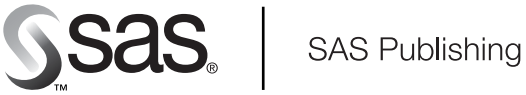


SAS® 9.1 Open Metadata Interface:

User's Guide

The Power to Know®

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2004. *SAS® 9.1 Open Metadata Interface: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® 9.1 Open Metadata Interface: User's Guide

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

1st printing, January 2004

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

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

SAS 9.1 Open Metadata Interface: User's Guide

This guide describes how to perform basic and advanced metadata management tasks using the SAS Open Metadata Interface. It also provides sample usage scenarios that describe how to use the metadata types in the SAS Metadata Model to store metadata.

The SAS Open Metadata Interface uses XML as its transport language. If you are unfamiliar with XML, see the W3C XML Specifications at `www.w3.org/TR/1998/REC-xml-19980210` before reading this guide.

The examples is this guide show how to issue calls without regard to the programming environment. For details about writing SAS Open Metadata Interface clients, see **SAS 9.1 Open Metadata Interface: Reference.**

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Introduction

The SAS Open Metadata Interface is an application programming interface that provides a set of methods and metadata types for reading and writing metadata about application elements. The SAS Open Metadata Interface can also be used to do such things as register repositories, define dependencies between them, define access controls for repositories and metadata objects, and to perform version control on repositories, but these are secondary tasks. The primary purpose of the SAS Open Metadata Interface is to enable you to read and write metadata.

To read or write a metadata object, you must pass a string of properties that describe that object to the appropriate SAS Open Metadata Interface method. This string identifies the appropriate metadata type to use to represent the application element and supplies or queries specific values for the object's attributes and associations.

This guide is divided into three parts:

- *Interface Usage* describes how to use the application programming interface to perform basic metadata management tasks. It identifies the correct method to use to add, query, update, and delete metadata objects and their associations.
- *Advanced Tasks* describes how to perform more advanced metadata management tasks, such as defining relationships between repositories, implementing an object locking scheme, and using change management methods, among other things.
- *Model Usage Scenarios* contains usage scenarios that describe the metadata types for representing common application elements. Its purpose is to help you identify the correct metadata type to use to represent an application element.

Prerequisite Information

In order to successfully use SAS Open Metadata Interface methods, you must understand how to use the metadata types documentation in the **SAS 9.1 Open Metadata Interface: Reference**. Review Understanding Metadata Types in the reference carefully before attempting to follow the examples in this guide. Metadata types for application elements are described in SAS Namespace Types.

The SAS Open Metadata Interface enables you to instantiate objects for method parameters and issue method calls directly from the client, or to define all of a method's parameters in an XML string and submit it to the server via a generic DoRequest method. The first approach is referred to as the standard interface. The second is referred to as the DoRequest method. The examples in this guide use the DoRequest method. For detailed information about the interfaces, see Call Interfaces in the **SAS 9.1 Open Metadata Interface: Reference**. Method calls formatted for the DoRequest method can be passed in Java, Visual Basic, C++ clients, and in PROC METADATA.

For reference information about the methods described in this guide, see IOMI Class in the **SAS 9.1 Open Metadata Interface: Reference**.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Adding Metadata Objects

Overview

The SAS Open Metadata Interface allows you to create metadata objects and their associations directly or to create the metadata objects first and add associations to them later.

You create objects with the AddMetadata method. With AddMetadata, you can

- create an object
- create an object and an association to an existing object
- create an object, an association, and the associated object.

You add associations to existing objects by using the UpdateMetadata method. This topic describes how to create metadata objects with the AddMetadata method. For information about adding associations to existing objects, see Updating Metadata Objects.

Concepts

To create a metadata object, you must supply a metadata property string that defines the object to be created, specify an identifier that indicates where the object is to be stored, and specify the namespace (set of metadata types) in which to process the request. For example,

- to create a metadata object describing a repository, you specify a metadata property string defining the repository's properties, you specify the repository manager identifier as the storage location, and you specify the REPOS namespace.
- to create a metadata object describing an application element, you specify a metadata property string defining the item's properties, you specify a repository identifier as the storage location, and you specify the SAS namespace.

At a minimum, the metadata property string must specify the metadata type and any required properties of the object to be created. Most metadata types have required attributes. Some also have required associations. Refer to the metadata type documentation in the **SAS 9.1 Open Metadata Interface: Reference** for information about the required properties of the metadata object that you are creating. Information about repository metadata types is in REPOS Namespace Types. Information about application metadata types is in SAS Namespace Types. Although it is not required, it is recommended that you supply values for all of an object's attributes and as many associations as possible in order to make your metadata more meaningful.

The metadata property string is formatted in XML and is submitted to the metadata server in the *inMetadata* parameter of the AddMetadata method. For information about the format of the property string, see Constructing a Metadata Property String in the **SAS 9.1 Open Metadata Interface: Reference**.

As previously stated, the AddMetadata method can be used to create an object; to create an object and an association to an existing object; or to create an object, an association, and the associated object. By default, associated objects are created in the same repository. However, if relationships have been defined between repositories, associations can be made to objects in another repository. An association that is made to an object that exists or is created in another repository is referred to as a **cross−repository reference**. For more

information, see Creating Relationships Between Repositories.

The AddMetadata method creates a new object for every object described in the *inMetadata* parameter unless the ObjRef attribute is used to specify the object's instance identifier. That is, the metadata server will assign a new identifier whether you omit the Id attribute, specify the Id attribute with null value (Id=""), specify a symbolic name in the Id attribute, or specify a real value in the Id attribute, in either the main element or an association subelement. The ObjRef attribute is supported only in an association subelement and instructs the server to create an object reference to the specified object, without modifying the specified object in any other way.

A cross−repository reference is created by specifying a repository identifier along with an object's instance identifier. To create an association to an existing object in another repository, specify the desired object's two−part instance identifier (*Reposid.Objectid*) in the ObjRef attribute. To create an association and a new object in a foreign repository, specify a symbolic name in the Id attribute in the form: Id=*Reposid.SymbolicName*.

Multiple, dependent (associated) objects are created in an AddMetadata request by nesting their object definitions in a metadata property string. For more information, see Creating Multiple Related Metadata Objects. Multiple unrelated objects are created by stacking their metadata property strings in the *inMetadata* parameter. For an example of stacking, see Creating Multiple Unrelated Metadata Objects.

A request that creates a repository object specifies the repository manager identifier A0000001.A0000001 in the *Reposid* parameter. You can determine the available repositories and their unique identifiers by using the GetRepositories method.

Symbolic Names

A symbolic name is an alias preceded by a dollar sign ($) that you assign to a metadata object to enable you to reference a metadata object before it is created. In AddMetadata and UpdateMetadata, symbolic names are used to create associations and new associated objects in other repositories. In AddMetadata, symbolic names also enable you to create multiple unrelated metadata objects and references between them in a single XML request.

When creating an association and a new object in another repository, the symbolic name has the form `reposid.alias` and the symbolic name is specified in the Id= attribute of the association subelement.

When used in an AddMetadata request that creates multiple unrelated objects, the form `alias` is used and the symbolic name is specified in the Id= attribute of the main XML element.

The alias can be any name. At the successful completion of the AddMetadata or UpdateMetadata call, the alias and any references to it are automatically replaced with a real object identifier.

For an example of how symbolic names are used, see Creating Multiple Unrelated Metadata Objects.

Examples

- Creating a Repository Object
- Creating a Metadata Object Representing an Application Element

- Creating a Metadata Object and an Association to an Existing Object
- Creating Multiple Related Metadata Objects
- Creating Multiple Unrelated Metadata Objects
- Creating a Metadata Object and an Association to an Object in Another Repository

Creating a Repository Object

Creating a repository object registers a repository in a metadata server's repository manager. You must register at least one repository in the repository manager before you can create metadata objects representing application elements. The repository object provides the server with information necessary to access the metadata.

The following is an example of an AddMetadata request that creates a repository:

```
<AddMetadata>
  <Metadata>
    <RepositoryBase
      Name="Test repository 1"
      Desc="Repository created to illustrate registering a repository using AddMetadata."
      Path="C:\testdat\repository1"/>
   </Metadata>
   <Reposid>A0000001.A0000001</Reposid>
   <NS>REPOS</NS>
   <!-- OMI_TRUSTED_CLIENT flag -->
   <Flags>268435456</Flags>
   <Options/>
 </AddMetadata>
```

In the method call:

- The <AddMetadata> element identifies the method.
- The <Metadata> element contains the metadata property string. In this string,
  RepositoryBase
  is the metadata type used to define a repository object.
  Name
  is a unique name for the repository.
  Desc
  is an optional description of the repository.
  Path
  is the path to the repository directory. The pathname must be unique and point to an existing directory.
- The <Reposid> element specifies the storage location. **A0000001.A0000001** signifies the repository manager.
- The <NS> element identifies the namespace in which to process the request. The **REPOS** namespace contains repository metadata types.
- The <Flags> element supplies special processing instructions. The OMI_TRUSTED_CLIENT flag, **268435456**, is required to write a metadata object.
- The <Options> element is blank. *Options* is a required parameter in all SAS Open Metadata Interface methods; however, AddMetadata currently does not support any options.

The output from the call is returned in the DoRequest method's *outMetadata* parameter. Here is an example of the output received from the server:

```
<!-- Using the ADDMETADATA method. -->

<RepositoryBase Name="Test repository 1" Desc="Repository created to illustrate
  registering a repository using AddMetadata." Path="C:\testdat\repository1"
  Id="A0000001.A53TPPVI" Access="0"/>
```

The output string mirrors the input string; however, the server assigns a two−part object identifier in the form *Reposmgrid.Reposid* to the object in its Id attribute. You will specify this identifier in the *Reposid* parameter of AddMetadata calls to create metadata objects representing application elements. You can also use the identifier to create relationships between repositories. For more information, see Creating Relationships Between Repositories.

Creating a Metadata Object Representing an Application Element

The following example creates a metadata object describing a SAS library. A SAS library is represented in the SAS Open Metadata Model by the SASLibrary metadata type.

```
<AddMetadata>
 <Metadata>
   <SASLibrary
      Name="NW Sales"
      Desc="NW region sales data"
      Engine="base"
      IsDBMSLibname="0"
      Libref="nwsales"/>
 </Metadata>
 <Reposid>A0000001.A53TPPVI</Reposid>
 <NS>SAS</NS>
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

In this method call, note:

- The <Reposid> element specifies a repository identifier.
- The <NS> element specifies the SAS namespace.
- The <Flags> element specifies the OMI_TRUSTED_CLIENT flag (268435456).
- The metadata property string specifies
    - ♦ SASLibrary as the metadata type of the object being created
    - ♦ Name, Desc, Engine, IsDBMSLibname, and Libref as XML attributes
    - ♦ that the string does not specify any nested XML elements.

Here is an example of the output received from the server:

```
<!-- Using the ADDMETADATA method. -->

<SASLibrary Name="NW Sales" Desc="NW region sales data" Engine="base"
  IsDBMSLibname="0" Libref="nwsales" Id="A53TPPVI.A1000001"/>
```

The output string mirrors the input string except that a two−part object instance identifier in the form *Reposid.Objectid* is assigned to the new object. In the following section, this identifier will be used to create an association to the object.

Creating a Metadata Object and an Association to an Existing Object

The following XML string creates a ResponsibleParty object and associates it with the SASLibrary object created in the previous example. The ResponsibleParty metadata type is used to associate a set of Person objects with a particular role or responsibility. This ResponsibleParty object is created with the role of "Owner".

```
<AddMetadata>
 <Metadata>
   <ResponsibleParty Name="LibraryAdministrator" Desc="NW Region Sales Data"
      Role="Owner">
     <Objects>
       <SASLibrary ObjRef="A53TPPVI.A1000001" Name="NW Sales"/>
     </Objects>
   </ResponsibleParty>
 </Metadata>
 <Reposid>A0000001.A53TPPVI</Reposid>
 <NS>SAS</NS>
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

The method parameters are the same as in the previous example. In the metadata property string, note:

- ResponsibleParty is the metadata type; Name, Desc, and Role are its attributes.
- Objects is the association name and is specified as a nested element.
- The ObjRef attribute identifies the SASLibrary object in the association subelement. The ObjRef attribute instructs the metadata server to create an association to an existing metadata object.

Here is an example of the output received from the server:

```
<!-- Using the ADDMETADATA method. -->

<ResponsibleParty Name="LibraryAdministrator" Desc="NW Region Sales Data"
  Role="Owner" Id="A53TPPVI.A2000001">
  <Objects>
    <SASLibrary ObjRef="A53TPPVI.A1000001" Name="NW Sales"/>
  </Objects>
</ResponsibleParty>
```

The output string mirrors the input string except that a two−part object instance identifier is assigned to the new object.

Creating Multiple Related Metadata Objects

The following XML string creates a table object and associates it with the SASLibrary object. The Open Metadata Model supports several metadata types for describing tables. Here the PhysicalTable metadata type is used to represent a table that is materialized in a file system. A PhysicalTable object is associated to a SASLibrary object via a TablePackages association. The request creates four column objects that describe the contents of the table as nested subelements.

```
<AddMetadata>
 <Metadata>
   <PhysicalTable Name="Sales Offices" Desc="Sales offices in NW region">
```

```
      <TablePackages>
        <SASLibrary ObjRef="A53TPPVI.A1000001" Name="NW Sales"/>
      </TablePackages>
      <Columns>

       <Column
          Name="City"
          Desc="City of Sales Office"
          ColumnName="City"
          SASColumnName="City"
          ColumnType="12"
          SASColumnType="C"
          ColumnLength="32"
          SASColumnLength="32"
          SASFormat="$Char32."
          SASInformat="$32."/>

       <Column
          Name="Address"
          Desc="Street Address of Sales Office"
          ColumnName="Address"
          SASColumnName="Street_Address"
          ColumnType="12"
          SASColumnType="C"
          ColumnLength="32"
          SASColumnLength="32"
          SASFormat="$Char32."
          SASInformat="$32."/>

       <Column
          Name="Manager"
          Desc="Name of Operations Manager"
          ColumnName="Manager"
          SASColumnName="Manager"
          ColumnType="12"
          SASColumnType="C"
          ColumnLength="32"
          SASColumnLength="32"
          SASFormat="$Char32."
          SASInformat="$32."/>

       <Column
         Name="Employees"
         Desc="Number of employees"
         ColumnName="Employees"
         SASColumnName="Employees"
         ColumnType="6"
         SASColumnType="N"
         ColumnLength="3"
         SASColumnLength="3"
         SASFormat="3.2"
         SASInformat="3.2"/>

       </Columns>
     </PhysicalTable>
  </Metadata>
  <Reposid>A0000001.A53TPPVI</Reposid>
  <NS>SAS</NS>
  <Flags>268435456</Flags>
  <Options/>
</AddMetadata>
```

Here is an example of the output received from the server:

```
<!-- Using the ADDMETADATA method. -->

<PhysicalTable Name="Sales Offices" Desc="Sales offices in NW region" Id="A53TPPVI.A4000001">
 <TablePackages>
  <SASLibrary ObjRef="A53TPPVI.A1000001" Name="NW Sales"/>
 </TablePackages>
 <Columns>
  <Column Name="City" Desc="City of Sales Office" ColumnName="City" SASColumnName="City"
    ColumnType="12" SASColumnType="C" ColumnLength="32" SASColumnLength="32"
    SASFormat="$Char32." SASInformat="$32." Id="A53TPPVI.A5000001">
    <Table>
     <PhysicalTable ObjRef="A53TPPVI.A4000001"/>
    </Table>
  </Column>
  <Column Name="Address" Desc="Street Address of Sales Office" ColumnName="Address"
    SASColumnName="Street_Address" ColumnType="12" SASColumnType="C" ColumnLength="32"
    SASColumnLength="32" SASFormat="$Char32." SASInformat="$32." Id="A53TPPVI.A5000002">
    <Table>
     <PhysicalTable ObjRef="A53TPPVI.A4000001"/>
    </Table>
  </Column>
  <Column Name="Manager" Desc="Name of Operations Manager" ColumnName="Manager"
    SASColumnName="Manager" ColumnType="12" SASColumnType="C" ColumnLength="32"
    SASColumnLength="32" SASFormat="$Char32." SASInformat="$32." Id="A53TPPVI.A5000003">
    <Table>
     <PhysicalTable ObjRef="A53TPPVI.A4000001"/>
    </Table>
  </Column>
  <Column Name="Employees" Desc="Number of employees" ColumnName="Employees"
    SASColumnName="Employees" ColumnType="6" SASColumnType="N" ColumnLength="3"
    SASColumnLength="3" SASFormat="3.2" SASInformat="3.2" Id="A53TPPVI.A5000004">
    <Table>
     <PhysicalTable ObjRef="A53TPPVI.A4000001"/>
    </Table>
  </Column>
</Columns>
</PhysicalTable>
```

The PhysicalTable object has a TablePackages association to the SASLibrary object and a Columns association to the Column objects. For each column, the ColumnName=, ColumnType=, and ColumnLength= attributes describe the names and values of the items in a DBMS; the SASColumnName=, SASColumnType=, and SASColumnLength= attributes indicate their corresponding values in a SAS table. A ColumnType value of 12 indicates VARCHAR; a ColumnType value of 6 indicates FLOAT. For more information about PhysicalTable and Column attributes, see the metadata type documentation in the **SAS 9.1 Open Metadata Interface: Reference**.

Creating Multiple Unrelated Metadata Objects

The following XML string shows an alternate way to format a request that creates multiple objects. The request creates a second table object, named "Sales Associates", and creates objects representing the table's columns by stacking their metadata property strings. A Column object cannot be created without an association to a table object. Therefore, a symbolic name is assigned to the PhysicalTable object to enable the column objects to reference the PhysicalTable object before it is created.

```
<AddMetadata>
 <Metadata>
   <PhysicalTable Id="$Employees" Name="Sales Associates"
     Desc="Sales associates in NW region">
     <TablePackages>
       <SASLibrary ObjRef="A53TPPVI.A1000001" Name="NW Sales"/>
     </TablePackages>
   </PhysicalTable>

   <Column
      Name="Name"
      Desc="Name of employee"
      ColumnName="Employee_Name"
      SASColumnName="Employee"
      ColumnType="12"
      SASColumnType="C"
      ColumnLength="32"
      SASColumnLength="32"
      SASFormat="$Char32."
      SASInformat="$32.">
      <Table>
         <PhysicalTable ObjRef="$Employees"/>
      </Table>
    </Column>

    <Column
      Name="Address"
      Desc="Home Address"
      ColumnName="Employee_Address"
      SASColumnName="Home_Address"
      ColumnType="12"
      SASColumnType="C"
      ColumnLength="32"
      SASColumnLength="32"
      SASFormat="$Char32."
      SASInformat="$32.">
      <Table>
         <PhysicalTable ObjRef="$Employees"/>
      </Table>
    </Column>

    <Column
      Name="Title"
      Desc="Job grade"
      ColumnName="Title"
      SASColumnName="Title"
      ColumnType="12"
      SASColumnType="C"
      ColumnLength="32"
      SASColumnLength="32"
      SASFormat="$Char32."
      SASInformat="$32.">
      <Table>
         <PhysicalTable ObjRef="$Employees"/>
      </Table>
    </Column>

 </Metadata>
 <Reposid>A0000001.A53TPPVI</Reposid>
 <NS>SAS</NS>
 <Flags>268435456</Flags>
```

```
 <Options/>
</AddMetadata>
```

Here is an example of the output received from the server:

```
<!-- Using the ADDMETADATA method. -->

<PhysicalTable Id="A53TPPVI.A4000002" Name="Sales Associates"
  Desc="Sales associates in NW region">
 <TablePackages>
  <SASLibrary ObjRef="A53TPPVI.A1000001" Name="NW Sales"/>
 </TablePackages>
</PhysicalTable>
<Column Name="Name" Desc="Name of employee" ColumnName="Employee_Name"
  SASColumnName="Employee" ColumnType="12" SASColumnType="C" ColumnLength="32"
  SASColumnLength="32" SASFormat="$Char32." SASInformat="$32." Id="A53TPPVI.A5000005">
  <Table>
   <PhysicalTable ObjRef="A53TPPVI.A4000002"/>
  </Table>
</Column>
<Column Name="Address" Desc="Home Address" ColumnName="Employee_Address"
  SASColumnName="Home_Address" ColumnType="12" SASColumnType="C" ColumnLength="32"
  SASColumnLength="32" SASFormat="$Char32." SASInformat="$32." Id="A53TPPVI.A5000006">
  <Table>
   <PhysicalTable ObjRef="A53TPPVI.A4000002"/>
  </Table>
</Column>
<Column Name="Title" Desc="Job grade" ColumnName="Title" SASColumnName="Title"
  ColumnType="12" SASColumnType="C" ColumnLength="32" SASColumnLength="32"
  SASFormat="$Char32." SASInformat="$32." Id="A53TPPVI.A5000007">
  <Table>
   <PhysicalTable ObjRef="A53TPPVI.A4000002"/>
  </Table>
</Column>
```

Note that the symbolic name is replaced by a repository identifier in the output.

Creating a Metadata Object and an Association to an Object in Another Repository

The following XML string creates another repository, creates a Person object in the repository, and associates the Person object with the ResponsibleParty object created in the second example. A Person object has a Responsibilities association to a ResponsibleParty object.

Before a reference can be created between two repositories, a relationship must have been defined between the repositories. We show how to create a relationship here. For more information about the relationships between repositories, see Creating Relationships Between Repositories.

The following example creates the second repository:

```
<AddMetadata>
  <Metadata>
    <RepositoryBase
      Name="Test repository 2"
      Desc="Second test repository."
      Path="C:\testdat\repository2">
```

```
      <DependencyUses>
       <RepositoryBase ObjRef="A0000001.A53TPPVI" Name="Test repository 1"/>
      </DependencyUses>
    </RepositoryBase>
   </Metadata>
   <Reposid>A0000001.A0000001</Reposid>
   <NS>REPOS</NS>
   <!-- OMI_TRUSTED_CLIENT flag -->
   <Flags>268435456</Flags>
   <Options/>
</AddMetadata>
```

Here is an example of the output received from the server:

```
<!-- Using the ADDMETADATA method. -->

<RepositoryBase Name="Test repository 2" Desc="Second test repository."
  Path="C:\testdat\repository2" Id="A0000001.A5KD78HW" Access="0">
  <DependencyUses>
    <RepositoryBase ObjRef="A0000001.A53TPPVI" Name="Test repository 1"/>
  </DependencyUses>
</RepositoryBase>
```

The metadata property string specifies a DependencyUses association between Test repository 2 and Test repository 1. This means that information about any cross−repository references created between the two repositories will be stored in Test repository 2.

The following XML string creates the Person object in Test repository 2.

```
<AddMetadata>
 <Metadata>
    <Person Name="John Doe" Desc="NW Region DB Administrator">
       <Responsibilities>
        <ResponsibleParty ObjRef="A53TPPVI.A2000001" Name="LibraryAdministrator"/>
       </Responsibilities>
    </Person>
 </Metadata>
 <Reposid>A0000001.A5KD78HW</Reposid>
 <Ns>SAS</Ns>
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

Note that the object instance identifier in the <Metadata> element and the repository identifier in the <Reposid> element specify different repository identifiers.

Here is an example of the output received from the server:

```
<!-- Using the ADDMETADATA method. -->

<Person Name="John Doe" Desc="NW Region DB Administrator" Id="A5KD78HW.A1000001">
 <Responsibilities>
   <ResponsibleParty ObjRef="A53TPPVI.A2000001" Name="LibraryAdministrator"/>
 </Responsibilities>
</Person>
```

Person object A5KD78HW.A1000001 was successfully created in repository A0000001.A5KD78HW.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Overview of Querying Metadata Objects

The SAS Open Metadata Interface provides methods to get information about namespaces, repositories, and metadata objects.

Namespaces

A *namespace* refers to the metadata model that can be accessed by the SAS Metadata Server. The SAS Open Metadata Interface is shipped with two namespaces. The "REPOS" namespace contains repository metadata types, and the "SAS" namespace contains all other metadata types. Other applications may define additional metadata types in one or more special−purpose namespaces.

The GetNamespaces method enables you to identify the namespaces defined in a particular repository manager. If a special−purpose namespace has been defined, you can use the GetTypes and GetTypeProperties methods to list the types in the namespace and their properties. Then you can use the GetSubtypes and IsSubtypeOf methods to determine the type hierarchy, if one exists. Together, these methods are referred to as *management methods* because they enable you to get information about the metadata environment. This information provides useful background information when preparing to create metadata objects.

For reference information about the GetNamespaces, GetTypes, GetTypeProperties, GetSubtypes, and IsSubtypeOf methods, see IOMI (OMI) Class in the **SAS 9.1 Open Metadata Interface: Reference**.

Repositories

A *repository* is a collection of metadata that is stored and can be accessed from a central location. However, before you can store metadata in a repository, you must store metadata about the repository itself, including information about its physical location, the engine used to access the data, and other pertinent details. Creation of a repository metadata object *registers* the repository in the repository manager and enables the repository to be accessed by the SAS Open Metadata Interface. You can determine the repositories that have been registered in a repository manager by using the GetRepositories method. The GetRepositories method lists the repository identifier and name of the the repositories registered in a particular repository manager. A repository identifier is required to add metadata to a repository. Therefore, the GetRepositories method is referred to as a *repository method*.

Other SAS Open Metadata Interface methods enable you to act on repository objects to do such things as impose access controls, define repository dependencies, and enforce change management processes. These aspects of a repository are queried using the same methods used to read application metadata objects. For more information, see the "Metadata Objects" section.

For reference information about the GetRepositories method, see IOMI (OMI) Class in the **SAS 9.1 Open Metadata Interface: Reference**.

Metadata Objects

A *metadata object* consists of a group of attributes and associations that uniquely describe a particular application object or repository. The SAS Open Metadata Interface provides two methods for reading metadata objects:

- the GetMetadata method retrieves the properties of a specific metadata object

- the GetMetadataObjects method retrieves all objects of a specified metadata type.

In addition, each of the methods supports flags and options that enable you to expand or to filter the metadata request. For more information, see Overview of Retrieving Objects of a Specified Metadata Type, Retrieving Properties for a Specific Metadata Object, and Using Templates.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Listing the Metadata Types in a Namespace

The SAS Open Metadata Interface provides the GetTypes method for listing metadata types. The default behavior of the GetTypes method is to list the metadata types that are defined in a given namespace. The following is an example of a GetTypes call that is formatted for the *inMetadata* parameter of the DoRequest method:

```
<GetTypes>
   <Types/>
   <Ns>SAS</Ns>
   <Flags/>
   <Options/>
</GetTypes>
```

The *Ns*, *Flags*, and *Options* parameters are input parameters. *Types* is the output parameter. This example does not specify any flags or options. The *Types* parameter will return the following information for each metadata type in the named namespace ("SAS" in this example). For brevity, only the first line of the output is shown.

```
<Type Id="AbstractExtension" Desc="Abstract Extension" HasSubtypes="1"/>
```

- Id= specifies a metadata type
- Desc= is a system−supplied description of the metadata type
- HasSubtypes is a boolean indicator that identifies whether a metadata type has subtypes. A value of 1 indicates that the type has subtypes. A value of 0 indicates that it does not.

You can use the GetSubtypes method to list the type's subtypes. Here is an example of a request to list the subtypes of the AbstractExtension metadata type:

```
<GetSubtypes>
  <Supertype>AbstractExtension</Supertype>
   <Subtypes/>
   <Ns>SAS</Ns>
   <Flags>0</Flags>
   <Options/>
</GetSubtypes>
```

The metadata type AbstractExtension is passed to the metadata server in the *Supertype* parameter. Here is an example of the output received from the server:

```
<!-- Using the GETSUBTYPES method. -->

<SubTypes>
 <Type Id="Extension" Desc="Object Extensions" HasSubtypes="0"/>
 <Type Id="NumericExtension" Desc="Numeric Extension" HasSubtypes="0"/>
</SubTypes>
```

The AbstractExtension metadata type has two subtypes: Extension and NumericExtension. A query specifying the AbstractExtension metadata type will return objects of both subtypes.

For more information about metadata types in the SAS namespace, see SAS Namespace Types in the **SAS 9.1 Open Metadata Interface: Reference**. For more information about the methods, see GetTypes and GetSubtypes in the IOMI (OMI) Class section of the **SAS 9.1 Open Metadata Interface: Reference**.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Listing the Available Repositories

You can list the repositories registered on a metadata server by using the GetRepositories method. The following is an example of a GetRepositories call that is formatted for the *inMetadata* parameter of the DoRequest method.

```
<GetRepositories>
   <Repositories/>
   <Flags>0</Flags>
   <Options/>
 </GetRepositories>
```

The request retrieves the Id and Name attributes of all repositories registered on the current metadata server. Here is an example of the output received from the server:

```
<!-- Using the GETREPOSITORIES method. -->

<Repositories>
 <Repository Id="A0000001.A53TPPVI" Name="Test repository 1"
   Desc="Repository created to illustrate registering a repository using AddMetadata."
   DefaultNS="SAS"/>
 <Repository Id="A0000001.A5KD78HW" Name="Test repository 2"
   Desc="Second test repository." DefaultNS="SAS"/>
</Repositories>
```

The current repository manager has two repositories registered in it: Test repository 1 and Test repository 2.

To list the repositories' Access and PauseState attributes, set the OMI_ALL flag (1) in the GetRepositories request. The Access attribute stores a repository's default access mode. A value of 0 indicates a repository is available for read, write, and update access. A value of 1 indicates it is read–only. The PauseState attribute indicates whether a repository has been temporarily paused to another state. A repository that has a state value in the PauseState attribute will remain in that state until it is resumed. A repository is generally paused in order to perform a backup.

To request attributes for a specific repository, use the GetMetadata method.

For more information about the metadata types used to represent repository objects, see REPOS Namespace Types in the **SAS 9.1 Open Metadata Interface: Reference**. For more information about the GetRepositories method and GetMetadata methods, see GetRepositories and GetMetadata in the **SAS 9.1 Open Metadata Interface: Reference**.

Previous Page | Next Page | Top of Page

Listing the Metadata Types in a Repository

After adding objects, you can verify the metadata types that have been added to a repository by using the GetTypes method with the OMI_SUCCINCT (2048) flag. When used with OMI_SUCCINCT and its corresponding <Reposid> element, the GetTypes method returns the metadata types for which objects have been defined in a specific repository.

The following input XML string issues a GetTypes call that includes OMI_SUCCINCT:

```
<GetTypes>
   <Types/>
   <NS>SAS</NS>
   <!-- specify the OMI_SUCCINCT flag -->
   <Flags>2048</Flags>
   <Options>
     <!-- include <Reposid> XML element and a repository identifier -->
     <Reposid>A0000001.A53TPPVI</Reposid>
   </Options>
</GetTypes>
```

The *NS*, *Flags*, and *Options* parameters are input parameters. The *Types* parameter is an output parameter. The *Options* parameter passes a <Reposid> XML element that identifies the target repository, which in this case is Test repository 1.

The following is example output received from the server:

```
<!-- Using the GETTYPES method. -->

<Types>
<Type Id="Column" Desc="Columns" HasSubtypes="0"/>
<Type Id="PhysicalTable" Desc="Physical Table" HasSubtypes="1"/>
<Type Id="ResponsibleParty" Desc="Responsible Party" HasSubtypes="0"/>
<Type Id="SASLibrary" Desc="SAS Library" HasSubtypes="0"/>
</Types>
```

Test repository 1 contains metadata objects of four metadata types: Column, PhysicalTable, ResponsibleParty, and SASLibrary.

- Id= specifies the metadata type.
- Desc= returns a system−supplied description of the metadata type.
- When OMI_SUCCINCT is set, HasSubtypes has no meaning.

To list actual objects of each type, you must use the GetMetadataObjects method. For more information about the GetTypes method, see GetTypes in the **SAS 9.1 Open Metadata Interface: Reference**. For usage information about the GetMetadataObjects method, see Overview of Retrieving Objects of a Specified Metadata Type.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Overview of Retrieving Objects of a Specified Metadata Type

The SAS Open Metadata Interface provides the GetMetadataObjects method for retrieving metadata objects of a specified metadata type. The default behavior of the GetMetadataObjects method is to retrieve general, identifying information for each object of the specified type in the specified repository. The method also provides flags and options that enable you to expand or to filter the object request.

The following is an example of a GetMetadataObjects call that does not contain flags or options. The call lists the Id and Name attributes of all objects of type PhysicalTable in Test repository 1. The call is formatted for the *inMetadata* parameter of the DoRequest method.

```
<GetMetadataObjects>
<!--Reposid specifies Test repository 1 -->
  <Reposid>A0000001.A53TPPVI</Reposid>
  <Type>PhysicalTable</Type>
  <Objects/>
  <NS>SAS</NS>
  <Flags>0</Flags>
  <Options/>
</GetMetadataObjects>
```

Here is example output received from the server:

```
<!-- Using the GETMETADATAOBJECTS method. -->

<Objects>
<PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices"/>
<PhysicalTable Id="A53TPPVI.A4000002" Name="Sales Associates"/>
</Objects>
```

Test repository 1 has two objects of metadata type PhysicalTable defined.

You can expand a GetMetadataObjects request to retrieve additional properties, to include subtypes of the specified metadata type, and to include additional repositories in the retrieval request.

You filter a GetMetadataObjects request by specifying search criteria.

For reference information about the GetMetadataObjects method, see GetMetadataObjects in the **SAS 9.1 Open Metadata Interface: Reference**. For additional usage information, see:

- Expanding a GetMetadataObjects Request to Retrieve Additional Properties
- Expanding a GetMetadataObjects Request to Include Subtypes
- Expanding a GetMetadataObjects Request to Search Additional Repositories
- Filtering a GetMetadataObjects Request

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Expanding a GetMetadataObjects Request to Retrieve Additional Properties

Overview

You expand a GetMetadataObjects call to retrieve additional attributes by setting the OMI_GET_METADATA (256) flag in the GetMetadataObjects call. The OMI_GET_METADATA flag executes a GetMetadata request for each object that is returned by the GetMetadataObjects method. When OMI_GET_METADATA is set without any other GetMetadata flags, the GetMetadata method returns the Id and Name attributes for each object. Specifying additional GetMetadata flags or a template enables you to retrieve specific properties or categories of properties.

To specify a GetMetadata flag in a GetMetadataObjects request, simply add its value to that of the OMI_GET_METADATA flag and any other GetMetadataObjects flags that you have already set. For example, if OMI_GET_METADATA (256) and OMI_XMLSELECT (128) are already set and you want to set OMI_ALL_SIMPLE (8) to retrieve all of the attributes of each object, add their values together (256+128+8=392) and specify the sum in the Flags parameter. To request specific attributes or associated objects for each object, add the value of OMI_TEMPLATE (4) to the Flags parameter (for example, 256+128+4=388) and supply a template defining the requested items in a <Templates> XML element in the Options parameter. For more information about templates, see Using Templates.

Retrieving All Properties for All Objects

The following is an example of a GetMetadataObjects request that sets the OMI_GET_METADATA and OMI_ALL flags. The OMI_ALL flag will list all attributes and associations for all PhysicalTable objects in Test repository 1.

```
<GetMetadataObjects>
<!-- Reposid parameter specifies Test repository 1 -->
   <Reposid>A0000001.A53TPPVI</Reposid>
   <Type>PhysicalTable</Type>
   <Objects/>
   <NS>SAS</NS>
   <!-- Specify OMI_GET_METADATA(256) + OMI_ALL (1) flags -->
   <Flags>257</Flags>
   <Options/>
</GetMetadataObjects>
```

The following is sample output from such a call.

```
<!-- Using the GETMETADATAOBJECTS method. -->

<Objects>
<PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices" DBMSType=""
```

```
  Desc="Sales offices in NW region" IsCompressed="0" IsEncrypted="0" LockedBy=""
  MemberType="" MetadataCreated="05Feb2002:09:37:00" MetadataUpdated="05Feb2002:09:37:00"
  NumRows="-1" SASTableName="" TableName="">
<AccessControls/>
<Aggregations/>
<AnalyticTables/>
<Changes/>
<Columns>
<Column Id="A53TPPVI.A5000001" Name="City" Desc="City of Sales Office"/>
<Column Id="A53TPPVI.A5000002" Name="Address" Desc="Street Address of Sales Office"/>
<Column Id="A53TPPVI.A5000003" Name="Manager" Desc="Name of Operations Manager"/>
<Column Id="A53TPPVI.A5000004" Name="Employees" Desc="Number of employees"/>
</Columns>
<Documents/>
<Extensions/>
<ExternalIdentities/>
<ForeignKeys/>
<Groups/>
<Implementors/>
<Indexes/>
<Keywords/>
<ModelResults/>
<Notes/>
<PrimaryPropertyGroup/>
<Properties/>
<PropertySets/>
<ReachThruCubes/>
<ResponsibleParties/>
<Roles/>
<SASPasswords/>
<SourceClassifierMaps/>
<SourceTransformations/>
<SpecSourceTransformations/>
<SpecTargetTransformations/>
<TablePackage/>
<TargetClassifierMaps/>
<TargetTransformations/>
<Timestamps/>
<TrainedModelResults/>
<Trees/>
<UniqueKeys/>
<UsedByPrototypes/>
<UsingPrototype/>
</PhysicalTable>
<PhysicalTable Id="A53TPPVI.A4000002" Name="Sales Associates" DBMSType=""
  Desc="Sales associates in NW region" IsCompressed="0" IsEncrypted="0" LockedBy=""
  MemberType="" MetadataCreated="05Feb2002:09:50:56" MetadataUpdated="05Feb2002:09:50:56"
  NumRows="-1" SASTableName="" TableName="">
<AccessControls/>
<Aggregations/>
<AnalyticTables/>
<Changes/>
<Columns>
<Column Id="A53TPPVI.A5000005" Name="Name" Desc="Name of employee"/>
<Column Id="A53TPPVI.A5000006" Name="Address" Desc="Home Address"/>
<Column Id="A53TPPVI.A5000007" Name="Title" Desc="Job grade"/>
</Columns>
<Documents/>
<Extensions/>
<ExternalIdentities/>
<ForeignKeys/>
```

```
<Groups/>
<Implementors/>
<Indexes/>
<Keywords/>
<ModelResults/>
<Notes/>
<PrimaryPropertyGroup/>
<Properties/>
<PropertySets/>
<ReachThruCubes/>
<ResponsibleParties/>
<Roles/>
<SASPasswords/>
<SourceClassifierMaps/>
<SourceTransformations/>
<SpecSourceTransformations/>
<SpecTargetTransformations/>
<TablePackage/>
<TargetClassifierMaps/>
<TargetTransformations/>
<Timestamps/>
<TrainedModelResults/>
<Trees/>
<UniqueKeys/>
<UsedByPrototypes/>
<UsingPrototype/>
</PhysicalTable>
</Objects>
```

OMI_ALL retrieves all attributes and associations for each object, whether a value has been defined for them or not. This is useful when you want to view both actual and potential properties. To limit the request to properties that have values, additionally set the OMI_SUCCINCT (2048) flag. Here is an example of the output when OMI_SUCCINCT is set.

```
<!-- Using the GETMETADATAOBJECTS method. -->

<Objects>
<PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices" Desc="Sales offices in NW region"
  IsCompressed="0" IsEncrypted="0" MetadataCreated="05Feb2002:09:37:00"
  MetadataUpdated="05Feb2002:09:37:00" NumRows="-1">
<Columns>
<Column Id="A53TPPVI.A5000001" Name="City" Desc="City of Sales Office"/>
<Column Id="A53TPPVI.A5000002" Name="Address" Desc="Street Address of Sales Office"/>
<Column Id="A53TPPVI.A5000003" Name="Manager" Desc="Name of Operations Manager"/>
<Column Id="A53TPPVI.A5000004" Name="Employees" Desc="Number of employees"/>
</Columns>
</PhysicalTable>
<PhysicalTable Id="A53TPPVI.A4000002" Name="Sales Associates"
  Desc="Sales associates in NW region" IsCompressed="0" IsEncrypted="0"
  MetadataCreated="05Feb2002:09:50:56" MetadataUpdated="05Feb2002:09:50:56" NumRows="-1">
<Columns>
<Column Id="A53TPPVI.A5000005" Name="Name" Desc="Name of employee"/>
<Column Id="A53TPPVI.A5000006" Name="Address" Desc="Home Address"/>
<Column Id="A53TPPVI.A5000007" Name="Title" Desc="Job grade"/>
</Columns>
</PhysicalTable>
</Objects>
```

The output is considerably shorter.

Retrieving Specific Properties for All Objects

The following is an example of a GetMetadataObjects call that requests specific attributes and associated objects for each PhysicalTable object. The example sets the OMI_GET_METADATA (256) and OMI_TEMPLATES (4) flags and supplies a template defining the metadata to retrieve in a <Templates> XML element.

```
<GetMetadataObjects>
<!-- Reposid parameter specifies Test repository 1 -->
   <Reposid>A0000001.A53TPPVI</Reposid>
   <Type>PhysicalTable</Type>
   <Objects/>
   <NS>SAS</NS>
   <!-- Specify OMI_GET_METADATA(256) + OMI_TEMPLATE (4) flags -->
   <Flags>260</Flags>
   <Options>
    <Templates>
      <PhysicalTable DBMSType="" IsCompressed="" IsEncrypted="" MemberType="">
        <Extensions/>
      </PhysicalTable>
    </Templates>
   </Options>
</GetMetadataObjects>
```

The template requests that the server retrieve the DBMSType=, IsCompressed=, IsEncrypted=, and MemberType= attributes for each of the PhysicalTable objects in repository A53TPPVI. In addition, the template specifies to retrieve any Extension objects that were defined for each PhysicalTable object. Here is the output received from the server:

```
<!-- Using the GETMETADATAOBJECTS method. -->

<Objects>
<PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices" DBMSType="" IsCompressed="0"
  IsEncrypted="0" MemberType="">
<Extensions/>
</PhysicalTable>
<PhysicalTable Id="A53TPPVI.A4000002" Name="Sales Associates" DBMSType="" IsCompressed="0"
  IsEncrypted="0" MemberType="">
<Extensions/>
</PhysicalTable>
</Objects>
```

The server returns the requested properties in addition to the Id and Name attributes that are returned by default.

Retrieving All Attributes for All Objects

The following is an example of a GetMetadataObjects request that sets the OMI_ALL_SIMPLE (8) flag.

```
<GetMetadataObjects>
<!-- Reposid parameter specifies Test repository 1 -->
   <Reposid>A0000001.A53TPPVI</Reposid>
   <Type>Column</Type>
   <Objects/>
   <NS>SAS</NS>
```

```
   <!-- Specify OMI_GET_METADATA (256) + OMI_ALL_SIMPLE (8) _ OMI_SUCCINCT (2048) flags -->
   <Flags>2312</Flags>
   <Options/>
</GetMetadataObjects>
```

Unlike with OMI_ALL, the OMI_SUCCINCT flag has no effect on OMI_ALL_SIMPLE. OMI_ALL_SIMPLE returns all attributes of all objects of the requested type, including those that do not have values. Here is the output received from the server:

```
<!-- Using the GETMETADATAOBJECTS method. -->

<Objects>
<Column Id="A53TPPVI.A5000001" Name="City" BeginPosition="0" ColumnLength="32"
  ColumnName="City" ColumnType="12" Desc="City of Sales Office" EndPosition="0"
  IsDiscrete="0" IsNullable="0" MetadataCreated="05Feb2002:09:37:00"
  MetadataUpdated="05Feb2002:09:37:00" SASColumnLength="32" SASColumnName="City"
  SASColumnType="C" SASExtendedLength="0" SASFormat="$Char32." SASInformat="$32."
  SASPrecision="0" SASScale="0"/>
<Column Id="A53TPPVI.A5000002" Name="Address" BeginPosition="0" ColumnLength="32"
  ColumnName="Address" ColumnType="12" Desc="Street Address of Sales Office"
  EndPosition="0" IsDiscrete="0" IsNullable="0" MetadataCreated="05Feb2002:09:37:00"
  MetadataUpdated="05Feb2002:09:37:00" SASColumnLength="32" SASColumnName="Street_Address"
  SASColumnType="C" SASExtendedLength="0" SASFormat="$Char32." SASInformat="$32."
  SASPrecision="0" SASScale="0"/>
<Column Id="A53TPPVI.A5000003" Name="Manager" BeginPosition="0" ColumnLength="32"
  ColumnName="Manager" ColumnType="12" Desc="Name of Operations Manager" EndPosition="0"
  IsDiscrete="0" IsNullable="0" MetadataCreated="05Feb2002:09:37:00"
  MetadataUpdated="05Feb2002:09:37:00" SASColumnLength="32" SASColumnName="Manager"
  SASColumnType="C" SASExtendedLength="0" SASFormat="$Char32." SASInformat="$32."
  SASPrecision="0" SASScale="0"/>
<Column Id="A53TPPVI.A5000004" Name="Employees" BeginPosition="0" ColumnLength="3"
  ColumnName="Employees" ColumnType="6" Desc="Number of employees" EndPosition="0"
  IsDiscrete="0" IsNullable="0" MetadataCreated="05Feb2002:09:37:00"
  MetadataUpdated="05Feb2002:09:37:00" SASColumnLength="3" SASColumnName="Employees"
  SASColumnType="N" SASExtendedLength="0" SASFormat="3.2" SASInformat="3.2" SASPrecision="0"
  SASScale="0"/>
<Column Id="A53TPPVI.A5000005" Name="Name" BeginPosition="0" ColumnLength="32"
  ColumnName="Employee_Name" ColumnType="12" Desc="Name of employee" EndPosition="0"
  IsDiscrete="0" IsNullable="0" MetadataCreated="05Feb2002:09:50:56"
  MetadataUpdated="05Feb2002:09:50:56" SASColumnLength="32" SASColumnName="Employee"
  SASColumnType="C" SASExtendedLength="0" SASFormat="$Char32." SASInformat="$32."
  SASPrecision="0" SASScale="0"/>
<Column Id="A53TPPVI.A5000006" Name="Address" BeginPosition="0" ColumnLength="32"
  ColumnName="Employee_Address" ColumnType="12" Desc="Home Address" EndPosition="0"
  IsDiscrete="0" IsNullable="0" MetadataCreated="05Feb2002:09:50:56"
  MetadataUpdated="05Feb2002:09:50:56" SASColumnLength="32" SASColumnName="Home_Address"
  SASColumnType="C" SASExtendedLength="0" SASFormat="$Char32." SASInformat="$32."
  SASPrecision="0" SASScale="0"/>
<Column Id="A53TPPVI.A5000007" Name="Title" BeginPosition="0" ColumnLength="32"
  ColumnName="Title" ColumnType="12" Desc="Job grade" EndPosition="0" IsDiscrete="0"
  IsNullable="0" MetadataCreated="05Feb2002:09:50:56" MetadataUpdated="05Feb2002:09:50:56"
  SASColumnLength="32" SASColumnName="Title" SASColumnType="C" SASExtendedLength="0"
  SASFormat="$Char32." SASInformat="$32." SASPrecision="0" SASScale="0"/>
</Objects>
```

The output includes all of the attributes of all Column objects in Test repository 1.

For more information about the flags, see Using IOMI Flags and Summary Table of IOMI Flags in the **SAS 9.1 Open Metadata Interface: Reference**.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Expanding a GetMetadataObjects Request to Include Subtypes

The GetMetadataObjects method supports the OMI_INCLUDE_SUBTYPES (16) flag for efficiently retrieving information about a type's subtypes. A subtype is a metadata type that inherits properties from a supertype. A supertype can have many subtypes. OMI_INCLUDE_SUBTYPES specifies to retrieve all objects of all possible subtypes defined for a supertype in a given repository. This prevents you from having to query each subtype for information individually.

The following is an example of a GetMetadataObjects request that sets OMI_INCLUDE_SUBTYPES and specifies to retrieve all subtypes of supertype DataTable.

```
<GetMetadataObjects>
<! -- Reposid parameter specifies Test repository 1 -->
   <Reposid>A0000001.A53TPPVI</Reposid>
   <Type>DataTable</Type>
   <Objects/>
   <NS>SAS</NS>
   <! -- Specify OMI_INCLUDE_SUBTYPES (16) flag -->
   <Flags>16</Flags>
   <Options/>
</GetMetadataObjects>
```

The DataTable supertype has the following subtypes defined for it in the SAS Open Metadata Model: ExternalTable, JoinTable, PhysicalTable, QueryTable, RelationalTable, and WorkTable. OMI_INCLUDE_SUBTYPES will return metadata objects of these subtypes for which objects have been defined in Test repository 1. Here is an example of the output received from the server:

```
<!-- Using the GETMETADATAOBJECTS method. -->

<Objects>
<PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices"/>
<PhysicalTable Id="A53TPPVI.A4000002" Name="Sales Associates"/>
</Objects>
```

Thus far, Test repository 1 contains two objects of subtype PhysicalTable and no objects of the other subtypes.

To optimize use of this flag, use the hierarchical listing of SAS Namespace Types in the **SAS 9.1 Open Metadata Interface: Reference** to identify supertype/subtype relationships and assess the hierarchical level at which to target your request.

The default behavior of the GetMetadataObjects method is to return the Id and Name properties for all objects that are found. When OMI_INCLUDE_SUBTYPES is set with other SAS Open Metadata Interface flags, the GetMetadataObjects method will retrieve the requested properties for all subtype objects. For information about the use of specific flags, see Expanding a GetMetadataObjects Request to Retrieve Additional Properties.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Expanding a GetMetadataObjects Request to Search Additional Repositories

A GetMetadataObjects request that searches multiple repositories is called a *federated query*. A federated query is a GetMetadataObjects request that sets either or both of the OMI_DEPENDENCY_USES (8192) and OMI_DEPENDENCY_USED_BY (16384) flags to search for objects of a specified type in repositories that have the specified dependency (uses or used by). Before a federated query can be performed, the specified dependency must have been established between the repositories being queried. In Creating a Metadata Object and an Association to an Object in Another Repository, we created a DependencyUsedBy association between Test repository 2 and Test repository 1. This means that Test repository 2 uses Test repository 1. We also created a Person object in Test Repository 2.

The following is an example of a federated query that searches repositories that either use or are used by Test repository 1 for objects of metadata type Person.

```
<GetMetadataObjects>
<!-- Reposid parameter specifies Test repository 1 -->
   <Reposid>A0000001.A53TPPVI</Reposid>
   <Type>Person</Type>
   <Objects/>
   <NS>SAS</NS>
 <!-- Specify OMI_DEPENDENCY_USES (8192) and OMI_DEPENDENCY_USED_BY (16384) flags -->
   <Flags>24576</Flags>
   <Options/>
</GetMetadataObjects>
```

Here is example output received from the server:

```
<!-- Using the GETMETADATAOBJECTS method. -->

<Objects>
<Person Id="A5KD78HW.A1000001" Name="John Doe"/>
```

The query found one Person object, whose object instance identifier indicates that it came from Test Repository 2. Test Repository 1 is used by Test repository 2. For more information about dependency relationships, cross–repository references, and federated queries, see Creating Relationships Between Repositories.

Previous Page | Next Page | Top of Page

Filtering a GetMetadataObjects Request

Overview

The GetMetadataObjects method supports an OMI_XMLSELECT (128) flag to enable you to pass search criteria and filter the objects that are retrieved by the SAS Metadata Server. You specify the search criteria in an <XMLSelect> element in the *Options* parameter. The search syntax enables you to filter the objects that are selected by

- attributes
- association name
- subelement name
- association path
- a combination of the above.

In addition, the syntax supports concatenated attribute and path criteria through the use of logical operators. Concatenation of attribute criteria enables inclusive and exclusive filtering in an object request. For example, you could specify to select objects that have a Name= attribute value of 'John Doe' or 'Jane Doe'. Or you could specify to select objects that have the attribute/value pairs Name='John Doe' and Title='Manager'.

Concatenation of association paths permits filtering of object requests at multiple association levels. For example, you could specify to select Document objects that have a ResponsibleParties association to any Person objects that have a Locations association to a Location object that has an Area= attribute value of 'New York'.

<XMLSelect> Element Form and Search Criteria Syntax

The <XMLSelect> element is specified in the *Options* parameter in the form:

```
<XMLSelect search="criteria"/>
```

The syntax of *criteria* is:

```
object[attribute_criteria][association_path]
```

The brackets denote optional criteria and are also part of the search string specification. For example, a

statement that specifies attribute criteria and association path criteria is specified as:

```
object[attribute_criteria][association_path]
```

A statement that specifies only attribute criteria is specified as:

```
object[attribute_criteria]
```

A statement that specifies only association path criteria is specified as:

```
object[association_path]
```

A statement that specifies a subelement as the criteria is specified as:

```
object
```

Attribute criteria are concatenated by specifying logical operators within the brackets. For example:

```
object[attribute_criteria and|or attribute_criteria]
```

If only attribute criteria are specified, the server will accept search strings that omit the *object* component, as follows:

```
attribute_criteria and|or attribute_criteria
(attribute_criteria and|or attribute_criteria)
```

Path criteria are concatenated by placing a forward−slash character (/) between the path specifications in the brackets. For example:

```
object[association_path1/association_path2]
```

A detailed description of the syntax components is provided in the sections that follow.

Object Component

The *object* component is required for all but attribute criteria and specifies the object class type. Valid values are a metadata type name or an asterisk (*).

- The metadata type name can be the metadata type passed in the GetMetadataObjects *Type* parameter or a subtype. Refer to the SAS Namespace Types documentation in the **SAS 9.1 Open Metadata Interface: Reference** to determine the subtypes of a given metadata type.
- An * applies the specified search criteria to all objects of the metadata type named in the *Type* parameter.

The *object* component is specified once outside of the brackets of any other search components. For example:

```
metadata_type[attribute_criteria][association_path]
```

or

```
metadata_type[attribute_criteria]
```

or

```
*[association_path]
```

Attribute_Criteria Component

The *attribute_criteria* component is optional and enables you to filter the objects that are retrieved to those matching a specified attribute/value pair. The syntax of *attribute_criteria* is:

```
[@attrname cop 'value' lop attribute_criterian]
```

where:

- the @ symbol denotes an attribute name, such as "Name" or "Desc"
- *cop* is a comparison operator. Currently supported values are: =, eq, EQ, ?, contains, CONTAINS.
- '*value*' is a character or numeric string, blank, or . (period −− to denote a missing numeric value), enclosed within single quotation marks. **Note:** Searches of date, time, and datetime values are handled differently than searches of other values. For more information, see Searching Date, Time, and Datetime Values.
- *lop attribute_criterion* is an optional, concatenated *attribute_criteria* string that is appended to the first by the logical operator "and" or "or".

  An example of concatenated attribute criteria is:

  ```
  [@Name = 'John Doe' or @Name = 'Jane Doe']
  ```

  The string is an example of an inclusive search: it instructs the metadata server to select for retrieval all metadata objects that have the Name attribute value of 'John Doe' or 'Jane Doe'.

  Compound attribute criteria are also supported. Use parentheses to control evaluation order. For example:

  ```
  search="*[@ProductName='SAS/CONNECT' and (@Name contains 'test - SAS/CONNECT Server'
    or @Name='test')]"
  ```

  In this example, the expression contained within the parenthesis is evaluated first.

Note: When concatenated criteria are used, the attribute test is applied only if all of the specified attributes are valid for the object. That is, if one of the attribute names in the concatenated attribute string is misspelled, then no objects are selected. If the OMI_INCLUDE_SUBTYPES flag is set, the specified attributes must be valid for both the type and its subtypes.

Association_Path Component

The *association_path* component is optional and it specifies an association name, an associated object, and optional attribute criteria. The syntax of *association_path* is:

```
[association/subelement_name[attribute_criteria]/association_pathn]
```

where:

- *association* is an association name, as specified in the SAS Namespace Types documentation in the **SAS 9.1 Open Metadata Interface: Reference**, and is a required component.
- *subelement_name* is a metadata type or an asterisk, as described in Object Component, and is a required component.
- *attribute_criteria* is an optional attribute_criteria string, as described in Attribute_Criteria Component.
- */association_pathn* is an optional association path concatenated to the first by using a slash (/). The / functions like an "and" operator.

An example of concatenated association paths is:

```
[ResponsibleParty/Persons/Locations/Location[Area='New York']]
```

The concatenation operator is bolded.

Understanding Association Paths

The term "association path" refers to a set of objects that are identified in the form:

```
[association/subelement_name[attribute_criteria]]
```

However, to better understand how the metadata server processes this component, it is important to consider the association path within the context of the complete search criteria syntax, which prepends an object component to the path:

```
object[association/subelement_name[attribute_criteria]]
```

The *object* component sets the scope of the retrieval request:

- If *object* is an asterisk (*), then *association* can be any association name defined for the metadata type specified in the GetMetadataObjects *Type* parameter.
- If *object* is a metadata type, then *association* is an association name that is defined for the specified metadata type.

If the OMI_INCLUDE_SUBTYPES (16) flag is set, * additionally selects objects representing any subtypes of the metadata type specified in the *Type* parameter.

The *subelement_name* component has a similar effect:

- If *subelement_name* is an asterisk, the metadata server selects subelement objects of all subelement types supported by the specified association name.
- If *subelement_name* is a metadata type, the metadata server selects for retrieval only objects of the specified subelement type.

As an example, consider the following search path specifications:

```
*[ReportLocation/*]
Report[ReportLocation/Email]
```

The first request specifies to select all objects of the metadata type specified in the *Type* parameter that have a ReportLocation association, and all objects that qualify as subelement objects.

The second request restricts the retrieval request to the Report metadata type and Email objects that have ReportLocation association to a Report object.

Report is a subtype of Classifier. If Classifier is the metadata type specified in the GetMetadataObjects *Type* parameter and the OMI_INCLUDE_SUBTYPES flag is set, specifying an * in the first object position potentially selects objects of up to 12 subtypes. In addition, the ReportLocation association supports 18 potential subelement types for the Report metadata type. Specifying an asterisk in the second position selects objects of all of these types. When choosing whether to specify a metadata type or an asterisk, consult the SAS Namespace Types documentation. For some metadata types and associations, specifying an asterisk or a metadata type can make a big difference in the number of objects that are selected for retrieval.

The *attribute_criteria* component enables you to qualify the subelement objects that are retrieved to those containing specified attributes. For example, the following request retrieves Report objects that have a ReportLocation association to objects of type Document and also have the attribute TextType="XML".

```
Report[ReportLocation/Document[@TextType='XML']]
```

When attribute criteria are specified in a query that has an * in the *subelement_name* component, the filter is applied to all subelements.

Concatenated path names further filter a retrieval request. For example, the following request selects for retrieval Report objects that have a ReportLocation association to Document objects that have the attribute TextType="XML" and a ResponsibleParties association to a ResponsibleParty object that has the Name= attribute value "Jane Doe."

```
Report[ReportLocation/Document[@TextType='XML']/ResponsibleParties/ResponsibleParty
  [@Name='Jane Doe']]
```

Searching by Date, Time, and Datetime Values

In SAS 9.1, searches by date or by time are not supported. However, the metadata server supports limited datetime queries on the MetadataCreated and MetadataUpdated attributes using two operators:

- LT (or "lt")
- GT (or "gt")

The DATETIME formats that are supported are:

- *ddmmmyyyy:hh:mm:ss.s*
- *ddmmmyyyy:hh:mm:ss*
- a SAS date value that represents a *ddmmmyyyy:hh:mm:ss* value.

Note: The "<" and ">" operators are *not* supported; nor is the DATE format *ddmmmyyyy*.

Datetime queries are supported only in the standard call interface. For more information about SAS Open Metadata Interface call interfaces, see Call Interfaces in the **SAS 9.1 Open Metadata Interface: Reference**. Objects are persisted to disk with a GMT datetime value; therefore, an object created in local time might have

a different datetime value on disk. For example, an object created at '30May2003:**16**:20:01' CST could have a persisted datetime value of '30May2003:**21**:20:01'. To accommodate the storage conversion, the server converts values that you specify in an XMLSELECT search string to GMT for you. However, the datetime values returned by the server will look different than the values that you submitted in the search string.

The following are examples of queries in the supported formats:

```
<XMLSELECT search="*[@MetadataCreated GT '27May2003:09:20:17.2']"/>
<XMLSELECT search="*[@MetadataCreated LT '27May2010:09:20:17']"/>
<XMLSELECT search="*[@MetadataCreated GT '1309907400']"/>
```

In the third example, '1309907400' is the SAS date value for '30May2003:19:03:11' GMT.

To retrieve objects created on a specific date, submit a concatenated datetime query as follows:

```
<XMLSELECT search="*[@MetadataCreated GT '28May2003:00:00:00'
  and @MetadataCreated LT '28May2003:23:59:59.9']"/>
```

This request will retrieve all objects created between 00:00:00 and 23:59:59.9 on May 28, 2003.

Examples

- Filtering Objects By a Single Attribute
- Filtering Objects By Concatenated Attributes
- Filtering Objects By Association Name
- Filtering by Association Name and Attribute Criteria
- Filtering by Concatenated Association Paths
- Specifying the <XMLSelect> Element in a GetMetadataObjects Call

Filtering Objects By a Single Attribute

The following <XMLSelect> element selects all objects that have a Name= attribute value of 'John Doe':

```
<XMLSelect search="*[@Name='John Doe']"/>
```

The following <XMLSelect> element selects Person objects that have a Name= attribute value of 'John Doe':

```
<XMLSelect search="Person[@Name='John Doe']"/>
```

The following <XMLSelect> element selects WorkTable objects that have a missing numeric value in the NumRows attribute:

```
<XMLSelect search="WorkTable[@NumRows='.']"/>
```

The following <XMLSelect> element selects WorkTable objects that have a blank string in the MemberType attribute:

```
<XMLSelect search="WorkTable[@MemberType='']"/>
```

Filtering Objects By Concatenated Attributes

The following <XMLSelect> element selects objects that have either the Name= attribute value of 'John Doe' or 'Jane Doe':

```
<XMLSelect search="*[@Name='John Doe' OR @Name='Jane Doe']"/>
```

It is an example of an exclusive search. It will return objects that have either of these two names in the Name attribute. The logical operators AND and OR can be specified in uppercase or lowercase letters.

Filtering Objects By Association Name

The following <XMLSelect> element selects objects that have any objects associated with them through the ResponsibleParties association:

```
<XMLSelect search="*[ResponsibleParties/*]"/>
```

It is an example of a simple Path specification (**association/object**).

Filtering by Association Name and Attribute Criteria

The following <XMLSelect> element selects any objects that have a Role= attribute value of OWNER associated with them through the ResponsibleParties association:

```
<XMLSelect search="*[ResponsibleParties/*[@Role='OWNER']]"/>
```

It is an example of a Path specification with embedded attribute criteria (**association/object[criteria]**).

Filtering by Concatenated Association Paths

The following <XMLSelect> element selects objects owned by a Person object with the Name= attribute value of 'John Doe':

```
<XMLSelect search="*[ResponsibleParties/*[@Role='OWNER']/Persons/Person[@Name='John Doe']]"/>
```

The first path selects objects that have a Role= attribute value of OWNER associated with them through the ResponsibleParties association. The second path selects Person objects that have a Name= attribute value of 'John Doe' associated with them through the Persons association name (**association/object[criteria]/association/object[criteria]**).

The following <XMLSelect> element selects objects owned by any type of object with a Name= attribute value of 'John Doe':

```
<XMLSelect search="*[ResponsibleParties/*[@Role='OWNER']/Persons/*[@Name='John Doe']]"/>
```

This request is identical to the preceding example, except that an asterisk is substituted for the Person object to signify any object in the second path.

Specifying the <XMLSelect> Element in a GetMetadataObjects Call

The following example illustrates how the <XMLSelect> element is passed in a GetMetadataObjects call:

```
<GetMetadataObjects>
  <Reposid>A0000001.A50TC1Z2</Reposid>
  <!-- specify the initial object set -->
  <Type>Document</Type>
  <NS>SAS</NS>
  <!-- specify the OMI_XMLSELECT flag -->
  <Flags>128</Flags>
  <Options>
  <!-- specify the <XMLSelect> search string -->
 <XMLSelect search="*[ResponsibleParties/*[@Role='OWNER']/
     Persons/*[@Name='Joe E. Accountant']]"/>
  </Options>
</GetMetadataObjects>
```

Using OMI_XMLSELECT with Other Flags

By default, XML searches are not case−sensitive and there is no error checking of search syntax . A case−sensitive search can be performed by specifying the OMI_MATCH_CASE (512) flag in addition to the OMI_XMLSELECT flag.

If the OMI_INCLUDE_SUBTYPES (16) flag is specified with OMI_XMLSELECT, the server will retrieve subtypes for the metadata type identified in the first object component of the search criteria. If an asterisk is specified, this selects for retrieval all subtypes of the metadata type indicated in the *Type* parameter. If a metadata type is specified, the server will retrieve subtypes only for that metadata type.

Previous Page | Next Page | Top of Page

Retrieving Properties for a Specific Metadata Object

Overview

To retrieve properties for a specific metadata object, use the GetMetadata method. The default behavior of the GetMetadata method is to return specified properties for the metadata object identified in the *inMetadata* parameter. This includes specified attributes (properties within brackets) for the requested object and general, identifying information for any specified associated objects. For example, the following GetMetadata request retrieves the Name, Desc, ColumnType, and SASFormat attributes and whatever object has a Table association to the Column that has the object instance identifier A53TPPVI.A5000001. The request is formatted for the *inMetadata* parameter of the DoRequest method.

```
<GetMetadata>
 <Metadata>
   <Column Id="A53TPPVI.A5000001" Name="" Desc="" ColumnType="" SASFormat="">
    <Table/>
   </Column>
 </Metadata>
 <NS>SAS</NS>
 <Flags>0</Flags>
 <Options/>
</GetMetadata>
```

Here is an example of the output received from the server:

```
<!-- Using the GETMETADATA method. -->

<Column Id="A53TPPVI.A5000001" Name="City" Desc="City of Sales Office"
  ColumnType="12" SASFormat="$Char32.">
  <Table>
    <PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices"
       Desc="Sales offices in NW region"/>
   </Table>
</Column>
```

The metadata server returns values for the requested attributes and the Id and Name attributes of the associated table object.

To retrieve additional attributes for the associated object, or to retrieve all attributes, all attributes and associations, or associated objects meeting specified criteria, the GetMetadata method supports flags and templates.

Review of Terms

An *attribute* is a characteristic of a metadata type that defines the metadata type. For example, in addition to its Id and Name attributes, the Column metadata type additionally has ColumnName, ColumnLength, ColumnType, and SASFormat attributes.

An *association* is a relationship between a metadata type and another metadata type. For example, the Column metadata type requires a Column object to have a Table association to an object of one of various table metadata types. In addition, it optionally supports associations to metadata types defining keys, indexes, etc.

A *metadata object* is an instance of a metadata type that is uniquely defined by its attribute values and associations.

In this documentation, the term "properties" is used to collectively refer to an object's attributes and associations.

GetMetadata flags and the use of templates are described in the sections that follow.

Retrieving All Properties of a Specified Object

To retrieve all of a metadata object's properties, the GetMetadata method provides the OMI_ALL (1) flag. Here is an example of a GetMetadata request that has the OMI_ALL flag set.

```
<GetMetadata>
 <Metadata>
   <Column Id="A53TPPVI.A5000001"/>
 </Metadata>
 <NS>SAS</NS>
 <!--OMI_ALL flag -->
 <Flags>1</Flags>
 <Options/>
</GetMetadata>
```

Here is an example of the output returned by the metadata server:

```
<!-- Using the GETMETADATA method. -->

<Column Id="A53TPPVI.A5000001" BeginPosition="0" ColumnLength="32"
  ColumnName="City" ColumnType="12" Desc="City of Sales Office" EndPosition="0"
  IsDiscrete="0" IsNullable="0" LockedBy="" MetadataCreated="05Feb2002:09:37:00"
  MetadataUpdated="05Feb2002:09:37:00" Name="City" SASAttribute="" SASColumnLength="32"
  SASColumnName="City" SASColumnType="C" SASExtendedColumnType="" SASExtendedLength="0"
  SASFormat="$Char32." SASInformat="$32." SASPrecision="0" SASScale="0" SortOrder=""
  SummaryRole="">
<AccessControls/>
<AnalyticColumns/>
<Changes/>
<DisplayForKey/>
<Documents/>
<Extensions/>
<ExternalIdentities/>
<ForeignKeyAssociations/>
<Groups/>
<Implementors/>
```

```
<Indexes/>
<Keys/>
<Keywords/>
<MLAggregations/>
<Notes/>
<PrimaryPropertyGroup/>
<Properties/>
<PropertySets/>
<QueryClauses/>
<ResponsibleParties/>
<SourceFeatureMaps/>
<SourceTransformations/>
<SpecSourceTransformations/>
<SpecTargetTransformations/>
<Table>
<PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices"
  Desc="Sales offices in NW region"/>
</Table>
<TargetFeatureMaps/>
<TargetTransformations/>
<Timestamps/>
  <Trees/>
<UniqueKeyAssociations/>
<UsedByPrototypes/>
<UsingAggregations/>
<UsingPrototype/>
</Column>
```

The OMI_ALL flag retrieves all possible attributes and associations for the specified Column object, including those for which values have not been defined. To limit the output to properties that have values, additionally set the OMI_SUCCINCT (2048) flag. Any properties that are null or empty will be omitted from the returned metadata.

Retrieving All Attributes of a Specified Object

To retrieve only an object's attributes, set the OMI_ALL_SIMPLE (8) flag in the GetMetadata call. An example of a GetMetadata request that has the OMI_ALL_SIMPLE flag set follows:

```
<GetMetadata>
 <Metadata>
   <Column Id="A53TPPVI.A5000001"/>
 </Metadata>
 <NS>SAS</NS>
 <!--OMI_ALL_SIMPLE flag -->
 <Flags>8</Flags>
 <Options/>
</GetMetadata>
```

Here is an example of the output returned by the metadata server:

```
<!-- Using the GETMETADATA method. -->

<Column Id="A53TPPVI.A5000001" BeginPosition="0" ColumnLength="32"
  ColumnName="City" ColumnType="12" Desc="City of Sales Office"
  EndPosition="0" IsDiscrete="0" IsNullable="0" LockedBy=""
  MetadataCreated="05Feb2002:09:37:00" MetadataUpdated="05Feb2002:09:37:00"
  Name="City" SASAttribute="" SASColumnLength="32" SASColumnName="City"
```

```
  SASColumnType="C" SASExtendedColumnType="" SASExtendedLength="0"
  SASFormat="$Char32." SASInformat="$32." SASPrecision="0" SASScale="0"
  SortOrder="" SummaryRole=""/>
```

The GetMetadata method retrieves all possible attributes for the specified Column object, including those for which values have not been defined. To limit the output to attributes that have values, additionally set the OMI_SUCCINCT (2048) flag. Any attributes that are null or empty will be omitted from the returned metadata.

Retrieving Properties of Associated Objects

To retrieve additional attributes for associated objects, the GetMetadata method supports templates. A template is an additional metadata property string that is supplied to the metadata server in the *Options* parameter of the GetMetadata method. Any properties identified in the template will be retrieved in addition to the properties requested in the *inMetadata* parameter or requested by GetMetadata flags. The following is an example of a GetMetadata request that submits a template.

```
<GetMetadata>
 <Metadata>
   <Column Id="A53TPPVI.A5000001" Name="" Desc="" ColumnType="" SASFormat="">
    <Table/>
   </Column>
 </Metadata>
 <NS>SAS</NS>
 <!-- OMI_TEMPLATE -->
 <Flags>4</Flags>
 <Options>
  <Templates>
    <Column Id="" ColumnLength="" BeginPosition="" EndPosition=""/>
    <PhysicalTable Id="" Name="" Desc="" DBMSType="" MemberType=""/>
  </Templates>
 </Options>
</GetMetadata>
```

Here is an example of the output returned by the metadata server:

```
<!-- Using the GETMETADATA method. -->

<Column Id="A53TPPVI.A5000001" Name="City" Desc="City of Sales Office"
  ColumnType="12" SASFormat="$Char32." ColumnLength="32" BeginPosition="0"
  EndPosition="0">
<Table>
<PhysicalTable Id="A53TPPVI.A4000001" Name="Sales Offices"
  Desc="Sales offices in NW region" DBMSType="" MemberType=""/>
</Table>
</Column>
```

In addition to the Id and Name attributes of the associated PhysicalTable object, the template returns the table object's Desc, DBMSType, and MemberType attributes. It also retrieves the ColumnLength, BeginPosition, and EndPosition attributes of the specified Column object. For more information about templates, see Using Templates.

Filtering the Associated Objects Returned by a GetMetadata Request

The GetMetadata method supports search criteria on the metadata property strings supplied in both the *inMetadata* parameter and in a template to filter the associated objects that are retrieved by a given request. The search criteria enable you to select to retrieve only associated objects matching a specified attribute=value pair. The search string is specified on the association name element of the XML property string in the form:

```
<Main_element>
  <Association_name search="attribute_criteria"/>
</Main_element>
```

Attribute_criteria has the syntax:

```
search="@attname cop 'value' <lop attribute_criterion>"
```

where:

- the @ symbol denotes an attribute name such as "Name" or "Desc".
- *cop* is one of the following comparison operators: =, eq, EQ, ?, contains, CONTAINS.
- '*value*' is a string or number, enclosed in single quotation marks. In Version 9, date and time values are not supported.
- *lop* is the optional logical operator "and" or "or".
- *attribute_criteria2* is an additional attribute and value pair.

An example of a GetMetadata request that specifies a search string is shown below:

```
<GetMetadata>
 <Metadata>
   <Column Id="A53TPPVI.A5000001" Name="" Desc="">
    <Indexes search="@IsUnique='Yes'"/>
   </Column>
 </Metadata>
 <NS>SAS</NS>
 <Flags>0</Flags>
 <Options/>
</GetMetadata>
```

The request retrieves Index objects associated with the specified Column object that have the value "Yes" in the IsUnique attribute.

When a search string is specified in both the *inMetadata* parameter and a template, the following rules apply:

- If the criteria are specified on different association names, both will be applied.
- If the criteria are specified on the same association name, the criteria specified in the *inMetadata* parameter is applied, and the template criteria is ignored. This is true even if the *inMetadata* parameter does not have a "search" parameter specified.

Retrieving Subtypes

The GetMetadata method provides the OMI_INCLUDE_SUBTYPES (16) flag for retrieving the subtypes of a metadata object. A subtype is a metadata type that inherits properties from another metadata type and stores additional properties. The flag can be useful for retrieving common metadata for related metadata types. OMI_INCLUDE_SUBTYPES cannot be set in a GetMetadata call unless OMI_TEMPLATES (4) is also set. When OMI_INCLUDE_SUBTYPES is set, the GetMetadata method returns the metadata requested in the template for the metadata object identified in the *inMetadata* parameter and any subtypes.

Combining GetMetadata Flags

When OMI_SUCCINCT (2048) is added to any flag combination, only attributes that are non−null and association containers that hold associated objects are returned.

When OMI_ALL (1) and OMI_ALL_SIMPLE (8) are set together, the behavior is the same as if OMI_ALL was set alone. The server returns all possible attributes for the requested object and the Id and Name attributes of all possible associated objects.

When OMI_ALL_SIMPLE (8) is set with OMI_TEMPLATE (4), the server returns all possible attributes for the specified object and any associated objects and attributes for associated objects specified in the template.

When OMI_ALL (1) is set with OMI_TEMPLATE (4), the server returns all possible attributes for the specified object, the Id and Name attributes of all possible associated objects, and any additional attributes for associated objects specified in the template.

When OMI_INCLUDE_SUBTYPES (16) is set with OMI_TEMPLATE (4), the server returns the properties requested in the template for the specified object and all subtypes of the specified object. When OMI_INCLUDE_SUBTYPES is set without OMI_TEMPLATE, it is ignored.

Previous Page | Next Page | Top of Page

Using Templates

- Creating a Template
- Specifying Search Criteria
- Using a Template in a GetMetadataObjects Call
- Additional Template Examples

The GetMetadata method enables you to supply user−defined templates to request metadata for a given object. A template is a metadata property string that is passed to the SAS Metadata Server in the *Options* parameter of the GetMetadata call. The purpose of a template is to expand or to filter the properties requested by other GetMetadata parameters.

The default behavior of the GetMetadata method is to return specified properties for the metadata object identified in the *inMetadata* parameter. This includes specified attributes (properties within brackets) for the requested object and the Id= and Name= attributes for any specified associated objects. That is, any attributes specified for associated (nested) objects in the metadata property string are ignored.

When a template is used, you can request additional attributes for the requested object as well as specific attributes for associated objects and additional associated objects. The properties requested in the template are returned in addition to any properties requested in the *inMetadata* parameter and by SAS Open Metadata Interface flags, unless the template specifies search criteria. The search criteria enables you to filter the associated objects that are retrieved.

Templates can also be specified to retrieve attributes and associated objects in a GetMetadataObjects method call. When a template is passed in a GetMetadataObjects method call, it returns the specified attributes and associations for all metadata objects returned by GetMetadataObjects. For more information, see Using a Template in a GetMetadataObjects Call.

Creating a Template

A template is an XML property string that describes the information that will be returned for a particular metadata type. The string includes the attributes and associations that should be returned for that type. The string should not request properties for nested subelements.

For example, a template for a PhysicalTable object that returns the metadata object Id, date when the table was created, and the columns associated with the table is:

```
<PhysicalTable Id="" CreatedDT="">
   <Columns/>
</PhysicalTable>
```

To retrieve properties for the requested column objects, you would submit an additional template that looks like this:

```
<Column Name="" SASFormat=""/>
```

The metadata server will retrieve the Name and SASFormat attributes for the column objects requested by the first template.

Templates are passed to the metadata server in a <Templates> element in the *Options* parameter of the GetMetadata method. When the <Templates> element is used, the OMI_TEMPLATE (4) flag must also be set to instruct the server to look for the element. The following is an example of a <Templates> element that passes the two templates previously described:

```
<Templates>
   <PhysicalTable Id="" CreatedDT="">
     <Columns/>
   </PhysicalTable>
   <Column Name="" SASFormat=""/>
</Templates>
```

The metadata type specified in a template can be the same one specified in the *inMetadata* parameter of the GetMetadata call, a subtype, or the metadata type of an associated object. In the preceding examples, the object requested in the *inMetadata* parameter is assumed to be the PhysicalTable metadata type.

The order of the templates in the <Templates> element is not important unless the OMI_INCLUDE_SUBTYPES (16) flag is used. The default behavior of the GetMetadata method is to search for objects of every type listed in the <Templates> element, and to retrieve the specified properties if found. When OMI_INCLUDE_SUBTYPES is set, the method cycles through the templates iteratively, beginning with the first template and proceeding in order to the last, and retrieves the specified properties for objects of the specified type and its subtypes. If a template for a subtype is found before one for its supertype, then that template is applied, and no more searching is done for that particular type.

Specifying Search Criteria

The GetMetadata method supports search criteria on the metadata property strings supplied in both the *inMetadata* parameter and in a template to filter the associated objects that are retrieved by a given request. The search criteria enable you to select to retrieve only associated objects matching a specified attribute/value pair. The search string is specified on the association name element of the XML property string.

An example of a template that specifies search criteria is:

```
<PhysicalTable Id="">
   <Columns search="@SASAttribute='XV_A_UNICODE'"/>
</PhysicalTable>
```

This example specifies to retrieve only Column objects that have a value of "XV_A_UNICODE" in the SASAttribute attribute, meaning that the columns they describe support unicode data. For a detailed description of search criteria syntax, and for information about the behavior when a search string is specified in both the *inMetadata* parameter and a template, see Filtering the Associated Objects Returned by a GetMetadata Request.

Using a Template in a GetMetadataObjects Call

The GetMetadataObjects method enables you to execute a GetMetadata call from within a GetMetadataObjects call by setting the OMI_GET_METADATA flag and one or more other GetMetadata flags. When a template is used in a GetMetadataObjects call, the properties specified in the template are returned in addition to any properties requested by any other flags for each of the objects returned for the

GetMetadataObjects method. By including search criteria, a template can also be used to filter the associated objects that are retrieved.

To use a template in a GetMetadataObjects call, simply add the values of the OMI_GET_METADATA (256) and OMI_TEMPLATE (4) flags to the value of any GetMetadataObjects flags specified in the *Flags* parameter and supply a template in the *Options* parameter.

The following is an example of a GetMetadataObjects call that uses a template to request attributes for objects of the requested metadata type. The request retrieves the Id, Name, and UseType attributes of all Document objects in the named repository.

```
<GetMetadataObjects>
   <Reposid>A00000001.A50TC12Z</Reposid>
   <Type>Document</Type>
   <Objects/>
   <NS>SAS</NS>
   <Flags>260</Flags>  <!-- OMI_GET_METADATA(256) + OMI_TEMPLATE(4) -->
   <Options>
      <Templates>
         <Document Id="" Name="" UseType=""/>
      </Templates>
   </Options>
</GetMetadataObjects>
```

The following is an example of a GetMetadataObjects call that uses templates to retrieve attributes for objects of the requested metadata type and their associated objects. In addition to the attributes requested in the previous example, the call retrieves the Id, Name, and Role attributes of all ResponsibleParty objects associated with the Document objects.

```
<GetMetadataObjects>
   <Reposid>A00000001.A50TC12Z</Reposid>
   <Type>Document</Type>
   <Objects/>
   <NS>SAS</NS>
   <Flags>260</Flags>  <!-- OMI_GET_METADATA(256) + OMI_TEMPLATE(4) -->
   <Options>
      <Templates>
         <Document Id="" Name="" UseType="">
           <ResponsibleParties/>
         </Document>
         <ResponsibleParty Id="" Name="" Role=""/>
      </Templates>
   </Options>
</GetMetadataObjects>
```

The following is an example of a GetMetadataObjects call that uses an <XMLSelect> element to filter the initial set of objects that are retrieved and a template to filter the associated objects. The <XMLSelect> element specifies to retrieve only Document objects that contain the word 'Customer' in the Name attribute. The template specifies to retrieve only associated ResponsibleParty objects that have the Role attribute value of OWNER.

```
<GetMetadataObjects>
   <Reposid>A00000001.A50TC12Z</Reposid>
   <Type>Document</Type>
   <Objects/>
   <NS>SAS</NS>
```

```
   <Flags>388</Flags>  <!-- OMI_XMLSELECT(128) + OMI_GET_METADATA(256)
    + OMI_TEMPLATE(4) -->
   <Options>
      <XMLSelect search="@Name contains 'Customer'"/>
      <Templates>
         <Document Id="" Name="" UseType="">
           <ResponsibleParties search="@Role='OWNER'"/>
         </Document>
      </Templates>
   </Options>
</GetMetadataObjects>
```

Additional Template Examples

The following is an example of a GetMetadata request that specifies search criteria in both the *inMetadata* parameter of the method call and in a template. Even though the search criteria are specified on the same association name, both requests are honored because they name different attribute criteria.

```
<GetMetadata>
 <Metadata>
  <Document Id="A5000002.A7000001">
   <ResponsibleParties search="@Name='Jim'"/>
  </Document>
 </Metadata>
 <NS>SAS</NS>
 <Flags>4</Flags>
 <Options>
  <Templates>
   <Document Id="" Name="" UseType="">
    <ResponsibleParties search="@Role='owner'"/>
   </Document>
  </Templates>
 </Options>
 </GetMetadata>
```

The following is an example of a GetMetadata request that filters associated objects at several levels:

```
<GetMetadata>
 <Metadata>
  <Document Id="A5000002.A7000001"/>
 </Metadata>
 <NS>SAS</NS>
 <Flags>4</Flags>
 <Options>
   <Templates>
      <Document Id="" Name="" UseType="">
         <ResponsibleParties search="@Role='OWNER'"/>
      </Document>
      <ResponsibleParty Id="" Name="">
        <Persons search="@Name?'Joe'"/>
      </ResponsibleParty>
      <Person Id="" Name="" Title=""/>
    </Templates>
```

The first template requests attributes for a Document object and filters objects that have a ResponsibleParties association using the Role attribute. The second template requests attributes for the returned ResponsibleParty

objects and specifies a second filter to retrieve only Person objects that have a Name attribute value of 'Joe'. The third template requests the Id, Name, and Title attributes of the returned Person objects.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Updating Metadata Objects

- Overview
- Concepts
- Examples

Overview

The SAS Open Metadata Interface provides the UpdateMetadata method for updating the properties of an existing metadata object. Using the UpdateMetadata method, you can

- modify an existing metadata object's attributes
- add an association between two existing metadata objects
- add an association between an existing and a new metadata object
- delete an association
- modify the properties of an associated object.

The UpdateMetadata method does not allow you to explicitly create new objects. To explicitly create a metadata object, use the AddMetadata method. Note, however, that when you use UpdateMetadata to add an association from an existing object to a new object, the method will create the new associated object when the association is created.

The UpdateMetadata method also cannot be used to explicitly delete a metadata object. However, some dependent objects might be deleted when an association is deleted to enforce cardinality rules. For example, if a table is updated to remove an association to a given column, then the column object, which is dependent on the table, will be deleted as well. However, a column cannot be updated to remove its association to a table and be deleted. To explicitly delete a metadata object, use the DeleteMetadata method. For usage information, see Deleting Metadata Objects.

Concepts

The UpdateMetadata method enables you to add or modify any attribute and association in the metadata type documentation that is not marked as "required for add". An association that is marked as "required for add" typically indicates the object is a dependent object. A required association is modified by deleting the metadata object and creating a new one with the desired properties. For more information, see Deleting Metadata Objects.

To modify an object's attributes, specify the attributes that you wish to modify and the values that you want to assign to them in a metadata property string. The new values will replace the existing values in the repository and any unmodified attributes will remain unchanged.

To add, modify, or delete an association, or to modify an associated object's attributes, specify an association name element and subelement in the metadata property string. For more information about metadata property strings, see Constructing a Metadata Property String in the **SAS 9.1 Open Metadata Interface: Reference**. Include in the association name element the appropriate Function directive and in the association subelement the appropriate identifier. For example:

```
<association_name Function="directive">
  <subelement Identifier="value"/>
</association_name>
```

The metadata server uses the identifier attribute that you specify in the association subelement to determine whether you want to create the object defined in the association subelement, to modify its properties, or to simply add an association. The following table describes the supported identifiers and their behavior when used with the UpdateMetadata method.

Identifiers Supported in the Association Subelement

Identifier	Result
Id="", Id="*Symbolic*", or no identifier	Creates an association and the associated object. For more information about symbolic names, see Creating an Association to an Object in Another Repository.
Id="*realvalue*"	Modifies the specified object with the specified properties, if the object is found; otherwise, the update fails.
ObjRef="*realvalue*"	Creates an association to, but does not modify the properties of the specified object, if the object is found. Otherwise, the update fails.

The Function= attribute specified in the association_name element tells the metadata server how to process the subelements, depending on whether the association is a single or a multiple association. A single association is an association between two metadata objects that has a 0 to 1 or 1 to 1 cardinality in the metadata type documentation. It means that only one association of a given name is supported between the specified metadata objects. A multiple association is one that has a 0 to many or 1 to many (denoted as 0 to * and 1 to *) cardinality and supports many associations of a given name between the specified metadata objects.

The Function= attribute supports the directives shown in Function Directives. If the Function= attribute is omitted, Modify is the default directive for a single association and Merge is the default directive for a multiple association.

Function Directives

Directive	Single	Multiple	Description
Append		*	Appends the specified associations to the specified object's association element list without modifying any of the other associations on the list.
Merge	*	*	Adds and modifies associations to the specified object's association element list as necessary.
Modify	*		Modifies an existing association or adds an association if an association does not already exist.
Replace	*	*	**Single:** Overwrites the existing association with the specified association. **Multiple:** Replaces the existing association element list

			with the specified association elements. Any existing associations that are not represented in the new association list are deleted.
Remove	*	*	Deletes the specified associations from the specified object's association element list without modifying any of the other associations on the list.

Identifier Summary

- To add an association and the associated object, either omit an identifier or supply a null or symbolic identifier (Id="" or Id="*Symbolic*) in the association subelement and define properties for the associated object.
- To add an association to an existing object, specify the associated object's instance identifier in an ObjRef attribute. The ObjRef attribute will prevent you from updating any of the object's other properties.
- To add an association to an existing object and modify the associated object's properties, specify the associated object's instance identifier in the Id attribute and specify the attributes to be modified.

Summary of Function Directives

- To add associations to an existing association element list without modifying the properties of existing associated objects, specify Function="Append".
- To modify the properties of existing associated objects and add new associations as necessary, specify Function="Merge".
- To overwrite an existing association or association element list with a new association or association list, specify Function="Replace".
- To remove an association without modifying any other associations in an existing association element list, specify Function="Remove".
- To modify a single association, specify Function="Modify".

Creating an Association to an Object in Another Repository

To create an association to an existing object in another repository, specify the desired object's two−part instance identifier (*Reposid.Objectid*) in the ObjRef attribute. To create an association and a new object in a foreign repository, specify a symbolic name in the Id attribute in the form: Id=*Reposid.SymbolicName*. For more information about symbolic names, see Symbolic Names in Adding Metadata Objects.

Deleting Associations

Deleting an association does not delete the associated object, unless the association is required for the associated object (the associated object has a 1 to 1 cardinality to the object being updated). If you wish to delete an object's associated objects as well as its associations, you must delete each object individually using the DeleteMetadata method. For more information, see Deleting Metadata Objects.

To view the dependent objects deleted by an update operation, set the OMI_RETURN_LIST (1024) flag.

Examples

The following examples are formatted to the *inMetadata* parameter of the DoRequest method.

- Modifying a Metadata Object's Attributes
- Modifying an Association
- Merging Associations
- Replacing Associations
- Appending Associations

Modifying a Metadata Object's Attributes

The following is an example of an UpdateMetadata request that modifies an object's attributes. The specified attributes and values will replace those stored for the object of the specified metadata type and object instance identifier. The example is formatted for the *inMetadata* parameter of the DoRequest method.

```
<UpdateMetadata>
   <Metadata>
      <SASLibrary
        Id="A53TPPVI.A1000001"
        Engine="oracle"
        IsDBMSLibname="1"/>
   </Metadata>
   <NS>SAS</NS>
   <!-- OMI_TRUSTED_CLIENT flag -->
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

The request submits new values for SASLibrary object A53TPPVI.A1000001's Engine and IsDBMSLibname attributes.

Modifying an Association

The following is an example of an UpdateMetadata request that adds a single association. PhysicalTable object A53TPPVI.A4000001 is updated to add a PrimaryPropertyGroup association to a PropertyGroup object. A PhysicalTable object has a 0 to 1 cardinality to a PropertyGroup object in a PrimaryPropertyGroup association; therefore, creating the association using the Modify statement will have the following results:

- If a PrimaryPropertyGroup association does not already exist, the association will be created.
- If it does exist, the association will be modified.

To replace an existing PrimaryPropertyGroup association with a new one, you would need to specify Function="Replace".

```
<UpdateMetadata>
   <Metadata>
      <PhysicalTable Id="A53TPPVI.A4000001">
         <PrimaryPropertyGroup Function="Modify">
```

```
            <PropertyGroup Id="" Name="Read Options"/>
          </PrimaryPropertyGroup>
       </PhysicalTable>
   </Metadata>
   <NS>SAS</NS>
   <!-- OMI_TRUSTED_CLIENT Flag -->
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

Merging Associations

The following examples illustrate the use of a Merge directive. Merge is the default directive for multiple associations when the Function attribute is omitted from an UpdateMetadata request. It is useful for adding and modifying associations without fear of overwriting any existing associations.

The first example adds unique key and foreign key associations to the table objects created in Adding Metadata. The update request consists of two main parts:

- It adds a UniqueKeys association to PhysicalTable A53TPPVI.A4000002 and identifies the table's Name column (A53TPPVI.A5000005) as the keyed column.
- It associates the UniqueKey with PhysicalTable A53TPPVI.A4000001 by creating a ForeignKeys association. The ForeignKey identifies the table's Employees column (A53TPPVI.A5000004) as its keyed column.

Since the associations are added to any associations already defined for these table and column objects, there is no need to specify the Function directive.

```
<UpdateMetadata>
   <Metadata>
      <PhysicalTable Id="A53TPPVI.A4000002">
         <UniqueKeys>
            <UniqueKey Id="" Name="Sales Associates in NW Region">
              <KeyedColumns>
                <Column Objref="A53TPPVI.A5000005" Name="Name"/>
              </KeyedColumns>
              <ForeignKeys>
                <ForeignKey Id="" Name="Link to Sales Associates table">
                  <Table>
                    <PhysicalTable Objref="A53TPPVI.A4000001" Name="Sales offices in NW Region"
                  </Table>
                  <KeyedColumns>
                    <Column Objref="A53TPPVI.A5000004" Name="Employees"/>
                  </KeyedColumns>
                </ForeignKey>
             </ForeignKeys>
           </UniqueKey>
        </UniqueKeys>
      </PhysicalTable>
   </Metadata>
   <NS>SAS</NS>
   <!-- OMI_TRUSTED_CLIENT Flag -->
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

In the request, note:

- The null Id values in the UniqueKey and ForeignKey elements instruct the metadata server to create objects.
- The ObjRef attribute in the Column elements indicate to create an association to existing objects.

Here is example output received from the server:

```
<!-- Using the UPDATEMETADATA method. -->

<PhysicalTable Id="A53TPPVI.A4000002">
<UniqueKeys>
<UniqueKey Id="A53TPPVI.A8000001" Name="Sales Associates in NW Region">
<KeyedColumns>
<Column Objref="A53TPPVI.A5000005" Name="Name"/>
</KeyedColumns>
<ForeignKeys>
<ForeignKey Id="A53TPPVI.A9000001" Name="Link to Sales Associates table">
<Table>
<PhysicalTable Objref="A53TPPVI.A4000001" Name="Sales offices in NW Region"/>
</Table>
<KeyedColumns>
<Column Objref="A53TPPVI.A5000004" Name="Employees"/>
</KeyedColumns>
<PartnerUniqueKey>
<UniqueKey ObjRef="A53TPPVI.A8000001"/>
</PartnerUniqueKey>
</ForeignKey>
</ForeignKeys>
<Table>
<PhysicalTable ObjRef="A53TPPVI.A4000002"/>
</Table>
</UniqueKey>
</UniqueKeys>
</PhysicalTable>
```

The request created seven associations and two new objects (UniqueKey and ForeignKey).

The following example updates the UniqueKey object created in the previous request to add an association to an additional keyed column and to modify the Name attribute of the first keyed column.

```
<UpdateMetadata>
   <Metadata>
      <UniqueKey Id="A53TPPVI.A8000001">
         <KeyedColumns Function="Merge">
             <Column Id="A53TPPVI.A5000005" Name="EmployeeName"/>
             <Column Objref="A53TPPVI.A5000006" Name="Address"/>
          </KeyedColumns>
      </UniqueKey>
   </Metadata>
   <NS>SAS</NS>
   <!-- OMI_TRUSTED_CLIENT Flag -->
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

In the request, note the use of the Id and Objref attributes. Use of the Id attribute to identify the original keyed column allows the column's properties to be updated. Use of the Objref attribute to identify the new keyed

column creates the association without modifying any of the column object's other attributes.

Here are the results of a GetMetadata call that requests a list of the UniqueKey object's KeyedColumns associations:

```
<!-- Using the GETMETADATA method. -->

<UniqueKey Id="A53TPPVI.A8000001" Name="Sales Associates in NW Region">
<KeyedColumns>
<Column Id="A53TPPVI.A5000005" Name="EmployeeName" Desc="Name of employee"/>
<Column Id="A53TPPVI.A5000006" Name="Address" Desc="Home Address"/>
</KeyedColumns>
</UniqueKey>
```

Replacing Associations

The following example deletes the keyed column association added in the previous example:

```
<UpdateMetadata>
  <Metadata>
     <UniqueKey Id="A53TPPVI.A8000001">
       <KeyedColumns Function="Replace">
           <Column Id="A53TPPVI.A5000005" Name="EmployeeName"/>
       </KeyedColumns>
     </UniqueKey>
  </Metadata>
  <NS>SAS</NS>
  <!-- OMI_TRUSTED_CLIENT Flag -->
  <Flags>268435456</Flags>
  <Options/>
</UpdateMetadata>
```

The Function="REPLACE" directive instructs the server to recreate the KeyedColumns association list with the specified associations, from which the second association is omitted.

Here are the results of a GetMetadata call that requests a list of the UniqueKey object's KeyedColumns associations:

```
<!-- Using the GETMETADATA method. -->

<UniqueKey Id="A53TPPVI.A8000001" Name="Sales Associates in NW Region">
<KeyedColumns>
<Column Id="A53TPPVI.A5000005" Name="EmployeeName" Desc="Name of employee"/>
</KeyedColumns>
</UniqueKey>
```

Appending Associations

The following example adds an association and a new Column object to PhysicalTable A53TPPVI.A4000002 using the APPEND directive.

```
<UpdateMetadata>
   <Metadata>
      <PhysicalTable Id="A53TPPVI.A4000002">
```

```
          <Columns Function="Append">
              <Column Id="" Name="Salary"/>
          </Columns>
       </PhysicalTable>
   </Metadata>
   <NS>SAS</NS>
   <!-- OMI_TRUSTED_CLIENT Flag -->
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

In the example, the null Id value in the association subelement indicates the associated object is to be created. Here is the output received from the server:

```
<!-- Using the UPDATEMETADATA method. -->

<PhysicalTable Id="A53TPPVI.A4000002">
<Columns Function="Append">
<Column Id="A53TPPVI.A500002U" Name="Salary">
<Table>
<PhysicalTable ObjRef="A53TPPVI.A4000002"/>
</Table>
</Column>
</Columns>
</PhysicalTable>
```

The new association is added to the existing association list without affecting other associated objects. Here is the output of a GetMetadata call that lists Column objects associated with PhysicalTable A53TPPVI.A4000002.

```
<!-- Using the GETMETADATA method. -->

<PhysicalTable Id="A53TPPVI.A4000002" Name="Sales Associates">
<Columns>
<Column Id="A53TPPVI.A5000005" Name="EmployeeName" Desc="Name of employee"/>
<Column Id="A53TPPVI.A5000006" Name="Address" Desc="Home Address"/>
<Column Id="A53TPPVI.A5000007" Name="Title" Desc="Job grade"/>
<Column Id="A53TPPVI.A500002U" Name="Salary" Desc=""/>
</Columns>
</PhysicalTable>
```

Note: The OMI_TRUSTED_CLIENT flag is required anytime you update a metadata object.

Previous Page | Next Page | Top of Page

Deleting Metadata Objects

A metadata object is deleted by using the DeleteMetadata method. The DeleteMetadata method does not allow for deletion of specific properties. To delete specific properties, see Updating Metadata Objects.

When a metadata object is deleted, all of its associations and dependent objects are automatically deleted as well. A dependent object is an object that is associated only with the deleted object and has a 1 to 1 cardinality to the object. An example of a dependent object is a Column object. A Column object cannot exist without an association to some type of table object. When a delete is requested upon a table object, all of its associated Column objects are deleted as well. This is referred to as a "cascading delete".

Cardinality rules are enforced on the current repository and on any repositories on which the current repository depends. That is, when a delete request is issued, the request will delete any dependent objects in both the current repository and any repositories to which the repository has a DependencyUses association. If any dependent objects exist in repositories that depend on (have a DependencyUsedBy association to) the current repository, you will need to explicitly delete them.

You can identify associated objects that exist in repositories that have a DependencyUsedBy association by issuing a GetMetadata call that sets the OMI_ALL (1) and OMI_DEPENDENCY_USED_BY (16384) flags before deleting a metadata object. These flags will list general, identifying information for all objects associated with a given object in all repositories to which the host repository has a relationship. Then use the metadata type documentation to determine which associations are to dependent objects.

In addition, you might want to identify metadata objects that do not have a 1 to 1 cardinality but are associated only with the metadata object to be deleted. An example of such an object is a Note defined for a PhysicalTable. Any associated Note objects will remain in a repository after a PhysicalTable object is deleted, even though one or more might have been created solely to describe the PhysicalTable object.

Regardless of the number of objects that may have been deleted during a cascading delete, the content of the *outMetadata* parameter will mirror the content of the *inMetadata* parameter. To include dependent objects that were deleted, set the OMI_RETURN_LIST flag (1024) flag. OMI_RETURN_LIST returns a complete list of deleted object Ids.

Multiple metadata objects can be deleted in a DeleteMetadata call by stacking their object identifiers, as shown below. Nested deletions are not supported.

CAUTION:
The DeleteMetadata method is destructive. Its changes cannot be reversed. When using this method, verify the delete request before issuing the method call.

Example

The following is an example of DeleteMetadata request that deletes a SASLibrary object and a PhysicalTable object.

```
<DeleteMetadata>
<Metadata>
    <SASLibrary Id="A7654321.A2000001"/>
    <PhysicalTable Id="A7654321.A400000C"/>
</Metadata>
<NS>SAS</NS>
```

```
<!--OMI_TRUSTED_CLIENT flag-->
<Flags>268436480</Flags>
<Options/>
</DeleteMetadata>
```

The request deletes the specified objects and any Column objects associated with the PhysicalTable that exist in the current repository and any repositories on which the current repository depends.

Note: The OMI_TRUSTED_CLIENT flag is required to delete metadata objects.

Previous Page | Next Page | Top of Page

Creating Relationships Between Repositories

Overview

The SAS Open Metadata Architecture permits relationships to be defined between repositories.

- A relationship links two repositories so that you can create cross−repository references between the repositories. A *cross−repository reference* is an association that exists between metadata objects in different repositories. An example of a cross−repository reference is a UsingPackages association that is created between a SASLibrary object that exists in a Corporate repository and a SASLibrary object that exists in a Department repository. A cross−repository reference cannot be created unless a relationship exists between the repositories.
- When linking multiple repositories, relationships can be used to create a repository chain. Cross−repositories references will be supported only between repositories that have a direct relationship; however, the existence of the chain enables you to query metadata objects across the repositories. For example, if a repository chain were defined that included a Corporate repository, a Department repository, and a Group repository, you could request to list all metadata objects of type SASLibrary that exist in these repositories from the perspective of any repository in the chain. A query that includes multiple repositories is known as a "federated query".

A relationship is created by defining a *dependency association* between two repositories. A dependency association is an association that marks a repository as "using" or "being used by" another repository.

Note: A user must have server administrative status in order to add a dependency association to a repository. For more information, see "Server Administrative Privileges" in the **SAS 9.1 Metadata Server: Setup Guide**.

Creating Dependency Associations

A dependency association is created by defining a DependencyUses or a DependencyUsedBy association between two repository objects. A repository object is an object that is created using a metadata type from the REPOS namespace of the SAS Open Metadata Interface. The only repository metadata type supported at this time is RepositoryBase.

- A repository that is defined as having a **DependencyUses** association to another repository is said to *use* the other repository.

- A repository that is defined as having a **DependencyUsedBy** association to another repository is said to be *used by* the other repository.

The "direction" of the association is significant in two ways:

- Cross−repository references are stored in the repository that has the DependencyUses association. Therefore, queries of cross−repository references in the repository that has a DependencyUses association are much faster than queries of the repository that has the DependencyUsedBy association. When defining a dependency association between two repositories, define the DependencyUses association for the repository that is expected to be queried the most frequently.
- In a federated query, the "uses" and "used by" relationships determine what repositories are included in the object search. For example, if a repository chain were defined such that a Group repository had a DependencyUses association to a Department repository, and the Department repository had a DependencyUses association to a Corporate repository, a query of each repository would have the results summarized in Table 1. A graphic representation of the repository chain is provided in Figure 1.

Table 1. Repositories Searched in a Federated Query

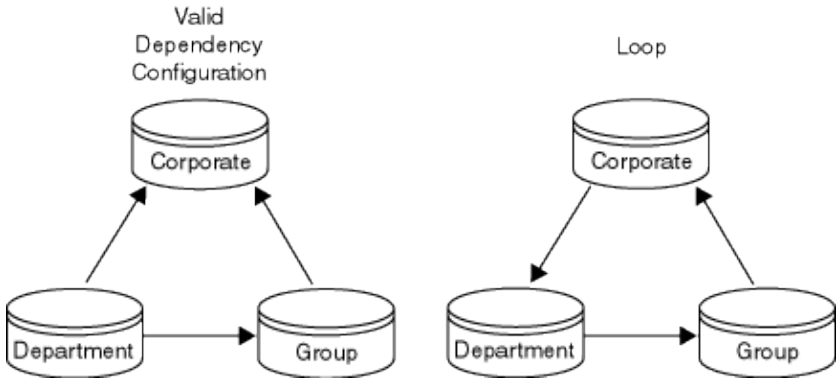
Repository Queried	Flag	Repositories Searched
Group	OMI_DEPENDENCY_USES	All
Group	OMI_DEPENDENCY_USED_BY	Group
Department	OMI_DEPENDENCY_USES	Department, Corporate
Department	OMI_DEPENDENCY_USED_BY	Department, Group
Corporate	OMI_DEPENDENCY_USES	Corporate
Corporate	OMI_DEPENDENCY_USED_BY	All

Figure 1. Representation of a Repository Chain

Rules for creating dependency associations are as follows:

- The repositories in a relationship must use the same metadata server.
- A repository chain cannot contain loops or cycles. Figure 2 illustrates what is considered a valid configuration and what is considered a loop.

Figure 2. Example of a Valid Configuration and a Loop



- When a repository chain is defined, cross−repository references can be created only between repositories that have a direct dependency. For example, if a dependency association is defined between a Corporate repository and a Department repository, and another is defined between the Department repository and a Group repository, then you cannot create cross−repository references between the Corporate repository and the Group repository. In order to create references between them, you must also define a dependency association between the Corporate and the Group repositories.

A dependency association can be created when a repository is created (using the AddMetadata method) or it can be added to an existing repository (using the UpdateMetadata method). For an example of how a dependency association is created, see Example: Adding a Dependency Association to a Repository. To learn more about cross−repository references, see Creating Cross−Repository References. To learn more about federated queries, see Retrieving Objects Across Multiple Repositories.

Creating Cross−Repository References

As previously stated, a cross−repository reference is an association that is created between an object in one repository and an object in another repository. A dependency association must have been established between the repositories in question before a cross−repository reference can be created.

Like other associations, cross−repository references can be added both to objects that already exist, or can be defined when the objects are created.

- To create a cross−repository reference to an existing object, simply specify its repository identifier along with its object instance identifier in the form *reposid.instanceid* in the ObjRef attribute of the XML subelement defining the association.
- To create a cross−repository reference and a new object in a foreign repository, specify the repository identifier and a symbolic name for the object using the Id attribute. Use of the Id attribute with a symbolic name is equivalent to passing a null value in the Id attribute; it indicates to the server that a new object is to be created. A symbolic name is an alias that is preceded by $ (dollar sign). The alias enables you to refer back to the object that is being created before the server assigns it an identifier, and is automatically replaced by the real identifier when the object is created in the specified repository.

For more information, see Example: Creating Cross−Repository References.

Querying Dependency Associations

Information about dependency associations is stored in the repository manager and can be retrieved by using the GetMetadata method or the GetMetadataObjects method.

- Use the GetMetadata method to retrieve dependency information for a specific repository.
- Use the GetMetadataObjects method to retrieve dependency information for all repository objects in the repository manager.

When GetMetadataObjects is used, the OMI_GET_METADATA (256) flag and a GetMetadata flag that enables you to retrieve associations, such as OMI_ALL (1) or OMI_TEMPLATE (4), must also be set. For more information, see Example: Querying Dependency Associations for a Specific Repository and Example: Querying Dependency Associations for All Repositories.

Querying Cross−Repository References

Information about cross−repository references is obtained by using the GetMetadata method. The GetMetadata method retrieves specified attributes and associations for a specific metadata object. Because cross−repository references are stored in the repository that has a DependencyUses association, no additional flags are required to retrieve cross−repository references when a repository that has a DependencyUses association is queried. However, when querying a repository that has a DependencyUsedBy association, the OMI_DEPENDENCY_USED_BY (16384) flag must be set to retrieve the cross−repository references. The OMI_DEPENDENCY_USED_BY flag instructs the metadata server to search the partner repository's association list. For more information, see Example: Retrieving Cross−Repository References.

Retrieving Objects Across Multiple Repositories

When a repository chain has been defined, you can retrieve all objects of a specified type from all or some of the repositories in the chain by using the GetMetadataObjects method. This is referred to as a federated query. To perform a federated query, simply specify the metadata type of the objects that you want to retrieve in the *Type* parameter, then set one or both of the OMI_DEPENDENCY_USES (8192) or OMI_DEPENDENCY_USED_BY (16384) flags.

- OMI_DEPENDENCY_USED_BY instructs the metadata server to search for objects in repositories that have a DependencyUsedBy association in relation to the repository in which the call was issued.
- OMI_DEPENDENCY_USES instructs the metadata server to search repositories that have a DependencyUses association in relation to the repository in which the call was issued.
- When set together, the flags instruct the server to search repositories on both sides of the repository chain.

Note that queries that cross three or more repositories have more overhead associated with them than a query between two repositories. In addition, a query that sets the OMI_DEPENDENCY_USES flag is considerably faster than one that sets the OMI_DEPENDENCY_USED_BY flag.

The default behavior of the GetMetadataObjects method is to return general, identifying information for each object of the specified type. To retrieve additional properties for each type, set the OMI_GET_METADATA (256) flag and one or more other GetMetadata flags. However, exercise caution when using these flags. When

a GetMetadata call is issued from within a federated query, the server retrieves the specified properties **for each object** returned by GetMetadataObjects in each repository, **including any cross−repository references** defined for each object, under the constraints described in Querying Cross−Repository References. When repositories that have a DependencyUsedBy association are queried, this can result in significant processing overhead.

Deleting a Dependency Association

A dependency association is deleted by using the UpdateMetadata method. For more information, see Example: Deleting a Dependency Association.

Deleting a Cross−Repository Reference

Cross−repository references are also removed by using the UpdateMetadata method, from the repository that has a DependencyUses association. For more information, see Example: Deleting a Cross−Repository Reference.

Examples

- Adding a Dependency Association to a Repository
- Querying Dependency Associations for a Specific Repository
- Querying Dependency Associations for All Repositories
- Creating Cross−Repository References
- Retrieving Cross−Repository References
- Issuing a Federated Query
- Deleting a Cross−Repository Reference
- Deleting a Dependency Association

Example: Adding a Dependency Association to a Repository

The following example creates three repositories −− Corporate, Department, and Group1 −− then after the repositories are created, uses the UpdateMetadata method to add relationships between them.

This method call creates the repositories:

```
<AddMetadata>
  <Metadata>
   <!-- Create Corporate repository -->
   <RepositoryBase Name="corporate" Desc="Corporate repository"
              Path="c:\testdat\xrpos\corporate"/>
   <!-- Create Department repository -->
   <RepositoryBase Name="dept" Desc="Department repository"
              Path="c:\testdat\xrpos\dept"/>
   <!-- Create Group 1 repository -->
   <RepositoryBase Name="group1" Desc="Group1 repository"
              Path="c:\testdat\xrpos\group1"/>
   </Metadata>
```

```
   <Reposid>A0000001.A0000001</Reposid>
   <Ns>REPOS</Ns>
   <!-- OMI_TRUSTED_CLIENT flag -->
   <Flags>268435456</Flags>
   <Options/>
</AddMetadata>
```

This is the output received from the server:

```
<RepositoryBase Name="corporate" Desc="Corporate repository"
 Path="c:\testdat\xrpos\corporate" Id="A0000001.A3WB4KWB" Access="0" RepositoryType=""/>
<RepositoryBase Name="dept" Desc="Department repository"
 Path="c:\testdat\xrpos\dept" Id="A0000001.A3POH2A5" Access="0" RepositoryType=""/>
<RepositoryBase Name="group1" Desc="Group1 repository"
 Path="c:\testdat\xrpos\group1" Id="A0000001.A39V7REZ" Access="0" RepositoryType=""/>
```

The following method call creates dependency associations between the repositories.

```
<UpdateMetadata>
  <Metadata>
   <!-- Update Corporate repository -->
   <RepositoryBase Objref="A0000001.A3WB4KWB">
    <DependencyUsedBy>
   <!-- Specify Department repository -->
      <RepositoryBase Objref="A0000001.A3POH2A5"/>
    </DependencyUsedBy>
    </RepositoryBase>
   <!-- Update Department repository -->
   <RepositoryBase Objref="A0000001.A3POH2A5">
    <DependencyUsedBy>
    <!-- Specify Group 1 repository -->
      <RepositoryBase Objref="A0000001.A39V7REZ"/>
    </DependencyUsedBy>
    </RepositoryBase>
   </Metadata>
   <Ns>REPOS</Ns>
   <!-- OMI_TRUSTED_CLIENT flag -->
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

Here is the output received from the server:

```
<RepositoryBase Id="A0000001.A3WB4KWB">
 <DependencyUsedBy>
     <RepositoryBase Id="A0000001.A3POH2A5"/>
 </DependencyUsedBy>
</RepositoryBase>
<RepositoryBase Id="A0000001.A3POH2A5">
  <DependencyUsedBy>
    <RepositoryBase Id="A0000001.A39V7REZ"/>
  </DependencyUsedBy>
</RepositoryBase>
```

A DependencyUsedBy association is created between the Corporate repository and the Department repository and between the Department repository and the Group1 repository. This means the Group1 repository uses the Department repository and the Department repository uses the Corporate repository. In the method calls, note:

- the AddMetadata request specifies the repository manager identifier (A0000001.A0000001) in the *Reposid* parameter
- both calls specify the REPOS namespace
- both calls specify the OMI_TRUSTED_CLIENT flag (268435456), which is required for write operations.

Example: Querying Dependency Associations for a Specific Repository

To retrieve dependency associations for a specific repository, use the GetMetadata method. The following GetMetadata method call queries the dependency associations defined for the Corporate repository.

```
<GetMetadata>
 <Metadata>
 <!-- Corporate repository -->
 <RepositoryBase Id="A0000001.A3WB4KWB" Name="corporate">
   <DependencyUses/>
   <DependencyUsedBy/>
 </RepositoryBase>
 </Metadata>
 <Ns>REPOS</Ns>
 <Flags>0</Flags>
 <Options/>
</GetMetadata>
```

Here is the output received from the server:

```
<RepositoryBase Id="A0000001.A3WB4KWB" Name="corporate">
  <DependencyUses/>
  <DependencyUsedBy>
     <RepositoryBase Id="A0000001.A3POH2A5" Name="dept" Desc="Department repository"/>
  </DependencyUsedBy>
</RepositoryBase>
```

The Corporate repository does not have any DependencyUses associations defined for it. It has one DependencyUsedBy association to the Department repository.

Example: Querying Dependency Associations for All Repositories

To retrieve the dependency associations for all repositories registered in a repository manager, use the GetMetadataObjects method as shown below:

```
<GetMetadataObjects>
 <!--Specify repository manager identifier -->
 <Reposid>A0000001.A0000001</Reposid>
 <Type>RepositoryBase</Type>
 <Objects/>
 <Ns>REPOS</Ns>
 <!-- Specify OMI_GET_METADATA (256) + OMI_TEMPLATE (4) flags -->
 <Flags>260</Flags>
 <Options>
```

```
  <Templates>
    <RepositoryBase Id="" Name="" Desc="">
     <DependencyUses/>
    </RepositoryBase>
  </Templates>
</Options>
</GetMetadataObjects>
```

In the call, note:

- the *Reposid* parameter specifies the repository manager identifier
- the *Type* parameter specifies the RepositoryBase metadata type
- the *Flags* parameter specifies OMI_GET_METADATA (256) and OMI_TEMPLATE (4)
- the *Options* parameter includes a <Templates> element that requests the Id=, Name=, and Desc= attributes and all DependencyUses associations found for each object.

The output from the server is as follows:

```
<RepositoryBase Id="A0000001.A3WB4KWB" Name="corporate" Desc="Corporate repository">
 <DependencyUses/>
</RepositoryBase>
<RepositoryBase Id="A0000001.A3POH2A5" Name="dept" Desc="Department repository">
 <DependencyUses>
  <RepositoryBase Id="A0000001.A3WB4KWB" Name="corporate" Desc="Corporate repository">
 <DependencyUses/>
</RepositoryBase>
</DependencyUses>
</RepositoryBase>
<RepositoryBase Id="A0000001.A39V7REZ" Name="group1" Desc="Group1 repository">
<DependencyUses>
<RepositoryBase Id="A0000001.A3POH2A5" Name="dept" Desc="Department repository">
<DependencyUses>
<RepositoryBase Id="A0000001.A3WB4KWB" Name="corporate" Desc="Corporate repository">
<DependencyUses/>
</RepositoryBase>
</DependencyUses>
</RepositoryBase>
</DependencyUses>
</RepositoryBase>
```

Example: Creating Cross–Repository References

The following XML method calls create a SASLibrary object in each of the three repositories and create associations between them.

This method call creates a SASLibrary object in the Group 1 repository named Group1Lib:

```
<AddMetadata>
 <Metadata>
 <!-- Metadata specifies library object named Group1Lib -->
    <SASLibrary Name="Group1Lib" Desc="Group 1 Library"/>
 </Metadata>
 <!-- Reposid specifies Group 1 repository -->
 <Reposid>A0000001.A39V7REZ</Reposid>
 <Ns>SAS</Ns>
 <!-- OMI_TRUSTED_CLIENT flag -->
```

```
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

This is the information received from the server:

```
<SASLibrary Name="Group1Lib" Desc="Group 1 Library" Id="A39V7REZ.A1000001"/>
```

This method call creates a SASLibrary object in the Corporate repository named CorpLib:

```
<AddMetadata>
 <Metadata>
 <!-- Metadata specifies library object named CorpLib -->
   <SASLibrary Name="CorpLib" Desc="Corporate Library"/>
 </Metadata>
 <!-- Reposid specifies the Corporate repository -->
 <Reposid>A0000001.A3WB4KWB</Reposid>
 <Ns>SAS</Ns>
 <!-- OMI_TRUSTED_CLIENT flag -->
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

This is the output received from the server:

```
<SASLibrary Name="CorpLib" Desc="Corporate Library" Id="A3WB4KWB.A1000001"/>
```

This method call creates a SASLibrary object in the Department repository named DeptLib and creates cross−repository references between DeptLib, CorpLib, and Group1Lib. The cross−repository references are created at the same time as DeptLib because library will have an association to the other two libraries and the Department repository has a dependency association with both of the other repositories.

```
<AddMetadata>
 <Metadata>
  <!-- Metadata specifies library object named DeptLib -->
  <SASLibrary Name="DeptLib" Desc="Department library">
    <!-- The next two statements create an association to DeptLib -->
    <UsingPackages>
       <!-- ObjRef= specifies the Corporate library -->
       <SASLibrary ObjRef="A3WB4KWB.A1000001" Name="CorpLib"/>
    </UsingPackages>
    <UsedByPackages>
      <!-- ObjRef= specifies the Group1Lib -->
      <SASLibrary ObjRef="A39V7REZ.A1000001" Name="Group1Lib"/>
    </UsedByPackages>
  </SASLibrary>
 </Metadata>
 <!-- Reposid specifies the Department repository -->
 <Reposid>A0000001.A3POH2A5</Reposid>
 <Ns>SAS</Ns>
 <!-- OMI_TRUSTED_CLIENT flag -->
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

Here is the output received from the server:

```
<SASLibrary Name="DeptLib" Desc="Department library" Id="A3POH2A5.A1000001">
```

```
  <UsingPackages>
    <SASLibrary ObjRef="A3WB4KWB.A1000001" Name="CorpLib"/>
  </UsingPackages>
  <UsedByPackages>
    <SASLibrary ObjRef="A39V7REZ.A1000001" Name="Group1Lib"/>
  </UsedByPackages>
</SASLibrary>
```

A UsingPackages association is created between DeptLib and CorpLib. A UsedByPackages association is created between DeptLib and Group1Lib.

Example: Retrieving Cross−Repository References

Information about cross−repository references is retrieved by using the GetMetadata method. This example queries the Department repository without flags.

```
<GetMetadata>
 <Metadata>
  <!-- Metadata specifies DeptLib and requests information about the associations  -->
  <SASLibrary Id="A3POH2A5.A1000001">
   <UsingPackages/>
   <UsedByPackages/>
  </SASLibrary>
 </Metadata>
 <Ns>SAS</Ns>
 <!-- No flags are included-->
 <Flags>0</Flags>
 <Options/>
</GetMetadata>
```

Here is the output received from the server:

```
<SASLibrary Id="A36LIBAV.A1000001">
  <UsingPackages>
    <SASLibrary Id="A3ZX5267.A1000001" Name="CorpLib"
     Desc="Corporate Library"/>
  </UsingPackages>
  <UsedByPackages/>
</SASLibrary>
```

Note that the server returns information about the cross−repository reference between CorpLib and DeptLib but not about DeptLib and Group1Lib. This is because information about cross−repository references is stored in the repository that has a DependencyUses association, and the Department repository has a DependencyUsedBy association to the Group1 repository.

Example: Issuing a Federated Query

The following is an example of a federated query. Issued from the Corporate repository, it retrieves all SASLibrary objects in repositories that have a DependencyUsedBy association in the repository chain.

```
<GetMetadataObjects>
<!-- Corporate repository -->
<Reposid>A0000001.A3WB4KWB</Reposid>
```

```
<Type>SASLibrary</Type>
<Objects/>
<Ns>SAS</Ns>
<!-- OMI_DEPENDENCY_USED_BY flag  -->
<Flags>16384</Flags>
<Options/>
</GetMetadataObjects>
```

Here is the output received from the server:

```
<Objects>
  <SASLibrary Id="A3WB4KWB.A1000001" Name="CorpLib"/>
  <SASLibrary Id="A3POH2A5.A1000001" Name="DeptLib"/>
  <SASLibrary Id="A39V7REZ.A1000001" Name="Group1Lib"/>
</Objects>
```

No GetMetadata flags are set; therefore, the server retrieves only the Id and Name attributes of each object.

Example: Deleting a Cross−Repository Reference

The following method call removes the UsingPackages and UsedByPackages associations defined for DeptLib in the Department repository. Passing the Function="REPLACE" directive on each association name without specifying any association subelements deletes any associations of those names defined for the specified SASLibrary object.

```
<UpdateMetadata>
   <Metadata>
    <!-- Metadata specifies DeptLib and associations to be deleted -->
    <SASLibrary Id="A3POH2A5.A1000001" Name="DeptLib">
      <UsingPackages Function="REPLACE"/>
      <UsedByPackages Function="REPLACE"/>
   </SASLibrary>
   </Metadata>
   <Ns>SAS</Ns>
   <!-- OMI_TRUSTED_CLIENT flag -->
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

Here is the output received from the server:

```
<SASLibrary Id="A3POH2A5.A1000001" Name="DeptLib">
  <UsingPackages Function="REPLACE"/>
  <UsedByPackages Function="REPLACE"/>
</SASLibrary>
```

The cross−reference between CorpLib and DeptLib is removed, but the cross−reference between DeptLib and Group1Lib remains. This is because the REPLACE directive is executed on the current repository and the cross−repository reference between DeptLib and Group1Lib is stored in the Group 1 repository (the repository that has the DependencyUses association in that particular association). To remove the cross−reference between DeptLib and Group1Lib, you must update the Group1 repository.

Example: Deleting a Dependency Association

The following method call removes the dependency association between the Department repository and the Group 1 repository. Passing the Function="REPLACE" directive on the association name without specifying any association subelements deletes any associations of that name defined for the specified RepositoryBase object.

```
<UpdateMetadata>
   <Metadata>
    <!--Metadata specifies Department repository -->
    <RepositoryBase Id="A0000001.A3POH2A5">
      <DependencyUsedBy Function="REPLACE"/>
   </RepositoryBase>
   </Metadata>
   <Reposid>A0000001.A0000001</Reposid>
   <Ns>REPOS</Ns>
   <Flags>268435456</Flags>
   <Options/>
</UpdateMetadata>
```

Here is the output received from the server:

```
<RepositoryBase Id="A0000001.A3POH2A5">
  <DependencyUsedBy Function="REPLACE"/>
</RepositoryBase>
```

The request deletes all DependencyUsedBy associations that might have been defined for the Department repository.

Previous Page | Next Page | Top of Page

Creating a Repository on an External DBMS

Overview

The SAS Metadata Server enables you to create and use SAS metadata repositories on the Oracle and IBM DB2 external database management systems.

SAS metadata repositories are SAS libraries. Like base engine libraries, DBMS libraries and tables are accessed by passing engine−connection and LIBNAME options to the appropriate LIBNAME engine.

You create SAS metadata repositories on an external DBMS by specifying an alternate engine and passing SAS/ACCESS LIBNAME options at repository registration. The SAS Metadata Server repository manager is also a SAS metadata repository. You can define the server's repository manager in an external DBMS by specifying an alternate engine and passing SAS/ACCESS LIBNAME options in the omaconfig.xml file. For more information about defining the repository manager on an external DBMS, see "Creating an omaconfig.xml File" in the **SAS 9.1 Metadata Server: Setup Guide.** This document describes how to register a SAS metadata repository on an external DBMS using the SAS Open Metadata Interface.

Software Requirements

A host that will create or access a repository that exists on an external DBMS must have SAS 9.1 software, the appropriate DBMS client software, and the appropriate SAS/ACCESS interface software. For example, in addition to SAS 9.1,

- an Oracle client needs the latest version of Oracle Client software and SAS/ACCESS Interface to Oracle software
- a DB2 client needs the latest version of DB2 Client software and SAS/ACCESS Interface to DB2 under UNIX and PC hosts software.

The SAS/ACCESS software and DBMS software must be installed on the same machine as SAS 9.1 software.

Host Requirements

Both SAS/ACCESS to DB2 software and SAS/ACCESS to Oracle software use shared libraries. When a metadata server is running in a UNIX host environment, you must specify the location of the shared libraries in a system environment variable, and sometimes indicate the software version of the DBMS that you have installed at your site. For DB2, you must also set the INSTHOME environment variable to your DB2 home

directory. For Oracle, you must set the ORACLE_HOME environment variable to your Oracle home directory.

The environment variables are platform−specific. You can set them in the script that starts the metadata server. See "Using an External DBMS" in the *SAS 9.1 Metadata Server: Setup Guide* for examples of the environment variables that are needed for a metadata server that runs on AIX, HP−UX, Linux, or Solaris.

DBMS Table Requirements

The libraries in which repositories will be created and their subordinate tables can be created by the metadata server, via SAS/ACCESS software, or you can create the tables in the database before registering a repository.

Repository Registration Requirements

The metadata server uses the following parameters to register (create) a repository or repository manager on an external DBMS:

ENGINE=
: specifies the engine to use to access the repository. This can be the default base engine or a SAS/ACCESS engine, such as Oracle or DB2.

NAME=
: is a name for the repository. The repository name is required.

DESC=
: is an optional description of the repository.

PATH=
: is the location of the repository directory. When creating a repository on an external DBMS, this parameter is left blank. The location of the DBMS table is identified using database−specific engine−connection and SAS/ACCESS LIBNAME options in the *Options* parameter.

OPTIONS=
: specifies options for accessing the external DBMS. For more information, see Oracle Connection Options and DB2 Connection Options.

Oracle Connection Options

SAS/ACCESS Interface to Oracle requires the following engine−connection and LIBNAME options to connect to and register a repository on Oracle.

PATH=
: specifies an alias representing the Oracle driver, node, and database. If you do not know how to create an alias representing the necessary Oracle driver, node, and database for your repository, ask your database administrator to provide you with one.

USER=
: specifies an Oracle user name. If the user name contains blanks or national characters, enclose the name in quotation marks. If you omit an Oracle user name and password, the default Oracle user ID OPS$sysid is used, if it is enabled. USER= must be used with PASSWORD=.

PASSWORD=
: specifies an Oracle password that is associated with the Oracle user name. If you omit PASSWORD=, the password for the default Oracle user ID OPS$sysid is used, if it is enabled.

CONNECTION=shared

specifies that all tables that are opened for reading by this LIBNAME or libref share this connection.

DBCHAR_CONSTANT_IS_SPOOFED=YES
this option is required and is only valid when used for access to SAS Metadata Repositories by the Oracle engine.

INSERTBUFF=1
specifies the number of rows in a single Oracle insert operation. A value of 1 of is required.

PRESERVE_NAMES=YES
preserves spaces, special characters, and mixed case in DBMS column and table names.

READBUFF=1
specifies the number of rows in a single Oracle fetch. A value of 1 of is required.

REREAD_EXPOSURE=YES
specifies that the SAS/ACCESS engine will behave like a random access engine for the scope of the connection.

SCHEMA=schema_name
enables you to register multiple repositories using the same Oracle user account. When SCHEMA= is omitted, the software uses the default schema, which is the requesting user ID, to create the repository. A *schema_name* such as "SASrmgr" is recommended when creating the repository manager. The *schema_name* must be a valid SAS name that is unique for the USER=, PASSWORD=, and PATH.

SPOOL=NO
prevents spooling in the Oracle engine. Spooling data by the Oracle engine is unnecessary because the SAS Metadata Server uses the SAS In Memory Database for the repository tables. Although the SAS Metadata Server should not cause spooling to occur in the Oracle engine, explicitly setting the option to NO should guarantee that no spooling is done.

UTILCONN_TRANSIENT=NO
specifies that a utility connection is maintained for the lifetime of the libref.

For a more detailed description of Oracle engine−connection and LIBNAME options, see "SAS/ACCESS LIBNAME Statement: Oracle Specifics" in the **SAS 9.1 Language Reference: Dictionary**.

DB2 Connection Options

The metadata server requires that you specify the following engine−connection and LIBNAME options to connect to and create a repository on DB2.

DATABASE= (DB=)database_name
specifies the DB2 data source or database to which you want to connect.

USER=userid
enables you to connect to a DB2 database with a user ID that is different from the user ID requesting the connection. The USER= and PASSWORD= connections are optional in DB2. If you specify USER=, you must also specify PASSWORD=. If USER= is omitted, your default user ID for your operating environment is used.

PASSWORD=passwd
specifies the DB2 password that is associated with your DBMS user ID, if the USER= option is specified.

CONNECTION=SHARED
specifies that all tables that are opened for reading by this LIBNAME or libref share this connection.

DBCHAR_CONSTANT_IS_SPOOFED=YES
this option is required and is only valid when used for access to SAS Metadata Repositories by the DB2 engine.

PRESERVE_NAMES=YES
: preserves spaces, special characters, and mixed case in DBMS column and table names.

REREAD_EXPOSURE=YES
: specifies that the SAS/ACCESS engine will behave like a random access engine for the scope of the connection.

SCHEMA=schema_name
: enables you to register multiple repositories using the same DB2 user account. When SCHEMA= is omitted, the software uses the default schema, which is the requesting user ID, to create the repository. A *schema_name* such as "SASrmgr" is recommended when creating the repository manager. The *schema_name* must be a valid SAS name that is unique for the USER=, PASSWORD=, and DB=.

SPOOL=NO
: prevents spooling in the DB2 engine. Spooling data by the DB2 engine is unnecessary because the SAS Metadata Server uses the SAS In Memory Database for the repository tables. Although the SAS Metadata Server should not cause spooling to occur in the DB2 engine, explicitly setting the option to NO should guarantee that no spooling is done.

INSERTBUFF=1
: specifies the number of rows in a single DB2 insert operation. A value of 1 of is required.

READBUFF=0
: specifies the number of rows in each DB2 fetch. A value of 0 is required.

UTILCONN_TRANSIENT=NO
: specifies that a utility connection is maintained for the lifetime of the libref.

For a detailed description of these options, see "SAS/ACCESS LIBNAME Statement: DB2 Specifics" in the **SAS 9.1 Language Reference: Dictionary**.

Using a Repository on an External DBMS

Client requests to a repository on an external DBMS are processed the same way as they are when the repository is a SAS library.

Example of Registering a Repository on an External DBMS

You can register a repository on an external DBMS by using the SAS Open Metadata Interface or by using SAS Management Console. The following are examples of registering a repository using the SAS Open Metadata Interface. In the requests, note that a repository is created by instantiating an object of metadata type RepositoryBase. For information about the metadata types used to represent repositories, see REPOS Namespace Types in the **SAS 9.1 Open Metadata Interface: Reference**. For more information about the AddMetadata method, see IOMI Class in the **SAS 9.1 Open Metadata Interface: Reference**.

Register an Oracle Repository

The following method call creates a repository named Scratch1 in an Oracle database identified by the alias "alien". The request is formatted for the *inMetadata* parameter of the DoRequest method. For information about the DoRequest method, see Call Interfaces in the **SAS 9.1 Open Metadata Interface: Reference**.

```
<AddMetadata>
 <Metadata>
  <RepositoryBase
      Engine="oracle"
```

```
      Name="scratch1"
      Desc="test repository"
      PATH=""
      Options="path=alien user=ot1 password=ot11
      reread_exposure=yes preserve_names=yes dbchar_constant_is_spoofed=yes
      connection=shared readbuff=1 insertbuff=1 spool=no utilconn_transient=no">
  </RepositoryBase>
 </Metadata>
 <Reposid>A0000001.A0000001</Reposid>
 <Ns>REPOS</Ns>
 <!-- OMI_TRUSTED_CLIENT flag -- >
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

Register a DB2 Repository

The following method call creates a repository named Scratch1 in a DB2 database named "sample". The request is formatted for the *inMetadata* parameter of the DoRequest method.

```
<AddMetadata>
 <Metadata>
   <RepositoryBase
       Engine="db2"
       Name="scratch1"
       Desc="test repository"
       Path=""
       Options="db=sample user=ot1 password=Lupin1 reread_exposure=yes
       preserve_names=yes dbchar_constant_is_spoofed=yes connection=shared
       spool=no readbuff=0 insertbuff=1 utilconn_transient=no">
   </RepositoryBase>
 </Metadata>
 <Reposid>A0000001.A0000001</Reposid>
 <Ns>REPOS</Ns>
 <!-- OMI_TRUSTED_CLIENT flag -- >
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

The following example creates two repositories named Scratch2 and Scratch3 in a DB2 database called "sample". Note the use of the SCHEMA= SAS/ACCESS LIBNAME option. When SCHEMA= is not used, the software uses the default schema, which is the user ID, to create the repository. The request is formatted for the *inMetadata* parameter of the DoRequest method.

```
<AddMetadata>
 <Metadata>
   <RepositoryBase
       Engine="db2"
       Name="scratch2"
       Desc="test repository 2"
       Path=""
       Options="db=sample schema=ot18 user=ot1 password=Lupin1
       reread_exposure=yes preserve_names=yes dbchar_constant_is_spoofed=yes
       connection=shared spool=no readbuff=0 insertbuff=1 utilconn_transient=no">
   </RepositoryBase>
   <RepositoryBase
       Engine="db2"
       Name="scratch3"
```

```
      Desc="test repository 3"
      Path=""
      Options="db=sample schema=ot19 user=ot1 password=Lupin1
      reread_exposure=yes preserve_names=yes dbchar_constant_is_spoofed=yes
      connection=shared spool=no readbuff=0 insertbuff=1 utilconn_transient=no">
   </RepositoryBase>
 </Metadata>
 <Reposid>A0000001.A0000001</Reposid>
 <Ns>REPOS</Ns>
 <!-- OMI_TRUSTED_CLIENT flag -- >
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

Previous Page | Next Page | Top of Page

Invoking a Repository Audit Trail

Introduction

The SAS 9.1 Metadata Server supports saving information about changes to metadata in a set of audit files. The information in the audit files can be used to restore the metadata to a previous state. The information in the audit files is referred to as an "audit trail".

Auditing is performed on individual repositories and is enabled by specifying values for four new attributes for RepositoryBase metadata objects:

AuditType
: turns auditing on and off and specifies the type of records that will be logged.

AuditPath
: specifies an existing directory where the audit trail is to be created.

AuditEngine
: specifies the engine with which to create the audit files.

AuditOptions
: specifies engine connection options.

The audit trail stores information about added, deleted, and updated records and supports audit files created with the base engine. A subsequent release will support additional engines.

Starting the Audit Trail

Auditing is enabled by specifying values for a repository's AuditType and AuditPath attributes. Values for the AuditType and AuditPath attributes can be assigned in the AddMetadata method that creates a repository. Values can be added to existing repositories by using the UpdateMetadata method. The UpdateMetadata method is also used to change the audit trail location and to turn off auditing. For examples of how to issue the method calls, see Using AddMetadata to Enable Auditing, Using UpdateMetadata to Enable Auditing, Changing the Audit Trail Location, and Turning Off Auditing.

AuditType Attribute

The AuditType attribute is required in any method call related to auditing. In calls that turn auditing on and off, it specifies the action to perform and the type of records to audit. When a call is made to update an audit attribute, such as the AuditPath, the AuditType attribute enables auditing at the new location. Only one audit trail location is supported for a repository. Changing the AuditPath turns off auditing in the previous location and turns auditing on in the new location.

The AuditType attribute supports six values:

1
: turns on auditing and writes an audit record for all added metadata records.

2
: turns on auditing and writes an audit record for all deleted metadata records.

4
: turns on auditing and writes an audit record for all updated metadata records.

7
: turns on auditing and writes an audit record for all record types (Add, Delete, and Update).

0
: (zero) turns off auditing and indicates that auditing has been disabled.

−1
: a system−supplied value (representing a missing value) that indicates auditing has not been initiated for the repository.

When AuditType="2" is set, the audit trail writes an audit record for each metadata object that is directly or indirectly deleted from a repository. A metadata object is directly deleted when it is specified in a DeleteMetadata method call and can be indirectly deleted by the DeleteMetadata or UpdateMetadata method, if it is dependent on another object that is deleted. The DeleteMetadata method also supports flags that enable you to delete all objects in a repository, to unregister a repository, and to destroy a repository. Audit records are not written for DeleteMetadata calls that set these flags.

When AuditType="4" is set, the audit trail writes two audit records for each update transactions: a before−update image and an after−update image. For more information, see Audit File Format.

AuditPath Attribute

The AuditPath attribute is required to turn on auditing; omit it from method calls that disable auditing. AuditPath identifies the location to write the audit trail. The specified directory must exist, it must be empty, and it must be a different location than the repository directory. It is recommended that you use the same name for the AuditPath directory as the repository directory and create it in an Audit subdirectory of the server directory. For example:

```
repository path: omasvr/repository1
audit path: omasvr/audit/repository1
```

Storing the audit trail in a subdirectory of the server directory enables audit files to inherit the file and directory access permissions set for the server directory.

Audit File Format

The audit trail consists of a copy of the repository's metadata for the type of records being audited. For example, the audit trail stores a copy of all added, deleted, and updated metadata records.

In addition, each record has the following variables assigned to it:

*_ATID_*
: contains a 36−character Global Unique Identifier (GUID). An example of a GUID is 46629670−59C4−4AE8−B089−2E861ED884C5.

*_ATIME_*
: contains a datetime value indicating when the audit record was created.

*_ATYPE_*
: contains a letter indicating the audit record type:

- "N" identifies an added (new) metadata record
- "D" identifies a deleted metadata record
- "B" identifies an before−update metadata record
- "A" identifies an after−update metadata record.

DELETED
: contains a datetime value indicating when the record was deleted. Records that were logically deleted in a repository by previous releases of SAS software and migrated to the audit file will have the DELETED variable set to the datetime when the record was logically deleted. Otherwise, the DELETED value will be missing. For more information about migrating records from previous releases, see Migrating Deleted Records.

The _ATID_ and _ATIME_ values in the audit file records are the same for all records created by a single method call. For example, all before− and after−record images for all metadata records updated by a single UpdateMetadata method call will have the same _ATID_ and _ATIME_ values. Also, the audit records for all metadata records created by a single AddMetadata call will have the same _ATID_ and _ATIME_ values, and the audit records for all metadata records deleted by a single DeleteMetadata method call will have the same _ATID_ and _ATIME_ values.

Audit records are stored according to metadata type in separate files in the audit directory. For example, all deleted PhysicalTable records are stored in a PhysicalTable container, deleted Column records are stored in a Column container, and so on. You can list the audit files created for a repository by using PROC DATASETS. PROC DATASETS can also be used to list the variables in an audit file. To read the records in an audit file, you use PROC PRINT. For more information, see Example: Listing Audit Files, Example: Listing Audit File Variable Names, and Example: Viewing Audit File Records.

Physical versus Logical Deletion of Metadata Records

The default behavior of the SAS 9.0 Metadata Server was to logically, rather than physically, delete metadata records from a repository. This allowed a record of deleted metadata objects to be kept until an auditing capability was added to the server. In the SAS 9.1 Metadata Server, the default was changed to physically delete any new metadata records that are deleted; however, any logical records created by the previous server remain in a repository.

You can migrate logical records that exist in a repository to the audit trail by issuing a DeleteMetadata method on the desired repository(ies) that sets the OMI_PURGE (1048576) flag. The OMI_PURGE flag removes previously deleted metadata records from a repository without disturbing the current metadata objects. For the migration to be successful, repository auditing must have been enabled with AuditType="2". For an example of the method call, see Migrating Deleted Records.

Logical versus physical deletion of deleted metadata records is controlled by the OMA PHYSICAL_DELETE server configuration option. For more information about the tradeoffs associated with logically versus physically deleting metadata files, see **OMA PHYSICAL_DELETE Option** in the **SAS 9.1 Metadata Server: Best Practices Guide**.

Usage Considerations

The auditing functionality is not intended to provide system backups. A system backup is a copy of a repository usually made using operating system tools on a daily basis. The system backup provides a complete repository image for protection from a catastrophic disk failure. The audit restore provides recovery for a logically damaged repository. This could happen if an update destroyed or corrupted the metadata.

For information about backing up a repository, see **Backing Up Metadata** in Administering the Server in the **SAS 9.1 Metadata Server: Setup Guide**.

Recovering Audit Records

Metadata that has gaps in data may corrupt a repository if restored. Incomplete data can result from:

- turning repository auditing on and off
- not auditing all the metadata record types in a repository
- not auditing all of the repositories on a server, particularly if cross−references are supported between the repositories.

As this release does not support a rollback mechanism, it cannot be used to restore a damaged repository to its original state. However, you can use information in the audit trail to restore specific records. When restoring a deleted record, note that a metadata record can have multiple dependent records that might have been indirectly deleted as well. When deleting records, you can set the OMI_RETURN_LIST flag to keep track of the dependent records. The OMI_RETURN_LIST flag returns a list of deleted object IDs, as well as any cascading object IDs that were deleted.

The general steps for restoring a metadata record from the audit trail are described in Restoring a Metadata Record.

Deleting the Audit Trail

The audit trail is deleted by deleting the audit files in the audit directory. The audit files can be deleted using PROC DATASETS. The audit trail should be deleted when its companion repository has been deleted.

Here is a sample program that deletes an audit trail:

```
/* delete all audit files */
libname audit 'repos\audit\repos1';
proc datasets library=audit KILL;
```

```
run;
quit;
```

Examples

- Using AddMetadata to Enable Auditing
- Using UpdateMetadata to Enable Auditing
- Changing the Audit Trail Location
- Migrating Deleted Records
- Turning Off Auditing
- Restoring a Metadata Record

Using AddMetadata to Enable Auditing

The following is an example of an AddMetadata method call that creates a repository and turns on auditing. The call is formatted for the *inMetadata* parameter of the DoRequest method.

```
<AddMetadata>
  <Metadata>
    <RepositoryBase
      Name="repository1"
      Desc="my repository"
      Path="omasvr/repository1"
      AuditPath="omasvr/audit/repository1"
      AuditType="2"/>
  </Metadata>
  <Reposid>A0000001.A0000001</Reposid>
  <Ns>REPOS</Ns>
  <!--OMI_TRUSTED_CLIENT flag -->
  <Flags>268435456</Flags>
  <Options/>
</AddMetadata>
```

In the metadata property string, note:

- the Path and AuditPath attributes specify directories of the same name, but different locations.
- the AuditType parameter specifies "2", which turns on auditing of deleted records. AuditType is an optional parameter. Omitting AuditType causes the repository to be created without an audit trail.

In the remainder of the call, note:

- the *Ns* parameter specifies the REPOS namespace, which is required to write to a repository.
- the *Reposid* parameter specifies the repository manager identifier. The repository manager manages repositories.
- the *Flags* parameter specifies the OMI_TRUSTED_CLIENT flag (268435456), which is required to write a metadata object.

Using UpdateMetadata to Enable Auditing

The following is an example of an UpdateMetadata method call that turns on auditing. The call is formatted for the *inMetadata* parameter of the DoRequest method.

```
<UpdateMetadata>
  <Metadata>
    <RepositoryBase
      Id="A0000001.A1234566"
      Name="Repository2"
      AuditPath="audit/repository2"
      AuditType="7"/>
  </Metadata>
  <Ns>REPOS</Ns>
  <!--OMI_TRUSTED_CLIENT flag -->
  <Flags>268435456</Flags>
  <Options/>
</UpdateMetadata>
```

In the metadata property string, note:

- the Id and Name attributes identify an existing repository for which to enable auditing.
- there is no need to specify a Path attribute for the repository.
- the AuditType attribute specifies the value "7", which creates audit records for added and updated metadata records in addition to deleted metadata records.
- the AuditPath attribute specifies the location to write the audit trail.

The other parameters are the same as in the AddMetadata request.

Changing the Audit Trail Location

The following is an example of an UpdateMetadata request that changes the audit trail location. The audit trail location is specified in the AuditPath attribute of a RepositoryBase object.

```
<UpdateMetadata>
  <Metadata>
    <RepositoryBase
      Id="A0000001.A1234566"
      Name="Repository2"
      AuditType="2"
      AuditPath="omasvr/audit/repository2"/>
  </Metadata>
  <Ns>REPOS</Ns>
  <!--OMI_TRUSTED_CLIENT flag -->
  <Flags>268435456</Flags>
  <Options/>
</UpdateMetadata>
```

Any location specified in the AuditPath attribute that does not exactly match the current registration is considered a new location. For example, a relative pathname and an absolute pathname are considered different locations. As a result of the method call, audit records for Repository2 are written to the omasvr/audit/repository2 directory instead of the audit/repository2 directory.

Migrating Deleted Records

The following is an example of a DeleteMetadata method call that migrates logically deleted records from the specified repository to the audit trail. For the migration to be successful, repository auditing must have been enabled with AuditType="2" (delete). The migration will not work if auditing was enabled with

AuditType="7" (all).

```
<DeleteMetadata>
  <Metadata>
    <RepositoryBase Id="A0000001.A1234566"/>
  </Metadata>
  <NS>REPOS</NS>
  <!--OMI_TRUSTED_CLIENT (268435456) and OMI_PURGE (1048576) flags -->
  <Flags>269484032</Flags>
  <Options/>
</DeleteMetadata>
```

The OMI_PURGE flag removes previously deleted metadata records from the repository without disturbing the current metadata objects. Meanwhile, an audit record is created for every metadata object deleted by the DeleteMetadata method.

Turning Off Auditing

The following is an example of an UpdateMetadata method call that turns off auditing. The call is formatted for the *inMetadata* parameter of the DoRequest method.

```
<UpdateMetadata>
  <Metadata>
    <RepositoryBase
      Id="A0000001.A1234566"
      AuditType="0"/>
  </Metadata>
  <Ns>REPOS</Ns>
  <!--OMI_TRUSTED_CLIENT flag -->
  <Flags>268435456</Flags>
  <Options/>
</UpdateMetadata>
```

The AuditType value of "0" turns off auditing and signifies that auditing was disabled for Repository2. Here is sample output from a GetMetadata method call on a repository that had auditing disabled:

```
Information received from server :
<GetMetadata>
  <Metadata>
    <RepositoryBase Id="A0000001.A1234566" Access="0" AuditEngine=""
    AuditOptions="" AuditPath="omasvr/audit/repository2" AuditType="0"
    Desc="" Engine="" MetadataCreated="24Oct2002:12:55:15"
    MetadataUpdated="24Oct2002:12:55:15" Name="Repository2" Options=""
    Path="omasvr/repository2" RepositoryType=""/>
  </Metadata>
 <NS>REPOS</NS>
 <Flags>
 </Flags>
 <Options/>
</GetMetadata>
```

The AuditType attribute is "0" but the AuditPath attribute continues to show the location of the audit trail.

Restoring a Metadata Record

The SAS 9.1 metadata audit trail can be used to restore specific metadata records to a repository. The process for restoring metadata in this release is accomplished through the use of SAS programs and not the metadata server.

Caution: The repository must be paused to an offline state before data is restored. A repository can be paused by issuing a PAUSE action in PROC METAOPERATE. A repository that is paused must eventually be resumed. This is done by issuing a RESUME action in PROC METAOPERATE. See METAOPERATE Procedure in the **SAS 9.1 Open Metadata Interface: Reference** for details.

This section describes through examples how to

- list audit files
- list audit file variable names
- view audit file records
- restore an audit record.

Example: Listing Audit Files

The audit trail stores records according to metadata type. A separate audit file is maintained for each metadata type that an audit record has been written. You can list the audit file names comprising an audit trail using PROC DATASETS. First, assign an AUDIT libref to the location specified in the AuditPath attribute; then, specify the libref in the Library= statement, as follows:

```
libname audit 'repos\audit\repos1';
proc datasets library=audit;
run;
quit;
```

Here is an example of the information that is written to the SAS Log:

```
                 Directory

   Libref         AUDIT
   Engine         V9
   Physical Name  U:\tkmeta\repos\audit\repos1
   File Name      U:\tkmeta\repos\audit\repos1

             Member    File
#  Name      Type      Size  Last Modified
1  CNTAINER  DATA      5120   28OCT2002:12:06:22
2  EMAIL     DATA     17408   28OCT2002:12:09:21
3  MRRGSTRY  DATA      9216   28OCT2002:12:06:22
4  PERSON    DATA     17408   28OCT2002:12:09:21
5  PHONE     DATA     17408   28OCT2002:12:09:21
```

The CNTAINER and MRRGSTRY files are audit control files. They do not contain audit records. In this example, there are three audit files: EMAIL, PERSON, and PHONE.

Example: Listing Audit File Variable Names

The variable names in an audit file can also be listed using PROC DATASETS. To list the variable names, specify an audit file name in the CONTENTS statement. In this example, the CONTENTS statement specifies the audit file PERSON.

```
libname audit 'repos\audit\repos1';
proc datasets library=audit;
   contents data=audit.person;
run;
quit;
```

Here is the information written to the SAS Log:

```
DATASETS Procedure   Monday, October 28, 2002

 Data Set Name       AUDIT.PERSON             Observations         2
 Member Type         DATA                     Variables            17
 Engine              V9                       Indexes              0
 Created             Monday, October 28, 2002 Observation Length   1504
 Last Modified       Monday, October 28, 2002 Deleted Observations 0
 Protection                                   Compressed           NO
 Data Set Type                                Sorted               NO
 Label
 Data Representation WINDOWS
 Encoding            any  Any encoding

Engine/Host Dependent Information

 Data Set Page Size          16384
 Number of Data Set Pages    1
 First Data Page             1
 Max Obs per Page            10
 Obs in First Data Page      2
 Number of Data Set Repairs  0
 File Name                   U:\tkmeta\repos\audit\repos1\person.sas7bdat
 Release Created             9.0100A0
 Host Created                XP_PRO

Alphabetic List of Variables and Attributes

  #    Variable    Type    Len
 13    CHNGSTT     Char    128
  1    CLASNAME    Char     70
  3    DEFCREDT    Num       8
  4    DEFMODDT    Num       8
  5    DELETED     Num       8
  6    EXPLABEL    Char     80
 11    ICON        Char     80
  7    ID          Char     34
  8    LDESC       Char    400
 12    LOCKEDBY    Char     64
  9    MRACCESS    Num       8
  2    OBJNAME     Char    120
 14    TITLE       Char    400
 10    VERSION     Num       8
 17    _ATID_      Char     72
 16    _ATIME_     Num       8
 15    _ATYPE_     Char      2
```

In this example there are 14 repository variables and three audit variables: _ATYPE_, _ATIME_ and _ATID_.

Example: Viewing Audit File Records

To view audit file records, use PROC PRINT. The following example prints the contents of the PERSON audit file:

```
libname audit 'repos\audit\repos1';
proc print data=audit.person;
   var clasname id objname ldesc deleted;
run;
```

The VAR statement specifies the variables to print. Here is the information written to the SAS Log:

```
12:12 Monday, October 28, 2002
Obs CLASNAME
 1  O M S p e r s o n
 2  O M S p e r s o n

Obs ID
 1  A 5 J Z B Y C 7 . A 1 0 0 0 0 0 1
 2  A 5 J Z B Y C 7 . A 1 0 0 0 0 0 2

Obs OBJNAME
 1  W i l l
 2  C a r l a

Obs LDESC
 1  S t u d e n t
 2  S t u d e n t

Obs    DELETED
 1  1351447597.5
 2  1351447598.2
```

There are two deleted metadata records stored in the PERSON audit file.

CLASNAME
: displays the records' SAS Metadata Server object class name.

ID
: displays the records' unique object instance identifier.

OBJNAME
: displays the value in the records' Name attribute.

LDESC
: displays the value in the records' Desc attribute.

DELETED
: displays a SAS datetime value indicating when each record was deleted.

Example: Restoring Audit Records

The following is an example of a DATA step that restores the metadata record describing "Carla" to a

repository.

```
/* Set the repository and audit librefs */
libname repos 'repos\repos1';
libname audit 'repos\audit\repos1';

/* Select audit records to be restored */
/* the DELETED field is set to missing which activates the record. */
data work.select (drop=find);
  set audit.person( drop= _ATYPE_ _ATIME_ _ATID_ );
  /* find 'A5JZBYC7.A1000002' */
     find='A' || '00'x ||
          '5' || '00'x ||
          'J' || '00'x ||
          'Z' || '00'x ||
          'B' || '00'x ||
          'Y' || '00'x ||
          'C' || '00'x ||
          '7' || '00'x ||
          '.' || '00'x ||
          'A' || '00'x ||
          '1' || '00'x ||
          '0' || '00'x ||
          '0' || '00'x ||
          '0' || '00'x ||
          '0' || '00'x ||
          '0' || '00'x ||
          '2' || '00'x ;
  DELETED = . ;
  if ID = find then OUTPUT;
run;

/* Print the selected audit records. */
proc print data=work.select;
   var clasname id objname ldesc deleted;
run;

/* Append selected audit records to the repository file. */
data repos.person;
   set repos.person work.select;
run;
```

Here is the output written to the SAS Output window:

```
12:12 Monday, October 28, 2002

Obs CLASNAME
 1  O M S p e r s o n

Obs ID
 1  A 5 J Z B Y C 7 . A 1 0 0 0 0 0 2

Obs OBJNAME
 1  C a r l a

Obs LDESC
 1  S t u d e n t

Obs DELETED
 1  .
```

Here is the information written to the SAS Log:

```
NOTE: Libref REPOS was successfully assigned as follows:
      Engine:        V9
      Physical Name: U:\tkmeta\repos\repos1
NOTE: Libref AUDIT was successfully assigned as follows:
      Engine:        V9
      Physical Name: U:\tkmeta\repos\audit\repos1
NOTE: There were 2 observations read from the data set AUDIT.PERSON.
NOTE: The data set WORK.SELECT has 1 observations and 14 variables.
NOTE: DATA statement used (Total process time):
      real time           0.73 seconds
      cpu time            0.04 seconds
NOTE: There were 1 observations read from the data set WORK.SELECT.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
NOTE: There were 4 observations read from the data set REPOS.PERSON.
NOTE: There were 1 observations read from the data set WORK.SELECT.
NOTE: The data set REPOS.PERSON has 5 observations and 14 variables.
NOTE: DATA statement used (Total process time):
      real time           0.14 seconds
      cpu time            0.01 seconds
```

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Locking Metadata Objects

The SAS Open Metadata Interface enables you to control concurrent access to metadata by multiple users in a number of ways:

- You can perform object−level locking within a given repository by setting the SAS Open Metadata Interface OMI_LOCK and OMI_UNLOCK flags.
- You can impose a change management process in which metadata objects are locked and checked from a primary repository to a project repository by using the SAS Open Metadata Interface change management facility.
- You can set up dedicated metadata servers representing the phases of the metadata development life cycle and promote repositories between the servers.
- You can use a combination of these mechanisms.

In order to use any of these mechanisms, a user must have a registered identity on the metadata server. For more information, see "Registering Users" in the **SAS 9.1 Metadata Server: Setup Guide**.

For more information about each option, see:

- Using SAS Open Metadata Interface Flags to Lock Objects
- Promoting Repositories Between Servers
- Using the Change Management Facility

Using SAS Open Metadata Interface Flags to Lock Objects

The SAS Open Metadata Interface provides the OMI_LOCK, OMI_UNLOCK, and OMI_UNLOCK_FORCE flags to lock metadata objects. The flags represent the simplest of the concurrency controls provided by the SAS Open Metadata Architecture. Before making an update, you simply issue a GetMetadata call on the objects that you wish to modify that sets the OMI_LOCK flag. The software locks the specified object and any associated objects specified by GetMetadata flags and options, and updates the objects' LockedBy attribute with the metadata identifier representing the calling user. The objects remain locked from update by users other than the calling identity until an UpdateMetadata method is issued that sets the OMI_UNLOCK or OMI_UNLOCK_FORCE flags.

OMI_UNLOCK releases a lock held by the calling user and is the recommended method of releasing a lock. OMI_UNLOCK_FORCE unlocks a lock held by another user and is intended to be used as an emergency override mechanism. While locked, objects can be read by other users; they simply cannot be updated by other users until the user holding the lock completes his update.

The LockedBy attribute is set and cleared automatically by the lock flags and can be queried to determine whether an object is locked and who holds the lock. The LockedBy attribute cannot be changed or cleared with the UpdateMetadata method.

In SAS 9.1., object−level locking is supported only in the SAS Open Metadata Interface. That is, you must write a SAS Open Metadata Interface client to lock objects, update them, then unlock them. For information about writing a SAS Open Metadata Interface client, see Client Requirements in the **SAS Open Metadata Interface: Reference**. For more information about the GetMetadata and UpdateMetadata methods, see GetMetadata and UpdateMetadata in IOMI (OMI) Class in the **SAS Open Metadata Interface: Reference**.

Promoting Repositories Between Servers

Repository promotion is the most restrictive of the concurrency controls. It involves maintaining multiple metadata servers and versions of repositories.

The promotion process is implemented by using the Metadata Manager plug−in to SAS Management Console. The Metadata Manager contains Replication and Promotion wizards that enables you to create a job definition for copying a repository and its contents from one metadata server to another. The replication process copies a repository on the source metadata server and recreates it on the target metadata server using the same repository and object identifiers. The promotion process copies a repository on the source metadata server and recreates it on the target metadata server while modifying certain repository attributes.

Before using SAS Management Console, you must set up and start a metadata server for each life cycle phase that you wish to represent. For example, if you plan to support development, test, and production environments, set up a metadata server for each of these environments. Instructions for setting up a metadata server are provided in the **SAS 9.1 Metadata Server: Setup Guide.** See "Replicating and Promoting Metadata" in the **SAS 9.1 Management Console: User's Guide** for information about using the wizards.

Previous Page | Next Page | Top of Page

Using the Change Management Facility

- Introduction
- Summary of Change Management Methods
- Overview of Change Management Tasks
- How Access Controls are Handled by the Change Management Facility

Introduction

The SAS Open Metadata Interface provides an optional change management facility for controlling concurrent access to metadata. Concurrent access is the ability for multiple users to safely update metadata at any given time. For example, an enterprise may have an environment set up that allows developers to make changes to metadata in an area that is independent of the existing "primary" metadata. The change management facility will allow changes to occur to existing metadata without interfering with existing usage. In addition, as changes are made, information relative to the change is maintained. This information can be used to track change history in determining who made the change to the object, when the change was made, and how it related to other objects in the metadata.

The facility consists of a set of methods that enable users to "check out" metadata that they wish to update from a primary repository to a project repository, then check it back in again when the update is complete. During checkout, metadata in the primary repository is locked to other users. The primary repository represents your stable environment of non−changing metadata and the project repository represents a development area where changes are made and tested. When metadata in a primary repository is locked by a user, other users may see the metadata in order to continue their work, but they will not be allowed to update it. When the modified metadata has been fully tested and is "checked in" to the primary repository, any subsequent reads by another user will reflect the changes just created. The change history is created as the objects are checked in to the primary repository from the project repository.

The facility also provides FetchMetadata and CopyMetadata methods. A "fetch" operation copies metadata objects from a primary repository to a project repository without locking them. Fetched metadata objects can be modified without fear of updating the corresponding primary repository object. Fetching metadata objects is useful when a developer needs to complete the most robust testing while only locking objects from other users that require changes. CopyMetadata is provided as a way to create new metadata objects in the project repository that can be added to a primary repository. For example, an enterprise may need to create a new primary repository that is a hybrid of other repositories. In this case, the developer could copy metadata from the two sources into a new project, modify it as needed, and incorporate the entire project into a new primary repository. The CopyMetadata method allows a metadata object to be copied from a project repository or a primary repository source.

In order to use the change management facility, you must set up a primary and a project repository, register users who will update the primary repository, and use change management methods to move metadata between the primary and project repositories.

Summary of Change Management Methods

The change management methods exist in the IOMI class and include:

CheckoutMetadata
: locks and copies metadata objects from a primary repository to a project repository.

CheckinMetadata
: copies metadata objects from a project repository to a primary repository and unlocks them.

CopyMetadata
: copies metadata objects within a repository or between repositories.

FetchMetadata
: copies metadata objects from a primary repository to a project repository without locking them.

UndoCheckoutMetadata
: deletes and unlocks metadata objects that were checked out to a project repository by mistake. Also deletes fetched objects from the project repository.

See the **SAS 9.1 Open Metadata Interface: Reference** for reference information about the methods.

Overview of Change Management Tasks

1. Setting Up Primary and Project Repositories
2. Registering Users
3. Assigning Users Permissions to the Primary and Project Repositories
4. Checking Out Metadata Objects that you wish to permanently modify using CheckoutMetadata
5. Fetching Metadata objects that you wish to temporarily modify then discard using FetchMetadata
6. Updating Metadata Objects in the Project Repository
7. Adding and Copying Metadata Objects in a Project Repository
8. Deleting a Metadata Object
9. Checking in Metadata Objects
10. Querying the Project Repository
11. Querying the Primary Repository
12. Emptying the Project Repository

Setting Up Primary and Project Repositories

A *primary repository* is a base−level or "production" repository. Objects in a primary repository are not updated directly when using the change management facility. There can be more than one primary repository for any given project repository.

The *project repository* is a playpen repository where metadata objects are modified. This repository is empty except for metadata objects copied from a primary repository to the project repository by using the CheckoutMetadata or FetchMetadata methods and new objects created for the purpose of updating the primary repository. New objects are created by using AddMetadata or CopyMetadata. There can be more than one project repository for any given primary repository.

In order for change management to be implemented between two repositories, you must establish a relationship between the repositories. A relationship is established by creating a DependencyUses or a DependencyUsedBy association between the repositories. For the purpose of change management, it is recommended that the project repository have a DependencyUses association to any primary repositories. This causes any cross−repository references created by change management methods to be stored in the project repository, which makes the references easier to query, particularly if multiple primary repositories are defined for a given project repository.

You are led through the steps for creating the necessary dependencies when you select to create a Project repository using the Add Repository wizard of the Metadata Manager plug−in of SAS Management Console. You can also create dependencies when you create a repository using the SAS Open Metadata Interface. The SAS Open Metadata Interface also allows you to add dependencies to existing repositories. For more information about creating repositories using SAS Management Console, see the Help for the Add Repository wizard. For information about creating repository dependencies using the SAS Open Metadata Interface, see Creating Relationships Between Repositories.

Registering Users

In order to use change management methods, a user must have an identity (Person or IdentityGroup metadata object) and associated Login object(s) defined for him or her on the metadata server. These identities are created in the SAS Management Console User Manager plug−in. You create Person metadata objects when you add a New User in the SAS Management Console User Manager. You create IdentityGroup metadata objects when you add a New Group in the User Manager. (An IdentityGroup represents a group of users that log into the server using a shared login.)

Identities are protected resources on a metadata server. A user requires Administrative User status on the server in order to create a metadata identity for an individual user. Only Administrative Users and users that have been granted specific permissions to an existing user or group identity can update it. Other requirements are described in "Registering Users" in "Using the Authorization Facility" in the **SAS 9.1 Metadata Server: Setup Guide**.

Assigning Users Permissions to the Primary and Project Repositories

An identity must have CheckinMetadata permission to a primary repository in order to update it with changes from a project repository. An Administrative User can assign the CheckinMetadata permission in the primary repository's Default ACT or on individual objects within a primary repository. An identity that has CheckinMetadata permission in a primary repository also needs ReadMetadata and WriteMetadata permission in the corresponding project repository, so that the objects can be written to the project repository when they are checked−out. ReadMetadata and WriteMetadata permission must be assigned in the project repository's Default ACT. The Default ACT is the only access control evaluated when making an authorization decision for an object in a project repository.

The Default ACT created for a project repository has different default settings than a Default ACT created for a custom or foundation repository. The Default ACT for a project repository denies the PUBLIC group ReadMetadata and WriteMetadata access to the repository and to all objects in the repository, and grants ReadMetadata and WriteMetadata permission to the repository owner. The repository owner is the user ID that created the repository. The repository owner must modify a project repository's Default ACT to grant ReadMetadata and WriteMetadata permission to other users.

For more information about assigning the CheckinMetadata, ReadMetadata, and WriteMetadata permissions, see "Defining Access Controls" in "Using the Authorization Facility" in the **SAS 9.1 Metadata Server: Setup Guide**.

How Access Controls are Handled by the Change Management Facility

In order to maintain the integrity of authorization metadata that might have been defined to protect the primary and project repositories, authorization metadata (AccessControl, AccessControlEntry, AccessControlTemplate, SecurityRule, SecurityRuleSchema, SecurityContainmentRule, Permission, and PermissionCondition metadata objects) cannot be checked out from a primary repository to a project repository. Also, SAS Management Console prevents objects of these types from being created in a project repository. The Default ACT is the only access control evaluated when making an authorization decision for an object in a project repository.

Checking Out Metadata Objects

The CheckoutMetadata method is the primary form of object movement from a primary repository to a project repository. Conceptually, the method

- locks the requested object in the primary repository
- copies it to the specified project repository
- stores the metadata identifier of the person who locked the object in the LockedBy attribute of the primary repository object
- stores information about the object's modified status in the ChangeState attribute of both the original and copied metadata objects.

When an object is checked out, associated objects specified in a default lock template are checked out as well. If a default template does not exist for a type, or you wish to override the default lock template, you can specify your own lock template by setting the OMI_LOCK_TEMPLATE (65536) flag and supplying a list of associated objects that you wish to include, in template form, in a <LockTemplates> element in the *Options* parameter.

Lock Templates

A *lock template* is an XML property string that specifies a set of associated objects that are processed in addition to the metadata object specified in a CheckoutMetadata, FetchMetadata, or CopyMetadata request. Because metadata objects are so interconnected, checkout of a single object is not very useful and could be dangerous to metadata integrity. To maintain metadata integrity, the change management facility uses lock templates to enable associated objects to be checked out, fetched, and copied as a group.

The change management facility provides a default lock template for the metadata types described in Default Lock Templates. You can specify your own lock template in any of these methods by setting the OMI_LOCK_TEMPLATE (65536) flag and supplying a template identifying the associated objects that you wish to include in a <LockTemplates> element in the *Options* parameter.

A lock template specifies a metadata type and the association names identifying the associated objects that you wish to retrieve. For example, if you are checking out a SASLibrary object and want to additionally check out all Property metadata objects associated with it, you would specify the following template in the <LockTemplates> element:

```
<SASLibrary>
   <Properties/>
</SASLibrary>
```

'SASLibrary' is the metadata type whose associated objects you are interested in. 'Properties' is the association name that represents the relationship between SASLibrary and Property metadata objects.

To additionally check out PropertyType objects defined for each Property object that is retrieved, additionally specify the following template:

```
<Property>
   <OwningType/>
</Property>
```

In this template, 'Property' is the metadata type whose associated objects you are interested in and 'OwningType' is the association name that represents the relationship between Property and PropertyType metadata objects.

The templates are submitted together in the <Options> parameter as follows:

```
<Options>
 <LockTemplates>
  <SASLibrary>
    <Properties/>
  </SASLibrary>
  <Property>
    <OwningType/>
  </Property>
 </LockTemplates>
</Options>
```

Note: The lock template used to check out or fetch a metadata object must be used to undo its checkout. Consult the metadata type documentation in the **SAS 9.1 Open Metadata Interface: Reference** to determine the association names required to retrieve associated objects.

Issuing a CheckoutMetadata Call

The following is an example of a CheckoutMetadata call. The CheckoutMetadata method locks and copies specified metadata objects to a specified project repository. You identify the metadata object to be checked out by passing a metadata property string consisting of the object's metadata type and a unique two−part primary repository object identifier in the method's *inMetadata* parameter. You can specify multiple metadata objects in a CheckoutMetadata request by stacking their property strings in the *inMetadata* parameter. You identify the target project repository in the *ProjReposid* parameter. The following is an example of a stacked CheckoutMetadata request:

```
<CheckoutMetadata>
   <Metadata>
      <PhysicalTable ID="PrimaryReposid.Objectid"/>
      <PhysicalTable ID="PrimaryReposid.Objectid"/>
   </Metadata>
   <ProjReposid>Reposmgrid.ProjectReposid</ProjReposid>
   <ns>SAS</ns>
   <flags>0</flags>
   <Options/>
</CheckoutMetadata>
```

The request locks and copies two PhysicalTable objects from a primary repository to a project repository. A default lock template is associated with the PhysicalTable metadata type, therefore, any Column, Key, Property, Index, Document, TextStore, Extension, and Role objects associated with the PhysicalTable objects will be checked out as well.

LockedBy Attribute

When a metadata object is checked out from a primary repository to a project repository, the metadata identifier of the person who initiated the action is stored in the primary metadata object's LockedBy attribute. An unlocked object has an empty string in the LockedBy attribute. You can determine which objects in a primary repository are locked by querying which objects have a value in the LockedBy attribute. For more information, see Querying the Primary Repository.

ChangeState Attribute

When a metadata object is copied from a primary repository to a project repository by using the CheckoutMetadata or FetchMetadata methods, information describing the action is recorded in a ChangeState attribute in one or both of the objects. Every metadata type in the SAS Metadata Model has a ChangeState attribute, which is used exclusively by the change management facility.

In a primary repository object, the ChangeState attribute stores an empty string or the value:

```
Checked-out to: <ProjIdentifier>
```

An empty string indicates the object is available for reading and writing in the primary repository. The value indicates the object is currently locked and checked−out to a project repository and stores the unique 17−character identifier of the object's project repository counterpart. The ChangeState attribute of a fetched object contains an empty string because a fetched object is not locked, and could be fetched to more than one project repository. In a primary repository, the ChangeState attribute can be queried to determine which objects are checked out and to what project repository. For more information, see Querying the Primary Repository.

In a project repository object, the value of the ChangeState attribute has the form:

CMState:*SrcIdentifier*

where

- *CMState* identifies the process that copied the object into the project repository: one of Checkout, Fetch, New, or an empty string. Objects created in a project repository by using the AddMetadata or CopyMetadata method either have an empty string or the word "New" in this position.
- *SrcIdentifier* is either the 17−character identifier of the object's primary repository counterpart or an empty string. For new objects, it is recommended that you update this attribute to supply the unique eight−character identifier of a destination primary repository.

The change management facility uses the ChangeState attribute to determine a project repository object's disposition at checkin as follows:

- an object whose *CMState* is Checkout will be used to update the primary repository object identified in *SrcIdentifier*.

- an object whose *CMState* is Fetch will be ignored.
- an object whose *CMState* is New or an empty string will be created in the primary repository identified in *SrcIdentifier*.
- an object whose *SrcIdentifier* is an empty string will be created in the first repository found that has a DependencyUsedBy association to the project repository during checkin.

In a project repository, you can query the ChangeState attribute to determine how a given object came to be in the project repository, to which primary repository a project repository object belongs, and to identify new objects. For more information, see Querying the Project Repository

Fetching Metadata

The FetchMetadata method creates object instances in the project repository that cannot be copied back to the corresponding primary repository. This enables you to modify the fetched object without fear of affecting the corresponding primary object.

Like CheckoutMetadata, the FetchMetadata method copies the specified metadata object and any objects identified in the object's default lock template from a primary repository to the specified project repository. Unlike CheckoutMetadata, the objects are not locked in the primary repository.

You identify the metadata object to fetch using its metadata type and unique object instance identifier in the method's *inMetadata* parameter. You can fetch multiple metadata objects by stacking their property strings in the *inMetadata* parameter. You can specify an alternate, user–defined lock template by setting the OMI_LOCK_TEMPLATE flag and submitting a lock template in a <LockTemplates> element in the *Options* parameter. For information about how to create a lock template, see Lock Templates. The following is an example of a FetchMetadata request that specifies an alternate lock template:

```
<FetchMetadata>
   <Metadata>
      <PhysicalTable ID="PrimaryReposid.Objectid"/>
   </Metadata>
   <ProjectReposid>Reposmgrid.ProjectReposid</ProjectReposid>
   <NS>SAS</NS>
   <!-- OMI_LOCK_TEMPLATE flag -->
   <Flags>65536</Flags>
   <Options>
    <LockTemplates>
       <PhysicalTable/>
         <UniqueKeys/>
       </PhysicalTable/>
    </LockTemplates>
</FetchMetadata>
```

The specified lock template overrides the default lock template for a PhysicalTable object and retrieves only objects that have a UniqueKeys association to the specified PhysicalTable object.

Updating Metadata Objects in a Project Repository

All metadata objects in a project repository are referenced using a project repository identifier. For example, an UpdateMetadata call modifying an object to create an association to another existing object identifies the

objects as shown below:

```
<UpdateMetadata>
  <Metadata>
    <PhysicalTable Id="projectReposid.Objectid">
      <TablePackage>
        <SASLibrary ObjRef="projectReposid.Objectid"/>
      </TablePackage>
    </PhysicalTable>
  </Metadata>
  <NS>SAS</NS>
  <!-- OMI_TRUSTED_CLIENT flag -->
  <Flags>268435456</Flags>
  <Options/>
</UpdateMetadata>
```

Adding and Copying Metadata Objects in a Project Repository

New metadata objects created in a project repository by using the AddMetadata, CopyMetadata, or UpdateMetadata methods are also created with the repository identifier of the project repository. In environments where a single project repository serves multiple primary repositories, it is recommended that you specify the target primary repository by defining a value for the ChangeState attribute along with the object's other properties. The following is an example of an AddMetadata call that creates a metadata object and an associated metadata object in a project repository and defines values for their ChangeState attributes.

```
<AddMetadata>
  <Metadata>
    <SASLibrary Name="Test library 1" Desc="Library object created in project repository"
      ChangeState="New:AFXFGX9C"/>
        <Tables>
          <PhysicalTable Name="Test table 1" Desc="Table object created in project repository"
             ChangeState="New:AFXFGX9C"/>
        </Tables>
    </SASLibrary>
 </Metadata>
 <!--Specify project repository identifier-->
 <Reposid>A0000001.A5NZA58I</Reposid>
 <NS>SAS</NS>
 <!-- OMI_TRUSTED_CLIENT flag -->
 <Flags>268435456</Flags>
 <Options/>
</AddMetadata>
```

The form of a ChangeState attribute value is described in ChangeState Attribute. In this example, "New" indicates the objects' state; "AFXFGX9C" is the eight−character identifier of the target primary repository. When specifying a SrcIdentifier for a new metadata object, there is no need to specify a full 17−character repository identifier.

The following is an example of an UpdateMetadata call that creates an association and a new associated object that will ultimately reside in a different primary repository than the main object.

```
<UpdateMetadata>
  <Metadata>
    <!-- Main element specifies project repository identifier -->
    <SASLibrary Id="A5NZA58I.A0000033" Name="Test library 1"/>
```

```
        <Tables>
          <!-- Subelement omits Id attribute and specifies ChangeState attribute -->
          <PhysicalTable Name="Test table 3" Desc="Table object destined for different primary
             ChangeState="New:ALCLMNOC"/>
        </Tables>
    </SASLibrary>
 </Metadata>
 <NS>SAS</NS>
 <!-- OMI_TRUSTED_CLIENT flag -->
 <Flags>268435456</Flags>
 <Options/>
</UpdateMetadata>
```

The ChangeState object of the associated object specifies a different primary repository identifier than the one defined for Text Library 1 in the preceding AddMetadata example.

You can use UpdateMetadata to specify a ChangeState value for an object created by CopyMetadata.

A project repository object whose ChangeState attribute is an empty string will be copied to the first repository found that has a DependencyUsedBy association to the project repository during checkin.

Deleting a Metadata Object

The method used to delete a metadata object from a project repository depends on whether the object will also be deleted from the primary repository:

- A metadata object that will be deleted from the primary repository is deleted using DeleteMetadata.
- A metadata object that will remain in the primary repository (it was checked out by mistake or was copied into the project repository using the FetchMetadata method) is removed using UndoCheckoutMetadata.

The following is an example of a DeleteMetadata request that is executed on a project repository:

```
<DeleteMetadata>
   <Metadata>
   <!--Main element specify the object's project repository identifier -->
      <PhysicalTable ID="A5NZA58I.AA000002"/>
   </Metadata>
   <NS>SAS</NS>
   <!-- OMI_TRUSTED_CLIENT flag -->
   <Flags>268436480</Flags>
   <Options/>
</DeleteMetadata>
```

The following is an example of an UndoCheckoutMetadata request:

```
<UndoCheckoutMetadata>
   <Metadata>
   <!--Main element specifies the object's project repository identifier -->
      <PhysicalTable ID="A5NZA58I.AB000004"/>
   </Metadata>
   <NS>SAS</NS>
   <Flags>0</Flags>
   <Options/>
```

```
</UndoCheckoutMetadata>
```

Note: The same lock template used to check out or fetch a metadata object should be used to undo the checkout.

The DeleteMetadata method does not support lock templates. The DeleteMetadata method will delete any associated objects that are dependent on the deleted objects. However, if you wish to delete other associated objects, you will need to issue additional DeleteMetadata requests. For more information, see DeleteMetadata.

Checking In Metadata Objects

Checkin is the process of integrating the changes from a project repository into the corresponding primary repository. Checkin of individual metadata objects is not supported; the CheckinMetadata method copies all eligible objects in a project repository to their corresponding primary repositories at once. An "eligible" metadata object is one that was copied to the project repository using CheckoutMetadata and created in the project repository using AddMetadata, CopyMetadata, or UpdateMetadata. Objects copied into the project repository using FetchMetadata are ineligible. Conceptually, the CheckinMetadata method

- updates and unlocks checked metadata objects in their corresponding primary repository
- copies and creates new metadata objects in the primary repository identified in their ChangeState attribute
- deletes metadata objects deleted by DeleteMetadata from their corresponding primary repository
- updates the primary repository's association list, as described in Association Handling
- creates a Change object in each primary repository that stores information about the changes.
- removes checked and new metadata objects from the project repository. Fetched objects remain in the project repository, unless the OMI_CI_DELETE_ALL (33) flag is set to delete them. You can keep copies of checked and new metadata objects in the project repository by setting the OMI_CI_NODELETE (524288) flag in the CheckinMetadatarequest.

Association Handling

Information about a metadata object's associations is maintained separately from its attributes in an association list. When an object is checked out to a project repository, both its attributes and association list are copied; however, only the attributes are locked. The association list in the primary repository can still be updated if the partner object in the association is updated. For example, if you check out a PhysicalTable object that has a Groups association to a Group object, but do not also check out the Group object, then the association list can still be modified by updating the Group object's Members association. (Group has a Members association to PhysicalTable.) To protect the integrity of an object's associations, it is recommended that you check out all important associated objects when checking out an object. You can check these objects out individually, or create a user−defined lock template to check them out.

At checkin, the change management facility compares the contents of the association list in the project repository to the association list in the primary repository and applies updates as follows:

1. Associations marked as deleted in the project repository association list are deleted from the primary repository association list.
2. New associations in the project repository association list are written to the primary repository association list.

3. Associations that exist in both the project and primary repository association lists are overwritten by those in the project association list.
4. Associations that exist only in the primary association list are left alone.

The association order is modified as little as possible.

Change Objects

The CheckinMetadata method creates a Change object in each primary repository in which it copies or deletes metadata objects. This Change object stores information about the checkin, including the date and time the Change object was created (and thus, the time its corresponding changes were integrated into the primary repository), a name and a description for the change as supplied in the CheckinMetadata ChangeName and ChangeDesc parameters, and two associations:

ChangeIdentity
: links the Change object to the identity object (Person or Group) representing the user who made the change.

Objects
: links the Change object to the metadata objects in the primary repository that were part of the change.

The purpose of the Change object is to enable you to query the primary repository for information about an update. For more information, see Querying the Primary Repository.

Querying the Primary Repository

You might want to query the objects in a primary repository to determine which objects are available for update, which objects are locked, who locked them, and what changes were made.

You can determine which objects are available for update in a primary repository by querying which objects have an empty string in the LockedBy attribute. The LockedBy attribute stores a value for objects locked by using either the OMI_LOCK flag or the CheckoutMetadata method. An empty string indicates an object is not locked. Use the GetMetadataObjects method to query the objects as follows:

```
<GetMetadataObjects>
  <!--Repository of primary repository -->
  <Reposid>A0000001.A3N4OMZV</Reposid>
  <Type>Root</Type>
  <Objects/>
  <NS>SAS</NS>
  <!-- OMI_INCLUDE_SUBTYPES, OMI_XMLSELECT, OMI_GET_METADATA, OMI_TEMPLATES flags -->
  <Flags>404</Flags>
  <Options>
   <XMLSelect search="*[@LockedBy = ' ']"/>
   <Templates>
     <Root Id="" LockedBy=""/>
   </Templates>
  </Options>
</GetMetadataObjects>
```

In the request, note:

- *Type* specifies the Root metadata type.
- The OMI_INCLUDE_SUBTYPES flag is set to return all metadata types that are subtypes of Root.
- The OMI_GET_METADATA and OMI_TEMPLATES flags are set to pass a template to retrieve the Id and LockedBy attributes for all objects.
- The OMI_XMLSELECT flag is set and <XMLSelect> search string is specified to filter the request to select objects that have an empty string in the LockedBy attribute.

To identify the objects locked by a given identity (user or group), specify the metadata identifier of the identity in the search string.

To determine which objects are checked−out by the CheckoutMetadata method, issue the same method call as above but specify the ChangeState attribute and a value in the <XMLSelect> search string instead of the LockedBy attribute. For example:

```
<GetMetadataObjects>
  <!--Repository of primary repository -->
  <Reposid>A0000001.A3N4OMZV</Reposid>
  <Type>Root</Type>
  <Objects/>
  <NS>SAS</NS>
  <!-- OMI_INCLUDE_SUBTYPES, OMI_XMLSELECT, OMI_GET_METADATA, OMI_TEMPLATES flags -->
  <Flags>404</Flags>
  <Options>
   <XMLSelect search="*[@ChangeState CONTAINS 'Checked-out to']"/>
   <Templates>
     <Root Id="" ChangeState="" LockedBy=""/>
   </Templates>
  </Options>
</GetMetadataObjects>
```

The search string will return all objects that have the words 'Checked−out to:' in the ChangeState attribute. To determine which objects are checked out to a specific project repository, include the project repository's 17−character identifier in the search string.

To obtain information about changes to a primary repository, issue a GetMetadataObjects call that specifies the Change metadata type in the *Type* parameter and sets the OMI_GET_METADATA (256), OMI_ALL (1), and OMI_SUCCINCT (2048) flags as follows:

```
<GetMetadataObjects>
  <!--Repository of primary repository -->
  <Reposid>A0000001.A5WW3LXC</Reposid>
  <Type>Change</Type>
  <Objects/>
  <NS>SAS</NS>
  <!-- OMI_GET_METADATA, OMI_ALL, and OMI_SUCCINCT flags -->
  <Flags>2305</Flags>
  <Options/>
</GetMetadataObjects>
```

In the call:

- The OMI_GET_METADATA and OMI_ALL flags specify to retrieve all properties for all Change objects that are found.
- The OMI_SUCCINCT flag limits the request to properties that have values defined for them. The output will be the Id, Name, and Desc attributes of all Change objects as well as any objects that have

a ChangeIdentity and Objects association to each Change object. The ChangeIdentity association will list the identity object representing the user who made the change. The Objects association will list all metadata objects affected by a change.

Querying the Project Repository

You might want to query the objects in a project repository to determine which came from a given repository and to identify how they came to be in the project repository (checkout, fetch, or new). All of this information can be obtained by querying the project repository objects' ChangeState attribute.

The following GetMetadataObjects example queries a project repository to determine which objects originated in primary repository AFXFGX9C.

```
<GetMetadataObjects>
  <!--Repository of project repository -->
  <Reposid>A0000001.A5NZA58I</Reposid>
  <Type>Root</Type>
  <Objects/>
  <NS>SAS</NS>
  <!-- OMI_INCLUDE_SUBTYPES, OMI_XMLSELECT, OMI_GET_METADATA, and OMI_TEMPLATES flags -->
  <Flags>404</Flags>
  <Options>
   <XMLSelect search="*[@ChangeState CONTAINS 'AFXFGX9C']"/>
   <Templates>
     <Root Id="" Name="" LockedBy=""/>
   </Templates>
  </Options>
</GetMetadataObjects>
```

In the call:

- *Type* specifies the Root metadata type.
- The OMI_INCLUDE_SUBTYPES flag is set to return all metadata types that are subtypes of Root.
- The OMI_XMLSELECT flag is set and an <XMLSelect> search string specifies to filter the request to select objects that have the value 'AFXFGX9C' in the ChangeState attribute.
- The OMI_GET_METADATA and OMI_TEMPLATES flags are set to pass a template to retrieve the Id, Name, and LockedBy attributes for the selected objects.

To identify fetched objects, issue the same method call modifying the search string as follows:

```
search="*[@ChangeState CONTAINS 'Fetch']"
```

To identify new objects, issue the method call with the search string:

```
search="*[@ChangeState CONTAINS 'New' or @ChangeState = ' ']"
```

For more information about the GetMetadataObjects method, see Overview of Retrieving Objects of a Specified Metadata Type.

Emptying the Project Repository

To clear a repository whose objects have already been checked in, issue a DeleteMetadata request on the repository, setting the OMI_REINIT (2097152) flag. The OMI_REINIT flag deletes all metadata objects from a repository without deleting the repository itself and reinitializes the repository.

Note: A user must have administrative user status in order to delete or clear a repository. For more information, see "Server Administrative Privileges" in the **SAS 9.1 Metadata Server: Setup Guide**.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Default Lock Templates

To protect metadata integrity, the change management facility supports a set of default lock templates for certain metadata types. Default lock templates are defined for the Document, ExternalTable, Job, PhysicalTable, Report, SASLibrary, ServerComponent, ServerContext, TextStore, and Tree metadata types. If a default template does not exist for a type, then only the indicated object is processed (no associated objects), unless you specify a user−defined lock template.

The associated objects defined in each default lock templates are shown in the sections that follow.

Document Lock Template

```
<Document>
  <Documents/>
  <Notes/>
  <Extensions/>
</Document>
```

ExternalTable Lock Template

```
<ExternalTable>
  <Columns/>
  <OwningFile/>
  <Extensions/>
  <Notes/>
  <Documents/>
  <Properties/>
</ExternalTable>
<Document>
 <Documents/>
 <Notes/>
 <Extensions/>
</Document>
```

Job Lock Template

```
<Job>
  <JobActivities/>
  <SourceCode/>
  <Extensions/>
  <Notes/>
  <Documents/>
  <Properties/>
  <TransformationSources/>
  <TransformationTargets/>
</Job>
<TransformationActivity>
  <Steps/>
  <TransformationTargets/>
  <SourceCode/>
  <Extensions/>
```

```
  <Notes/>
  <Documents/>
  <Properties/>
</TransformationActivity>
<TransformationStep>
  <SuccessorDependencies/>
  <PredecessorDependencies/>
  <Transformations/>
  <SourceCode/>
  <Extensions/>
  <Notes/>
  <Documents/>
  <Properties/>
  <UsingPrototype/>
  <PropertySets/>
</TransformationStep>
<Select>
  <ClassifierTargets/>
  <ClassifierSources/>
  <FeatureMaps/>
  <AssociatedRowSelector/>
  <HavingForSelect/>
  <GroupByForSelect/>
  <WhereForSelect/>
  <OrderByForSelect/>
  <SourceCode/>
  <Extensions/>
  <Notes/>
  <Documents/>
  <Properties/>
  <PropertySets/>
  <SubstitutionVariables/>
</Select>
<ClassifierMap>
  <FeatureMaps/>
  <ClassifierTargets/>
  <ClassifierSources/>
  <AssociatedRowSelector/>
  <SourceCode/>
  <Extensions/>
  <Notes/>
  <Documents/>
  <Properties/>
  <PropertySets/>
  <SubstitutionVariables/>
</ClassifierMap>
<RowSelector>
  <SourceCode/>
</RowSelector>
<OrderByClause>
 <Columns/>
</OrderByClause>
<FeatureMap>
  <FeatureSources/>
  <FeatureTargets/>
  <SourceCode/>
  <Extensions/>
  <Notes/>
  <Documents/>
  <Properties/>
  <SubstitutionVariables/>
```

```
</FeatureMap>
<WorkTable>
 <Columns/>
</WorkTable>
<PhysicalTable>
  <Columns/>
  <UniqueKeys/>
  <Properties/>
  <Indexes/>
  <Documents/>
  <Notes/>
  <Extensions/>
  <Roles/>
  <PropertySets/>
  <ForeignKeys/>
</PhysicalTable>
<Column>
  <Documents/>
  <Notes/>
  <Extensions/>
  <UniqueKeyAssociations/>
  <Keys/>
</Column>
<Index>
  <Documents/>
  <Notes/>
  <Extensions/>
</Index>
<Property>
  <SpecTargetTransformations/>
  <StoredConfiguration/>
</Property>
<Prototype>
  <PrototypeProperties/>
  <Properties/>
</Prototype>
<Transformation>
  <SourceCode/>
  <SourceSpecifications/>
</Transformation>
<PropertySet>
  <Properties/>
</PropertySet>
<Variable>
  <AssociatedObject/>
</Variable>
<UniqueKey>
  <ForeignKeys/>
</UniqueKey>
<ForeignKey>
  <KeyAssociations/>
</ForeignKey>
<Cube>
  <TargetTransformations/>
  <Aggregations/>
  <AssociatedFile/>
  <Dimensions/>
  <Hierarchies/>
  <Measures/>
  <Properties/>
  <PropertySets/>
```

```
  <Documents/>
  <Notes/>
  <Extensions/>
</Cube>
<Aggregation>
  <AggregateAssociations/>
  <Levels/>
  <AggregationTables/>
  <Properties/>
</Aggregation>
<AggregateAssociation>
  <AggregatedColumns/>
  <MeasureOrLevel/>
  </AggregateAssociation>
<Dimension>
  <TargetTransformations/>
  <Hierarchies/>
  <Levels/>
  <Measures/>
  <DefaultMeasure/>
  <Properties/>
  <Notes/>
</Dimension>
<Hierarchy>
  <Levels/>
  <OLAPProperties/>
  <Notes/>
  <Properties/>
</Hierarchy>
<Measure>
  <TargetTransformations/>
  <DefaultForDimension/>
  <MLAggregations/>
  <Notes/>
</Measure>
<Level>
  <MLAggregations/>
  <Notes/>
  <Properties/>
</Level>
<Document>
  <Documents/>
  <Notes/>
  <Extensions/>
</Document>
```

PhysicalTable Lock Template

```
<PhysicalTable>
  <Columns/>
  <UniqueKeys/>
  <Properties/>
  <Indexes/>
  <Documents/>
  <Notes/>
  <Extensions/>
  <Roles/>
</PhysicalTable>
<Column>
```

```
  <Documents/>
  <Notes/>
  <Extensions/>
  <UniqueKeyAssociations/>
  <Keys/>
</Column>
<Index>
  <Documents/>
  <Notes/>
  <Extensions/>
</Index>
<PropertySet>
  <Properties/>
</PropertySet>
<UniqueKey>
  <ForeignKeys/>
</UniqueKey>
<ForeignKey>
  <KeyAssociations/>
</ForeignKey>
<Document>
  <Documents/>
  <Notes/>
  <Extensions/>
</Document>
```

Report Lock Template

```
<Report>
  <ReportLocation/>
  <FileRefs/>
  <Documents/>
  <Notes/>
  <Extensions/>
  <Properties/>
  <PropertySets/>
</Report>
```

SASLibrary Lock Template

```
<SASLibrary>
  <UsingPackages/>
  <Properties/>
  <Documents/>
  <Notes/>
  <Extensions/>
</SASLibrary>
<DatabaseSchema>
  <Documents/>
  <Notes/>
  <Extensions/>
</DatabaseSchema>
<Document>
  <Documents/>
  <Notes/>
  <Extensions/>
```

```
</Document>
```

ServerComponent Lock Template

```
<ServerComponent>
   <Properties/>
   <ServiceTypes/>
   <SourceConnections/>
   <InitProcesses/>
   <DescriptiveComponent/>
</ServerComponent>
<Connection>
   <Properties/>
</Connection>
```

ServerContext Lock Template

```
<ServerContext>
   <UsingComponents/>
</ServerContext>
<LogicalServer>
   <UsingComponents/>
   <InitProcesses/>
</LogicalServer>
<ServerComponent>
   <Properties/>
   <ServiceTypes/>
   <SourceConnections/>
   <InitProcesses/>
   <DescriptiveComponent/>
</ServerComponent>
<Connection>
   <Properties/>
</Connection>
```

TextStore Lock Template

```
<TextStore>
  <Documents/>
  <Notes/>
  <Extensions/>
</TextStore>
<PropertySet>
  <Properties/>
</PropertySet>
<Document>
  <Documents/>
  <Notes/>
  <Extensions/>
</Document>
```

Tree Lock Template

```
<Tree>
  <Documents/>
  <Notes/>
  <Extensions/>
</Tree>
<Document>
  <Documents/>
  <Notes/>
  <Extensions/>
</Document>
```

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Clearing or Deleting a Repository

There may be times when you want to clear the contents of a repository, for example, to repopulate it with metadata. If many individual metadata objects have been deleted from a repository, you might also want to remove any structures that might have been left by the deleted objects in order to maintain performance efficiency. You might also have a need to destroy the repository itself.

You can do all of these things by using the DeleteMetadata method and setting one of the OMI_DELETE, OMI_PURGE, OMI_REINIT, or OMI_TRUNCATE flags.

- Set OMI_REINIT (2097152) to clear a repository if your wish is to repopulate it completely with different metadata.
- Set OMI_TRUNCATE (4194304) if your wish is to repopulate a repository using metadata object containers created for the previous repository.
- Set OMI_PURGE (1048576) to remove the remnants of any individually deleted metadata objects.
- Set OMI_DELETE (32) to destroy a repository. OMI_DELETE destroys the contents of a repository and also removes its registration from the repository manager.

These flags are issued in the REPOS namespace on the RepositoryBase type.

Note: A user must have Administrative User status in order to clear or delete a repository. For more information, see "Server Administrative Privileges" in the **SAS 9.1 Metadata Server: Setup Guide**.

Caution: You should not combine the REPOS namespace flags. Each one should be run exclusively. Combining them will potentially yield undesirable results.

Previous Page | Next Page | Top of Page

Unregistering and Re−registering a Repository

A repository is unregistered by issuing a DeleteMetadata method on its RepositoryBase object instance in the REPOS namespace and setting only the OMI_TRUSTED_CLIENT flag.

A repository is re−registered by issuing the AddMetadata method (omitting the repository's identifier from the Id= attribute) and specifying the repository directory in the Path attribute. The server reads the CNTAINER and MRRGSTRY files found at the path location and uses the repository identifier from them to create a new registration.

Note: A user must have Administrative User status in order to unregister and re−register a repository. For more information, see "Server Administrative Privileges" in the **SAS 9.1 Metadata Server: Setup Guide**.

Previous Page | Next Page | Top of Page

Model Usage Scenarios Summary

This section describes how to the use the SAS namespace metadata types to create metadata objects for the most commonly used application elements. It describes which metadata types to use to represent actual objects and the relationships between these objects. It also provides sample XML requests that show how to create the actual metadata objects.

The information is presented in a series of scenarios that describe how to use the metadata types in different situations. Note that the XML requests included represent only *one* way of creating the metadata, in order to demonstrate how to use the metadata types.

Usage Scenario: Creating Metadata Objects that Represent a DBMS
: describes the metadata types that represent DBMSs, database schemas, and the metadata types used to represent SAS software's access to DBMS information.

Usage Scenario: Creating a Prototype
: describes how to create a prototype.

Usage Scenario: Creating Metadata for a SAS Library
: describes the metadata types used to define a SAS library.

Usage Scenario: Creating Metadata for Tables, Columns, and Keys
: describes how to create metadata for tables, columns, and relational keys.

Usage Scenario: Creating Metadata for a SAS/SHARE Server
: describes how to use the metadata types that represent software deployment information, that is, information about how to run and access installed software and the resources (files and data) that the software can access.

Usage Scenario: Creating Metadata for a Stored Process
: describes how to define a stored process and how to define parameters for the stored process using PropertyGroups.

Usage Scenario: Creating Application Hierarchies Using Tree and Group Objects
: describes how to use the metadata types that represent an application hierarchy.

Usage Scenario: Creating Metadata for a Workspace Server
: describes how to define a SAS Integration Technologies workspace server using metadata types from the Software Deployment submodel. This scenario also includes information on creating a prototype for the server using the Property submodel.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Usage Scenario: Creating Metadata Objects that Represent a DBMS

Purpose

This usage scenario describes how to define a DBMS server. You must use metadata types from several submodels described in **SAS 9.1 Open Metadata Interface: Reference**, including the Software Deployment Submodel, Resource Submodel, and Relational Submodel. The example focuses on how to create the DBMS and define the server context of the SAS System that will be used to access the DBMS data.

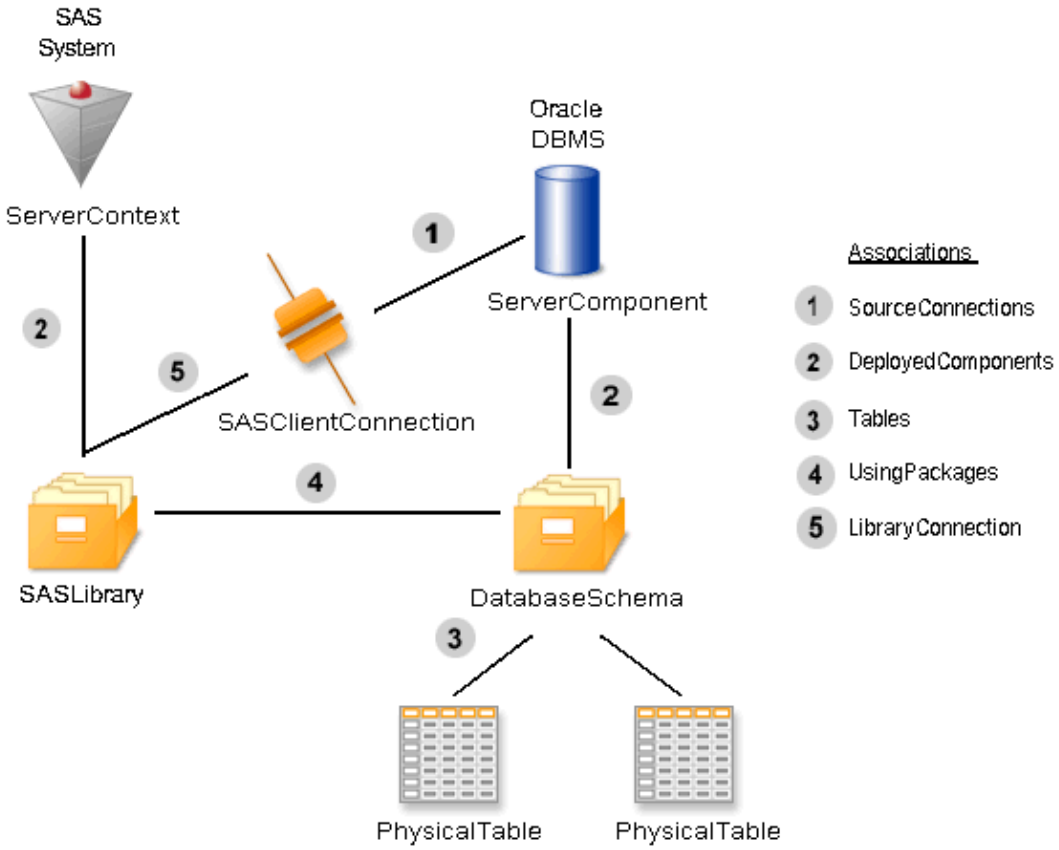
Requirements

This usage scenario assumes the reader has general understanding of the Software Deployment Submodel, Resource Submodel, and Relational Submodel of the SAS Metadata Model. Readers who are not familiar with the general concepts of the SAS Open Metadata Architecture should refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Creating Objects that Represent a DBMS

Figure 1 depicts the metadata objects that represent the information necessary to access tables in a DBMS from SAS software.

Figure 1. Metadata Objects that Represent the Information Necessary to Access DBMS Tables from SAS Software



The metadata type ServerComponent is used to represent installed, runnable software such as the Oracle DBMS represented in this scenario. A ServerContext is used to define an application context by grouping non−homogeneous servers together which each can access the same resources. Resources include file system directories, database schemas, and SAS libraries which are represented by metadata types subtyped from DeployedDataPackage. There is an association between a ServerComponent and/or ServerContext and the DeployedDataPackage objects that it can access. If the software is configured as a server, in this example the Oracle DBMS, the information used to access the server is contained in one of the Connection metadata types.

In the previous diagram, the Oracle DBMS is a type of ServerComponent. The association labeled "1" in the figure is called SourceConnections from the ServerComponent's perspective and Source from the Connection's perspective. The SourceConnections is the list of Connection objects that can be used by other software to access the server. There may be many Connection objects, for example, one that contains OLEDB connection information, JDBC connection information, or SAS software connection information. There are several attributes which describe the protocols supported by the Connection object. The CommunicationProtocol attribute contains a value such as APPC or TCPIP that describes the communication protocol used to access the component. The ApplicationProtocol contains a value such as HTTP, RMI, Bridge, or SHARE. Each Connection object has a Source association with one and only one ServerComponent object.

Each ServerContext and ServerComponent may also have a list of data packages it can access. This association is labeled "2" in the figure. The Oracle DBMS has an association to its DatabaseSchema object.

The SAS software has an association to its SASLibrary object. The list of packages available to a ServerComponent is the association named DataPackages. From the package's perspective, the list of ServerComponent objects that can access it are in its ServerComponents association.

A SASLibrary may be associated with a single DatabaseSchema. This association is labeled "4" in the figure. If a SASLibrary is associated with a DatabaseSchema, it also needs the connection information for the server, and that association is labeled "5" in the figure. From the SASLibrary object's perspective, this association is called LibraryConnection. From the Connection object's perspective, the association is called Libraries.

There is a subtype of DeployedDataPackage called RelationalSchema that has an association to DataTable objects. SASLibrary and DatabaseSchema are subtypes of RelationalSchema. The association labeled "3" in the figure is inherited from RelationalSchema and is called Tables from the RelationalSchema perspective and TablePackage from the DataTable perspective. PhysicalTable inherits this association from DataTable. This example uses PhysicalTable objects because the objects represent actual tables residing in a DBMS or file system. PhysicalTable objects should be associated with a DatabaseSchema, if they reside in a DBMS, or a SASLibrary, if they are SAS data sets.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Usage Scenario: Creating a Prototype

Purpose

This usage scenario describes how to create a prototype using the Property Submodel. The example focuses on creating a SAS/CONNECT connection prototype. An XML representation of the prototype is included to show how to construct prototype objects and their properties.

Requirements

This usage scenario assumes the reader has a general understanding of the Property Submodel and Software Deployment Submodel of the SAS Metadata Model. Readers who are not familiar with the general concepts of SAS Open Metadata Architecture should refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Description of a Prototype

A prototype is a template used for creating other metadata objects or sets of metadata objects that represent a concept. The prototype includes all possible valid options or properties that may be used to describe the metadata object that is represented by the prototype. The prototype may then be used to drive a user interface that will aid a user in the creation of metadata.

Prototype Objects

A Prototype object is used to as a template for creating other metadata objects. This example describes a prototype for a SAS/CONNECT connection. A Prototype object contains an attribute that contains the metadata type described by the prototype, in this case a SASClientConnection, and has associations to other objects that define the attributes and associations for the object.

Prototype and Properties

In this example, the prototype describes the information needed for a SAS/CONNECT connection. Its attribute, MetadataType, identifies which type is being described. Prototype objects may have AttributeProperty objects that describe the settings of the attributes of the templated metadata type. In this example, the AttributeProperty for ClassIdentifier indicates the only valid setting for this attribute for a Workspace Server.

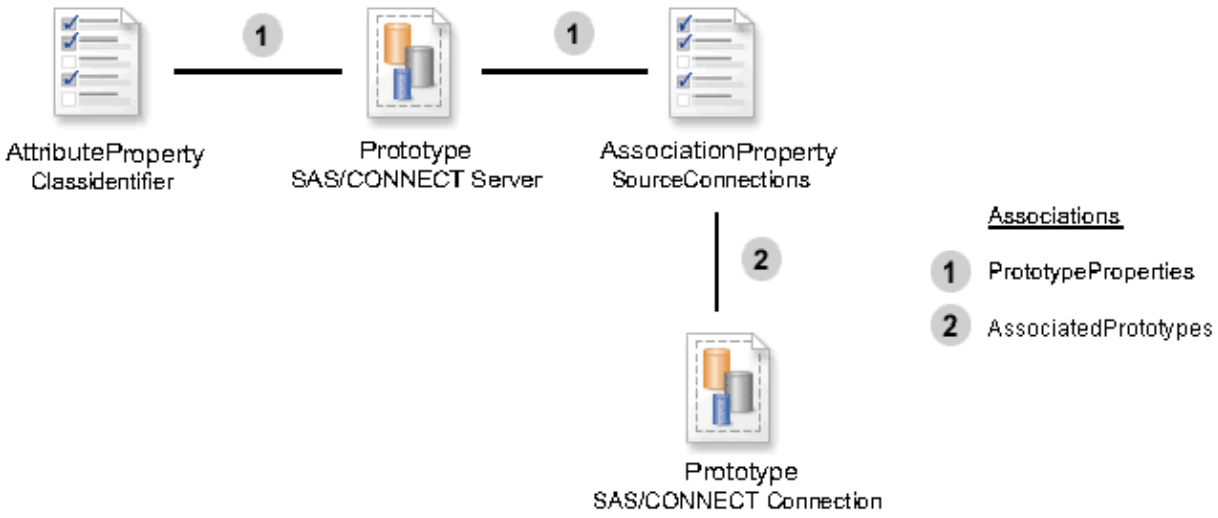
Prototype objects may also have AssociationProperty objects that describe the characteristics of associated objects. In this case, the AssociationProperty connects a prototype of a TCPIPConnection object to the prototype of a SASClientConnection.

Prototype objects locate AttributeProperty and AssociationProperty objects through the PrototypeProperties association or through PropertyGroup objects.

Objects created by using a prototype definition may maintain an association to the prototype that was used to create them through the UsingPrototype association.

Figure 1 depicts a prototype for a SASClientConnection, an AttributeProperty, and an AssociationProperty. The associations indicated by the number "1" in the figure represent the association called PrototypeProperties. A Prototype object can have a PrototypeProperties association to AttributeProperty and AssociationProperty objects. That is, a SAS/CONNECT server can have both a class identifier and source connections defined as properties. The association from the AssociationProperty object to the SAS/CONNECT Connection Prototype (numbered "2" in the figure) is called AssociatedPrototypes and is used to link prototype definitions together.

Figure 1.Metadata Objects that Represent a Prototype for a SASClientConnection



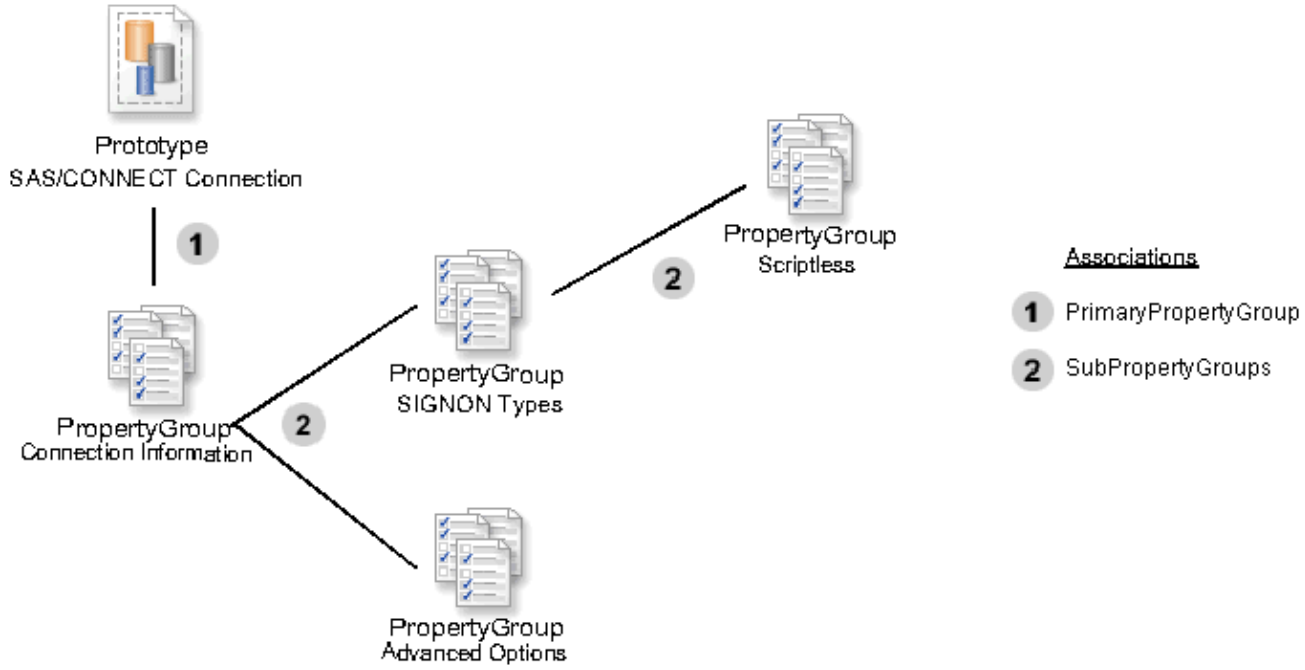
Prototype and Property

The TCPIPConnection that describes a Bridge connection has Property objects that describe the encryption level and algorithms. A prototype also uses Property objects to contain that information, but the Properties and PropertySet associations are not used. Unfortunately, due to our subtyping structure, these associations are available to a Prototype object. However, these associations should never be used.

If a prototype requires Property objects, then it should have a top–level PropertyGroup object associated to it through the PrimaryPropertyGroup association. PropertyGroup objects are used to logically group AttributeProperty, AssociationProperty, and Property objects. This grouping is primarily used to organize these objects for a user interface.

Figure 2 depicts how to group properties. Each prototype has a single top–level PropertyGroup, in this case named Connection Information. Each PropertyGroup may have subgroups. The association between the Prototype and its top–level group is called a PrimaryPropertyGroup association (number "1" in Figure 2). The association between the top–level PropertyGroup and its subgroups (numbered "2" in Figure 2) is called SubpropertyGroups.

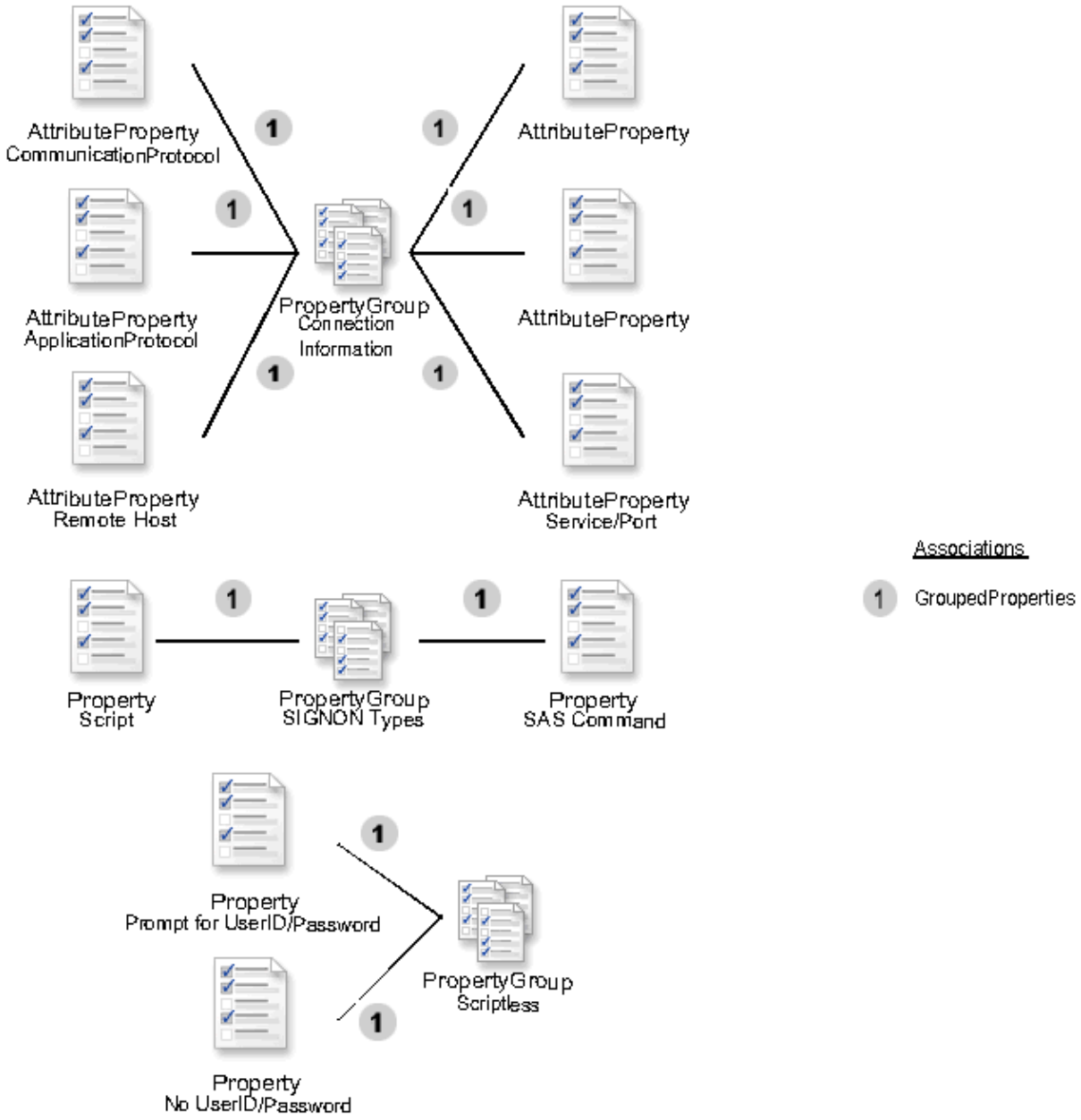
Figure 2. Metadata Objects for Grouping Properties



PropertyGroups and Properties

Figure 3 depicts the relationship between the subtypes of AbstractProperty objects and PropertyGroups. When used as part of a prototype definition, a Property object must be associated to a only one PropertyGroup. AttributeProperty and AssociationProperty are associated using the GroupedProperties association (numbered "1" in Figure 3).

Figure 3. Relationship between the Subtypes of AbstractProperty Objects and PropertyGroup Objects



XML Example

See Example of Creating a Prototype to view the XML representation of the prototype for a SAS/CONNECT server. This is the actual prototype which would be installed by SAS Management Console when initializing a new repository. This prototype is used by the server wizard to aid a user in creating a metadata definition for a SAS/CONNECT server.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Usage Scenario: Creating Metadata for a SAS Library

Purpose

This usage scenario describes the metadata objects used to define a SAS library. An XML representation of the metadata objects is provided.

Requirements

This usage scenario assumes the reader has a general understanding of the Resource Submodel, Software Deployment Submodel, and Relational Submodel of the SAS Metadata Model, which are described in **SAS 9.1 Open Metadata Interface: Reference**. Readers who are not familiar with the general concepts of SAS Open Metadata Architecture should also refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Defining a Library

The primary metadata type used to describe a SAS library is the SASLibrary type. It has attributes for the engine type, the libref name, and whether the library is already pre−assigned in the SAS session. A library has the IsPreassigned flag set to true if the library is assigned during initialization of the SAS session. SASLibrary is a subtype of RelationalSchema, because the SAS library concept acts much like a schema in that it is a grouping of tables (or data sets). RelationalSchema is a subtype of DeployedDataPackage because it contains data items.

In most situations, a SASLibrary will be associated with another DeployedDataPackage through the UsingPackages association to describe the physical location of the tables. This associated DeployedDataPackage could be a Directory, DatabaseSchema, or another SASLibrary object. The Directory object contains a DirectoryName attribute which is the name of the directory in the file system. If the IsRelative attribute is set to false (0), then it is assumed that the file system name is complete. If it is set to true (1), then there should be associated directory objects. The DirectoryName of these objects should be pre−pended to the DirectoryName of the base Directory object. When creating multiple levels of relative directory names, the relative names must be stored in such a way that simply pre−pending the relative name will create a valid directory name. This includes any required delimiters.

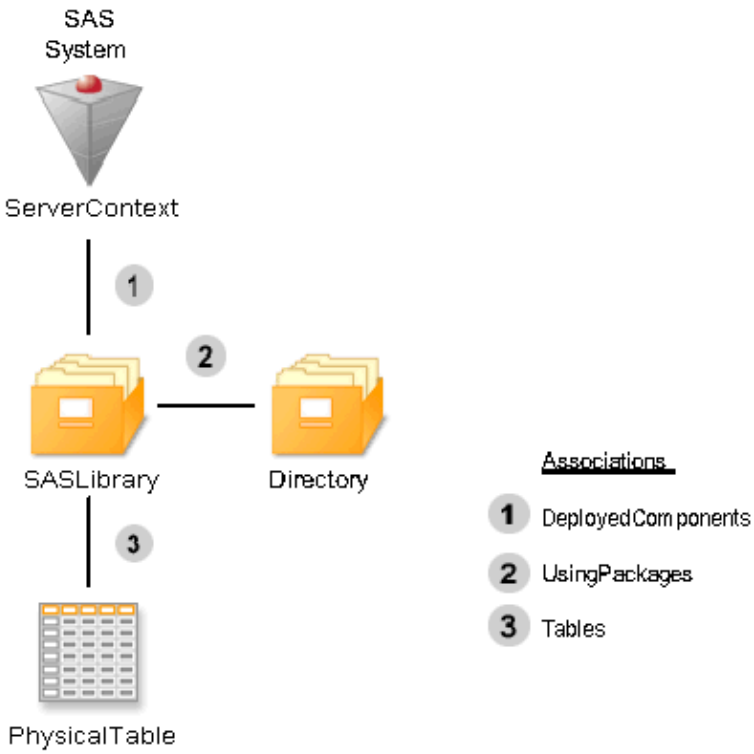
Tables may be associated to one and only one RelationalSchema. If the tables reside in a DMBS, they should be associated to a DBMS schema, which in turn is associated to a SASLibrary. If the tables reside in a directory, then they should be associated directly to a SASLibrary. If there are multiple ways of specifying the directory, such as c:\mystuff and \\mypc\mystuff, then the Alias association should be used. Each of these scenarios is described in the examples below.

Examples of SAS Library Objects

Library with Tables in a Directory

Figure 1 shows the metadata objects used to describe a SAS library, directory, and data sets. The library is represented with the SASLibrary object, a Directory object describes the physical location of the data, and the data set is a PhysicalTable object. The ServerContext object describes which SAS software systems can use this library definition.

Figure 1. Metadata Objects that Define a SAS Library, Directory, and Data Sets



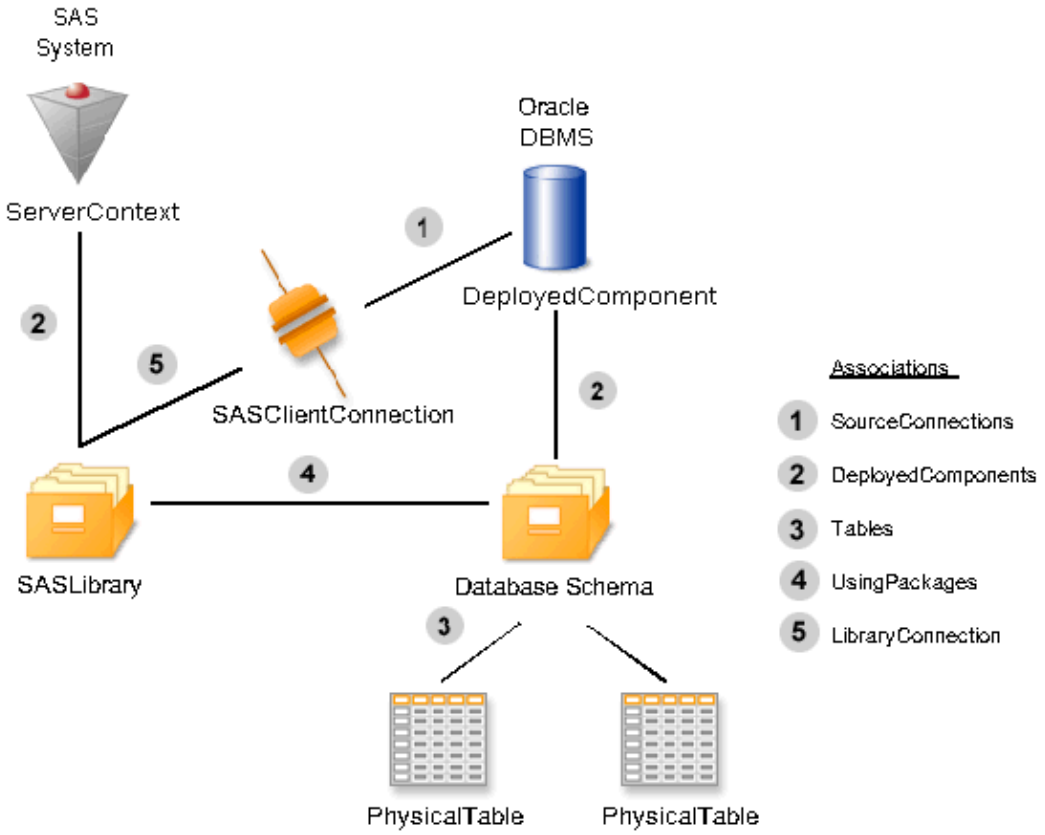
The SAS software that can use this library definition is associated to the SASLibrary through the DeployedComponents association (numbered "1" in the figure). The Directory is associated to the SASLibrary through the UsingPackages association (numbered "2" in the figure). This association is used to identify the "physical location" which can be represented by a Directory object, DatabaseSchema object, or in the case of concatenated libraries, a SASLibrary object. The DirectoryName attribute of Directory contains the pathname of the directory. This attribute should always end with the proper directory delimiter, in this case a "/". The PhysicalTables are associated to the SASLibrary through the Tables association (numbered "3" in the figure). If a library has tables associated to it, then it is the primary or owning library.

See XML Representation of Metadata Objects that Define a SAS Library, Directory, and Data Sets to view the XML representation of the metadata objects shown in Figure 1.

Library with Tables in a Foreign Database

Figure 2 shows the metadata objects used to describe a SAS library, directory and data sets. The library is represented by a SASLibrary object, the schema is represented by a DatabaseSchema object, and the relational table is represented by a PhysicalTable object. The ServerContext object describes which SAS software systems can use this library definition. The DeployedComponent is used to represent the database server where the schema and tables reside. The SASClientConnection defines the connection used by the SAS application to access the data in the foreign database.

Figure 2. Metadata Objects Used to Define a SAS Library with Tables in a Foreign Database



The database is a ServerComponent which has connection information used by SAS software in the SASClientConnection object. From the ServerComponent for the DBMS, the connection information is found through the SourceConnections association (number "1" in Figure 2). The DatabaseSchema is an owning package (it has tables associated to it through the Tables association (numbered "3" in Figure 2) and is associated to the ServerComponent through the DeployedComponents association (numbered "2" in Figure 2). The SASLibrary is associated to the DatabaseSchema which it can access through the UsingPackages association (number "4" in Figure 2) and to the SAS Software which used this library using the DeployedComponents association (number "2" in Figure 2). SASLibrary uses the LibraryConnection to get the connection information for the library (number "5" in Figure 2).

See XML Representation of Metadata Objects Used to Define a SAS Library with Tables in a Foreign Database to view the XML representation of the metadata objects shown in Figure 2.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Usage Scenario: Creating Metadata for Tables, Columns, and Keys

Purpose

This usage scenario describes how to create PhysicalTable, Column, and key objects.

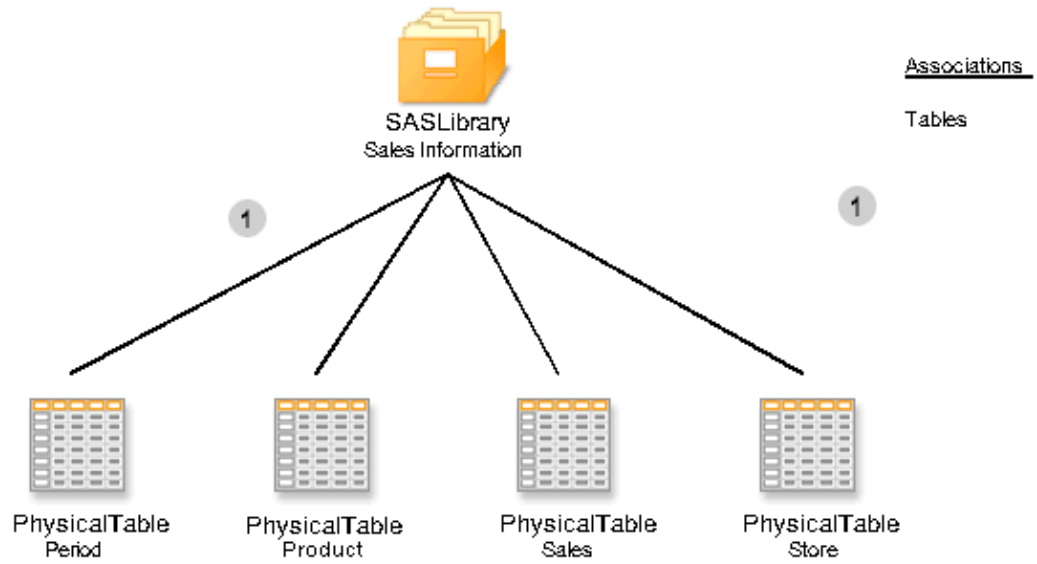
Requirements

This usage scenario assumes that the reader has general understanding of the Resource Submodel and Relational Submodel of the SAS Metadata Model, which are described in **SAS 9.1 Open Metadata Interface: Reference**. Readers who are not familiar with the general concepts of the SAS Open Metadata Architecture should refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Creating a SASLibrary and PhysicalTable Objects

Figure 1 depicts the metadata objects that represent the SAS library Sales Information and its four tables: PERIOD, PRODUCT, SALES, and STORE.

Figure 1. Metadata Objects that Represent SAS Library "Sales Information" and its four tables: "Period", "Product", "Sales", and "Store"



The metadata type SASLibrary is used to represent SAS libraries. SASLibrary is a subtype of RelationalSchema, which is a subtype of DeployedDataPackage, and inherits all of the attributes and associations of its supertype.

There are several metadata types that represent tables. PhysicalTable is the metadata type used to represent a relational table or data set that physically resides in a file system or DBMS. RelationalTable is used to represent tables that do not have a physical mapping, for example, a table created using a data modeling tool.

WorkTable is used to represent a transient table; that is, a table that resides in the SAS WORK library.

These objects can be created using the following XML request. See XML Representation of SASLibrary and PhysicalTable Objects to view the XML representation of the metadata objects shown in Figure 1.

In the sample XML request, the first element is "AddMetadata", which identifies the method that is invoked on the SAS Metadata server. The next element is "Metadata", which indicates that the elements that follow are metadata objects.

The element SASLibrary defines the SAS library and has the attributes Name, Libref, and Engine. Also included is the IsPreassigned attribute, which should be set to "true" if the library will be pre−assigned during initialization of the associated DeployedComponent. A metadata object of type SASLibrary will be created with those attributes set to the values provided in the document.

All of the immediate subelements of a metadata type are names of associations. The subelements of SASLibrary define the associations between SASLibrary and other metadata objects. In this example, Tables is a subelement of SASLibrary and defines a relationship with a set of tables. This association is numbered "1" Figure 1. The association Tables is inherited from SASLibrary's supertype: RelationalSchema. The Tables association is defined between the metadata type RelationalSchema and the DataTable type.

Each end of the association is named. A RelationalSchema object refers to its associated objects by using the XML tag Tables. DataTable objects refer to their associated RelationalSchema objects by using the XML tag TablePackages.

Any subtype of RelationalSchema and DataTable can use the Tables/TablePackages association to create an association between the two objects. SASLibrary is a subtype of RelationalSchema, so it inherits this association. PhysicalTable is subtype of DataTable, so it also inherits this association.

If you refer back to the XML, there are four PhysicalTable objects defined inside the Tables element. All of the elements inside the Tables will be associated with the SASLibrary.

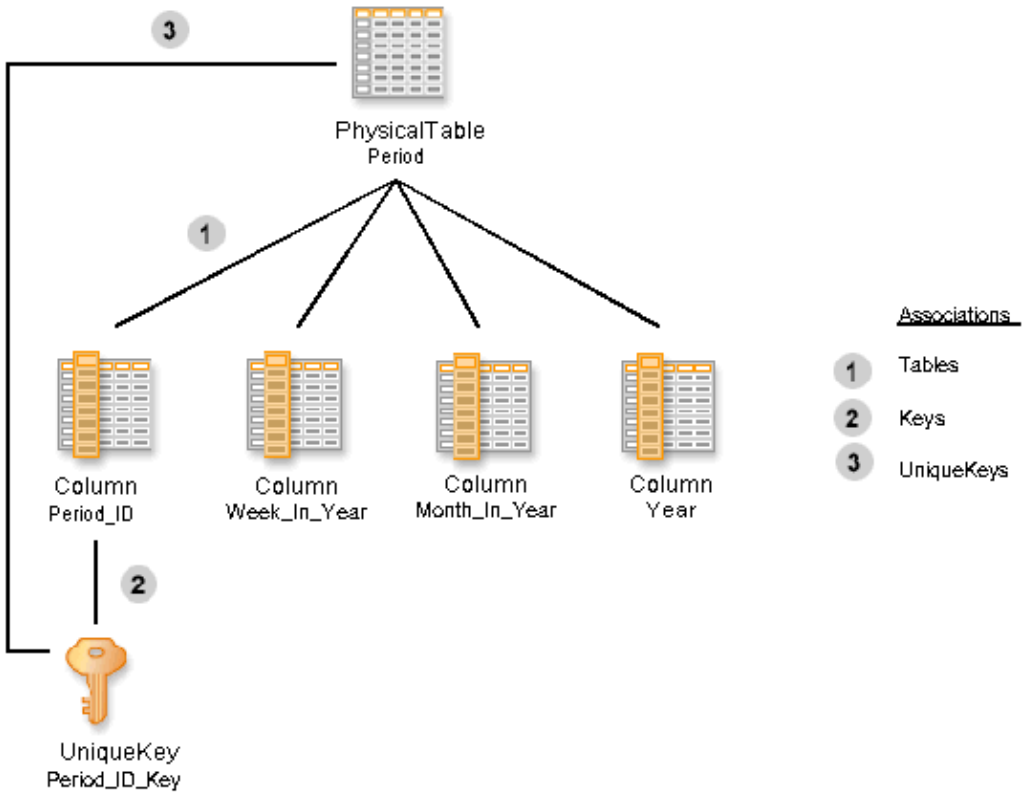
PhysicalTable has three name attributes: Name, SASTableName, and TableName. The Name attribute is a descriptive or display name. The SASTableName attribute is the name used by SAS software to refer to the table. The TableName attribute is only used if the table resides in a DBMS and the DBMS name is different than the name used by SAS. In this case, TableName is the DBMS name for the table.

When this XML document was sent to the server, it created five new metadata objects. The server allocated the following identifiers for the objects: for the SASLibrary, the ID is ABCDEFGH.A1000001; for the PERIOD table, the ID is ABCDEFGH.A2000001; for the SALES table, the ID is ABCDEFGH.A2000002; and so on.

Creating a Column Objects and Keys

Figure 2 depicts the columns and keys associated with the PERIOD table. The column named PERIOD_ID has a primary key associated with it.

Figure 2. Metadata Objects that Represent the Columns and Keys Associated with PhysicalTable "Period"



Column is the metadata type used to represent columns in a table. A unique key or a primary key is represented by the metadata type UniqueKey. The attribute IsPrimary defaults to 0, which means the key is not a primary key. This attribute should be set to 1 if the key is a primary key.

These objects can be created using the following XML request. See XML Representation of Column Objects for Period Table to view the XML representation of the metadata objects shown in Figure 2.

The Column object requires an association to a DataTable object or an object that is a subtype of DataTable. In this example, each Column element has a subelement Table, which is the name of the association to a DataTable. This association is identified with the number "1" in Figure 2. Because the metadata object for the table already exists, it is referred to by using the ObjRef attribute with the value of the identifier of the table. This syntax is explained fully in Adding Metadata Objects.

The first Column also has objects associated with it via the Keys association. Any type of key can be associated with a Column using the Keys association. In this example, the column named PERIOD_ID is associated with the unique key named PERIOD_ID_KEY. This association is identified with the number "2" in Figure 2.

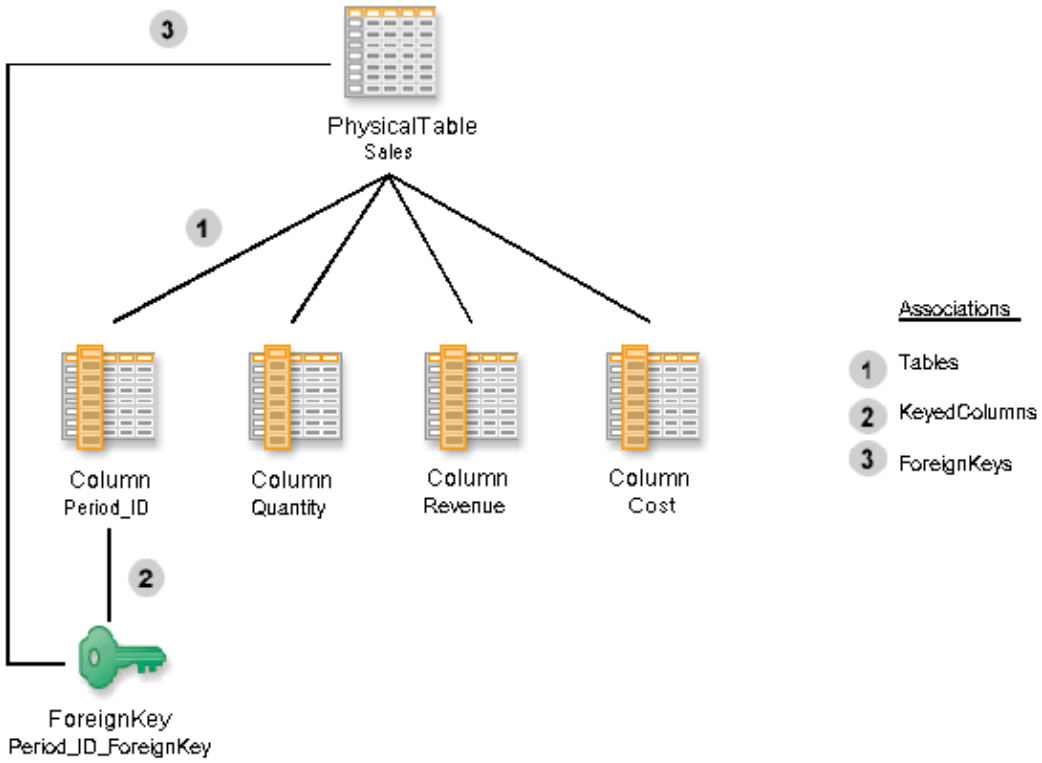
A key has a required association to both a table and a set of columns. The association between a UniqueKey and a DataTable from the UniqueKey's perspective is called Table. From the DataTable, it is called UniqueKeys. This association is identified by the number "3" in Figure 2. From the Column's perspective, the association is named Keys; from the Key's perspective, it is named KeyedColumns. In the XML request, the

UniqueKey element has the subelement Table followed by a reference to the PhysicalTable's identifier.

When this request was executed on the server, it created five new objects: four Column objects and one UniqueKey object. The PERIOD_ID column was given the identifier ABCDEFGH.A3000001, and the primary key was given the identifier ABCDEFGH.A4000001.

Figure 3 depicts the columns associated with the SALES table. The column named PERIOD_ID has a foreign key associated with it.

Figure 3. Metadata Objects that Represent the Columns and Keys Associated with PhysicalTable "Sales"



These Column objects can be created using the following XML request. See XML Representation of Column Objects for Sales Table to view the XML representation.

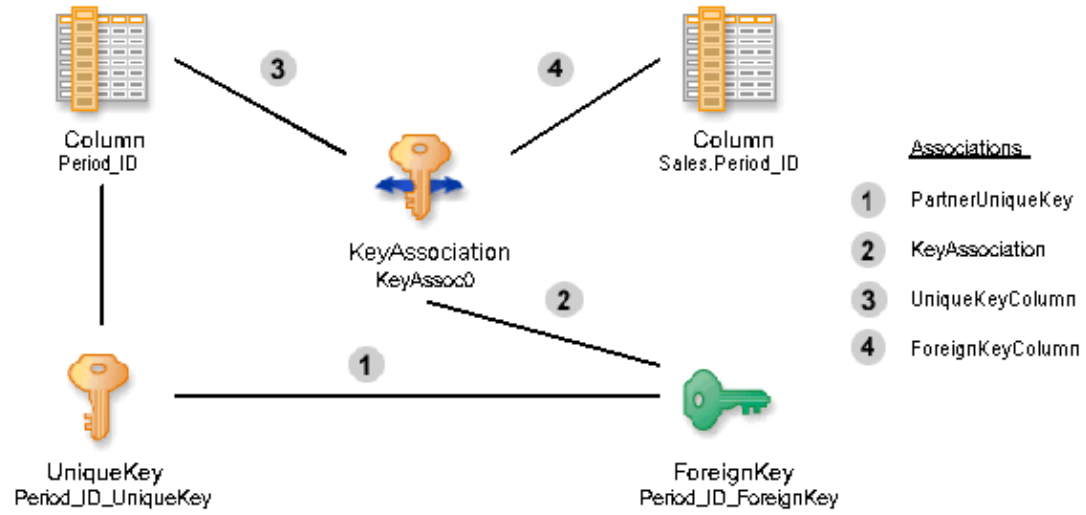
After this request is executed on the server, the Column objects are created, and the association numbered "1" in Figure 3 is created between the PhysicalTable object and the Column objects. PERIOD_ID column in the SALES table is given the identifier ABCDEFGH.A3000007.

The foreign key, its required associations, and additional objects are created using the following XML request. See XML Representation of Keys to view the XML representation.

The foreign key named PERIOD_ID_FOREIGN_KEY is associated with the PERIOD_ID column in the SALES table through the KeyedColumns association, which is numbered "2" in Figure 3. It is also associated with the SALES table through the Table association, which is numbered "3" in Figure 3.

ForeignKey objects have two additional required associations: they must have an association with a UniqueKey and a set of KeyAssociation objects. These associations are depicted in Figure 4.

Figure 4. Metadata Objects that are Associated with ForeignKey Objects



The ForeignKey object's association with the primary key in the PERIOD table is created through the PartnerUniqueKey association in the XML. This association is numbered "1" in 4.

All ForeignKey objects have a set of KeyAssociation objects, one for each column associated with the foreign key. This association numbered "2" in Figure 4.

The KeyAssociation object associates one column of the unique key with the corresponding column in the foreign key. From the KeyAssociation object, these associations are called the UniqueKeyColumn and the ForeignKeyColumn. These associations are identified by the numbers "3" and "4" in Figure 4. In this example, there is only one column for each key, so there is only one KeyAssociation object created. The KeyAssociation object has an association to one column in the UniqueKey and one column in the ForeignKey. This association is used to easily identify which columns correspond in a unique key/foreign key relationship. If the characteristics of one of the columns changes, it is easy to identify the associated columns and change their characteristics as well.

Previous Page | Next Page | Top of Page

Usage Scenario: Creating Metadata for a SAS/SHARE Server

Purpose

This scenario describes how to use the metadata types that represent software deployment information; that is, information about how to run and access installed software and the resources (files and data) that the software can access. An XML representation is included to show how to create software deployment objects and their associations.

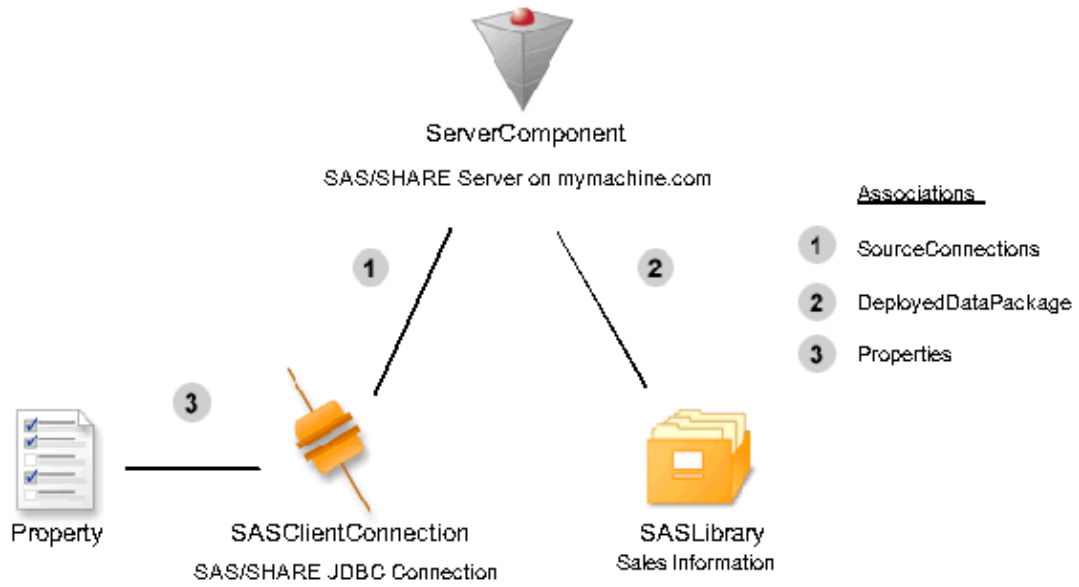
Requirements

This usage scenario assumes the reader has a general understanding of the Property Submodel and Software Deployment Submodel of the SAS Metadata Model, which are described in **SAS 9.1 Open Metadata Interface: Reference**. Readers who are not familiar with the general concepts of the SAS Open Metadata Architecture should refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Software Deployment Metadata Types

This scenario depicts metadata objects that represent a SAS/SHARE server, the information used by a JDBC driver to access the server, and the SAS library ("Sales Information"). Figure 1 shows these metadata objects and their associations.

Figure 1. Metadata Objects Representing a SAS/SHARE Server, Libraries, and Connections



The metadata type ServerComponent represents a configured instance of an installed and runnable deployed component. This metadata type contains attributes that describe the software, such as name, vendor, version,

and number. ServerComponent objects may be associated with a Transformation using the InitInfo association. The Transformation contains associations to other objects that provide startup information for the server. Startup information may include scripts, properties, login information for the startup process, log files, or any other information used at startup time. The DeployedDataPackage association identified by the number "2" in Figure 1 represents the data packages, such as directories, schemas or libraries, that are available to the server. In this scenario, the DataPackages association refers to a SASLibrary.

ServerComponent objects may also be associated with Connection objects using the SourceConnections association (identified by the number "1" in Figure 1). The connections may describe how other software components communicate with this object or how this component should communicate with other software components. The model has very few types to represent different connections. These types include: COMConnection, TCPIPConnection, and SASClientConnection. Each of these types contain a subset of attributes that are common for each type of connection. Associated Properties (identified by the number "3" in Figure 1) are used to provide additional information for specific connection types. In this example, we use a SASClientConnection to represent the SAS/SHARE JDBC connection. Additional properties that would be stored with this connection object include the server name, physical name, libref, and encryption algorithm key size. The SAS Management Console uses Prototypes (see also Usage Scenario: Creating a Prototype) to determine which properties are appropriate when creating connections.

Example

See XML Representation of Metadata Objects Representing a SAS/SHARE Server, Libraries, and Connections to view the XML representation of the metadata objects shown in Figure 1.

The ServerComponent, which represents the server, is the first object that is created. The subelement *SourceConnections* is the name of one of the associations from a ServerComponent to Connection objects. The Connection objects in the *SourceConnections* describe how to access this server (the ServerComponent). This association is numbered "1" in Figure 1.

Connection objects have a set of protocol attributes. For example, the information required for a DCOM connection is different from the information required for a CORBA connection. The *CommunicationProtocol* attribute of the Connection object describes the network protocol used to access the server, such as TCP or APPC. The *ApplicationProtocol* describes the application level protocol used by the application to communicate with the server. For example, the application protocol could be DCOM, CORBA, HTTP, RMI, Telnet, or in this case, the proprietary protocol, "SHARE". For TCPIP communication, the RemoteAddress is required, in addition to the port or service used by the server. For this type of connection, there is more information than is needed. This information is stored in PropertyObjects and associated with the Properties association (numbered "3" in Figure 1). The Property objects each must have a PropertyType which is associated using the OwningType association. This association is not pictured, but is represented in the XML request. Note that three of the properties expect the value of the property to be set as a string. Each of these properties point to a Property object already defined in the metadata for a string. The encryption algorithm key size property has an enumerated list of possible values. This enumeration is a TextStore object that is associated using the StoredConfiguration association on the PropertyType.

The metadata type SASClientConnection is a subtype of the metadata type Connection and contains information about how to access a SAS/SHARE server from a Java or other open client. For example, the SAS/SHARE driver for JDBC could use the information in this object to create the URL used to connect to the server. In order to find available SAS/SHARE servers, the JDBC application could search the repository for SASClientConnection objects that have the attribute *ApplicationProtocol* value set to "Share". These would provide connection information for all of the SAS/SHARE servers registered in the repository.

A ServerComponent can also have a set of Connection objects that are available through the *ProviderConnections*. These are the Connection objects used by this component to access other components. If the JDBC application was registered in the repository as a ServerComponent, its *ProviderConnections* could contain the *TCPIPConnection* objects it uses to access servers.

A ServerComponent could have Connection objects in both lists. For example, server one may forward a request to server two. Server one would have a connection object in *SourceConnections* that describes how to connect to it (Server one). Server one's *ProviderConnections* would contain the Connection object that describes how to connect to Server Two.

The other association to ServerComponent in this example is DataPackages (numbered "2" in Figure 1). The DataPackages identify data packages, which are SAS libraries, file system directories, DBMS catalogs, or schemas, that can be accessed by this ServerComponent. For SAS software, the DataPackages is the list of SASLibrary objects that are accessible. In this XML example, the SASLibrary object has already been created, so the XML contains a reference to the existing object using the ObjRef attribute.

Previous Page | Next Page | Top of Page

Usage Scenario: Creating Metadata for a Stored Process

Purpose

This usage scenario describes how to define a stored process using several submodels, including Software Deployment Submodel, Transform Submodel, and Property Submodel. It explains how to define parameters for the stored process using PropertyGroups. An example XML request is included that represents one way to create the metadata for this usage scenario.

The scenario depicts one usage pattern for the model, which integrates several submodels. It is not recommended that developers directly create or consume stored process metadata. Instead stored processes should be defined using the Stored Process Manager plug−in for SAS Management Console.

Requirements

This usage scenario assumes the reader has a general understanding of the Software Deployment Submodel, Transform Submodel, and Property Submodel of the SAS Metadata Model, which is described in **SAS 9.1 Open Metadata Interface: Reference**. Readers who are not familiar with the general concepts of the SAS Open Metadata Architecture should refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Description of a Stored Process

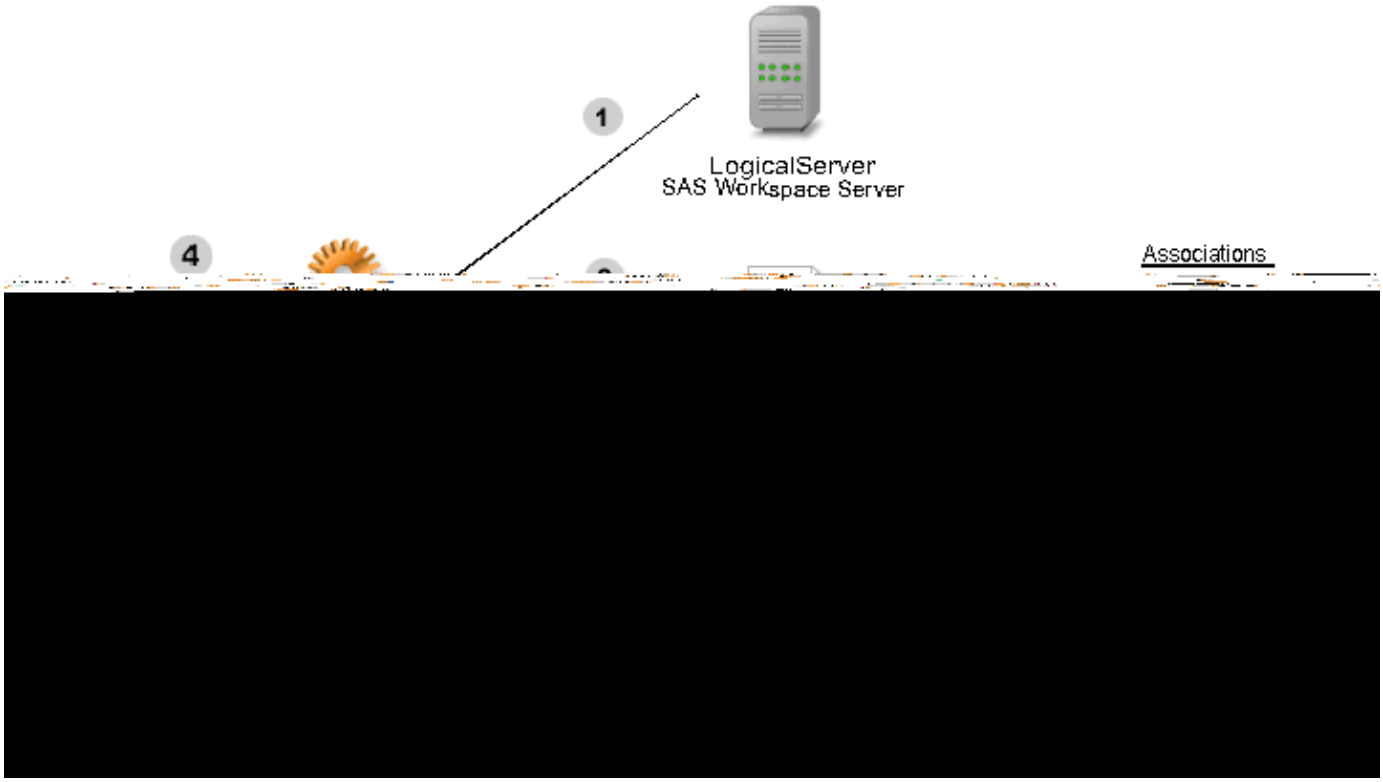
A stored process is a set of metadata objects that define the information necessary to run a process. This includes information about where the process can be run, the code that should be run, parameters that need to be supplied by a user or another process, the physical inputs to the process, and the outputs of the process.

How to Define A Stored Process

In this scenario, we define the Product Sales Demo. Central to defining a stored process is the ClassifierMap type. The ClassifierMap is a subtype of a transformation and is used to show the flow of data through a process. The ClassifierMap has properties that an application needs to determine what the process is and how to run it. The ClassifierMap is differentiated from other transformation types by the ClassifierSource and ClassifierTarget associations. A Classifier and its subtypes represent data in the model and may have features. For example, a PhysicalTable is a type of Classifier which has features called Columns. The process which represents the Product Sales demo consists of stored SAS code, which runs against a SAS data set to produce a sales report. The SAS code is associated to the ClassifierMap via the SourceCode association (numbered "4" in Figure 1). Source code may be any type of text object, including a TextStore (used for code stored in the repository), Document (a URI to an external document), SASCatalogEntry, ArchiveFile (Zip file), ArchiveEntry (an entry in the Zip file), or a File. In this scenario, the source code is stored in a File named PrdSale.sas. The File has a Directories association (numbered "5" in Figure 1) to the directory where the file is located. In addition, an application may choose to store Notes about the process and an external identifier for the process. Note text is stored in a TextStore object and associated to the ClassifierMap via a Notes association (numbered "2" in Figure 1). The external identifier is stored in an ExternalIdentity object and associated with the ClassifierMap via an ExternalIdentities association (numbered "3" in Figurer 1). An ExternalIdentity is an identifier that is assigned by an external source, a GUID in this scenario, but may be any string that uniquely identifies this process to the application.

The application also needs to know where the process can run, which LogicalServer to use, and how to access it. The ComputeLocations association (numbered "1" in Figure 1) has a list of the LogicalServer(s) that can run the process. In this scenario, the LogicalServer is a SAS Workspace Server, because SAS is necessary to run the SAS code.

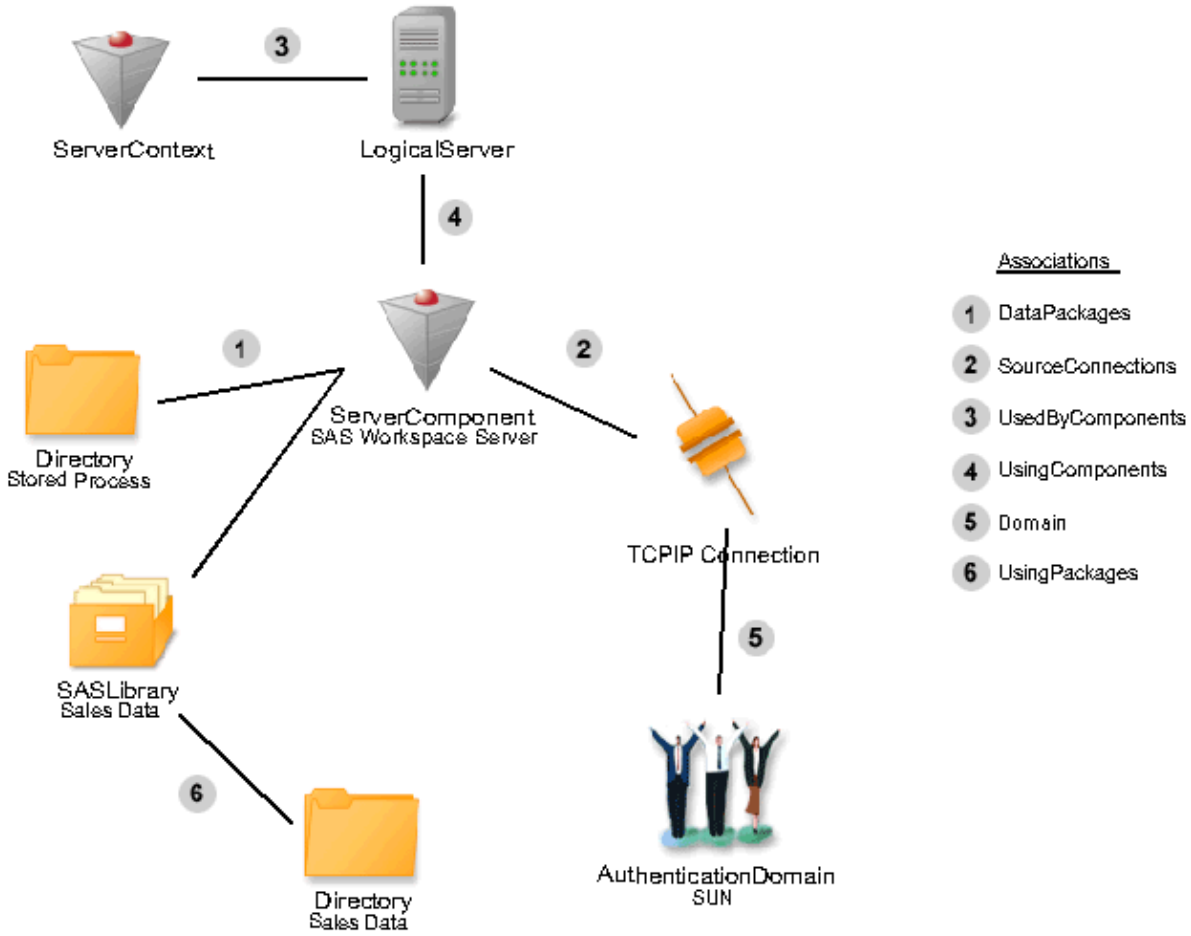
Figure 1. Metadata Objects Used to Define Stored Process for the Product Sales Demo



The SAS Workspace Server is defined, in part, using the LogicalServer type (Figure 2). (Also see Usage Scenario: Creating Metadata for a Workspace Server.) Once we know which component to use, we may need additional information about how to access this component. The LogicalServer has a ServerComponent associated using the UsingComponents association (number "4" in Figure 2). The ServerComponent has a SourceConnections association (numbered "2" in Figure 2) to a TCPIPConnection. The TCPIPConnection gives the information necessary for the process to connect to the Workspace Server and has information about the AuthenticationDomain for this server. The TCPIPConnection object is associated to the AuthenticationDomain via a Domain association (number "5" in Figure 2).

In addition, the Workspace Server LogicalServer object has a UsedByComponents association (number "3" in Figure 2) to a ServerContext object, which has associations to the DataPackages that are known to this server. In this example, two objects have a DataPackages association to the Workspace Server: a Directory and a SASLibrary (numbered "1" in Figure 2). When a SASLibrary is defined, often the directory information will also be stored. This occurs via the UsingComponents association between the ServerComponent and the LogicalServer (number 4 in Figure 2). The SAS Management Console supports the creation of library definitions. Refer to Usage Scenario: Creating Metadata for a SAS Library for more information.

Figure 2. Metadata Objects for Defining a SAS Workspace Server



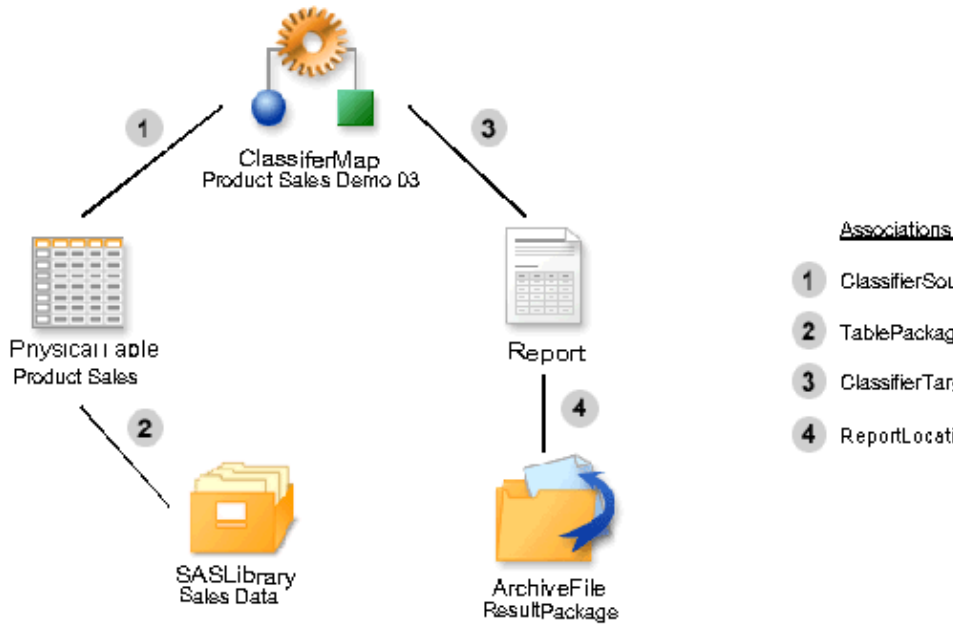
We now know where the code is for this process and where we can run this process. We now need to know something about the inputs (ClassifierSources) and outputs (ClassifierTargets). Note that only types which inherit from the Classifier type may be part of this association. Generally, types which are considered classifiers represent objects that have structure. Some examples of classifiers are data tables, reports, and cubes. The input and/or output information may or may not be embedded in the source code. However, even if it is part of the source code, this should be documented as part of this process to facilitate impact analysis. If a source is changed or moved, it is helpful to know all the processes that may be affected by the change.

In this scenario, a SAS data set is the only source and is associated to the ClassifierMap via the ClassifierSources association (number "1" in Figure 3). A SAS data set is represented by the PhysicalTable type and has an TablePackages association that helps to locate the dataset. In this scenario, it is associated to a SASLibrary (number "2" in Figure 3). Note that this library is a library that is known to the SAS Workspace Server where the process is run.

The output, or ClassifierTarget, for this scenario is a Report, illustrated by the association between the ClassifierMap and Report objects (number "3" in Figure 3). The Report is stored as an Archive file in this example. An ArchiveFile contains all the elements that are used by the report that was created. The Report could be output to any ContentLocation, including a Stream (stream of data), Email, File, or TextStore

(represents text stored in the metadata repository). For a complete list of ContentLocation types refer to the documentation for the ContentLocation metadata type in SAS Namespace Types in the **SAS 9.1 Open Metadata Interface: Reference**.

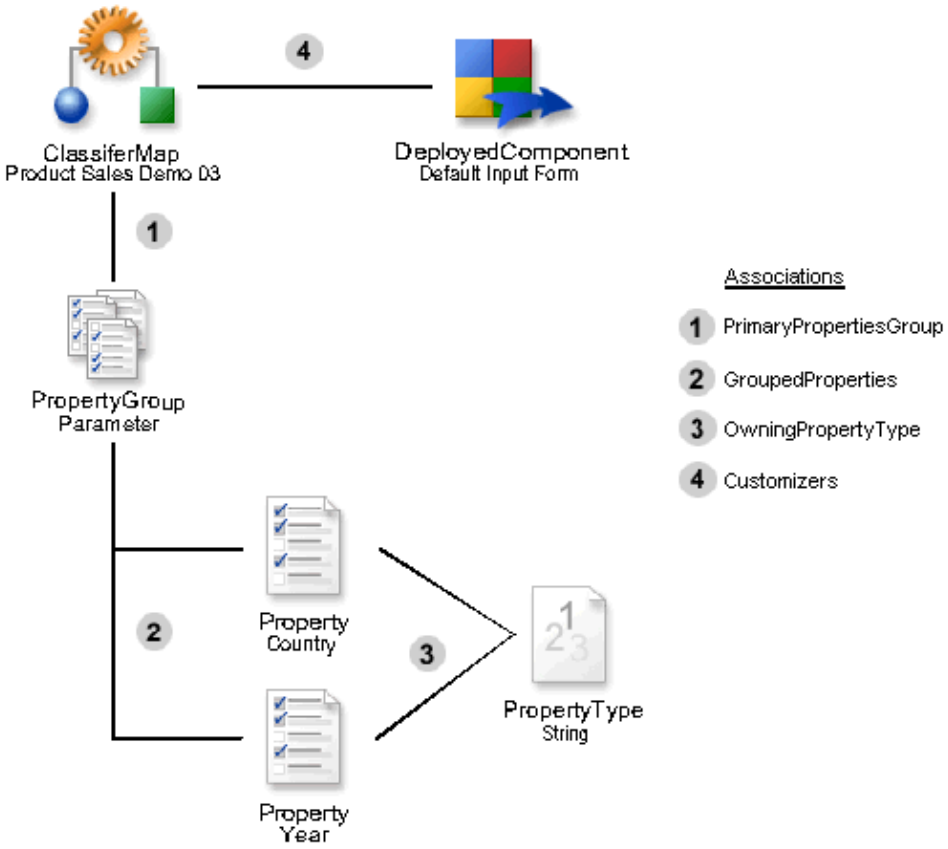
Figure 3. Metadata Objects for Creating a Report



This stored process requires a user to submit two parameters. These parameters are defined using Property objects in a PropertyGroup (Figure 4). The PropertyGroup is associated to the ClassifierMap using the PrimaryPropertiesGroup association (number "1" in Figure 4) and to Properties using the GroupedProperties association (number "2" in Figure 4). Each Property object represents one of the parameters necessary for the stored process to run. The Property objects also require an OwningPropertyType association to a PropertyType object (number "3" in Figure 4). The PropertyType gives information about the expected format of the property value. In this case, it is a simple string, but it could represent a more complex type like arrays or may have associated enumeration or local information. The associated configuration information would be stored in a Text object.

We now know what the parameters are but need to have a way for a user to enter values for the parameters. The ClassifierMap has a Customizers association (number "4" in Figure 4) to a DeployedComponent. The DeployedComponent in this scenario is a .jsp that provides a user interface for input of the parameter values.

Figure 4. Metadata Objects that Store Properties for a Stored Process



See XML Representation of Metadata Objects for a Stored Process to view the XML representation.

Previous Page | Next Page | Top of Page

Usage Scenario: Creating Application Hierarchies Using Tree and Group Objects

Purpose

This usage scenario describes how use the metadata types that represent an application hierarchy. The application hierarchy is used to group metadata in a meaningful way for an application.

Requirements

This usage scenario assumes the reader has a general understanding of the Grouping Submodel and Software Deployment Submodel of the SAS Metadata Model, which are described in the **SAS 9.1 Open Metadata Interface: Reference**. Readers who are not familiar with the general concepts of SAS Open Metadata Architecture should refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Trees and Groups

The SAS Metadata Model provides the following metadata types for grouping metadata objects together:

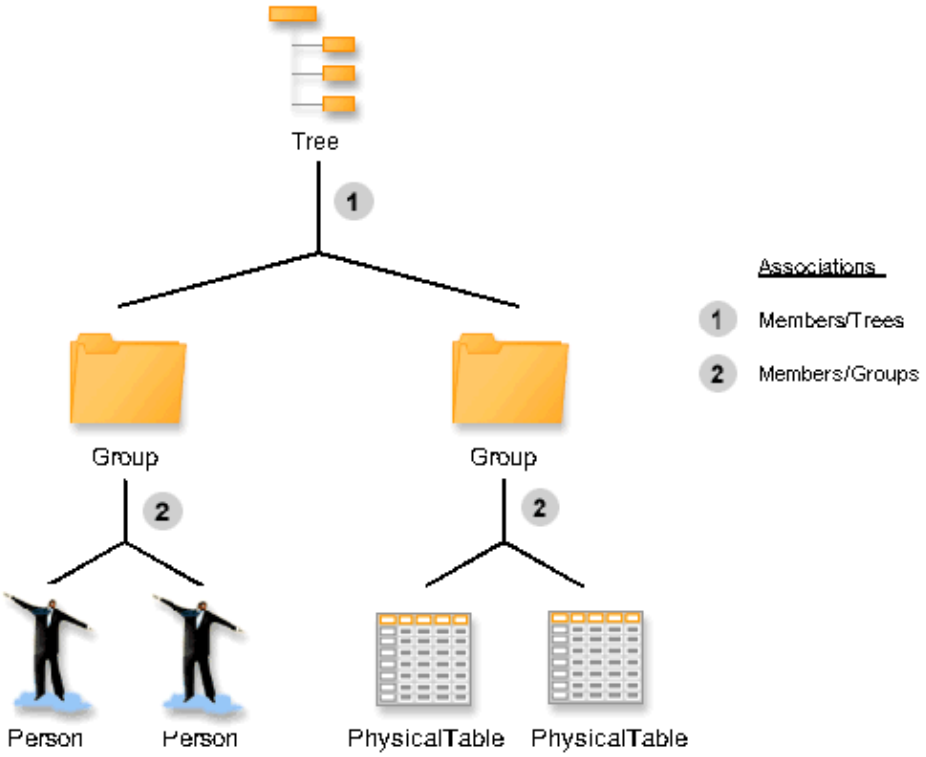
- Tree – used to create a tree hierarchy.
- Group – used to group objects together.

These metadata types have an association called Members, which can be associated with any other object. Usually, a Tree metadata object is created that contains references to other Tree objects as well as Groups. All of these objects are associated to the Tree through the Members association. To create the tree hierarchy, the trees are nested. A ParentTree has associations to its SubTrees. In addition, a tree object may be associated with a SoftwareComponent. This association gives the tree context via an application. The hierarchy of the tree is traversed through the ParentTree and SubTrees associations.

Tree, Group, and Members

The scenario depicted in Figure 1 is a single Tree with two Groups, each of which contain two objects. There is a Group that contains Person objects and a Group that contains PhysicalTable objects.

Figure 1. Metadata Objects Used to Represent a Single Tree with Two Groups



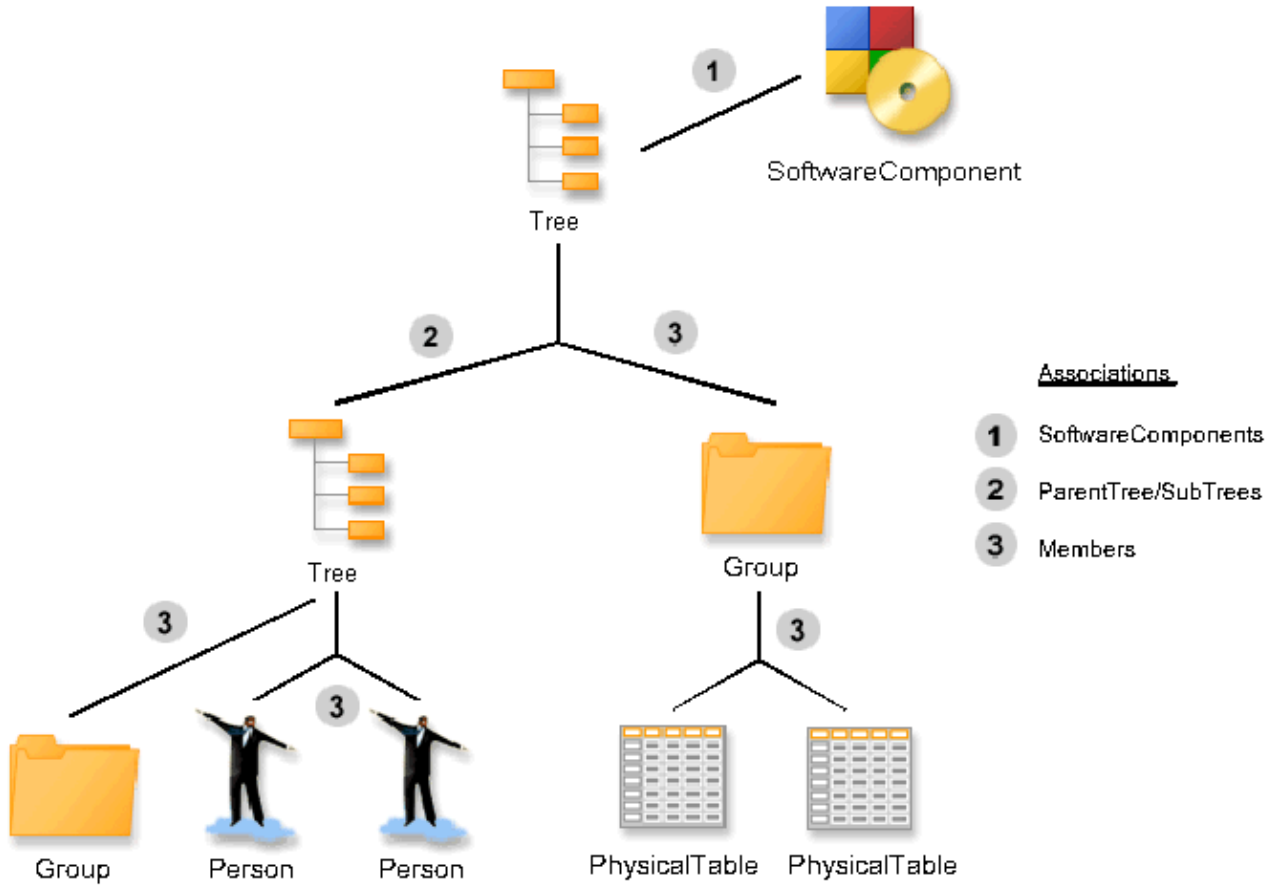
The following XML request shows how to create the Tree and Group objects. In this example, the Person and PhysicalTable objects are already created in the repository. See XML Representation of a Single Tree and Two Group Objects to view the XML representation.

The Tree object named All my metadata" has two Group objects in its Members association. The Tree could have other objects in this list, but this simple example only has Group objects. Each Group has a reference to the tree it is a part of, the reference being a symbolic reference to the Tree. The association is represented by the number "1" in Figure 1. Each Group also has a Members association to two other objects. These associations are represented by the number "2" in Figure 1.

Tree Hierarchy, Groups, Members, SoftwareComponent

The scenario in Figure 2 depicts a Tree hierarchy in which two trees are nested. Each Tree has a Group and/or other members. There is also an association from one Tree to a SoftwareComponent.

Figure 2. Metadata Objects that Represent a Tree Hierarchy



The following XML request creates the Tree and Group objects. In this example, the Person and PhysicalTable objects are already created in the repository. See XML Representation of a Tree Hierarchy to view the XML representation.

The Tree has several Members associations (numbered "3" in Figure 2), a SubTrees association (numbered "2" in Figure 2), and a SoftwareComponents association (numbered "1" in Figure 2). The Group objects are associated to the Tree objects using the Members association. From the Group, the association is Trees. Trees are associated to other trees using the ParentTree/SubTrees association. Note that a Tree can have only one ParentTree association but can have multiple Subtrees. In our example, there is only one SubTree. The Tree also has an association to a SoftwareComponent. This association helps an application determine which trees contain information for this application.

Previous Page | Next Page | Top of Page

Usage Scenario: Creating Metadata for a Workspace Server

Purpose

This usage scenario describes how to define a server using several submodels, including the Software Deployment Submodel, Transform Submodel, and Property Submodel. The example focuses on using the Property submodel to describe a SAS Integration Technologies Workspace server and explains how to create a template for this type of server. An XML representation is included to show how to construct Prototype objects and their properties.

Requirements

This usage scenario assumes the reader has a general understanding of the Property Submodel, Software Deployment Submodel, and Transform Submodel of the SAS Metadata Model, which are described in **SAS 9.1 Open Metadata Interface: Reference**. Readers who are not familiar with the general concepts of the SAS Open Metadata Architecture should refer to **Getting Started with the SAS 9.1 Open Metadata Interface**.

Description of a Server

A server is described in metadata using a set of DeployedComponent types (used to describe installed software) that has associated Connection objects (used to describe how to communicate with the server). IOM servers, DBMSs, and other servers are all described using these metadata types. There are several subtypes of Connection that provide additional information based on the communication protocol. For example, TCPIPConnection contains additional attributes used by TCP/IP communication, such as the host name and the port number.

DeployedComponent and its subtypes, along with TCPIPConnection, are generic types. Specific information about the software or the connection is provided through associated Property objects.

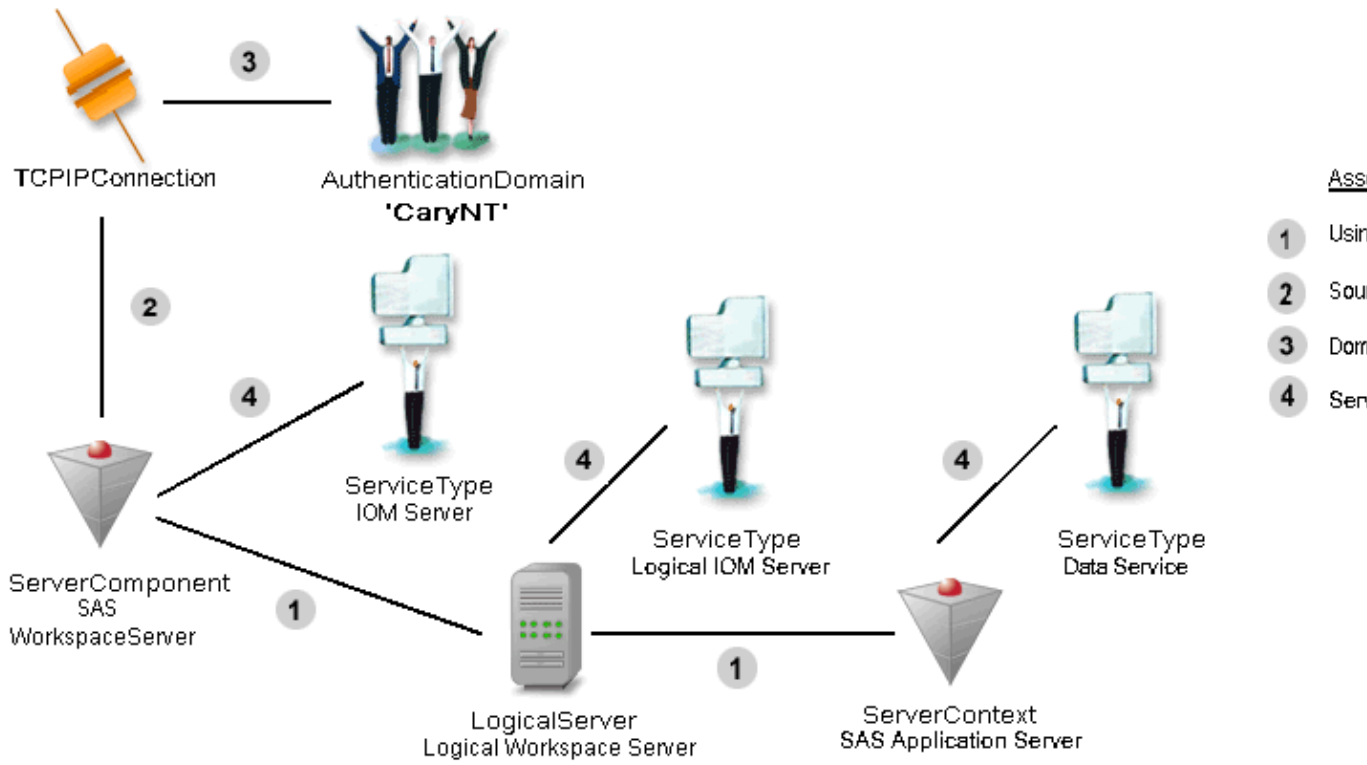
Workspace Server Metadata Objects

Figure 1 depicts the main objects used to describe a Workspace server. A ServerContext allows an application to group non−homogeneous servers together that share resources. The resources that are shared include SAS Libraries and directories. The ServerContext has an associated LogicalServer (number "1" in Figure 1). The LogicalServer is used to group homogeneous servers together. The servers grouped by the LogicalServer would typically be used for pooling or load balancing. However, it is important to note that even a single workspace server would be represented using a set of ServerContext, LogicalServer, and ServerComponent types. The ServerComponent represents the actual Workspace server. A ServerComponent type represents a configured instance of installed software. One installation of SAS, for example, may have several configurations of a server, each with different resources or options defined. The installation of the software is represented by a DeployedComponent type. Each of these software deployment types inherit from the DeployedComponent type. Each DeployedComponent has a ClassIdentifier attribute that is used to identify the kind of component. The ClassIdentifier can be a unique identifier, such as the GUID generated through CDTool, or a Java class name (including the package), or some other unique string. In this example, the ServerContext has a class identifier of SAS Application Server. The LogicalServer and the ServerComponent both use 440196D4−90F0−11D0−9F41−00A024BB830C.

The ServiceType object describes the kinds of services available from the server and is associated to each of the DeployedComponent types (see "4" in Figure 1). In this case, the ServerComponent is tagged as an IOM server. The spawner uses this information to locate the servers it should start. The LogicalServer has a service type of Logical IOM Server and the ServerContext has a service type of Data Service.

The TCPIPConnection object describes how to access the server and is associated to the ServerComponent using the SourceConnections association (number "2" in Figure 1). The AuthenticationDomain object describes the authentication domain used by the server when accessed using this connection information (number "3" in Figure 1).

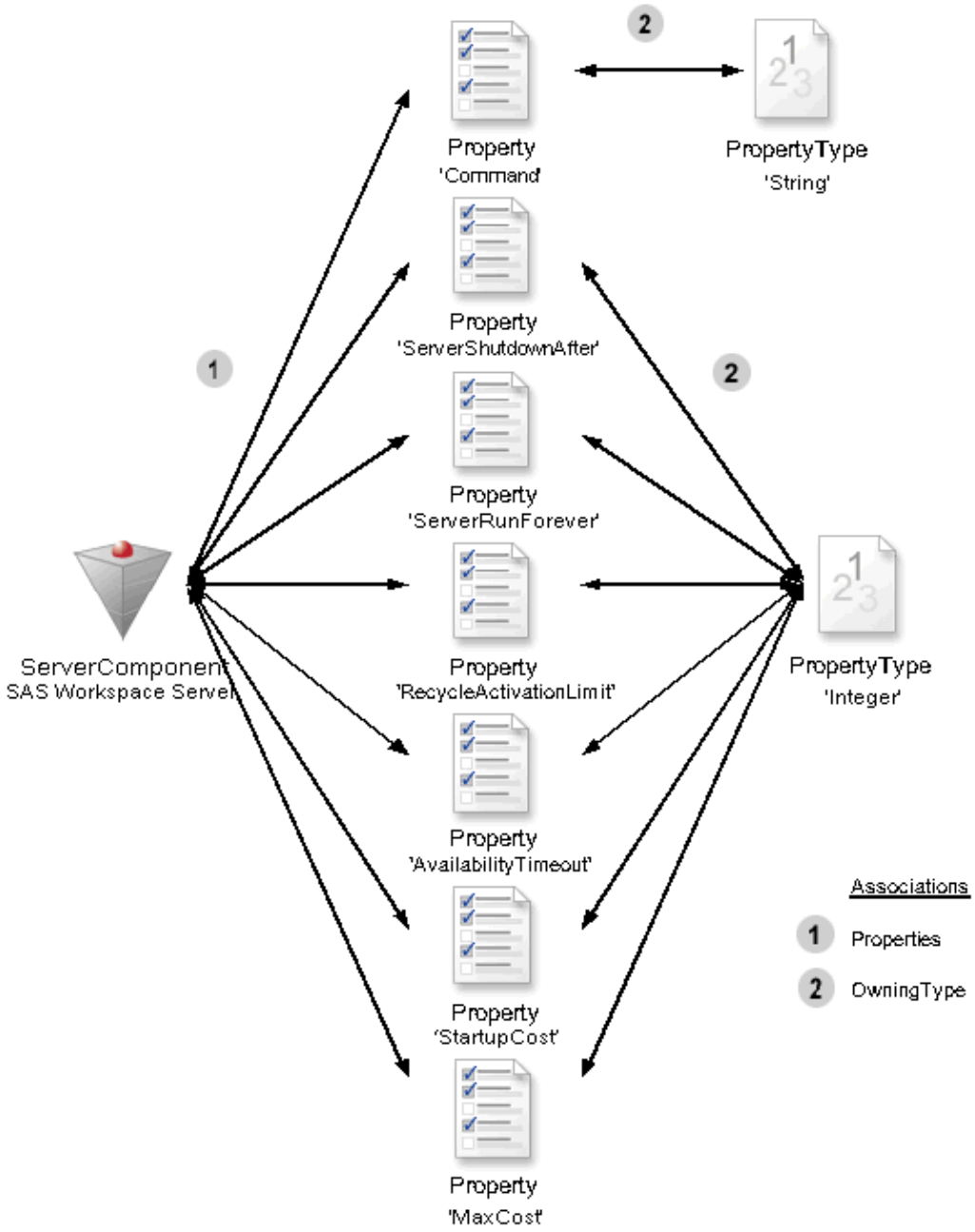
Figure 1. Metadata Objects Used to Describe a Workspace Server



DeployedComponent Types and Properties

Both the workspace server objects and the connection object require more information than is defined in the DeployedComponent types or a TCPIPConnection. The additional information is contained in Property objects. Figure 2 depicts the ServerComponent and its associated Property and PropertyType objects.

Figure 2. Property Objects for a WorkSpace Server



The Workspace server must provide information about how to start and manage it. These are not specific attributes of ServerComponent, so the information is kept in Property objects. The Property objects shown in Figure 2 are the default settings for the server. The associations numbered "1" depict the association of a Property object to its owning object, in this example, a ServerComponent. This association (from the owning object's perspective) is called Properties. All of the default Property objects should be available through this association.

If there are valid alternative values for Property objects, other than the default values, then PropertySet objects

are used. If someone wants to find ALL of the default Property objects, they will find them through the Properties association. If they are looking for a specific set of Property objects, then they should obtain a PropertySet object through the PropertySets association. The Property objects are available from the PropertySet object through the SetProperties association.

A PropertySet is also required to have a complete set of Property objects for a particular usage. There may be duplicate Property objects available through the Properties association, as well as through PropertySet objects, but that is because each of these are required to provide a complete set. An application should be able to obtain either the Property objects in the Properties association, or those in a PropertySet, and have all of the required information needed. There is no merging of Property objects through these associations, and there is no hierarchy of Property objects allowed through these associations.

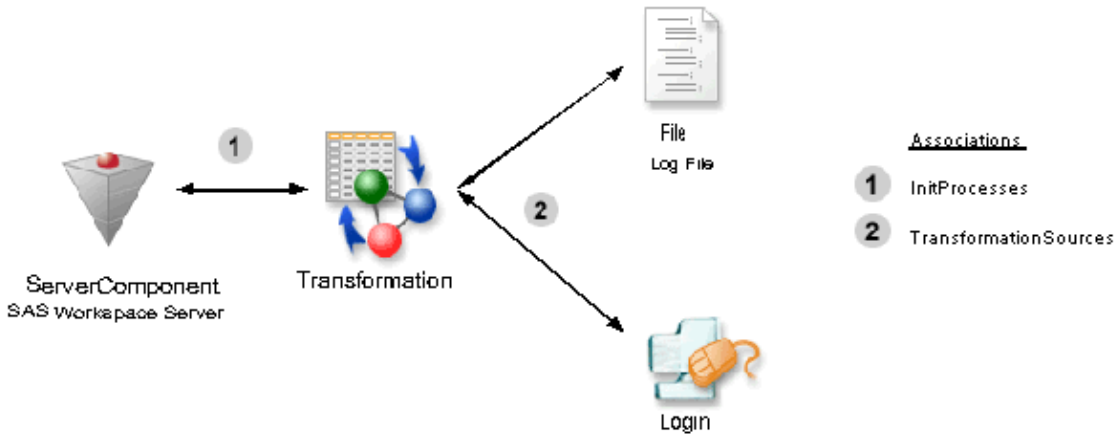
The Property object contains the name of the property used by a program (PropertyName). The value of the Property is saved in the DefaultValue attribute if it is a single value, or in an associated Text object if it refers to an array of values. This usage of the Text object is explained later in the TCPIPConnection and Property Objects section. Each Property object has a set of attributes that describe whether the Property is required, can be updated, and should be visible in a user interface. If a Property indicates that it is required, it does NOT mean that the model requires the Property; it means that a value must be provided for proper usage of this object. If a Property indicates that it cannot be updated, then the DefaultValue contains the only valid setting for the Property.

Each Property object must be associated to a PropertyType object that describes the SQL type of information stored in the DefaultValue or associated Text object. The association between a Property and its PropertyType (numbered "2" in Figure 2) is called OwningType. From the PropertyType, this association is called TypedProperties.

ServerComponent and Initialization

If the ServerComponent requires initialization information that should be stored in other metadata types, such as file names and locations, login information, or SAS library information, then a Transformation object that describes initialization process should be defined to associate that information to the ServerComponent. Figure 3 depicts an example of an initialization process.

Figure 3. Metadata Objects Used to Define an Initialization Process that Uses a File and Login as Input



There are two metadata types that represent processes. Processes that do not represent data flow are represented as a Transformation object, in this case, the initialization process. Processes that represent data flow, for example a SAS procedure that accepts input data and creates output data, are represented as ClassifierMaps.

This example shows an initialization process that uses a file and login as input to the process. The ServerComponent locates its initialization processes through the InitProcesses association (number "1" in Figure 3). The Transformation identifies the inputs to the initialization process through the TransformationSources association (number "2" in Figure 3).

TCPIPConnection and Property Objects

The metadata type TCPIPConnection provides connection information for SAS/SHARE, SAS/CONNECT, and SAS Integration Technologies servers as well as DBMSs and other types of servers. The TCPIPConnection object has a set of attributes that contain generic information used to access a server accessible through TCP/IP, but it does not contain all of the specific information required for a particular server. For example, servers provided by SAS may require a set of encryption options as part of the connection information. Other options and their values may be needed to complete a connection to a server. Rather than create a large number of subtypes of TCPIPConnection, one can be tailored to include specific connection information for each type of server. Additional connection information is contained in the Property objects.

Figure 4 depicts the metadata objects that contain the information needed to connect to a SAS Integration Technologies server using Bridge protocol.

Figure 4. Metadata Objects Representing a SAS Integration Technologies Server Connection

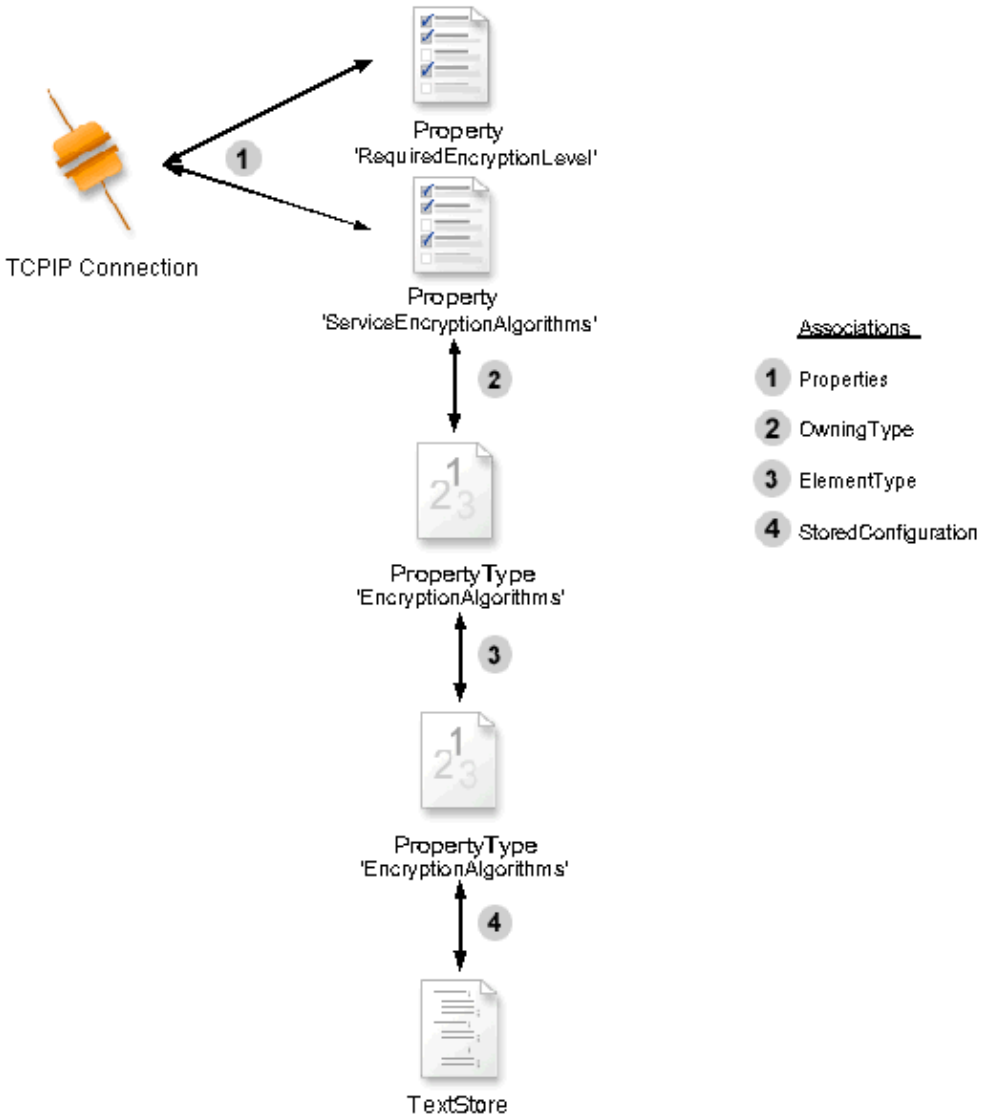


Figure 4 is very similar to Figure 2, but it includes additional information about the Property values and constraint information for the PropertyType.

Just like DeployedComponent, TCPIPConnection contains generic connection information used to communicate with any server accessible through TCP/IP. In this example, there are two Property objects that contain additional information used to establish the connection: one for the RequiredEncryptionLevel and one for the ServerEncryptionAlgorithms. These Property objects represent the default values and are accessible from the TCPIPConnection object through the Properties association (number "1" in Figure 4).

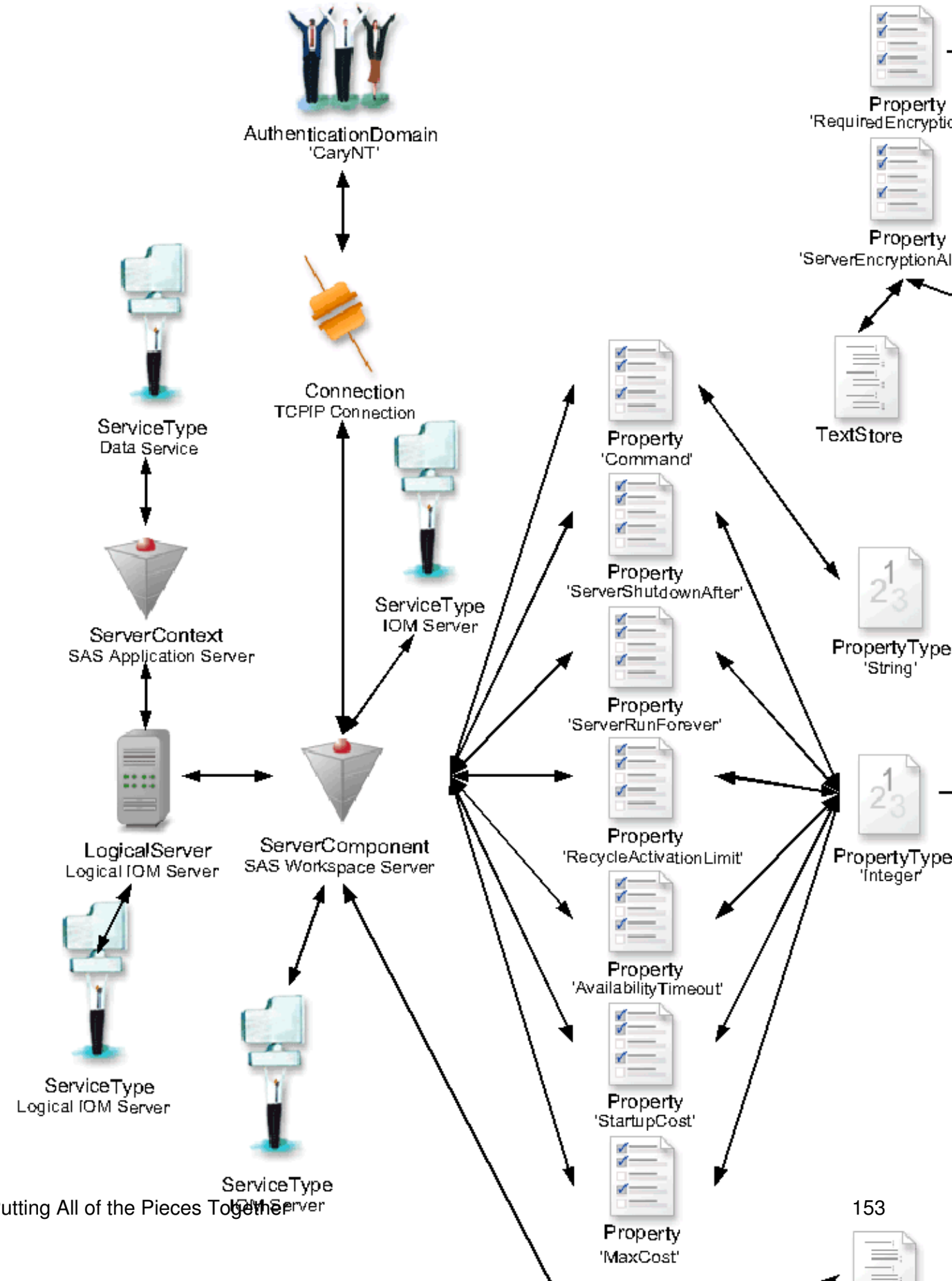
The Property for EncryptionAlgorithms differs from the other Property objects because the type of the object is Array. A PropertyType that has an SQLType of Array should use the ElementType association (number "3" in Figure 4) to find the type of the elements of the array. The valid values of the EncryptionAlgorithm PropertyType are constrained to a list of four values: RC2, RC4, DES and Triple DES. This list is stored in a

Text object, in this case, the subtype TextStore. The association between the Text object and PropertyType is called StoredConfiguration (number "4" in Figure 4).

Putting All of the Pieces Together

Figure 5 depicts all of the objects used to describe this Workspace server.

Figure 5. Summary of Objects Used to Describe the WorkSpace Server



Prototype Objects

We just described how to use Property objects to store additional information for an object, but how do you describe a property sheet that describes all of the information required for a scenario? Which Property objects should be included?

A Prototype object is used as a template for creating other metadata objects. In this example, we describe a Prototype for a Workspace server. A Prototype object contains an attribute that contains the metadata type described by the Prototype, in this case, a DeployedComponent, and has associations to other objects that define the attributes and associations for the object.

Prototype and Properties

In this example, the Prototype describes the information needed for a Workspace server. Its attribute, MetadataType, identifies which type is being described. Prototype objects may have AttributeProperty objects that describe the settings of the attributes of the templated metadata type. In this example, the AttributeProperty for ClassIdentifier indicates the only valid setting for this attribute for a Workspace Server.

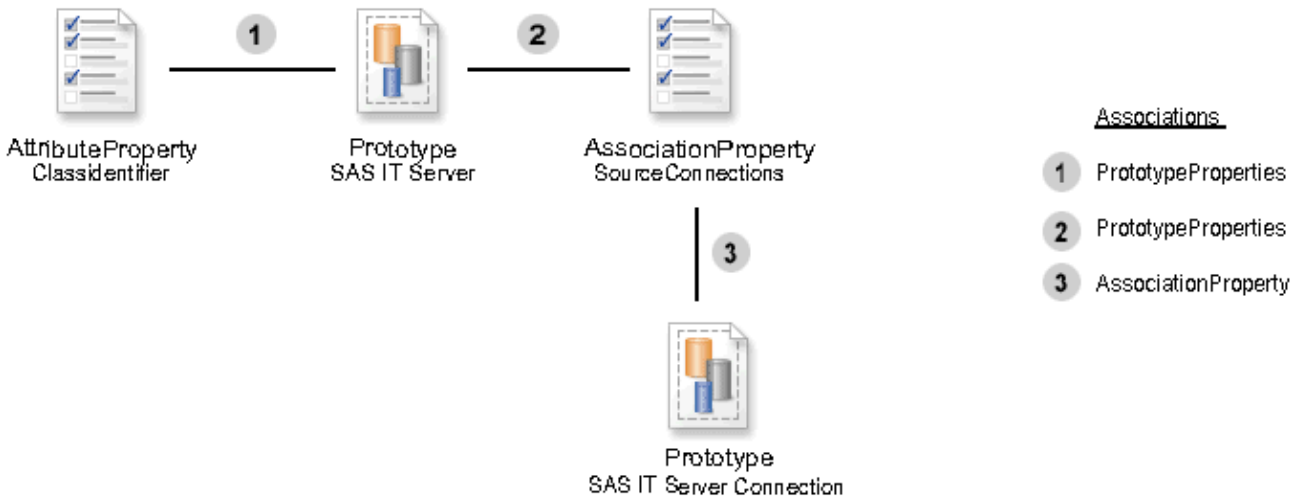
Prototype objects may also have AssociationProperty objects that describe the characteristics of associated objects. In this case, the AssociationProperty connects a Prototype of a TCPIPConnection object to the Prototype of a DeployedComponent.

Prototype objects locate AttributeProperty and AssociationProperty objects through the PrototypeProperties association or through PropertyGroup objects.

Objects created by using a Prototype definition may maintain an association to the Prototype that was used to create them through the UsingPrototype association.

Figure 6 depicts a Prototype for a DeployedComponent and an AttributeProperty and an AssociationProperty. The numbers "1" and "2" in Figure 6 represent the association called PrototypeProperties from the Prototype. The association from the AssociationProperty is called AssociatedPrototypes (numbered "3" in Figure 6) and is used to link prototype definitions together.

Figure 6. WorkSpace Server Prototype



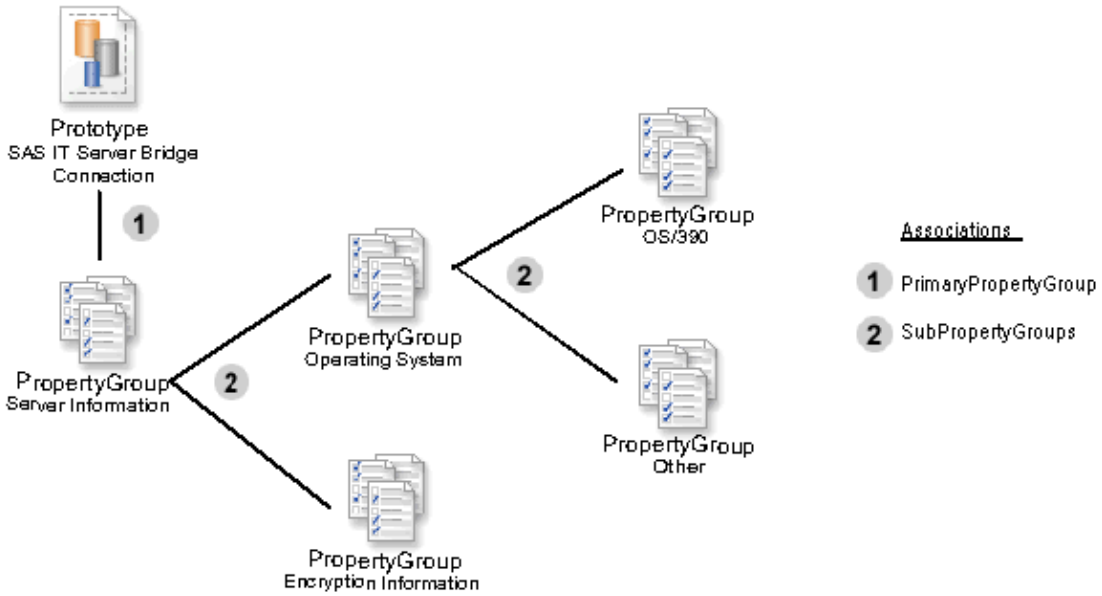
Prototype and Property

The TCPIPConnection that describes a Bridge connection has Property objects that describe the encryption level and algorithms. A Prototype also uses Property objects to contain that information, but the Properties and PropertySet associations are not used. Unfortunately, due to our subtyping structure, these associations are available to a Prototype object. However, these associations should never be used.

If a Prototype requires Property objects, then it should have a top–level PropertyGroup object associated to it through the PrimaryPropertyGroup association. PropertyGroup objects are used to logically group AttributeProperty, AssociationProperty, and Property objects. This grouping is primarily used to organize these objects for a user interface.

Each Prototype has a single top–level PropertyGroup, in this case named Server Information. Each PropertyGroup may have subgroups. The association between the Prototype and its top level group is identified as a PrimaryPropertyGroup (number "1" in Figure 7). The association between the top level PropertyGroup and its subgroups is SubpropertyGroups (number "2" in Figure 7). The subgroup Operating System has two subgroups labeled 'OS/390' and 'Other'. This association is numbered "3" in Figure 7 and is also known as SubpropertyGroups.

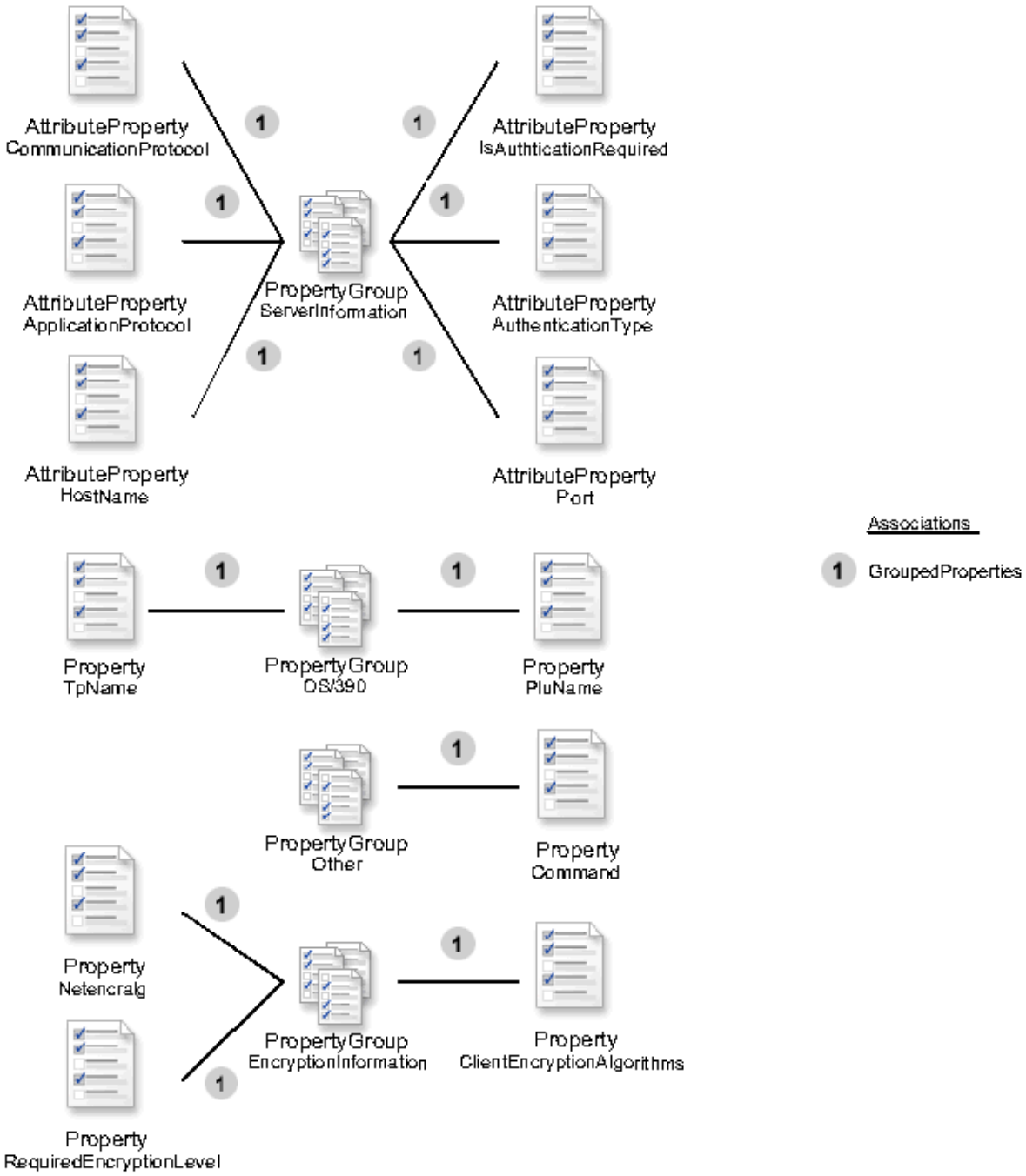
Figure 7. WorkSpace Server Bridge Connection Prototype



PropertyGroups and Properties

Figure 8 depicts the relationship between the subtypes of AbstractProperty objects and PropertyGroups. When used as part of a Prototype definition, a Property object must be associated to a only one PropertyGroup. AttributeProperty and AssociationProperty are associated using the GroupedProperties association (number "1" in Figure 8).

Figure 8. Relationship between the Subtypes of AbstractProperty Objects and PropertyGroup Objects



XML Example

See XML Representation of Metadata Objects for a Workspace Server to view the XML representation.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Usage Scenario: Creating Metadata Objects for Business Information

Purpose

Business information is defined as information about documents, people, and descriptions of other objects. This scenario describes the metadata types that are used to associate business information with other metadata objects. This desc riptive information can be associated to any other metadata object.

Requirements

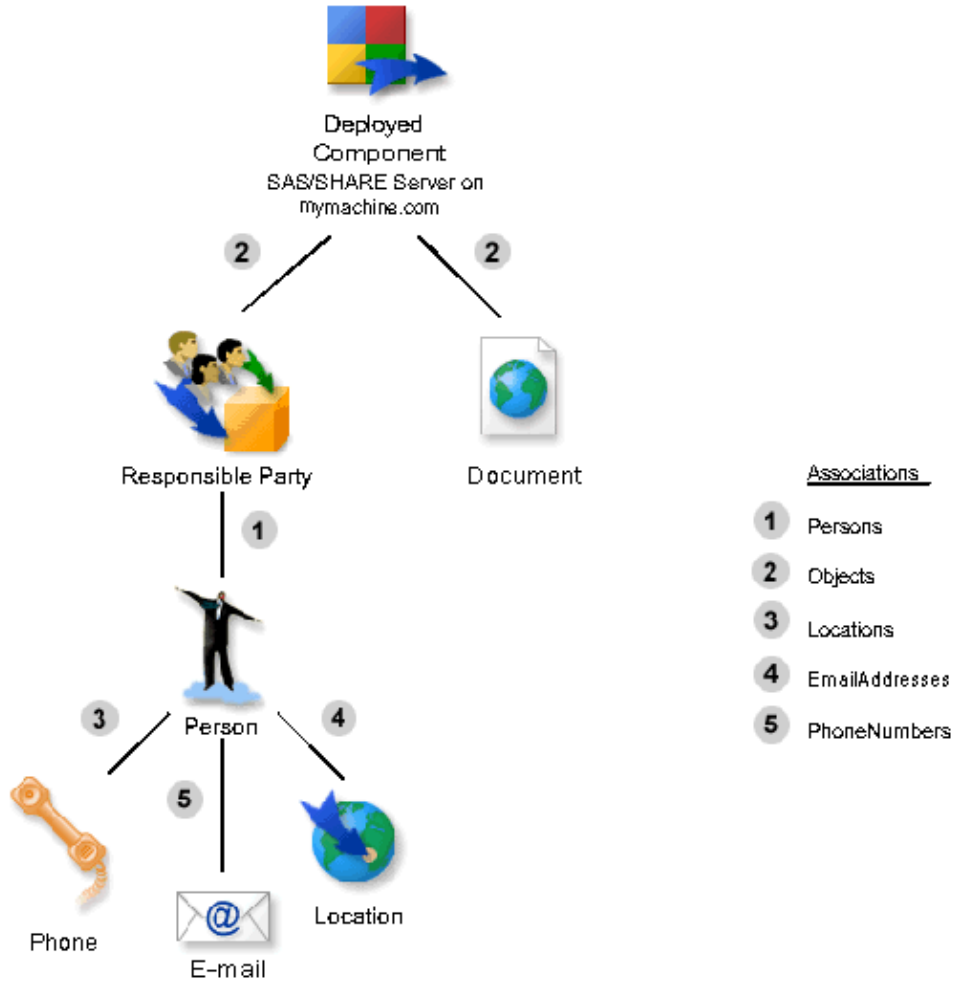
This usage scenario assumes the reader has general understanding of the Business Information Submodel of the SAS Open Metadata Model. Readers whom are not familiar with the general concepts of Op en Metadata Architecture should refer to the **Getting Started with the SAS 9.1 Open Metadata Interface**.

Creating Metadata Objects for Business Objects

Information about documents, people, descriptions of other objects is considered business information. This descriptive information can be associated to any other metadata object. The following metadata types are used to represent business information:

- Document – this type contains a URL that contains documentation
- TextStore – this type contains textual information
- Person – this type identifies an individual
- Location – the address of the person
- Email – the email address of the person
- Phone – the phone number of the person
- ResponsibleParty – this type identifies a relationship between a group of people and another object, and contains the role in which the people are acting.

In this example, information about a SAS/SHARE server, its administrator, and documentation, will be added to the metadata repository. Figure 1 depicts the metadata objects that represent a SAS/SHARE server, a document, a person, and an object that identifies the person as the server administrator.



See XML Representation of Metadata Objects for Business Objects to view the XML request for the metadata object shown in Figure 1.

In the sample XML request, the first object that is created is the ResponsibleParty. This object is used to associate people with a particular role. A role could be Administrator, Owner, User, or any other term used to describe the relationship between a person and another object. In this example, the role is "Administrator". ResponsibleParty has an association with one or more Person objects who act in the role defined in the Role attribute. This association is labeled Figure 1–1 in the diagram and is created through the Persons element in the XML request. The metadata data objects for ResponsibleParty and Person will be created when this request is sent to the server.

The Person object has associations to Phone (Figure 1–3), Location (Figure 1–4), and Email (Figure 1–5). These associations are created through the Locations, EmailAddresses, and PhoneNumbers elements in the XML request.

Usually there is another association to ResponsibleParty that lists the objects that are acted upon. The other association is labeled Figure 1–2 in the diagram and is created through the Objects element in the XML request. In this example, a SAS/SHARE server has already been defined in the metadata and has the identifier ABCDEFGH.A9000001. The metadata object that represents the SAS/SHARE server is associated with the ResponsibleParty object and is the object that has John Doe identified as an administrator.

The Document metadata type is used to associate documentation objects with other objects. The Documentation object contains an attribute URL that is used to locate the document. In this example, the server administration document is associated with the SAS/SHARE server. This association is labeled in the diagram as Figure 1−3 and is created by the Objects element in the XML. A Document object may be associated a set of objects, so all SAS/SHARE servers could be included in this list.

Previous Page | Next Page | Top of Page

SAS 9.1 Open Metadata Interface: User's Guide

Usage Scenario: Creating Metadata Objects for Mapping Data

Purpose

This scenario documents how to map a physical table to another and how to split one table into two.

Requirements

This usage scenario assumes the reader has general understanding of the Transform Submodel of the SAS Open Metadata Model. Readers whom are not familiar with the general concepts of Open Metadata Architecture should refer to the **Getting Started with the SAS 9.1 Open Metadata Interface**.

Mapping a Physical Table to Another Physical Table

The transformation of one table to another table could be a simple replication process from one location to another location. The location of the tables is modeled with types included in the Software Deployment Submodel and is not depicted in this usage scenario.

Figure 1 depicts the mapping of one physical table to another. A TransformationActivity is always the starting point when defining a Transformation. The TransformationActivity may in turn be grouped together in a Job that will be scheduled to run as a batch job or triggered by some other event. Grouping the TransformationActivity is not represented in this scenario.

A TransformationActivity is used to group TransformationSteps. The TransformationActivity has one TransformationStep in this scenario and is associated using the Steps association (Figure 1–1). It also has an association, TransformationTargets (Figure 1–2), to the output of this activity, TBL2. The actual Transformation is represented by the ClassifierMap and is associated to the TransformationStep using the Transformations association (Figure 1–3). The idea here is that each of level of this definition is a reusable part. TransformationActivities may be combined in different jobs, or a ClassifierMap may be part of a multiple TransformationSteps which in turn may be part of different TransformationActivities.

The ClassifierSources (Figure 1–4), or input, for this ClassifierMap contains the PhysicalTable, TBL1. The ClassifierTargets (Figure 1–5), or output, for the ClassifierMap contains another PhysicalTable named TBL2. A ClassifierMap also has a set of FeatureMaps (Figure 1–6). The columns on TBL1 are mapped directly to columns with the same name in the new table using the FeatureMap objects. The FeatureSources (Figure 1–7) are the inputs for this map, and FeatureTargets (Figure 1–8) are the output of this mapping. A FeatureMap could have an association to SourceCode that gives more detail about how to transform the column or columns. This source code may be a query that specifies the condition in which a records will be part of the target column or perhaps a mathematical function that performs a static operation or combines one or more columns. This scenario does a simple copy keeping all data the same in the target column.

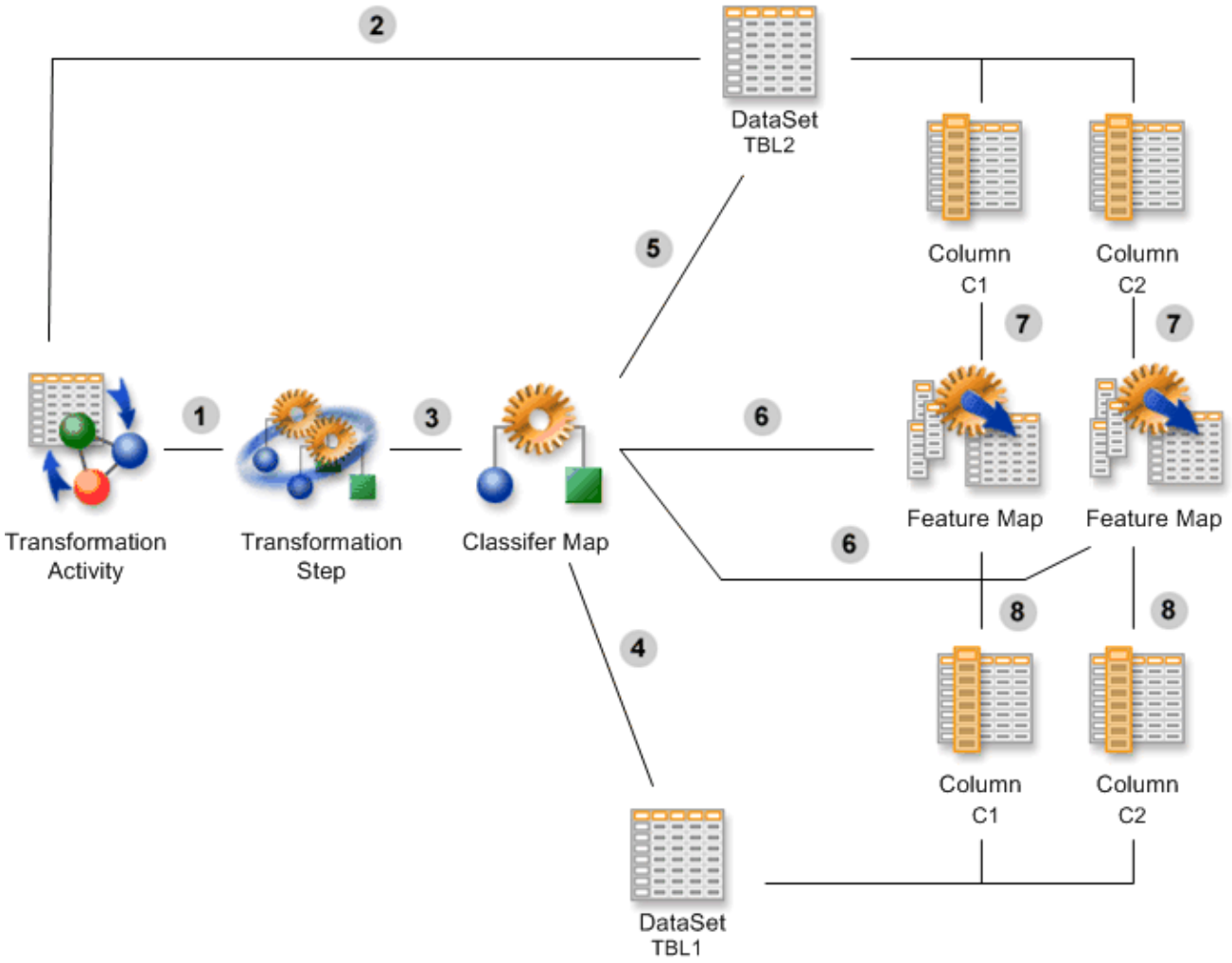


Figure 1. The transformation of one table to another table.

Splitting One Table Into Two

Figure 2 shows the objects used to represent the following SAS data step code:

```
data TBL2(keep=c1 c2 options=?)  TBL3(keep=c1 c3 c4 options=?)  set TBL1;  if c1 <
10 then output TBL3;  output TBL2;run;
```

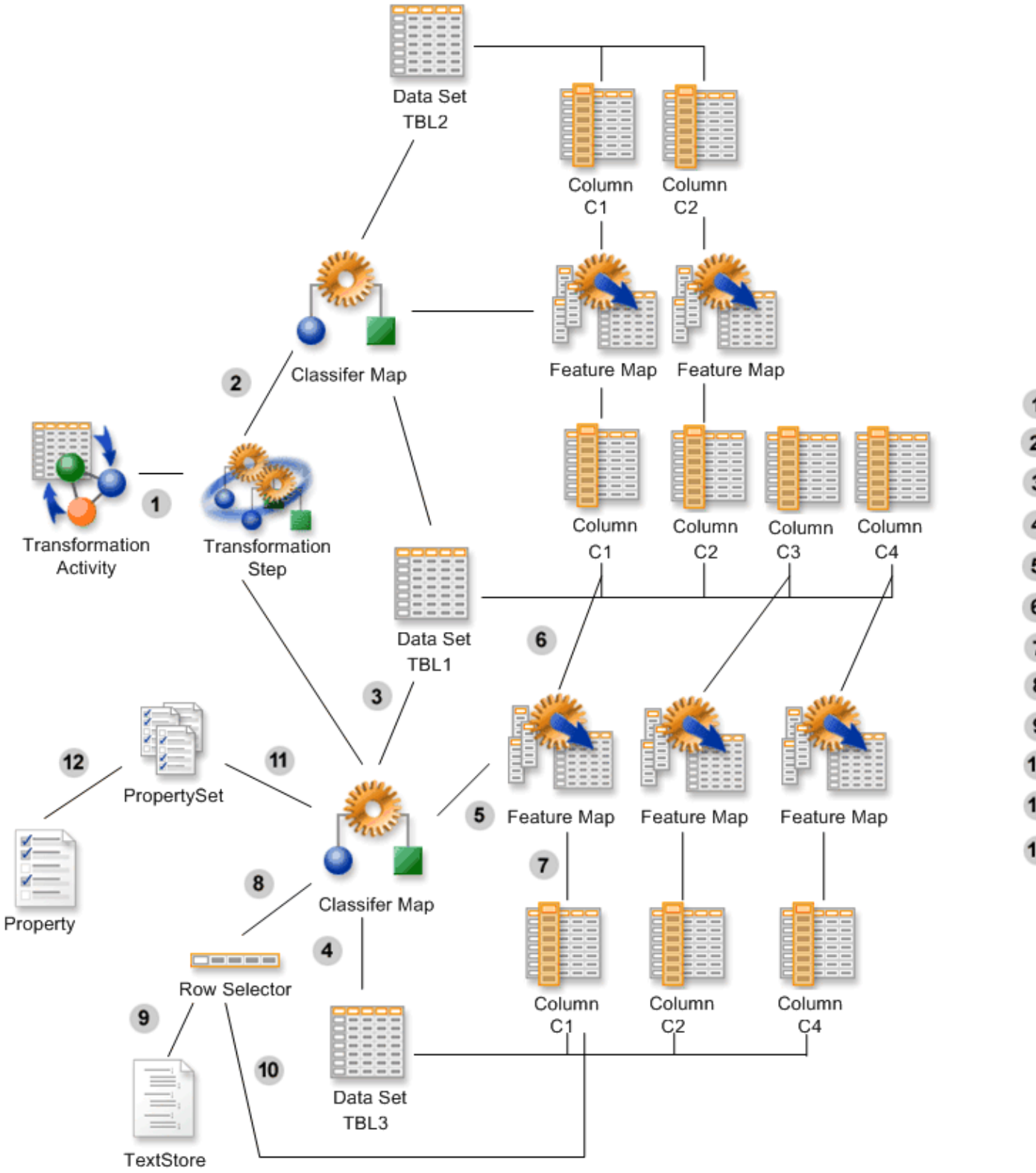


Figure 2. The splitting of one table into two tables.

Here again, the TransformationActivity has a set of Steps (Figure 1–1). In this scenario, there is one TransformationStep associated to the TransformationActivity. The TransformationStep has a set of

ClassifierMaps (Figure 1–2). This step has two associated ClassifierMap objects. Each one of these represent a process which is used to create a new table. Each ClassifierMap has a set of ClassifierSources (Figure 1–3). ClassifierSources may be any type of Classifier. In this scenario, the source for both ClassifierMap objects is the PhysicalTable, TBL1. This is the table that will be split into two other tables. The ClassifierMap has a set of ClassifierTargets (Figure 1–4). ClassifierTargets may be any type of Classifier. The target in this scenario is the PhysicalTable, TBL3. The other ClassifierMap in this scenario has TBL2 as a target.

Each ClassifierMap has a set of FeatureMaps (Figure 1–5). One ClassifierMap has three FeatureMaps, and the other has two FeatureMaps in this scenario. The FeatureMap is used to map any number of features to other features. For example, the FeatureMap could be used to combine multiple columns using a mathematical function. In this scenario, the columns which are selected are not transformed. The FeatureMap has a set of FeatureTargets (Figure 1–7). There is one target, Column C1 in TBL3, for the FeatureMap represented with this association. The FeatureMap also has a set of FeatureSources (Figure 1–6). There is one source, Column C1 in TBL1, for this FeatureMap.

The ClassifierMap may have an AssociatedRowSelector (Figure 1–8). The RowSelector is used to help determine which rows should be part of the ClassifierTargets. The RowSelector has a SourceCode (Figure 1–9) association to a TextStore object. The TextStore objects stores the actual expression used to determine which rows to select. The expression may contain macro variables which the application would need to replace with a value from the metadata. If this is the case, there would be associated Variable objects. The Variable object contains information about the string or Marker which is to be substituted in the expression and an association to the metadata object where the substitution value would be found. The Variable object also has an attribute, ValueType, which may contain the name of the attribute in the associated metadata object which is to be used to get the value. The RowSelector also has a Columns (Figure 1–10) association to the Columns used by this RowSelector.

If there were options in this scenario, they would be stored as a Property object and be grouped in a PropertySet and associated to the ClassifierMap using the PropertySets (Figure 1–11) association. The properties are associated to the PropertySet using the Properties (Figure 1–12) association. The Property contains the name of the property or option and the value for the property. There is also a Role that may be assigned to the Property.

Previous Page | Next Page | Top of Page