method will not work.)
	Parameters:
	• *item*: the member of the hierarchy that will become the new slicer. It can be an index into the hierarchy or a member code.
	Returns: **`True`** if the change succeeded; otherwise, **`False`**.
	In this example, a member of the ACCOUNT hierarchy is being used as a slicer. The code changes the member to **`A1000`**:
	`Set hierarchy = addin.Tables(0).Hierarchies("ACCOUNT")`
	`rc = hierarchy.ChangeSlicer("A1000")`

Method	**Description**

Function **`GetMembers`** (item, recurse As Boolean, reverse As Boolean) As FMMembersCollection	Returns a collection of members of this hierarchy, beginning with the first child of the member that is selected by *item*.
	Parameters:
	• *item*: the hierarchy member on which to begin processing. This parameter can be an index into the hierarchy or a member code.
	• *recurse*: the recursion flag. If **`True`**, the function performs a recursive search and returns all descendants. If **`False`**, it returns only the member's children.
	• *reverse*: the reverse order flag. If **`True`**, the function returns the results in reverse order. If **`False`**, the children are returned in the same order in which they appear in the hierarchy.
	This example returns all descendants of the first member of the ACCOUNT hierarchy for a table:
	`Dim txt As String`
	`Dim member As FMMember`
	`Set hierarchy = addin.Tables("NewTable0").Hierarchies("ACCOUNT")`
	`txt = "All descendants of " _`
	`   & hierarchy.Members(0).Description & " in " _`
	`   & hierarchy.Description & " hierarchy" & Chr$(10)`
	`For Each member In hierarchy.GetMembers(0, True, False)`
	`   txt = txt & " " & member.Code`
	`Next member`
	`MsgBox txt`

Function **`IsFlatDimensionType`** () As Boolean	Returns: **`True`** if this hierarchy belongs to a flat dimension type; otherwise, **`False`**. Read-only. There are three dimension types that must have flat hierarchies: ANALYSIS, CURRENCY, and FREQUENCY. In addition, a client attribute hierarchy is a flat dimension type. (For more about client attribute hierarchies, see the description of the FMTable.Hierarchies property at "The FMTable Class" on page 100.)
Function **`IsNonVirtualChildDimensionType`** () As Boolean	Returns: **`True`** if the hierarchy does not include virtual children; otherwise, **`False`**. Read-only.
Function **`IsServer`** () As Boolean	Returns: **`True`** if this hierarchy is defined on the server.

Method	Description
Function **`ShowMemberSelectionDialog`** (item, exclude As FMMembersCollection, displayMode As fmDisplayMode) As FMMember	Displays a dialog box from which users can select a member of the specified hierarchy. Parameters: • *item*: the member that you want to be highlighted in the dialog box. • *exclude*: a collection of members to be excluded from the dialog box. To display all members, create a collection but do not assign it any values, as in this example: **`Dim excmems as New FMMembersCollection`** • *displayMode*: the way displayed members are identified (the labeling method). The value can be one of the following: **`fmDisplayMode_Code`**, **`fmDisplayMode_Name`**, **`fmDisplayMode_Description`**, **`fmDisplayMode_CodeAndName`**, or **`fmDisplayMode_CodeAndDescription`**. Returns: the selected member of the hierarchy, which can be used in several ways. For example, the selected member could be used to change the slicer for a table. It could also be used as input to code that modifies a CDA expression.

The FMMember Class

The FMMember class represents a member of a hierarchy, which can be displayed or not. For members of displayed hierarchies, the selection rules can be modified.

Table 6.13 *FMMember Class Property Summary*

Property	Description
Property **`Asof`** As Double	The as-of date for this member. Read-only. To view this date as an Excel date, store it in a Date field.
Property **`Code`** As String	The code for this hierarchy member. Read-only.
Property **`Column`** As Long	The column position of the top left cell of this member's position in the table. Read-only.
Property **`Description`** As String	The description of a member of a hierarchy. Read-only.
Property **`ID`** As Long	A unique numeric identifier from the Solutions data mart. Read-only.
Property **`Level`** As Long	The level of this member in the current hierarchy. The top member of the hierarchy has a level of `0`. Read-only.
Property **`Name`** As String	The name of a member of a hierarchy. Read-only.

Property	**Description**
Property **`Row`** As Long	The row position of the top left cell of this member's position in the table. Read-only.
Property **`SelectionRule`** As fmSelection	Gets or sets the selection rule for a displayed hierarchy member. Read/write. The value can be one of the following enumerated constants: **`fmSelection_Member`**: selects the designated member. **`fmSelection_Descendants`**: selects the entire subhierarchy subordinate to the designated member but not including the designated member itself. **`fmSelection_MemberAndChildren`**: selects the designated member and all members that are immediately subordinate to it. **`fmSelection_MemberAndDescendants`**: selects the entire subhierarchy from the designated member down. **`fmSelection_MemberAndLeaf`**: selects the designated member and all members that are subordinate to it but that have no members under them. For example, in a Time hierarchy that included years, quarters, and months, this value would select year and months, but not quarters. **`fmSelection_Children`**: selects all members that are immediately subordinate to the designated member. **`fmSelection_Leaf`**: selects all members that are subordinate to the designated member but have no members subordinate to them. **`fmSelection_NoMember`**: excludes the designated member from the subset. All members that are subordinate to the designated member are also excluded, unless you apply additional rules to one or more of these subordinate members. **`fmSelection_NoRule`**: removes any rules from the designated member.

	This code modifies selection rules for the **`ACCOUNT`**, **`ANALYSIS`**, and **`ORG`** hierarchies in a table:
	`Set addin = conn.FMAddIn`
	`Set table = addin.Tables("NewTable0")`
	
	`Set hier = table.Hierarchies("ACCOUNT")`
	`hier.Members("A8000").SelectionRule = fmSelection_NoMember`
	`hier.Members("A7400").SelectionRule = fmSelection_Member`
	`hier.Members("A6300").SelectionRule = fmSelection_NoMember`
	`hier.Members("A7800").SelectionRule = _`
	`   fmSelection_MemberAndDescendants`
	`hier.Members("A7900").SelectionRule = fmSelection_NoMember`
	`hier.Members("A8100").SelectionRule = fmSelection_Member`
	
	`Set hier = table.Hierarchies("ANALYSIS")`
	`hier.Members("ACTUAL").SelectionRule = fmSelection_Member`
	`hier.Members("BUDGET").SelectionRule = fmSelection_NoMember`
	
	`Set hier = table.Hierarchies("ORG")`
	`hier.Members("PLD").SelectionRule = fmSelection_Member`
	
	`' Refresh the table to see the results`
	`table.Refresh (True)`

Table 6.14 *FMMember Class Method Summary*

Method	Description
Sub **Collapse**()	Collapses the member to hide all its descendants.
Sub **Expand** ()	Expands the member to display all its children.
Sub **ExpandAll** ()	Expands the member to display all its descendants.
Function **getParent**() As FMMember	Returns: the parent of this member. If this member is a top-level member, it returns this member.
Function **IsClientAttributeFilter**() As Boolean	Returns: **True** if this member is a member attribute filter.
Function **IsClientCalculatedMember**() As Boolean	Returns: **True** if this member is a client calculated member.
Function **IsLeaf**() As Boolean	Returns: **True** if this member is a leaf member.
Function **IsReadable**() As Boolean	Returns: **True** if this member is readable by the current user.
Function **IsServer**() As Boolean	Returns: **True** if this member is defined on the server.
Function **IsVirtual**() As Boolean	Returns: **True** if this member is a virtual child.
Function **IsWriteable**() As Boolean	Returns: **True** if this member is writable by the current user.

The FMMembersCollection Class

The FMMembersCollection class represents members of a hierarchy. This class is a subclass of FMCollections.

Properties inherited from FMCollections: Count.

Methods inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

The FMTable Class

An FMTable object represents a read-only table or a data entry table. Some elements, such as layout, scale, and result model, can be manipulated directly on the table object. Other operations, such as filtering virtual children and showing or hiding members, must be manipulated on the FMHierarchy or FMMember objects that belong to the table.

Table 6.15 *FMTable Class Property Summary*

Property	**Description**
Property **`Code`** As String	The code for this table. Read-only.
Property **`Credit`** As fmCreditsDebitsDisplay	The manner in which credit values are displayed in this table. Read/write. Possible values are as follows: **`fmCreditsDebitsDisplay_Default`**: uses the default for the result model **`fmCreditsDebitsDisplay_Negative`**: displays credits as negative numbers **`fmCreditsDebitsDisplay_Positive`**: displays credits as positive numbers
Property **`Crossings`** As FMCrossingsCollection	A collection of crossings in this table. Read-only.
Property **`Debit`** As fmCreditsDebitsDisplay	The manner in which debit values are displayed in this table. Read/write. Possible values are as follows: **`fmCreditsDebitsDisplay_Default`**: uses the default for the result model **`fmCreditsDebitsDisplay_Negative`**: displays debits as negative numbers **`fmCreditsDebitsDisplay_Positive`**: displays debits as positive numbers
Property **`DisplayDebitCreditOnLabel`** As Boolean	This setting applies to account member labels. If **`True`**, each row and column heading contains the word **`(debit)`** or **`(credit)`**, whichever is applicable. Read/write.
Property **`FilterInvalid`** As Boolean	If **`True`**, rows or columns that contain only invalid values are not displayed. Read/write.
Property **`FilterInvalidOnColumns`** As Boolean	If **`True`**, columns that contain only invalid values are not displayed. Read/write.
Property **`FilterInvalidOnRows`** As Boolean	If **`True`**, rows that contain only invalid values are not displayed. Read/write.
Property **`FilterZeros`** As Boolean	If **`True`**, rows or columns that contain only zero values are not displayed. Read/write.
Property **`FilterZerosOnColumns`** As Boolean	If **`True`**, columns that contain only zero values are not displayed. Read/write.
Property **`FilterZerosOnRows`** As Boolean	If **`True`**, rows that contain only zero values are not displayed. Read/write.
Property **`FreezeCells`** As Boolean	If **`True`**, users cannot alter the table layout by operations such as changing the role of dimensions, expanding or collapsing hierarchies, adding or removing filters, and adding or removing calculated members. Read/write.

Property	Description
Property **`Hierarchies`** As FMHierarchiesCollection	A collection of hierarchies (both server hierarchies and client attribute hierarchies) in this table. Read-only. Server hierarchies are hierarchies that are defined on the server and that are being used in the specified results model, either directly (via an FMCube object) or via a virtual cube that is attached to an FMTable object. They are based on the required dimensions (such as **`ACCOUNT`**, **`ANALYSIS`**, and **`TIME`**) and any custom defined dimensions (such as **`PRODUCT`** or **`COSTCENTER`**). Client attribute hierarchies are virtual hierarchies that exist only on the client side. They are based on dimension attributes—both system properties and custom properties. *Note:* Custom properties hierarchies are treated like any other hierarchy.
Property **`Index`** As Long	The position of this table within the tables collection. Read-only.
Property **`Model`** As String	The results model for this table. Read/write.
Property **`Name`** As String	The name of this table. Read-only.
Property **`ReadOnly`** As Boolean	If **`True`**, this table is read-only. Read-only.
Property **`RefreshOnOtherTableUpdate`** As Boolean	If **`True`**, this table is refreshed when other tables in the same worksheet change. Read/write. For example, you might set this property to **`True`** for a read-only table so that it is refreshed when a user enters a value in a data entry table in the same worksheet.
Property **`ScaleValue`** As Double	The value by which displayed values are scaled. The actual computed values are divided by this number before they are displayed. Read/write.
Property **`ServerHierarchies`** As FMHierarchiesCollection	The collection of server hierarchies that are associated with the table. Read-only. For more information about server hierarchies, see the description of the Hierarchies property.

Table 6.16 *FMTable Class Method Summary Areas of a Table*

Method	Description

Sub **`BatchWrite`** ()	Writes the accumulated transactions to the server.
	Writeback is the process of writing back facts to the server. A writeback occurs when the user enters a value in a data entry table and presses ENTER. Normally, this operation requires one trip to the server for each value that the user enters. The BatchWrite method enables you to perform multiple updates with a single writeback. The process is as follows:
	1. Call the TransactionBegin method for a data entry table, to begin accumulating values.
	2. Write values to the table cells, either manually or programmatically. At this point, only the client-side representation is updated.
	3. Call the BatchWrite method to perform the writeback of the accumulated values.
	Here is an example:
	`Table.TransactionBegin`
	`For Each c In target`
	`   If c.Value <> newValue Then`
	`      table.Crossing(c.row, c.column).newValue = newValue`
	`   End If`
	`Next c`
	`Table.BatchWrite`

Sub **`Collapse`** (member As FMMember)	Collapses the selected member to hide all its descendants. Parameters: • *member*: the hierarchy member to collapse.
Function **`Crossing`** (row As Long, column As Long) As FMCrossing	Returns the crossing at (*row*, *column*). Parameters: • *row* and *column*: the Excel row and column values, after converting the column letter to a number (**`A`**=**`1`**, **`B`**=**`2`**, and so on).
Sub **`Expand`** (member As FMMember)	Expands the selected member to display all its children. Parameters: • *member*: the hierarchy member to expand.
Sub **`ExpandAll`** (member As FMMember)	Expands the selected member to display all its descendants. Parameters: • *member*: the hierarchy member to expand.
Sub **`FilterMemberCombinations`** (mems As FMMembersCollection)	Creates and applies a multi-member table filter based on the selected members. For more information, see the description of the **Filter Member Combination** option in the online Help for the SAS Financial Management Add-In for Microsoft Excel. Parameters: • *mems*: a collection of members.

Method	Description
Sub **`FilterMembers`** (row As Long, column As Long)	Creates and applies a single-member table filter based on the selected row or column heading. For more information, see the description of the **Filter Member Combination** option in the online Help for the SAS Financial Management Add-In for Microsoft Excel. Parameters: • *row*, *column*: the coordinates of the row or column heading that is used as the filter.
Function **`isDataArea`** (sheetName As String, row As Long, column As Long) As Boolean	Returns **`True`** if the specified position is within the data area. For more information about the data area, see the description of the Position function. Parameters: • *sheetName*: the name of the worksheet. • *row*, *column*: the coordinates of the position.

Function **`Pivot`** (hierarchy As FMHierarchy, role As fmRole, position As Long) As Boolean	Changes the layout of the selected table by changing the role of the hierarchy that was passed in. You must refresh the table in order to see the effects of the pivot operation.
	Parameters:
	• *hierarchy*: the hierarchy whose role is to be changed.
	• *role*: the new role for this hierarchy. A role of **`fmRole_Slicer`**, **`fmRole_Row`**, or **`fmRole_Column`** places the hierarchy in the slicer, row, or column section of the table. A role of **`fmRole_Available`** removes the hierarchy from its previous role as a slicer, row, or column and places it in the list of available hierarchies.
	• *position*: the hierarchy's position within its section (slicer, row, or column). If the section contains more than one hierarchy, existing hierarchies are pushed up or down as necessary to accommodate the position that you specify for this hierarchy. A position of **`0`** represents the highest position for the specified role.
	Returns: **`True`** if the operation succeeded; **`False`** if the role is the same as the current role or if an error is encountered.
	This example uses the Pivot method of the Table object to change the layout of a table. Assume that the column headings for a table are **`TIME`** and **`ANALYSIS`**, and the only row heading is **`ACCOUNT`**. The following code removes **`ANALYSIS`** from the column headings and adds it to the row headings. The new row headings would be **`ANALYSIS`** and **`ACCOUNT`**, in that order.
	`Dim table as FMTable`
	`Dim hierarchy as FMHierarchy`
	`set table = addin.tables("NewTable0")`
	`Set hierarchy = table.Hierarchies("ANALYSIS")`
	`rc = table.Pivot(hierarchy, fmRole_Row, 0)`
	`table.refresh(true)`

Method	Description

Function **`Position`** (area As fmArea, type As fmType) As Long	Returns the position of an element within the table. One common use for this method is to determine a range for applying custom formats to a table.
	Parameters:
	• *area*: the area of the table for which you want to know the position. This parameter can be one of the following:
	`fmArea_Table`: the entire table
	`fmArea_Slicer`: the table slicer area
	`fmArea_Row`: the row heading area
	`fmArea_Column`: the column heading area
	`fmArea_Data`: the data area
	`fmArea_Drillpath`: the drill-path area of the table
	The diagram in Figure 6.2 on page 107 shows the location of the areas in an example table.
	• type: the type of position to return, which can be one of the following:
	`fmType_startRow`: the position of the starting row of the specified area
	`fmType_endRow`: the position of the ending row of the specified area
	`fmType_startColumn`: the position of the starting column of the specified area
	`fmType_endColumn`: the position of the ending column of the specified area
	`fmType_width`: the width of the specified area, in terms of number of columns
	`fmType_height`: the height of the specified area, in terms of number of rows
	`fmType_rowOffset`: the number of rows before the start of this table (regardless of the area)
	`fmType_columnOffset`: the number of columns before the start of this table (regardless of the *area* parameter)
	Returns: a value for the specified area and type.
	This example finds the positions of the start and end rows and columns in the data area of a table:
	`startRow = table.Position(fmArea_Data, fmType_startRow)`
	`endRow = table.Position(fmArea_Data, fmType_endRow)`
	`startColumn = table.Position(fmArea_Data, fmType_startColumn)`
	`endColumn = table.Position(fmArea_Data, fmType_endColumn)`

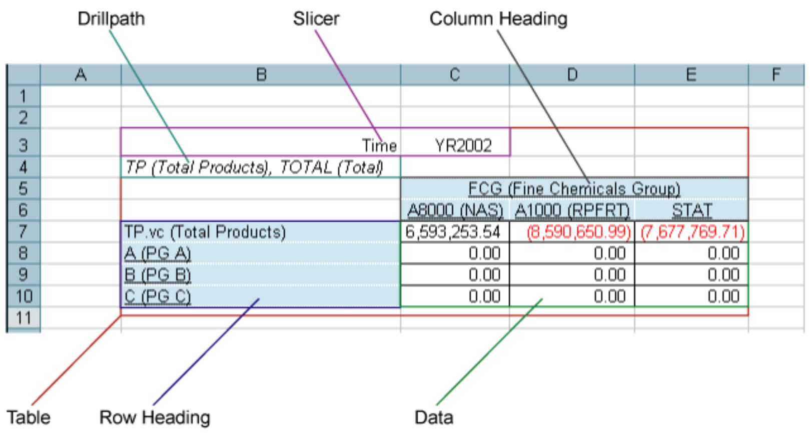
Method	Description

Sub **`Refresh`** (reQuery As Boolean)	Refreshes the table.
	Parameters:
	• *reQuery*: the requery flag. If **`True`**, the function performs a requery and repaint. If **`False`**, the function performs a repaint only.
	Consider carefully whether a requery is needed or whether a repaint is sufficient. The refresh operation requires more resources when a requery is included.
	This example refreshes the specified table and performs a requery:
	`table.Refresh(True)`
	This example repaints the table without performing a requery:
	`table.Refresh(False)`

Sub **`RemoveAllMemberCombinationFilters`**()	Deletes all table filters.
Function **`TargetHierarchy`**() As FMHierarchy	Returns the target hierarchy for this table and model.
Sub **`UnfilterMemberCombinations`** (mems As FMMembersCollection)	Deletes the table filter that is specified by the combination of members. Parameters: • *mems*: a collection of members.
Function **`Write`** (row As Long, column As Long, newValue As Double) As fmWriteBackReturn	Writes *newValue* to the crossing that is specified by *row* and *column*. Parameters: • *row* and *column*: the row and column values that determine the crossing. For column values, convert letters to numbers (for example, cell **`A3`** is the crossing that is determined by a row value of **`3`** and a column value of **`1`**). • *newValue*: the value to write. Returns: **`fmWriteBackReturn_Succeeded`**, **`fmWriteBackReturn_FailedCantUpdateForm`**, **`fmWriteBackReturn_FailedReadOnly`**, **`fmWriteBackReturn_FailedNoValueChange`**, **`fmWriteBackReturn_FailedNoValue`**, **`fmWriteBackReturn_FailedNoXRate`**, or **`fmWriteBackReturn_FailedUnknown`**.
Sub **`TransactionBegin`**()	Begins accumulating transactions for later writeback using the BatchWrite method.
Function **`Writeable`** (row As Long, column As Long) As Boolean	Determines whether a specified crossing is writable. Read-only. Parameters: • *row* and *column*: the row and column values that determine the crossing. For column values, convert letters to numbers. Returns: **`True`** if the crossing is writable; otherwise, **`False`**.

This diagram illustrates the areas of a table. (See the Position method of the FMTable class.)

Figure 6.2 *Areas of a Table*



	A	B	C	D	E	F
1						
2						
3		Time	YR2002			
4		*TP (Total Products), TOTAL (Total)*				
5			FCG (Fine Chemicals Group)			
6			A8000 (NAS)	A1000 (RPFRT)	STAT	
7		TP.vc (Total Products)	6,593,253.54	(8,590,650.99)	(7,677,769.71)	
8		A (PG A)	0.00	0.00	0.00	
9		B (PG B)	0.00	0.00	0.00	
10		C (PG C)	0.00	0.00	0.00	
11						

Table 6.17 *FMTable Class: Event Summary*

Event	**Description**

Event **`AfterRefresh`**()	Triggered after the table has been refreshed. This event might occur if the code calls the **`Refresh`** method of a table object, if the user selects **Refresh** or **RefreshAll** from the **View** menu, or if the user performs some other manual action that triggers a refresh. If the user refreshes a worksheet, the table refresh event and the addin refresh event are triggered, in that order.
	
	Here is an example of an event handler for a table called **`table1`**:
	
	`Private Sub table1_AfterRefresh()`
	`  ' Assumes that screen updating has been disabled`
	`  ' in the BeforeRefresh event handler`
	`  ...`
	`  ' Perform some actions`
	`  ...`
	`  ' Re-enable screen updating`
	`  Application.ScreenUpdating = True`
	`End Sub`
	
	Notice that the name of the event handler includes the name of the object (in this case, **`table1`**). It applies only to this table, not to any other tables in the worksheet. If you wanted to affect a different table (for example, **`table2`**), you would write a second event handler, **`table2_afterRefresh()`**. To handle all table refresh events identically, you could use the **`addin_AfterTableRefresh`** event handler.

Event	Description

Event **BeforeRefresh**()	Triggered before the table is refreshed. One use for this event handler is to disable screen updating. You could re-enable screen updating in the AfterRefresh event handler. For more information, see the description of AfterRefresh.
	Here is an example of an event handler for a table called **table1**:
	`Private Sub table1_BeforeRefresh()`
	`  ' Disable screen updating`
	`  Application.ScreenUpdating = False`
	`  ' Perform some actions`
	`  ...`
	`End Sub`

The FMTablesCollection Class

The FMTablesCollection class represents a collection of tables. This class is a subclass of FMCollections.

Properties inherited from FMCollections: Count.

Methods inherited from FMCollections: Add, AddAll, Clear, Contains, IndexOf, Insert, InsertAll, Remove, RemoveAt, ToString.

The FMUser Class

The FMUser class contains information about the user who is currently logged on.

Table 6.18 *FMUser Class Property Summary*

Property	Description
Property **BudgetMode** As FMBudgetMode	Returns one of the following values: • **fmBudgetMode_Create**: if the user is editing a template • **fmBudgetMode_Entry**: if the user is editing a form Otherwise, the value is **fmBudgetMode_None**. Read-only.
Property **FormId** As Long	The form ID. Read-only.
Property **FormSetId** As Long	The form set ID. Read-only.
Property **FormSetName** As String	The name of the form set. Read-only.
Property **FormTemplateId** As Long	The form template ID. Read-only.

Property	**Description**
Property **`LockName`** As	If the form is opened in data-entry mode, this property contains the name that is associated with the lock. The lock name can be viewed in the log. (See "Activating the Log" on page 79.)
Property **`ReadOnly`** As Boolean	This property applies only if a user has a form open. It has a value of **`True`** if the form is read-only. The user might have launched the form for viewing only, or the user might not have permission to edit the form. Read-only.
Property **`UserContext`** As String	Returns the user context (session ID). Read-only. With this property, a user can log on to the middle tier without reauthorization—for example, to run a stored process.
Property **`UserId`** As String	The user ID (for example, **`sasdemo`**). Read-only.
Property **`UserName`** As String	The user display name (for example, **`SAS Demo User`**). Read-only.
Property **`WorkflowMethod`** As String	The workflow method, which can be one of the following: **`TopDown`** or **`BottomUp`**. Read-only.

Chapter 7

Auditing in SAS Strategy Management

Configure Auditing in SAS Strategy Management

Audit logging in SAS Strategy Management enables site administrators to track and report on model changes, usage patterns, value changes, and permission changes. Four levels of auditing can be configured:

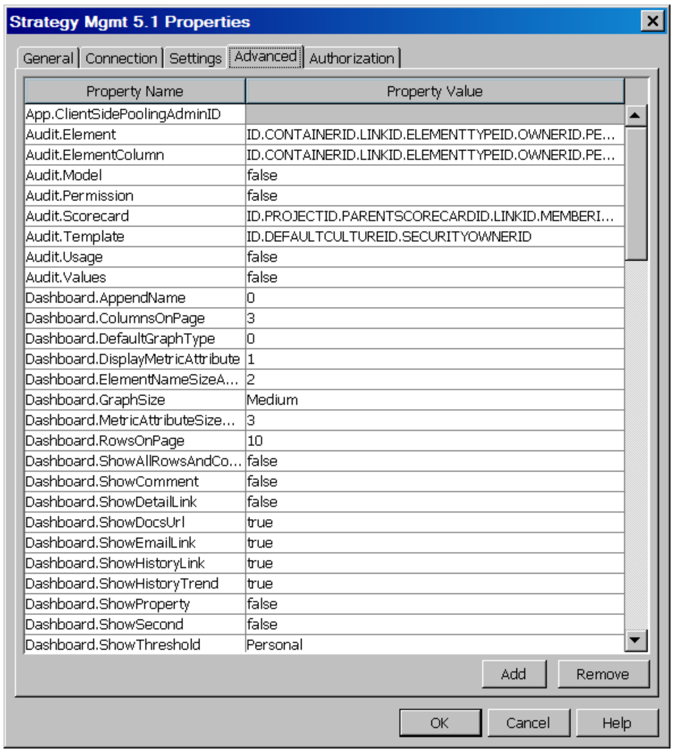
- **Audit.Model:** Tracks all changes to templates, projects, scorecards, and elements.
- **Audit.Usage:** Tracks the usage of table views, aggregate views, association views, and diagram views.

  Note: This level produces a large auditing table.
- **Audit.Values:** Tracks all changes to the values of metric attributes.
- **Audit.Permission:** Tracks changes to permission settings.

By default, auditing is disabled for SAS Strategy Management. To enable auditing for one or more levels, follow these steps:

1. Log on to SAS Management Console as a member of the SAS Administrators group.
2. On the **Plug-ins** tab, navigate to **Application Management** ⇨ **Configuration Manager**.
3. Right-click **Strategy Mgmt 5.1** and select **Properties**.
4. In the properties dialog box, click the **Advanced** tab.
5. Select a level (**Audit.Model**, **Audit.Permissions**, **Audit.Usage**, or **Audit.Values**) and change its value to `true`.

   Note: Use lowercase. The property value is case sensitive.



Property Name	Property Value
App.ClientSidePoolingAdminID	
Audit.Element	ID.CONTAINERID.LINKID.ELEMENTTYPEID.OWNERID.PE...
Audit.ElementColumn	ID.CONTAINERID.LINKID.ELEMENTTYPEID.OWNERID.PE...
Audit.Model	false
Audit.Permission	false
Audit.Scorecard	ID.PROJECTID.PARENTSCORECARDID.LINKID.MEMBERI...
Audit.Template	ID.DEFAULTCULTUREID.SECURITYOWNERID
Audit.Usage	false
Audit.Values	false
Dashboard.AppendName	0
Dashboard.ColumnsOnPage	3
Dashboard.DefaultGraphType	0
Dashboard.DisplayMetricAttribute	1
Dashboard.ElementNameSizeA...	2
Dashboard.GraphSize	Medium
Dashboard.MetricAttributeSize...	3
Dashboard.RowsOnPage	10
Dashboard.ShowAllRowsAndCo...	false
Dashboard.ShowComment	false
Dashboard.ShowDetailLink	false
Dashboard.ShowDocsUrl	true
Dashboard.ShowEmailLink	true
Dashboard.ShowHistoryLink	true
Dashboard.ShowHistoryTrend	true
Dashboard.ShowProperty	false
Dashboard.ShowSecond	false
Dashboard.ShowThreshold	Personal

6. If you are enabling **Audit.Model** or **Audit.Usage**, you must also configure the fields to be audited, as follows:

   a. Select one of the following properties:

      - **Audit.Element**
      - **Audit.ElementColumn**
      - **Audit.Scorecard**
      - **Audit.Template**

   b. Each of those properties is configured with a default set of values. In the **Property Value** column, add or remove field names, separating the entries with a period (.).

Audit.Element	ID.CONTAINERID.LINKID.ELEMENTTYPEID.OWNERID.PE...

7. Click **OK** to save your changes.

The changes go into effect when you restart all the managed servers.

The following table lists the field names that can be set in the **Property Value** column. Your selections apply to both **Audit.Model** and **Audit.Usage** (if auditing for those levels is enabled).

Table 7.1 *Auditing Levels and Fields*

Level	Field Name	Description
Audit.Element	ID	The GUID that identifies this element.
	CONTAINERID	The GUID for the scorecard or project that contains this element.

Level	Field Name	Description
	ELEMENTTYPEID	The GUID for the element type for this element.
	LINKID	The GUID for any element that is associated with this element. If the element is not associated with any other elements, then this value is identical to the ID value.
	OWNERID	The user ID of the current owner of this element.
	PERIODTYPE	The periodicity of this element.
	SECURITYOWNERID	The user ID of the user who created this element.
	SECURITYUSETYPE	The security use type for this element. Possible values are: • **N**: None • **C**: Container • **E**: Entity • **H**: Hierarchy
	FROMPERIODID	The beginning effective period for this element.
	TOPERIODID	The ending effective period for this element.
	ORDERNUM	An internal value that is used to order elements for viewing.
Audit.ElementColumn	ELEMENTID	The GUID used to identify the element that this attribute belongs to.
	COLUMNID	The GUID used to identify the element attribute.
	PERIODID	The GUID used to identify the period associated with the element attribute.
	VALUE	The current value of the element attribute
	METRICTEXTVALUE	The associated metric text value for this cell.
	LASTMODIFIEDDATE	The date the element attribute was last modified.
	RANGEID	The GUID used to identify the range associated with the element attribute.
	MEASUREID	The GUID used to identify the measure associated with this cell.
	FORMULA	The formula assigned to the element attribute.
	THRESHOLD	The value for which an associated threshold is crossed.
	THRESHOLDOPERATOR	The operator for the associated threshold.

Level	Field Name	Description
	THRESHOLDTYPE	The threshold type for the associated threshold.
	THRESHOLDINTERVALID	The interval ID of the associated threshold.
	SUPPORTINGDOCUMENTURL	The associated URL string for this cell.
	ISUSEROVERRIDEVALUE	An override flag. A value of `1` indicates that the cell value has been overridden by the user. Otherwise, the value is `0`.
	DIRECTIVE	The directive associated with the element attribute.
	DIRECTIVE_PARMS	The parameters used with any associated directive.
	STOREDPROCESSID	The ID of an associated stored process.
	STOREDPROCESSPARMS	The parameters to be sent to the associated stored process, in a string separated by semicolons.
Audit.Scorecard	ID	The GUID that identifies this scorecard.
	PROJECTID	The GUID for the project that contains this scorecard.
	PARENTSCORECARDID	The GUID for the parent scorecard for this scorecard.
	MEMBERID	The GUID for the dimension ID, if the project for this scorecard is linked to a dimensional hierarchy.
	SECURITYOWNERID	The user ID for the owner of this scorecard.
	SECURITYUSETYPE	The security use type for this scorecard (see the description of this field in the **Audit.Element** level).
	ORDERNUM	An integer value for the scorecard ordering.
Audit.Template	ID	The GUID used to identify the template.
	DEFAULTCULTUREID	The GUID used to identify the default language for this template.
	SECURITYOWNERID	The user ID of the template owner.
	SECURITYUSETYPE	The security use type for this template. (See the description of the SECURITYUSETYPE field in the **Audit.Element** level.)

Create an Audit Report

Auditing information is recorded in three tables in the SHAREDSERVICES database: SAS_ACTION_EXECUTOR, SAS_AUDIT, and SAS_AUDIT_ENTRY.

Here is an example SAS program that creates an audit report. The query includes this filter to return only items that have been logged for SAS Strategy Management: **`where sas_action_executor.executor_nm = "Strategy Mgmt 5.1"`**.

Note: This code is intended only as an introduction to audit reporting.

Example Code 7.1 *Sample Audit Report*

```
/* Create a libref to the SharedServices database */
/* (Replace mysqlusername, serverpassword, servername, serverport) */

libname auditref MYSQL user=mysqlusername password=serverpassword
   database=SharedServices server=servername port=serverport;

/* Use PROC SQL to create an audit table with entries of interest */
proc sql;
   create table audit as  select distinct sas_audit.user_id,
      sas_audit.timestamp_dttm, sas_audit.session_id,
      sas_type_object.type_object_cd, sas_audit.object_id,
      sas_audit.audit_id, sas_audit_entry.property_nm,
      sas_audit_entry.new_value_txt

   from auditref.sas_action_executor, auditref.sas_audit,
      auditref.sas_audit_entry, auditref.sas_type_object

   /* Include only SAS Strategy Management audit records */
   where sas_action_executor.executor_nm = "Strategy Mgmt 5.1" and
      sas_audit.object_type_id = sas_type_object.type_object_id and
      sas_audit.audit_id = sas_audit_entry.audit_id;
run;
proc sort data=audit;
   by user_id audit_id object_id;
run;
```

The following columns are referenced in the example program:

- **SAS_AUDIT.USER_ID**: the user ID of the user performing the action
- **SAS_AUDIT.TIMESTAMP_DTTM**: a timestamp of when the action occurred
- **SAS_AUDIT.SESSION_ID**: the session ID for the action
- **SAS_AUDIT.OBJECT_ID**: the GUID of the object that the audit is being performed on (for example, the SAS Strategy Management project)
- **SAS_TYPE_OBJECT.TYPE_OBJECT_CD**: the object type, such as **`SPMProject`**
- **SAS_AUDIT.AUDIT_ID**: the ID of the audit record
- **SAS_AUDIT_ENTRY.PROPERTY_NM**: the name of the property that was affected
- **SAS_AUDIT_ENTRY.NEW_VALUE_TXT**: the new value of the property

Chapter 8

Using Secure Sockets Layer (SSL)

About SSL

The Secure Sockets Layer protocol (SSL) provides secure connections by allowing two applications connecting over a network connection to authenticate the other's identity and by encrypting the data exchanged between the applications. Authentication allows a server to verify the identity of the client application on the other end of a network connection.

Encryption makes data transmitted over the network intelligible only to the intended recipient.

Using SSL is computationally intensive and adds overhead to a connection. Avoid using SSL in development environments when it is not necessary. Use SSL in a production environment if a customer site has policies requiring that all network traffic must be encrypted.

References

The *SAS Intelligence Platform: Web Application Administration Guide* contains instructions for enabling SSL for Web applications, including the portal and other applications that are part of the SAS Intelligence Platform. It also contains information about one-way versus two-way SSL. This book is available at **`http://support.sas.com/92administration`**. See the following section: "Using Secure Sockets Layer (SSL) for Web Applications."

The third-party support center has information about configuring WebLogic servers. See **`http://support.sas.com/resources/thirdpartysupport/v92m2/appservers/weblogicdoc.html`**—in particular, "SAS 9.2 Web Applications: Tuning for Performance and Scalability."

In addition, the Oracle WebLogic documentation contains extensive information about configuring SSL for the WebLogic servers, including key and certificate management. See **`http://download.oracle.com/docs/cd/E12840_01/wls/docs103/secmanage/ssl.html`**.

Configuring SSL for the Solutions

To configure SSL for the solutions, follow these steps:

Note: The instructions in this chapter assume that the site is configuring one-way SSL. For more information about two-way SSL, see the *SAS Intelligence Platform: Web Application Administration Guide* and the Oracle WebLogic documentation.

1. Make sure that the solutions are running correctly without SSL.
2. Obtain the necessary digital certificates.

   For testing purposes, you can use the DemoIdentity.jks and DemoTrust.jks keystores in the WL_HOME\server\lib directory. See "Configure keystores" in the online Help for the WebLogic Administration Console.

   For production environments, the customer or client site must provide a trusted certificate that has been digitally signed by a valid certificate authority. You can then import this certificate into the WebLogic environment.

   Note: Be sure to use the correct spelling and case for the WebLogic server names whenever you reference them.
3. Configure SSL for the managed servers.

   See "Configure the Managed Servers" on page 119.
4. Configure the Web applications.

   See "Configure the Web Applications" on page 120.

5. Configure the SAS Content Server.

   See “Configure the SAS Content Server” on page 121.

6. Change the Content Mapping properties for the **`SAS Folders`**.

   See “Modify the Content Mapping” on page 122.

7. If the site has any remote portlets, update the protocol and port numbers for those portlets. Update the URL within the portlet.xml file, recreate the PAR file, and redeploy it.

   If the site uses the SAS BI Dashboard JSR 168 remote portlet, perform the following steps:

   a. Update its portlet.xml file and change the URL to the server where SAS BI Dashboard is deployed.

   b. Redeploy the sas.bidashboardjsr1684.2.ear application.

      For more information, see the installation instructions for SAS BI Dashboard applications.

8. Modify the WebDAV connection information in the foundation services.

   See “Modify the Foundation Services” on page 124.

9. Configure the remote services to support SSL.

   See “Modify the Remote Services” on page 122.

10. (SAS Human Capital Management only) Update the HCM-config.xml file and the hcm-metric.dsx file.

    See “Modify SAS Human Capital Management Files” on page 125.

11. Modify the sas-environment.xml file and the EnvironmentFactory.xml file.

    See “Modify the SAS Environment Files” on page 126.

12. Restart the remote services and the managed servers.

    See “Restart and Test” on page 129.

13. Verify the SSL connection by logging on to one of the Web applications using the HTTPS protocol and new port number.

Configure the Managed Servers

1. If the site requires it, configure SSL for the Admin server and the Node Manager (if used).

2. Configure SSL for each managed server (including any secondary ODCS servers). For instructions, see “Using Secure Sockets Layer (SSL) for Web Applications” in the *SAS Intelligence Platform: Web Application Administration Guide*, as well as the WebLogic documentation. Keep the following points in mind:

   - If the **`allowQuotes`** option is not set, add it to the server start arguments for each server, as follows:

     ```
     -Dweblogic.serverStart.allowQuotes=true
     ```

   - Be sure to enable the SSL listen port and set the correct port number for each managed server.

For HTTPS port numbers, see "Configuring your WebLogic Application Server (Domain Configuration)" in the Instructions.html file from your installation. This file is located in the ***SAS-config-dir*\Lev1\Documents** folder on the middle tier.

- Add the following options to the server start arguments for each server:

  ```
  -Djavax.net.ssl.trustStore=C:\Java\jdk1.6.0_16\jre\lib\security\cacerts _
  -Djavax.net.ssl.trustStorePassword=changeit _
  -Djavax.net.ssl.keyStore=C:\bea\wlserver_10.3\server\lib\WLStore.jks _
  -Djavax.net.ssl.keyStorePassword=weblogic
  ```

  Note: Line breaks ("_") added for readability only.

  Replace the paths in the **trustStore** and **keyStore** arguments with the correct paths for your installation.

- As you configure the SASServer1 and SASServer2 managed servers, modify the following JVM argument:

  ```
  -Dsas.auto.publish.port
  ```

  Change the port number to the secure port for the managed server, and add an argument to set the protocol. For example:

  ```
  -Dsas.auto.publish.port=7002 -Dsas.auto.publish.protocol=https
  ```

  Note: Line breaks added for readability only.

- In the Foreign JNDI Providers service configuration, change the Provider URL of the SharedServicesJNDIProvider to reference the t3s protocol and the secure port number that applies to SASServer1.

- (Optional) To enable SSL debugging, you can temporarily add this command-line option to the node manager, managed server, or client application:

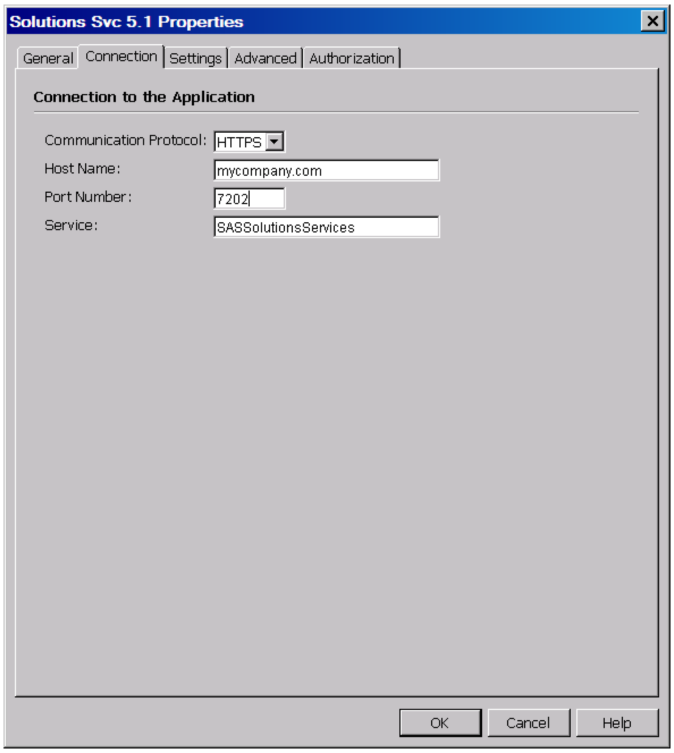
  ```
  -Dssl.debug=true -Dweblogic.StdoutDebugEnabled=true
  ```

Note: If you installed the managed servers as services, uninstall each service and reinstall it so that the changes take effect.

Configure the Web Applications

Configure the Web applications to support SSL, as follows:

1. Log on to SAS Management Console as an administrator and select the **Plug-ins** tab.
2. Navigate to **Application Management** ⇨ **Configuration Manager**.
3. Right-click the first application in the list and select **Properties**.
4. Click the **Connection** tab.
5. From the **Communication Protocol** drop-down box, select **HTTPS**.
6. In the **Port Number** field, enter the secure port number for the managed server to which you deployed this application.



Connection to the Application

Communication Protocol: HTTPS
Host Name: mycompany.com
Port Number: 7202
Service: SASSolutionsServices

7. Save your changes.

Repeat steps 1–7 for each application that has a **Connection** tab and that you want to enable for SSL. Make sure that the SAS Logon Manager (on SASServer1) uses the HTTPS protocol.

For the BI Web Services for Java 9.2, follow these additional steps:

1. Expand the entry to find and open the properties for the Corr, CorrGroup, CorrRegGroup, MultReg, and SingleReg services.
2. Modify the connection properties for these services.

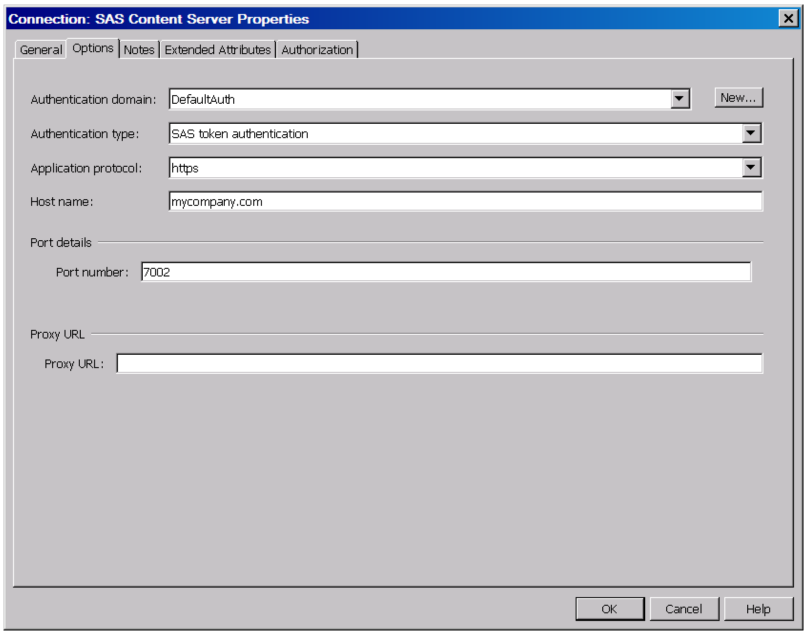
Note: If an application (such as the SAS Web Application Themes) uses static content only, you might want that application to continue to use the HTTP protocol. Using the HTTP protocol for some applications can improve performance. However, it requires that you leave both the secure and nonsecure ports open for that managed server.

Configure the SAS Content Server

Configure the communication protocol and port number for the SAS Content Server, as follows:

1. Log on to SAS Management Console as an administrator and select the **Plug-ins** tab.
2. Navigate to **Server Manager** and select **SAS Content Server**.
3. In the right-hand panel, right-click **SAS Content Server** and select **Properties**.
4. Click the **Options** tab.
5. From the **Application protocol** drop-down list, select `https`.

6. In the **Port number** field, enter the same secure port number that you used for the Web Infrastructure Platform (SASServer1).



Connection: SAS Content Server Properties

Authentication domain: DefaultAuth
Authentication type: SAS token authentication
Application protocol: https
Host name: mycompany.com
Port number: 7002
Proxy URL:

7. Save your changes.

Modify the Content Mapping

Modify the content mapping for the `SAS Folders`, as follows:

1. On the **Folders** tab of SAS Management Console, right-click **SAS Folders** and select **Properties**.
2. Click the **Content Mapping** tab.

   Select **WebDAV location** if it is not already selected.
3. From the **Server** drop-down list, select **SAS Content Server**.

   As a result, the **URL** field displays the updated URL for the SAS Content Server, including the HTPS protocol and the new port number.
4. Click **OK**.

   A pop-up message appears, asking you to confirm your change. Click **Yes**.

Modify the Remote Services

Overview

Modify the JVM parameters for the SAS Remote Services to include the certificate authority (CA) keystore:

- If you run the remote services as a service, see "Modify the wrapper.conf File for the Service" on page 123.
- If you run the remote services from a start-up script, see "Modify the Start-up Script" on page 123.

Modify the wrapper.conf File for the Service

If you run the SAS Remote Services as a Windows service, follow these steps:

1. Open the wrapper.conf file for editing.

   The wrapper.conf file is in the ***SAS-config-dir*\Lev1\Web\Application \RemoteServices** directory.

2. Add two wrapper.java.additional parameters, similar to the following:

   ```
   wrapper.java.additional.12=
      -Djavax.net.ssl.trustStore="C:\Java\jdk1.6.0_16\jre\lib\security\cacerts" _
   wrapper.java.additional.13=
      -Djavax.net.ssl.trustStorePassword=changeit
   ```

   Note: Each parameter should go on a separate line. A line break ("_") is added at the end of the second line for readability.

   These parameter values must match the values that you defined for the managed servers. The parameter numbers might be different if your site's file has more or fewer parameters. Replace the example path shown above with the correct path to the Java Development Kit (JDK).

3. Save the file.

Modify the Start-up Script

If you run the SAS Remote Services from a batch script, follow these steps:

1. Open the start-up script, RemoteServices.bat, for editing.

   This file is located in the ***SAS-config-dir*\Lev1\Web\Applications \RemoteServices** directory.

2. Find the following line in the file:

   ```
   set SERVERTYPE=services
   ```

3. After that line, create a variable called SSL_OPTS, as follows:

   ```
   set SSL_OPTS=-Djavax.net.ssl.trustStore= _
      "C:\Java\jdk1.6.0_16\jre\lib\security\cacerts\jre\lib\security\cacerts"
   ```

   Note: Line break ("_") added for readability at the end of the first line. To use this code, make the SSL_OPTS variable definition one continuous line.

   These parameter values must match the values that you defined for the managed servers.

   The parameter numbers might be different if your site's file has more or fewer parameters. Replace the example path shown above with the correct path to the JDK.

4. Find the **:start3** label and insert the %SSL_OPTS% string into the command. For example:

   ```
   :start3
      "%JAVA_JRE_COMMAND%" ^
   ```

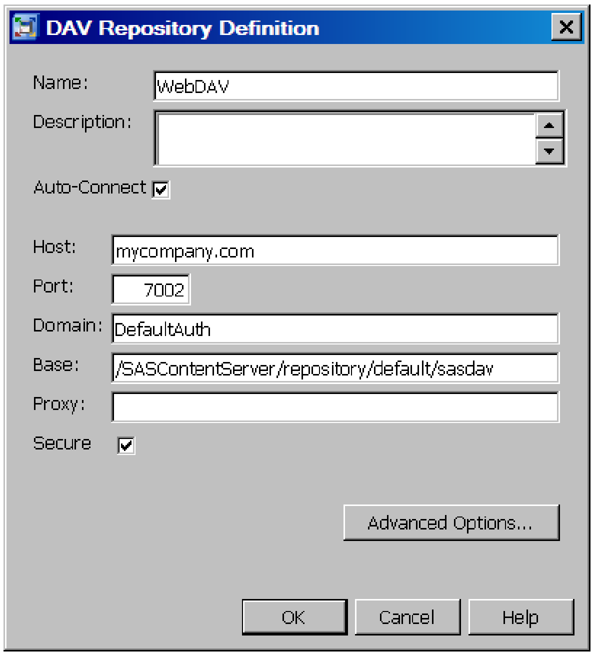
```
-classpath "%CLASSPATH%"  ^
...
%SSL_OPTS% ^
com.sas.framework.services.bootstrap.SASRemoteServices
goto end
```

5. If the file also contains a **:start2** label, insert the %SSL_OPTS% variable in that command as well.
6. Save the file.

Modify the Foundation Services

Modify the WebDAV connection information in the foundation services, as follows:

1. On the **Plug-ins** tab of SAS Management Console, navigate to **Environment Management** ⇨ **Foundation Services Manager** ⇨ **Remote Services** ⇨ **Core**.
2. Right-click **Information Service** and select **Properties**.
3. On the **Service Configuration** tab, click the **Configuration** button.
4. In the Configuration dialog box, click the **Repositories** tab.
5. In the **Information Repositories** list, select **WebDAV**. At the bottom of the page, click **Edit**.
6. In the DAV Repository Definition dialog box, change the port number to the same port number that you used for SAS Content Manager (typically `7002`).
7. Select the **Secure** check box.


DAV Repository Definition

Name: WebDAV
Description:
Auto-Connect ☑
Host: mycompany.com
Port: 7002
Domain: DefaultAuth
Base: /SASContentServer/repository/default/sasdav
Proxy:
Secure ☑

8. Save your changes.
9. Repeat these steps for the following foundation services:
   - SAS Package Viewer Local Services

- SAS Portal Local Services
- SAS Stored Process Local Services
- SAS Web Report Studio Local Services

Modify SAS Human Capital Management Files

Update the HCM-config.xml File

If you installed SAS Human Capital Management, follow these steps to update the HCM-config.xml file:

1. Open the HCM-config.xml file for editing.

   This file is located on the middle-tier machine, in the ***`SAS-config-dir`*** **`\Lev1\AppData\SASHumanCapitalManagement5.1`** directory.

2. Find the following entry:

   ```
   <Property Id="ProviderURL" Name="Provider URL"
      Value="t3://hostname:port "
      ReadOnly="true"/>
   ```

   Hostname and *port* are specific to your site.

3. Change **`t3`** to **`t3s`**, and change the port number to the secure port number for SASServer3.

4. Save the file.

Modify the DSX File for SAS BI Dashboard

If you installed SAS Human Capital Management, you need to modify its metric definition file for SAS BI Dashboard. Follow these steps:

1. Open the hcm-metric.dsx file for editing.

   This file is located on the middle tier in the ***`SAS-config-dir`*****`\Lev1\AppData\SASBIDashboard4.2\dataSourceDefs`** directory.

2. Find the <WSUrl> entry near the end of the file.

3. Change the protocol to **`https`**.

4. Change the port number to the secure port number for the SAS Human Capital Management application (typically, **`7202`**).

5. Save the file.

Modify the SAS Environment Files

Overview

You must modify the SAS environment files to reflect the new protocol and port numbers. (Make a backup copy of each file before you modify it.)

If you have not already done so, publish the environment files to an HTTP server, as described in “Installing the Client Applications” in the *SAS Solutions Services: System Administration Guide*.

Note: You can use the original configuration of these files to validate an installation. However, after validation, the SAS environment files should be published to an HTTP server, regardless of whether the site has a single SAS environment or multiple environments, and regardless of whether the site is using SSL or not.

Update the EnvironmentFactory.xml File

Edit the EnvironmentFactory.xml and EnvironmentFactory.odcs.xml files as follows:

1. On the middle-tier server, change directory to ***SAS-config-dir*\Lev1\Web\Applications\SASSolutionsServices5.1**.
2. In the EnvironmentFactory.xml file, modify the URLs as follows:
   a. Change **http** to **https**.
   b. Change **t3** to **t3s**.
   c. Change the port numbers to reference the secured ports.

   For example:

   ```
   <java.naming.provider.url>t3s://server-name:7202</java.naming.provider.url>
   ```
3. Copy the modified EnvironmentFactory.xml file to the sas.solutions.common.war directory within the sas.solutionsservices5.1.ear application (replacing the current version of that file). If you are deploying SAS Solutions Services as an exploded EAR, this location is ***SAS-config-dir*\Lev1\Web\Staging\exploded\sas.solutionsservices5.1.ear\sas.solutionscommon.war**.
4. Make similar changes to the EnvironmentFactory.odcs.xml file. It is located in the ***SAS-config-dir*\Lev1\Web\Applications\SASODCSForSolutions5.1** directory.
5. In the deployed files for your HTTP server, modify the EnvironmentFactory.xml file as follows:
   a. Find the section that matches your environment.
   b. Within that section, modify the URLs as described in Step 2.

Update the sasv9_usermods.cfg File

Modify the sasv9_usermods.cfg file, as follows:

1. On the data tier, change directory to ***SAS-config-dir*\Lev1\SASApp**.

2. Open the sasv9_usermods.cfg file for editing.
3. In the JREOPTIONS, change the protocol and port number for the EnvironmentFactory.xml file as follows:

   ```
   -JREOPTIONS=(-Denv.factory.location= _
   http://hostname:secure-port/SASConfig/EnvironmentFactory.xml)
   ```

   Note: Line break ("_") was added for readability.

   - *hostname* is the name of the middle-tier server.
   - *secure-port* is the secure port number for SASServer3, where SAS Solutions Services is deployed. The default port number is 7202.

Update the sas-environment.xml File

Update the sas-environment.xml file as follows:

1. Open the sas-environment.xml file for editing.

   This file is located in the ***`SAS-config-dir`*****`\Lev1\Web\Common`** directory.
2. In the section that matches your SAS environment, find this entry:

   ```
    <service-registry>
     http://server:port/SASWIPServices/remote/serviceRegistry
   </service-registry>
   ```
3. Change the protocol from **`http`** to **`https`**.
4. Change *port* to reflect the secure port number for the sas.wip.services92.ear application. Typically, this secure port is **`7002`**.
5. Save the file. The change applies when you restart the server.

Configuring Java Desktop Clients for Use with an SSL-Enabled Server

Overview

Some customer sites require that all client communication be conducted over secure communication channels. You can configure SAS Financial Management Studio, SAS Solutions Services Add-In for Microsoft Office, SAS Financial Management Add-In for Microsoft Excel, and SAS Solutions Dimension Editor to communicate via a secure connection to the middle tier, as follows:

- Modify the .INI files to use a secure connection to the environment files.
- Import the CA certificate to the client machines.

Modify the .INI Files

1. On each client machine, edit the appropriate .INI file:
   - ***`SAS-install-dir`*****`\SASSolutionsServicesAdd-InforMicrosoftOffice\5.1\SASSolutionsOfficeClient.ini`**

(applies to both SAS Solutions Services Add-In for Microsoft Office and SAS Financial Management Add-In for Microsoft Excel)

- ***SAS-install-dir*\SASFinancialManagementStudio \5.1\fmstudio.ini** (applies to SAS Financial Management Studio)
- ***SAS-install-dir*\SASSolutionsDimensionEditor \5.1\soldimedit.ini** (applies to SAS Solutions Dimension Editor)

2. In the .INI file, change the URL to the environment file so that it uses the HTTPS protocol and the secure port. For example:

```
[Environment Factory]
https://myhttpserver:secure-port/EnvironmentFactory.xml
```

3. Save the file.

Import the Certificate to the Client Machines

A customer site will deploy a signed certificate on the server. Because it is signed, there are no issues with any client, and no client component must be modified to trust this connection.

However, it is possible to use a demo or test certificate that is not signed. In those cases (and only in those cases), it is necessary to update the client's JRE to import the demo certificate, so that the client can communicate over SSL to the test configuration on the middle-tier server.

To import the CA certificate to a client machine, follow these steps:

1. On the client machine, find the **lib\security** directory of the JRE that is used by SAS Financial Management Studio.

   Typically, the JRE is specified in the -vm parameter of the .INI file for SAS Financial Management Studio. Otherwise, the default JRE on the client machine is used. For details, see “Installing the Client Applications” in the *SAS Solutions Services: System Administration Guide*.

2. Copy the certificate file from the middle-tier server to the **lib\security** directory on the client machine.
3. On the client machine, open a command prompt window and change directory to the **lib\security** directory.
4. Execute the following command:

```
..\..\bin\keytool.exe -import -alias WLrootcert _
   -file certificate-name.cer -keystore cacerts
```

   Note: Line break (“_”) was added for readability.

   certificate-name.cer is the name of the certificate file that you copied from the middle-tier server (for example, myCA.cer).

5. The keytool program prompts you for a password. The password is **changeit**.
6. Respond **Y** to the prompt **Trust this certificate?** .

The keytool program displays a message confirming that the certificate was imported.

Restart and Test

1. Restart the remote services, the HTTP server (if there is one), and the managed servers.
2. Verify the SSL connection by logging on to one of the Web applications using the HTTPS protocol and new port number.
3. Confirm that the connection is secure by observing the padlock icon. To examine the certificate and certificate chain, click the padlock.
4. In a production environment, you might also want to disable the nonsecure ports.

Index

Your Turn