SDO XML Data Access Service

Introduction

In order to use the XML Data Access Service for Service Data Objects, you will need to understand some of the concepts behind SDO: the data graph, the data object, XPath and property expressions, and so on. If you are not familiar with these ideas, you might want to look first at the section on SDO.

The job of the XML DAS is to move data between the application and an XML data source, which can be either a file or a URL. SDOs are always created and maintained according to a model which defines type names and what property names each type may have. For data which is from XML, this SDO model is built from a schema file written in XML schema language (an xsd file). This schema file is usually passed to the create method when the XMLDAS is initialised. The » SDO 2.0 specification defines the mapping between XML types and SDO types. There are a number of small limitations in the PHP support - not everything which is in the specification can be done - and these limitations are summarised in a later section.

Installing/Configuring

Requirements

The SDO XML Data Access Service requires PHP 5.1.0 or higher. It is packaged with the SDO extension and requires SDO to have been installed. See the SDO installation instructions for the details of how to do this.

Installation

The XML Data Access Service is packaged and installed as part of the SDO extension. Please Refer SDO installation instructions.

Runtime Configuration

This extension has no configuration directives defined in *php.ini*.

Resource Types

This extension has no resource types defined.

Predefined Constants

This extension has no constants defined.

Examples

Several of the following examples are based on the letter example described in the SDO documentation. The examples assume the XML Schema for the letter is contained in a file *letter.xsd* and the letter instance is in the file *letter.xml*. These two files are reproduced here:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:letter="http://letterSchema"
 targetNamespace="http://letterSchema">
 <xsd:element name="letters" type="letter:FormLetter"/>
 <xsd:complexType name="FormLetter" mixed="true">
   <xsd:sequence>
     <xsd:element name="date" minOccurs="0" type="xsd:string"/>
     <xsd:element name="firstName" minOccurs="0" type="xsd:string"/>
     <xsd:element name="lastName" minOccurs="0" type="xsd:string"/>
   </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

```
<letter:letters xmlns:letter="http://letterSchema">
 <date>March 1, 2005</date>
 Mutual of Omaha
 Wild Kingdom, USA
 Dear
 <firstName>Casy</firstName>
 <lastName>Crocodile</lastName>
 Please buy more shark repellent.
 Your premium is past due.
</letter:letters>
```

Example #1 - Loading, altering, and saving an XML document

The following example shows how an XML document can be loaded from a file, altered, and written back.

```
<?php
/**
* Load, update, and save an XML document
*/
try {
  $xmldas = SDO_DAS_XML::create("letter.xsd");
  $document = $xmldas->loadFile("letter.xml");
  $root_data_object = $document->getRootDataObject();
  $root_data_object->date = "September 03, 2004";
  $root_data_object->firstName = "Anantoju";
  $root_data_object->lastName = "Madhu";
  $xmldas->saveFile($document, "letter-out.xml");
  echo "New file has been written:\n";
  print file_get_contents("letter-out.xml");
```

```
} catch (SDO_Exception $e) {
  print($e->getMessage());
}
?>
```

An instance of the XML DAS is first obtained from the SDO_DAS_XML::create() method, which is a static method of the SDO_DAS_XML class. The location of the xsd is passed as a parameter. Once we have an instance of the XML DAS initialised with a given schema, we can use it to load the instance document using the **loadFile()** method. There is also a **loadString()** method if you want to load an XML instance document from a string. If the instance document loads successfully, you will be returned an object of type SDO_DAS_XML_Document, on which you can call the **getRootDataObject()** method to get the SDO data object which is the root of the SDO data graph. You can then use SDO operations to change the graph. In this example we alter the *date*, *firstName*, and *lastName* properties. Then we use the **saveFile()** method to write the changed document back to the file system. The saveFile method has an optional extra integer argument which if specified will cause the XML DAS to format the XML, using the integer as the amount to indent by at each change in level on the document.

This will write the following to *letter-out.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<FormLetter xmlns="http://letterSchema" xsi:type="FormLetter"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <date>September 03, 2004</date>
 Mutual of Omaha
 Wild Kingdom, USA
 Dear
 <firstName>Anantoju</firstName>
 <lastName>Madhu</lastName>
 Please buy more shark repellent.
 Your premium is past due.
</FormLetter>
```

Example #2 - Creating a new XML document

The previous example loaded the document from a file. This example shows how to create an SDO data graph in memory. In this example it is then saved to an XML string. Furthermore, because the letter contains both structured and unstructured content, it uses the Sequence API as well assignments to properties to construct the data graph.

```
<?php
/**
* Create an XML document from scratch
*/
try {
  $xmldas = SDO_DAS_XML::create("letter.xsd");
  try {
      $doc = $xmldas->createDocument();
      $rdo = $doc->getRootDataObject();
      $seq = $rdo->getSequence();
      $seq->insert("April 09, 2005", NULL, 'date');
```

```
      $seq->insert("Acme Inc. ", NULL, NULL);
      $seq->insert("United Kingdom. ");
      $seq->insert("Dear", NULL, NULL);
      $seq->insert("Tarun", NULL, "firstName");
      $seq->insert("Nayaraaa", NULL, "lastName");
      $rdo->lastName = "Nayar";
      $seq->insert("Please note that your order number ");
      $seq->insert(12345);
      $seq->insert(" has been dispatched today. Thanks for your business
with us.");
      print($xmldas->saveString($doc));
  } catch (SDO_Exception $e) {
      print($e);
  }
} catch (SDO_Exception $e) {
  print("Problem creating an XML document: " . $e->getMessage());
}
?>
```

The **createDocument()** method on the XML DAS returns a document object with a single root data object corresponding to an empty document element. The element name of the document element is known from the schema file. If there is any ambiguity about what the document element is, as there can be when more than one schema has been loaded into the same XML DAS, the element name and the namespace URI can be passed to the **createDocument()** method.

This will emit the following output (line breaks have been inserted for readability):

```
<?xml version="1.0" encoding="UTF-8"?>
<FormLetter xmlns="http://letterSchema" xsi:type="FormLetter"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<date>April 09, 2005</date>
Acme Inc. United Kingdom.
Dear
<firstName>Tarun</firstName>
<lastName>Nayar</lastName>
Please note that your order number 12345 has been
dispatched today. Thanks for your business with us.
</FormLetter>
```

Example #3 - Setting XML document properties

This third example shows you how to set the XML version and encoding on the document object. These will be used when the XML is written out. If no XML declaration is wanted at all (perhaps you want to generate the XML as a string to embed in something) then you can use the **setXMLDeclaration()** method to suppress it.

```
<?php
/**
* Illustrate the calls that control the XML declaration
*/
  $xmldas = SDO_DAS_XML::create("letter.xsd");
  $document = $xmldas->loadFile("letter.xml");
  $document->setXMLVersion("1.1");
```

```
  $document->setEncoding("ISO-8859-1");
  print($xmldas->saveString($document));
?>
```

The XML version and encoding are set in the XML declaration at the top of the XML document.

```
<?xml version="1.1" encoding="ISO-8859-1"?>
.../...
```

Example #4 - Using an open type

This fourth example illustrates the use of an SDO open type and the use of the **createDataObject()** method. For this example we use the following two schema:

```
<schema
 xmlns="http://www.w3.org/2001/XMLSchema">

 <element name="jungle">
   <complexType>
     <sequence>
       <any minOccurs="0" maxOccurs="unbounded"/>
     </sequence>
   </complexType>
 </element>

</schema>
```

Note the presence of the *any* element in the definition. This first schema defines the *jungle* complex type as containing a sequence of any other type. The other types that the example will use are defined in a second schema file:

```
<schema xmlns= "http://www.w3.org/2001/XMLSchema">

  <complexType name="snakeType">
    <sequence>
      <element name= "name" type="string"/>
      <element name= "length" type="positiveInteger" />
    </sequence>
  </complexType>

  <complexType name="bearType">
    <sequence>
      <element name= "name" type="string"/>
      <element name= "weight" type="positiveInteger" />
    </sequence>
  </complexType>

  <complexType name="pantherType">
    <sequence>
      <element name= "name" type="string"/>
      <element name= "colour" type="string" />
    </sequence>
```

```
  </complexType>

</schema>
```

Here is the example PHP code that uses these two schema files:

```
<?php

/**
* Illustrate open types and the use of the addTypes() method
*/

$xmldas = SDO_DAS_XML::create();
$xmldas->addTypes("jungle.xsd"); // this is an open type i.e. the xsd
specifies it can contain "any" type
$xmldas->addTypes('animalTypes.xsd');

$baloo            = $xmldas->createDataObject('','bearType');
$baloo->name      = "Baloo";
$baloo->weight    = 800;

$bagheera         = $xmldas->createDataObject('','pantherType');
$bagheera->name   = "Bagheera";
$bagheera->colour = 'inky black';

$kaa              = $xmldas->createDataObject('','snakeType');
$kaa->name        = "Kaa";
$kaa->length      = 25;

$document         = $xmldas->createDocument();
$do               = $document->getRootDataObject();
$do->bear         = $baloo;
$do->panther      = $bagheera;
$do->snake        = $kaa;

print($xmldas->saveString($document,2));

?>
```

These two schema files are loaded into the XML DAS with first the **create()** and **addTypes()** methods. The **createDataObject()** method is used to create three separate data objects. In each case the namespaceURI and typename of the type are passed to the **createDataObject()** method: in this example the namespace URI is blank because no namespace is used in the schema. Once the three data objects - representing a bear, a panther and a snake - have been created, a document object is created with the **createDocument()** method. In this case there is no ambiguity about what the document element of the document should be - as the second schema file only defines complex types, the document element can only be the global *jungle* element defined in the first schema. This document will have a single root data object corresponding to an empty document element *jungle*. As this is an open type, properties can be added at will. When the first assignment is made to *$do->bear*, a property *bear* is added to the root data object: likewise for the next two assignments. When the document is written out by the **saveString()** method, the resulting document is:

```
<?xml version="1.0" encoding="UTF-8"?>
<jungle xsi:type="jungle"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <bear xsi:type="bearType">
   <name>Baloo</name>
   <weight>800</weight>
 </bear>
 <panther xsi:type="pantherType">
   <name>Bagheera</name>
   <colour>inky black</colour>
 </panther>
 <snake xsi:type="snakeType">
   <name>Kaa</name>
   <length>25</length>
 </snake>
</jungle>
```

Example #5 - Finding out what you can from the document

This example is intended to illustrate how you can find the element name and namespace of the document element from the XML Document object, and the SDO type and namespace from the root data object of the XML data object, and how they relate to one another. This can be difficult to understand because there are four method calls: two can be made against the Document object, and two that can be made against any data object including the root data object. Because of the rules that define how the SDO model is derived from the XML model, when the data object concerned is the root object that represents the document object for the document, only three possible values can come back from these four method calls.

The two method calls that can be made against the document are **getRootElementName()** and **getRootEelementURI()**. These return the element name and namespace of the document element, respectively.

The two method calls that can be made against any data object are **getTypeName()** and **getTypeNamespaceURI()**. These return the SDO type name and type namespace of the data object, respectively.

Always, calling **getRootElementURI()** on the document object will return the same value as calling **getNamespaceURI()** on the root data object. Essentially, the information is all derived from the first few lines of the schema file, where there are three distinct pieces of information. For illustration, here again are the first few lines of the letter.xsd that we used above.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns:letter="http://letterSchema"
           targetNamespace="http://letterSchema">

 <xsd:element name="letters" type="letter:FormLetter"/>

 <xsd:complexType name="FormLetter" mixed="true">
   .../...
```

The three important values are:

- *letters*, the name of the document element
- *FormLetter*, the name of the complex type of the document element. This is also the name of the SDO type of the root data object.
- *http://letterSchema*, the namespace to which the document element belongs. This is also the namespaceURI of the SDO type of the root data object.

It is part of the XML-SDO mapping rules that when the SDO model is built from the schema file, the typename and namespaceURI of the SDO types for the root element are taken from those of the complex type of the document element, where it exists. Hence in this example the typename of the root data object is FormLetter. In the event that there is no separate complex type definition for the document element, when the the type is defined inline and is anonymous, the SDO type name will be the same as the element name.

The following program loads the letter document and checks the return values from each of the four calls.

```
<?php
/**
* Finding out what you can about the document and document element
* This can be quite hard to understand because there are four calls
* Two calls are made against the document
* Two calls are made against the root data object and its model
* Because of the SDO-XML mapping rules and how the SDO model is derived
* from the XML model, only three possible values can come back from these
four calls.
* Always, $document->getRootElementURI() == (type of root data
object)->namespaceURI
* Essentially, it all comes form the first few lines of the xsd:
* <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
*   xmlns:letter="http://letterSchema"
*   targetNamespace="http://letterSchema">
*   <xsd:element name="letters" type="letter:FormLetter"/>
*/

$xmldas = SDO_DAS_XML::create("letter.xsd");
$document = $xmldas->loadFile("letter.xml");
$root_do = $document->getRootDataObject();

/**
* The "root element name" is the element name of the document element
* in this case 'letters'
* This matches the 'name' attribute of the document element in the xsd and
matches
* the element name from the xml
*/
echo "The document element name is " . $document->getRootElementName() .
"\n";
assert($document->getRootElementName() == 'letters'); // a property of the
document

/**
* The "root element URI" is the namespace part of the element name of the
document element
* in this case 'http://letterSchema' since 'letters' is in that namespace
```

```
* This is taken from the xsd and matches the namespace picked up from the
xml
*/
echo "The document element is in the namespace " .
$document->getRootElementURI() . "\n";
assert($document->getRootElementURI() == 'http://letterSchema'); // a
property of the document

/**
* The type name is taken from the SDO model
* The XML-SDO mapping rules make this either:
*   The name of the complexType if there is one (in this case there is)
*   The document element name if there no complexType
* This is taken from the xsd
*/
echo "The type name of the root data object is " . $root_do->getTypeName() .
"\n";
assert($root_do->getTypeName() == 'FormLetter');

/**
* The type's namespaceURI is taken from the SDO model
* The XML-SDO mapping rules ensure that this will always be the same as
* the namepace URI of the document element
*/
echo "The namespaceURI of the root data object is " .
$root_do->getTypeNamespaceURI() . "\n";
assert($root_do->getTypeNamespaceURI() == 'http://letterSchema');

?>
```

The output from this program is as follows:

```
The document element name is letters
The document element is in the namespace http://letterSchema
The type name of the root data object is FormLetter
The namespaceURI of the root data object is http://letterSchema
```

Example #6 - Printing the SDO model

The XML DAS provides a simple means to see what types and properties have been loaded. The php "print" or "echo" instruction will print out the types and properties.

```
<?php
/**
* Illustrate printing out the model
*/

$xmldas = SDO_DAS_XML::create("letter.xsd");
print $xmldas;

?>
```

The output from this program is as follows:

```
object(SDO_XML_DAS)#1 {
18 types have been defined. The types and their properties are::
1. commonj.sdo:BigDecimal
```

```
2. commonj.sdo:BigInteger
3. commonj.sdo:Boolean
4. commonj.sdo:Byte
5. commonj.sdo:Bytes
6. commonj.sdo:ChangeSummary
7. commonj.sdo:Character
8. commonj.sdo:DataObject
9. commonj.sdo:Date
10. commonj.sdo:Double
11. commonj.sdo:Float
12. commonj.sdo:Integer
13. commonj.sdo:Long
14. commonj.sdo:Short
15. commonj.sdo:String
16. commonj.sdo:URI
17. http://letterSchema:FormLetter
   - date (commonj.sdo:String)
   - firstName (commonj.sdo:String)
   - lastName (commonj.sdo:String)
18. http://letterSchema:RootType
   - letters (http://letterSchema:FormLetter)
```

SDO DAS XML Functions

Predefined Classes

The XML DAS provides two main classes. The first is SDO_DAS_XML which is the main class used to fetch the data from the XML source and used to write the data back. The second is the SDO_DAS_XML_Document class, which represents the data in the XML document.

There are also some exception classes which can be thrown if errors are found when looking for or parsing the xsd or xml files.

SDO_DAS_XML

This is the main class of the XML DAS, which is used fetch the data from the xml source and also used to write the data back. Other than the methods to load and save xml files,

Methods

- create This is a static method available in the SDO_DAS_XML class. Used to construct an SDO_DAS_XML object.
- addTypes Works in much the same way as **create()** but used to add the contents of a second or subsequent schema file to an XML DAS that has already been created.
- createDataObject Can be used to construct an SDO data object of a given type.
- createDocument Can be used to construct an XML Document object from scratch.
- loadFile Loads the xml instance document from a file. This file can be at local file system or it can be on a remote host.
- loadString same as the above method. Loads the xml instance which is available as string.
- saveFile save SDO_DAS_XML_Document object as a xml file.
- saveString save SDO_DAS_XML_Document object as a xml string.

SDO_DAS_XML_Document

This class can be used to get to the name and namespace of the document element, and to get to the root data object of the document. Lastly, it can also be used to set the XML version and encoding of a document on output.

Methods

- getRootDataObject gets the root DataObject.
- getRootElementName gets the root DataObject's name.
- getRootElementURI gets the root DataObject's URI.
- setEncoding sets the encoding string with the given value.
- setXMLDeclaraion to set/unset the xml declaration.
- setXMLVersion sets the xml version with the given value.

SDO_DAS_XML_ParserException

Is a subclass of SDO_Exception. Thrown for any parser errors while loading the xsd/xml file.

SDO_DAS_XML_FileException

Is a subclass of SDO_Exception. Thrown by any of the methods that load data from a file, when the file cannot be found.

Limitations compared with SDO 2.0 specification

The » SDO 2.0 specification defines the mapping between XML types and SDO types. With Java SDO, this mapping is implemented by the XMLHelper. With SDO for PHP, this mapping is implemented by the XML Data Access Service. The XML DAS implements the mapping described in the SDO 2.0 specification with some restrictions. A detailed list is of the limitations is:

XML Simple Types

1. Simple Type with sdoJava:instanceClass - no PHP equivalent provided.
2. Simple Type with sdoJava:extendedInstanceClass - no PHP equivalent provided.
3. Simple Type with list of itemType.
4. Simple Type with union.

XML Complex Types

1. Complex Type with sdo:aliasName - no PHP support for SDO Type aliases.

XSD Attribute

1. Attribute with sdo:aliasName - no PHP support for SDO property aliases.
2. Attribute with default value - no PHP support for SDO property defaults.
3. Attribute with fixed value - no PHP support for SDO read-only properties or default

values.

4. Attribute referencing a DataObject with sdo:propertyType - no support for sdo:propertyType="...".
5. Attribute with bi-directional property to a DataObject with sdo:oppositeProperty and sdo:propertyType - no PHP support for SDO opposite.

XSD Elements

1. Element with sdo:aliasName - no PHP support for SDO property aliases.
2. Element with substitution group.

XSD Elements with Simple Type

1. Element of SimpleType with default - no PHP support for SDO defaults
2. Element of SimpleType with fixed value - no PHP support for SDO read-only properties or default values.
3. Element of SimpleType with sdo:string - no support for sdo:string="true".
4. Element referencing a DataObject with sdo:propertyType - no support for sdo:propertyType="..."
5. Element with bi-directional reference to a DataObject with sdo:oppositeProperty and sdo:propertyType - no PHP support for SDO opposite.

SDO_DAS_XML_Document::getRootDataObject

SDO_DAS_XML_Document::getRootDataObject -- Returns the root SDO_DataObject

Description

SDO_DataObject **SDO_DAS_XML_Document::getRootDataObject** (void)

Returns the root SDO_DataObject.

Parameters

Return Values

Returns the root SDO_DataObject.

SDO_DAS_XML_Document::getRootElementName

SDO_DAS_XML_Document::getRootElementName -- Returns root element's name

Description

string **SDO_DAS_XML_Document::getRootElementName** (void)

Returns root element's name.

Parameters

Return Values

Returns root element's name.

SDO_DAS_XML_Document::getRootElementURI

SDO_DAS_XML_Document::getRootElementURI -- Returns root element's URI string

Description

string **SDO_DAS_XML_Document::getRootElementURI** (void)

Returns root element's URI string.

Parameters

Return Values

Returns root element's URI string.

SDO_DAS_XML_Document::setEncoding

SDO_DAS_XML_Document::setEncoding -- Sets the given string as encoding

Description

void **SDO_DAS_XML_Document::setEncoding** (string `$encoding`)

Sets the given string as encoding.

Parameters

`encoding`
Encoding string.

Return Values

None.

SDO_DAS_XML_Document::setXMLDeclaration

SDO_DAS_XML_Document::setXMLDeclaration -- Sets the xml declaration

Description

void **SDO_DAS_XML_Document::setXMLDeclaration** (bool `$xmlDeclatation`)

Controls whether an XML declaration will be generated at the start of the XML document. Set to *true* to generate the XML declaration, or *false* to suppress it.

Parameters

`xmlDeclatation`
Boolean value to set the XML declaration.

Return Values

None.

SDO_DAS_XML_Document::setXMLVersion

SDO_DAS_XML_Document::setXMLVersion -- Sets the given string as xml version

Description

void **SDO_DAS_XML_Document::setXMLVersion** (string `$xmlVersion`)

Sets the given string as xml version.

Parameters

`xmlVersion`
: xml version string.

Return Values

None.

SDO_DAS_XML::addTypes

SDO_DAS_XML::addTypes -- To load a second or subsequent schema file to a SDO_DAS_XML object

Description

void **SDO_DAS_XML::addTypes** (string `$xsd_file`)

Load a second or subsequent schema file to an XML DAS that has already been created with the static method **create()**. Although the file may be any valid schema file, a likely reason for using this method is to add a schema file containing definitions of extra complex types, hence the name. See Example 4 of the parent document for an example.

Parameters

`xsd_file`
: Path to XSD Schema file.

Return Values

None if successful, otherwise throws an exception as described below.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if a type is not defined in the underlying model.

`SDO_DAS_XML_ParserException`
: Thrown for any problems while parsing the given XSD File.

`SDO_DAS_XML_FileException`
: Thrown if the specified file cannot be found.

SDO_DAS_XML::create

SDO_DAS_XML::create -- To create SDO_DAS_XML object for a given schema file

Description

SDO_DAS_XML **SDO_DAS_XML::create** ([mixed `$xsd_file` [, string `$key`]])

This is the only static method of SDO_DAS_XML class. Used to instantiate SDO_DAS_XML object.

Parameters

`xsd_file`
: Path to XSD Schema file. This is optional. If omitted a DAS will be created that only has the SDO base types defined. Schema files can then be loaded with the **addTypes()** method. Can be string or array of values.

`key`

Return Values

Returns SDO_DAS_XML object on success otherwise throws an exception as described below.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if a type is not defined in the underlying model.

`SDO_DAS_XML_ParserException`
: Thrown for any problems while parsing the given XSD File.

`SDO_DAS_XML_FileException`
: Thrown if the specified file cannot be found.

SDO_DAS_XML::createDataObject

SDO_DAS_XML::createDataObject -- Creates SDO_DataObject for a given namespace URI and type name

Description

SDO_DataObject **SDO_DAS_XML::createDataObject** (string `$namespace_uri`, string `$type_name`)

Creates SDO_DataObject for a given namespace URI and type name. The type should be defined in the underlying model otherwise SDO_TypeNotFoundException will be thrown.

Parameters

`namespace_uri`
: Namespace URI of the type name.

`type_name`
: Type Name.

Return Values

Returns SDO_DataObject on success.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if a type is not defined in the underlying model.

SDO_DAS_XML::createDocument

SDO_DAS_XML::createDocument -- Creates an XML Document object from scratch, without the need to load a document from a file or string.

Description

SDO_DAS_XML_Document **SDO_DAS_XML::createDocument** ([string `$document_element_name`])

SDO_DAS_XML_Document **SDO_DAS_XML::createDocument** (string `$document_element_namespace_URI`, string `$document_element_name` [, SDO_DataObject `$dataobject`])

Creates an XML Document object. This will contain just one empty root element on which none of the properties will have been set. The purpose of this call is to allow an application to create an XML document from scratch without the need to load a document from a file or string. The document that is created will be as if a document had been loaded that contained just a single empty document element with no attributes set or elements within it.

createDocument() may need to be told what the document element is. This will not be necessary in simple cases. When there is no ambiguity then no parameter need be passed to the method. However it is possible to load more than one schema file into the same XML DAS and in this case there may be more than one possible document element defined: furthermore it is even possible that there are two possible document elements that differ only in the namespace. To cope with these cases it is possible to specify either the document element name, or both the document element name and namespace to the method.

Parameters

`document_element_name`
: The name of the document element. Only needed if there is more than one possibility.

`document_element_namespace_URI`
: The namespace part of the document element name. Only needed if there is more than one possible document element with the same name.

`dataobject`

Return Values

Returns an SDO_XML_DAS_Document object on success.

Errors/Exceptions

SDO_UnsupportedOperationException

Thrown if an element name or element name and namespace URI is passed, but not found in the underlying model.

SDO_DAS_XML::loadFile

SDO_DAS_XML::loadFile -- Returns SDO_DAS_XML_Document object for a given path to xml instance document

Description

SDO_XMLDocument **SDO_DAS_XML::loadFile** (string `$xml_file`)

Constructs the tree of SDO_DataObjects from the given address to xml instance document. Returns SDO_DAS_XML_Document Object. Use SDO_DAS_XML_Document::getRootDataObject method to get root data object.

Parameters

`xml_file`
: Path to Instance document. This can be a path to a local file or it can be a URL.

Return Values

Returns SDO_DAS_XML_Document object on Success or throws exception as described.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if a type is not defined by the underlying model.

`SDO_PropertyNotFoundException`
: Thrown if a property within a type is not defined in the underlying model.

`SDO_DAS_XML_ParserException`
: Thrown for any problems while parsing the given XSD File.

`SDO_DAS_XML_FileException`
: Thrown if the specified file cannot be found.

SDO_DAS_XML::loadString

SDO_DAS_XML::loadString -- Returns SDO_DAS_XML_Document for a given xml instance string

Description

SDO_DAS_XML_Document **SDO_DAS_XML::loadString** (string `$xml_string`)

Constructs the tree of SDO_DataObjects from the given xml instance string. Returns SDO_DAS_XML_Document Object. Use SDO_DAS_XML_Document::getRootDataObject method to get root data object.

Parameters

`xml_string`
: xml string.

Return Values

Returns SDO_DAS_XML_Document object on Success or throws exception as described.

Errors/Exceptions

`SDO_TypeNotFoundException`
: Thrown if a type is not defined by the underlying model.

`SDO_PropertyNotFoundException`
: Thrown if the a property within a type is not defined in the underlying model.

`SDO_DAS_XML_ParserException`
: Thrown for any problems while parsing the given XSD File.

SDO_DAS_XML::saveFile

SDO_DAS_XML::saveFile -- Saves the SDO_DAS_XML_Document object to a file

Description

void **SDO_DAS_XML::saveFile** (SDO_XMLDocument `$xdoc`, string `$xml_file` [, int `$indent`])

Saves the SDO_DAS_XML_Document object to a file.

Parameters

`xdoc`
: SDO_DAS_XML_Document object.

`xml_file`
: xml file.

`indent`
: Optional argument to specify that the xml should be formatted. A non-negative integer is the amount to indent each level of the xml. So, the integer 2, for example, will indent the xml so that each contained element is two spaces further to the right than its containing element. The integer 0 will cause the xml to be completely left-aligned. The integer -1 means no formatting - the xml will come out on one long line.

Return Values

None.

Errors/Exceptions

`SDO_DAS_XML_FileException`
: Thrown if the specified file cannot be found.

SDO_DAS_XML::saveString

SDO_DAS_XML::saveString -- Saves the SDO_DAS_XML_Document object to a string

Description

string **SDO_DAS_XML::saveString** (SDO_XMLDocument `$xdoc` [, int `$indent`])

Saves the SDO_DAS_XML_Document object to string.

Parameters

`xdoc`
: SDO_DAS_XML_Document object.

`indent`
: Optional argument to specify that the xml should be formatted. A non-negative integer is the amount to indent each level of the xml. So, the integer 2, for example, will indent the xml so that each contained element is two spaces further to the right than its containing element. The integer 0 will cause the xml to be completely left-aligned. The integer -1 means no formatting - the xml will come out on one long line.

Return Values

xml string.