Adding Intelligence to Media

XMP TOOLKIT SDK

ADDENDUM FOR THE PROGRAMMER'S GUIDE

January 2020

Adobe

Extensible Metadata Platform (XMP) Toolkit SDK, Addendum for the Programmer's Guide.

Contents

Preface

The XMPCore component of the XMP Toolkit SDK provides path-based APIs for manipulating and serializing XMP data, along with support utilities for building particular structures and iterations. This release introduces new DOM based APIs in XMPCore to access metadata tree hierarchy. This document describes these DOM based APIs in XMPCore.

Additions to the Programmer's Guide

This document extends the following sections of the *XMP Toolkit SDK Programmer's Guide* for DOM based APIs.

- Reading XMP Properties
- Modifying XMP data in the XMP object
- Working with Schemas

How this document is organized

This document has the following sections:

- Chapter 1, "Getting started, describes how to build and use the DOM-based APIs.
- Chapter 2, "Reading XMP properties," provides information about the two types of XMP properties - simple and complex.
- Chapter 3, "Modifying XMP data in the XMP object," describes how to use XMPCore to modify XMP properties.
- Chapter 4, "Working with schemas," describes the process of creating or extending existing metadata schemas.

Conventions used in this document

The following type styles are used for specific types of text:

Typeface Style	Used for:
`Monospaced bold`	XMP property names. For example, **`xmp:CreateDate`**
`Monospaced Regular`	XML code and other literal values, such as value types and names in other languages or formats

1 Getting started

This section describes how to build and use the DOM-based APIs.

XMPCore uses **`INCLUDE_CPP_DOM_SOURCE`** flag to include source files for the DOM APIs. By default, this flag is set to true. If you do not want to compile the DOM APIs, set this flag to false in `build/XMP_Config.cmake`.

Client application need to set **`ENABLE_CPP_DOM_MODEL`** to 1 to use the DOM APIs.

2 Reading XMP properties

NOTE: This section provides only a brief overview of the XMP Data Model. Before working with the XMP Toolkit SDK, developers should understand the XMP Data Model, as documented in *XMP Specification Part 1, Data and Serialization Models*.

XMP properties are simple or complex:

- Simple properties have literal values such as strings and booleans
- Array and structures are sets of related values.
  - Arrays are sets of indexed items, with each item holding a value.
  - Structures are sets of named properties (fields), with each field holding a value.

Structures and arrays can contain other structures or arrays, nested to any depth. See the *XMP Specification Part 1* for complete details on data types and properties.

In addition to these basic types, there is special handling for Property qualifiers and language alternatives, and also for dates and times.

Basic property types

In DOM-based model, each property can be viewed as a node. Top most node is called *Metadata node,* which is basically a structure node. Metadata node contains the entire XMP data in the form of tree hierarchy. It contain nodes which can be one of the following types:

- Simple nodes
- Array nodes
- Structure nodes

Simple nodes represent simple properties with values. Similarly, an Array nodes represent array data type and Structure nodes represent structure data type.

To create a new Metadata DOM tree, you can use the `CreateMetadata()` static function of `IMetadata` interface as follows:

```
spIMetaData metaNode = IMetaData::CreateMetadata();
```

This function creates an empty Metadata node, which you can populate according to your needs.

Also you can set the information about the resource, you need to call `SetAboutURI()` function. Provide the function with the AboutURI and AboutURI length, as shown in the following example:

```
metaNode->SetAboutURI("http://ns.adobe.com/exif/1.0/", AdobeXMPCommon::npos );
```

Simple properties

To access value of any simple property, you need to call `GetSimpleNode()` function. Provide the function with the namespace URI, namespace URI length, property name, and property name length. If you know that namespace URI and property name is null terminated, you can pass `AdobeXMPCommon::npos` instead of length.

```
spISimpleNode simpleNode = metaNode->GetSimpleNode(kXMP_NS_XMP, AdobeXMPCommon::npos,
"CreatorTool", AdobeXMPCommon::npos);
```

The `GetSimpleNode()` function returns shared pointer of `ISimpleNode` interface. This returned shared pointer is pointing to specific simple node if that node exists with the specified namespace and property name, otherwise it points to NULL. You should always check the return shared pointer to discover whether you can use the returned contents.

```
AdobeXMPCore::spISimpleNode simpleNode = metaNode->GetSimpleNode(kXMP_NS_XMP,
AdobeXMPCommon::npos, "CreatorTool", AdobeXMPCommon::npos);

if (simpleNode == NULL)

{

    //Error handling code.

}
```

To check the value of the specified property name, you can use `GetValue()` function. It returns the `const` shared pointer of `std::string` type, which contains the property value.

```
AdobeXMPCore::spISimpleNode simpleNode = metaNode->GetSimpleNode(kXMP_NS_XMP,
AdobeXMPCommon::npos, "CreatorTool", AdobeXMPCommon::npos);

if (simpleNode != NULL)

{

    string simpleNodeValue = simpleNode->GetValue()->c_str();

}
```

Array properties

To access array elements, you need to call `GetArrayNode()` function. Provide the function with the namespace URI, namespace URI length, property name, and property name length. If you know that namespace URI and property name is null terminated, you can pass `AdobeXMPCommon::npos` instead of length.

```
spIArrayNode arrayNode = metaNode->GetArrayNode(kXMP_NS_DC, AdobeXMPCommon::npos,
"creator", AdobeXMPCommon::npos);
```

The `GetArrayNode()` function returns shared pointer of `IArrayNode` interface. This returned shared pointer is pointing to specific array node if that node exists with the specified namespace and property name, otherwise it points to NULL. You should always check the return shared pointer to discover whether you can use the returned contents.

```
spIArrayNode arrayNode = metaNode->GetArrayNode(kXMP_NS_DC, AdobeXMPCommon::npos,
"creator", AdobeXMPCommon::npos);

if (arrayNode == NULL)
```

```
{
//Error code.
}
```

Array elements can be one of the three forms: unordered, ordered, and alternative. If you need to know the form of any particular array, you can call `GetArrayForm()`.This function returns enum `eArrayForm`, which determines the form of a particular array.

```
eArrayForm  form = arrayNode-> GetArrayForm();
```

Arrays can contain simple array or structure node. To know about the type of elements of an array, you need to call `GetChildNodeType()` function.

```
eNodeType nodeType = arrayNode-> GetChildNodeType();
```

The function returns value of type enum `eNodeType`, which can be either `kNTNone`, `kNTSimple`, `kNTArray`, or `kNTStructure`.

To retrieve simple node from the array element at a specific index, you need to call `GetSimpleNodeAtIndex(index)` function.

```
spISimpleNode nodeChild = arrayNode->GetSimpleNodeAtIndex(1);
```

Once you get the simple node, you can retrieve the property value using `GetValue()` function as described below.

```
string simpleNodeValue = nodeChild->GetValue()->c_str();
```

To retrieve array node from the array element at the specific index, you need to call `GetArrayNodeAtIndex(index)` function.

```
spIArrayNode nodeChild = arrayNode-> GetArrayNodeAtIndex (1);
```

Similarly, you can obtain structure node from an array element by calling `GetStructureNodeAtIndex(index)` function.

```
spIStructureNode nodeChild = arrayNode-> GetStructureNodeAtIndex (1);
```

NOTE: This is a 1-based index; that is, the index for the first element is 1, not 0.

You can use `ChildCount()` function to discover the number of elements in an array.

```
int numItems= arrayNode->ChildCount();
```

Structure properties

To access structure node elements, you need to call `GetStructureNode()` function. Provide the function with the namespace URI, namespace URI length, property name, and property name length. If you know that namespace URI and property name is null terminated, you can pass `AdobeXMPCommon::npos` instead of length.

```
spIStructureNode structNode = metaNode->GetStructureNode(kXMP_NS_EXIF,
AdobeXMPCommon::npos, "Flash", AdobeXMPCommon::npos);
```

The `GetStructureNode()` function returns shared pointer of `IStructureNode` interface. This returned shared pointer is pointing to specific structure node if that node exists with the specified namespace and property name, otherwise it points to NULL. You should always check the return shared pointer to discover whether you can use the returned contents.

```
spIStructureNode structNode = metaNode->GetStructureNode(kXMP_NS_EXIF,
AdobeXMPCommon::npos, "Flash", AdobeXMPCommon::npos);

if (structNode == NULL)

{

//Error code.

}
```

To know about the type of particular property node in a structure element, you need to call `GetChildNodeType ()` function

```
eNodeType type = structNode-> GetChildNodeType((kXMP_NS_EXIF, AdobeXMPCommon::npos,
"Flash", AdobeXMPCommon::npos);
```

The `GetChildNodeType()` function returns the type of node with the specified namespace URI and property name within the structure node. If no such node exist within structure element, it returns `kNTNone`.

You can also count the number of fields present within structure element. For this, you need to call `ChildCount()` function.

```
int numItems= structNode ->ChildCount();
```

Because arrays and structures can contain nested arrays and structures, you may need a path to access an item or field value below the top level. In the XMP Toolkit SDK, paths are similar to, but not identical to, those defined by the XML path language, XPath. It is highly recommended that you use the provided utility functions to construct complex paths, rather than constructing them manually.

You need to call `GetPath()` function on any node object whose path from the top node is required.

```
spIStructureNode structNode = metaNode->GetStructureNode(kXMP_NS_EXIF,
AdobeXMPCommon::npos, "Flash", AdobeXMPCommon::npos);

if (structNode != NULL)

{

    spIPath pathToNode = structNode->GetPath();

}
```

`GetPath()` function returns shared pointer to `IPath`, which contains the path from top most node to the specified node.

Similarly, you can retrieve the particular node using the `IPath` construct. You need to call the `GetNodeAtPath()` function. Provide the function with `IPath` construct reference as input parameter.

```
spINode pNode = structNode-> GetNodeAtPath(path);
```

Special value handling

The API provides helper functions for dealing with more complex values, including language alternatives and date-times.

Property qualifiers and language alternatives

Properties themselves may have their own properties attached to them. These *properties of properties* are known as property *qualifiers*. Any node may or may not have qualifiers node. Qualifiers node can be of the type simple, array, or structure.To check whether the particular node has any qualifiers or not, you need to call `HasQualifiers()` function on that node construct.

```
bool pHasQualifier = structNode->HasQualifiers ();
```

`HasQualifier()` function returns true if the particular node construct has any of the qualifiers, otherwise it returns false.

To retrieve the qualifier node of the particular node, you need to call `GetQualifier()` function. You need to provide the namespace URI, namespace URI length, property name, and property name length as input parameters. If you do not know the length of the namespace URI and property name, you can pass **`AdobeXMPCommon::npos`** instead of length.

```
spINode pQualNode = structNode->GetQualifier(kXMP_NS_EXIF, AdobeXMPCommon::npos,
"Flash", AdobeXMPCommon::npos);
```

`GetQualifier()` function returns the shared pointer pointing to the qualifier node. If no qualifier node exist with specified namespace URI and property name, `GetQualifier()` function returns an invalid shared pointer. You should always check the function return value to discover whether you can use the returned contents.

```
spINode pQualNode = structNode->GetQualifier(kXMP_NS_EXIF, AdobeXMPCommon::npos,
"Flash", AdobeXMPCommon::npos);


if(pQualNode == NULL)
{
    //Error handling code
}
```

Also, you can use specialized functions `GetSimpleQualifier()`, `GetStructureQualifier()` or `GetArrayQualifier()` if you already know the type of the qualifier node.

To know the type of qualifier node, you need to call `GetQualifierNodeType()` function. You need to provide namespace URI, namespace URI length, property name, and property name length as input parameters. If you do not know the length of namespace URI and property name, you can pass **`AdobeXMPCommon::npos`** instead of length.

```
eNodeType type =  structNode->GetQualifierNodeType (kXMP_NS_EXIF,
AdobeXMPCommon::npos, "Flash", AdobeXMPCommon::npos);
```

If any qualifier node with specified namespace and property name exists, `GetQualifierNodeType()` function returns the type of node i.e., `kNTSimple`, `kNTArray`, `kNTStructure`, otherwise it returns `kNTNone`.

A node's type can also be retrieved by calling `GetNodeType()` method.

```
spINode pQualNode = structNode->GetQualifier(kXMP_NS_EXIF, AdobeXMPCommon::npos,
"Flash", AdobeXMPCommon::npos);

eNodeType type  = pQualNode->GetNodeType();
```

Also, if you need to count the number of qualifiers attached to a particular node, you need to call `QualifiersCount()` function.

```
int pQualCount = structNode-> QualifiersCount();
```

If no qualifier is attached to the calling node, `QualifierCount()` function returns 0.

Similarly, if you need to know whether the specified node is a qualifier node or not, you need to call `IsQualifierNode()` function. It returns true if the calling node is a qualifier node, otherwise it returns false.

```
bool value =  structNode->IsQualifierNode();
```

3 Modifying XMP data in the XMP object

This section discusses how to use XMPCore to modify XMP properties. These techniques are illustrated using the samples provided with the SDK.

NOTE ON HANDLING NEWLINES IN USER INTERFACE: The way a user interface handles newlines in text values is important to the global and cross-platform portability of XMP. When displaying text, applications should recognize common newline characters and sequences and ensure that they display as such. One technique is to modify the displayed text, substituting appropriate local newlines. You must take care, however, that the stored XMP value is not modified simply as a result of display.

Typical newlines are a single ASCII linefeed (LF, U+000A), a single ASCII carriage return (CR, U+000D), or ASCII CR-LF. Section 2.11 of the XML 1.0 specification includes other sequences as recognized newlines for normalization purposes: U+0085, U+2028, and the pair U+000D U+0085.

It is recommended that applications store all newlines in XMP text values as ASCII linefeed.

Creating and modifying simple properties

The simplest way to create a new property is with `CreateSimpleNode()` function. It creates a simple node which is not a part of any metadata. You need to provide namespace URI, namespace URI length, property name, and property name length as input parameters with optional value field and value length field. If you do not know the length of the namespace URI and property name, you can pass **`AdobeXMPCommon::npos`** instead of length.

```
spISimpleNode creatorChild1 =ISimpleNode::CreateSimpleNode(kXMP_NS_DC,
AdobeXMPCommon::npos, "AuthorName", AdobeXMPCommon::npos, "abc",
AdobeXMPCommon::npos);
```

`CreateSimpleNode()` function returns shared pointer to `ISimpleNode` interface. In case namespace URI or property values have invalid contents or property name is not a valid XML name, error will be raised.

You can set or modify the value of already created property with `SetValue()` function. You need to provide valid value and length of value as input parameters. In case of unknown length of value field, you can pass **`AdobeXMPCommon::npos`**.

```
creatorChild1-> SetValue("Updated By XMP SDK", AdobeXMPCommon::npos);
```

Also, if a simple property is of URI type, you need to call `SetURIType()` function with boolean true as input parameter.

```
creatorChild1-> SetURIType(true);
```

Similarly, you can query whether the particular simple property is URI Type or not. You need to call `IsURIType()` function, it returns true in case the property is of URI Type, otherwise it returns false.

Creating and modifying arrays

Array nodes can also be created in similar fashion like simple properties. You need to call any of the following functions depending on the required array form:

CreateUnorderedArrayNode(), CreateOrderedArrayNode(), or CreateAlternativeArrayNode().

```
spIArrayNode arrayNode = IArrayNode::CreateUnorderedArrayNode(kXMP_NS_DC,
AdobeXMPCommon::npos, "creator", AdobeXMPCommon::npos);
```

This function call creates an unordered array node, which is not part of any metadata. It returns invalid shared pointer in case of invalid XML name or namespace URI contains NULL data.

In order to insert a node in this newly created array node, you need to call `InsertNodeAtIndex()` function. It takes the node to be inserted into the metadata node and the index where the node is to be inserted. Array indexes are 1-based; that is, the index for the first element is 1, not 0.

```
arrayNode-> InsertNodeAtIndex(pNode, 1);
```

`InsertNodeAtIndex()` function throws an error if the inserting node is not valid, type of the node is not same as other child of the array node, node is already a child of any other node, or the index value is not correct i.e., index is less than 1 or greater than current child count + 1.

Creating and modifying structures

Similar to simple nodes and array nodes, structure nodes can also be created with `CreateStructureNode()`.

```
spIStructureNode structNode = IStructureNode:: CreateStructureNode (kXMP_NS_DC,
AdobeXMPCommon::npos, "creator", AdobeXMPCommon::npos);
```

This function creates a structure node, which is not part of any metadata. This function returns invalid shared pointer if namespace URI or property name is NULL or property name is not a valid XML name.

To insert any node in structure node, you need to call `InsertNode()` function. It requires the valid node as input parameter, which needs to be inserted in the structure node.

```
structNode-> InsertNode(anyValidNode);
```

If the node is already a part of the structure node or child of any other node in the metadata, an exception is raised.

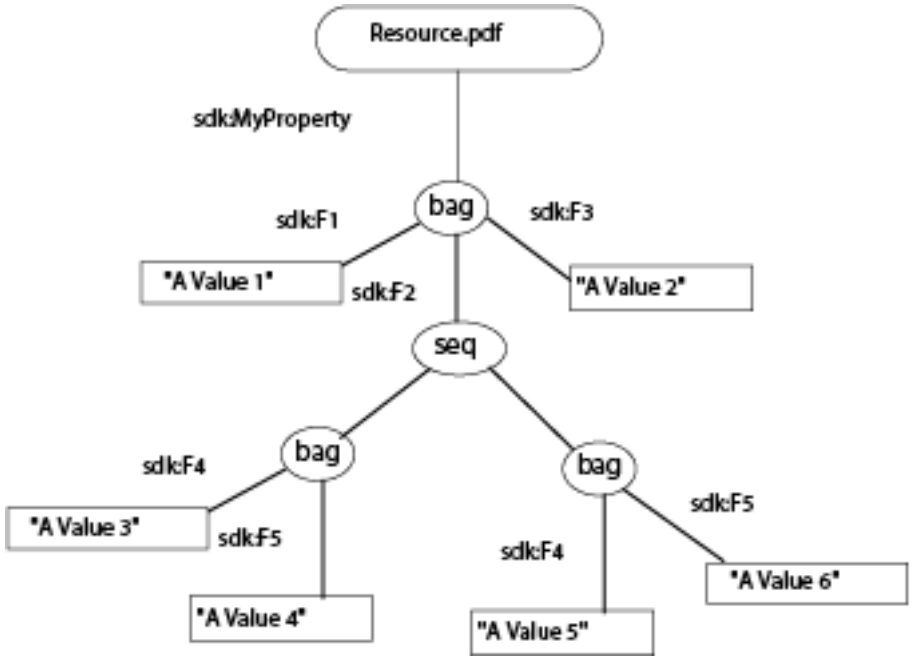
Modifying and creating complex properties

Schemas can contain very complex properties, such as arrays nested within structures. For instance, a complex property can be:

- A structure with nested arrays.
- A structure with nested structures.
- An array with nested structures.
- An array with nested arrays.

Each property can have an arbitrary number of nested levels. For example, a structure can have arrays as its field values; the array items themselves could also be arrays or structures, and so on.

This figure shows a conceptual diagram of a complex property:



In this example, `Resource.pdf` has a single property named `MyProperty`. The property type is a structure which has several fields, one of which is an ordered array. The array holds items which themselves are structures.

The same XMP serialized to RDF looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

   <rdf:Description xmlns:sdk="http://ns.adobe.com/xmp/sdk/" rdf:about="">

      <sdk:MyProperty rdf:parseType="Resource">

         <sdk:F1>A Value1</sdk:F1>

         <sdk:F2>

            <rdf:Seq>

               <rdf:li rdf:parseType="Resource">

                  <sdk:F4>A Value3</sdk:F4>

                  <sdk:F5>A Value4</sdk:F5>

               </rdf:li>

               <rdf:li rdf:parseType="Resource">

                  <sdk:F4>A Value5</sdk:F4>

                  <sdk:F5>A Value6</sdk:F5>
```

```
            </rdf:li>
          </rdf:Seq>
        </sdk:F2>
        <sdk:F3>A Value2</sdk:F3>
      </sdk:MyProperty>
   </rdf:Description>
</rdf:RDF>
```

In order to access deeply nested properties, such as **sdk:F5**, the name alone does not point to the correct property; you must provide a path. To retrieve the path for **sdk:F5**, you need to call `GetPath()` function on the node object holding property **sdk:F5**.

```
spIPath path = node->GetPath();
```

If you need to serialize the path, which is returned by `GetPath()` function, you need to call `Serialize()` function. You can provide the reference of `INameSpacePrefixMap` for mapping between namespace and prefixes. If you do not want to provide the mapping, default mapping will be used, but the preference will always be given to user given mapping.

```
spIUTF8String serializedPath-> Serialize();
```

For example, if you serialize the path for property **sdk:F5**, it will be serialized in the following manner.

```
Sdk:MyProperty/sdk:F2[2]/sdk:F5
```

You should not try to compose a complex path manually. The XMP API provides utility functions, which should be used to compose paths to deeply nested properties.

You can create the complex node and modify the current value of any node using the functions provided by XMPCore.

For example, to add a value to **sdk:F4** in the complex property structure shown in the diagram above. You first retrieve the node which is holding the property **sdk:F4**.

Considering you have the metadata node, which is the top most node. Perform the following steps to retrieve the node holding the property **sdk:F4** (shown in the preceding diagram):

1. Retrieve the structure node, which contains `MyProperty` structure. To do so, call:

   ```
   spIStructureNode MyPropertyNode =
   metaNode->GetStructureNode("http://ns.adobe.com/xmp/sdk/", AdobeXMPCommon::npos, "
   MyProperty ", AdobeXMPCommon::npos);
   ```

   This structure contains 3 fields, namely F1,F2, and F3. F2 is an ordered array.

2. Retrieve the array node from the structure node `MyPropertyNode`, which is retrieved in Step 1.

   ```
   spIArrayNode arrayNode = MyPropertyNode
   ->GetArrayNode("http://ns.adobe.com/xmp/sdk/", AdobeXMPCommon::npos, "F2",
   AdobeXMPCommon::npos);
   ```

3. Now this array node contains two structures and property `sdk:F4` is presented in the structure which is indexed at position 1. To retrieve the structure node presented at the 1st index of the array node, call:

   ```
   spIStructureNode structureNodeChild = arrayNode-> GetStructureNodeAtIndex (1);
   ```

4. To retrieve simple node named `sdk:F4` from the structure node, call:

```
spISimpleNode simpleNode = structureNodeChild
->GetSimpleNode("http://ns.adobe.com/xmp/sdk/", AdobeXMPCommon::npos, "F4",
AdobeXMPCommon::npos);
```

5. Now you can modify or set the property value of `sdk:F4`. To do so, call:

```
simpleNode ->SetValue("A modified value", AdobeXMPCommon::npos);
```

6. You can use the `GetPath()` function to get the path to this node and use that path to directly access this node in future. To retrieve the node using the path, you need to call `GetNodeAtPath()` function. This function takes the `IPath` construct as an input parameter and returns the node at the specified path.

Similarly, you can create any node and insert it into any other complex node like arrays and structures. Suppose you want to create and insert a new node in the property `MyProperty`.

To do so, perform the following steps:

1. Create a simple node using `CreateSimpleNode()` function. It is a static function so you do not require any construct of `ISimpleNode` interface.

```
spISimpleNode node = ISimpleNode::CreateSimpleNode("http://ns.adobe.com/xmp/sdk/",
AdobeXMPCommon::npos, "F0", AdobeXMPCommon::npos, "A Value 0",
AdobeXMPCommon::npos);
```

   This function creates a new simple property and sets its value to "A Value 0".

2. To insert this newly created property to the `MyProperty` property, call `InsertNode()` function:

```
MyPropertyNode->InsertNode(node);
```

   Also, you can replace any existing node with the new node using `ReplaceNode()` function or remove any node using `RemoveNode()` function.

```
spINode node = MyPropertyNode->RemoveNode("http://ns.adobe.com/xmp/sdk/"
AdobeXMPCommon::npos, "F0", AdobeXMPCommon::npos);
```

   It removes the existing node with specified namespace and property name. If no such node exist, this function throws an exception.

Similarly, you can insert, remove, or replace any node in an array using `InsertNodeAtIndex()`, `RemoveNodeAtIndex()`, or `ReplaceNodeAtIndex()` function. These functions take the node to be inserted and the index at which it needs to be inserted.

Modifying qualifiers in complex properties

To add a qualifier to a property, you must provide qualifier node and property node to which it is attached. The following example shows how to add a qualifier for `sdk:F4` (shown in the preceding diagram):

```
spISimpleNode qualifierNode =
ISimpleNode::CreateSimpleNode("http://ns.adobe.com/xmp/sdk/", AdobeXMPCommon::npos,
"Qualifier Node", AdobeXMPCommon::npos, "Qualifier", AdobeXMPCommon::npos);
```

```
spIStructureNode MyPropertyNode =
metaNode->GetStructureNode("http://ns.adobe.com/xmp/sdk/", AdobeXMPCommon::npos, "
MyProperty ", AdobeXMPCommon::npos);


spIArrayNode arrayNode = MyPropertyNode ->GetArrayNode("http://ns.adobe.com/xmp/sdk/",
AdobeXMPCommon::npos, "F2", AdobeXMPCommon::npos);


spIStructureNode structureNodeChild = arrayNode-> GetStructureNodeAtIndex (1);


spISimpleNode simpleNode = structureNodeChild
->GetSimpleNode("http://ns.adobe.com/xmp/sdk/", AdobeXMPCommon::npos, "F4",
AdobeXMPCommon::npos);


simpleNode->InsertQualifier (qualifierNode);
```

`InsertQualifier()` function inserts a qualifier node in the leaf property. If the qualifier node is already a child of any property then this function throws an exception. You can remove or replace the qualifier from the specified property using `RemoveQualifier()` and `ReplaceQualifier()` functions.

4 Working with schemas

There are a range of metadata schemas available for you to take advantage of, as described in the *XMP Specification Part 2, Standard Schemas*. If you need, you can either extend an existing schema or create a new one.

Although you can add new properties to extend an existing schema, the standard schemas are generally intended for specific uses and should not be altered. Technically you can add a property to, say, the Dublin Core schema, however, it is not recommended. If you need specific set of properties, you should create a new schema. For example, you might create a schema `com.companyName.xmp/1.0`, and properties within that schema for `itemCode`, `orderNumber`, and so on.

Creating custom schemas

In order to avoid collisions with properties in other schemas, a schema must have a unique name. The schema name is in the form of a URI and must comply with the XML 1.0 namespace rules, as defined at http://www.w3.org/TR/2006/REC-xml-names-20060816/. You can also define a preferred prefix to use with your namespace; defining a prefix is not mandatory, but it is recommended.

You will add properties to your namespace as you would to an existing schema. For your own schema, you should also create a *specification document* that lists and describes all of the properties and data types. The specification document is plain text, human readable, and does not need to be in any specialized format. You should make it available to anyone wishing to work with your custom schema; however, it is not necessary to publish it in the public domain.

Registering namespaces

To use your custom schema, you must register the namespace and prefix that you have chosen. There are following ways to do this:

- Create an instance of `INameSpacePrefixMap` using static function **`NameSpacePrefixMap::CreateNameSpacePrefixMap().`**

  ```
  spINameSpacePrefixMap myCustomMap =
  NameSpacePrefixMap::CreateNameSpacePrefixMap();
  ```

  Now, to register a namespace explicitly, you need to call `Insert()` function. It requires a prefix, length of buffer holding prefix, namespace, and length of namespace buffer. It returns true in case of successful registration on the given namespace and prefix, otherwise it returns false.

  ```
  bool retValue = myCustomMap->Insert("MyPrefix", AdobeXMPCommon::npos
  ,"http://ns.adobe.com/MyNamespace/", AdobeXMPCommon::npos);
  ```

  You can pass this custom namespace prefix map to `IPath::Serialize()` function.

- Create a new XMP object from valid RDF. Unknown namespaces are registered automatically.
- Register the namespace explicitly using the static function `SXMPMeta::RegisterNameSpace()`.

To register a namespace explicitly, you provide the namespace URI and a preferred prefix:

```
string actualPrefix SXMPMeta::RegisterNamespace(
"http://ns.adobe.com/MyNamespace/","MyPrefix", &actualPrefix );
```

The prefix you pass is a suggestion; it is not guaranteed to be registered as the prefix for the namespace.

The last argument, `actualPrefix`, returns the actual prefix that will be used for the registered namespace.

- If you register a new namespace with a new prefix (that is, one not yet in use), the namespace is registered with your suggested prefix, and the function returns it in the `actualPrefix` buffer.
- If the prefix you supply for a new namespace is already in use, the function generates a new, unique prefix based on the one you supplied. For example, if there is already a prefix named `MyPrefix`, then the actual prefix returned is `MyPrefix_1_`; or, if `MyPrefix_1_` is already used, `MyPrefix_2_`, and so on.
- If you register an existing namespace with a new prefix, the function returns the prefix that is already registered for that namespace.

You can retrieve the default register namespace map by calling static function `INameSpacePrefixMap::GetDefaultNameSpacePrefixMap()`; This function returns the shared pointer pointing to `INameSpacePrefixMap` instance, which contains mapping of standard namespace and their prefixes. Also, it contains mapping of namespace and prefixes registered by calling `RegisterNamespace()` function above.

Prefix collisions can occasionally occur, both at run time and when XMP is serialized and stored. You should never depend on the suggested prefix being used, but always check for and use the actual returned prefix when registering a namespace.

Extending schemas

There are no restrictions as to what properties you can add to a namespace. However, it is recommended that you try to keep properties consistent with those already existing in a schema.

- New properties that you add to a schema do not interfere with existing applications, as they have no knowledge of your extensions. This also means, of course, that they cannot take advantage of them. Your extensions are of use only to your own applications.
- You should not change existing property definitions from other schemas. This may cause existing applications to perform unexpectedly and produce incorrect results.

To add a new property to a schema you need to only create that property and set a value, even if the value is empty. For example, if you wish to add a new text property to a schema you have created, the following is sufficient:

```
spISimpleNode node = ISimpleNode::CreateSimpleNode("http://ns.adobe.com/xmp/sdk/",
AdobeXMPCommon::npos, "F0", AdobeXMPCommon::npos, "A Value 0", AdobeXMPCommon::npos);

MyPropertyNode->InsertNode(node);
```

Any property type can be used when extending a schema and there are is no limitation to the number of properties that can be added. When the XMP object is written back to the resource, the new property is also written.

As long as you have provided a unique namespace URI for your schema, new properties that you define are guaranteed to not interfere with existing schemas or their properties, even if your new properties have the same local name as a property defined elsewhere.

You can also define new property value types. For example, your properties can define structures and arrays. The specification document for your schema should document all schema properties and any new property types.

DOM Implementation Registry

To use, parse, and serialize functions of DOM based APIs in XMPCore, client should call the following function to obtain `IDOMImplementationRegistry`.

```
spIDOMImplementationRegistry registry =
DOMImplementationRegistry::GetDOMImplementationRegistry();
```

This function returns a shared pointer of `IDOMImplementationRegistry` interface. Using `IDOMImplementationRegistry` interface pointer, client can set or get their own serializer and parser. Also, client can obtain the default RDF parser and serializer by using the following function call:

```
spIDOMParser parser = registry->GetParser( "rdf" );
```

The above function gets the default RDF parser of the new XMPCore.

```
spIDOMSerializer serializer = registry->GetSerializer("rdf");
```

The above function gets the default RDF serializer of the new XMPCore.

Also, client can register their own parser using the following function:

```
bool APICALL RegisterParser( const char * key, pIClientDOMParser_base parser );
```

`RegisterParser` function takes two input parameters:

1. `key` contains the name of the parser the clients want to give to their parser.
2. `pIClientDOMParser_base parser` contains the pointer of the class which implements `IClientDOMParser` interface. Clients have to implement

```
virtual spINode APICALL Parse( const char * buffer, sizet bufferLength, pcIConfigurable
configurationParameters, ReportErrorAndContinueFunctor proc );
```

so that the provided buffer can be parsed according to the clients' requirements.

`Parse` function takes four input parameters:

1. `const char* buffer`: pointer to the buffer which needs to be parse.
2. `sizet bufferLength`: length of the buffer which is pointed by the `buffer`.
3. `pcIConfigurable configurationParameters`: pointer of `IConfigurable` interface.
4. `ReportErrorAndContinueFunctor proc`: A **`Functor`** for reporting errors back to the library while performing the parsing operation.

For Example:

```
class customParser : public IClientDOMParser{

public:

virtual spINode APICALL Parse( const char * buffer, sizet bufferLength, pcIConfigurable
configurationParameters, ReportErrorAndContinueFunctor proc );
```

```
};

spINode customParser:: Parse( const char * buffer, sizet bufferLength, pcIConfigurable
configurationParameters, ReportErrorAndContinueFunctor proc ){

//Some Parsing functionality here

}

To register customParser to DOMImplementationRegistry:

spIDOMImplementationRegistry registry =
DOMImplementationRegistry::GetDOMImplementationRegistry();

registry-> RegisterParser("CustomParser", new customParser());
```

To get the `customParser` from `DOMImplementationRegistry`:

```
spIDOMParser parser = registry->GetParser( " CustomParser " );
```

Now to parse the buffer using client customized parser, call the following function with appropriate arguments as shown above:

```
spIMetadata  meta = parser ->Parse(…);
```

In a similar way, client can register its own custom serializer and use it to serialize the XMP Meta Object.

To Register the serializer:

- Get the `IDOMImplementationRegistry` using the following function:

  ```
  spIDOMImplementationRegistry registry =
  DOMImplementationRegistry::GetDOMImplementationRegistry();
  ```

- Call the `RegisterSerializer` function, using the obtained `IDOMImplementationRegistry` pointer

  ```
  registry ->RegisterSerializer( const char * key, const spcIDOMSerializer &
  serializer );
  ```

  `RegisterSerializer` function takes two input parameters:

  - `const char * key` contains the name of the serializer, which clients want to give to their serializer.
  - `const spcIDOMSerializer & serializer` contains the pointer of the class which implements `IClientDOMSerializer` interface. Clients have to implement

```
virtual void APICALL Serialize( const spINode & node, const spcINameSpacePrefixMap &
nameSpacePrefixMap, pcIConfigurable configurationParameters,
ReportErrorAndContinueFunctor functor, const spIUTF8String & string );
```

so that the provided node can be serialized according to the clients' requirements.

Serialize function takes five input parameters:

- `const spINode& node`: node to be serialized.
- `const spcINameSpacePrefixMap & nameSpacePrefixMap`: An object of type `INameSpacePrefixMap`, which contains preferred prefixes for namespaces.

- `pcIConfigurable configurationParameters`: An object of type **`AdobeXMPCommon::IConfigurable`** containing all configuration parameters requested by the client to be handled while serializing.
- `ReportErrorAndContinueFunctor proc`: A function object to be used by the serializing operation to report back any encountered errors/warnings.
- `const spIUTF8String & string`: A shared pointer to an `IUTF8String` object which should be filled with the serialized form of XMP Data Model.

For Example:

```
class customSerializer : public IClientDOMSerializer{

public:

virtual void APICALL Serialize( const spINode & node, const spcINameSpacePrefixMap &
nameSpacePrefixMap, pcIConfigurable configurationParameters,
ReportErrorAndContinueFunctor functor, const spIUTF8String & string );

};

void customSerializer::Serialize( const spINode & node, const spcINameSpacePrefixMap &
nameSpacePrefixMap, pcIConfigurable configurationParameters,
ReportErrorAndContinueFunctor functor, const spIUTF8String & string ) {

//Some Serialization functionality here

}
```

To register `customSerializer` to `DOMImplementationRegistry`:

```
spIDOMImplementationRegistry registry =
DOMImplementationRegistry::GetDOMImplementationRegistry();

registry-> RegisterSerializer("CustomSerializer", new customSerializer());
```

To get the `customSerializer` from `DOMImplementationRegistry`:

```
spIDOMSerializer serializer = registry->GetSerializer( " CustomSerializer " );
```

Now to serialize the node using client customized serializer, call the following function with appropriate arguments as shown above:

```
serializer ->Serialize(...);
```